Robot driving itself into a van

CS39440 Major Project Report

Author: Matt Morgan (mam107@aber.ac.uk)
Supervisor: Dr Fred Labrosse (ffl@aber.ac.uk)

19th April 2019
Version: 0.1 (Draft)

This report was submitted as partial fulfilment of a BSc degree in Artificial Intelligence and Robotics (GH76)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, U.K.

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name Matt Morgan

Date 03/05/2019

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name Matt Morgan

Date 03/05/2019

Acknowledgements

I am grateful to...

I'd like to thank Fred Labrosse for giving guidance during the project and for his time spent installing additional libraries on the robot.

Abstract

A van is used to transport Aberystwyth Universities large robots to and from test sites. These robots usually have to be manually driven into and out of the van for transportation. The Robot driving itself into a van project is an application that can be used to automate and streamline the process of setting up large robots for field trials by loading and unloading robots into and out of the transport van.

This project uses a forward facing LIDAR to guide the robot into and out of the van. This LIDAR senses a specific succession of patterns when driving into or out of the van and these patterns are used to control the robot to drive the correct path into and out of the van, given a roughly similar starting position.

Contents

List of Figures

Chapter 1

Background & Objectives

1.1 Background

When the robot drives into the van, the data the LIDAR captures will be a specific set of patterns. These patterns should be the same each time the robot drives into the van because the van itself is a static environment where not a lot can change. By recording these specific patterns and matching what the robot currently sees to the patterns, the robot should be able to be guided into the van.

There are some interesting research papers reviewing the use of sensors and techniques to get an accurate localisation measure, in particular one titled Mobile Robot Positioning & Sensors and Techniques [1]. In this paper they evaluate different sensors and how they can be used effectively to track a robots position and navigate accordingly. The paper mentions model matching in relation to maps (pre-existing maps and having to build the map when running) and this points to a possible solution to the problem if the map of the area is already known.

Another relevant research paper that was looked at is titled Autonomous Driving Based on Accurate Localization Using Multilayer LiDAR and Dead Reckoning [2] and this paper is about autonomous driving using odometry with the help of multiple LIDAR's to correct the odometry. The issue they tacked is the fact that using odometry on its own leads to unreliable localisation, but combining it with the use of multiple LIDAR's can result in an accurate localisation method. While interesting to see how the two can be combined to create accurate localisation, the vehicle they use this method on is travelling a lot faster than the vehicle that will be driving into the van and therefore this level of accuracy will not be needed.

An original idea on how to tackle the problem of which way to drive was to use computer vision to detect the edges of the van and calculate the middle point in which the robot then drives to. Accurate edge detection would result in 2 specific points being detected as the

edges and the difference in distance between these two points will be able to give an indication of the rotation needed for the robot to be square with the van.

More research into model matching with the use of laserscan data came up with some lecture slides from Berkeley titled Scan Matching Overview [3]. These slides were extremely useful and suggested a good way to match the incoming laser data to the model would be to use an Iterative Closest Point algorithm (ICP). This would result in a translation and rotation that could be used to guide the robot in the correct direction.

The issue with using ICP is the possibility of the algorithm taking a long time to output an answer. This is because the algorithm iterates over itself many times while closing the gap between the model scan and the source scan to get a good result. More research was therefore needed to find out if ICP can be fast enough to process in real time while the robot is moving and a suitable time limit would be 100ms due to the robot moving slowly (0.1m/s). A research paper titled Efficiet Variats of the ICP Algorithm [4] answered if ICP could be fast enough with its section on high speed ICP variants. It suggested that through certain ways, ICP could output a result for a 3D scan of an elephant within 30ms. Due to the data from the LIDAR being a 2D point-plane it should take even less time than a 3D point-cloud and therefore ICP is capable of returning a result within the time required.

I am interested in this project because of my interest in robotics. There were several robotics related projects which I could have chosen but using the big robots that the department has seemed like the most interesting of all of these projects. I am also interested in automation and so the motivation for working on this project was to automate a repetitive task for the department and make a system that could hopefully be used on the large robots the University has for years to come.

1.2 Analysis

Looking at what was learned from the research, the most thought out and likely best way to tackle getting the direction to drive in is using model matching with ICP. This way has been used before in robotic systems and is fast enough to work in real time on a moving vehicle. A program that takes input from the LIDAR, performs ICP model matching on the data against a model and outputs the resulting transformation is therefore needed as an objective of work.

Due to the model matching only working when there is already a model to match with, a recording program is needed as an objective of work which will record the data from the LIDAR while the robot is driven manually. This introduces a configuration phase in which the robot has to be driven into the van manually at least once before it can drive the path manually.

The final part of the program is taking the transformation from the output of the ICP model matching program and using it to guide the robot along the recorded path. This is the last objective needed for a complete system.

Because this project has to run on large robots, testing it first in a simulator before trying with the real hardware is needed. Therefore, a test environment needs to be made with all models made to scale to make the environment as close to how the real world environment.

That brings the list of objectives to: The ICP Model matching program, The recording model program, The robot control program and simulated test environment with models to scale.

1.3 Process

The methodology that was used for this project is closely related to being agile. A sprint lasts 1 week due to the regular meetings with the project supervisor. The sprint lasts from Saturday to Friday because the meeting with the project supervisor is on Friday morning and therefore Friday afternoon is spent re-evaluating what has been achieved in the last sprint and what needs to be achieved within the next sprint.

