

Final Year Project Report

Full Unit – Final Report

SELF-DRIVING F1TENTH VEHICLE USING PLANNING & LOCALISATION ALGORITHMS WITHIN ROS – Final Report

Kacper Buksa

A report submitted in part fulfilment of the degree of

**BSc (Hons) in Computer Science (Software
Engineering)**

Supervisor: Francisco Ferreira Ruiz

Final Word count: 15,938 Words



Department of Computer Science
Royal Holloway, University of London

Table of Contents

Chapter 1:	Introduction	5
Chapter 2:	Aims.....	6
2.1	Aims Explained in Detail.....	6
2.2	Software Engineering.....	7
2.2.1	Object-Oriented Programming	7
2.2.2	Testing.....	8
Chapter 3:	Background Overview on Robotics and F1Tenth.....	9
3.1	F1Tenth Simulator & Algorithms	9
3.2	Robotics Operating System (ROS).....	10
3.3	F1Tenth Hardware	10
3.4	ROS Topics.....	11
3.4.1	LaserScan Topic.....	11
3.4.2	AckermannDriveStamped Topic	11
3.4.3	Odometry Topic	11
Chapter 4:	F1Tenth car hardware, sensors, and communication.....	12
4.1	Introduction	12
4.2	F1Tenth Hardware	12
4.2.1	Traxxas Slash 4x4 Premium Chassis	12
4.2.2	High-capacity GTR Shocks (suspension).....	13
4.2.3	Traxxas Velineon Power System	13
4.2.4	LiPo batteries & Charger	13
4.2.5	VESC 6 Mk VI	13
4.2.6	NVIDIA Jetson Xavier NX	14
4.3	Ranging Sensors – Lidar & radar	14
4.3.1	Hokuyo UST-10LX Scanning Laser Rangefinder	14
4.4	Hardware Communication with ROS	15
4.5	F1Tenth Car Comparisons to Simulator	16
Chapter 5:	Installation and Configuration of ROS and the F1Tenth Simulator.....	17
5.1	Installation of Ubuntu Operating System	17
5.1.1	Installing Oracle VM VirtualBox	17
5.1.2	Ubuntu Installation – Version 18.04 LTS (Bionic Beaver).....	17
5.2	Installation & Configuration of ROS Environment	17

5.3	Installation of F1Tenth Simulator	18
5.4	Running Programs with ROS and F1Tenth launcher	18
Chapter 6:	Path Planning and Localization Algorithms.....	20
6.1	Bug Algorithm (Path Planning).....	20
6.1.1	Bug-0	20
6.1.2	Bug-1	20
6.1.3	Bug-2	21
6.1.4	Limitations of Bug Algorithm	22
6.2	SLAM	23
6.2.1	Implementation within ROS.....	23
6.3	Case study SLAM/ Path Planning: Search & Rescue using a 'Robot Team'	24
Chapter 7:	Important Programs in this Project.....	26
7.1	Message Passing	26
7.1.1	Nodes & Topics	26
7.1.2	Publisher	26
7.1.3	Subscriber	26
7.1.4	LidarScan message reading.....	27
7.2	ROS Basic movement control.....	28
7.2.1	Auto-drive	28
7.2.2	Emergency Brake System	28
7.3	Car's Localization	29
7.4	Wall Following.....	30
7.4.1	Mathematical Approach	30
7.4.2	Logical Approach.....	30
7.5	Obstacle Avoidance.....	31
7.6	Testing Results	31
7.6.1	Emergency brake	31
7.6.2	Wall follower	32
7.6.3	Obstacle avoidance	32
Chapter 8:	Professional Issues in the Real-World.....	34
Chapter 9:	Self-assessment.....	36
9.1	Achievements.....	36
9.2	Skills and Knowledge Gained	36
9.3	Areas of Improvement	37
Chapter 10:	Conclusion.....	38
10.1	Future work.....	38

Chapter 11: References & Citations.....	39
Chapter 12: Appendix	41
12.1 Demonstration of Programs Video Link.....	41
12.2 Table of Figures.....	41
12.3 Code	42
12.3.1 Turtle_auto.py.....	42
12.3.2 Publisher.py.....	43
12.3.3 Subscriber.py.....	43
12.3.4 Lidar_scan.py	44
12.3.5 Auto_drive_test.py	44
12.3.6 Emergency_brake.py	45
12.3.7 Emergency_brake_test.py	46
12.3.8 Localization.py	47
12.3.9 Wall_follow_pid.py	48
12.3.10 Wall_follow_state.py	51
12.3.11 Wall_follow_test.py	53
12.3.12 Obstacle_avoid.py.....	54
12.3.13 Obstacle_avoid_test.py	57
12.4 Diary.....	59

Chapter 1: Introduction

In this project, I aim to create a driverless vehicle that can navigate through virtual maps with no human assistance. To achieve this, the project utilises the Robotics Operating System (ROS) in conjunction with the F1Tenth library, which offers a customizable template for the robot that can be adapted to perform various tasks.

The project begins with an overview of robotics in autonomous vehicles, explaining how the F1Tenth hardware and software can be utilized to achieve this goal. This is followed by an exploration of various algorithms that are commonly used in autonomous vehicle technology.

The implementation of the F1Tenth simulator is then demonstrated through custom programs, which showcases the fundamental aspects of how an autonomous vehicle operates in its environment.

This project is particularly fascinating because it allows for the exploration and monitoring of the progress of autonomous driving technology, which is still in the developmental stages worldwide. Artificial Intelligence technology has made significant strides over the past decade, with AI becoming increasingly intelligent every day. Self-driving vehicles would offer numerous benefits, particularly in terms of reducing road accidents. Since over 90% of accidents caused by human error [1], by taking out the driver, this would drastically reduce the number of accidents on the road, as 'human factors' such as fatigue or late reactions do not affect machines, which will now be in control of the vehicle. Therefore, I am eager to contribute to this growing field by developing my autonomous vehicle that can handle various scenarios that drivers may encounter.

Although physical access to the car was not possible during the project, the F1Tenth virtual simulator was used to communicate between different components of the virtual car and simulate driving through different scenarios. The car's laser scan feature was utilized to enable the car to employ various driving and steering techniques, allowing it to handle different scenarios. Using mathematical and logical approaches, I was able to produce to a certain extent accurate and precise reactions of the car in the environments that it is tested on.

Chapter 2: Aims

This project consists of multiple aims, which is separated into four sections:

1. Installation, configuration, and explanation of software & environments.
 - a. Early Deliverable 1 – “Familiarity with Robot Operating System (ROS) environment and F1tenth simulator. Demonstrate manual control of the simulated car”.
 - b. Early deliverable 3 – “Report on installation and configuration of ROS and the F1tenth simulator”
2. Discussion on the physical car’s hardware and how it links to the virtual simulator.
 - a. Early Deliverable 2 – “Report on F1tenth car hardware, sensors and communication.”
3. Report on the motivation for this project, discussing the background, theories, and explanation of the implementations.
 - a. Final Deliverable 3 – “Final report, covering motivation and background around autonomous vehicles and F1tenth, description of implemented algorithms, and experimental results.”
4. Appropriate algorithm and program implementation to show the F1Tenth in action.
 - a. Early Deliverable 4 – “Implementation and evaluation of basic algorithms (PID control, wall-following, obstacle avoidance).”
 - b. Final Deliverable 1 – “Implementation and evaluation of at least one localization algorithm of choice.”
 - c. Final Deliverable 2 – “Implementation and evaluation of at least one planning algorithm of choice.”

2.1 Aims Explained in Detail

With the deliverables listed, I have created a plan on objectives that I accomplish throughout the project to achieve all the listed deliverables:

1. To set up F1Tenth simulator, by following the ROS download documentation, then the F1Tenth specific environment. Afterwards, setting up custom Catkin packages to store and run ROS programs on the F1Tenth simulator launcher in ‘Rviz’.
2. To create a simple demonstration showing how user assistance is used to drive the car, via keyboard commands.
3. To create a report on installation and configuration of ROS and the simulator. How the installation was done, configuring environments to run and work on ROS. This is closely related with Objective 1.
4. To study ROS documentation, learn how to use it, follow F1Tenth lecture tutorials, use F1Tenth lab worksheets as inspiration for the creation of programs to show different functions of the robot. Links to Objectives 6-8.

5. To research and report on the hardware of F1Tenth, its sensors and other components. Explain how simulator programming helps in working with actual car, with transition from virtual execution to live execution.
6. To show how ROS works with message writing, publishing messages and listening to such messages. Show subscriber-publisher system in ROS.
7. To create a program that shows car's use of laser scan sensors, how it detects boundaries.
 - a. Show the use of wall detection by creating an emergency brake system when too close to wall. This shows how the subscriber-publisher systems works with the car and test track, by subscribing to laser scan and publishing the car stop.
8. Research and implement self-driving features in ROS F1Tenth and report on the self-driving components:
 - a. Show how car can drive automatically by finding a wall and following it without collisions.
 - b. Show how a car can manoeuvre through a track avoiding obstacles, without any collisions.
9. Implement and evaluate path planning algorithm in ROS using F1Tenth components, allowing car to identify target location and use an algorithm to safely reach goal destination.
10. Implement automated tests for all the programs written to show that the code written can perform as expected in different expected scenarios.

2.2 Software Engineering

Alongside making sure that dead code is barely present, commenting lines of code, calling variables appropriate names and other software engineering practices, the sections below are important and require more explanation.

2.2.1 Object-Oriented Programming

Using OOP to create programs in this project has proven to be very useful in many ways. By separating each program into their separate classes, it allows code to be reused either through inheritance, or by using the class itself through creation of a new object. As a result, instead of re-writing already existing code, I can just make use of my program and further show how code in my project can connect to create a fully functional automated car with more time.

The class system offers modularity for an easier method to run tests and troubleshooting of programs. With the objects being encapsulated, when a bug is found it can easily be traced back to the class instead of having to read all the code line-by-line trying to find the root of the problem.

OOP also helps with eliminating issues with use of global variables, such running out of memory since programs hold these variables until execution is completed. Instead, these variables can be contained within the class and called, when necessary, inside the object's methods. Since ROS relies on initialising

subscribers, publishers, their topics and messages, or other variables, having them initialised inside the class allows them to be reused throughout the code while also taking up far less memory.

2.2.2 Testing

Testing is an important part of this project help reach my set aims. This allows me to automate checks for if my written programs successfully implement all desired features. With three of my aims being algorithm-heavy, these tests help me see if the programs are successful and are adequate to complete my aims. Alongside that, in my 3rd final deliverable, tests help show the experimental results of my programs, showing that the results I expect written in tests pass when running on the program.

Since the project is worked on using Python, I make use of the 'unittest' library that offers extensive tests for all Python programs. When working on this project, I have discovered that to accomplish such tests, I would need to optimise my programs to all be in classes. As a result, when writing unit tests, I can set up the necessary resources or pre-conditions for running these tests. This can be achieved by creating an instance of the class to be tested, using the code written to create simulations of different scenarios. ROS also includes its own 'rotest' library to create a ROS environment to test such code which acts as a master node to simulate the tests. By using these libraries, making automated quick tests help identify that the results I want are what is expected of the code written.

When it comes to testing programs written for ROS, manual testing by running the code in Rviz visualiser is inevitably be used alongside automated unit tests, with the program automating the car's movement on the simulator. Since unit tests and ROS tests do not show a visual execution of the code in a simulator, using this technique allows me to clearly see how each program runs in real-time. The simulator run helps show if the car's movement are at the right speed, if the turns it makes are well done and if any potential optimisation is needed.

To conclude the topic of tests, unit tests help identify if the code written works in set scenarios. Meanwhile, running the programs on the visualiser helps show the program in execution, showing in greater detail how it runs, allowing me to identify any weaknesses or places to improve the program.

Chapter 3: Background Overview on Robotics and F1Tenth

In this project, I use the F1Tenth simulator to create a program that allows a driverless car to navigate through a virtual map without any human assistance. This allows me to simulate a scenario that could then further be developed for use in fast time trials and racing, since this is what the F1Tenth car was created for. Alongside autonomous driving, I also implement track learning, where the car must navigate through an unknown terrain, like rally car racing, where the drivers tend to learn the track on the go. The simulator can provide all the necessary frameworks and tools to accomplish this project.

3.1 F1Tenth Simulator & Algorithms

The F1Tenth Simulator is the selected way of testing such ideas. This software allows the testing of wall-following capabilities by programming the car to sense walls and drive within their boundaries. By doing so, the car does not need human interaction to drive it, since it can create its own path using the environment it is in. With the use of odometry and a camera, the car can view what is in front of it, being able to track the current environment it is at. Once it tracks the current path, it can store that path and overtime, once it ends driving, the car would be able to recreate the whole path in its storage, even if it has never seen it beforehand [2].

ROS F1/10 Simulator technology has a variety of frameworks that give the capabilities of creating such system that the car can drive automatically and learn the track using its sensor libraries. You can program a track that the car attempts to stay within and drive to specific locations through different localisation and path planning algorithms.

There are many localisation algorithms that can be used as a solution for the robot. One algorithm could be the Monte-Carlo localisation algorithm, which is specific for robots to calculate their



Figure 1 – Bug-0 algorithm behaviour performed, [4] pg. 1

coordinates in the environment, with taking the robot's odometry (robot's current location and its current heading direction), the map's environment (known landmarks on the map and the robot's distance to them) and sensor data, which is used to gather the distances, [3] pg. 162.

The simulator also requires a use of planning algorithms to be used for the robot to reach its destination from its starting point. One of these algorithms is called the Bug Algorithm. This algorithm can be implemented on a map that is unknown to the robot, where it uses its starting point and compares it to the target point. The car then travels in the target's general direction, while following any walls on the way (Figure 1). This way the car learns the path while also reaching its destination without knowing the whole map's layout.

Another planning algorithm that plays a part in the robot's steering is the Ackermann Steering Geometry, which is used in ROS as a series of dependencies which allows robots with wheels to turn realistically. In the simulator, the car's wheels would be connected via trapezoid linkage, labelled ψ (ABCD in Figure 2). The wheels would turn right once all wheels are faced in the direction of centre O , with left wheel turned θ_1 and right wheel turned θ_2 . The equation for BC (l_{20}) when turning angle is zero can be seen in Equation 1.

$$l_{20} = l_4 - 2l_1 \cos \psi$$

Equation 1 - Length of link BC when turning angle equals zero

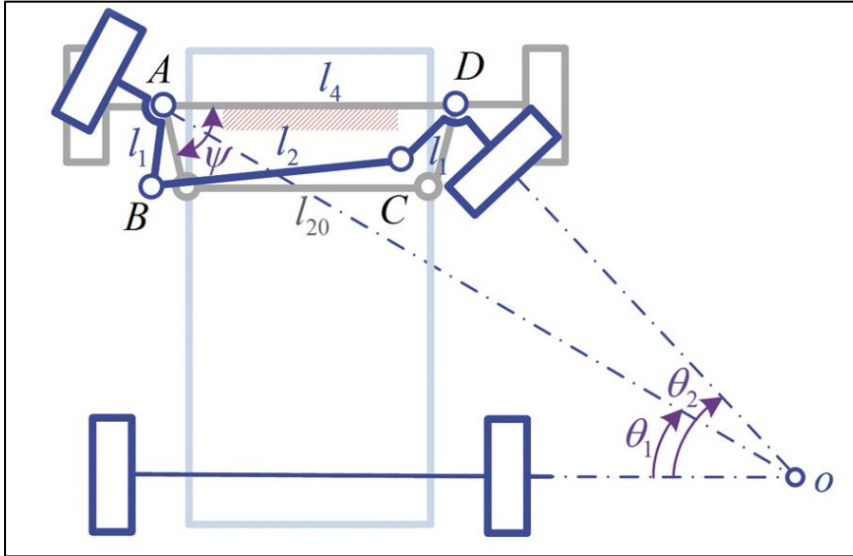


Figure 2 - Ideal Ackermann turning geometry with 4 bar linkages [7]

3.2 Robotics Operating System (ROS)

Since ROS is a new concept to me, there are a wide variety of documentations and tutorials online to swiftly help me understand the technology to be used within this project. The F1Tenth website [2] supports set up guides, software optimisations and lecture tutorials for understanding ROS. By using these resources that are tailored to the F1Tenth simulator, the learning curve becomes much more manageable. To install, configure and run Python-coded ROS programs, Section 5 explains these points in detail.

3.3 F1Tenth Hardware

Although this project mainly focuses on the simulator, the F1Tenth hardware can be used to test the car's capabilities without any risk of damaging the physical car, instead, creating a virtual environment first. After creating this scenario, these components can then be transferred to the physical machine itself to perform the tasks in the physical environment, to see how it performs in a real-life scenario. Within this project's early deliverables, a section is present, explaining how the simulator's programming can be transferrable to F1Tenth's hardware, how sensors and its other components are be used from the simulator's programming. The car's hardware expanded on in Section 5, where all

the main components are talked about and how the simulator would interact with the physical car, alongside the difference between the hardware and software variants of the F1Tenth.

3.4 ROS Topics

Within this project, ROS software uses a series of topics that programs subscribes or publishes to. Topics consist of data structures that can be an integer, Boolean or other data types, which can either be changed by the user or viewed, depending on if they are publishing or subscribing to the topic. I am using these tools constantly as they are a compulsory part of this project.

When subscribing to topics, you can access that topic's variables and their data. As a result, programs can be created that react to specific readings of those topic variables, for example, if a specific LaserScan range reading is reached, some action can occur. To make the car do some action in the virtual environment, you publish data to topics. With publishing to a topic, it manipulates the variable that is updated within the program and changes it, outputting it within the executed program. For example, by updating the speed of the car to 2m/s in the AckermannDriveStamped topic, the car now moves at 2m/s.

The main topics used in this project are LaserScan, AckermannDriveStamped and Odometry.

