Autonomy Science and Systems

Howard Brand, Dept. of Automotive Engineering
Max Diekel, Dept. of Automotive Engineering
Yuhao Liu, Dept. of Automotive Engineering
Mayukh Sattiraju, Dept. of Electrical Engineering
Ramraj Segur-Mahadevaraja, Dept. of Automotive Engineering

1. INTRODUCTION

With the increase in automation and the advancement of robotic technologies, comes an increasing push to incorporate automation into the development of self-driving vehicles. Many companies have developed initiatives, devoting resources to this aim. Google began work on the development of self-driving vehicles since 2009 leading to the offspring of Waymo in 2016 [2]. Samsung is partnering with Hyundai and planning to develop vehicles with advanced sensors and machine-learning systems [4]. Apple received a permit to test self-driving vehicles on California roads [3]. Uber is now using self-driving cars to pick up passengers in Arizona [1]. With this forward momentum towards introducing self-driving vehicles to the general public, there is an increase in automation sciences being applied to automotive technologies.

The Autonomy Sciences and Systems course was offered on the Spring of 2017. This course engaged students into a survey of state-of-the art robotics algorithms found in research while introducing tools used to develop these technologies. Among the tools introduced in the course was the robotic operating system (ROS) which is utilized in many current robotics research applications. ROS is an open source programming environment which supports a community of robotics researchers. It provides a flexible framework where sub-applications from many different developers can be connected over a network to accomplish a main application. This enhances collaborative research, providing a means of integrating different applications together and a means of comparing different methods. Open source applications are made available through GitHub. GitHub is an Internet hosting service which hosts version control repositories. This allows one to have access to source code at different versions of its development and as new versions are added. ROS supports a community of autonomy science implementations, facilitating sharing of source code and links to research publications of current state of the art autonomy solutions. Throughout the course, autonomous navigation and mapping applications were developed in ROS. This was first developed for a slow or low dynamic application utilizing the Turtlebot as robotic platform. This eventually led to a final project where autonomous navigation and mapping were performed on an RC race car in a self-driving, high-speed, racing application on a test track. For the project core requirements regarding autonomous navigation at the fastest and safest speed possible were emphasized. This presented that challenge of choosing methods oriented for speed and fast navigation. For a demonstration bonus challenges were presented, related to partial and full obstructions.

This is a report documenting Team As implementation of the final project. This will begin with a background of robotics challenges similar to the final project. This will continue with a discussion of the software tools and methods used to develop in ROS. Afterwards the implementation system including hardware and software will be discussed along with challenges. Following this will be a presentation of

the mapping results will be given as well as other deliverables and results. Finally summarizing and concluding statements of the project will be given, along with future work.

The background is meant to orient the reader to the technologies, robotics courses, and robotics challenges currently being implemented. From this the tasks and requirements of the final project can be compared to current tasks being explored in research challenges and initiatives. The section discussing software tools orients the reader real procedures and software tools that enhance the developing process in ROS. Visualizations tools are presented as well as debugging tools. When viewing the implementation section, it is important bear in mind the objective being realized in the solution.

The objective of the Team As implementation was based on core requirements emphasized throughout the course. These requirements included developing an autonomous navigation application optimized for high speeds around the track. In response to these requirements, an implementation of pure pursuit was chosen in an attempt to reach the highest speeds possible. The gains associated with this method, however, were in response to bonus challenges presented for the demonstration. The results illustrated in the report display the mapping performance of the robot as the robot executes high-speed dynamic maneuvers while meeting by the project and demo requirements. Conclusions will be made related to the project implementation as well as the benefits of the experiences facilitated by the course.

2. BACKGROUND

A introduction of Robotics Challenges and ventures that are similar to the objective presented in this report.

2.1 F1 Tenth

2.1.1 *Overview.* The F1/10 competition is an education, research, and development program started and organized by University of Pennsylvania with the goal creating autonomy that goes beyond simple safety (conservative algorithms). As autonomous vehicles leave research labs and drive onto our roads, we need new solutions which are better than human drivers exhibiting lightning quick reflexes, expert tactics, and scenario understanding. The F1/10 competition offers researchers an opportunity to test such concepts in a cost effective and safe manner by competing against other state-of-the-art driving algorithms using small, cost-effective, unmanned vehicles.

The F1/10 competition focuses on creating a meaningful and challenging design experience for students. The competition involves designing, building, and testing an autonomous 1/10th scale F1 race car (capable of speeds in excess of 40MPH) all while learning about perception, planning, and control for autonomous navigation. In addition to providing lectures and reading material as an online teaching kit, organizer will also introduce an autonomous racing.

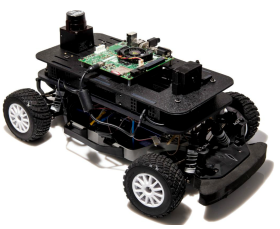


Fig. 1. F1 Tenth - Car

2.1.2 *Objective for Participants*

—Construct a 1/10th scale autonomous vehicle within the constraints de- scribed herein

—Provide a robust and safe mechanism for bringing an errant vehicle to a stop as described in this document

—Demonstrate teleoperation of the vehicle in order to verify basic functionality

—Persistently complete a mission defined by the safe traversal of ordered series of checkpoints with the objective of minimizing completion time.

—Interpret static obstacles within the environment in order to maintain collision free progress

—Exhibit context dependent speed and angular velocity control in a static environment.

—Interpret dynamic obstacles within the race environment in order to enable predictive controls and planning, such as is necessary to ensure collision free progress.

—Exhibit context dependent speed and angular velocity control in a dynamic environment

—Navigate in areas where sensors may not provide map-based localization (ie LiDAR range is insufficient)

—Accomplish these goals using a low power embedded processor specified by the organizers

2.1.3 *Hardware*

—Sensor Configuration: Each team must choose a sensor configuration only from this subset of sensors. Alternate sensors will not be considered, the purpose of this competition is the development of driving algorithms. Please see the bill of materials for part numbers and ordering information.
  —Camera: ZED, Structure Sensor, Pointgrey, Minoru, Webcam
  —LIDAR: Hokuyo 10LX, RP Lidar, Hokuyo 04LX
  —IMU: Memsic IMU440CA-200, Razor
  —WIFI: Ubiquiti Picostation M2

—Computing
  —Planning and Perception (exactly 1): NVIDIA Jetson TK1
  —Control: MBed or Teensy

—Chassis and Vehicle
  —Vehicle: Traxxas Rally 1/10
  —Suspension
  —AxleConversion: Stock, STRacingConceptsST3654-17S CNC Machined Aluminum 17mm Hex Conversion
  —Tires (swappable): Stock, Duratrax Bandito Buggy Tire C2 Mounted White (1/8 Scale)

2.1.4 *Test Scenario*

—The race track will be located at Carnegie Mellon University in Pittsburgh. Although the final layout has not yet been selected, teams may expect that racing will occur in hallways roughly 2-5 meters wide. Furthermore, teams may expect multiple turns (in both directions!), uneven walls, varying lighting conditions, and in later rounds another autonomous competitor.

—Maximum and minimum width of the course: 4 meters. Minimum centerline radius turn on the course: 3 meters details will be provided in the RNDF and ROSBAG data.

—A ROSBAG containing sensor data collected by the organizers F1/10 car of a superset of the course will be provided by 8/10/2016

Fig. 2. F1 Tenth - Test Scenario

—We note that such data will be a superset of possible course layouts in order to discourage railroad-like approaches where the majority of planning is done offline (rewarding teams with better access to the facilities and more expensive ancillary equipment).