There are several tools used to keep track of tasks and issues within sprints. A general overview was made using trello which gets updated at the start of every sprint. Any issues that occurred were put into github as tracked issues and this was used to update the tasks listed on trello. The final trello overview can be seen in figure TODO.

Sprints are extremely useful when working with robots due to the ability to keep track of the original plan of things to do but also change and update the plan based on any issues or new priorities that occur.

Chapter 2 explains the original idea for the design before any sprints occurred and chapter 3 explains the final design and what needed changing as each sprint re-evaluated and iterated upon the system's design.

Chapter 2

Original Design

The design should describe what you expected to do, and might also explain areas that you had to revise after some investigation.

Typically, for an object-oriented design, the discussion will focus on the choice of objects and classes and the allocation of methods to classes. The use made of reusable components should be described and their source referenced. Particularly important decisions concerning data structures usually affect the architecture of a system and so should be described here.

How much material you include on detailed design and implementation will depend very much on the nature of the project. It should not be padded out. Think about the significant aspects of your system. For example, describe the design of the user interface if it is a critical aspect of your system, or provide detail about methods and data structures that are not trivial. Do not spend time on long lists of trivial items and repetitive descriptions. If in doubt about what is appropriate, speak to your supervisor.

You should also identify any support tools that you used. You should discuss your choice of implementation tools - programming language, compilers, database management system, program development environment, etc.

Some example sub-sections may be as follows, but the specific sections are for you to define.

2.1 Overall architecture

This project had the limitation of having to use ROS [5] and therefore has to be designed with the different parts of the system as different nodes. Details on node designs and interactions are provided below. The general design is to have 3 separate nodes which interact with each other. These 3 nodes include a recording node which records the path driven by the robot, a model matching node which matches what the robot LIDAR senses in real time with the data which the recording node gathered and a driving node that takes the resulting match and turns it into driving instructions for the robot.

Due to the fact that ICP can be implemented in many ways which affect the speed and accuracy of the match, a pre-existing ICP library is to be used to make the model matching more consistent. Tests were devised and used on a few different ICP libraries to find the best and most customisable library which is explained in the testing section.

Extra helper scripts are needed to drive the robot and manually control which model is being matched with the incoming data. Other scripts such as moving the latest recording to the model file and flipping the model so the robot can drive the same path backwards can be made for convenience.

2.2 Tools

The major tools that will be used for this project include ROS [5] for running the software on the robot, Gazebo [6] for simulating the test environment, MRPT [7] for its ICP library [8], catkin for compilation and TODO: think of more

2.3 Languages

When using ROS, the 2 main languages that can be used are Python and C++. Python is easy to use and quick to get running but may take some time with processing whereas C++ is more difficult to use but is more efficient with its processing. C++ is therefore going to be used for the nodes that needed to run at maximum efficiency such as the ICP model matching and the Automatic Driving node and Python is to be used for all the helper scripts and any early prototyping.

2.4 Detailed node designs and interactions

Here is the original detailed node designed as planned from the start. These nodes and interactions can be visualised in a diagram as shown in figure TODO.

2.4.1 Recording node

The recording node first has to open a file to write the incoming data to. This file is going to be made within a data/ folder of the root directory for the project. The recording node needs to subscribe to a few different topics one being cmd_vel and double_ackermann movement messages that are being sent from any other nodes to control the robot. Due to the need to capture the front LIDAR data, this node also needs to subscribe to the incoming LaserScan data so it is ready to extract the distances when a snapshot is being taken.

This node should take snapshots of data at specific times which the user must specify. It should take a snapshot when the user presses a specific button and adds the extracted LaserScan distances to a new line in the file. Any number of snapshots can be taken and should be taken at critical points along the desired path. When at the end of the path the user should take one last snapshot to mark the end of the path and then close the recording node.

After recording the full path, the robot can be driven back to the original starting position and the automatic driving part of the system can be launched which will automatically move the robot along the recorded path.

2.4.2 ICP compare node

The ICP Compare node first has to load the recorded model from the file that the recording node created. When this happens, the number of lines get counted so the number of models is known and this number is published so any nodes that need this information such as the automated driving node knowing when to stop can easily get it. This node has to subscribe to the incoming LaserScan data as this will be the source data to compare against the model which was recorded. This node also subscribes to a state topic which is used to control the model snapshot that the ICP is using as its comparison data. When the node is started, the state always starts from 0 and gets changed when the subscribed state topic is changed.

When the robot publishes the LaserScan data from the front LIDAR, the distance data gets extracted and used as the source to compare against the model with ICP. The ICP algorithm is implemented using MRPT's 2D ICP library which has been tested to be fast enough to process the data in real time. More information about this testing is in the Testing chapter.

Each time the LaserScan data gets published, ICP is used on the extracted distances and compared with the selected model's distances. The output from the ICP library should be a 3x1 transformation with an x position translating to if the robot needs to move forwards or backwards and how far, a y position translating to if the robot needs to move sideways and how far and finally an angle in which the robot needs to align itself to. These three values will get used for an ICP direction message which will be published so the automated driving node can use the result to guide the robot along the correct path.

