Robotic Navigation: Automatic floor planning

by

Damion Shillinglaw

Dan Simons

Dec 8, 2023

Table of Contents

1 Introduction

1.1 Literature Review

Accurate localization is a fundamental component of autonomous robotic navigation. The ability for a robot to discern where it is on a given map is critical for its ability to navigate its environment. Indeed, available maps are not always as accurate as one might expect. Typical architectural floor plans contain all of the necessary information about the location and orientation of walls, but do not contain any information about the placement of objects within the environment. Cluttered environments affect the performance of localization algorithms, such as adaptive monte carlo localization (AMCL) [1]. Instead of just localizing within these cluttered environments, we propose using simultaneous localization and mapping (SLAM) techniques to refine the supplied map to include the clutter should lower the localization error and improve the overall performance of the algorithm.

Various open source libraries have been developed to implement SLAM including Steve Macenski's "slam toolbox". Slam toolbox is a graph based implementation of SLAM which is built on a modified version of open karto [1]. It was designed for non-expert technicians to be able to map large areas in a relatively quick manner. In contrast, some other SLAM libraries such as GMapping are very difficult to get quality results and require a specific methodology to generate good maps [1]. The ease of use of Slam Toolbox is valuable when used in an autonomous system. Slam toolbox is the current default SLAM library in ROS2 [1] and contains a breadth of functionality useful for this problem. Most notably is the "lifelong mapping" function. This function allows the loading of a saved map and removal of inaccurate scans from previous sessions. Lifelong mapping, with some adjustments, would allow us to load the supplied architectural floor plan map as the previously generated map and start a new mapping session to add the clutter that was previously missing.

On the hardware and systems the project F1TENTH autonomous vehicle system is an open source which utilizes a traxxas slash rc car [2]. The platform includes ROS2 packages for remote control of the vehicle and publishing of sensor data. The F1TENTH community also hosts races where teams compete to develop the fastest autonomous vehicle. One such team is the F1TENTH_WS team out of the University of Waterloo [3]. Navigation methodologies from these projects can be borrowed from. The team from Waterloo uses the Pure pursuit algorithm to steer the vehicle in a manner that will follow a series of waypoints along the planned path.

Another similar project is the TurtleBot 4 from Clearpath Robotics [4]. The TurtleBot 4 is a ROS2 integrated mobile robot. It was designed for educational purposes to allow students without a background in electronics to get hands-on experience with autonomous vehicles. Unlike our project, TurtleBot 4 is a differential drive system, meaning that it can spin in one location compared to a typical car-like system which has a limited turn radius. TurtleBot's navigation system can be modified to create waypoints for use in the pure pursuit algorithm.

1.2 Goals

The goal of this project is to create a robot, named MacNav, that can autonomously navigate from a predefined starting and endpoint using MacEwan second, floor plan maps.

2 Methodology

From figure one the general design for the project consists of two separate docker containers running on both the bot and a remote laptop. Within these containers ROS nodes are communicating over the network using cyclone DDS middleware. The nodes in container one are responsible for handling sensor data while another node handles commands related to the robots movement. Moreover, in container two the nodes handle navigation tasks and provide visual displays. In the event the robot is stuck or performs risky maneuvers, such as driving over a hole,then this is mitigated by manual control via Xbox 360 controller.

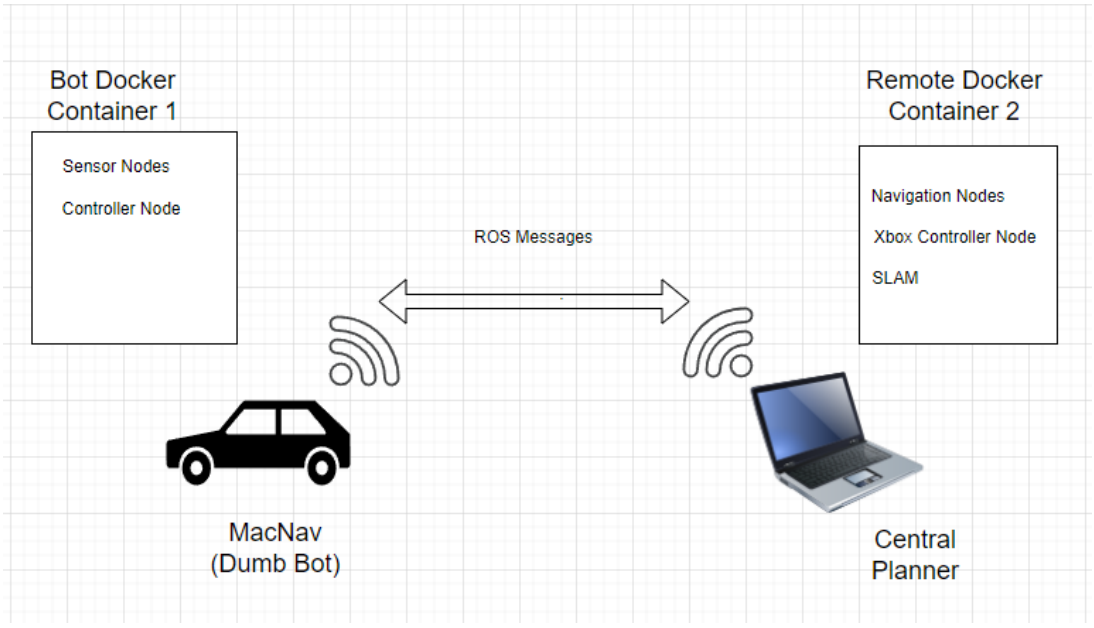


Figure 1: Bot's Architecture

The dumb bot label given to the robot in figure one refers to a simple bot that only sends available sensor data and awaits commands for movement, while the heavy calculations and navigation algorithms are handled by the central planner (robot). In addition, the planner is also responsible for running the GUIs for visualizing maps, tracking robots progress, and specifying waypoints.

2.2 Hardware

The hardware involved a rc-car, two batteries, three Arduino Unos, a Jetson Nano, motor controller paired with a 12V DC motor, and three sensors used in determining odometry.

2.2.1 Microcontrollers and MacNav's Computer

	Device Name	Voltage	Current
Arduino	Uno	2.6V - 12V	500mA
Motor Controller	L298N Dual H-Bridge	3.2V - 40V	2A
Jetson	Nano Development Kit	5V - 5.25V	4A

Table 1**:** Controllers

The main computer on the bot is the Jetson Nano which is responsible for taking sensor data from two Arduino Unos and sending velocity data to another Uno. Essentially, Nano handles all incoming and outgoing data between Arduinos and remote planer. From table one, it is noted that the Nano has a max current draw of 4A, but this is for the power jack; instead the micro usb port was used since the power bank (table three) satisfies the recommended 2.5A for supplying this port.

Three Arduinos were used since each was given separate tasks: writing encoder data, writing IMU's heading data, and waiting on velocity data. Velocity data was used to interpret inputs for the actuators found in the drive system. Serial communication was utilized between Arduinos and Nano.

2.2.2 Sensors and Actuators

Two sensors were required for odometry, a LIDAR scanner and motor encoder,

while another IMU sensor was used to help improve its orientation. From table 4, the bot was able to sample many points in its immediate environment, which is used to determine the bot's positioning. In order to determine the bot's speed, encoder counts were read from a Dongguan E1ZY-E1-6 motor, which was chosen for its high RPM, given that torque is not needed for the indoor terrain– as slippery floors are not prevalent, mainly carpet surfaces. More details on power requirements for the sensors and actuators is in table four and five, as well as specifying other capabilities.

	Device Name	Voltage	Current	Range	Sample Rate
Lidar Scanner	SLAMTEC A1M8	5V - 5.5V	600mA	12 meters	8000Hz
IMU	BNO055	3.3V - 5.0V	12.3mA		1Hz

Table 2**:** Sensors

	Device Name	Voltage	Current	RPM (no load)	Pulses Per Revolution
DC Motor	Tubular Motor w/ encoder	12V	0.5A	4000	11
Servo	Power HD 1812MG	4.8V - 6V	TBD		
LIDAR motor	SLAMTEC A1M8	5V - 9V	TBD		

Table 3**:** Motors

2.2.3 Power

MacNav is powered by two batteries. First, a 12V Lipo, 3s battery supplies power to the drive arduino, motor controller, and steering servo. Second, a power bank supplies power to the Jetson Nano, both sensor Arduinos, and laser scanner. The 3s LIPO was chosen to handle the higher current draw from the 12V DC motor driving the Rover, while the power bank supplies the Nano with the recommended voltage. Moreover, the higher mAh on the power bank enables longer usage of the Nano, especially convenient when wanting to develop and test out in the open away from any nearby power outlets. These battery specs are summarized in table 3.

	Voltage	mAh	C-Rating	Max Draw
12S LIPO	12V	3800	45	171A
Power Bank	5V	26800		2.5A

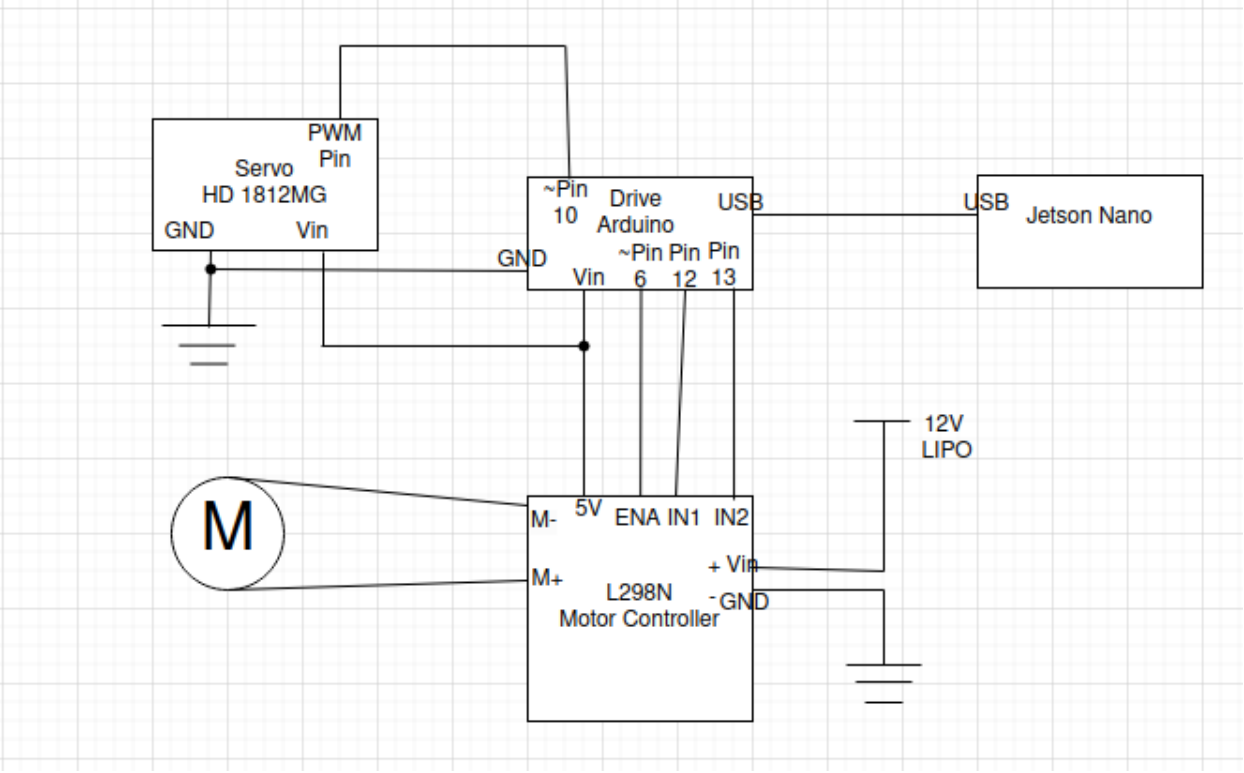
Table 4: Batteries

2.2.4 Chassis

All of these components are mounted on a modified Tamiya Plasma Edge II 1/10th scale 4WD RC buggy. 3D printed platforms were added to the buggy to mount the required hardware. For example, a red platform that is approximately the size of the base chassis, supports the Jetson Nano, both Arduinos, and the motor controller. Two smaller sub-platforms, one on top of the main red platform is blue and holds the laser scanner. The other sub-platform is suspended from the main platform that holds the power bank. The base chassis holds the steering servo, drive motor, and 12V lipo battery. On the rear of the buggy two antennas are placed to allow for wireless connection between MacNav and remote computers.