2.1.5 *Online Learning Materials.* Our fun and informative lectures walk you through all the steps to get car moving, driving straight, making turns and taking loops. This gets you to the baseline autonomous operation of the vehicle. The course takes 2-3 weeks and then you are ready to build on top of the software to win the race. No prior knowledge is assumed and this is a course meant to make autonomous systems available and accessible F1 tenth website provide enough information for car build and assembly and also some basic tutorial to let participants to get a basic overview of ROS system, PID controller and Hector SLAM, which will support participants a good start to finish autonomous walkthrough.

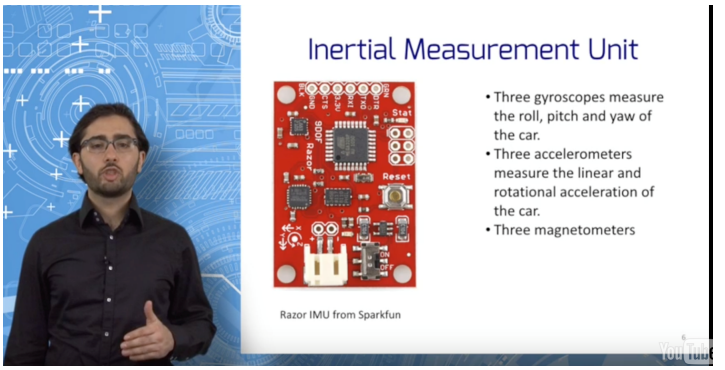


Fig. 3. F1 Tenth - Learning Material

2.1.6 *Benefit for Students.* F1 Tenth provides a vast source of documentation and tutorials.

Unique Solutions for Autonomous Driving

The F1/10 Competition will give students an incomparable meeting of the minds, all focused on the algorithmic problems presented by self-driving cars. You can expect to see a diverse portfolio of solutions to long-standing autonomy problems, such as static and dynamic obstacle collision avoidance, scene

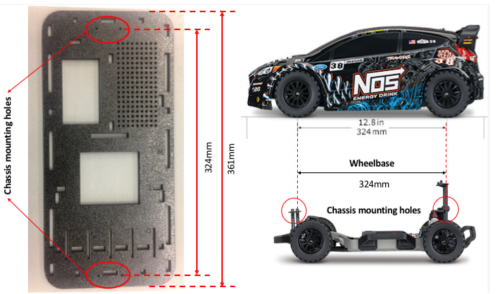


Fig. 4. F1 Tenth - Learning Material

understanding, obstacle interpretation, navigation of unmapped areas, and many others. F1/tenth provide an affordable way for college student to working on autonomous vehicle technology with all different kinds of sensors, as LIDAR and depth camera. we can implement these sensors to work on study and research for connected vehicle and advanced control algorithm.

Feedback on New Products and R and D

F1/10 allows for you to generate excitement and interest on your cutting-edge technology products and services, as well as opportunities to form partnerships with worlds leading research labs that specialize in autonomy and cyber-physical systems. Present your products and research findings for instant feedback from our base of highly knowledgeable participants on both hardware and software algorithms.

Recruit Highly Motivated Candidates

Participants are experts in control systems, robotics, embedded systems, machine learning, and software development. Your company will have direct access to top masters and PhD-level technical talent in autonomous driving from the worlds leading academic institutions. This event provides for the perfect opportunity to interact face-to-face with top talent from around the world and assess a specialized pool of quality candidates.

2.2 Turtle Bot

TurtleBot is a low-cost, personal robot kit with open-source software. TurtleBot was created at Willow Garage by Melonee Wise and Tully Foote in November 2010. The TurtleBot kit consists of a mobile base, 3D Sensor, laptop computer, and the TurtleBot mounting hardware kit. Since 2010, turtlebot has already develop 3 versions.

Fig. 5. Turtle Bot - Connectivity

2.2.1 *Overview.* The turtle bot models

TurtleBot 1 TurtleBot 1 consists of an iRobot Create base, a 3000 mAh battery pack, a TurtleBot power board with gyro, a Kinect sensor, an Asus 1215N laptop with a dual core processor, and a hardware mounting kit attaching everything together and adding future sensors. Assembling the kit is quick and easy using a single allen wrench (included in the kit).

TurtleBot 2 TurtleBot 2 consists of an Yujin Kobuki base, a 2200 mAh battery pack, a Kinect sensor, an Asus 1215N laptop with a dual core processor, fast charger, and a hardware mounting kit attaching everything together and adding future sensors. Assembling the kit is quick and easy using a single allen wrench (included in the kit). Additionally, TurtleBot 2 comes with a fast charging dock that TurtleBot can autonomously dock with, enabling continuous operation.

TurtleBot 3 Turtlebot 3, announced and developed in collaboration with ROBOTIS and OSRF, is the smallest and cheapest of its generation.[4] It has outstanding structural expansion capability due to ROBOTIS renown modular structure with the DYNAMIXEL. Turtlebot 3 will become available in 2 kits, the Turtlebot3 Burger and Turlebot3 Waffle.

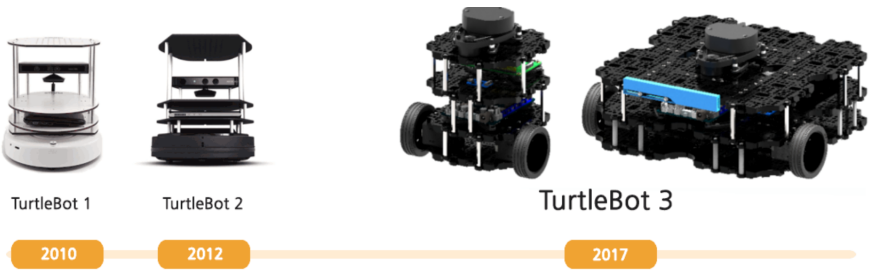


Fig. 6. Turtle Bot - Models

As shown in the above figure, the latest version of TurtleBot -TurtleBot3 is a small, low cost, fully programmable, ROS based mobile robot. It is intended to be used for the purpose of education, research, product prototyping, and hobby application.

The goal of the TurtleBot3 is to drastically reduce the size and the price of the platform without sacrificing capability, functionality, and quality. Since the additional options, such as the chassis, the computers, and the sensors, are available, the TurtleBot3 can be customized in various ways. The TurtleBot3 is willing to be in the center of the maker movement by applying the latest technical advances of the SBC (SingleBoard Computer), the Depth sensor, and the 3D printing.

2.2.2 *Features of Turtle Bot.* **Affordable cost** TurtleBot was built for the cost-conscious needs from the educations and the research and developments. TurtleBot3 is the most affordable robot among the SLAM-able mobile robots equipped with a general 360-degree LiDAR.

Small size The size of the TurtleBot3 Burger is 138mm x 178mm x 192mm (L x W x H). Its size is about 1/4 of the size of the TurtleBot 1, 2. Imagine the TurtleBot in your backpack and bring it anywhere.