2.4.3 Automatic driving node

The automated driving node will subscribe to the ICP direction topic so when the directional data gets published, it can be used to guide the robot in the correct direction. This node should also control which snapshot the ICP comparing node is looking at by publishing the value on the state topic which the ICP node is subscribed to. The state should

always start from 0 which is the first model and starting position for when the model got recorded. This node is also subscribed to the number of states topic which the ICP node publishes the number of lines in the file, and therefore the number of states that exist in the model. This will get used to determine whether the robot is at the final state and can stop or not.

2.5 Messages

There are 3 main messages that the system will make use of. The first is the Laserscan message from the built in ROS sensor_msgs. This message contains a lot of different information but the only part needed for this system are the ranges in which the LIDAR can see and therefore this information gets extracted from the message and either recorded in the recording node or used as the incoming source data for the ICP node.

The second message is a custom message that was made for this system. It will be named ICP_direction and it will contains 3 floats, one for the x distance, one for the y distance and one for the angle. This message will get created every time the source laserdata gets compared to the current model and is sent to the automated driving node to be turned into driving instructions.

The last message is used for controlling the movement of the robbot. There are 2 different messages that can be used, the main one being the standard cmd_vel messages which should work on all robots regardless of steering method. The other message is named double_ackermann and is used for double ackermann steering exclusively which idris uses. Because cmd_vel messages are more general purpose and the fact that this system is meant to work on a variety of robots, cmd_vel messages should be the main way to control robots in this system. Double_ackermann messages can still be used, but they are the less preferred way of using the system.

The structure of all 3 of these messages is shown in figure TODO.

2.6 User configuration

This system requires some set-up before the robot can drive the path automatically. First the robot needs to have a clear starting and ending position. The robot also needs to be placed at the starting position in the orientation best suited to get to the end position as the steering is for correction rather than turning.

Once the robot is in position, the user has to start the recording node and manually drive the robot along the desired path while choosing good places to take snapshots. When the

robot is at the end of its path, the recording node needs to be stopped and the robot needs to be driven back to its starting location. The newly recorded path file needs to be put in the model file that the ICP node will load from and there will be a python script that gets the latest recorded file and uses it as the model.

When the robot is roughly at the same starting position, the automated node can be started and the robot will drive along the last recorded path until its completion where it will stop and display a message stating it is at the final state and close enough to stop.

2.7 Additional scripts

A few additional scripts that help debug the system or help the user automate the process will be used. These scripts include getting the most recent recording and putting the data into the model file ready for launching the automated driving system, flipping the data in the model file so the robot can use the same data to drive the route in reverse, manual control of the robot so the recording part of the system works and manual control of the model states for debugging any issues that occur if a state has been missed.

2.8 Polish

Because this system is designed to work on all robots, the steering needs to be changed depending on the steering method used by each robot. Therefore, as an extra addition, there needs to be different launch files which contain a parameter for each of the different steering methods. This parameter changes how the program steers when reversing so that all robots steer in the correct direction as this is the biggest difference between steering methods.

Chapter 3

Design after iterations and improvements

Due to using a form of the agile methodology, each sprint iterated and improved parts of the design. This chapter goes into depth on what these changes and improvements were and why they needed to happen. Some of these changes will be referred to in the implementation chapter.

3.1 Overall architecture changes

The file format for storing the LaserScan distance data wasn't originally decided. During a sprint a couple of weeks in when the recording node was being made, a decision needed to be made for what format the values should be stored in. The data structure the file needs to store is an array of floats for each snapshot and therefore the format could be quite simple. XML and json were both considered and both rejected due to usually being used for more complex structures even though they both have tools and libraries to parse the data into the program. A much simpler format such as spreadsheets tsv (tab spaced values) or csv (comma spaced values) were considered due to being lightweight and simple to use. In the end csv format is used due to its simplicity of only needing to split the values by comma to retrieve the data and its readability from a human perspective.

3.2 Detailed node designs and interaction changes

Here is the updated detailed node designed as of the last sprint of the project. These nodes and interactions can be visualised in a diagram as shown in figure TODO.

3.2.1 Recording node

The method in which snapshots get taken was changed from relying on user input to being taken automatically depending on the distance travelled. The distance gets calculated when movement messages get received by this node and the distance since the last snapshot is calculated and then added to the distance counter. When the distance counter gets above a set threshold, a snapshot gets taken and stored in a new line in the csv file. This method repeats until the robot has driven the full path manually. When the robot is at the end of its route and the recording can be stopped, one last snapshot is taken before closing the file to make sure the end position is accurate.

This improvement made the whole system more automated as it is no longer waiting on user input. It also made the model more consistent with a set distance between snapshots and should therefore help with guiding the robot along the correct path.

3.2.2 ICP compare node

A useful metric on deciding if the match fit well enough or not is the ”goodness” factor in which MRPT’s ICP library generates for every match. This goodness factor represents how well the source matches the model and if this value is low then it is an indicator that the wrong model is being used. This goodness factor is therefore published with the movement data so the automatic node can use it and react accordingly by changing state if this value gets too low.

3.2.3 Automatic driving node

There were multiple issues with the original automatic driving node plan, one of which was deciding how to convert the directional transformation into robotic movement. There are multiple ways to work out where the robot needs to drive such as using pre-existing path planning algorithms or by putting weights on the y and angle values so that both eventually line up. A path planning library is built into ROS as a [9] but this way seemed unnecessary complex for this system and is usually used for checkpoints that are far away and may have obstacles in the way. I therefore went with a reactionary based system to simplify the design.