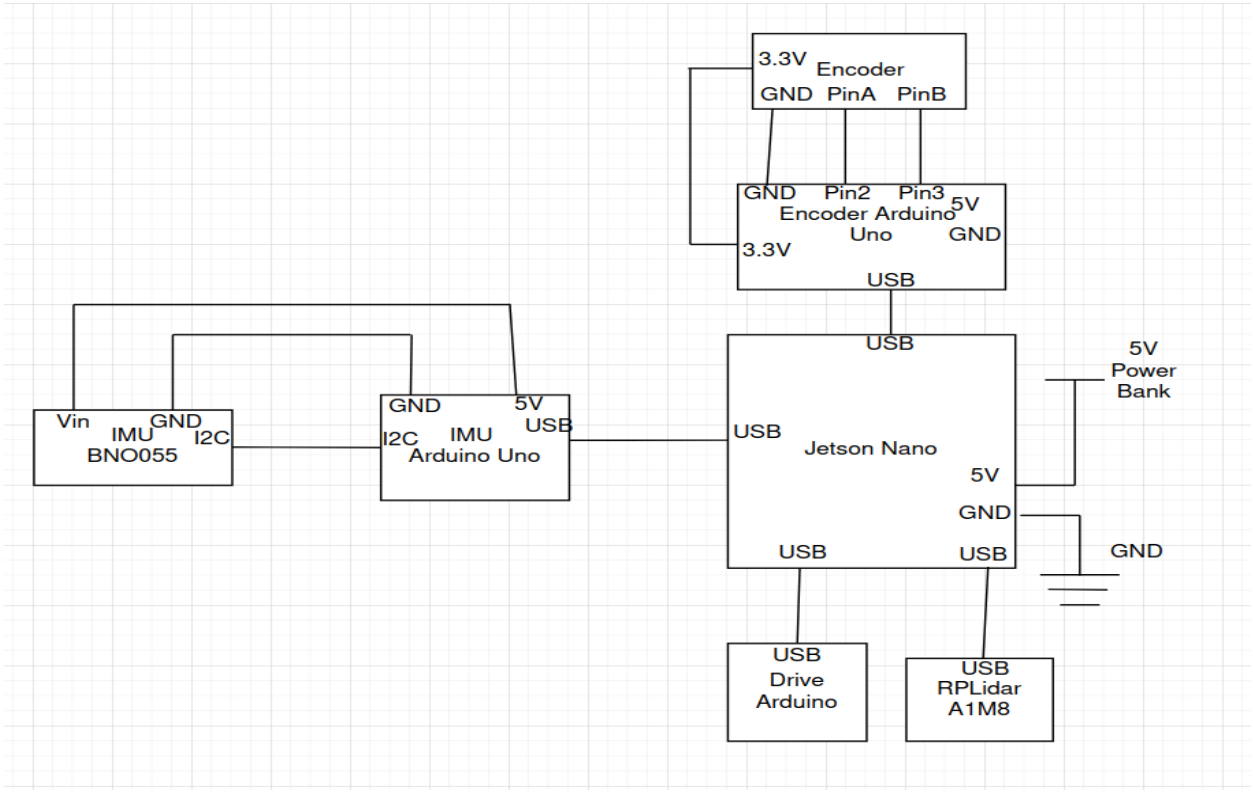
2.2.5 Schematics

Ending the hardware section with schematics, below are two schematics showing the drive system and the setup for communication between controllers and sensors.



Schematic 1: Drive System

Schematic one displays the drive system that is controlled by the Arduino drive. Incoming data representing linear and angular velocities is used to determine the inputs for the steering servo and motor.



Schematic 2: Sensors, Jetson Nano, and Arduinos

Finally, schematic two provides an overview of where sensor data is read and passed onto the Nano. Mainly, exchange of data was done by serial communication.

2.3 Software

The software section is split into navigation and software setup.

2.3.1 Navigation

In order to navigate through the environment, the Rover needs access to a map of the environment, a method of localizing itself within said maps, a way to plan a path and a method of detecting and avoiding obstacles not listed on the map.

2.3.1.1 Mapping

A static map of the environment is generated by manually driving the robot through the environment while running SLAM. The chosen SLAM library is "SLAM_Toolbox," which implements a graph-based SLAM algorithm. Graph-based SLAM algorithms construct a graph consisting of nodes representing positions the robot has been and landmarks the laser scans observed. The graph also includes edges connecting these nodes, representing constraints derived from sensor data, specifically, the LIDAR scanner and movement data

from an IMU and wheel encoders. As the Rover continues to move, it accumulates these nodes and edges until it revisits a landmark it has encountered before. Upon reaching the landmark, it establishes a constraint that loops back to a previously generated node, causing the entire graph to adjust its positions to reconcile the constraints. This process is known as loop closure and is crucial for correcting errors in sensor readings. As more loop closures occur, the graph becomes more accurate in representing the real-world environment.

2.3.1.2 Localization

Localizing the Rover within the static SLAM map requires the ability to dead reckon and make observations about the environment around it. Dead reckoning is accomplished with a combination of the data from the wheel encoders and the heading provided by the IMU sensor. The wheel encoders provide the linear distance the Rover has traveled while the IMU determines the direction of travel. When heading data is received the Rover calculates the average heading the rover was facing since the last time data was collected by dividing the difference between the two angles by two and adding it to the previous heading. It then adjusts the current position by the distance it traveled in that average calculated direction. This method assumes that the Rover was turning at a consistent rate over each interval. Since the sampling rate of the IMU is relatively low (1Hz) this is rarely the case. To combat this, at a more frequent interval the Rover calculates its heading from the steer commands it received. Due to play in the steering assembly these calculations are more prone to drift than the IMU so a combination of both methods are used.

Localization is then accomplished with Adaptive Monte Carlo Localization (AMCL). AMCL is a particle filter algorithm, which involves randomly distributing potential positions, called particles, throughout the map and iteratively filtering out positions that do not align with the sensor data. The algorithm begins with an initial pose estimation, where an approximate initial position of the robot is provided, and particles are scattered around that location. Each particle is assigned a likelihood, known as its weight, which is adjusted as the robot starts to move. After a predefined time period, the algorithm re-samples the particles. The probability of a particle being placed in a particular position is increased based on its proximity to particles with a high weight in the previous iteration. The "adaptive" aspect of the algorithm means that as iterations progress and particles start to converge on a single location, the algorithm will reduce the number of particles sampled in each iteration to conserve computational resources.

2.3.1.3 Pathfinding

In addition to localizing the Rover, it requires pathfinding. Here, a variant of the A* algorithm called hybrid A* is used. This algorithm aims to create a smooth path that doesn’t contain too tight of turn that the ackermann steering of the Rover is incapable of completing. The static map of the environment is processed to create a costmap of the environment to pass to hybrid A*. The values in this map are set such that the cells with a closer proximity to obstacles have a higher cost which gradually decreases as distance to the obstacle increases. This is to discourage the Rover from navigating too close to the obstacles and

potentially turning into them as it goes around a corner. This algorithm was proposed by Dolgov, Thrun, Montemerlo, and Diebel in [5] and implemented by Steve Macenski in the default ROS navigation package nav2 [6].

Once the path is created the Rover needs to be able to follow it. Path following is accomplished using the regulated pure pursuit algorithm, which was also implemented by Steve Mackenski in nav2 [6]. A typical pure pursuit algorithm finds the furthest point along a given path within a given lookahead radius of the Rover and steers the Rover to face that point. Regulated pure pursuit expands on this by varying the lookahead distance as the speed of the rover increases. It also slows the Rover as it drives close to obstacles and as it rounds corners in case there is something around the corner that would require the Rover to stop. Furthermore, it implements a collision detection system that checks where the rover will be in a certain amount of time based on the current velocity and detects if that will result in a collision. This collision detection only detects potential collisions, it doesn't recalculate the path to avoid the obstacle.

2.3.1.4 Obstacle avoidance

Obstacles detected by the LIDAR scanner are added to the costmap used by the pathfinding algorithm. The pathfinder replans the path with these new obstacles addressed and drives around them.

2.3.2 Basis for software setup

Robot Operating System 2 (ROS2) is a framework that utilizes nodes which are abstract representations of various components found in robotic systems. Moreover, these nodes can represent sensors, microcontrollers, actuators, data processing units, and so forth. Communication is standardized by using ROS2 messages which are sent over topics by publishers and received by subscribers of each individual topic. The ease of representing components in a robotic system with the concept of nodes and providing a way of standardizing communication between them enables developers to work on higher level functions for the robot which is why this framework was chosen.

In order to utilize ROS2 on the Jetson Nano, Docker containers were used. For this project, the version of ROS2 used was "humble". The setup for the navigation system required building two separate docker containers on the Jetson Nano and remote machine. Moreover, to allow for efficient and timely communication between both machines on the same network, this required setting up middleware called cyclone DDS that comes with the ROS2 framework.