3.4.1 LaserScan Topic

The LaserScan topic is one of the main topics subscribed to within this project. This topic holds messages of type Float, where the primary message used from LaserScan is called ranges[], which holds over 1080 values of range measurements, each storing a value of distance between the car and the obstacle detected. This specific message is used in all programs where scan is used to identify object distances from the car at different angles.

3.4.2 AckermannDriveStamped Topic

AckermannDriveStamped (abbreviated to Ackermann) is the primary topic to be published to in this project. This topic allows the control of the car's movement, with all the messages using type Float. In this topic, the message 'speed' is used to publish how fast the car's constant speed should be, measured in metres per second (m/s). Another message which is frequently used is 'steering_angle', which controls how the car turns, measured in radians. This message can range between -1 to 1, with negative numbers meaning the car turns left and positive for right turns. The larger the number, the tighter the turn.

3.4.3 Odometry Topic

The odometry topic is used to track the position and orientation of a robot relative to its starting point. The topic publishes messages that contain information on the robot's linear and angular velocities, as well as its position and orientation. The messages are of type Odometry, which consists of various fields, including position, orientation, linear velocity, and angular velocity. The primary use of its messages is that programs can access the robot's current position and orientation, allowing them to plan and execute precise movements.

Chapter 4: F1Tenth car hardware, sensors, and communication

4.1 Introduction

In this chapter, I discuss the hardware components of the F1Tenth, mainly their importance of how they work together to create an autonomous vehicle. It covers the basis of the robot's chassis, describing the different features that create the smooth manoeuvrability on different indoor surfaces. Afterwards, a discussion how the hardware's aspects may appear similar within the F1Tenth simulator, seeing how similar their performances can be transferred from testing in the virtual environment and the live one.

4.2 F1Tenth Hardware

Scroll right to see links	Qty	Cost per Unit
Chassis		
Traxxas Slash 4x4 Platinum Edition, 1/10 Scale Brushless Pro 4WD	1	\$269.95
2 Lipos and Charger Combo	1	\$199.95
Antenna Mount (3D print)	1	\$10.00
Platform Deck (laser-cut)	1	\$20.00
Fasteners		
M3 Socket Head Kit	1	\$17.98
M5 Socket Head Kit	1	\$8.99
6mm hex 14mm M3 standoffs	2	\$3.12
6mm hex 25mm M3 standoffs	4	\$4.56
6mm hex 45mm M3 standoffs	6	\$3.69
M2.5 x 0.45mm, 6mm long	4	\$6.87
M2.5 x 0.45mm, 10mm long FF Standoffs	4	\$0.62
Compute module		
Jetson Xavier NX	1	\$599.99
Micro SD Card 32 GB	1	\$7.50
NVMe SSD Card 250 Gb	1	\$75.00
Sensors		
Hokuyo 10LX	1	\$1,600.00
Electronics		
VESC 6 MkV	1	\$249.00
Joystick Opt 1: Logitech	1	\$39.00
Joystick Opt 2: DualShock 4 for PS4	1	\$69.99
LiPo safety bag like the Aketec Silver Large Size Lipo Battery Guard Sleeve/Bag for Charge & Storage.	2	\$9.99
Barrel Jack to Pigtail	1	\$4.33
12V 5A Power Adapter (optional)	1	\$13.99
Antenna	1	\$7.99
Intel RealSense D345i (optional)	1	\$179.00
Miscellaneous		
TRX to XT90 Adapter	1	\$11.99
Traxxas id charge lead adapter	1	\$15.99
Header Pins	1	\$7.99
VESC ppm cable	1	\$6.95
HDMI emulator	1	\$8.50
short (~1 ft) A USB-to-microUSB cable	1	\$7.98
Bullet Adapter 4mm Male 3.5mm Female	1	\$5.99
Total:		\$3,478.75

Figure 4 - Bill of Contents showing F1Tenth components. Main components highlighted [2]

The car has multiple components that are combined to create the final product, I discuss the main ones that have a major impact on its performance (as highlighted in Figure 4).

4.2.1 Traxxas Slash 4x4 Premium Chassis

The car chassis is based off the Traxxas Slash 4x4 Premium Chassis (Figure 3), where all its components are attached onto its body [8]. This chassis was created to have a low centre of gravity by having the battery and electronics to be held low to position the weight of car low in the chassis. Because of the low centre of gravity, it is possible that the car's stability and ability to take on corners at an increased speed is improved. The chassis itself is ultra-smooth, which is documented by the Traxxas saying it reduces drag and improves ability to take on many types of terrain.

Included with the chassis, you have the GTR Shocks suspension system, a power system, wheels, and the body itself to hold all these components [8].

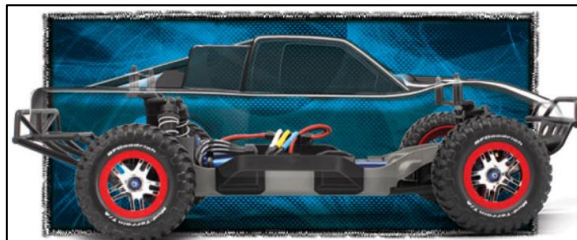


Figure 3 - Traxxas 4x4 Premium Chassis used for F1Tenth car – Traxxas [8]

4.2.2 High-capacity GTR Shocks (suspension)

[8] The shock system has a near frictionless piston travel due to its “PTFE-coating”, allowing for smooth suspension work when driving through ridged terrain. This also allows for smooth and comfortable suspension action for extended periods of time before potential replacing. The shocks have an option to also change spring reload and ride height of the car, which can be done simply by turning threaded spring collars. This allows for fast and easy adjustment to the suspension, making the F1Tenth versatile.

4.2.3 Traxxas Velineon Power System

The power system has been optimised for high-speed performance while maintaining smooth driving, to preserve control over the vehicle [8].

4.2.4 LiPo batteries & Charger

¹These rechargeable Traxxas lithium polymer (LiPo) batteries (Figure 5) are the main source of power for the car to work, along with a charger for these batteries. Due to them being rechargeable, it allows the car to be a long-term investment, saving up on money as you do not have to restock on additional batteries. With the two ‘5000mAh’ batteries, the car has a possibility of reaching speeds of over 60mph (Figure 6).

SLASH 4X4 lets you choose the performance you want! Select the right Traxxas iD battery to fit your needs from the options below			
Speed	35+mph	40+mph*	60+mph*
Battery	2923X (1) 3000mAh NiMH	2843X (1) 5800mAh 2S LiPo	2872X (1) 5000mAh 3S LiPo
Pinion/Spur	13-T / 54-T	19-T / 54-T	19-T / 54-T
Skill Level	1	4	5

Larger pinion gear/smaller spur gear combinations are for high-speed running on hard, smooth surfaces only.
*With included optional gearing.

Figure 6 - Traxxas EZ-Peak 3S "Completer Pack" Dual Multi-Chemistry Battery Charger w/Two Power Cell Batteries (5000mAh) – Traxxas



Figure 5 - Traxxas EZ-Peak 3S "Completer Pack" Dual Multi-Chemistry Battery Charger w/Two Power Cell Batteries (5000mAh) - Traxxas

4.2.5 VESC 6 Mk VI

The TRAMPA VESC (Figure 7) allows you to control and adjust the speed and steering of the car. This is caused by a command (from the Jetson Xavier NX as discussed in next section) fed to the VESC to increase or lower the voltage of the motor as necessary, which changes the propeller speed. The MK VI contains three phase shunts and an adjustable current/voltage filter, which allows a more precise and reliable detection of attached motors. This also allows motors to run smoothly and react linear to throttle input commands.

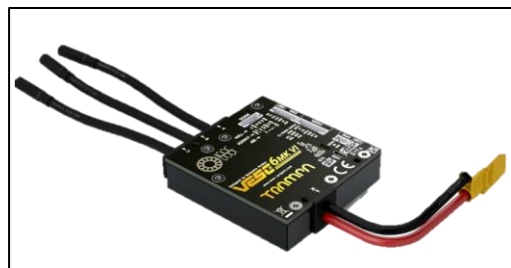


Figure 7 - VESC 6 Mk VI – TRAMPA.

¹ LiPo batteries & Charger – <https://www.amainhobbies.com/traxxas-ezpeak-3s-completer-pack-dual-multichemistry-battery-charger-tra2990/p474595>

4.2.6 NVIDIA Jetson Xavier NX

The NVIDIA Jetson Xavier NX Developer Kit [9] (will be simplified to JXNX) is the primary component of the F1Tenth's system, due to it having control over the VESC, which is a controller that oversees controlling and regulating the speed of the car's motor [2]. The JXNX has the possibility to give out commands on the steering and speed control of the car due to having control over VESC, feeding it these commands. The JXNX also can receive information from the LIDAR sensor (Figure 8), which is one of the primary sources of information that dictate the regulation of speed.

The way that JXNX receives these commands is through the host computer that is used to code and control the software of the F1Tenth, connecting remotely through SSH [2].

4.3 Ranging Sensors – Lidar & radar

Lidar sensors in the robotics field are used for many different reasons to help the robot in completing its set tasks. It can, for example, assist in navigation and obstacle avoidance. It is an active remote sensor, which judges an object's (or wall's) distance by measuring the time taken for the light beam to return to the sensor [8].

4.3.1 Hokuyo UST-10LX Scanning Laser Rangefinder

This Lidar light sensor² (Figure 8) is used for the physical car, which can detect obstacles at a long distance of maximum being 30 metres, allowing the F1Tenth to conduct localisation of the car, with the scan being at a 270-degree angle (Figure 9). The sensor uses a light source as a way of obstacle detection, which in effect allows for a near immediate detection time, with scan speed measured to

Model Number	URG-04LX-UG01	URG-04LX	URG-04LX-F01	NEW UST-10LX	NEW UST-20LX
Appearance					
Light source	Semiconductor laser diode (FDA approval, Laser safety class 1)				
Scanning range	0.02 to 5.6m 240-degree		0.02 to 10m 270-degree		0.02 to 20m 270-degree
Measuring Accuracy	±/-30mm		±/-10mm		±/-40mm
Angular resolution	0.352 degrees (360/1,024)		0.25 degrees (360/1,400)		
Scanning frequency	10Hz (600rpm)		36Hz (2160rpm)		40Hz (2,400rpm)
Multi Echo Function	N/A				
Communication Protocol	SCIP 2.0	SCIP 1.1/2.0	SCIP 2.0	SCIP 2.2	
Communication and I/O Interface	USB1.1/2.0	USB1.1/2.0, RS-232C		Ethernet 100BASE-TX	
Power source	N/A	Scanner synchronous and Failure output			
Power source	USB bus power	5VDC	12VDC	12V or 24VDC	
Power Consumption	0.5A or less		0.4A or less		0.15A or less (on 24VDC)
Ambient Illuminance	10,000 lx or less				
Protective Structure	—		IP40	IP65	
Weight	160g		185g		130g
Size (W x D x H) mm	50 x 50 x 70		60 x 75 x 60		50 x 50 x 70
Reference price					
Note	Inventoried item	Inventoried item	MTO item	It will be scheduled to go on sale in January 2014. Specifications are subject to change without notice.	

Figure 9 - Lidar UST-10LX Scanning Laser Rangefinder Specifications – [UST-10LX]

Figure 8 - Hokuyo UST-10LX Scanning Laser Rangefinder – UST-10LX.



² <https://www.robotshop.com/en/hokuyo-ust-10lx-scanning-laser-rangefinder.html> - Light sensor details

4.4 Hardware Communication with ROS

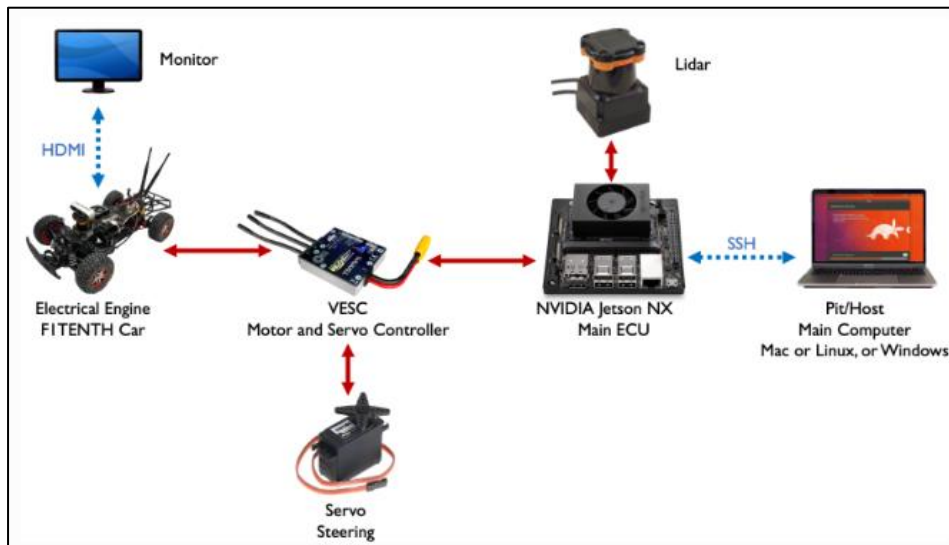


Figure 10 - Depiction of Data Flow of the F1Tenth Car [2]

The NVIDIA Jetson NX is the primary way that programs from the F1Tenth Autonomous Vehicle System are run for the car to operate and can communicate with the vehicle (as seen in Figure 10)³.

The Host laptop is the main computer that access the car and its data, also to start its software. This is the main way of connecting with the NVIDIA Jetson NX via SSH in a remote way (Figure 11), due to the Jetson having Wi-Fi capabilities. This allows the user to create a way to use the car for different purposes. Figures 10 & 11 both show that a monitor, mouse, and keyboard can be used to connect to the car, but only if its stationary. With Wi-Fi capabilities, the car can be connected to at any time without any wired connections needed. This creates a possibility to access the car while driving.

As discussed in the Jetson NX section, this component takes data input from the LIDAR sensor to assess the current environment it is driving in and use it to regulate speed by overseeing the VESC. This is where this data is used, regulating the motor speed and steering control.

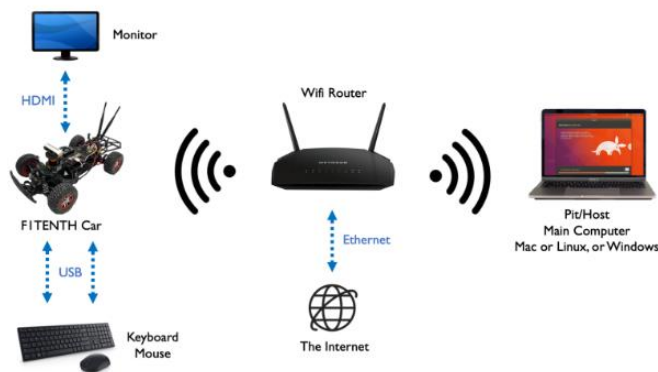


Figure 11 - Diagram showing possibility to connect to Jetson via WiFi [2]

³ https://f1tenth.readthedocs.io/en/stable/getting_started/software_setup/index.html - F1Tenth Configuration Site

4.5 F1Tenth Car Comparisons to Simulator

One similarity that the F1Tenth car has to the simulator is the way that obstacle detection works. With the car itself, the Lidar sensor (Figure 8) is used to detect obstacles via light sensor readings. This allows a precise detection of up to 30 metres from the sensor and near instant data collection.

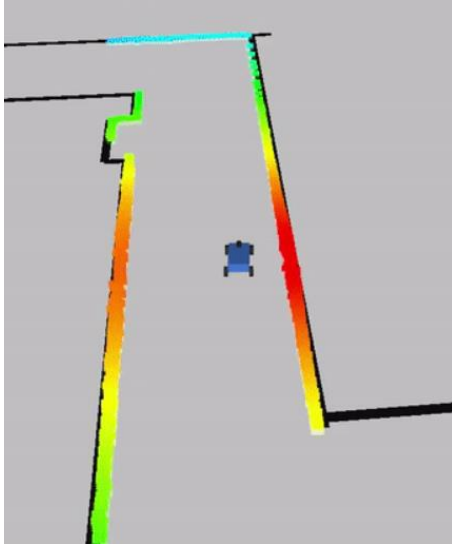


Figure 12 - F1Tenth Simulator heatmap of obstacles

Meanwhile, the simulator uses a 'sensor' of its own, by tracking in a 270-degree cone (just like the actual Lidar sensor) all obstacles to a specific distance. The simulator produces constant updates of obstacle distance from the car every frame. To display the sensor distance, a heat map is used to show the distance of the object detected (Figure 12), the closer the obstacle is, the hotter the colour (e.g., red for very close, blue/purple for far).

PID control is an important aspect of the simulator, where the car's speed and steering are regulated by the constant PID loop. For steering, the car may be programmed to follow a wall at a specific distance, in this case PID function calculates the distance error and correct it by getting closer or further from the wall. Overtime, the car can navigate its way to be at that specific distance from the wall (Figure 13) with each cycle being a smaller error margin until there is no error in distance from the wall. The physical car works the same as

the simulator, with the process being same as Figure 17, with instead the VESC (Figure 11) overseeing this steering and speed adjustment of the car.

The difference in the simulator and real-life car is that there is no noticeable obstacle detection delay in the real car, since speed of light travel is near instant within the workspace the car is tested in (usually corridors). Meanwhile, processing instructions within the simulator is slower, albeit also a miniscule delay, since there are multiple instruction lines that need to be processed about an obstacle. The number of instructions will be larger the further away the obstacle is. Moreover, for each obstacle, a line needs to be traced from the car to the obstacle, which is updated constantly, requiring even more instructions in need of processing. The simulator may be further delayed from program execution time if computer specifications are weak.