With the driving being a reaction based system, the x position is used to determine the speed the robot should be going and when to change state. The closer the robot is from the x position, the slower it should go so there aren’t many anomalies. When the x position reaches a pre-determined threshold, the state can be incremented if the current state isn’t the final one. When the x position is low enough and the model is on the last state, the end condition has been reached and the robot can stop due to being at the end of its route. For steering, if the y position is out by a pre-determined threshold, it gets added to the direction in which the robot needs to steer. If the angle is out by a pre-determined threshold, it also gets added to the direction in which the robot needs to steer. Weights were put onto the y and angle values in such a way that the system priorities getting into the correct position

first and then the correct alignment. Once the speed and direction have been calculated, it gets put into a cmd_vel message and published to the robot which gets guided through the checkpoints until the end state has been reached.

3.3 Message changes

3.3.1 ICP direction message

As mentioned in the ICP compare node, a goodness factor was included into the ICP_direction message as another float and used for recognising when the source and model had a bad match. The updated message format can be seen in figure TODO.

3.4 User configuration changes

Due to the recording node improving on when snapshots are taken, no user input is needed when manually driving the robot along the path for the first time. This is a good improvement because it stops the user from having to multi-task with driving the robot and taking snapshots at the same time. It also streamlines the whole setup of the system as it simplifies what the user has to do before the automated part of the system can run.

Chapter 4

Implementation

4.1 Issues

4.1.1 ICP library

The ICP library from MRPT has options that can change how well the source fit the model and these options needed to be tuned before ICP gave a good result all the time. The example provided in the documentation for the library suggests certain settings to get a good result but these settings weren't outputting a result every time. These settings therefore had to be changed. The issue with changing the settings was that there were only 3 of the 10+ possible settings documented on what they do. The thresholdDist, thresholdAng and corresponding_points_decimation settings are well documented and therefore are easily changeable. Other options from the ICP library had to be fiddled with to figure out what they changed and this took a while. Once it was known what effect the option has on the algorithm, a comment was put along where the option was being set explaining what the option did.

4.1.2 ICP memory

The ICP library that was used was very C like instead of C++ which meant instead being able to use vectors and objects, it used standard types such as float arrays for the model data. This meant that the lasersensor data the robot was sending had to be put into a float array before being used. It also meant the data loaded from the model needed to be put into a multidimensional float array because there were many different models and due to the multidimensional float array needing to be made as the file is being read, this caused a large memory leak. To fix this issue, vectors were used to create the original data structure loaded from the file and when the incoming sensor data needed to be compared, the model for the selected state gets copied into a float array from the vector with all the models in. This new float array is used as the model to compare the incoming source data to.

4.1.3 Reversing on the ramp

There were issues with the robot regularly getting the front end just into the first ramp and then the ICP match for the next 10 or more states was suggesting for the robot to reverse back down the ramp. At first the thought was that the states were inaccurate so the robot was trying to make up for this by reversing to get close enough to the threshold and move the state forwards but when manually controlling the states, the same issue occurred. The same issue occurred when the robot reverses out of the van and therefore this issue points to the difference in pitch when driving up or down the ramp compared to what the model is looking for. This issue only happens in the simulator and not on the real robot and therefore it could also just be a simulator issue or a model issue.

4.1.4 Robot sinking into van model

When testing if the robot could fit into the model of the van in the simulator, the robot would clip through the bottom of the van and get stuck. This was fixed after looking at the model file and changing each links collision max_vel from 0.1 to 0. This change made it so every element in the van model can't move and therefore nothing can clip through it.

4.1.5 Robot sliding off the ramps

When the robot is driving on either of the ramps, the robot tends to slide slightly to the right depending on the speed. The ramps were therefore changed to have the maximum friction and no slip however, this didn't change much. The robot wheels still have a tendency to slip when on the ramps and this is believed to be another simulator issue as simulators can't ever be perfect.

4.1.6 Jerky movement

When running the automated movement node, the speed and steering changed very quickly which meant the robot was unstable and this was creating a lot of noise for the LIDAR sensor. To fix this issue, a PID controller was implemented for both the robot speed and direction so the movement is a smooth curve which gets updated every 300ms. The only part of the PID controller that needed to be implemented was the proportional part and this was set to a value below 1 so that the oscillation is always being reduced. This fixed the issue completely and made the whole system a lot more stable.

4.1.7 ROS controller

The controllers used for idris in gazebosim aren't included in the default ROS installation and therefore needed to be added manually. The specified controllers "position_controllers/JointPositionController" were also unable to be found in any standard ROS packages for the ROS version installed however there were controllers that were commented out named

”effort_controllers/JointEffortController” which could be found for the most up to date ROS version. Installing these and uncommenting them fixed the issue. It is likely this issue is just a ROS version issue with different controllers that work in different versions.