ROS standard The TurtleBot brand is managed by Open Source Robotics Foundation, Inc. (OSRF), which develops and manages ROS. Nowadays, ROS has become the go-to platform for all the roboticists around the world. The TurtleBot obviously can be integrated by the existing ROS-based robots,

but basically can be an affordable platform for whom want to get started learning ROS. The simulation in Gazebo in shown as below

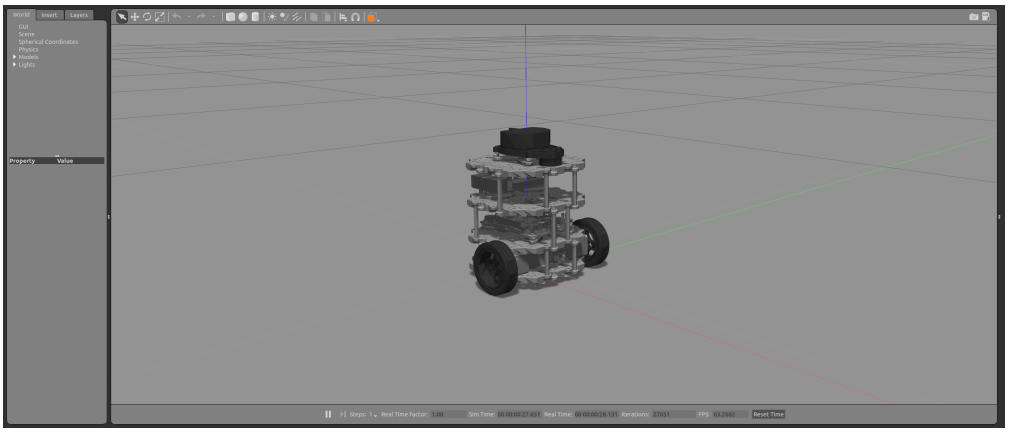


Fig. 7. Turtle Bot - Gazebo

Extensibility TurtleBot3 encourages users to customize its mechanical structure themselves using some alternative options: the open source embedded board (as a control board), the computer, and the sensor. The Turtlebot3 Burger is a two-wheeled differential drive type platform, but is able to be structurally and mechanically customized in many ways: The Cars, the Bikes, the Trailers and so on. Extend your ideas beyond imagination with various SBC, sensors, motors on a scalable structure.

Modular actuator for mobile robot TurtleBot3 allows to get precise spatial data by using 2 Dynamixels in the wheel joints. The Dynamixel XM series can be operated by one of 6 operating modes (XL series: 4 operating modes): the Velocity control mode for wheels, the Torque control mode or the Position control mode for joint, etc. The Dynamixel can be used even to make a mobile manipulator, since it is light but can be precisely controlled with the velocity, the torque and the position. The Dynamixel is a core component that makes the TurtleBot be completed. It is easy to assemble, maintain, replace and reconfigure.

Open control board for ROS The control board which is open-sourced in hardware wise and in software wise for ROS communication, or also called OpenCR, has the bandwidth to support not only for controlling the Dynamixel but also the ROBOTIS sensors which are frequently being used for basic recognition tasks in a cheaper way, such as Touch sensor, Infrared sensor, Color sensor and so on. It has a IMU sensor inside the board so that it can strengthen many precise controls. The board has a 3.3V, 5V, 12V power supplies to reinforce the available computer device lineups.

Strong sensor lineup TurtleBot3 Burger uses enhanced 360 LiDAR, 9-Axis Inertial Measurement Unit and precise encoder for your robot. The TurtleBot3 Waffle model is equipped with the 360 Lidar as well, but additionally proposes a powerful Intel RealSense with the recognition SDK. This will be the best solution for making a mobile robot.

Open source TurtleBot3s Hardware, Firmware, and Software are provided as open source. Basically, all components of the TurtleBot3 will be provided as the injection molded, and it achieves the low cost, but the CAD data for 3D printing is also available. The CAD data is released to the Onshape, which is a full-cloud 3D CAD editor. Get access through a web browser by using the computer or even by the portable devices. Here allows the works of drawing, assemblying with the co-workers. Besides, all details of the OpenCR board that are necessary for the users who want to make it by themselves, including the Schematics, the PCB Gerber, the BOM and the Firmware source are fully opened under the open-source licenses for the user and the ROS community. You can modify the downloaded source code, hardware and share it with your friends.

2.3 MIT - Race Car

Rapid Autonomous Complex-Environment Competing Ackermann-steering Robot

2.3.1 *Introduction.* The RACECAR is an open-source powerful platform for robotics research and education. The platform houses state-of-the-art sensors and computing hardware, placed on top of a powerful 1/10-scale mini race car. The design and development of the RACECAR a joint effort between the BeaverWorks Initiative of the Lincoln Laboratory, the Department of Aeronautics and Astronautics, and the Laboratory for Information and Decision Systems at the Massachusetts Institute of Technology

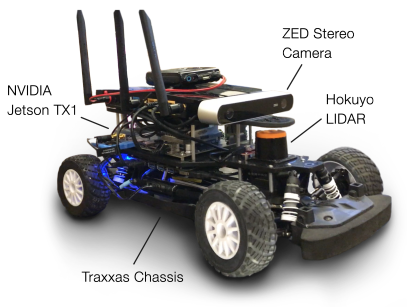


Fig. 8. MIT - Race Car

2.3.2 *Hardware.* The RACECAR platform is based on the 110-scale platform and the *Nvidia Jetson TK1* embedded super-computer. The vehicle includes the following sensors:

—*Hokuyo UST-10LX laser range finder*

—*Stereolabs ZED stereo camera*

—*Structure.io depth camera*

—*Sparkfun IMU*

The vehicles are equipped with an open-source Electronic Speed Control unit called the *VESC*

2.3.3 *Software.* The RACECAR platform runs on the Robot Operating System (ROS). The software for the RACECAR platform is available as open source software here: https://github.com/mit-racecar/racecar

The software for the VESC is available here: https://github.com/mit-racecar/vesc

2.3.4 *Uniqueness.* Visual odometry using stereo camera

2.4 MIT Tunnel Racing

It was a new activity offered during Januarys Independent Activities period in 2015. It showcased fully autonomous 1:10-scale model cars navigating through MIT's underground tunnels in a robot race. Four teams were provided with an assembled robot and basic software architecture. Each team was tasked with designing, implementing, and testing autonomy algorithms to guide the vehicle through MITs tunnels at high speeds while avoiding contact with walls, doorways, and other obstacles. The robot chassis was based on a 1:10-scale radio-controlled racecar modified to accept onboard control of its steering and throttle actuators. To perceive its motion and the local environment, the robot was outfitted with a heterogeneous set of sensors, including a scanning laser range finder, camera, inertial measurement unit, and visual odometer. Sensor data and autonomy algorithms were processed on board with an NVIDIA Jetson Tegra K1 embedded computer.

A key challenge for the students was developing algorithms capable of crunching the data from all sensors in real time and making quick decisions to accelerate, brake, or turn the car without losing control. Each team pursued its own solution, although most converged on reactive planning approaches using the laser ranger to track its position relative to corridor walls.

The class featured seven lectures on algorithmic robotics. Topics included the robot operating system; selected topics in algorithmic robotics, such as sensing, perception, control, and planning algorithms; and several case studies. The lectures were followed by a two-day hackathon in which most of the software development took place.

2.5 AutoRally Georgia Tech

The *AutoRally* platform is a high-performance testbed for advanced perception and control research. The robot, developed at Georgia Tech, is integrated with ROS and designed as a self-contained system that requires no external sensing or computing. The robot is a robust, cost-effective, and safe platform that opens the space of aggressive autonomous off-road driving to researchers and hobbyists.

The core software and simulation environment for the platform are publicly available along with tutorials. Build *instructions* for the chassis and compute box, complete parts list, CAD models, and operating procedures are released in a separate GitHub repository.