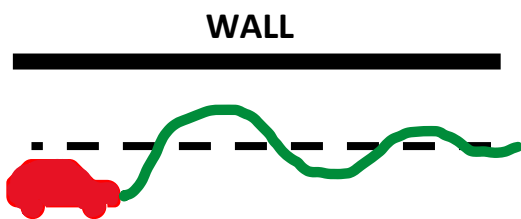


Figure 13 - Example of PID control and car navigating to follow wall

Chapter 5: Installation and Configuration of ROS and the F1Tenth Simulator

This section discusses the installation and configuration process to be able to use the ROS environment and implement it within the F1Tenth simulator that are used within this project. From installation of a virtual machine, use of Linux operating system, to installing the necessary packages and software to use the F1Tenth simulator.

5.1 Installation of Ubuntu Operating System

Ubuntu OS is the most stable setup to use the ROS environment and the F1Tenth simulator. This is because the simulator is only supported in Ubuntu at this moment in time, while ROS can only run natively in Linux [2]. ROS is one of the prerequisites, which should be installed before setting up the F1Tenth simulator. Since the work on this report is done locally, installing Ubuntu on a virtual machine was most suitable.

5.1.1 Installing Oracle VM VirtualBox

Oracle VM VirtualBox is the Virtual Machine of choice for this project (version used for this project is 6.1.38). It can easily be installed from their official website, by either selecting the current most stable version (as of 22/11/2022, version 7.0.x is the most recent version), or older version of the software from their website⁴. The installation type depends on the host OS. Having VirtualBox set-up is a prerequisite for Windows hosts, with the installation software having easy to follow steps.

5.1.2 Ubuntu Installation – Version 18.04 LTS (Bionic Beaver)

It is required to install an ISO image file of the OS to be able to emulate Ubuntu on the VM. In the project's case Desktop image of Ubuntu Version 18.04 LTS (Bionic Beaver)⁵. The use for this specific version is explained in the "Installation & Configuration of ROS Environment" section. Configuration of this program is followed from the Oracle VirtualBox official documentation [5], with in-depth information regarding set-up of virtual machines after downloading the program.

5.2 Installation & Configuration of ROS Environment

After the successful launch of the virtual machine, with full support of Ubuntu OS, the next step is to download all the required ROS packages. The ROS packages have been built on the Debian package format system, which is supported on Linux. To complete this installation, it required to use the terminal built in to download and configure all necessary files.

Within the F1Tenth build webpage, it is stated that the version of ROS required is ROS Melodic, which is most stable within the Ubuntu 18.04 LTS Operating System, hence why this version was chosen [2]. To download the environment, follow the ROS Melodic installation documentation [6].

⁴ <https://www.virtualbox.org/wiki/Downloads> - VirtualBox download section.

⁵ https://releases.ubuntu.com/18.04/?_ga=2.28871484.1441637138.1669127045-503508063.1662037585 – Ubuntu version 18.04 LTS Bionic Beaver ISO image file.

5.3 Installation of F1Tenth Simulator

With all the pre-requisites installed, the simulator is also downloaded using terminal commands. To set up the F1Tenth Simulator, the website [2] offers an installation guide with all terminal commands required. Once everything is installed, you are now ready to run the simulator through the command:

```
roslaunch f1tenth_simulator simulator.launch
```

This command launches all the required tools needed for the simulation: roscore, the simulator, a preselected map, a model of the race car, and the joystick server [2]. When running, the Rviz acts as the master node, which allows the use of subscribing and publishing between programs. Figure 6 shows the full simulator launched:

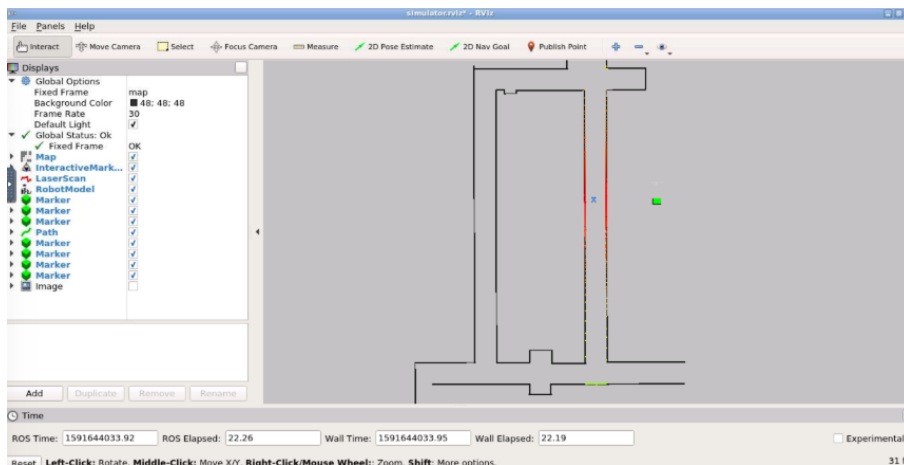


Figure 14 - F1Tenth simulator launched

5.4 Running Programs with ROS and F1Tenth launcher

To run any custom-made programs with the F1Tenth simulator, it is recommended to use two separate terminal windows – one for running the launcher, which acts as a master node, while the other runs the program itself. The steps are:

1. In the first terminal, enter your catkin workspace:
`cd ~/catkin_ws`
2. When in there, run the same command as before to set up the environment that allows ROS features to run:
`source devel/setup.bash`
3. Finally, launch the simulator:
`roslaunch f1tenth_simulator simulator.launch`

With the simulator now running (Figure 6), this has created a master node connection that allows the communications of topics between nodes, and to use all features within the visualiser launched (Rviz).

To run the wanted programs, you use the second terminal window, following these steps below:

1. Repeat steps 1 and 2 from previous terminal window to set up the environment for ROS

2. Inside the catkin workspace, find the program and run it with python:

```
cd ~/catkin_ws/src/<packages holding node>
```

```
python <program name>.py
```

With these steps, when you enter the command to run python program, you can view the actions done by the code within the visualiser, such as a program that has the car avoiding obstacles would start driving and meandering around walls.

Chapter 6: Path Planning and Localization Algorithms

6.1 Bug Algorithm (Path Planning)

The Bug Algorithm is one of the better-known algorithms for robot navigation when traversing an unknown, continuous environment, while using non-deterministic uncertainty [12]. This means that due to the randomness of different environments that the algorithm may be used, the results and the process of the program may be not possible to guess. This algorithm involves having the robot travelling towards its desired location, and following any obstacles along the way until the goal is reached [11]. Although the robot does not know the environment, it will know where its current location is, the location of the target goal, and the orientation of its travel and can locally sense obstacles [11].

The way this algorithm works is by initially having the robot move in a straight line towards the direction of the goal until an obstacle is encountered. Once this happens, the robot follows the boundary of the wall, left or right depending on how the algorithm is implemented, with the distance maintained from the obstacle [11]. This will continue until the robot is able to continue travelling straight towards to the goal, repeating this process until the goal is reached.

There are different ways of implementing the algorithm, each with better (or worse) approaches than the previous, with some thriving in different scenarios more than others, that are discussed in the sub-sections below.

6.1.1 Bug-0

Bug-0 is the base version of this algorithm, doing precisely what was explained in the Bug Algorithm explanation: robot goes in a straight line to the goal, if an obstacle is detected the robot will follow the boundary in a pre-determined direction. Once the sight of the target is visible again, it goes to goal again, repeating until reaching the target location [11].

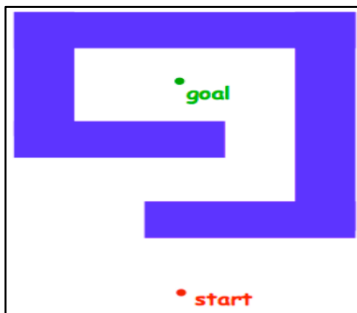


Figure 15 - Map example that causes a failure in Bug-0 execution [11]

Although this implementation satisfies many basic map grids, there are some limitations that may confound the Bug-0, an example is seen in Figure 15 where the robot would have to follow obstacles in both the left and right directions at some point. We know that this program would have the robot pre-determine a specific direction that walls would be followed, either left or right, but not both ways in one program run – therefore never reaching the goal.

An example is if the pre-determined turning is set to be to the left, the first obstacle met will work as normal, but once the second obstacle is reached, the robot will go endlessly to the left and around the obstacle without termination. With this limitation, an updated version of this algorithm was created to avoid such problems – called Bug-1.

6.1.2 Bug-1

This implementation improves upon the previous version by adding memory to the program, where the robot stores its past locations on the grid. This then allows the program to run as before, but when it reaches an obstacle, the robot circumnavigates and as a result, trace the boundaries [12]. Once the robot reaches an already mapped part of the obstacle, it turns and return to the point that has the closest approach to the goal, then head towards it, repeating the process until success [11]. The way

that the car knows the closest point to goal is by having a marker variable which stores this location. An example of this program is in Figure 16 of Bug-1 running.

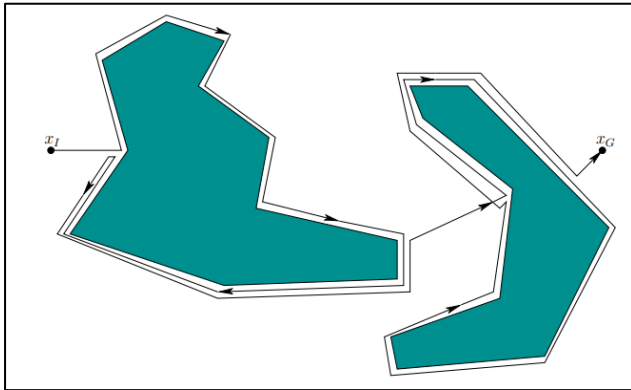


Figure 16 - Example map showing execution of Bug-1 with path shown [12]

Figure n+1 gives us a clear view to understand that the Bug-1 implementation is an exhaustive search algorithm, whereby it identifies and looks at all its choices before committing to a decision on where to continue traversing from [11]. Due to the algorithm's nature, it has a predictable performance, as at some point the algorithm reaches its goal or terminate if failed.

6.1.3 Bug-2

Bug-2 Algorithm is known as a greedy algorithm – as it takes the first opportunity that looks best [11], pre-determined to be the line from start to the goal. This algorithm was an alternative to Bug-1, where it strives to be faster by not doing an exhaustive search, instead, the robot always attempts to move along a line that connects the initial and goal positions [12]. Whenever the robot is on this line, the direction of the goal will be the same as from the initial location or may be inverse if travelling from the other side of the goal [12]. This process is then repeated until the goal is reached. An example of this algorithm running is seen in Figure 17.

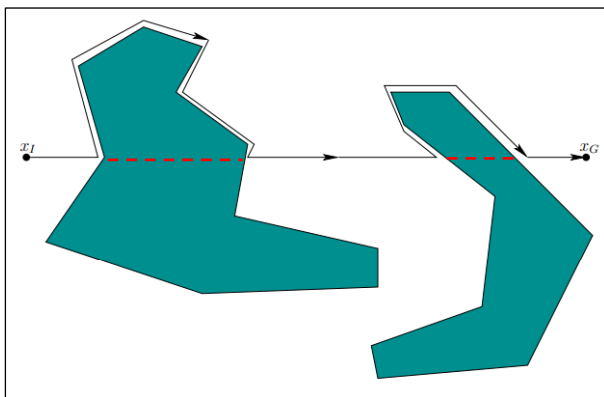


Figure 17 - Example map showing execution of Bug-2 with path shown [12]

6.1.4 Limitations of Bug Algorithm

Limitations of algorithms		
Bug 0	Bug 1	Bug 2
<ul style="list-style-type: none"> A very basic algorithm, which fails to get to goal location if met with dead-ends, or obstacles as seen in Figure 15. Bug-0 struggles in a complex environment such as narrow passages or with many obstacles. The robot may get stuck in an endless loop, with no memory of where it has already travelled or a failsafe to exit the loop. 	<p>There are multiple limitations when this is met with multiple obstacles:</p> <ul style="list-style-type: none"> Bug-1 will take a significant amount of time to reach goal – inefficient execution time. It takes up a large amount of memory with the car storing many paths done. <p>Example for this is seen in Figure 18, where each obstacle is traversed 1½ times, which is the worst case for the algorithm.</p>	<ul style="list-style-type: none"> With the algorithm following the m line towards the end goal, it may get stuck in a recursive loop as seen in Figure 19. The algorithm may not choose the most optimal route.

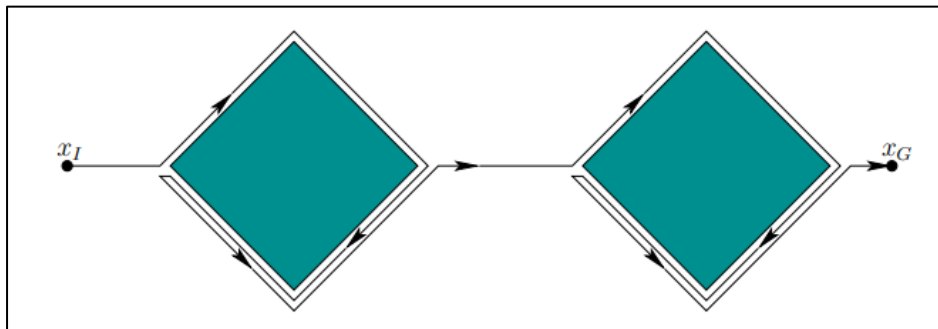


Figure 18 - Map example that causes a bad case scenario in Bug-1 execution with path shown [12]

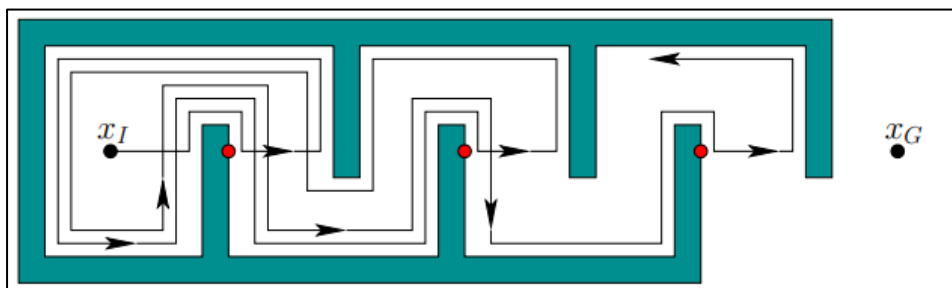


Figure 19 - Map example that causes a bad case scenario in Bug-2 execution with path shown [12]

There are some general problems with all implementations of the algorithm, such as sensor noises and inaccuracies, as the algorithm relies on having accurate sensing when detecting obstacles. With this limitation, it may cause the robot to become lost in its environment, stuck or potentially make an incorrect or inefficient choice [4].

This may be applied in the real-world scenario where the robot may not be placed in a pre-planned environment, which could have unpredictable obstacles and terrain. With this unplanned terrain, the robot may struggle to reach its goal. Weather that may obscure visibility, such as fogs, can also hinder the algorithm to be limited as obstacle detection may be late or not at all which may lead to a potential crash. Another real-world scenario limitation is obstacles in an outdoor environment may have complex shapes and be non-convex. Since the algorithm is created to traverse smooth and convex obstacles, the robot using this program may struggle to navigate around or potentially get stuck due to tight interior angles. Therefore, the probability of the robot reaching its goal could be lowered by these issues.

Another challenge in implementing of the Bug Algorithm is deciding when the robot has reached a point where it can resume its straight-line trajectory towards the goal, which can be narrowed down to two common approaches. The first approach is to define a "line-of-sight" specification, where the robot continues following the boundary until it can see the goal location without any obstacles in the way, which occurs in the Bug-0 version. The other approach is to define a "minimum-distance-to-goal" criterion, where the robot follows the boundary until it reaches a point where the distance to the goal is shorter than any other point on the boundary it has encountered so far, which is seen used slightly differently in Bug-1 and Bug-2 versions.

6.2 SLAM

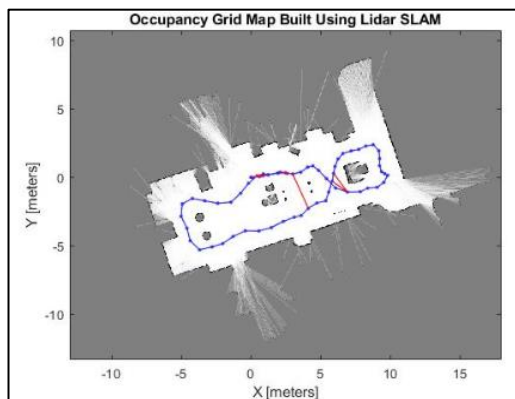


Figure 20 - Example of SLAM mapping through localization

The Simultaneous Localization and Mapping (SLAM) algorithm is used in autonomous robots that offers map building of the environment it is placed in and helps locate the robot in this map [13]. The way that this map is built by the robot over time as the car drives through the environment, storing its scans to build the map (Figure 20). SLAM allows you to implement robots into unknown environments and create maps. This tool could help greatly with mapping locations in the real world that may be hard or impossible to access by a human, such as cave explorations in geological studies. Scans of the cave can be done to draw the outline of cave systems without putting lives under risk.

This is a learning algorithm, where the more runs in an unknown area happen, the better the map quality is generated [14]. The way that this algorithm runs is by collecting odometry data to identify its location and laser scan data to gather information regarding robot's surroundings from its current location. SLAM then creates a 2-D occupancy grid map from the data collected [14].

6.2.1 Implementation within ROS

ROS software can provide message passing that can be used to transfer data from the robot to the program running, which then allows the code to use and manipulate the data gathered in

implemented functions. With the case of SLAM, it requires Odometry data and laser scan data to work, then implement this algorithm onto a grid map. When it comes to using a grid map, ROS offers the use of a library called 'grid_map' that converts the map into grids and can be used for navigation. Within the ROS visualiser – Rviz – you are also able to show laser scan lines that the program can detect, which can be used for creating a map from the car driving within the environment.