4.2 Easier than expected

4.2.1 Recoding lasersensor data

The recording node had to take snapshots of the lasersensor data at specific intervals. At the start this was just done on a timer which recorded a snapshot every 2 seconds. This wasn't a very good way of deciding when to take snapshots because the robot could be stationary during the 2 seconds and then the snapshots would be extremely similar. The solution to this was to take snapshots based on the distance travelled. This was easier to implement than expected because only the timer needed to be modified and a distance variable added. The calculation for tracking distance is the time taken since last movement message sent multiplied by the speed in meters per second of the last movement message. The result gets added to a distance variable and when this distance variable goes over a specified value in meters, a snapshot is taken. This worked first time which is why it was easier than expected.

4.2.2 Additional Scripts

The small helper scripts that were written in Python were fast to be made and started off as prototyping however these scripts worked extremely well and due to them only being helper script, they didn't need to be converted into a more efficient language. These scripts improved the quality of life of the system so that actions which were done manually got converted into easy to use programs. An example of this is the ”change_model _to_latest.py” script which targeted the most recently edited file in the data directory and copied the contents of this file to the model.csv file that the system uses. All of these extra Python scripts were simple to implement and easy to use.

4.3 Harder than expected

4.3.1 CMake

Learning and using CMake was a lot harder than expected. The most difficult part is trying to debug library configuration errors, especially as CMake outputs hardly any useful information in its error messages. After reading a lot of CMake documentation and stackoverflow threads, a solution was found for the library configuration issues but several weeks was spent on this issue. A similar experience occurred when installing the required libraries on idris in which at least a whole day was spent trying to configure CMake to install and find the correct libraries.

4.3.2 Changing states

Keeping an accurate track of which state the ICP should be comparing against was more of a difficult task than anticipated. This issue was amplified by the reversing on the ramp issue because when checked manually, the next state which made the robot move in the correct direction on the ramps was at least 10 states in front of the current state. This was made more accurate by including the ”goodness” measure to make sure the source and the model were at least similar and giving a good enough output. More additions to the code were made to help advance the state if the robot had been previously moving in one direction and suddenly wanted to change direction. This addition incremented the state every time the robot wanted to change direction and improved the accuracy of the states if the state is inaccurate. If the state was accurate, the robot would eventually catch up with the advanced state by just moving towards the new goal and updating the state when this goal had been reached.

4.3.3 Steering

The steering was an extremely difficult problem due to the likelihood of the y direction wanting to make the steering go a certain direction and the angle wanting to go the opposite direction. This was difficult to implement due to the steering method needing to prioritise the y direction before the angle. This way, the robot can get into the correct position before straightening up and aligning itself to the correct angle. A lot of fiddling with the steering values happened and a couple of weeks were spent on perfecting it. When tried on the real robot, there were still some issues with the robot over steering which had to be tuned in such a way that the movement is smooth.

4.4 Limitations

4.4.1 Starting angle

The starting angle is very important in whether the system can successfully drive the path or not. The issue is that ICP needs a good enough starting position to be able to match correctly. After thorough testing, it seems the maximum angle is roughly 10 degrees off centre for the system be successful. Anything more than 10 degrees off centre and resulting match’s ”goodness” factor decreases below the set threshold of 70%.

4.4.2 Filtered laserscan

The incoming laserscan data gets filtered so that it only looks at the middle 80 degrees and only up to 10 meters. This is because the van is likely to be closer than 10 meters and directly in front of the robot. This was implemented to help the ICP matching find a clear match and make sensor recordings more generalised so the same recording could be used multiple times in a mostly static environment. This change however, contributed to

the starting angle limitation because it limited the amount of data the matching algorithm had to get a clear match when not completely aligned.

4.5 Review against the requirements

TODO

Chapter 5

Testing

5.1 Overall Approach to Testing

In general, robots that work with sensors are difficult to reliably test due to factors such as sensor noise, unpredictable behaviours and errors which can throw the whole system off. Due to this system being fully reactive, any inaccurate sensor readings have the possibility of throwing the system into error states in which it has to try to recover from. These possible error states have to be thoroughly tested to make sure the recovery is safe and reliable. All testing was done manually through either observing robot behaviours when the robot was driving automatically or by controlling the robot manually and observing the desired robot movement.

Video recordings of most of the following tests can be viewed in the tests folder within the project. All of these recordings are taken of the final working system and do not represent the system as it was when the issues occurred.

5.2 Testing

5.2.1 ICP algorithm speed

3 individual ICP libraries were tested for how fast they could run and give an answer on the same set of data. These 3 libraries were MRPT-ICP, libicp and libtopointmatcher. Figure TODO shows the results of each algorithm. The results ended up being fairly similar in speed and therefore any of the three libraries could be used for the system in processing data in real time. The decision to use MRPT came down to its ease of use and extra information that it returns.

5.2.2 Simulated environment testing

Multiple test environments were created in a simulator, the first being a maze of walls which can be driven through by the robot. These walls were also used as reference points for further testing and simple box shaped objects were added to make the different areas more unique for matching.

A scale model of the van was created in the simulator which consisted of 2 same sized ramps and a box shaped container for the robot to drive into. Measurements were taken of the real van so the simulated van was an accurate scale model. Figure TODO shows the measurements taken for each part of the van and figure TODO shows the scale model van in the gazebo simulator.

Due to the van having hydraulics control the height, measurements were taken of the lowest possible height and also the highest possible height so that the simulated van's height could be the middle of the two. The minimum height is 44 cm from the ground and the maximum height is 57 cm from the ground therefore, the simulated van was created with a height of 50 cm off the ground which is near the middle. Different height van models were planned but not created due to the time constraint and because other issues were a priority.