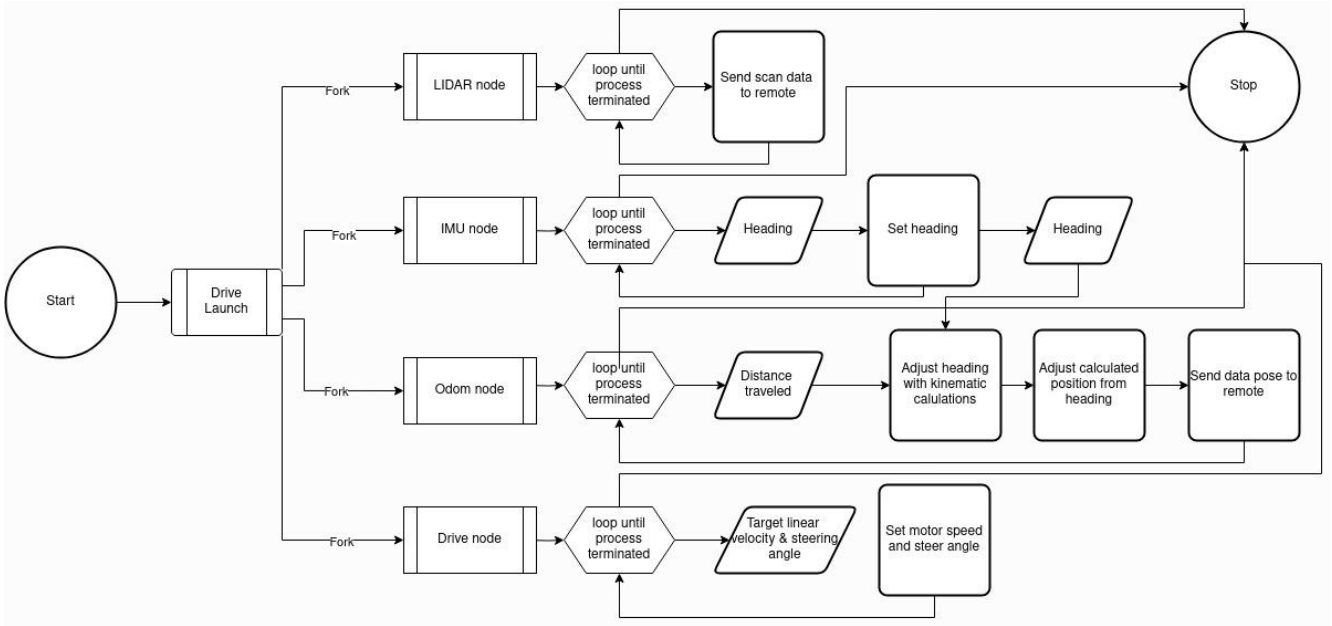


Figure 2: ROS2 flowchart for software running on the Rover.

ROS includes a launch system where a launch file defines any number of executables to run with just one function call on the command line. The Jetson Nano is connected to from the remote machine via ssh so nodes related to hardware directly on the Rover can be launched. The flowchart of this is shown in Fig. 2. It launches three nodes which handle the data collected from the sensors and one node which sends velocity and steering commands to the dc motor and steering servo. The first node is responsible for processing the data received from the LIDAR. This node was created by the manufacturer of the sensor and publishes an “laserscan” message on the “/scan” topic. A laserscan message contains some metadata about the data collected such as the angle between each data point and the maximum and minimum range of the sensor. It also contains an array of the ranges recorded along the path which can be used to determine the local environment around the Rover. The next node is the IMU node which connects to an arduino via serial communication that transmits heading values from a gyrometer. Initially, a magnetometer was used for this heading detection that uses the earth’s magnetic field to determine heading. However, magnetic fields present within the building cause large errors in these readings that cause incorrect readings. The gyrometer is prone to a small amount of drift over extended periods of time but is largely correct in its readings. The drawback of this particular sensor is that it can only be polled for the current heading at a rate of approximately 1Hz. This can cause errors in the pose calculations depicted in Fig. 3. If Rover travels along the black path but the heading is only checked at the blue marks then the calculated direction of travel will be along the red line resulting in the calculated pose being significantly off of the true pose. This problem is solved in the third node, the odometry node. The odometry node listens to the data collected from the encoder and the steering commands issued to the rover. It calculates the change in heading using the steering angle of the front tires multiplied by the linear distance travelled divided by the wheelbase and adjusts the stored heading value by that amount. Calculating the odometry by this method alone causes error over time due to play in the connection

between the servo and the wheels so every time the IMU receives data it sets the heading directly to that value. This combination of sensors minimizes error in odometry calculations to provide the most accurate estimate of the Rover's pose possible. The final node launch on the Rover is the drive node. This node is responsible for listening to velocity and steering commands supplied by the remote and interfacing with the hardware to achieve these commands.

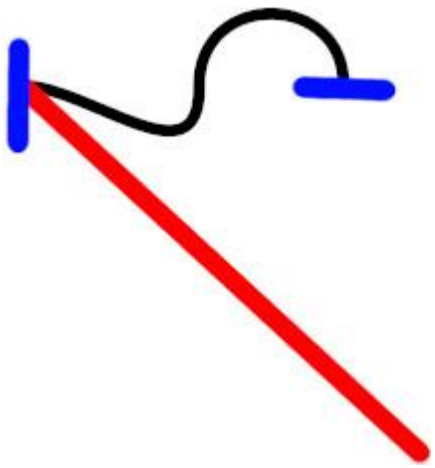


Figure 3: Potential issue with heading detection rate

While these nodes are running on the Rover, nodes related to processing the data and sending movement commands are run on the remote machine. Fig. 4 shows the launch file used for using slam to map the environment. The first two nodes publish the description of the Rover and the state of its joints. The "robot_state_publisher" node takes in a URDF file that describes the Rover. The URDF is a xml file that contains information about the physical components of the Rover and their position relative to each other. The most important relationship is the one between the "base_link" and "laser_frame". The base link is the position between the rear wheels of the Rover that it rotates around as it drives and the laser frame is the location of the lidar sensor. This relationship is crucial for identifying where LIDAR scans are relative to the actual position of the Rover in the world. Next, Slam toolbox is launched in online asynchronous mode. "Online" means that the incoming scans are coming from a live source as opposed to pre-recorded data being played back and "asynchronous" means that the newest data in the buffer should be processed first. Processing every single scan isn't as important as the maps can still be constructed with some data points missing but keeping up with the data as it comes in is important to ensure that map generation keeps up with the Rover. Lastly, Rviz2 is a visualization tool that allows the map to be checked as the Rover drives around. This node is not vital for the actual driving of the rover but is included in the launch file to aid in map creation.

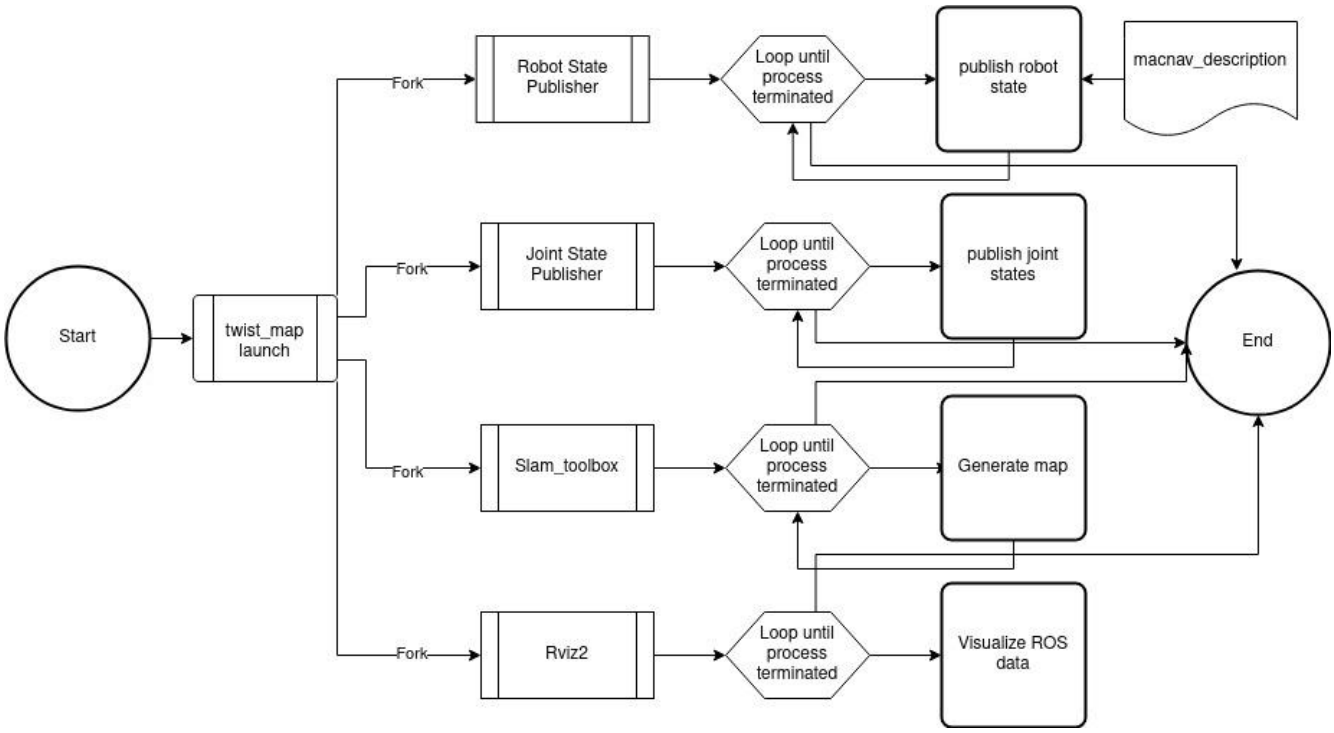


Figure 4: Ros2 flowchart for mapping

Shown in Fig 5 is the flowchart depiction of the navigation launch run on the remote machine. This launch contains 8 nodes that are all a part of the nav2 package built by Steve Macenski[6]. The first node is the path planner node. The goal of this node is to run the hybrid A* algorithm to lay out the path for the Rover to take. It examines the costmap and finds an appropriate path to reach the goal. This path has to take into account the specific physical properties of the Rover such as the minimum turn radius which are defined in a parameters file that gets distributed to all of the navigation nodes. Once the planner node generates the plan it gets sent to the smoother node. This node is responsible for smoothing the plan so the Rover doesn't get sent jerky input commands from the plan being jagged. The next node is responsible for the control of the Rover. The control node implements the regulated pure pursuit algorithm and checks progress towards the goal. If the goal has been reached this node notifies the other nodes so the whole navigation process can be completed. Once the controller node decides how the Rover should drive it passes this information to the velocity smoother node. This node implements a PID controller that prevents the Rover from accelerating or steering too quickly and potentially damaging itself. The behavior node receives behavior commands from the bt navigator and runs the requested behavior. These behaviors include typical navigation requirements such as computing the path to the goal and following the path as well as recovery behaviors. Recovery behaviors include instructions designed to potentially allow the Rover to recover from unforeseen circumstances causing problems in the navigation. These behaviors include clearing the costmaps in case some old data was causing the Rover to plan around obstacles that are no longer present, waiting in place if a dynamic obstacle enters the path, or reversing

in case the Rover drives itself into a difficult situation. The next node, the waypoint follower, is in place if the Rover is instructed to navigate through a series of waypoints. It passes subsequent goal poses to the planner after the previous one is reached so the system can hold a buffer of goals to reach instead of having to receive them individually from the operator. It also determines the behavior of the Rover when it reaches the waypoint. In our system all it does is wait a small amount of time but in other types of rover it can be assigned tasks to complete such as a warehouse robot picking up a box. The BT navigator node contains the behavior tree that determines the decisions the Rover makes to reach its goal which is shown in Fig. 6. The final node, the lifecycle manager, is the node in the navigation system that is responsible for transitioning other nodes between states. This is important so the nodes always start in the same order to prevent issues arising from nodes depending on other nodes to execute.