6.3 Case study SLAM/ Path Planning: Search & Rescue using a 'Robot Team'

Autonomous vehicles would become a high-valued asset when it comes to travelling through environments that may seem hazardous to human life and put them at risk. As an example, this would apply to search and rescue missions in locations that may seem hard to reach for humans or may be dangerous to send a rescuer. When a person is lost in extreme environments, such as inside a burning building or if the building is filled with poisonous substances, this terrain will be unknown and most likely unmapped. This requires the ability of being able to locate where you are and map the terrain to be able to manoeuvre this space swiftly.

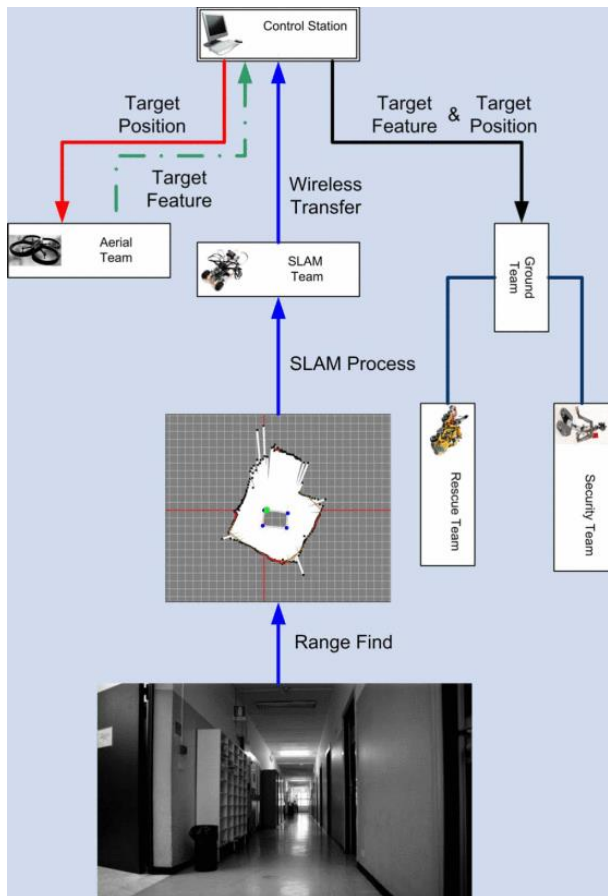


Figure 21 - Search & Rescue robot team system architecture [15]

These situations require the robot to navigate through the environment precisely and fast, so if the robot gets lost in the terrain or doesn't find its target in a short time, this could potentially cost lives, which cannot be afforded in this scenario. When discussed the Bug Algorithm section about its limitations, if the environment consists of complex, non-convex walls or obstacles, the robot may get stuck, crash into the obstacle, or take a much longer time to navigate through the terrain. This means that when creating a robot this task, the developers must carefully consider what algorithms will be best for the purpose of fast navigation and that it should have a low error rate.

An idea has been developed to tackle this task and make it a reality where autonomous robots could be used in indoor search and rescue missions. This solution uses a team of robots that work heterogeneously to achieve the goals of finding the targets and taking the appropriate course of action whether to rescue or secure [15]. By having multiple robots completing tasks they each would specialise in, it could drastically reduce the time it would take to locate and rescue targets.

In a scenario of a search and rescue operation, the team consists of multiple robots, shown in Figure 21. This team consists of the SLAM team – a ground vehicle responsible for simultaneous localization and mapping (SLAM), an aerial vehicle (aerial team), an extra ground vehicle (ground team) and finally the backup team on standby [15].

The SLAM ground vehicle gathers information regarding the building and explore the search area, while the aerial vehicle will search the area that was explored by the SLAM team using range finders such as LidarScan or cameras. Once the target is found, the camera feed is relayed to the control station to assess the situation and determine the type of threat. If the situation calls for action, the ground vehicle proceeds towards the target and complete the mission set by the control station, whether it is a rescue or security mission. In case the ground vehicle needs further assistance, a backup team will be dispatched to provide additional support [15].

By deploying multiple robots with different capabilities, the team can efficiently explore and assess the search area, quickly identify potential threats or victims, and respond accordingly to the situation at hand. In this study, this test has conducted 48 trials – 28 using both ground and aerial vehicles, and the other 28 for just ground vehicles. The results of these trials are seen in Table 1, where the time was recorded by taking the average time taken of all trials for each scenario. This table shows that the team consisting of ground and aerial vehicles have significantly faster in all tests, taking around half the time in comparison to the ground vehicle only team. As a result, the statement before proves that the support of an aerial vehicle to split tasks of location and execution could reduce the search time.

Time		
Distance	MGV & MAV	MGV
5m	195 seconds	320 seconds
10m	343 seconds	795 seconds
15m	504 seconds	1230 seconds

Table 1 - summary of test results, average time of trials [15] MGV – Mini Ground Vehicle. MAV – Mini Aerial Vehicle

Chapter 7: Important Programs in this Project

7.1 Message Passing

7.1.1 Nodes & Topics

[10] The primary way that you program within ROS is using nodes. These are processes that perform defined basic tasks and computations within the systems, such as sending and receiving messages. Alongside these sets of nodes, there is a *Master Node* that holds all the data about the other nodes within the ROS network.

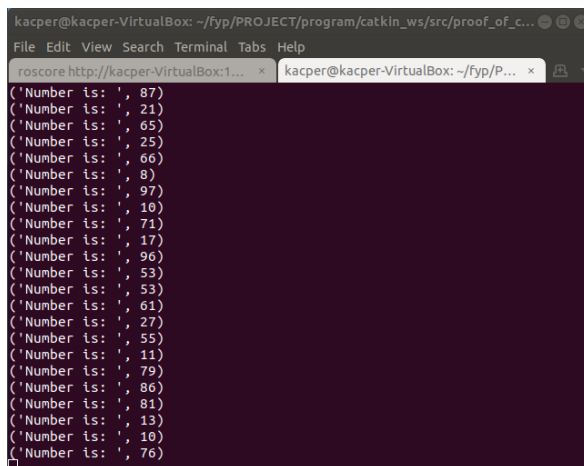
Nodes communicate by sending messages on topics, which they can become a part of. Each node which is part of some topic can receive all messages sent from other nodes that are part of this topic. These messages consist of a simple data structure, which can be an *int*, *string* or any other type.

The most common form of communication is by using a publisher-subscriber model. A topic *'foo'* can exist, where a publisher node sends a message to, and where a subscriber node can receive messages from *'foo'*, this is because of this topic channel being shared between these two nodes. All information that is to be shared with other nodes of that topic are stored in that message.

The programs below are important to understand how topics are accessed and used throughout the project and its programs.

7.1.2 Publisher

The publisher program would create a new publisher that would send a series random numbers generated as messages to the *'num_output'* topic. This program outputs *"Number is: [random number]"* message into the terminal in an endless loop until the program is terminated, as seen in Figure 22.



```

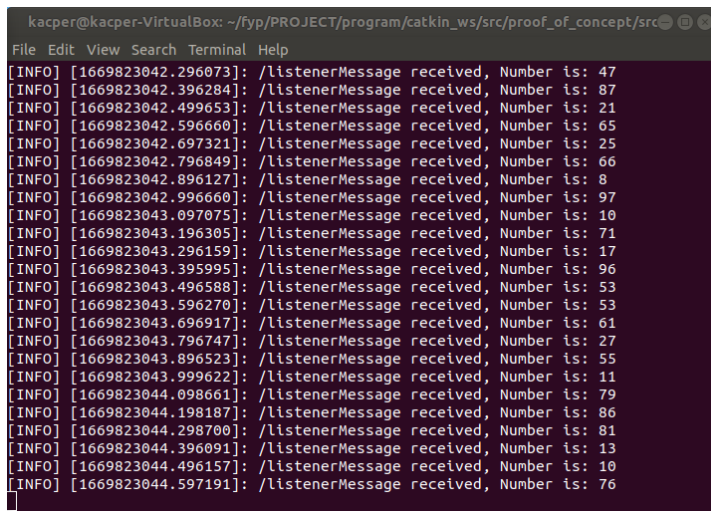
kacper@kacper-VirtualBox: ~/fyp/PROJECT/program/catkin_ws/src/proof_of_c...
File Edit View Search Terminal Tabs Help
roscore http://kacper-VirtualBox:1... x kacper@kacper-VirtualBox: ~/fyp/P... x
('Number is: ', 87)
('Number is: ', 21)
('Number is: ', 65)
('Number is: ', 25)
('Number is: ', 60)
('Number is: ', 8)
('Number is: ', 97)
('Number is: ', 10)
('Number is: ', 71)
('Number is: ', 17)
('Number is: ', 96)
('Number is: ', 53)
('Number is: ', 53)
('Number is: ', 61)
('Number is: ', 27)
('Number is: ', 55)
('Number is: ', 11)
('Number is: ', 79)
('Number is: ', 86)
('Number is: ', 81)
('Number is: ', 13)
('Number is: ', 10)
('Number is: ', 76)

```

Figure 22 - Output of Publisher.py

7.1.3 Subscriber

The subscriber program receives a series random numbers generated as messages in the *'num_output'* topic thanks to the publisher program before. This program outputs the local node ID alongside *"Message received, Number is: [random number]"* message into the terminal in an endless loop until the program is terminated or no more messages are passed in the topic, as seen in Figure 23.



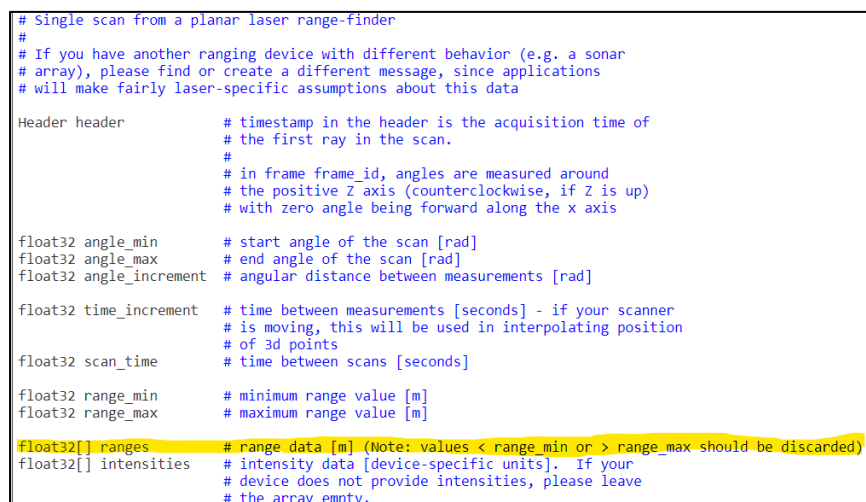
```
kacper@kacper-VirtualBox: ~/fyp/PROJECT/program/catkin_ws/src/proof_of_concept/src
File Edit View Search Terminal Help
[INFO] [1669823042.296073]: /listenerMessage received, Number is: 47
[INFO] [1669823042.396284]: /listenerMessage received, Number is: 87
[INFO] [1669823042.499653]: /listenerMessage received, Number is: 21
[INFO] [1669823042.596660]: /listenerMessage received, Number is: 65
[INFO] [1669823042.697321]: /listenerMessage received, Number is: 25
[INFO] [1669823042.796849]: /listenerMessage received, Number is: 66
[INFO] [1669823042.896127]: /listenerMessage received, Number is: 8
[INFO] [1669823042.996660]: /listenerMessage received, Number is: 97
[INFO] [1669823043.097075]: /listenerMessage received, Number is: 10
[INFO] [1669823043.196305]: /listenerMessage received, Number is: 71
[INFO] [1669823043.296159]: /listenerMessage received, Number is: 17
[INFO] [1669823043.395995]: /listenerMessage received, Number is: 96
[INFO] [1669823043.496588]: /listenerMessage received, Number is: 53
[INFO] [1669823043.596270]: /listenerMessage received, Number is: 53
[INFO] [1669823043.696917]: /listenerMessage received, Number is: 61
[INFO] [1669823043.796747]: /listenerMessage received, Number is: 27
[INFO] [1669823043.896523]: /listenerMessage received, Number is: 55
[INFO] [1669823043.999622]: /listenerMessage received, Number is: 11
[INFO] [1669823044.098661]: /listenerMessage received, Number is: 79
[INFO] [1669823044.198187]: /listenerMessage received, Number is: 86
[INFO] [1669823044.298700]: /listenerMessage received, Number is: 81
[INFO] [1669823044.396091]: /listenerMessage received, Number is: 13
[INFO] [1669823044.496157]: /listenerMessage received, Number is: 10
[INFO] [1669823044.597191]: /listenerMessage received, Number is: 76
```

Figure 23 - Output of Subscriber.py showing message from Publisher being listened to

7.1.4 LidarScan message reading

To test the capabilities of subscribing to topics, I have decided to transfer this skill towards the F1Tenth simulator, where I created a 'scan_listener.py' program to access the robot's LidarScan readings through subscribing to this topic and output specific readings. The readings chosen would be to display ranges reading of the left, right and front of the car, where for each I would print the desired scan.ranges[] values.

I was required to read up on the topic's messages to see what they hold, as seen in Figure 24. The scan.ranges[] is an array of 1080 elements that each store data readings from different angles of the car, so I had to discover the elements holding the front, left and right readings of the car. At first, I believed that LaserScan in ROS worked in 270 degrees like its physical car's scanner, but I later discovered that the scan in the visualiser works at a 360-degree scan that starts from the bottom right and goes anti-clockwise. With this fact, to discover what the scan for every 90 degrees was, I took the number of elements in scan.ranges[] and divided it by 4 (1080/4) to give me the increment for every 90 degrees. Then I would take the elements for the right (90 degrees), front (180 degrees) and left (270 degrees) and print them out, as seen in Figure 25.



```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data
Header header          # timestamp in the header is the acquisition time of
                        # the first ray in the scan.
                        #
                        # in frame frame_id, angles are measured around
                        # the positive Z axis (counterclockwise, if Z is up)
                        # with zero angle being forward along the x axis
float32 angle_min      # start angle of the scan [rad]
float32 angle_max      # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]
float32 time_increment  # time between measurements [seconds] - if your scanner
                        # is moving, this will be used in interpolating position
                        # of 3d points
float32 scan_time       # time between scans [seconds]
float32 range_min       # minimum range value [m]
float32 range_max       # maximum range value [m]
float32[] ranges        # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities   # Intensity data [device-specific units]. If your
                        # device does not provide intensities, please leave
                        # the array empty.
```

Figure 24 - LidarScan topic messages shown with descriptions. ranges[] message highlighted



Figure 25 - Output of scan_listener.py

7.2 ROS Basic movement control

7.2.1 Auto-drive

The next step to understand the use of the F1Tenth software is a program that allows the robot to move. To make the car move, I am required to publish to AckermannDriveStamped, which uses a message called 'speed' that controls the movement of the car and its speed. By publishing the speed of the car to be more than 0, the car moves forwards.

With this program, I can demonstrate that ROS has the capability to create programs that allow robots to drive around a map that they are placed within. This also further proves the importance of message passing, as the main computation of this program is to publish a message to maintain the robot's constant forward movement to its respective topics used to handle such movement.

7.2.2 Emergency Brake System

With the understanding of how to access and update topics within the F1Tenth environment and ROS, I now expand the concept of basic movement by creating conditions that prevent collisions. As a result, the creation of an emergency braking system is created, whereby the car continues driving forwards until an obstacle in front of the car is detected too close to set threshold, making the car stop moving.

This code was attempted at in two different methods – mathematically and logically. I have decided to take inspiration from the F1Tenth lab sheets [2] to create this the mathematical prototype, which would calculate the time-to-collision (TTC) using the subscribed Odometry and LidarScan topics. From there, the TTC is worked out by taking the coordinates of the car, the reading of the scan to get distance between the two objects and taking the time derivative of that distance as seen in Equation 2.

$$TTC = \frac{r}{[-\dot{r}]_+}$$

Equation 2 - Time-to-collision equation. r = distance between two objects. \dot{r} = time derivative of the distance.

However, I soon encountered a problem with attempting to using the calculation, where the car would be stuck in an endless loop and problems with the car moving in general. This may be because of Python being an interpreted language, making it slower to create calculations for all range values and to react on time to obstacles if necessary.

With this encountered problem, I remodelled the program into the second method, using logic. This was done by subscribing to the LidarScan topic to access the scan ranges and gather the closest distance to an obstacle from the car. With this gathered, I would be publishing to AckermannDriveStamped to either maintain a constant speed of >0 or set speed to 0 (stationary) to simulate the robot braking if the scan range is smaller than the threshold, meaning the car is too close to the wall.

7.3 Car's Localization

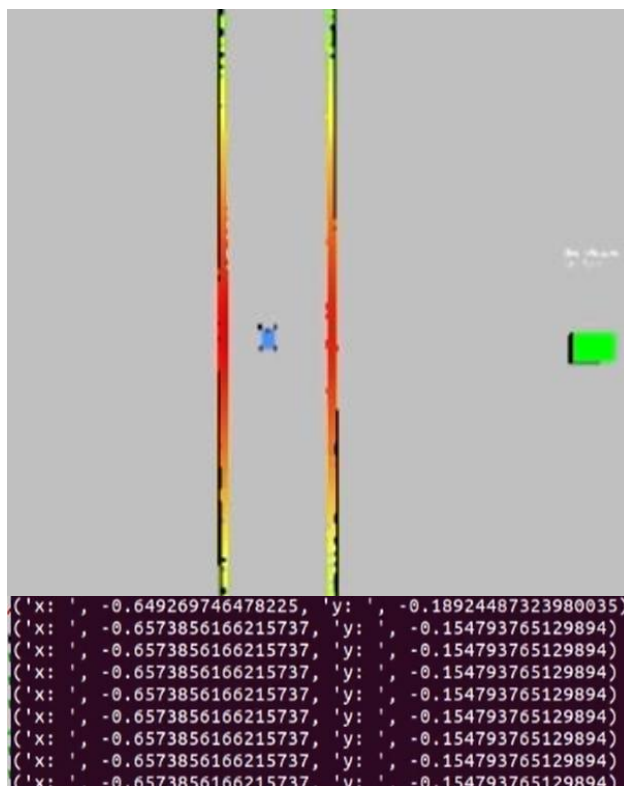


Figure 26 - Localization algorithm showing the x & y-coordinates of the car

I have now made use of the Odometry topic to create a simple localization program that is able to track the car's position on the map, with (0,0) being the initial pose.