5.2.3 Manual environment testing

The first part that needed testing in the simulated environment was that the robot could use the ramps from the van model to drive fully into the van without any issues. As talked about in section 3, there were issues with the robot clipping through the van's interior floor and also issue with the robot sliding off the van's ramps. These were glaring issues highlighted when first testing the van model by trying to drive the robot manually into the van. Other tests were done such as pausing the simulator and using the simulator tools to pick the robot up and move it into the van in such a way that when unpaused, the robot is fully in the van at its likely end location. This was a good first test to do because it highlighted a lot of issues with the original van model which could be fixed.

The recording node needed to be tested to make sure the data it was recording is accurate data. This was done by using the boundary walls of the maze world at one of the corners. The robot was set to record all data while slowly driving towards the outside corner of the maze. This way the data was predictable in that half of the laserscan data should be values which decrease and the other half should be infinite values. This test showed that the recording node was working properly as the values matched the prediction.

Another test was done to make sure these values were accurate by using a blank wall of the outside of the maze. The robot was set to slowly drive towards the wall with the

data being recorded and therefore the data should show all values starting at large numbers and decreasing with every snapshot. The resulting data showed exactly what was predicted and was therefore working.

5.2.4 Automated node testing

5.2.4.1 Driving towards a wall

The automated driving node was first tested by using a wall in the maze to drive towards. To make this easier for the model matching algorithm, a large cylinder was placed just in front of the wall with two cubes placed either side of the cylinder. The robot was set to record and manually driven up to just in front of the cylinder to gather data for the model. The robot was then placed in roughly the same starting position and orientation and the automated driving launch file was started so that the robot behaviour could be observed.

The ideal behaviour was for the robot to follow the path that was just recorded perfectly from start to finish and stop when the robot was close to the cylinder. This was mainly for testing the steering correction as little changes in steering could cause big ripple effects with the robot over compensating if the steering oscillation isn't being reduced. This test was usually successful but it did highlight an issue with both the y direction and the angle needing to be reversed to get the correct direction.

After doing this test from start to finish, the model was flipped to test if the robot could complete the same path in reverse. This way of testing the robot first driving towards the wall and then using the same model to drive away from the wall was repeated multiple times to test how reliable the system is. It usually ended up with the robot slowly getting more misaligned after 3 to 4 times driving the path forwards and then in reverse. Changes were made each time until the steering could recover from these misalignments.

5.2.4.2 Driving into the van

The automated driving node was also regularly tested at how well it could drive into the van model. This was tested by recording the robot driving into the van manually and then reloading the world so the robot was in the exact same position that it started from. Many issues were highlighted from running this test such as how the robot reliably reverses down the ramps for about half a meter before continuing back up the ramp into the van.

While driving the incorrect way on the ramp, it was noted that the robot still tries to correct itself by steering towards the centre of the ramps like it should do. The only anomaly found in the data is the distance to the next checkpoint being incorrect. This was determined to be the robot seeing different distances to the back of the van to what it should have seen due to the ramps making the LIDAR get data from a different angle of pitch.

5.2.4.3 Driving out of the van

Driving out of the van was usually tested after the robot was successful when driving into the van. Occasionally this required a manual adjustment so the robot was aligned to correctly reverse out of the van as the steering doesn't correct very quickly and is likely to collide with the van walls. Reversing out of the van was always a harder challenge for the system due to only using the front LIDAR which effectively made the robot blind to whatever was behind it.

Many adjustments to the steering and state tracking were made after this test as the robot regularly failed by either slipping off the final ramp, which was likely a simulation issue, or by driving back into the van just after the first ramp. The robot driving back into the van issue was likely due to a combination of the ICP matching at different pitches and the state being slightly behind.

5.2.5 Stress testing

Stress testing was done mostly by observing the maximum angle in which the robot could be placed at and still recover from. Different starting positions were also tested to make sure the system could recognise when the robot needs to reposition.

5.2.5.1 Driving towards a wall at different angles

The first stress test that was made used the maze level with the cylinder and two cubes. The normal recording in which the robot is placed face on to the cylinder and made to drive straight towards it was made and then the robot was reversed and put at increasingly misaligned angles. With the angles that were roughly under 5 degrees, the system could recover and straighten up to go directly towards the cylinder. With angles above 5 degrees, the robot had occasionally recovered and drove to the cylinder but if it didn't recover, it would drive towards the side of one of the boxes. With any angle above 10 degrees, the robot wouldn't recover and instead just drive towards the side of the box it was oriented towards. It likely did this due to the field of view on the LIDAR being limited to 80 degrees and therefore could only detect 2 of the 3 objects. It did however reliably get guided towards the side of one of the cubes. This suggested the match was still trying to work because the side of the cube gives mostly the same values the side of a cylinder gives.

5.2.5.2 Driving through the maze to test the maximum steering

The maze level was also used for the purpose it was made, to test how sharp of a turn the robot can take. The robot was placed somewhere in the middle of the maze just before a u-turn and was recorded while driving manually around the corner. The robot was then placed in roughly the same position and observed trying to make the u-turn automatically. The robot never successfully automatically drove around the u-turn but this test did give

some valuable insight into how much the system can deal with sharp turns. The robot would usually get half-way around the turn and then fail to recover. This suggests the system can deal with roughly 90 degree turns if the environment is well-defined but not 180 degree turns. Due to the system only needing to deal with small misalignments, this should be sufficient steering to drive into the van.