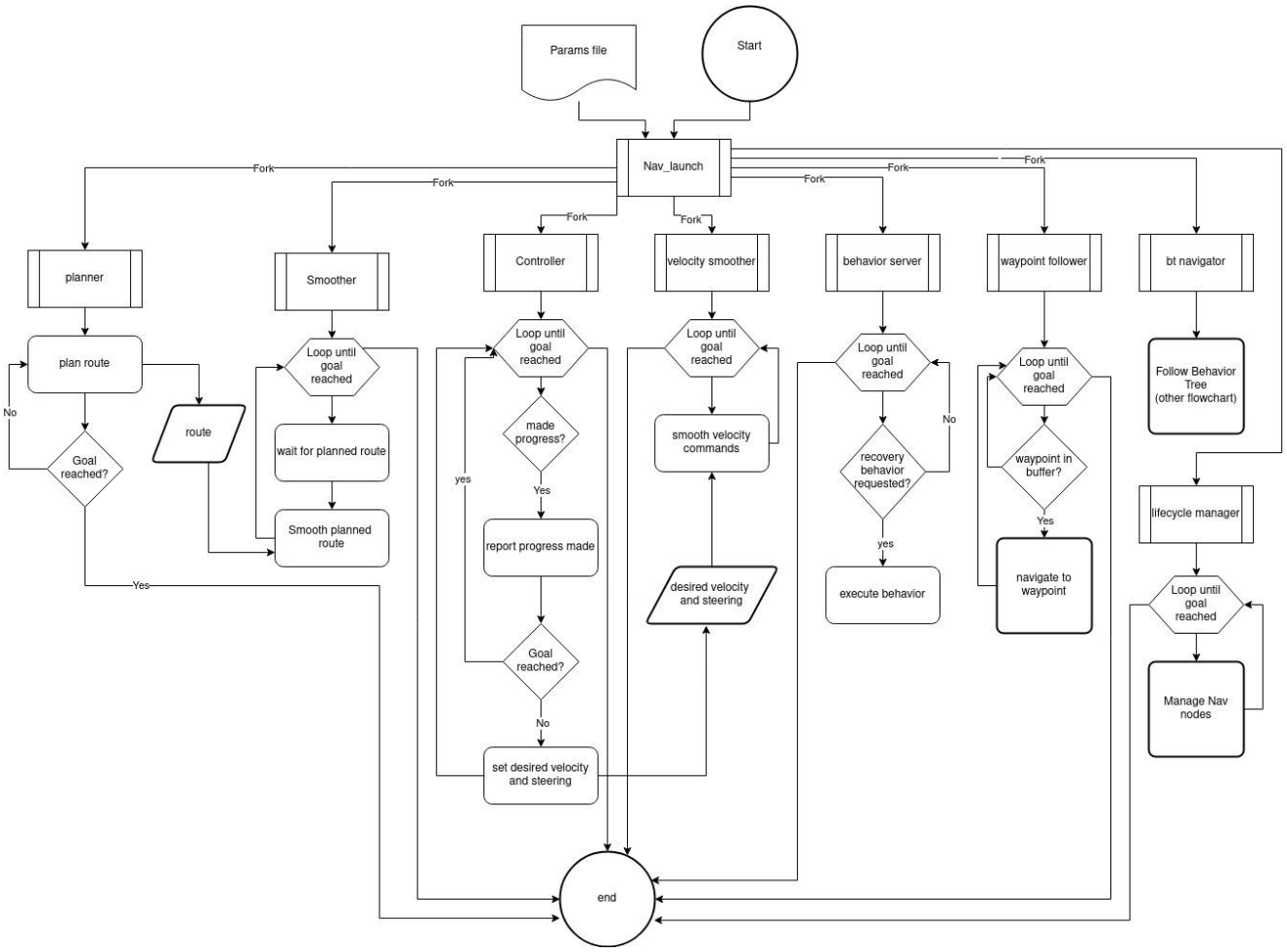


Figure 5: flowchart for navigation launch (note bt navigator node is described in Fig 6)

Nav2 behavior trees are defined in xml files and contain several types of behavior useful in executing complex tasks. The flowchart of the behavior tree for navigation is shown in Fig 6. The primary behavior in the tree is a recovery node called “navigateRecovery”. Recovery nodes attempt to complete the primary task then contain and only if that task fails attempt the recovery behavior. These nodes also contain a maximum number of retries so the

Rover can eventually give up on its task if deemed impossible. The primary task for the “NavigateRecovery” node is another recovery node called the “LaserErrorRecovery”. This node contains a pipeline sequence which is another type of behavior in which the system loops through each of its tasks in order at a given rate until either the goal is reached or one of them fails. This pipeline sequence attempts to compute the path to the goal then follow said path. If either of these tasks fail, a costmap is cleared and the task is retried. The recovery behavior of the “LaserErrorRecovery” is just a short wait with a maximum of twenty retries. The reason behind this behavior is an issue with the data from the LIDAR scanner. Occasionally, the scanner can return false positives that cause the Rover to assume an entire hallway is blocked. If 20 loops are completed and the Rover still cannot make progress it moves to the main recovery section. This section contains a round robin sequence where the Rover alternates between waiting and reversing. A round robin behavior tries to do the first time in the sequence then returns to the main navigation behavior to see if it can make progress now. If the system enters this round robin recovery six times and doesn’t reach the goal it gives up on the navigation and declares failure.

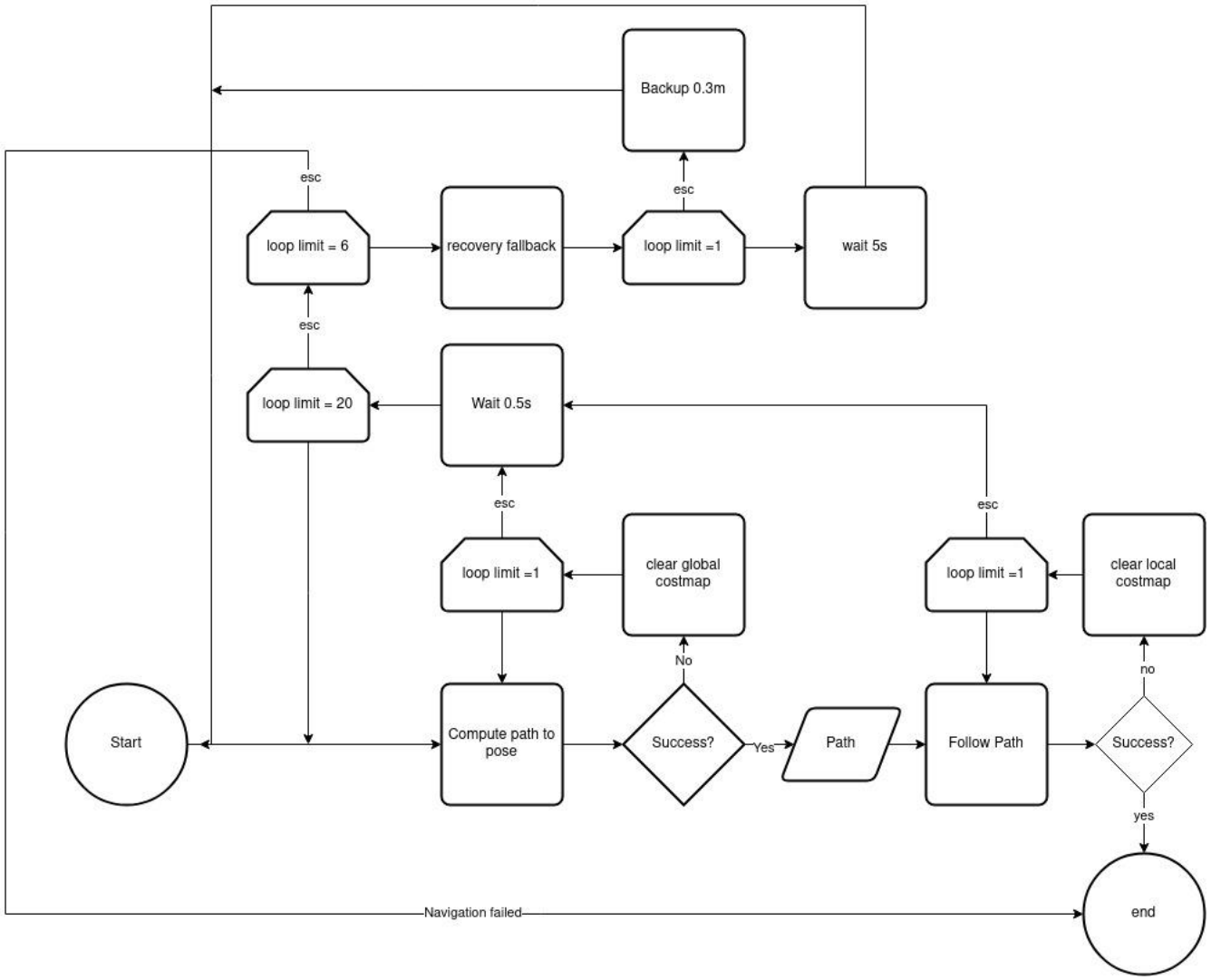


Figure 6: navigation behavior flowchart

3 Results

Trials	Time (s)	Velocity (m/s)	Avg Velocity (m/s)
1	3.05	0.66	0.66
2	3.01	0.66	
3	3.03	0.66	
4	3.04	0.66	
1	2.31	0.87	0.85
2	2.32	0.86	
3	2.44	0.82	
4	2.39	0.84	
1	1.56	1.28	1.23
2	1.7	1.18	
3	1.63	1.23	
4	1.62	1.23	

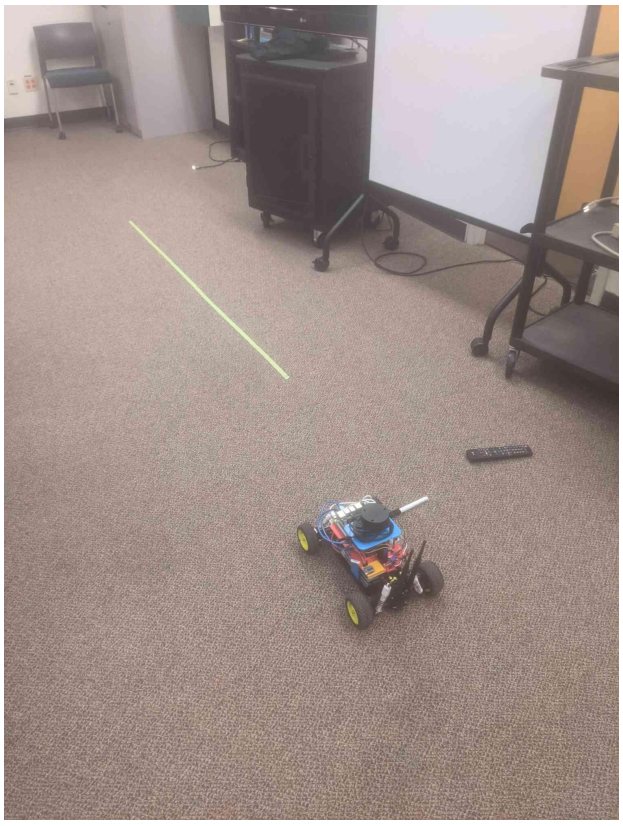


Figure 7: straight line velocity test

3.1 Odometry Testing

Two tests were done for odometry. First, a straight line test where the robot was driven an arbitrary distance, stopped then measured the actual distance traveled.

The purpose of this experiment was to determine the ratio between pulses sent from the encoder to distance traveled. The calculated gear ratio of the drivetrain was 0.0042457542 which came from adding up all of the gear ratios within the system as well as the manufacturer supplied ratio of encoder counts to rpm. This ratio is then multiplied by 2πr where r is the measured radius of the tires(0.0402m) to find the distance traveled. The data collected with this ratio (fig. 8) shows a clear pattern of the real measured distance being approximately 4 cm greater than the calculated distance, regardless of the total distance traveled. This implies that either the measurement methodology had some consistent error or there is some slack in the system where between the start of acceleration and the Rover actually moving something in the Rover has to bend first.