Fig. 9. AutoRally Georgia Tech

GitHub repository for codes: https://github.com/AutoRally/autorally

2.5.1 *Sensor Suite*

—High-precision IMU, raw data up to 1KHz

—RTK-corrected GPS, position at 20Hz

—Hall-effect rotation sensor on each wheel at 70Hz

—2 front facing machine vision cameras, 1280x1024, 70fps, global shutter, synchronously triggered

2.5.2 *Computing*

—Intel Skylake Quad-core i7
—32GB DDR4 RAM
—512GB m.2 SSD
—1TB SATA3 SSD
—Nvidia GTX 750ti GPU
—WiFi and XBee communication
—6s 11000mAh battery gives typical run time of 3 hours, full-load run time 1 hour
—DC power supply
—Dynamically switching power selection between internal battery and external power
—Robust fabricated aluminum enclosure protects sensitive electronics from shock
—Standardized mounting and communication interface to chassis

2.5.3 *Software*

—ROS-compatible, open-source core code
—State estimation by fusing IMU and GPS using GTSAM optimization toolbox
—Flexible launch system to pair any compute box with any chassis
—Operator Control Station (OCS) for remote monitoring
—GPS waypoint steering controller
—Constant velocity controller

2.5.4 *Simulation*

—Gazebo-based
—Identical ROS interface as physical robot
—Vehicle model parameters measured from physical robot

2.5.5 *Uniqueness*

—No LIDAR
—Stereo Camera
—RTK-corrected GPS
—Off road Autonomous Vehicle

Build Instructions: https://github.com/AutoRally/autorally_platform_instructions
Code and Software tutorials: https://github.com/AutoRally/autorally

3. METHODS

The Robot Operating System (ROS) was the programming environment used in development of the self-driving f1tenth car. ROS provides a myriad tools and libraries allow for developing robotic applications. It is described by its architects as a meta-operating system allowing data from multiple sensors to be obtained and managing communication between applications. A more detailed introduction of ROS can be found at http://wiki.ros.org/. In the ROS workspace applications are referred to as nodes and the data they receive or broadcast are referred to as messages. Message passing between nodes are set up on a network basis where a master node, ROS Master, runs on a static IP and sets up communication between nodes on a network. In this network, messages or data are broadcast or received

by a node defined under a name referred to as a topic. This allows the ROS Master to establish communication to the appropriate nodes. Nodes that broadcast data are defined in the network as publishers to a particular topic and nodes that receive data are defined as subscribers to a particular topic. In applications such as simultaneous localization and mapping, transforms between coordinate frames are usually broadcasted over the network at some defined rate. This broadcast can be received by any nodes listening for the transformations over the network; the broadcast is not defined under a topic name. Among the tools provided by ROS are applications which allow for visualizing data and application communication and broadcasting information. RViz (standing for ROS Visualization) is a node which provides a GUI for subscribing to different topics and visualizing the data. Other tools for debugging such as RQt Graph and view_frames provide visualizations for communication networks between nodes or broadcasts. ROS provides libraries for programing nodes in C++ and Python. Associated GUI tools can be used to view and broadcast data. An example is provided in Python. MATLAB provides a Robotics Systems Toolbox (RST) which allows for programming nodes in ROS. This allows for rapid prototyping of ROS nodes which can later be written C++ or Python. Many examples of MATLAB/RST will be provided and how it was used for rapid prototyping. ROS also interfaces with Gazebo simulator where simulated data can be published to topics on a ROS network. This allows for testing ROS nodes in simulation before realizing its use on a real system with real-time sensor data.

This section of the report will present processes which were utilized to develop the final product which will be discussed in the Implementation section. These processes highlight real procedures showing how developing in ROS can be enhanced with the use the aforementioned tools. This also illustrates the invaluable impact that these tools make in concert in the developing process. This section will continue with presentation of an MITs Gazebo model of a race car which was useful for testing the initial iteration of the race car navigation controller. Afterwards Hector SLAM and RViz will be presented. The view_frames and RQt Graph debugging tools will also be presented. A procedure using OpenSSH server will also be mentioned as it allowed for remotely interfacing with the TK1. Afterwards MATLAB/RST will be presented followed by Python and Python GUI which led to the final implementation.

3.1 Gazebo - Racecar Simulation

The RACECAR (Rapid Autonomous Complex-Environment Competing Ackermann-steering Robot) group at MIT develop race car model in Gazebo. This model was included linkages for the Ackermann steering system and include a laser scanner with a 270-degree field of view (FOV). A Gazebo model of the MIT tunnel was also constructed. The Gazebo models along with servo command and safety nodes were organized into RACECARs Racecar Simulator Git repository on Github at https://github.com/mit-racecar/racecar-simulator. Figure 10 displays the Gazebo model of the race car set within the MIT tunnel.

Instructions for running the Gazebo model are included in the README file of the repository on Github. From Figure 10, the rays of the laser data are displayed as blue lines. This is part of the visual feature in Gazebo which can be set via a parameter in race model file. The visualization allows the laser observations of a given environment or structure to modeled according to the FOV and maximum range of the laser sensor. This can be seen in shape of the laser FOV changing due in the walls and structures in range. The version of Gazebo shown in Figure 10 is Gazebo 7. This is a version affiliated with ROS Kinetic. The visualization may not work show on other versions of Gazebo such as Gazebo 2. The Gazebo 2 is affiliated with ROS Indigo.

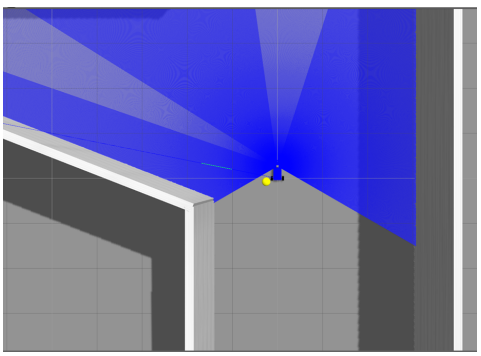


Fig. 10. MIT's gazebo Model of a Race Car in the MIT Tunnel

3.2 RViz - Hector SLAM Application

The procedure for implementing Hector SLAM is shown in the implementation section. This section will present simulated results for Hector SLAM to illustrate the uses of RViz as a visualization and debugging tool. Figure 11 shows snap shots of a Hector SLAM result being from played rosbag files supplied by Upenn. Note that map construction can be viewed as the robot is navigating through a floor. This allows an analysis on the performance of the SLAM algorithm to be conducted. Trouble areas or maneuvers that cause the SLAM algorithm to become lost can be viewed. The trajectory of the robot base can also be displayed.