This test also highlighted if any of the steering components were making the robot drive in the wrong direction and was very useful in fixing these issues.

5.2.5.3 Driving into the van at different starting positions and angles

The van model was used extensively to test different starting positions and angles. A recording of the robot driving straight into the van was taken and then the robot was placed at different starting positions in front or behind the original starting position. The robot was also tested by being placed at varying angles to the ramp of the van to see if the system could recover from certain starting positions. The system always seemed to recover from being in front or behind the original starting position. If the starting angle was below 5 degrees, the system would always be able to recover and drive up the ramp. If the starting angle was above 10 degrees misalignment, the system would never be able to recover and would fail to drive into the van. This seems to suggest a limitation with the model matching in which the robot needs to be in a starting position which isn't too far off the original recording to be successful in driving its set path.

5.2.5.4 Driving out of the van at different angles

Driving back out of the van from different angles was aimed at testing how fast the system could cope with misalignments. Due to the walls of the van being less than a meter away from the robot, the system would have to change direction quickly to avoid colliding with them. The system regularly failed this test if the angle was too large due to the steering being fully reactive and needing time to correct itself. This adds to the limitations of the system needing some space to correct itself but should not be much of an issue due to the robot needing to be correctly aligned at the back of the van for it to be tied to the van and not roll around in transport.

5.2.6 Real world robot testing

A few tests were done with the real robot first around the robot shed and then driving into the real van.

5.2.6.1 Driving around the robot shed

The first tests were to make sure the system could deal with driving correct paths in the real world. The robot was recorded driving up to a bin which was placed just in front of a

wall to make model matching easier. The robot was then recorded driving in a straight line up to the bin and then driven back to roughly the same starting position. The real robot was set to automatically drive the recorded path and it successfully drove the correct path up to the bin and stopped. This test was done multiple times to make sure the system is reliable and it completed the path successfully each time.

The second test using the same data was to see if the system could deal with positions starting in front and behind the original starting position. This test was also done multiple times to make sure it was reliable and the system successfully dealt with the different distances each time.

With different distances tested, the last part that needed testing was the different starting angles. This test was ran multiple times at different angles just to be thorough. As with the same test in the simulator, with low angles, the system usually is able to recover and with large angles, the system usually fails to recover. Therefore, the starting position for the robot must be mostly aligned for the system to work.

5.2.6.2 Driving into the van

The last test was getting the robot to automatically drive into the van which is the original purpose of the project. The robot was set up just before the ramps and aligned correctly before being set to record and manually being driven into the van. The robot was then reversed out of the van and the automatic driving system launched to test if the robot could drive into the van automatically. The robot then successfully drove into the van and stopped at the end of the recorded path. It corrected for small errors which would have made the robot fall off the ramp but finished slightly angled to the right. The same test was ran again for reliability and the exact same result happened with the robot ending slightly angled to the right. This angling is likely due to the model recording noisy data at the end state and this therefore guided the robot to steer to the right at the end.

Chapter 6

Evaluation

6.1 Requirements

The requirements for this system were correctly identified at the beginning. The system needed to be able to automatically drive into the van and in doing so, a specific set of patterns can be detected with use of the LIDAR.

6.2 Design

The design was thought out and planned accordingly. Some changes were needed to the overall design in the amount of nodes and the node interactions.

The major changes that need to be made to make this system better if started again would be to use the ICP model matching algorithm as a service that can be called on rather than having it continuously running and sending the resulting transformation. By using it as a service, the whole system can be controlled by the automated driving node which would result in a hybrid control system which could choose if the robot needed to act upon updated information or continue with the path originally planned.

The hardest part of this project was translating the result from the ICP algorithm into movement which guided the robot along the correct path while recovering from any errors that might make it move slightly off course. An easier way of doing this would be to use pre-existing path planning tools to plan ahead so the system could be more robust.

The recording node was well thought out from the start and needed minimal changes to work as intended. When it was realised that the best way to record the data would be at specific distances rather than user defined due to the ease of use and reliability, it improved the overall workflow of the system drastically.

6.3 Use of tools

The tools that were used were suitable for this project however, a few extra tools could have been used to make the system more reliable and to help debug the issues that occurred.

6.3.1 ROS path planning

Possible tools that would have helped include using the ROS navigation stack to plan a path to the target location. This would have saved a lot of time in trying to get the robot steering to match the direction that the ICP algorithm was returning. It also would have resulted in a likely better coping system with misalignment and bad positioning due to turning the steering into a deliberative system which can react to errors rather than a fully reactive system.

6.3.2 Visualisation of the data

A tool that would have also helped would have been some sort of tool to visualise the data that the recording node took. This would have been a helpful debugging tool when figuring out certain behaviours instead of looking at the raw numbers. A tool such as rvis [10], which is built into ROS, could help with such visualisations as it can take the data produced by the laserscan in real time and visualise it against the model.

6.4 Testing

The testing for this system has been thorough. None of the testing is automated but this is difficult to accomplish when working with a fully reactive robotic system. Tests were always ran multiple times to make sure the result is always reliable.