Odom Calculated Distance (m)	Real Distance Traveled (m)	Difference (m)
2.37	2.4	0.03
1.35	1.38	0.03
1.57	1.5	0.03
1.88	1.9	0.02
1.94	1.97	0.03
1.64	1.69	0.05
1.57	1.6	0.03
1.57	1.6	0.03
1.15	1.17	0.02

Figure 8: Straight line test data

Circle test

The purpose of this test was to determine the function to convert the steering command to the actual angle the front wheels faced. To determine this, the rover was driven in a complete circle while collecting distance traveled from the encoder. The circumference of the circle can then be used to calculate the radius of the circle to plug into the formula: Steer angle = arctan(radius/wheelBase). The wheelbase was measured to be 0.265m and the test

was conducted four times at each steering command.

Figure 9: circle test

input command (deg)	Results (m)				Avg (m)	R (m)	Steer Angle (Rad)
76	4.17489099	4.1514	4.14915	4.2156	4.172760248	0.6641154197	0.379667324
70	5.1229	5.022	5.0103	4.9985	5.038425	0.8018902441	0.3191705925
64	6.2908	6.1921	6.0173	6.1256	6.15645	0.9798294494	0.2641360853
58	7.6474	7.7031	7.8136	7.3932	7.639325	1.215836336	0.2146007564
52	11.8554	11.3129	11.1927	11.4319	11.448225	1.822041598	0.1444285543

Figure 10: right turn steering command data

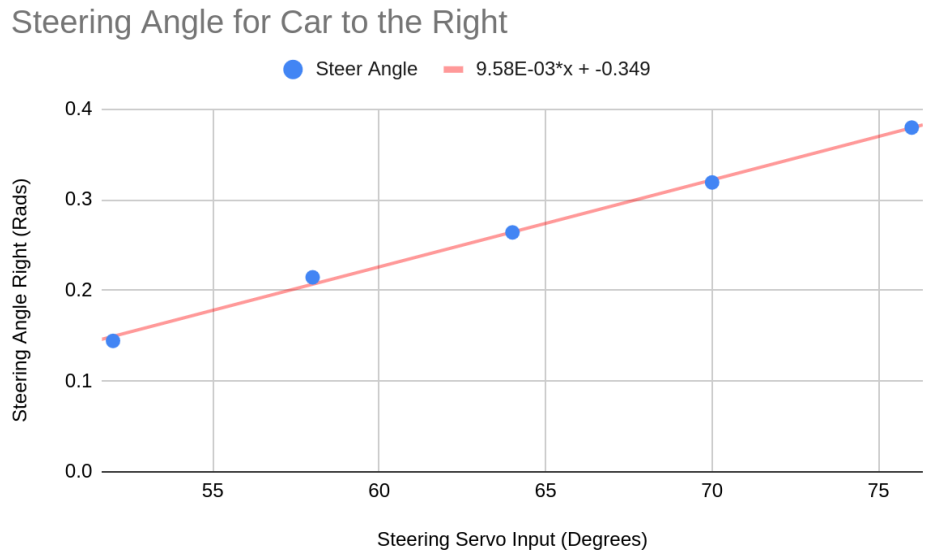


Figure 11: right turn steering graph

Input command (deg)	Results (m)				Avg (m)	R (M)	Steer Angle(rad)
0	4.7561	4.6693	4.6521	4.6286	4.676525	0.7442920702	0.3420481839
6	5.0639	5.1347	5.2055	5.2258	5.157475	0.8208376401	0.3122778707
12	5.551	5.7342	5.8393	5.9411	5.7664	0.9177510638	0.2811033834
18	7.6205	7.6667	7.6838	7.8865	7.714375	1.227780914	0.212575614
24	11.0265	11.1251	10.8431	10.6855	10.92005	1.737979936	0.1513104509

Figure 12: left turn steering command

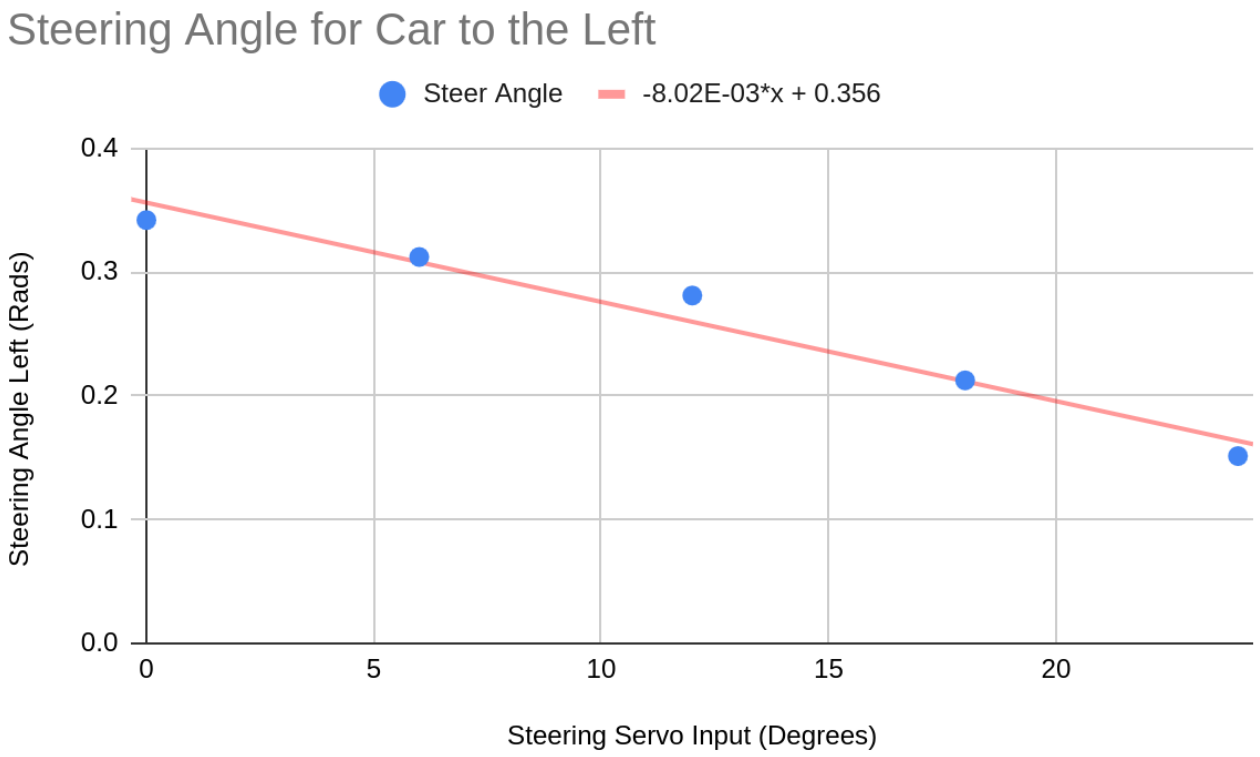


Figure 13: left steering graph

The possible steering commands range from 0-76 where commands > 38 steer to the right and < 38 steer to the left. Input commands closer to the midpoint weren't tested due to not having adequate space to complete the necessary size of circles. The data from Figures 10 & 12 where graphed and best fit lines were created to determine the formulas to convert the functions to convert servo input to steering angle.

SLAM test

In this test, the Rover was controlled manually while running all of the nodes required for mapping (fig 4). The Rover started in the senior lab and was driven clockwise around the main loop of building 5 while running SLAM_toolbox async online mapping. The map (fig. 14 is intact and well formed.

Figure 14: Slam generated map of building 5

Conclusion

This project demonstrated that it is possible to build an autonomous robot from scratch using an RC hobby car as a chassis, simple hardware, and set up a navigation application on top of the ROS2 framework. Despite not being able to utilize the floor plan maps, navigation was still made possible using the physical maps generated utilizing SLAM. Numerous unforeseen challenges were encountered and solved that spanned many fields of computer science including networking, graphics, operating systems, and robotic kinematics.

Future work

Localization in floor plan maps is a non-trivial problem. Running AMCL directly on these tends to lead to a high degree of error as non-fixed objects such as tables and chairs in the environment aren't included in the map. A solution to this was proposed by Chan et al [7]. In this solution a local slam map is generated where only the most recent data is used. Both this map and the architectural maps are processed with a sobel filter to determine the edge direction at each pixel. Then, a particle cloud is generated like in AMCL but the local slam map is used to calculate the weight of each particle. This is accomplished by calculating how well the edge direction of each pixel in the local map matches up with the corresponding pixel of the global map. The pixels that match poorly are disregarded and the total of all pixels with a positive relation is used as the weight for the particle. This method is so effective because pixels that don't match between the maps do not negatively affect how well of a match the images are to each other. This allows localization in areas with moderate amounts of clutter given some portions of the structural walls are still visible. We were able to implement this method (included in appendix) in a way that worked with small test maps, however when testing on the larger maps used in actual navigation this method proved to be too computationally expensive to keep up with the movement of the Rover. Methods to optimize the code such as multi-threading or re-working some aspects of the algorithm could be explored to improve performance.

Another future change in the project is changing from a "smart central planner, dumb robot" model to a "dumb central planner, smart robot" model. This means migrating all of the navigation software from the remote machine onto the Rover so interruptions in the network don't cause catastrophic errors. With our current system, moving around macewan causes frequent short interruptions in the connection between the central planner and the Rover as different routers need to be connected to. This can cause problems when the remote machine tries to send time sensitive drive commands to the Rover but the network is slow. The smart robot model allows all time sensitive commands to be issued locally on the Rover so the disconnection issue doesn't result in a crash.

References

[1] Macenski, S., Jambrecic I., "SLAM Toolbox: SLAM for the dynamic world", Journal of Open Source Software, 6(61), 2783, 2021.

[2] “F1TENTH,” f1tenth.org. https://f1tenth.org/index.html (accessed Oct. 10, 2023).

[3] “f1tenth_ws,” *GitHub*, Oct. 09, 2023. https://github.com/CL2-UWaterloo/f1tenth_ws (accessed Oct. 10, 2023).

[4] turtlebot, “Features · User Manual,” *turtlebot.github.io*, Feb. 08, 2022. https://turtlebot.github.io/turtlebot4-user-manual/overview/features.html

[5] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, “Practical Search Techniques in Path Planning for Autonomous Driving.” Available: https://ai.stanford.edu/~ddolgov/papers/dolgov_gpp_stair08.pdf

[6] S. Macenski, F. Martín, R. White, J. Clavero. The Marathon 2: A Navigation System. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2020.

[7] Chee Leong Chan, J. Li, Jian Le Chan, Z. Li, and K.-W. Wan, “Partial-Map-Based Monte Carlo Localization in Architectural Floor Plans,” *Lecture Notes in Computer Science*, Jan. 2021, doi: https://doi.org/10.1007/978-3-030-90525-5_47.

[8] Macenski, S., "On Use of SLAM Toolbox, A fresh(er) look at mapping and localization for the dynamic world", ROSCon 2019.

[9] Hanten, Richard & Buck, Sebastian & Otte, Sebastian & Zell, Andreas. (2016). Vector-AMCL: Vector based Adaptive Monte Carlo Localization for Indoor Maps. Advances in Intelligent Systems and Computing. 531. 10.1007/978-3-319-48036-7_29.