Here, the algorithm takes a mathematical approach to determine the car's estimate location on the map. By taking the lidar scanner readings, the program finds the closest obstacle and use it to calculate the angle to that obstacle. This is done by using the scan degree resolution, which takes a scan for every value (per 0.25 degrees) around the car.

Finally, the x and y coordinates of the robot are found by calculating the robot's current position from Odometry topic and using the distance and angle to the closest obstacle.

In Figure 26, it shows the car in its initial position, along with the coordinates being printed out of the car, being around (0,0).

7.4 Wall Following

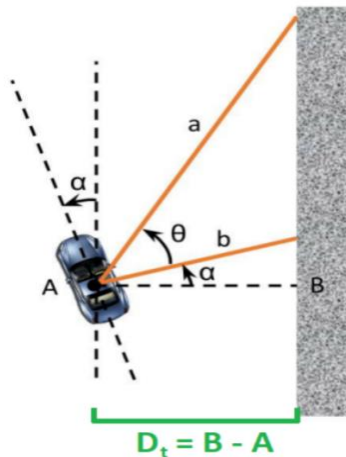


Figure 27 - Distance and orientation of the car relative to the wall [2]

To show the capabilities of a more complex program, a system that would allow the robot to follow a wall without any collision occurring is a must. To make this program, I have come up with two prototypes. The initial prototype was inspired by the F1Tenth lab sheets [2], which is implemented mathematically, while the final implemented version makes use of logic and states. These programs subscribe to LidarScan to gather the scan ranges between the car and wall which is used to determine the car's actions. To allow the car any movement and steering, AckermannDriveStamped is once again used, which offers the speed message alongside with the 'steering_angle' message that allows the car to turn left and right.

7.4.1 Mathematical Approach

This prototype's gathers LidarScan data from the vehicle to identify the distance between the car and the wall, while also navigating the error bound of the car's orientation to be parallel of the wall by gathering its steering angle from the AckermannDriveStamped topic. Figure 27 shows how this algorithm is to work, where two readings are taken to find the alpha angle, which is the error bound that must be fixed by steering the car to its wanted orientation. During implementation, the primary problem faced was that when attempting to use ROS' Time function, it would cause errors which did not allow me to track differences in car's steering and orientation to compare between current and previous time readings. This was the primary cause to switch to the logical approach.

7.4.2 Logical Approach

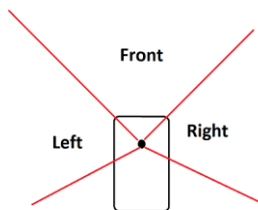


Figure 28 - Scan regions

This prototype has created a series of variables that store required information inside dictionaries. The two dictionaries are for storing states of the car and the lidar scan regions (Figure 28). Within the states dictionary, there are 3 states stored: find a wall, turn left, and follow wall. The scan dictionary stores a list of scans for 3 regions of the car: front, left and right regions of the car, each storing about 70 degrees worth of scans (Figure 28) and taking the closest range scanned, as seen in Figure 29.

```
scan_dict = {
    "right": min(scan_data.ranges[225:434]),
    "front": min(scan_data.ranges[435:644]),
    "left": min(scan_data.ranges[645:854]),}
```

Figure 29 - Dictionary holding scanner values for regions

Afterwards, I create 3 functions for each state, where if in follow wall, the car travels forwards looking for a wall, left turn would steer the car to the left and follow wall once again travels forwards parallel to the wall.

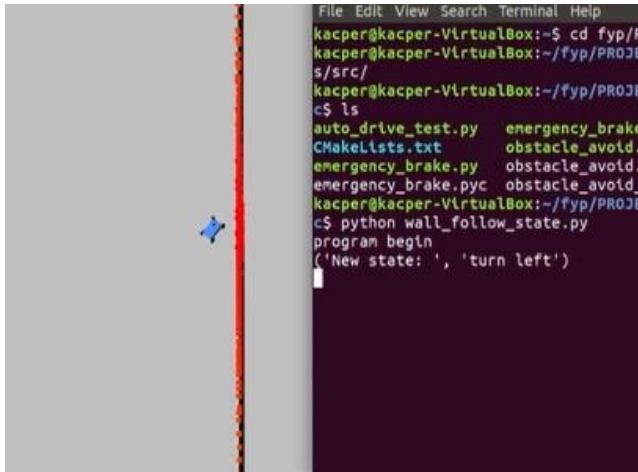


Figure 30 - Wall follower example of turning left

The scan readings are then compared to the threshold set, where there are multiple if-statements that then sets the car to an appropriate state depending on what scenario this car is currently in and running the appropriate drive function. An example of this is in Figure 30, where the program has changed the car's state to 'turn left' because of detecting the wall in front and to its right, which triggers this change of state, turning the car left as a result.

7.5 Obstacle Avoidance

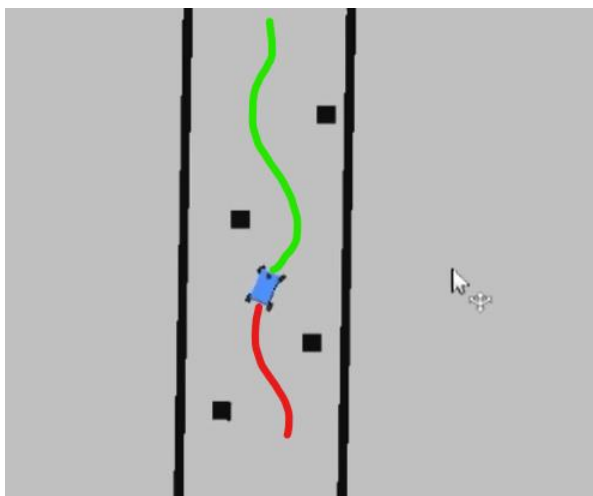


Figure 31 - Example run of obstacle avoidance program

Following the wall follower algorithm, I have continued using the logical approach to implement the program for avoiding obstacles while manoeuvring across the map. Here, I continue using one of the dictionaries – the lidar scan data dictionary, as seen in Figures 28 and 29. I reuse the case system that compares Figure 29 readings to the threshold, and depending on where the obstacle is detected, the appropriate action is taken.

There are 3 functions that each serve as a unique action, drive forward, turn left, or turn right. Within these functions, steering is the main message changed within AckermannDriveStamped, where a right turn is in

the range of $-1 < \text{'steering_angle'} < 0$. Meanwhile, a left turn is $0 < \text{'steering_angle'} < 1$. Figure 31 shows the program in action, where it manoeuvres between the obstacles placed on the track.

7.6 Testing Results

7.6.1 Emergency brake

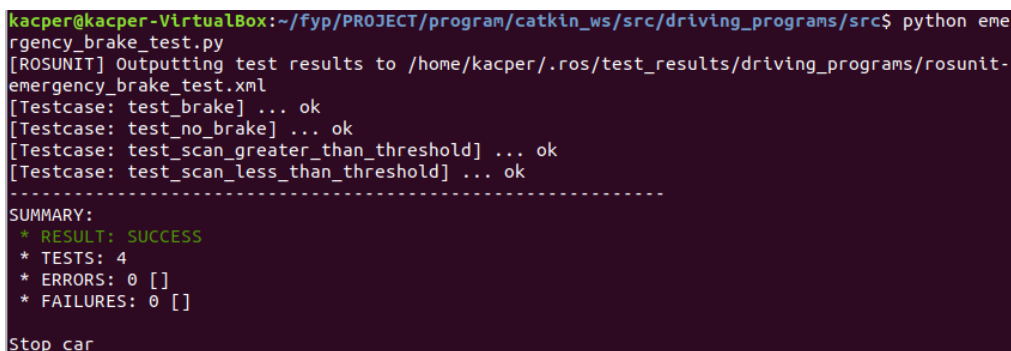


Figure 32 - Emergency brake test outcome

Figure 32 test is conducted on Emergency_brake.py program, testing conditions that the program should and should not stop, while also testing if scan passes the set threshold or not. This is done by making an instance of the program for each test and input certain values into the scan_listener method to simulate ranges in each test. Example run of this test is in Figure 32.

7.6.2 Wall follower

```
kacper@kacper-VirtualBox:~/fyp/PROJECT/program/catkin_ws/src/driving_programs/src$ python wall_follow_test.py
[ROSUNIT] Outputting test results to /home/kacper/.ros/test_results/driving_programs/rosunit-wall_follow_test.xml
[Testcase: test_close_front_left_right_wall] ... ok
[Testcase: test_close_front_left_wall] ... ok
[Testcase: test_close_front_right_wall] ... ok
[Testcase: test_close_left_right_wall] ... ok
[Testcase: test_close_left_wall] ... ok
[Testcase: test_scan_state_find_wall] ... ok
[Testcase: test_scan_state_follow_wall] ... ok
[Testcase: test_scan_state_turn_left] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 8
* ERRORS: 0 []
* FAILURES: 0 []
```

Figure 33 - Wall follower test outcome

Figure 33 test is conducted on Wall_follow_state.py program. Three of the tests created check that the three states (find wall, turn left, follow wall) are able to be accessed correctly. Along these tests, additional ones are made to test the remaining unique conditions (eight in total) separately to see that all of the conditions can be met in their appropriate scenarios. To test these actions, I had to create a variable that would identify each of the conditions uniquely, printing out a specific number for each condition that would be checked in the tests. Example run of this test is in Figure 33. To test these actions, I had to create a variable that would identify each of the conditions uniquely within the wall follower program, printing out a specific number for each condition that would be checked in the tests.

7.6.3 Obstacle avoidance

```
kacper@kacper-VirtualBox:~/fyp/PROJECT/program/catkin_ws/src/driving_programs/src$ python obstacle_avoid_test.py
[ROSUNIT] Outputting test results to /home/kacper/.ros/test_results/driving_programs/rosunit-obstacle_avoid_test.xml
[Testcase: test_close_all_walls_left_turn] ... ok
[Testcase: test_close_all_walls_right_turn] ... ok
[Testcase: test_close_front_left_wall] ... ok
[Testcase: test_close_front_right_wall] ... ok
[Testcase: test_close_front_wall_left_turn] ... ok
[Testcase: test_close_front_wall_right_turn] ... ok
[Testcase: test_close_left_right_wall] ... ok
[Testcase: test_close_left_wall] ... ok
[Testcase: test_close_no_wall] ... ok
[Testcase: test_close_right_wall] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 10
* ERRORS: 0 []
* FAILURES: 0 []
```

Figure 34 - Obstacle avoidance test outcome

Figure 34 test is conducted on Wall_follow_state.py program. Three of the tests created check that the three actions (forward, left, and right) are able to be accessed correctly. To achieve this, since there are specific scanner readings that affect the conditions to enter their required functions, I needed to simulate specific scan results that would trigger those specific conditions. Such as, to enter left turn when scan for front is too small:

```
scan_data.ranges = [1.2] * 580 + [0.4] + [1.3] * 500
```

This puts data on the left and right of car to be above the set threshold while the data for front of car would be below the threshold, triggering the condition to turn left. This was repeated for all the tests, adjusting these values when necessary. This process was repeated to test the remaining unique conditions (eight in total) separately to see that all of the conditions can be met in their appropriate scenarios. To test these actions, I had to create a variable that would identify each of the conditions uniquely within the obstacle avoidance program, printing out a specific number for each condition that would be checked in the tests.

Chapter 8: Professional Issues in the Real-World

This project has been closely related to professional issues found in the real world when working on vehicle technology that may not require human assistance or potentially even replace the need for humans when it comes to different tasks. However, this technology is still relatively new and there are bound to be problems, which may be indirectly caused by poor implementation choices or bugs. Due to the limitation that there still is no well-developed full self-driving (FSD) technology present in the public world, accessibility to such feature to test on a wide scale is impossible at the time.

Currently, Tesla is one of the most advanced companies that are focusing on FSD technology, with currently having developed semi-autonomous vehicle capabilities via the 'Autopilot' mode, offering extensive driver assistance features [16]. these features listed in [16] include:

- Traffic-Aware Cruise Control
- Autosteer
- Auto Lane Change
- Auto park

These features have been operational in most Tesla vehicles for over 4 years, as all cars post April 2019 come standard with 'Autopilot' [16]. However, during early stages of deployment – and to this day – there have been numerous accounts where the software is still limited [17]. One way to state these limitations to the company and other users of the feature has been through social media to document their experiences and to seek answers [17].



Figure 35 - Tesla Autopilot error when no lanes are drawn on the road [17]

In May 2022, it has been stated that more than 750 Tesla owners have faced and reported issues with the FSD [17]. Some of these issues include failure to identify emergency vehicles (police cars, ambulances, etc.), traffic cones in proximity and other issues [17]. These issues, although may not always carry serious problems, would as a result need to have the driver take control over the car to prevent the problem from becoming major.

However, examples such as Figure 35 may prove as major problems, where the car relies on road markings, but the road it is currently on has none.

This results where the car may be caught with an oncoming vehicle driving the opposite way and may risk the driver with a head-on collision if the reaction is not fast enough to take back control. Alongside this problem, other more serious issues reported have been that drivers experienced phantom braking [17]. This could potentially cause accidents if there was a vehicle behind unaware of the sudden brake, especially in environments such as a motorway, where vehicles are driving at high speeds and have much shorter time to react, that may lead to crashes. There have been reports of multiple deaths due to the use of Autopilot, namely with the program not identifying stopped emergency vehicles, leading to fatal crashes [18].

Irresponsible use of autopilot is also a danger to both the driver and other vehicles, where there were numerous reports of drivers filmed using Autopilot without holding the steering wheel or even sitting in the driver's seat [18]. Tesla has stated officially stated that "Before enabling Autopilot, you must

agree to *'keep your hands on the steering wheel at all times'* and to always *'maintain control and responsibility for your car.'*" [16], but somehow users found ways to bypass this regulation [18].

Alongside the 'Autopilot' mode, Tesla also has begun to take this feature a step further to begin implementing more complex features that would require the driver's choice making, with the feature 'Traffic Light and Stop Sign Control'. This allows the car to identify stop signs and traffic lights, decelerating the car to a stop [16]. This component, alongside the 'Autopilot' feature, are still in early stages of development, with the systems not being yet fully operational and may require the driver to take control due to the system still being not fully tested.

Chapter 9: Self-assessment

In this section I will be reflecting on my experience working on this project. I will be discussing the work I have achieved, the skills and knowledge I have gained both in research and putting the skills to practice, and finally talk about areas I could have improved in.

9.1 Achievements

Throughout the project, I had the opportunity to contribute to the development of a self-driving vehicle by utilizing the ROS and F1Tenth libraries. Within the vehicle, I implemented a series of algorithms that provided planning and localization capabilities, all accomplished through ROS. My tasks included emergency braking, obstacle avoidance, and wall following, which required a combination of mathematical and logical components.

Of all the programs I developed, I am particularly proud of the obstacle avoidance algorithm. This program allowed the car to navigate around obstacles without any collisions, quickly selecting the best path forward. It seamlessly cycled between steering left and right or driving forward, as necessary, while keeping a safe distance from the obstacles. Additionally, when the car detected no obstacles nearby, it was programmed to drive in the centre of the track, maintaining a safe distance from the walls and other potential collision points.

In addition to these technical accomplishments, I also took great pride in conducting extensive research on the F1Tenth's physical car and various algorithms used in robotics. This included explaining the importance of the F1Tenth car's components and how they work together with the programs developed to create a fully functional, real-world self-driving car. I also addressed potential issues that could arise in the real world, including those with Tesla's 'Autopilot' technology.

9.2 Skills and Knowledge Gained

Through this project, I have gained practical experience in working with ROS and implementing various algorithms. I have developed a greater understanding of different algorithms, such as identifying the locations of objects within maps. Additionally, I have improved my Python skills, learning about the uses of various libraries, including NumPy, Math, Sys, and rospy libraries. I also delved deeper to understand the benefits of Object-Oriented Programming (OOP), including program reusability and compartmentalisation. By creating programs in OOP form, I was able to expand my knowledge of testing by conducting various unit tests on my algorithms to automate testing and ensure they functioned as intended.

Apart from practical programming skills, I have also learned a great deal about the future of Artificial Intelligence, specifically self-driving capabilities, through research on these topics. I gained valuable insights into how path planning and localization algorithms work in autonomous vehicles, with the help of case studies where they were implemented. This project has allowed me to enhance my research skills and learn how to reference and utilise the information found, which will benefit me when working on future projects.

Lastly, I have gained further knowledge on how to use the Ubuntu operating system, including utilizing its terminal, learning commands, and their functions. This increased familiarity has enabled me to work more efficiently and faster as I progressed through the project.

9.3 Areas of Improvement

Although I am proud of the work I accomplished within this project, there were areas in which I could have improved. For instance, I could have implemented a wider range of automated tests to showcase a greater variety of results, rather than solely utilising unit tests. Additionally, given more time for the project, I would have been able to produce more extensive algorithms to further demonstrate the capabilities of ROS and F1Tenth and how they can be utilised.

However, one primary setback I faced was the limited availability of resources regarding ROS and F1Tenth systems, which led to me spending a considerable amount of time researching the programs and how to utilize the findings. At times, this was not anticipated, and the management of time and resources had to be adjusted accordingly. The F1Tenth software is almost exclusively available on the website [2], with lab sheets and lectures available as guides, but no actual code to understand how certain aspects work.