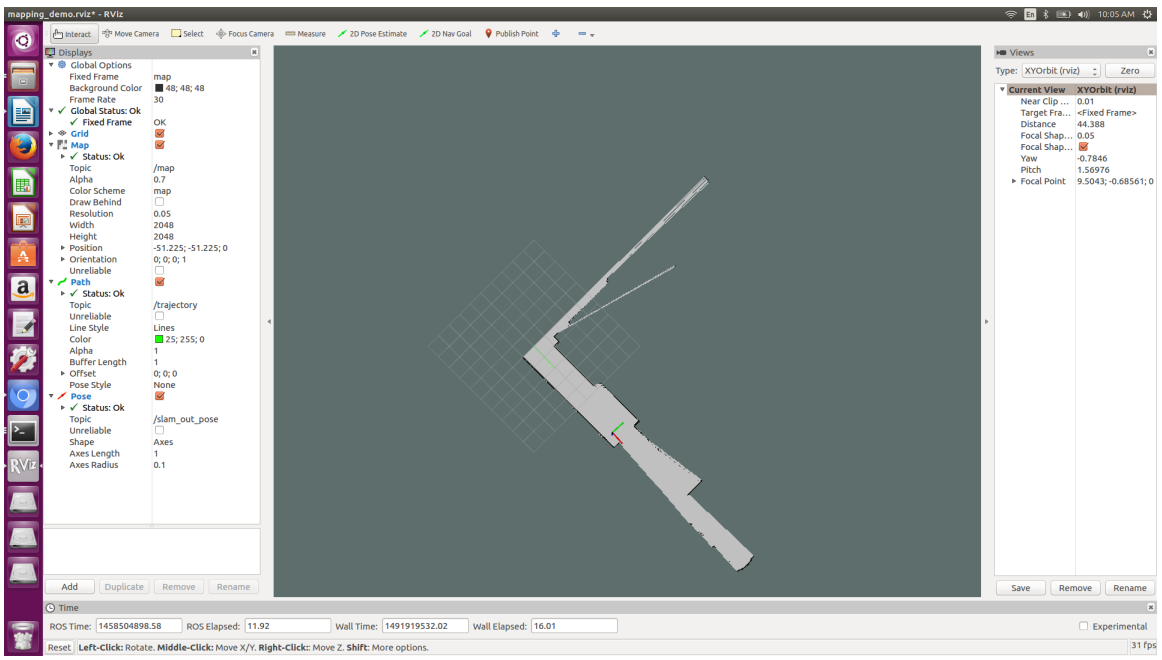


Fig. 11. Hector SLAM Mapping

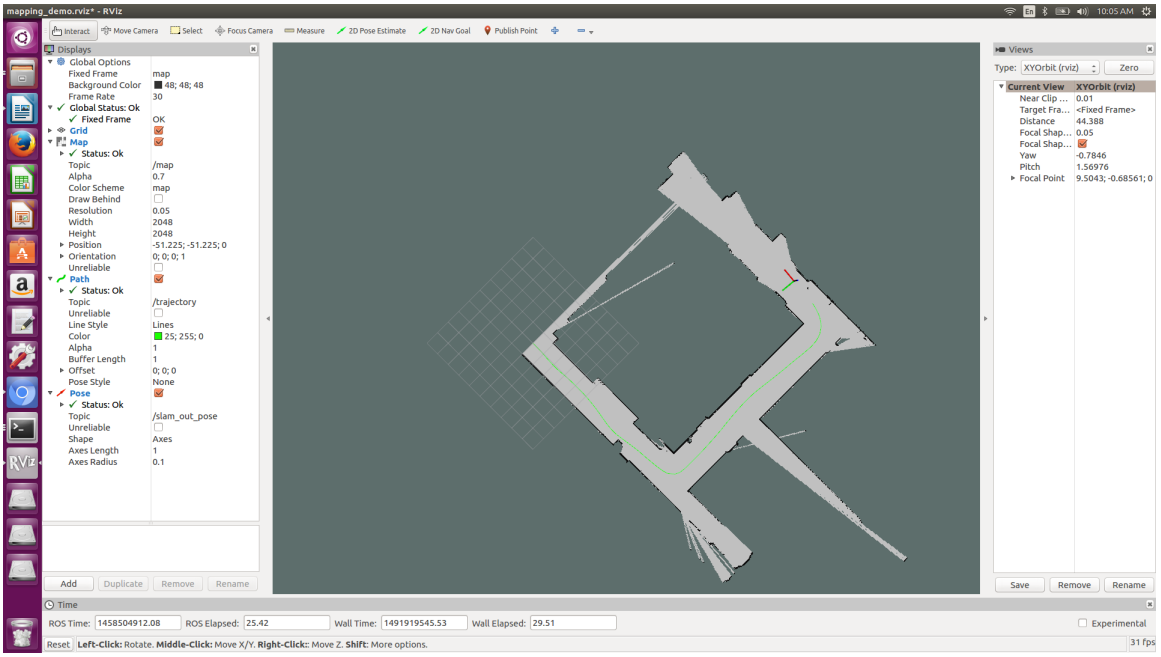


Fig. 12. Hector SLAM Mapping

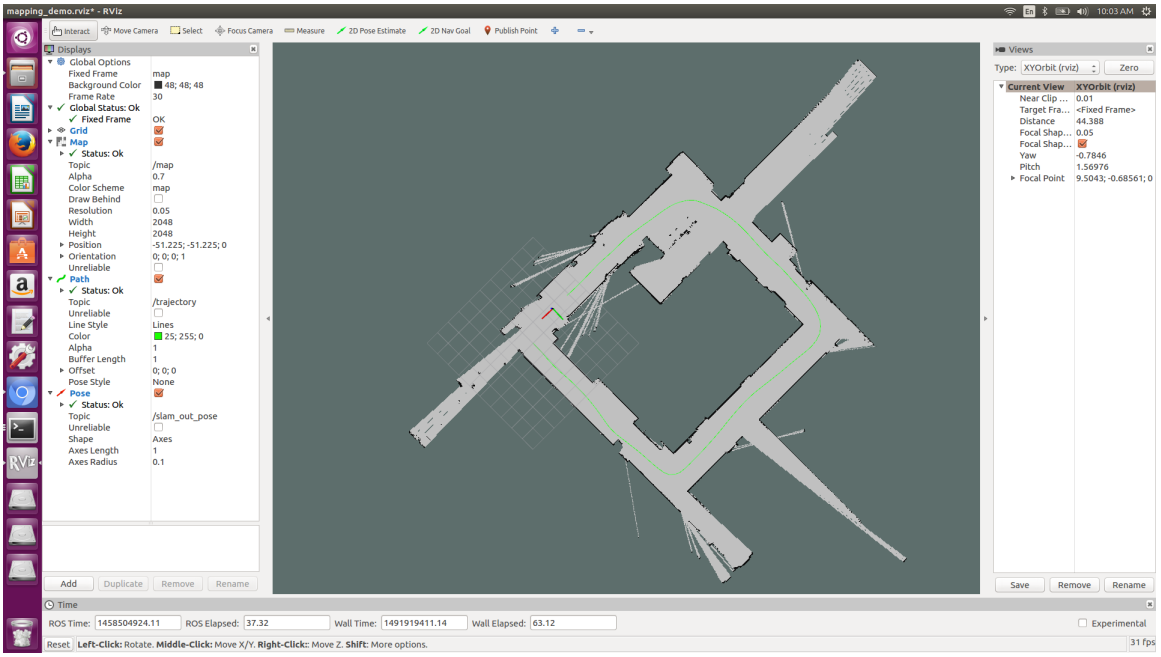


Fig. 13. Hector SLAM Mapping

Sensor data obtained in Gazebo can also used to build map of the simulated environment through SLAM. This allows localization algorithms and path planning algorithms to be tested in simulation before being tested on a real system. A good resource for exploring this functionality can be found at http://learn.turtlebot.com/2015/02/03/8/ using Turtlebot Gazebo.

In the event that the transform tree is not set properly and there is missing transformation between coordinate frames, the view_frames debugging tool under the tf package can be used determined which transformations are missing. This is can be used by opening a terminal and running the following statement: rosrun tf view_frames. This node analyzes the transformations being broadcast and saves a visual depiction of the transformation broadcasts along with their associated coordinate frames. Broadcast statistics are also shown. This is shown in Figure 14 where Hector SLAM was not able to broadcast a transform between the robot base. This allowed a problem in frame coordinate frame parameters to be identified.