[10] github.com, “Jetson Nano with Ubuntu 20.04 OS image”, *GitHub.* [Online]. Available: https://github.com/Qengineering/Jetson-Nano-Ubuntu-20-image. (Accessed: 10-Oct-2023)

Appendices

Appendix A - Code

Arduino Code

A.1 Drive System: Motors

```
#include <Servo.h>

//L298N pins
#define enA 6
#define in1 12
#define in2 13

//Pulse width range
#define MIN 981 //(steering turned farthest left)
#define MAX 1776 //(steering turned farthest right)
#define MIN_ANGLE 0
#define MAX_ANGLE 76
#define CENTER_ANGLE 38

float WHEEL_BASE = 0.265; //meters

Servo steeringServo; //rotate steering linkage
int steerPin = 10;

void setup() {
  Serial.begin(115200);
  steeringServo.attach(steerPin,MIN,MAX);
  steeringServo.write(CENTER_ANGLE); //center wheels
  pinMode(enA, OUTPUT);
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
}

void loop() {
  while(!Serial.available());

  String payload1 = Serial.readStringUntil('\n'); //Along x-axis, latterally along the center of Ackerman vehicle
  String payload2 = Serial.readStringUntil('\n'); //Associated with yaw
  float linear_vel = payload1.toFloat();
  float angular_vel = payload2.toFloat();

```

```
   if(linear_vel == 0.0){
     apply_brake();
   } else {
     motion(linear_vel);
     steer(angular_vel, linear_vel);
   }

}

void steer(float ang_v, float lin_v){
  float steering_angle; //rads
  int servo_angle_input; //degrees

  if (ang_v == 0.0) {
    steeringServo.write(CENTER_ANGLE);
  } else {
    steering_angle = atan(WHEEL_BASE * (ang_v / lin_v));
    if (ang_v > 0) {
      servo_angle_input = get_servo_left_input(steering_angle);
    } else {
      servo_angle_input = get_servo_right_input(steering_angle * (-1));
    }

    steeringServo.write(servo_angle_input);
  }
}

void motion(float speed){
  int pwm_input = get_motor_input(speed);

  if (speed > 0){ //drive forwards
      digitalWrite(in1, HIGH);
      digitalWrite(in2, LOW);
      analogWrite(enA, pwm_input);
  } else {
      digitalWrite(in1, LOW);
      digitalWrite(in2, HIGH);
      analogWrite(enA, pwm_input);
  }
}

void apply_brake(){
      digitalWrite(in1, LOW);
      digitalWrite(in2, LOW);
      delay(100);
}
```

```
83
84 //Inverse function bellow for getting linear velocity moving forward
85 //Calculated from driving forward data, we'll assume driving backwards has
86 //The same function
87 int get_motor_input(float lin_velocity) {
88   if (lin_velocity < 0) lin_velocity = lin_velocity * (-1);
89   int pwm_input;
90   pwm_input = (int) ((lin_velocity + 1.46) / 0.0105);
91   if (pwm_input < 200) pwm_input = 200;
92   if (pwm_input > 255) pwm_input = 255;
93
94   return pwm_input;
95 }
96
97 //Inverse functions bellow for getting servo input was derived from steering test data
98 int get_servo_right_input(float steering_angle) {
99   int angle;
100  angle = (int) ((steering_angle + 0.349) / 0.00958);
101  if (angle > 76) angle = 76;
102  return angle;
103 }
104
105 int get_servo_left_input(float steering_angle) {
106   int angle;
107   angle = (int) ((steering_angle - 0.356) / (0.00802 * -1));
108   if (angle < 0) angle = 0;
109   return angle;
110 }
```

A.2 Rotary Encoder

```ino
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>

/*Sketch for receiving and writing encoder counts */


// Define the digital pins to which the encoder is connected
const int encoderChannelA = 2;
const int encoderChannelB = 3;

// Variables for encoder counting
//float encoderCount = 0.0;
unsigned long lastTime = 0;    // Store the last time the count was read
unsigned long deltaTime = 100; // Set the time interval for writing sensor data (in milliseconds)
uint16_t IMU_DELAY = 500; //How often to grab IMU data
unsigned long lastTime2 = 0;

typedef union {
  volatile float floatingPoint;
  byte binary[4];
} binaryFloat;

binaryFloat encoder_count;
//binaryFloat heading;

void setup() {
  // Initialize the serial communication for debugging
  Serial.begin(115200);

  // Set the encoder channel pins as inputs
  pinMode(encoderChannelA, INPUT);
  pinMode(encoderChannelB, INPUT);

  // Attach interrupt service routines to the pins for detecting changes
  attachInterrupt(digitalPinToInterrupt(encoderChannelA), encoderISR, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(encoderChannelB), encoderISR, INPUT_PULLUP);
  encoder_count.floatingPoint = 0.0;
}

void loop() {
  unsigned long currentTime = millis();
  //unsigned long currentTime2 = millis();
  //Calculate RPM at regular intervals
  if (currentTime - lastTime >= deltaTime) {
    Serial.write(encoder_count.binary,4);
    encoder_count.floatingPoint = 0; // Reset the count

    lastTime = currentTime;

  }

}

// Interrupt Service Routine (ISR) for the encoder
void encoderISR() {
  encoder_count.floatingPoint++;
}
```

A.3 IMU

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>

uint16_t BNO055_SAMPLERATE_DELAY_MS = 10; //how often to read data from the board
uint16_t PRINT_DELAY_MS = 500; // how often to print the data
uint16_t printCount = 0; //counter to avoid printing every 10MS sample
unsigned long last_time = 0.0;
unsigned long DELAY_INTERVAL = 1000;

typedef union {
  volatile float floatingPoint;
  byte binary[5];
} binaryFloat;

binaryFloat heading;
//float heading = 0.0;
// Check I2C device address and correct line below (by default address is 0x29 or 0x28)
//                                   id, address
Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28);

void setup(void)
{
  Serial.begin(115200);

  while (!Serial) delay(10);  // wait for serial port to open!

  if (!bno.begin())
  {
    Serial.print("No BNO055 detected");
    while (1);
  }


  delay(1000);
  heading.floatingPoint = 0.0;
}

void loop(void)
{
  //unsigned long tStart = micros();
  unsigned long current_time = millis();
  sensors_event_t orientationData;
  bno.getEvent(&orientationData, Adafruit_BNO055::VECTOR_EULER);

  if (current_time - last_time  >= DELAY_INTERVAL) { //Grabbing heading data
    heading.floatingPoint = (float) orientationData.orientation.x;
    heading.binary[4] =  10;

    Serial.write(heading.binary, 5);
    last_time = current_time;
  }

}
```

Python Code

A.4 Xbox 360 Controller Node

```python
#!/usr/bin/env python3
import rclpy, math
from rclpy.node import Node
from xbox360controller import Xbox360Controller
from geometry_msgs.msg import Twist


#Node passes xbox input to topic /cmd_vel

class Xbox360_contr_node(Node):
    SPEEDS = [0.0, 0.6557, 0.8461, 1.23] # m/s

    #Scaled by order of 2 from joystick inputs
    JOY_MIN = -100
    JOY_MAX = 100

    #Steering servo angle range
    STEER_MIN = 0
    STEER_MAX = 76
    STEER_CENTER = 38
    WHEEL_BASE = 0.265 #meters

    prev_ang = 0
    prev_lin = 0

    def __init__(self):
        super().__init__("xbox_controller")
        self.speedIndex = 1
        self.cur_angular_vel =  0.0
        self.current_linear_vel = 0.0
        self.allow_joystick_steering = False

        #Initialize and setup xbox controller
        self.controller = Xbox360Controller(0,axis_threshold=0)
        self.controller.button_b.when_pressed = self.apply_brake
        self.controller.axis_r.when_moved = self.on_joy_move
        self.controller.button_x.when_pressed = self.on_speed #Forwards
        self.controller.button_y.when_pressed = self.on_speed #Reverse
        self.controller.button_a.when_pressed = self.steering_mode_toggle

        #create xbox controller publishing node
        self.xbox_pub = self.create_publisher(Twist,"/cmd_vel",10)

        self.get_logger().info("xbox node initialized")

    #Callback function associated with drive_mode change
    def apply_brake(self, button):
        msg = self.create_message(self.SPEEDS[0],self.cur_angular_vel)

        self.speedIndex = 1
        print("Brake applied!")
        self.xbox_pub.publish(msg)

    #Callback function disables/enables joystick steering
    def steering_mode_toggle(self, button):
        if self.allow_joystick_steering == False:
            print("Joystick steering enabled.")
            self.allow_joystick_steering = True
        else:
            print("Joystick steering disabled.")
            self.allow_joystick_steering = False

    #Callback function associated with steering servo angle
    def on_joy_move(self, axis):
        if self.allow_joystick_steering:
            #Check if joystick is not at neutral
            if not (axis.x > -0.25 and axis.x < 0.25):
                joy_input = int(axis.x * 100)
                servo_angle = self.map_range(joy_input)
                #steer_angle =  self.STEER_CENTER - servo_angle
                steer_angle = self.get_steering_angle(servo_angle)
                #if (servo_angle > self.STEER_CENTER): steer_angle *= -1
                angular_vel_z = self.current_linear_vel * \
                    ((math.tan(steer_angle)) / self.WHEEL_BASE)
            else:
                angular_vel_z = 0.0

            if self.current_linear_vel != self.prev_lin or angular_vel_z != self.prev_ang:
                msg = self.create_message(self.current_linear_vel, angular_vel_z )
                self.xbox_pub.publish(msg)
                self.prev_ang = angular_vel_z
                self.prev_lin = self.current_linear_vel

```

```
    #Callback function associated with shifting speed
    def on_speed(self, button):
        if button.name == "button_x":
            #Move forward
            lin_vel_x = self.SPEEDS[self.speedIndex]
            print("Moving forward")
            print(f"Speed: {self.SPEEDS[self.speedIndex]}")
        else:
            #Move backward
            print("Moving Backward")
            lin_vel_x = (-1) * self.SPEEDS[self.speedIndex]

        print(f"Speed shifted: {self.SPEEDS[self.speedIndex]} m/s")
        self.speedIndex = (self.speedIndex + 1) % len(self.SPEEDS)
        if lin_vel_x != self.prev_lin or self.cur_angular_vel != self.prev_ang:
            msg = self.create_message(lin_vel_x, self.cur_angular_vel)
            self.xbox_pub.publish(msg)
            self.prev_lin = lin_vel_x
            self.prev_ang = self.cur_angular_vel

    def create_message(self, lin_vel_x, ang_vel_z):
        msg = Twist()

        msg.linear.x = lin_vel_x
        msg.linear.y = 0.0
        msg.linear.z = 0.0

        msg.angular.x = 0.0
        msg.angular.y = 0.0
        msg.angular.z = ang_vel_z

        self.cur_angular_vel = ang_vel_z
        self.current_linear_vel = lin_vel_x

        return msg

    def map_range(self, joy_input) :
        return (joy_input - self.JOY_MIN) * (self.STEER_MAX - self.STEER_MIN) //  \
                (self.JOY_MAX - self.JOY_MIN) + self.STEER_MIN