6.5 Project aims

This project set out to get Aberystwyth universities large robots to be able to automatically load themselves into a van. The final part of testing proves that the system that has been created fulfils this aim. There are however some limitations which this system has which need to be known before using it.

These limitations are that the robot must be mostly aligned with the ramp before starting, the robot must be mostly in the right position before starting and the robot must not be too far away from the ramp before starting. If all three of these limitations are met then the system will work.

Annotated Bibliography

[1] J. Borenstein, H. R. Everett, L. Feng, and D. Wehe, “Mobile Robot Positioning & Sensors and Techniques,” vol. 14, no. 4.

Overview and review of sensors and localisation techniques

[2] N. Akai, Y. Morales, T. Yamaguchi, E. Takeuchi, Y. Yoshihara, H. Okuda, T. Suzuki, and Y. Ninomiya, “Autonomous Driving Based on Accurate Localization Using Multilayer LiDAR and Dead Reckoning,” 2017.

Using Multilayer LIDAR to improve Relative Position Measurments (dead-reckoning) AKA. odometry

[3] P. Abbeel, “Scan Matching Overview,” pp. 1–26. [Online]. Available: https://people.eecs.berkeley.edu/{~}pabbeel/cs287-fa11/slides/scan-matching.pdf

First ideas on how to match pointclouds, ICP algorithm first looked at

[4] M. Levoy, “Efficiet Variats of the ICP Algorithm.pdf.”

Fast ICP matching with 2D point planes. Main idea comes from this paper

[5] A. Blasdel, A. Leeper, A. Hendrix, C. Rockey, D. Coleman, and D. Hershberger, “ROS (robot operating system).” [Online]. Available: https://wiki.ros.org/

[6] A. Howard, N. Koenig, and J. Hsu, “Gazebo simulator.” [Online]. Available: http://gazebosim.org/

[7] J.-L. Blanco-Claraco and N. Koukis, “MRPT (Mobile Robot Programming Toolkit),” 2019. [Online]. Available: https://www.mrpt.org/

[8] J. L. Blanco, “MRPT Iterative Closest Point (ICP) and other matching algorithms,” 2013. [Online]. Available: https://www.mrpt.org/Iterative{_}Closest{_}Point{_}{%}28ICP{%}29{_}and{_}other{_}matching{_}algorithms

MRPT (Mobile Robot Programming Toolkit) ICP algorithm. Basically a tutorial on how to use it.

[9] D. V. Lu, M. Ferguson, and A. Hoy, “ROS Path Planner.” [Online]. Available: https://wiki.ros.org/global{_}planner

[10] D. Hershberger, D. Gossow, and J. Faust, “RVis 3D visualisation software.” [Online]. Available: https://wiki.ros.org/rviz

Appendices

Appendix A

Third-Party Code and Libraries

Appendix B

Ethics Submission

Appendix C

Code Examples

For some projects, it might be relevant to include some code extracts in an appendix. You are not expected to put all of your code here - the correct place for all of your code is in the technical submission that is made in addition to the Project Report. However, if there are some notable aspects of the code that you discuss, including that in an appendix might be useful to make it easier for your readers to access.

As a general guide, if you are discussing short extracts of code then you are advised to include such code in the body of the report. If there is a longer extract that is relevant, then you might include it as shown in the following section.

Only include code in the appendix if that code is discussed and referred to in the body of the report.

3.1 Random Number Generator

The Bayes Durham Shuffle ensures that the psuedo random numbers used in the simulation are further shuffled, ensuring minimal correlation between subsequent random outputs [**?**].

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
```

```
#define RNMX (1.0 - EPS)

double ran2(long *idum)
{
  /*-------------------------------------------------*/
  /* Minimum Standard Random Number Generator        */
  /* Taken from Numerical recipies in C              */
  /* Based on Park and Miller with Bays Durham Shuffle */
  /* Coupled Schrage methods for extra periodicity   */
  /* Always call with negative number to initialise  */
  /*-------------------------------------------------*/

  int j;
  long k;
  static long idum2=123456789;
  static long iy=0;
  static long iv[NTAB];
  double temp;

  if (*idum <=0)
  {
    if (-(*idum) < 1)
    {
      *idum = 1;
    }else
    {
      *idum = -(*idum);
    }
    idum2=(*idum);
    for (j=NTAB+7;j>=0;j--)
    {
      k = (*idum)/IQ1;
      *idum = IA1 *(*idum-k*IQ1) - IR1*k;
      if (*idum < 0)
      {
        *idum += IM1;
      }
      if (j < NTAB)
      {
        iv[j] = *idum;
      }
    }
    iy = iv[0];
  }
  k = (*idum)/IQ1;
  *idum = IA1*(*idum-k*IQ1) - IR1*k;
  if (*idum < 0)
  {
```

```
    *idum += IM1;
  }
  k = (idum2)/IQ2;
  idum2 = IA2*(idum2-k*IQ2) - IR2*k;
  if (idum2 < 0)
  {
    idum2 += IM2;
  }
  j = iy/NDIV;
  iy=iv[j] - idum2;
  iv[j] = *idum;
  if (iy < 1)
  {
    iy += IMM1;
  }
  if ((temp=AM*iy) > RNMX)
  {
    return RNMX;
  }else
  {
    return temp;
  }
}
```