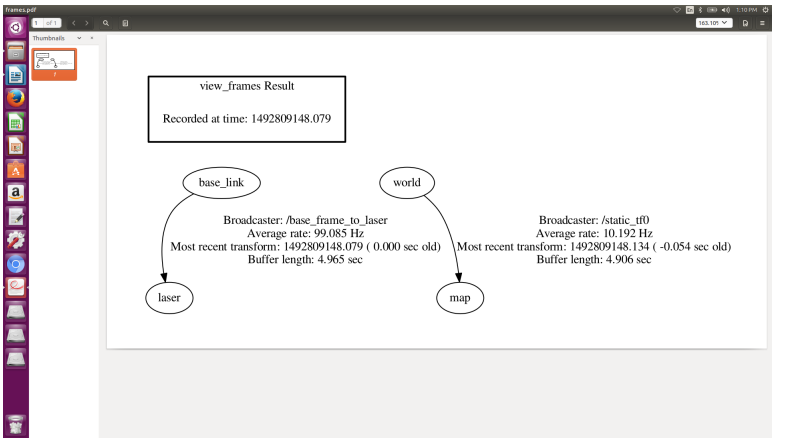


Fig. 14. Results of view frames on an Improper Hector Slam setup

3.2.1 *Debugging SLAM Transformation Problems with Tf view_frames.*

3.2.2 *Using RQt Graph to Analyze a ROS Network.* RQt Graph can be used to monitor the publishing and the subscribing of messages between nodes. It is a great tool for debugging launch files and a for understanding example code provided by the research community. An overview of ROS RQt Graph is provided at http://wiki.ros.org/rqt_graph. RQt Graph was used to understand the MIT gazebo model from https://github.com/mit-racecar/racecar-simulator.git and how it responds to a wall follower node of a ta_lab3 package from https://github.com/mit-racecar/TA_example_labs.git. The MIT gazebo model and world and the wall following controller was launched. ROS RQt Graph was then run to view the communication network of ROS nodes called by the two launch files. RQt Graph can be used to monitor the publishing and the subscribing of messages between nodes. It is a great tool for debugging launch files and a for understanding example code provided by the research community. An overview of ROS RQt Graph is provided at http://wiki.ros.org/rqt_graph. RQt Graph was used to understand the MIT gazebo model from https://github.com/mit-racecar/racecar-simulator.git and how it responds to a wall follower node of a ta_lab3 package from https://github.com/mit-racecar/TA_example_labs.git. The MIT gazebo model and world and the wall following controller was launched. ROS RQt Graph was then run to view the communication network of ROS nodes called by the two launch files.

```
Open a terminal
one tab
roslaunch racecar_gazebo racecar_tunnel.launch

another tab
roslaunch ta_lab3 example.launch

another tab
rosrun rosrun rqt_graph rqt_graph
```

The RQt Graph is shown in Figure 14. From the Figure 5 the topic which their base controller subscribes to is displayed. The wall follower node publishes to the /vesc/high_level/ackermann_cmd_mux/input/nav_0. Running rostopic info /vesc/high_level/ackermann_cmd_mux/input/nav_0 shows that its message type is ackermann_msgs/AckermannDriveStamped.msg. It then be determined through observations of of the code in RQt Graph that a node can be written to publish AckermannDriveStamped msgs to control the gazebo race car developed by MIT.

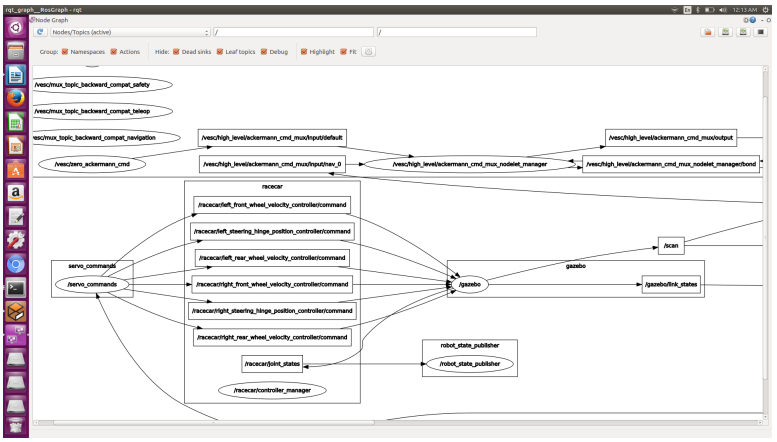


Fig. 15. RQt Graph Result MIT's Simulation and Wall Following Code

3.3 Using OpenSSH and Extending ROS Networks to Multiple Computer Systems

In testing the sensor ROS nodes and slam code on the TK1, it is advantageous to make use of data viewing software on a remote PC. The viewing software available for ROS nodes are RViz and MATLAB with RST toolbox. Additionally it is also advantageous to view the message publishing and subscribing through RQT graph. This becomes especially useful when managing the nodes which are run from a launch file. The TK1 comes with OpenSSH server and is accessible from a remote systems. In order to make use of software on a remote PC a procedure must be followed to allow both the TK1 and remote PC to be connected to one ROS MASTER over a network. ROS MASTER is responsible setting up a communication networks between ROS nodes. ROS MASTER runs on a static port computer_ip-11311 sets up network where its gives ROS nodes IDs and locations. With the publishing and subscribing information provided by the ROS nodes, it sets up hand shaking communication between nodes. For viewing data from sensors running on the TK1 on a remote PC it is best to run roscore on the TK1 and for the remote PC ROS nodes to be connected to the ROS MASTER of the TK1. A procedure is given

for viewing the structure camera data on RViz or MATLAB running on a remote PC. (OpenNI2 drivers are used to get data from the structure sensor).

```
In the TK1
Open a terminal
one tab
ssh -X ubuntu@10.42.0.225
roscore

another tab
ssh -X ubuntu@10.42.0.225
export ROS_HOSTNAME=10.42.0.225
roslaunch openni2_launch openni2.launch

On the Remote PC
Open another terminal
For rviz:
export ROS_MASTER_URI=http://10.42.0.225:11311/
rosrun rviz rviz

For Matlab:
matlab (run Matlab)
rosinit('http://10.42.0.225:11311')
run depth_viewer.m file
```

3.4 MATLAB/RST Rapid Prototyping of ROS Nodes

MATLAB/RST serves as a convenient environment for quickly developing ROS nodes. It was used in this project to quickly develop and test tele-operation nodes which publish messages for robot base controllers such as twist or ackermann messages. One convenient tool provided by Adhiti Raman was a MATLAB/RST keyboard teleop node for twist messages. This MATLAB file was easily modified to construct a keyboard teleop node which published ackermann messages. This ackermann keyboard teleop code is shown below in Figure 16 below. This node allows you to set a speed and steering angle.

Illustrated in the first line of code in Figure 16 is the convenient feature of MATLAB/RST is the ability to pass a topic to the rospublisher function to create a publisher object for that topic. This object could then be passed to AckermannTeleop to develop a node which publishes ackermann messages in response to keyboard input. MATLAB/RST can handle the message class imports automatically in the background. This also illustrated further down the code with the message call velMsg = rosmessage(velPub);. A message object is generated by the rosmessage() function by passing the publisher object, velPub, which has the message type stored as a member in the struct.

Similar to rospublisher(), a topic name can be passed to the rossubcriber() in order to generate a subscriber object for that particular topic. The message for that topic can then be passed by passing the subscriber object to the receive() function. This is illustrated in Figure 16 where the depth image of structure sensor subscribed from the /camera/depth/image topic.

```
vel_publisher = rospublisher('/vesc/high_level/ackermann_cmd_mux/input/nav_0');
AckermannTeleop(vel_publisher,0.5,1)

function AckermannTeleop(velPub,forwardVel,turnVel)

   % Modified MATLAB example code to publish only velocity messages and
   % nothing else.

   persistent velMsg


   keyObj = ExampleHelperTurtleBotKeyInput();

   disp('Keyboard Control: ');
   disp('i=forward, k=backward, j=left, l=right');
   disp('q=quit');
   disp('Waiting for input: ');

   % Use ASCII key representations for reply, initialize here
   reply = 0;

   while reply~='q'

      forwardV = 0;   % Initialize linear velocities and steering angle
      turnV = 0;

      reply = getKeystroke(keyObj);
      switch reply
         case 'i'        % i
            forwardV = forwardVel;
         case 'k'     % k
            forwardV = -forwardVel;
         case 'j'     % j
            turnV = turnVel;
         case 'l'     % l
            turnV = -turnVel;
      end

      if isempty(velMsg)
         velMsg = rosmessage(velPub);
      end

      velMsg.Drive.Speed = forwardV;
      velMsg.Drive.SteeringAngle = turnV;

      send(velPub,velMsg);

      pause(0.1);

   end

   closeFigure(keyObj);

end
```

Fig. 16. Keyboard Teleop for Ackerman Message

```
image_sub = rossubscriber('/camera/depth/image');

figure
while 1
   imgdata = receive(image_sub);
   img = readImage(imgdata);
   imshow(img)
end
```

It is important to note from the Figure above that the data is not accessed, manipulated, stored through a call back function that is defined within the code. The message data struct was accessed through a function return with a single line of code. Manipulating the data or displaying the data could then be accomplished down line line of script. From Figure 16 and Figure 17 RST is shown to not only facilitates an interface with the ROS environment through MATLAB this interface maintains the general flow and programming style of MATLAB programming languages.