   #Return steering angles (rads) based on linear regression lines
   #Derived from test data for estimating steering angle for MacNav
    def get_steering_angle(self, servo_angle_input):
        if servo_angle_input > 38: #Steering right
            return (9.58E-3*servo_angle_input - 0.349) * (-1)
        else: #Steering left
            return -8.02E-3*servo_angle_input + 0.356

def main(args=None):
    rclpy.init(args=args)
    contr_node = Xbox360_contr_node()
    rclpy.spin(contr_node)
    contr_node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()
```

A.5 Odometry Node

```python
#!/usr/bin/env python3
import rclpy, serial, threading, signal
from rclpy.node import Node
from nav_msgs.msg import Odometry
from tf2_msgs.msg import TFMessage
from geometry_msgs.msg import TransformStamped
from geometry_msgs.msg import Twist

import math
import numpy as np

odom_arduino = serial.Serial(port="/dev/ttyUSB0",baudrate=115200)

class Odom_node(Node):

    TIMER_INTERVAL = 0.1
    WHEEL_BASE = 0.265
    WHEEL_RADIUS = 0.0402
    R = 0.0042457542

    def __init__(self):
        super().__init__("odom_publsiher")
        self.current_steering_angle = 0.0
        self.heading = 0.0
        self.x = 0.0
        self.y = 0.0
        self.forwards = True

        self.odom_publisher = self.create_publisher(Odometry, 'odom', 10)
        self.tfpublisher = self.create_publisher(TFMessage, 'tf', 10)
        self.tmr = self.create_timer(self.TIMER_INTERVAL, self.publish)
        self.twist_sub = self.create_subscription(Twist, '/cmd_vel',
            self.update_kinematics, 10)

        self.get_logger().info("Kinematic Arduino odom node init")

    def euler_to_quaternion(self, r):
        (yaw, pitch, roll) = (r[0], r[1], r[2])
        qx = math.sin(roll/2) * math.cos(pitch/2) * math.cos(yaw/2) - math.cos(roll/2) * math.sin(pi
        qy = math.cos(roll/2) * math.sin(pitch/2) * math.cos(yaw/2) + math.sin(roll/2) * math.cos(pi
        qz = math.cos(roll/2) * math.cos(pitch/2) * math.sin(yaw/2) - math.sin(roll/2) * math.sin(pi
        qw = math.cos(roll/2) * math.cos(pitch/2) * math.cos(yaw/2) + math.sin(roll/2) * math.sin(pi
        return [qx, qy, qz, qw]

    def publish(self):
        if odom_arduino.is_open:
            try:
                counts_bytes = odom_arduino.read_until(size=4)
                counts = np.frombuffer(counts_bytes, dtype=np.float32)[0]
            except:
                counts = 0.0

            #calculate linear distance traveled
            if self.forwards:
                linear_d = (float(counts) * self.R * 2.0 * \
                    math.pi * self.WHEEL_RADIUS)
            else:
                linear_d = (-float(counts) * self.R * 2.0 * \
                    math.pi * self.WHEEL_RADIUS)


            #change in heading from last sample to this one
            d_heading = ((linear_d / self.WHEEL_BASE)) \
                * math.tan(self.current_steering_angle)

            #avg change in heading over that time
            avg_d_heading = d_heading / 2

            #trig to calc distance traveled
            self.x = linear_d * math.cos(self.heading + avg_d_heading)
            self.y = linear_d * math.sin(self.heading + avg_d_heading)

```

```
72
73              # update heading
74              self.heading += d_heading
75              self.get_logger().info(f"{counts}")
76
77              self.publish_odom()
78
79      def publish_odom(self):
80          msg = Odometry()
81          msg.header.stamp = self.get_clock().now().to_msg()
82          msg.header.frame_id = "odom"
83          msg.pose.pose.position.x = self.x
84          msg.pose.pose.position.y = self.y
85          msg.pose.pose.position.z = 0.0
86          r = self.euler_to_quaternion([self.heading, 0.0, 0.0])
87          msg.pose.pose.orientation.x = r[0]
88          msg.pose.pose.orientation.y = r[1]
89          msg.pose.pose.orientation.z = r[2]
90          msg.pose.pose.orientation.w = r[3]
91          msg.child_frame_id = "base_link"
92
93          self.publish_tf(msg)
94          self.odom_publisher.publish(msg)
95
96      def publish_tf(self, odom):
97          tf = TransformStamped()
98
99          tf.header = odom.header
100         tf.child_frame_id = odom.child_frame_id
101
102         tf.transform.translation.x = odom.pose.pose.position.x
103         tf.transform.translation.y = odom.pose.pose.position.y
104         tf.transform.translation.z = odom.pose.pose.position.z
105         tf.transform.rotation = odom.pose.pose.orientation
106
107         msg = TFMessage()
108
109         msg.transforms = [tf]
110         self.tfpublisher.publish(msg)
111
112     def update_kinematics(self, twist):
113         target_lin_velocity = twist.linear.x
114         target_ang_velocity = twist.angular.z
115         self.current_steering_angle = self.get_steer_angle(target_lin_velocity, \
116             target_ang_velocity)
117
118         if twist.linear.x > 0:
119             self.forwards = True
120         elif twist.linear.x < 0.0:
121             self.forwards = False
122         else:
123             pass
124
125     def get_steer_angle(self, x, z):
126         if z == 0:
127             return self.current_steering_angle
128         return math.atan((z/x) * self.WHEEL_BASE)
129
130 def main(args=None):
131     rclpy.init(args=args)
132     odom_node = Odom_node()
133     rclpy.spin(odom_node)
134     odom_node.destroy_node()
135     rclpy.shutdown()
136
137 if __name__ == "__main__":
138     main()
139     odom_arduino.close()
140
```

A.6 Drive Node

```python
#!/usr/bin/env python3
import rclpy, serial
from rclpy.node import Node
from geometry_msgs.msg import Twist
import struct

#Node passes xbox input to topic /arduino_channel
arduino = serial.Serial(port="/dev/ttyACM0",baudrate=115200)

prevAng = 0.0
prevLin = 0.0

class Arduino_node(Node):
    prevAng = 0.0
    prevLin = 0.0
    def __init__(self):
        super().__init__("manual_node")
        #create arduino subscribing node
        self.vel_sub = self.create_subscription(Twist,"/cmd_vel",
            self.send_drive_msg,10)
        self.get_logger().info("Arduino drive node initialized")

    def send_drive_msg(self,msg):
        if(msg.angular.z != self.prevAng or msg.linear.x != self.prevLin):
            sendTwist(arduino, msg.angular.z, msg.linear.x)
            self.prevAng = msg.angular.z
            self.prevLin = msg.linear.x

def sendTwist(ard, ang, lin):
    ard.write(f"{lin}\n{ang}\n".encode())


def main(args=None):
    rclpy.init(args=args)
    arduino_node = Arduino_node()
    rclpy.spin(arduino_node)
    arduino_node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()
    arduino.close()
```

A.7 IMU Node

```python

#!/usr/bin/env python3
import rclpy, serial, math
from rclpy.node import Node
import numpy as np
from mac_messages.msg import Imu

"""IMU node receives data from imu as heading values on one second inter

imu_arduino = serial.Serial(port="/dev/ttyACM1",baudrate=115200)

class IMU_Node(Node):
    TIMER_INTERVAL = 0.5

    def __init__(self):
        super().__init__('imu_publisher')

        self.tmr = self.create_timer(self.TIMER_INTERVAL, self.publish)
        self.IMU_publisher = self.create_publisher(Imu, '/imu',5)
        self.get_logger().info("Arduino IMU node initialized")

    def publish(self):
        msg = Imu()
        try:
            heading_bytes = imu_arduino.read_until()
            heading_bytes = heading_bytes[:4]
            new_heading = float((np.frombuffer(heading_bytes,dtype=np.fl
        except:
            new_heading = 0.0
        #self.get_logger().info(f"{new_heading}")
        msg.heading = new_heading
        self.IMU_publisher.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    imu_node = IMU_Node()
    rclpy.spin(imu_node)
    imu_node.destroy_node()
    rclpy.shutdown()

if __name__ == "__main__":
    main()
    imu_arduino.close()
```

XML

A.8 Navigation Behavior tree XML

```xml
<root main_tree_to_execute="MainTree">
  <BehaviorTree ID="MainTree">
    <RecoveryNode number_of_retries="6" name="NavigateRecovery">
      <RecoveryNode number_of_retries="20" name="LaserErrorRecovery">
        <PipelineSequence name="NavigateWithReplanning">
          <RateController hz="1.0">
            <RecoveryNode number_of_retries="1" name="ComputePathToPose">
              <ComputePathToPose goal="{goal}" path="{path}" planner_id="GridBased"/>
              <ClearEntireCostmap name="ClearGlobalCostmap-Context" service_name="global_costmap/clear_entirely_global_costmap"/>
            </RecoveryNode>
          </RateController>
          <RecoveryNode number_of_retries="1" name="FollowPath">
            <FollowPath path="{path}" controller_id="FollowPath"/>
            <ClearEntireCostmap name="ClearLocalCostmap-Context" service_name="local_costmap/clear_entirely_local_costmap"/>
          </RecoveryNode>
        </PipelineSequence>
        <Wait wait_duration="0.5"/>
      </RecoveryNode>
      <ReactiveFallback name="RecoveryFallback">
        <GoalUpdated/>
        <RoundRobin name="RecoveryActions">
          <Sequence name="ClearingActions">
            <ClearEntireCostmap name="ClearLocalCostmap-Subtree" service_name="local_costmap/clear_entirely_local_costmap"/>
            <ClearEntireCostmap name="ClearGlobalCostmap-Subtree" service_name="global_costmap/clear_entirely_global_costmap"/>
          </Sequence>
          <Wait wait_duration="5"/>
          <BackUp backup_dist="0.30" backup_speed="0.05"/>
        </RoundRobin>
      </ReactiveFallback>
    </RecoveryNode>
  </BehaviorTree>
</root>
```

CPP

A.9 cpp PSLAM localization

```
private:
    float * sobelX(float* old_map, size_t width, size_t height){
        float * map = new float[width * height];
        for(size_t i = 1; i<height-1; i++){
            for(size_t j = 0; j<width; j++){
                /*
                -1 0 1
                -2 0 2
                -1 0 1
                */
                map[j + i*width] =  lerp(-old_map[(j-1) + (i-1)*width] -
                                    2*old_map[(j-1) + (i)*width] -
                                    old_map[(j-1) + (i+1)*width] +
                                    old_map[(j+1) + (i-1)*width] +
                                    2*old_map[(j+1) + (i)*width] +
                                    old_map[(j+1) + (i+1)*width],
                                    -400, 400, -M_PI, M_PI);
                if(map[j + i*width] != 0){
                }
            }
        }
        return map;
    }

private:
    float* sobelY(float* old_map, size_t width, size_t height){
        float* map = new float[width * height];
        for(size_t i = 1; i<height-1; i++){
            for(size_t j = 0; j<width; j++){
                /*
                -1 -2 -1
                 0  0  0
                 1  2  1
                */
                map[j + i*width] = lerp(-old_map[(j-1) + (i-1)*width] -
                                    2*old_map[(j) + (i-1)*width] -
                                    old_map[(j+1) + (i-1)*width] +
                                    old_map[(j-1) + (i+1)*width] +
                                    2*old_map[(j) + (i+1)*width] +
                                    old_map[(j+1) + (i+1)*width],
                                    -400, 400, -M_PI, M_PI);
            }
        }
        return map;
    }
```