Another aspect that caused me to struggle during the project was setting up the software. The documentation for setting up Ubuntu, ROS, and configuring the F1Tenth simulator proved difficult to understand, as it required precision in all settings, or else the required software would not run.

Chapter 10: Conclusion

In conclusion, this project has provided a great opportunity to delve into the world of autonomous vehicles and explore the capabilities of ROS and F1Tenth libraries. Through this project, I have gained practical experience in implementing various algorithms to create a self-driving vehicle that requires no human assistance to function, while also improving my skills in Python programming, Object-Oriented Programming, and testing techniques. Additionally, I have expanded my knowledge of autonomous vehicle technology, particularly in path planning and localization algorithms, applying them in multiple ways: a logical or mathematical approach.

Reflecting on the world accomplished in this project, I feel proud of the progress made in the practical experience. Working with ROS to implement a variety of algorithms hands-on has helped me understand how robotics operates to create autonomous vehicles. Through my work, I was able to show how the F1Tenth simulator can be used to achieve the goal of self-driving cars, even without having access to the physical version.

However, there were areas where I could have improved. For instance, I could have implemented more types of automated tests for my programs, to show a variety of results, instead of just using unit tests. If I had further time for this project, I would be able to produce more extensive algorithms to further prove ROS' and F1Tenth's capabilities, with how they could be used. Despite these setbacks, I am satisfied with the progress I made and feel confident in applying my newfound skills in future projects.

10.1 Future work

In future work, this project could be extended by introducing additional challenges, such as making use of other types of sensors or cameras and exploring how the car reacts to unpredictable situations. Additionally, the project could be used to further investigate the ethical and legal implications of autonomous vehicles in society. Finally, if there was more funding available for the project, it could be expanded to simulate the programs written for the simulator with the physical car in the real-world environment.

In terms of career development, this project has provided valuable practical experience that can be utilized in software engineering roles, particularly in the field of robotics and artificial intelligence, or even entering the AI branch itself. The knowledge and skills gained through this project can also be applied to other software development projects, demonstrating higher expertise in programming, testing, and research. Overall, this project has been a significant step towards advancing autonomous vehicle technology and building a foundation for future developments.

Chapter 11: References & Citations

- [1] D. Matine, "What Percentage of Car Accidents Are Caused by Human Error? | Virginia Law Blog", *Buck, Toscano & Tereskerz, Ltd.*, 2022. [Online]. Available: <https://www.bttl.com/what-percentage-of-car-accidents-are-caused-by-human-error/>.
- [2] J. Benson, Ed., "F1Tenth Build - Build Documentation," *f1tenth.org*, 2020. [Online]. Available: <https://f1tenth.org/build.html>. [Accessed: 30-Sep-2022]. F1Tenth Community in the University of Pennsylvania compiled this documentation.
- [3] R. Ping Guan, B. Ristic, L. Wang and R. Evans, *Monte Carlo localisation of a mobile robot using a Doppler–Azimuth radar*, 97th ed. Automatica, 2018, pp. 161-166.
- [4] K.N. McGuire, G.C.H.E. de Croon, K. Tuyls, *A comparative study of bug algorithms for robot navigation*, Robotics and Autonomous Systems – Vol. 121, 2019.
- [5] "Oracle® VM VirtualBox® - User Manual," *virtualbox.org*, 2004. [Online]. Available: <https://www.virtualbox.org/manual/UserManual.html>. [Accessed: 22-Nov-2022]. Sections 1.5 to 1.8 of the documentation are relevant.
- [6] "Ubuntu install of ROS Melodic," *ros.org*, 25-Mar-2020. [Online]. Available: <http://wiki.ros.org/melodic/Installation/Ubuntu>.
- [7] J.-S. Zhao, X. Liu, Z.-J. Feng, and J. S. Dai, "Design of an Ackermann-type steering mechanism," *Proceedings of the Institution of Mechanical Engineers. Part C, Journal of mechanical engineering science*, vol. 227, no. 11, pp. 2549–2562, 2013, doi: 10.1177/0954406213475980.
- [8] Unknown, "Slash 4x4 Platinum: 1/10 scale 4WD Electric Short Course Truck with low CG chassis," *traxxas.com*, 02-Apr-2013. [Online]. Available: <https://traxxas.com/products/models/electric/6804Rslash4x4platinum?t=details>.
- [9] H. Gim et al., "Suitability of Various Lidar and Radar Sensors for Application in Robotics: A Measurable Capability Comparison," *IEEE robotics & automation magazine*, pp. 2–18, 2022, doi: 10.1109/MRA.2022.3188213.
- [10] S. Dehnavi, A. Sedaghatbaf, B. Salmani, M. Sirjani, M. Kargahi, and E. Khamespanah, "Towards an Actor-based Approach to Design Verified ROS-based Robotic Programs using Rebeca," in *The 16th International Conference on Mobile Systems and Pervasive Computing*, vol. 155, 2019, pp. 59–68. <https://doi.org/10.1016/j.procs.2019.08.012>.
- [11] D. L. Akin, "Bug Algorithms and Path Planning", Department of Aerospace Engineering – University of Maryland, 2022.
- [12] S. M. LaVille, "12.3.3 Planning in Unknown Continuous Environments," in *Planning Algorithms*, Cambridge University Press, 2006, pp. 667–673.
- [13] "What is SLAM (simultaneous localization and mapping) – matlab & simulink," *What Is SLAM (Simultaneous Localization and Mapping) – MATLAB & Simulink - MATLAB & Simulink*. [Online]. Available: <https://www.mathworks.com/discovery/slam.html>.
- [14] S. Pramod Thale, M. Mangesh Prabhu, P. Vinod Thakur, and P. Kadam, "Ros based slam implementation for autonomous navigation using Turtlebot," *ITM Web of Conferences*, vol. 32, p. 01011, 2020.
- [15] Cai Luo, A. P. Espinosa, D. Pranantha, and A. De Gloria, "Multi-robot search and Rescue Team," 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics, 2011.

- [16] "Autopilot and full self-driving capability: Tesla Support ," Tesla. [Online]. Available: https://www.tesla.com/en_gb/support/autopilot.
- [17] A. Linja, T. I. Mamun, and S. T. Mueller, "When self-driving fails: Evaluating social media posts regarding problems and misconceptions about Tesla's FSD mode," *Multimodal Technologies and Interaction*, vol. 6, no. 10, p. 86, 2022.
- [18] Isidore, C. "Tesla owners are complaining that their cars are suffering from a significant drop in range after the latest software update". *Business Insider*. <https://www.businessinsider.com/tesla-problems-2019-autopilot-elon-musk-tweets-2019-6?r=US&IR=T>. (2019, 4 June).

Chapter 12: Appendix

12.1 Demonstration of Programs Video Link

Video Demonstration of programs: <https://youtu.be/dtbb2hK6C3U>

12.2 Table of Figures

Figure 1 – Bug-0 algorithm behaviour performed, [4] pg. 1	9
Figure 2 - Ideal Ackermann turning geometry with 4 bar linkages [7]	10
Figure 3 - Traxxas 4x4 Premium Chassis used for F1Tenth car – Traxxas [8].....	12
Figure 4 - Bill of Contents showing F1Tenth components. Main components highlighted [2]	12
Figure 5 - Traxxas EZ-Peak 3S "Completer Pack" Dual Multi-Chemistry Battery Charger w/Two Power Cell Batteries (5000mAh) - Traxxas	13
Figure 6 - Traxxas EZ-Peak 3S "Completer Pack" Dual Multi-Chemistry Battery Charger w/Two Power Cell Batteries (5000mAh) – Traxxas	13
Figure 7 - VESC 6 Mk VI – TRAMPA.	13
Figure 8 - Hokuyo UST-10LX Scanning Laser Rangefinder – UST-10LX.	14
Figure 9 - Lidar UST-10LX Scanning Laser Rangefinder Specifications – [UST-10LX]	14
Figure 10 - Depiction of Data Flow of the F1Tenth Car [2]	15
Figure 11 - Diagram showing possibility to connect to Jetson via WiFi [2]	15
Figure 12 - F1Tenth Simulator heatmap of obstacles [2]	16
Figure 13 - Example of PID control and car navigating to follow wall	16
Figure 14 - F1Tenth simulator launched	18
Figure 15 - Map example that causes a failure in Bug-0 execution [11]	20
Figure 16 - Example map showing execution of Bug-1 with path shown [12]	21
Figure 17 - Example map showing execution of Bug-2 with path shown [12]	21
Figure 18 - Map example that causes a bad case scenario in Bug-1 execution with path shown [12] ..	22
Figure 19 - Map example that causes a bad case scenario in Bug-2 execution with path shown [12] ..	22
Figure 20 - Example of SLAM mapping through localization	23
Figure 21 - Search & Rescue robot team system architecture [15].....	24
Figure 22 - Output of Publisher.py.....	26
Figure 23 - Output of Subscriber.py showing message from Publisher being listened to.....	27
Figure 24 - LidarScan topic messages shown with descriptions. ranges[] message highlighted.....	27
Figure 25 - Output of scan_listener.py	28
Figure 26 - Localization algorithm showing the x & y-coordinates of the car	29

Figure 27 - Distance and orientation of the car relative to the wall [2]	30
Figure 28 - Scan regions	30
Figure 29 - Dictionary holding scanner values for regions.....	30
Figure 30 - Wall follower example of turning left.....	31
Figure 31 - Example run of obstacle avoidance program	31
Figure 32 - Tesla Autopilot error when no lanes are drawn on the road [17]	34

12.3 Code

12.3.1 Turtle_auto.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

#Code inspired by http://wiki.ros.org/turtlesim/Tutorials/Moving%20in%20a%20Straight%20Line

def auto_move():
    #name of node
    rospy.init_node('turtle_move')
    velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)
    vel_msg = Twist()

    #moving only on x axis at speed 1
    vel_msg.linear.x = 1

    #since moving only horizontally, y and z are 0 and no angles at x so also 0
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
    vel_msg.angular.z = 0

    while not rospy.is_shutdown():
        #constantly publish the travel distance of going forward in x axis
        velocity_publisher.publish(vel_msg)

if __name__ == '__main__':
    try:
        auto_move()
    except rospy.ROSInterruptException: pass
```

12.3.2 Publisher.py

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import Int16
import random

#Program inspired by ros.org tutorial http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

#Simple function to publish number to be read by subscriber
def num_publish():
    #Declaring node publishing to 'num_output' topic with message type of 16-bit int
    #queue size limits number of queued msgs if publishing/subscriber is slow
    publish_node = rospy.Publisher('num_output', Int16, queue_size=10)
    #Name of node
    rospy.init_node('publisher_int16')
    rate = rospy.Rate(10) # Loops msgs at 10hz p/s
    #Runs program until ctrl+C to shutdown command
    while not rospy.is_shutdown():
        #Creates a message outputting random number
        msg_int = random.randint(0,99)
        print("Number is: ", msg_int)
        publish_node.publish(msg_int)
        rate.sleep()

if __name__ == '__main__':
    try:
        num_publish()
    except rospy.ROSInterruptException:
        pass
```

12.3.3 Subscriber.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int16

#Program inspired by ros.org tutorial http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

#outputs subscribed message
def terminal_output(data):
    rospy.loginfo(rospy.get_caller_id() + "Message received, Number is: %s", data.data)

#Creates new subscriber node and loops subscriber output
def subscriber():
    #Name of node
    rospy.init_node('listener')
    #declares node subscribing to 'num_output' topic.
    #Will be able to see msgs published from publisher
    rospy.Subscriber("num_output", Int16, terminal_output)
    #Keeps python from exiting until shutdown
    rospy.spin()

if __name__ == '__main__':
    subscriber()
```

12.3.4 Lidar_scan.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan

def callback(data):
    print("Left of robot:", data.ranges[810]) #range value left of car
    print("In front of robot:", data.ranges[540]) #range value ahead of car
    print("Right of robot:", data.ranges[270]) #range value right of car

def lidar_scan_sub():
    #Name of node
    rospy.init_node("lidar_scanning")
    #Subscribe to scan
    rospy.Subscriber("/scan", LaserScan, callback)
    rospy.spin()

if __name__ == '__main__':
    lidar_scan_sub()
```

12.3.5 Auto_drive_test.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped

class Auto_drive_test:

    def __init__(self):
        sub_scan = rospy.Subscriber("scan", LaserScan, self.scan_listener)
        #initialise Subscribers and Publishers
        self.pub_drive = rospy.Publisher("drive", AckermannDriveStamped, queue_size = 1)
        rate = rospy.Rate(500)

    def scan_listener(self, scan_data):
        print("distance to wall: ", scan_data.ranges[360])
        ack_data = AckermannDriveStamped()
        ack_data.drive.speed = 2 #keeps speed of car at 2m/s
        self.pub_drive.publish(ack_data)

if __name__ == '__main__':
    try:
        print("program begin")
        rospy.init_node("brake_car")
        Auto_drive_test()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

12.3.6 Emergency_brake.py

```
#!/usr/bin/env python
import rospy
import unittest
import rostest
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped

class Emergency_brake:

    def __init__(self):
        #initialise Subscribers and Publishers
        sub_scanner = rospy.Subscriber("scan", LaserScan, self.scan_listener)
        self.pub_drive = rospy.Publisher("drive", AckermannDriveStamped, queue_size = 1)
        self.ack_data = AckermannDriveStamped()
        self.threshold = 0.5 #max distance from wall allowed

    def scan_listener(self, scan_data):
        front_scan = min(scan_data.ranges)
        if front_scan < self.threshold: #condition on how close wall is
            self.ack_data.drive.speed = 0.0 #stops car if true
            print("Stop car")
        else:
            self.ack_data.drive.speed = 2.0 #keeps speed of car at 2m/s
        self.pub_drive.publish(self.ack_data) #publishes results

if __name__ == '__main__':
    try:
        print("program begin")
        rospy.init_node("brake_car")
        brake = Emergency_brake()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

12.3.7 Emergency_brake_test.py

```
#!/usr/bin/env python
import emergency_brake
import rospy
import unittest
import rostest
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped

class Emergency_brake_test(unittest.TestCase): #class for unit tests

    def setUp(self): #set up test environment from Emergency_brake class
        self.brake = emergency_brake.Emergency_brake()

    def tearDown(self):
        pass

    def test_scan_less_than_threshold(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges to have these values
        scan_data.ranges = [0.4, 0.8, 1.2, 1.7, 2.3] #0.4 < threshold so program should brake
        self.brake.scan_listener(scan_data) #sends simulated values to Emergency_brake's scan_listener method

        #assertion test
        self.assertLess(min(scan_data.ranges), self.brake.threshold, "Smallest scan_data range not smaller than threshold!")

    def test_scan_greater_than_threshold(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges to have these values
        scan_data.ranges = [0.6, 0.8, 1.2, 1.7, 2.3] #all values > threshold so program should keep car driving
        self.brake.scan_listener(scan_data) #sends simulated values to Emergency_brake's scan_listener method

        #assertion test
        self.assertGreater(min(scan_data.ranges), self.brake.threshold, "Smallest scan_data range not larger than threshold!")

    def test_brake(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges to have these values
        scan_data.ranges = [0.4, 0.8, 1.2, 1.7, 2.3] #0.4 < threshold so program should brake
        self.brake.scan_listener(scan_data) #sends simulated values to Emergency_brake's scan_listener method

        #assertion test
        self.assertLess(min(scan_data.ranges), self.brake.threshold, "Smallest scan_data range not smaller than threshold!")
        self.assertEqual(self.brake.ack_data.drive.speed, 0.0, "Speed != 0.0, car has not stopped!")

    def test_no_brake(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges to have these values
        scan_data.ranges = [0.6, 0.8, 1.2, 1.7, 2.3] #all values > threshold so program should keep car driving
        self.brake.scan_listener(scan_data) #sends simulated values to Emergency_brake's scan_listener method

        #assertion test
        self.assertEqual(self.brake.ack_data.drive.speed, 2.0, "Speed != 2.0, car has different speed/stopped!")
        self.assertGreater(min(scan_data.ranges), self.brake.threshold, "Smallest scan_data range not larger than threshold!")

if __name__ == '__main__':
    rospy.init_node("brake_test")
    rostest.rostest("driving_programs", "emergency_brake_test", Emergency_brake_test)
```

12.3.8 Localization.py

```
#!/usr/bin/env python

import rospy
import math
import numpy as np
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
import sys

class Localization:

    def __init__(self):
        rospy.init_node("localization")
        #initialise subscribers
        sub_scan = rospy.Subscriber('/scan', LaserScan, self.scan_listen)
        sub_pose = rospy.Subscriber('/odom', Odometry, self.pose_listen)
        self.scan_ranges = None
        self.poses = None

    #get scan ranges and store in array
    def scan_listen(self, scan_data):
        self.scan_ranges = np.array(scan_data.ranges)

    #get car pose
    def pose_listen(self, msg):
        self.poses = msg.pose.pose
        self.locate()

    def locate(self):
        # takes closest obstacle range of scan
        closest_idx = np.argmin(self.scan_ranges)

        # calculates angle of closest obstacle
        angle = closest_idx * math.radians(0.25) # takes the radians of the degree difference between each scan

        # calculates x and y positioning of robot
        x = self.poses.position.x + self.scan_ranges[closest_idx] * math.cos(angle)
        y = self.poses.position.y + self.scan_ranges[closest_idx] * math.sin(angle)

        print ('x: ', x, 'y: ', y)

if __name__ == '__main__':
    print("program begin")
    localize = Localization()
    rospy.spin()
```

12.3.9 Wall_follow_pid.py


```
#!/usr/bin/env python

import rospy
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped
import math
import time