MATLAB/RST was used to develop and test a pure pursuit algorithm first in simulation on the MIT gazebo race car and afterwards on the f1tenth car. As previously mentioned, the MIT simulator of the robot base controller processed commands according to the ackermann_msgs/AckermannDriveStamped.msg format. The f1tenth race car controller interpreted commands according to the geometry_msgs/Twist.msg format. With MATLAB/RST the pure pursuit algorithm can quickly be tested on both race cars by simply changing the topics, message type, and message struct members. The code for pure pursuit with ackermann messages and pure pursuit with twist messages is shown in Figure below.

Figure 17 was the code used to control the f1tenth race car. This code was used to perform an initial test for the pure pursuit method. By initializing the RST node using rosinit('http://tegra-ubuntu:11311'), the ROS MASTER of the TK1 could set up RST as node on the network. The teensy board could then read twist messages sent from pursuit node and control the race car. Through the use of MATLABs uicontrol() function a sliding bar GUI on a touch screen interface could be used to rapidly tune the gains of the controller while observing the behavior of the vehicle on the race track. There were minor latency issues due to the control commands being sent over WiFi. This effect was minor and only seem to present more of a challenge in certain instances. The minor intermittent effects were outweighed by benefits of rapid prototyping with MATLAB, bring together the MATLAB tools within a node on the ROS environment. It is important, however, to note other issues concerning the use of MATLAB. The Linux version of MATLAB is not very stable. Eventually it may crash and need to be restarted. It is also challenging to run MATLAB and Gazebo together. One application at times crashes the other while attempting to start both of them. The rapid prototyping capability of MATLAB, however led to obtaining valuable insights for developing and testing an onboard controller in Python.

3.4.1 *Python and Pytho GUI-Pure Pursuit Application.* The pure pursuit node controlling the f1tenth race car was converted into a Python node to run on the TK1. The code was designed to have initialized gain settings for the pure pursuit controller. The code was also designed to allow for tuning of the gains from a remote PC. A gain setter node was developed to allow gains to be set through a GUI interface on the PC. The gains were published as a string message on a gain topic subscribe by the pure pursuit Python node. The Python node then parsed the string message to obtain the gain values. In this way there is no longer a latency concerning controller commands. The task is reduced to sending gain commands for which latency due to Wi-Fi communications of no consequence. The code for the pure pursuit Python code running on the TK1 and the gain setter code run on the remote PC is present in Figure 19 and Figure 20 respectively.

```
vel_publisher = rospublisher('/vesc/high_level/ackermann_cmd_mux/input/nav_0');
AckermannTeleop(vel_publisher,0.5,1)

function AckermannTeleop(velPub,forwardVel,turnVel)

  % Modified MATLAB example code to publish only velocity messages and
  % nothing else.

  persistent velMsg


  keyObj = ExampleHelperTurtleBotKeyInput();

  disp('Keyboard Control: ');
  disp('i=forward, k=backward, j=left, l=right');
  disp('q=quit');
  disp('Waiting for input: ');

  % Use ASCII key representations for reply, initialize here
  reply = 0;

  while reply~='q'

    forwardV = 0;   % Initialize linear velocities and steering angle
    turnV = 0;

    reply = getKeystroke(keyObj);
    switch reply
      case 'i'        % i
        forwardV = forwardVel;
      case 'k'     % k
        forwardV = -forwardVel;
      case 'j'     % j
        turnV = turnVel;
      case 'l'     % l
        turnV = -turnVel;
    end

    if isempty(velMsg)
      velMsg = rosmessage(velPub);
    end

    velMsg.Drive.Speed = forwardV;
    velMsg.Drive.SteeringAngle = turnV;

    send(velPub,velMsg);

    pause(0.1);

  end

  closeFigure(keyObj);

end
```

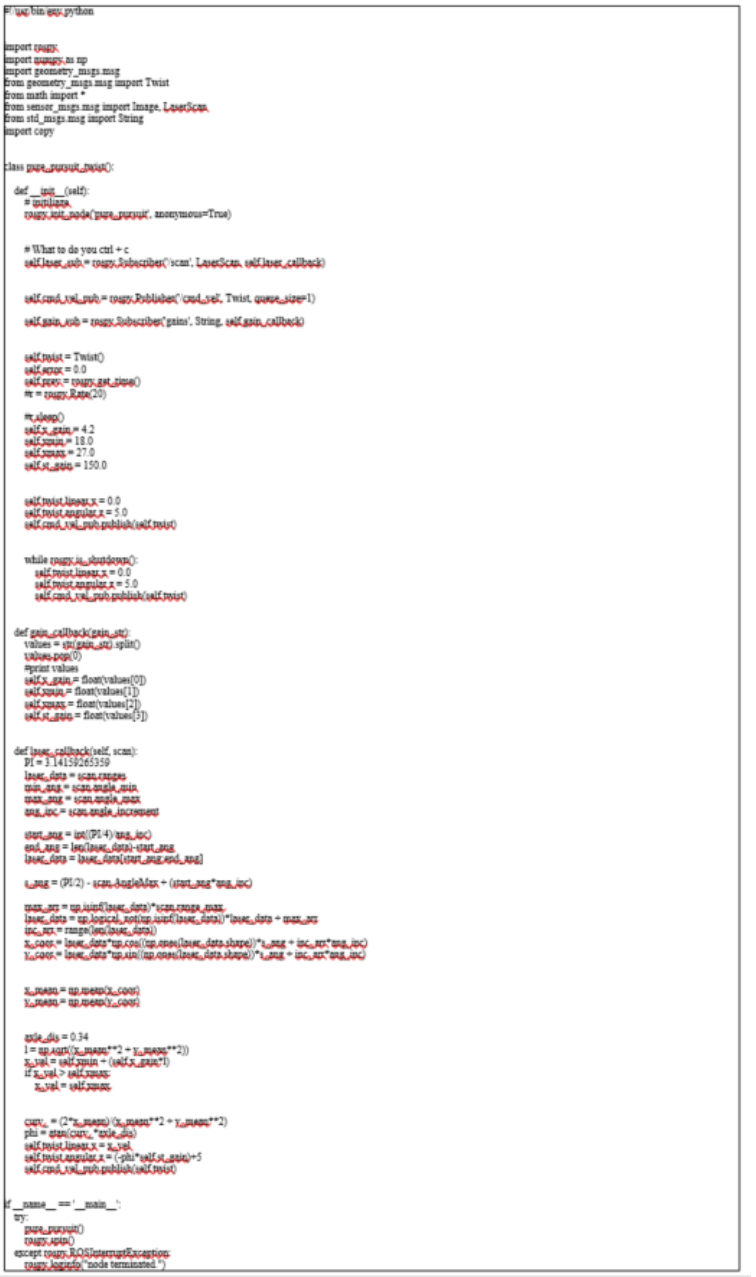
Fig. 17. Pure Pursuit Code with Ackermann (a) and Twist (b) Message Commands