```
float get_pixel_weight(unsigned int lx, unsigned int ly, unsigned int gx,
    unsigned int gy, geometry_msgs::msg::TransformStamped local_to_base_tf){
    float local_dir = get_direction(lx, ly, local_sobel_x_,
        local_sobel_y_, local_map_width_);
    float global_dir = get_direction(gx, gy, global_sobel_x_,
        global_sobel_y_, global_map_width_);

    float difdir = std::abs(local_dir)-std::abs(global_dir);
    float q = 1 - 2/M_PI * std::abs(std::atan(std::sin(difdir)/std::cos(difdir)));
    if(q>0){
        geometry_msgs::msg::PointStamped localPoint;
        localPoint.header.frame_id = "local_map";
        localPoint.point.x = local_origin_.position.x + lx*0.05;
        localPoint.point.y = local_origin_.position.y + ly*0.05;
        geometry_msgs::msg::PointStamped basePoint;
        tf2::doTransform(localPoint, basePoint, local_to_base_tf);
        float d = std::sqrt(std::pow(basePoint.point.x, 2) +
            std::pow(basePoint.point.y, 2));
        if(d!=0)
            return q * get_magnitude(lx, ly, local_sobel_x_,
                local_sobel_y_, local_map_width_) / d;
    }
    return 0;
}
```

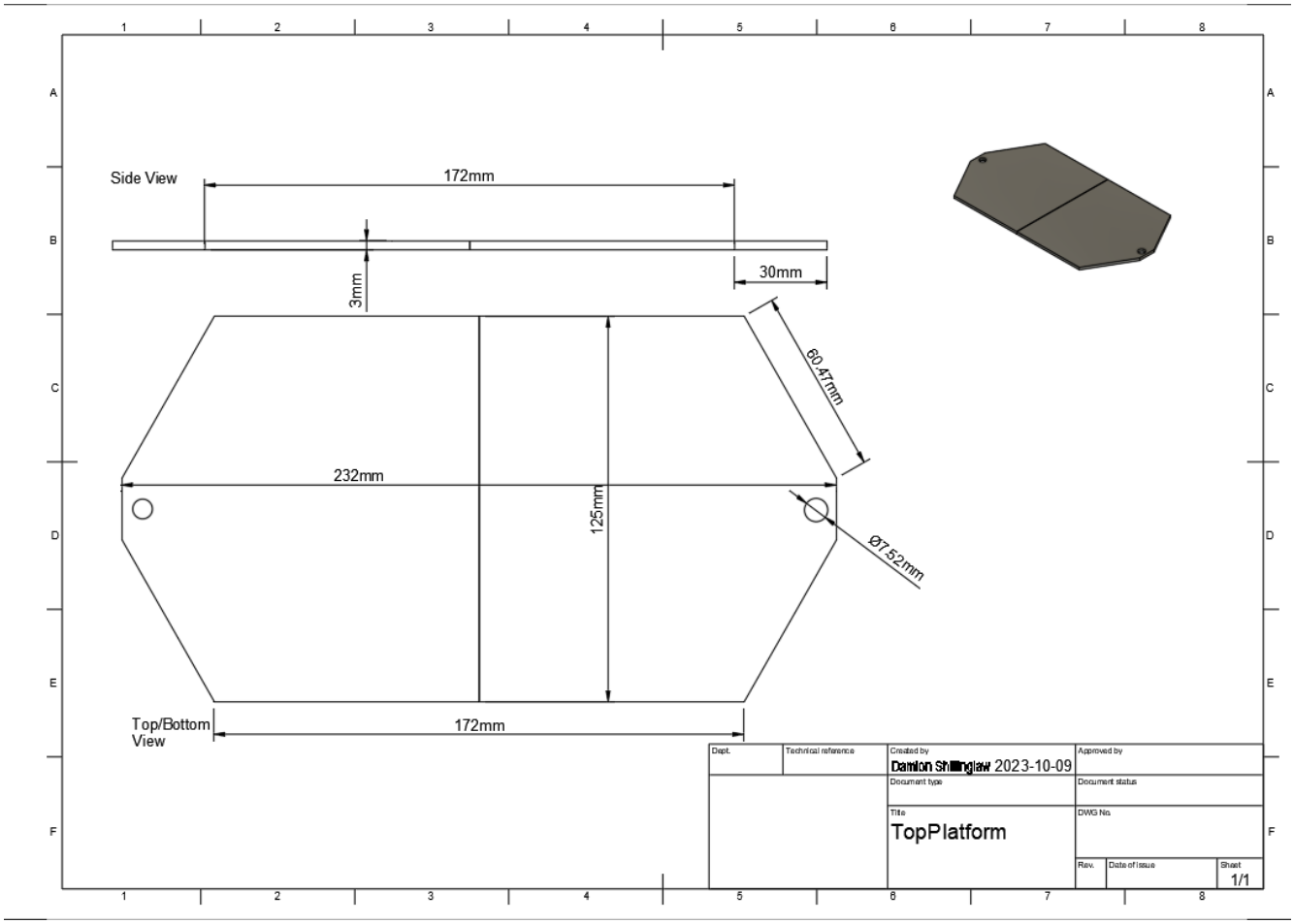
```
float get_weight(Eigen::Affine2d particle,
    geometry_msgs::msg::TransformStamped base_to_local_tf,
    geometry_msgs::msg::TransformStamped global_to_base_tf,
    geometry_msgs::msg::TransformStamped local_to_base_tf){
    float w = 0;

    RCLCPP_INFO(rclcpp::get_logger("l"), "width height = %d %d ",
        local_map_width_, local_map_height_);
    for(unsigned int i = 2; i<local_map_height_-2;i++){
        for(unsigned int j = 2; j<local_map_width_-2;j++){
            // find equivalent global pixel
            //find pixel as point in baselink frame
            geometry_msgs::msg::PointStamped pixel_local;
            pixel_local.header.frame_id = "local_map";
            pixel_local.point.x = local_origin_.position.x + j*0.05;
            pixel_local.point.y = local_origin_.position.y + i*0.05;

            geometry_msgs::msg::PointStamped pixel_base;
            tf2::doTransform(pixel_local, pixel_base, base_to_local_tf);

            // apply transformation to pixel
            Eigen::Vector2d base_point(pixel_base.point.x,
                pixel_base.point.y);
            Eigen::Vector2d transformed_point = particle * base_point;


            geometry_msgs::msg::PointStamped transformed_base_point;
            transformed_base_point.header = pixel_base.header;
            transformed_base_point.point.x = transformed_point.x();
            transformed_base_point.point.y = transformed_point.y();

            geometry_msgs::msg::PointStamped transformed_global_point;
            tf2::doTransform(transformed_base_point,
                transformed_global_point, global_to_base_tf);

            // get global pixel at transformed point
            int gx = static_cast<int>((transformed_global_point.point.x
                - global_origin_.position.x)/0.05);
            int gy = static_cast<int>((transformed_global_point.point.y
                - global_origin_.position.y)/0.05);
            w += get_pixel_weight(j, i, gx, gy, local_to_base_tf);
        }
    }
    return w;
```

```cpp
geometry_msgs::msg::PoseStamped find_best_particle(
    geometry_msgs::msg::PoseStamped globalPose,
    geometry_msgs::msg::TransformStamped global_to_local_tf,
    geometry_msgs::msg::TransformStamped base_to_local_tf,
    geometry_msgs::msg::TransformStamped global_to_base_tf,
    geometry_msgs::msg::TransformStamped local_to_base_tf)
    {
        std::vector<Eigen::Affine2d> particles;
        for(int i=0; i<100; i++){
            particles.push_back(gen_matrix());
        }
        float maxW = 0;
        float w = 0;
        Eigen::Affine2d bestParticle;

        for(long unsigned int i =0; i < particles.size(); i++){
            RCLCPP_INFO(rclcpp::get_logger("l"), "getting w  %ld", i);
            w = get_weight(particles[i], base_to_local_tf,
                global_to_base_tf, local_to_base_tf);
            RCLCPP_INFO(rclcpp::get_logger("l"), "w = %f", w);
            if(w > maxW){
                bestParticle = particles[i];
                maxW = w;
            }
        }
        Eigen::Vector2d base_point(0, 0);
        Eigen::Vector2d transformed_base_point = bestParticle * base_point;
        Eigen::Matrix2d rotationMatrix = bestParticle.rotation();
        Eigen::Rotation2Dd rotation(rotationMatrix);
        double angle = rotation.angle();

        geometry_msgs::msg::TransformStamped new_global_to_local_tf;
        new_global_to_local_tf.transform.translation.x =
            global_to_local_tf.transform.translation.x +
            transformed_base_point.x();
        new_global_to_local_tf.transform.translation.y =
            global_to_local_tf.transform.translation.y +
            transformed_base_point.y();
        tf2::Matrix3x3 euler = quaternionToEuler(global_to_local_tf.transform.rotation);
        double roll, pitch, yaw;
        euler.getRPY(roll, pitch, yaw);
        new_global_to_local_tf.transform.rotation = eulerToQuaternion(0, 0, yaw + angle);
        new_global_to_local_tf.header = global_to_local_tf.header;
        new_global_to_local_tf.child_frame_id = global_to_local_tf.child_frame_id;
        tf_broadcaster_->sendTransform(new_global_to_local_tf);
        RCLCPP_INFO(rclcpp::get_logger("localization_node"), "best: w: %f, xy: %f %f, ang %f",
            maxW, transformed_base_point.x(), transformed_base_point.y(), angle);
        //return bestParticle;
}
```

```cpp
void find_pose_estimate(){

    // find baselink in localmap
    geometry_msgs::msg::TransformStamped base_to_global_tf;
    geometry_msgs::msg::TransformStamped global_to_local_tf;
    geometry_msgs::msg::TransformStamped base_to_local_tf;
    geometry_msgs::msg::TransformStamped global_to_base_tf;
    geometry_msgs::msg::TransformStamped local_to_base_tf;
    bool tf_found = false;
    try{global_to_base_tf = tf_buffer_->lookupTransform("map", "base_link", tf2::TimePointZero);
        base_to_global_tf = tf_buffer_->lookupTransform("base_link", "map", tf2::TimePointZero);
        global_to_local_tf = tf_buffer_->lookupTransform("map", "local_map", tf2::TimePointZero);
        base_to_local_tf = tf_buffer_->lookupTransform("base_link", "local_map", tf2::TimePointZero);
        local_to_base_tf = tf_buffer_->lookupTransform("local_map", "base_link", tf2::TimePointZero);
        tf_found = true;} catch (tf2::TransformException &){}
    if(tf_found){
        RCLCPP_INFO(rclcpp::get_logger("l"), "found");
        geometry_msgs::msg::PoseStamped globalPose;
        geometry_msgs::msg::PoseStamped basePose;
        basePose.header.frame_id = "base_link";
        tf2::doTransform(basePose, globalPose, global_to_base_tf);
        // generate points within r of baselink
        // find highest weight point
        geometry_msgs::msg::PoseStamped bestPose =
            find_best_particle(globalPose,
            global_to_local_tf,
            base_to_local_tf,
            global_to_base_tf,
            local_to_base_tf);
        RCLCPP_INFO(rclcpp::get_logger("localization_node"),
            "best %s xy %f %f",
        bestPose.header.frame_id.c_str(),
        bestPose.pose.position.x, bestPose.pose.position.y);
    }
```

Appendix B - Support Designs

B.1 Top platform



B.2 Side Bracket