#constants
PI = 3.141592653589793
DESIRED_DIST = 0.8

#PID control constants
K_PROP = 1
K_DERIV = 0.1
K_INTEG = 0.5

class Wall_follower:

    prev_error = 0.0
    prev_t = time.time() #previous time - init as current time
    error_t = 0.0 #function e(t) used in equation
    time_delay = 0.0 #time delay between calculation of distance
    integration = 0.0 #used with K_INTEG

    def __init__(self):
        #initialising subs and pubs

        self.sub_scanner = rospy.Subscriber("scan", LaserScan, self.scan_listener)
        self.pub_drive = rospy.Publisher("drive", AckermannDriveStamped, queue_size = 1)
        self.rate = rospy.Rate(500)
        self.ack_data = AckermannDriveStamped()

    def scan_listener(self, scan_data):
        angle_b = 90.0 / 180.0 * PI #getting angle 90 degrees (rad) of car
        angle_a = 45.0 / 180 * PI #getting angle 45 degrees (rad) of car

        #calculating which element from LidarScan ranges[] to use to scan distance to wall for side a & b
        index_a = int(math.floor((angle_a - scan_data.angle_min) / scan_data.angle_increment))
        index_b = int(math.floor((angle_b - scan_data.angle_min) / scan_data.angle_increment))
        print(index_a, index_b)
        range_a = scan_data.ranges[index_a]
        range_b = scan_data.ranges[index_b]

        #using distances a & b to calculate angle alpha between car's x axis and the right wall
        angle_alpha = math.atan((range_a * math.cos(angle_a) - range_b) / range_a * math.sin(angle_a))

        #using alpha angle at current time, now find the current distance (Dt) to wall from car
        dist_t = range_b * math.cos(angle_alpha)

        #using alpha angle at current time, now find the future distance (Dt+1) to wall from car after travelling 1m
        dist_t1 = dist_t + (1.0 * math.cos(angle_alpha))
        # self.pid_control(dist_t1)
        self.test()

    def test(self):
        print("publish")
        self.ack_data.drive.speed = 1.5
        self.pub_drive.publish(self.ack_data)
```

```

def pid_control(self, dist_t1):
    self.error_t = DESIRED_DIST - dist_t1 #calculating error from desired distance
    moment_now = time.time() #getting new current time reading
    self.time_delay = moment_now - self.prev_t
    self.integration = self.integration + (self.prev_error)
    (K_PROP * self.error_t + K_DERIV * (self.error_t - self.prev_error) / self.time_delay + K_INTEG * self.integration)
    self.prev_t = moment_now #sets current time as previous time for future calc
    self.prev_error = self.error_t #sets current error as previous error for future calc
    print("check")
    if abs(self.ack_data.drive.steering_angle) > 20.0 / 180.0 * PI:
        print("speed: 0.5")
        self.ack_data.drive.speed = 0.5
    elif abs(self.ack_data.drive.steering_angle) > 10.0 / 180.0 * PI:
        print("speed: 1.0")
        self.ack_data.drive.speed = 1.0
    else:
        print("speed: 1.5")
        self.ack_data.drive.speed = 1.5
    self.pub_drive.publish(self.ack_data)

if __name__ == '__main__':
    rospy.init_node("wall_follower")
    print("program begin")
    wall_follower()
    rospy.spin()

```

12.3.10 Wall_follow_state.py

```

#!/usr/bin/env python

import rospy
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped
import sys

class Wall_follow:

    #Dictionary that stores the 3 scan data directions
    scan_dict = {
        "right": 0,
        "front": 0,
        "left": 0,}

    #stores current state program is in

    def __init__(self):
        sub = rospy.Subscriber('/scan', LaserScan, self.scan_listen)
        self.pub_drive = rospy.Publisher('/drive', AckermannDriveStamped, queue_size=1)
        self.rate = rospy.Rate(20)
        #Dictionary that stores the 3 states of the program
        self.switch_case = -1 #initialise check with what case was taken up in if-then-elif statements
        self.current_state = 0 #start state at find wall
        self.threshold = 0.8 #Max acceptable distance to wall
        self.state_dict = {
            0: 'find the wall',
            1: 'turn left',
            2: 'follow the wall',}

    #Changes program' state
    def state_change(self, state):
        if state is not self.current_state:
            print("New state: ", self.state_dict[state])
            self.current_state = state

    #Stores scan data in scan dictionary
    def scan_listen(self, scan_data):

        global scan_dict

        #LidarScan works in 360 view starting from rear of car and going anti-clockwise
        scan_dict = {
            "right": min(scan_data.ranges[225:434]),
            "front": min(scan_data.ranges[435:644]),
            "left": min(scan_data.ranges[645:854]),}
        # print(scan_dict)
        self.follow_action()

    def follow_action(self):
        global scan_dict

        #Cases
        #Cases 0 - find wall
        if scan_dict['front'] > self.threshold and scan_dict['left'] > self.threshold and scan_dict['right'] > self.threshold:
            #close to no wall
            self.switch_case = 1
            self.state_change(0)
        elif scan_dict['front'] > self.threshold and scan_dict['left'] < self.threshold and scan_dict['right'] > self.threshold:
            #close to left wall
            self.switch_case = 2
            self.state_change(0)
        elif scan_dict['front'] > self.threshold and scan_dict['left'] < self.threshold and scan_dict['right'] < self.threshold:
            #close to left and right walls
            self.switch_case = 3
            self.state_change(0)

```

```

#Cases 1 - turn left
elif scan_dict['front'] < self.threshold and scan_dict['left'] > self.threshold and scan_dict['right'] > self.threshold:
    #close to wall in front
    self.switch_case = 4
    self.state_change(1)
elif scan_dict['front'] < self.threshold and scan_dict['left'] > self.threshold and scan_dict['right'] < self.threshold:
    #close to front and right walls
    self.switch_case = 5
    self.state_change(1)
elif scan_dict['front'] < self.threshold and scan_dict['left'] < self.threshold and scan_dict['right'] > self.threshold:
    #close to front and left walls
    self.switch_case = 6
    self.state_change(1)
elif scan_dict['front'] < self.threshold and scan_dict['left'] < self.threshold and scan_dict['right'] < self.threshold:
    #close to front, left and right walls
    self.switch_case = 7
    self.state_change(1)
#Cases 2 - follow wall
elif scan_dict['front'] > self.threshold and scan_dict['left'] > self.threshold and scan_dict['right'] < self.threshold:
    #close to right wall
    self.switch_case = 8
    self.state_change(2)
else:
    print("No known state")
    sys.exit()

if self.current_state == 0:
    self.find_wall()
elif self.current_state == 1:
    self.turn_left()
elif self.current_state == 2:
    self.follow_wall()
else:
    rospy.logerr('Unknown state!')

def find_wall(self):
    # print("find wall")
    ack_drive = AckermannDriveStamped()
    # ack_drive.drive.steering_angle = -0.05
    ack_drive.drive.speed = 0.8
    self.pub_drive.publish(ack_drive)

def turn_left(self):
    # print("turn left")
    ack_drive = AckermannDriveStamped()
    ack_drive.drive.speed = 0.5
    ack_drive.drive.steering_angle = 0.8
    self.pub_drive.publish(ack_drive)

def follow_wall(self):
    # print("following")
    ack_drive = AckermannDriveStamped()
    ack_drive.drive.speed = 0.8
    ack_drive.drive.steering_angle = -0.04
    self.pub_drive.publish(ack_drive)

if __name__ == '__main__':
    try:
        print("program begin")
        rospy.init_node("wall_follow_state")
        wall_follow = Wall_follow()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass

```

12.3.11 Wall_follow_test.py

```
#!/usr/bin/env python
import wall_follow_state
import rospy
import unittest
import rostest
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped

'''
This is the dictionary tested,
values in ranges are used to simulate numbers in the scan_data range variables

scan dict = {
    "right": min(scan_data.ranges[225:434]),
    "front": min(scan_data.ranges[435:644]),
    "left": min(scan_data.ranges[645:854]),}'''

class Wall_follow_test(unittest.TestCase): #class for unit tests

    def setUp(self): #set up test environment from Wall follow class
        self.wall_follow = wall_follow_state.Wall_follow()

    def tearDown(self):
        pass

    #test to find wall & test if case 'no wall'
    def test_scan_state_find_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 1080 #adds value 1.2 to all range elements (ranges has 1080 elements in)
        self.wall_follow.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertGreater(min(self.wall_follow.scan_dict), self.wall_follow.threshold, "Smallest scan_data range not smaller than threshold!")
        self.assertEqual(self.wall_follow.current_state, 0, "Current state is not in find wall")
        self.assertEqual(self.wall_follow.switch_case, 1, "not in 'no wall' case")

    #test to turn left & test if case 'wall front'
    def test_scan_state_turn_left(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 580 + [0.4] + [1.2] * 500 #add an element < threshold to be in "front" range and fill rest with values > threshold
        self.wall_follow.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertEqual(self.wall_follow.current_state, 1, "Current state is not in turn left")
        self.assertEqual(self.wall_follow.switch_case, 4, "not in 'wall front' case")

    #test to follow wall & test if case 'wall right'
    def test_scan_state_follow_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 280 + [0.4] + [1.2] * 800 #add an element < threshold to be in "right" range and fill rest with values > threshold
        self.wall_follow.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertEqual(self.wall_follow.current_state, 2, "Current state is not in follow wall")
        self.assertEqual(self.wall_follow.switch_case, 8, "not in 'wall right' case")

    #test if case 'left wall'
    def test_close_left_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 680 + [0.4] + [1.2] * 400 #add an element < threshold to be in "left" range and fill rest with values > threshold
        self.wall_follow.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertEqual(self.wall_follow.current_state, 0, "Current state is not in find wall")
        self.assertEqual(self.wall_follow.switch_case, 2, "not in 'wall left' case")

    #test if case 'left & right walls'
    def test_close_left_right_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 280 + [0.4] + [1.2] * 400 + [0.4] + [1.2] * 400 #add an element < threshold to be in "right" and "left" ranges and fill rest with values > threshold
        self.wall_follow.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertEqual(self.wall_follow.current_state, 0, "Current state is not in find wall")
        self.assertEqual(self.wall_follow.switch_case, 3, "not in 'wall left & right' case")

    #test if case 'front & right walls'
    def test_close_front_right_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 280 + [0.4] + [1.2] * 300 + [0.4] + [1.2] * 500 #add an element < threshold to be in "right" and "front" ranges and fill rest with values > threshold
        self.wall_follow.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertEqual(self.wall_follow.current_state, 1, "Current state is not in find wall")
        self.assertEqual(self.wall_follow.switch_case, 5, "not in 'wall front & right' case")

    #test if case 'front & left walls'
    def test_close_front_left_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 580 + [0.4] + [1.2] * 100 + [0.4] + [1.2] * 400 #add an element < threshold to be in "right" and "left" ranges and fill rest with values > threshold
        self.wall_follow.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertEqual(self.wall_follow.current_state, 1, "Current state is not in find wall")
        self.assertEqual(self.wall_follow.switch_case, 6, "not in 'wall front & left' case")

    #test if case 'front, left & right walls'
    def test_close_front_left_right_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 280 + [0.4] + [1.2] * 200 + [0.4] + [1.2] * 200 + [0.4] + [1.2] * 400 #add an element < threshold to be in "right", "left" and "front" ranges and fill rest with values > threshold
        self.wall_follow.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertEqual(self.wall_follow.current_state, 1, "Current state is not in find wall")
        self.assertEqual(self.wall_follow.switch_case, 7, "not in 'wall front & left & right' case")

if __name__ == '__main__':
    rospy.init_node("wall_follow_test")
    rostest.rostest("driving programs", "wall_follow_test", Wall_follow_test)
```

12.3.12 Obstacle_avoid.py

```

#!/usr/bin/env python

import rospy
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped
import sys

class Obstacle_avoidance:

    #Dictionary that stores the 3 scan data directions
    scan_dict = {
        "right": 0,
        "front": 0,
        "left": 0,}

    def __init__(self):
        sub = rospy.Subscriber('/scan', LaserScan, self.scan_listen)
        self.pub_drive = rospy.Publisher('/drive', AckermannDriveStamped, queue_size=1)
        self.threshold = 0.75 #Max distance acceptable to obstacle
        self.drive_mode = 0
        self.rate = rospy.Rate(20)

    def scan_listen(self, scan_data):
        global scan_dict

        #LidarScan works in 360 view starting from rear of car and going anti-clockwise
        scan_dict = {
            "right": min(scan_data.ranges[225:434]),
            "front": min(scan_data.ranges[435:644]),
            "left": min(scan_data.ranges[645:854]),}
        self.take_action()

    def take_action(self):
        global scan_dict

        if scan_dict['front'] > self.threshold and scan_dict['left'] > self.threshold and scan_dict['right'] > self.threshold:
            #close to no wall
            self.forward()

        elif scan_dict['front'] > self.threshold and scan_dict['left'] < self.threshold and scan_dict['right'] > self.threshold:
            #close to left wall
            self.turn_right()

        elif scan_dict['front'] > self.threshold and scan_dict['left'] < self.threshold and scan_dict['right'] < self.threshold:
            #close to left and right walls
            self.forward()

        elif scan_dict['front'] < self.threshold and scan_dict['left'] > self.threshold and scan_dict['right'] > self.threshold:
            #close to wall in front

            if scan_dict['left'] <= scan_dict['right']: #checks whether to turn left or right
                self.turn_right() #priority to turn right if bigger gap on right
            else:
                self.turn_left() #priority to turn right if bigger gap on right

        elif scan_dict['front'] < self.threshold and scan_dict['left'] > self.threshold and scan_dict['right'] < self.threshold:
            #close to front and right walls
            self.turn_left()

        elif scan_dict['front'] < self.threshold and scan_dict['left'] < self.threshold and scan_dict['right'] > self.threshold:
            #close to front and left walls
            self.turn_right()

        elif scan_dict['front'] < self.threshold and scan_dict['left'] < self.threshold and scan_dict['right'] < self.threshold:
            #close to front, left and right walls

            if scan_dict['left'] <= scan_dict['right']: #checks whether to turn left or right
                self.turn_right() #priority to turn right if bigger gap on right
            else:
                self.turn_left() #priority to turn right if bigger gap on right

        elif scan_dict['front'] > self.threshold and scan_dict['left'] > self.threshold and scan_dict['right'] < self.threshold:
            #close to right wall
            self.turn_left()

        else:
            print("No known state")
            sys.exit()

```

```
def forward(self):
    ack_drive = AckermannDriveStamped()
    self.drive_mode = 1
    ack_drive.drive.speed = 1
    self.pub_drive.publish(ack_drive)

def turn_left(self):
    ack_drive = AckermannDriveStamped()
    self.drive_mode = 2
    ack_drive.drive.speed = 0.4
    ack_drive.drive.steering_angle = 1
    self.pub_drive.publish(ack_drive)

def turn_right(self):
    ack_drive = AckermannDriveStamped()
    self.drive_mode = 3
    ack_drive.drive.speed = 0.4
    ack_drive.drive.steering_angle = -1
    self.pub_drive.publish(ack_drive)

if __name__ == '__main__':
    try:
        print("program begin")
        rospy.init_node("obstacle_avoid")
        obs_avoid = Obstacle_avoidance()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```


12.3.13 Obstacle_avoid_test.py

```

#!/usr/bin/env python
import obstacle_avoid
import rospy
import unittest
import rostest
from sensor_msgs.msg import LaserScan
from ackermann_msgs.msg import AckermannDriveStamped
...
This is the dictionary tested,
values in ranges are used to simulate numbers in the scan data range variables

scan_dict = {
    "right": min(scan_data.ranges[225:434]),
    "front": min(scan_data.ranges[435:644]),
    "left": min(scan_data.ranges[645:854]),}...

class Obstacle_avoid_test(unittest.TestCase): #class for unit tests

    def setUp(self): #set up test environment from Obstacle_avoid class
        self.obs_avoid = obstacle_avoid.Obstacle_avoidance()

    def tearDown(self):
        pass

    #test if case 'no wall'
    def test_close_no_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 1080 #adds value 1.2 to all range elements (ranges has 1080 elements in)
        self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

        #assertion tests
        self.assertGreater(min(self.obs_avoid.scan_dict), self.obs_avoid.threshold, "Smallest scan_data range not smaller than threshold!")
        self.assertEqual(self.obs_avoid.drive_mode, 1, "not in 'no wall' - forward mode")
        # self.assertEqual(self.obs_avoid.switch_case, 1, "not in 'no wall' case")

    #test if case 'left wall'
    def test_close_left_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 680 + [0.4] + [1.2] * 400 #add an element < threshold to be in 'left' range and fill rest with values > threshold
        self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

        #assertion tests
        self.assertEqual(self.obs_avoid.drive_mode, 3, "not in 'left wall' - turn right mode")
        # self.assertEqual(self.obs_avoid.switch_case, 2, "not in 'wall left' case")

    #test if case 'left & right walls'
    def test_close_left_right_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 280 + [0.4] + [1.2] * 400 + [0.4] + [1.2] * 400 #add an element < threshold to be in 'right' and 'left' ranges and fill rest with values > threshold
        self.obs_avoid.scan_listen(scan_data) #sends simulated values to Wall_follow's scan_listener method

        #assertion tests
        self.assertEqual(self.obs_avoid.drive_mode, 1, "not in 'left & right wall' - forward mode")
        # self.assertEqual(self.obs_avoid.switch_case, 3, "not in 'wall left & right' case")

    #test if case 'wall front', with left side having less space
    def test_close_front_wall_left_turn(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 580 + [0.4] + [1.3] * 500 #add an element < threshold to be in "front" range and fill rest with values > threshold
        self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

        #assertion tests
        self.assertEqual(self.obs_avoid.drive_mode, 2, "not in 'front wall' - turn left mode. Right side has less turn space")
        # self.assertEqual(self.obs_avoid.switch_case, 4, "not in 'wall front' case")

    #test if case 'wall front', with right side having less space
    def test_close_front_wall_right_turn(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.3] * 580 + [0.4] + [1.2] * 500 #add an element < threshold to be in "front" range and fill rest with values > threshold
        self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

        #assertion tests
        self.assertEqual(self.obs_avoid.drive_mode, 3, "not in 'front wall' - turn right mode. Left side has less turn space")
        # self.assertEqual(self.obs_avoid.switch_case, 4, "not in 'wall front' case")

    #test if case 'front & right walls'
    def test_close_front_right_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 280 + [0.4] + [1.2] * 380 + [0.4] + [1.2] * 500 #add an element < threshold to be in "right" and "front" ranges and fill rest with values > threshold
        self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

        #assertion tests
        self.assertEqual(self.obs_avoid.drive_mode, 2, "not in 'front & right wall' - turn left mode")
        # self.assertEqual(self.obs_avoid.switch_case, 5, "not in 'wall front & right' case")

    #test if case 'front & left walls'
    def test_close_front_left_wall(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 580 + [0.4] + [1.2] * 100 + [0.4] + [1.2] * 400 #add an element < threshold to be in "right" and "left" ranges and fill rest with values > threshold
        self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

        #assertion tests
        self.assertEqual(self.obs_avoid.drive_mode, 3, "not in 'front & left wall' - turn right mode")
        # self.assertEqual(self.obs_avoid.switch_case, 6, "not in 'wall front & left' case")

    #test if case 'front, left & right walls', with right side having less space
    def test_close_all_walls_left_turn(self):
        scan_data = LaserScan()
        #simulates scan_data.ranges for all regions
        scan_data.ranges = [1.2] * 280 + [0.3] + [1.2] * 280 + [0.4] + [1.2] * 280 + [0.4] + [1.2] * 400 #add an element < threshold to be in "right", "left" and "front" ranges and fill rest with values > threshold
        self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

        #assertion tests
        self.assertEqual(self.obs_avoid.drive_mode, 2, "not in 'front & left wall' - turn left mode. Right side has less turn space")
        # self.assertEqual(self.obs_avoid.switch_case, 7, "not in 'wall front & left & right' case")

```

```
#test if case 'front, left & right walls', with right side having less space
def test_close_all_walls_right_turn(self):
    scan_data = LaserScan()
    #simulates scan data ranges for all regions
    scan_data.ranges = [1.2] * 280 + [0.4] + [1.2] * 280 + [0.4] + [1.2] * 280 + [0.3] + [1.2] * 400 #add an element < threshold to be in "right", "left" and "front" ranges and fill rest with values > threshold
    self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

    #assertion tests
    self.assertEqual(self.obs_avoid.drive_mode, 3, "not in 'front & left wall' - turn right mode, Left side has less turn space")
    # self.assertEqual(self.obs_avoid.switch_case, 7, "not in 'wall front & left & right' case")

#test if case 'wall right'
def test_close_right_wall(self):
    scan_data = LaserScan()
    #simulates scan data ranges for all regions
    scan_data.ranges = [1.2] * 280 + [0.4] + [1.2] * 800 #add an element < threshold to be in "right" range and fill rest with values > threshold
    self.obs_avoid.scan_listen(scan_data) #sends simulated values to obs_avoid's scan_listener method

    #assertion tests
    self.assertEqual(self.obs_avoid.drive_mode, 2, "not in 'right wall' - turn left mode")
    # self.assertEqual(self.obs_avoid.switch_case, 8, "not in 'wall right' case")

if __name__ == '__main__':
    rospy.init_node('obstacle_avoid_test')
    rostopp.run("driving_programs", "obstacle_avoid_test", Obstacle_avoid_test)
```

12.4 Diary

Term 1 Week 03/10/22 - 09/10/22

03/10

Completed overview of project, explained motivation behind doing this and how it can be solved using F1Tenth ROS simulator.

04/10

Created a planned timeline on what I will work on during this project. Described tasks and length of time each will take.

Completed risk assessment on the project and how it will be mitigated.

06/10

First meeting with supervisor, questions answered about where the simulator will be (installed in Ubuntu). Got comments about re-arranging parts of abstract. Comments on re-wording risks to make them more targeted for this project.

Project plan submitted after corrections were made from the meeting

Week 10/10/22 - 16/10/22

Focus was shifted to catching up with other modules on the course.

Some attention put on checking how to set up ROS and F1Tenth simulator.

Week 17/10/22 - 23/10/22

Goals for this week:

Set up ROS on computer (via Ubuntu)

Begin researching and learning documentation of ROS.

17/10

Started with downloading Ubuntu desktop to work with simulator

18/10

Begun reading lecture notes that are accessible on F1Tenth website (<https://f1tenth.org/learn.html>).

23/10

Began writing report on the F1Tenth hardware, sensors and communications. Created reports and programs folders.

Week 24/10/22 - 30/10/22

Goals for this week:

Complete research on hardware, sensors, and communication of the F1Tenth car

25/10

Continued research on hardware of the F1Tenth. Researched Chassis, NVIDIA Jetson NX, Lidar sensor. Found this to be important for the report due to these parts of the car being primary components that make it work.

29/10

Continued hardware report write-up. Completed writing majority of F1Tenth physical car's hardware. Plan to start writing how aspects link to simulator soon. Hopefully can finish report by mid next week to organise meeting with supervisor.

Week 31/9/22 - 06/11/22

Week focusing on courseworks for other modules. Plans to fully initialise Ubuntu.

Week 07/11/22 - 13/11/22

Week focusing on finishing report on hardware, potentially finish set up for F1Tenth simulator and ROS system

07/11

Downloaded Ubuntu. Began setup of ROS-Melodic but encountered problems. Meeting with supervisor on Wednesday will discuss this problem

08/11

Finished Hardware communication with ROS section of hardware report.

Week 14/11/22 - 20/11/22

15/11

Discovered that ROS Melodic works most stable on Ubuntu v18.04. Downloaded this version, ROS, and simulator. Can now start working on writing code.

Week 21/11/22 - 27/11/22

22/11

Began writing configuration and installation of ROS environment and F1Tenth simulator report

23/11

Completed configuration and installation of ROS environment and F1Tenth simulator report

24/11

Saved simulator and files in the repository under 'program' package.

Week 28/11/22 - 04/12/22

29/11

Researched how to use ROS in more detail. Created 2 simple programs for POC on ROS. 'Publisher.py' will 'publish' a random number in the ROS system at a constant rate. 'Subscriber.py' will read and display such information published from 'Publisher.py'

30/11

Interim report created. Added introduction talking about why reasoning behind the project, how I am going to overcome and prove this concept.

01/12

Added aims and objectives of the project. Further research on algorithms (Bug algorithms, Monte Carlo Localisation, Ackermann Steering Geometry).

02/12

Added Installation & Config report to interim report. Added F1Tenth hardware report. Talked about message passing, PID control and how it is used in simulator and physical car.

04/12

Created simple turtlesim robot program to show automatic forward movement.

Week 05/12/22 - 11/12/22

07/12

Talked about proof of concept of message passing (publisher-subscriber), importance in ROS. Talked about turtlesim program. Completed recordings of proof of concept demo.

Term 2: Week 23/01/23 - 29/01/23

23/01

After Winter holidays and getting back into university work/life styles, I have picked back up on the project, starting off with setting up a meeting on 24/01 with supervisor.

Main parts to talk about with supervisor:

- * Interim report grades
- * Feedback on code & report
- * Doubts about how to run code with the simulator
- * Potentially creating own launcher to show ROS working

24/01

After meeting with supervisor, marks have been received. This term's main focus is on creating code to show F1Tenth's capabilities.

Although receiving the green light on working to create a new launcher, I will give one final attempt on trying to run a program with the simulator.

Began writing a program which will attempt to read the LidarScanner readings. These readings are read through a subscription to LidarScan topic, by using the range variable to output left, right and front readings of the robot.

25/01

Completed writing the lidar scan program, added 3 print statements in the subscriber function to read range[0, 360, 719] readings.

Instead of trying to run the program by entering its details into the launcher file, mux, behaviour, etc., I have attempted to run the program by running the simulator, to create a ROS master_node connection. Once done, run the lidar scan program on a separate terminal using `python <file name>` command. This has successfully run the program, with it subscribing to the car's LidarScan and showing readings.

Tested the success by dropping car in different locations to see Lidar readings change.

Messaged supervisor to scrap the new launcher idea.

28/01

After discovering how to run programs with the simulator, the next step is to start creating an emergency brake program. This will cause the car to brake if it reaches too close to the wall.

Following the F1/10 lab sheet, the goal is to subscribe to LidarScan and Odometry topics, while publishing to AckermannDriveStamped (Ackermann). Using LidarScan readings & Odometry's coordinates, we calculate the time to collision (TTC) with wall and publishing a brake to Ackermann topic.

Week 30/01/23 - 05/02/23

30/01

When attempting to calculate the TTC, the program seemed to be stuck in a loop, with the car also not moving at all. This could have been a result of topics overwriting data while trying to scan.

As a result, I have decided to subscribe only to LidarScan and publish to Ackermann, where the car will scan what is in front of itself, and if the scan goes past set threshold, program will publish to Ackermann to brake.

I later have encountered multiple errors where when program is executed, an error occurs where reading topic's variables are not found.

31/01

After multiple attempts to debug the code, I decided to delete it and start from scratch. This has ended in success, where the problem was that when attempting to change topic values for speed, the variables were being declared in the wrong order.

When run, the car will drive forward until reaching near the wall, where car will brake and output a message of "stop car" to confirm that the program does work.

Week 13/02/23 - 19/02/23

13/02

Begun writing code for a wall follower by using the F1Tenth lab sheets for inspiration. The plan is to implement it via use of PID control to calculate the error distance from the followed wall so car drives in parallel to wall at specific distance.

The work will be done by subscribing to LidarScan to get laser scan of car-to-wall distance. Also will be publishing to AckermannDriveStamped to change speed and steering angle of car from scan readings.

15/02

Majority of wall follower code is completed. Use of constants and initialised variables to store data necessary for calculations. Main ones being calculating current and previous time between readings, calculation of cars current and prediction of future location.

With this information, can calculate the distance from the wall and the car's steering angle, with specific conditions for each, changing the speed and steering of the car when needed.

The main issue met today is that ROS time is not working with the initialisation of the node. As a result, I will attempt to change the time from using the ROS library to the Time python library.

16/02

After changing the time from ROS time to Time library, there are no more errors appearing. However, when running the program, it does not seem to publish the change of speed to the car, which as a result causes the car to be forever stationary.

Tomorrow will attempt to completely redo the wall follower program, without PID changes, instead using changing of states depending on what command should be done (eg: follow wall, find wall, etc.).

Plan will be to show use of PID control in a separate program which I will think about in the later stage.

Week 20/02/23 - 26/02/23

20/02

Today took a detour to create a quick and brief proof-of-concept to show how OOP can be used with ROS, by using a class to handle the program. The program itself shows a car that can drive forward automatically, with scanner message showing distance to object ahead.

This is achieved by subscribing to LidarScan to get the range ahead of car, and publishing the speed to AckermannDriveStamped.

With this done, I will now focus on creating a new prototype for wall-follower, using states and OOP class method this week.

26/02

I have now began creating the new prototype to show how a wall-following algorithm would work with the F1Tenth car. This program uses states to cycle between, which are stored in a dictionary. The states are:

* 0 : Find a wall

* 1 : Turn left if wall found

* 2 : Follow wall

The reason why I decided for this approach is that it requires much less calculations, while just performing actions based on the distance of the car to walls. This also would allow the program to be easier to understand as it runs.

Both prototypes subscribe to LidarScan and publish to AckermannDriveStamped.

LidarScan uses a dictionary variable that holds the scans for the left, front and right of the car, at 60 degree ranges and will store the lowest value in each of the ranges (distance closest to wall for each direction).

- * Right [121:280]

- * Front [281:440]

- * Left [441:600]

I have then stated different cases for the car using series of if...then...else statements for each of the measurements. For example:

- * If car's reading for left, front and right values are $>$ threshold, change state to 0 (find wall)

- * If readings for the front of car are $<$ threshold while readings for left & right $>$ threshold, change state to 1 (turn left)

- * If readings for right of car $<$ threshold and while readings for front & left $>$ threshold, change state to 2 (follow wall)

These cases take into account all scenarios the car may be in, and will switch to appropriate case. Once a state is declared, this will then run the appropriate code for that state

- * 0 will drive car forward, looking for wall

- * 1 will turn the car slowly to the left

- * 2 once car is perpendicular to right wall, it stops turning and drives forwards (follows wall)

Currently, the car is not able to publish the Ackermann values, but when the car is moved around the map, depending on how close/far to the wall it is, the states are shown to change through terminal output.

Week 27/02/23 - 05/03/23

28/02

I have found the solution to my problem with no publishing of Ackermann values. This was resolved when I moved the code that calls on functions for state actions from the main function into the action decider function with all cases.

I have also reduced the threshold for scan range from 1 to 0.8

Currently, the car is able to drive to find a wall and turn left if wall is found. However, the car tends to always overturn therefore never going into the follow wall state. It seems that the condition is never entered into.

02/03

After constant trial and error with the scan ranges, I have discovered that I have been using LaserScan ranges wrong, with multiple sources using different scan ranges. For the F1Tenth, it does a full 360 degrees scan, starting at the rear and going anti-clockwise. I discovered that:

- * scan.ranges[0] = 0 degrees from bottom (rear)

- * scan.ranges[270] = 90 degrees (right)

- * scan.ranges[540] = 180 degrees (front)

- * scan.ranges[810] = 270 degrees (left)

- * scan.ranges[1019] = 360 degrees (rear)

With this discovery, I have now changed the range scan dictionary for left, front and right values:

- * right: scan_data.ranges[225:434]

- * front: scan_data.ranges[435:644]

- * left: scan_data.ranges[645:854]

This now causes each area to have a scan range of 70 degrees per region.

I have also adjusted the speed and steering angle to make the program much easier to notice changes, however I may change the speed to be faster when making demo video.

On top of this, with the discovery of LaserScan ranges, I have now returned to my previous programs of lidar_scan and emergency_brake and changed the ranges values to the appropriate ones.

As a result, emergency brake now works correctly and can be visible to stop before touching the wall.

The next plan is to create another prototype for wall follower, which will include a new state that allows the car to follow the left wall, which will as a result keep the car driving within the track at all times without relying on a single wall.

04/03

Created program for the car to avoid obstacles using similar logic to the wall follower program. I have used a case system (if statements) to take appropriate action. However instead of using states, I made the appropriate function to be called within the case. By using measurements:

- * right: scan_data.ranges[225:434]

- * front: scan_data.ranges[435:644]

- * left: scan_data.ranges[645:854]

I was able to create cases where the car will turn left, right or drive forwards depending on what is in front of the car.

When case to drive forward, Ackermann's speed is updated only. When case to turn left, Ackermann's speed is reduced and steering angle is set to 1 to turn left. When case to turn right, Ackermann's speed is also reduced and steering angle is set to -1 to turn right.

As a result, the car is now able to complete a track without touching any walls, turning in time to not crash into a wall, while also centering itself to be in the middle of the track for safety. When adding obstacle points into the track, the car will take appropriate evasive action to avoid the obstacle depending on what the scan read shows. Afterwards it will then re-centre on the track again.

Week 06/03/23 - 12/03/23

06/03

I have now converted my primary files to be classes which will allow me to re-use my code for different instances. I have decided to do this as a result of wanting to follow Software Engineering standards, to make my code of higher quality.

My plan now is to use my wall follower code within the Bug-0 Algorithm program that I will create for path planning. This will be used whenever the car interacts with an obstacle on its way to the goal destination.

11/03

With the programs now being stored inside classes, I have decided to write unit tests for my code to show that when put in certain scenarios, the code should execute in the way that I want it to.

To start off with the test write ups, I have created a separate file to test the emergency_brake file, where I have created a test case for the car using brakes and the car driving when too far from an obstacle.

With these tests, I then connect it with the emergency_brake program by inputting certain values into the scan_listener method that takes the simulated ranges and run the code.

By using rostest and unittest imports, I was able to write assertions and then run the test in a ROS testing environment.

Week 20/03/23- 26/03/23

23/03

I have now continued to make more unit tests for my programs. This time I have tested the Wall_follower.py program.

The tests I have made would mainly check that the 3 states are able to be accessed correctly, one test per state.

Later, I have decided to test the 8 unique conditions separately to see that all 8 states can be accessed and do their appropriate actions.

To test these actions, I had to create a variable that would identify each of the conditions uniquely, printing out a specific number for each condition that would be checked in the tests.

25/03

I have now made a test file for `Obstacle_avoid.py`.

I have made tests that check each of the actions (forward, left, and right) are entered.

To do that, since there are specific scanner readings that affect the conditions to enter their required functions, I needed to simulate specific scan results that would trigger those specific conditions.

Such as, to enter left turn when scan for front is too small:

```
scan_data.ranges = [1.2] * 580 + [0.4] + [1.3] * 500
```

This puts data on the left and right of car to be above threshold while the data for front of car would be below the threshold, triggering the condition.

This was repeated for all the tests.

I have decided to test the 8 unique conditions separately to see that all 8 states can be accessed and do their appropriate actions.

To test these actions, I had to create a variable that would identify each of the conditions uniquely, printing out a specific number for each condition that would be checked in the tests.

Week 27/03/23 - 02/04/23

30/03

I have created an algorithm that locates the cars x and y coordinates, printing them out in terminal. Every time the car is moved, the coordinates change appropriately.

The initial car's pose is coordinates (0,0).

To do this, I had to subscribe to LidarScan and Odometry topics. LidarScan gathers the scan ranges that are used to get the closest distance to an obstacle. Odometry is used to get data of the car's pose, getting position, orientation, etc. .

To get the car position, I use the closest obstacle reading to find the angle of that obstacle, to then use in a calculation to find the x & y positioning.