The sliding bar GUI of the gain setter code is shown in Figure 21. The code is making use of the OpenCV Trackbar application to provide a adjustable sliding bar for the gains on the screen. This trackbar application is designed image processing integer positions to be set. For this reason the ranges for the gains were set to be order one or two orders of magnitude larger and then scaled to achieve

```
vel_publisher = rospublisher('/cmd_vel','geometry_msgs/Twist');
laser_sub = rossubscriber('/scan','sensor_msgs/LaserScan');
H = uicontrol('Style', 'slider', 'Min', 0.0, 'Max', 5.0, 'Position',[50 300 400 40]);
H2 = uicontrol('Style', 'slider', 'Min', 0.0, 'Max', 20.0, 'Position',[50 225 400 40]);
H3 = uicontrol('Style', 'slider', 'Min', 0.0, 'Max', 28.0, 'Position',[50 150 400 40]);
H4 = uicontrol('Style', 'slider', 'Min', 0.0, 'Max', 150.0, 'Position',[50 75 400 40]);
%%
while (ishandle(H))
  x_gain = get(H, 'value')
  xmin = get(H2, 'value')
  xmax = get(H3, 'value')
  st_gain = get(H4, 'value')+100
%    t1=text(.2,.95,num2str(st_gain));
%    t2=text(.2,.6,num2str(xmin));
%    t3=text(.2,.2,num2str(xmax));
  vel_msg = rosmessage(vel_publisher);



  laser_sub = rossubscriber('/scan','sensor_msgs/LaserScan');
  laser_msg = receive(laser_sub);
  laser_data = laser_msg.Ranges;
  ang_inc = laser_msg.AngleIncrement;
  start_ang = floor((pi()/4)/ang_inc);
  end_ang = length(laser_data)-start_ang;
  laser_data=laser_data(start_ang:end_ang);

  s_ang = (pi()/2) - laser_msg.AngleMax + (start_ang*ang_inc);
  x_coor = zeros(length(laser_data),1);
  y_coor = zeros(length(laser_data),1);
  for i = 1:length(laser_data)
    if laser_data(i) == Inf

      mag = laser_msg.RangeMax;

    else

      mag = laser_data(i);


    end
    x_coor(i) = mag*cos(s_ang + (i*ang_inc));
    y_coor(i) = mag*sin(s_ang + (i*ang_inc));
  end
  axle_dis = 0.34;
  x_mean = mean(x_coor);
  y_mean = mean(y_coor);
  l = sqrt((x_mean^2 + y_mean^2));
  x_vel = xmin + (x_gain*l);
  if x_vel > xmax
    x_vel = xmax;
  end
  x_vel
  curv_ = (2*x_mean)/(x_mean^2 + y_mean^2);
  phi = atan(curv_*axle_dis);
%%
  vel_msg.Linear.X = x_vel;
  vel_msg.Angular.Z = (-phi*st_gain)+5;
  send(vel_publisher,vel_msg);
end
%%
vel_msg.Linear.X = 0.0;
vel_msg.Angular.Z = 5.0;
send(vel_publisher,vel_msg);
```

Fig. 18. Pure Pursuit Code with Ackermann (a) and Twist (b) Message Commands

higher precision values. The orders of magnitude the gain values were scaled by were listed on the label next to each trackbar.



```
#! /usr/bin env python

import rospy
import numpy as np
import geometry_msgs.msg
from geometry_msgs.msg import Twist
from math import *
from sensor_msgs.msg import Image, LaserScan
from std_msgs.msg import String
import copy


class pure_pursuit_twist():

  def __init__(self):
    # initialize
    rospy.init_node('pure_pursuit', anonymous=True)

    # What to do you ctrl + c
    self.laser_sub = rospy.Subscriber('scan', LaserScan, self.laser_callback)

    self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)

    self.gain_sub = rospy.Subscriber('gains', String, self.gain_callback)

    self.twist = Twist()
    self.error = 0.0
    self.prev = rospy.get_time()
    r = rospy.Rate(20)

    r.sleep()
    self.x_gain = 4.2
    self.vmin = 18.0
    self.vmax = 27.0
    self.w_gain = 150.0

    self.twist.linear.x = 0.0
    self.twist.angular.z = 5.0
    self.cmd_vel_pub.publish(self.twist)

    while rospy.is_shutdown():
      self.twist.linear.x = 0.0
      self.twist.angular.z = 5.0
      self.cmd_vel_pub.publish(self.twist)

  def gain_callback(gain_str):
    values = str(gain_str).split()
    values.pop(0)
    #print values
    self.x_gain = float(values[0])
    self.vmin = float(values[1])
    self.vmax = float(values[2])
    self.w_gain = float(values[3])

  def laser_callback(self, scan):
    PI = 3.14159265359
    laser_data = scan.ranges
    min_ang = scan.angle_min
    max_ang = scan.angle_max
    ang_inc = scan.angle_increment

    start_ang = int((PI/4)/ang_inc)
    end_ang = len(laser_data)-start_ang
    laser_data = laser_data[start_ang:end_ang]

    t_ang = (PI/2) - scan.angle_min + (start_ang*ang_inc)

    max_arr = np.isinf(laser_data)*scan.range_max
    laser_data = np.logical_not(np.isinf(laser_data))*laser_data + max_arr
    inc_arr = range(len(laser_data))
    x_coor = laser_data*np.cos((np.ones(laser_data.shape))*t_ang + inc_arr*ang_inc)
    y_coor = laser_data*np.sin((np.ones(laser_data.shape))*t_ang + inc_arr*ang_inc)


    x_mean = np.mean(x_coor)
    y_mean = np.mean(y_coor)


    axle_dis = 0.34
    l = np.sqrt(x_mean**2 + y_mean**2)
    x_vel = self.vmin + (self.x_gain*l)
    if x_vel > self.vmax:
      x_vel = self.vmax


    curv = (2*x_mean)/(x_mean**2 + y_mean**2)
    phi = atan(curv * axle_dis)
    self.twist.linear.x = x_vel
    self.twist.angular.z = (-phi*self.w_gain)+5
    self.cmd_vel_pub.publish(self.twist)


if __name__=='__main__':
  try:
    pure_pursuit()
    rospy.spin()
  except rospy.ROSInterruptException:
    rospy.loginfo("node terminated.")
```

Fig. 19. Pure Pursuit and Gain Setting

It is important to note that the GUI had to be run in the same loop of the publisher. The loop had to be run without setting a rate so that the GUI wouldnt crash. Several attempts were made to run the GUI separate from a ROS publisher at its own rate. One attempt was made to run the GUI on a separate thread from the ROS publisher and then share the values over the same process. Another attempt was

```
import rospy
from std_msgs.msg import String
import cv2
import numpy as np


def nothing():
    pass


def setter():
  pub = rospy.Publisher('gains', String, queue_size=10)
  rospy.init_node('setter', anonymous=True)
  # Create a black image, a window
  #img = np.zeros((300,512,3), np.uint8)
  cv2.namedWindow('image')


  # create trackbars for color change
  cv2.createTrackbar('x_gain*1E2','image',0,1000,nothing)
  cv2.createTrackbar('xmin*1E1','image',0,220,nothing)
  cv2.createTrackbar('xmax*1E1','image',0,300,nothing)
  cv2.createTrackbar('st_gain*1E1','image',0,3000,nothing)


  # create switch for ON/OFF functionality
  #switch = '0 : OFF \n1 : ON'
  #cv2.createTrackbar(switch, 'image',0,1,nothing)


  while(1):
    #cv2.imshow('image',img)
    k = cv2.waitKey(1) & 0xFF
    if k == 27:
      break


    # get current positions of four trackbars
    x_gain = cv2.getTrackbarPos('x_gain*1E2','image')/100.0
    xmin = cv2.getTrackbarPos('xmin*1E1','image')/10.0
    xmax = cv2.getTrackbarPos('xmax*1E1','image')/10.0
    st_gain = cv2.getTrackbarPos('st_gain*1E1','image')/10.0

    gain_str = str(x_gain) + " " + str(xmin) + " " + str(xmax) + " " + str(st_gain)
    hello_str = "hello world %s" % rospy.get_time()
    rospy.loginfo(gain_str)
    pub.publish(gain_str)


  cv2.destroyAllWindows()


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

Fig. 20. Pure Pursuit and Python Code

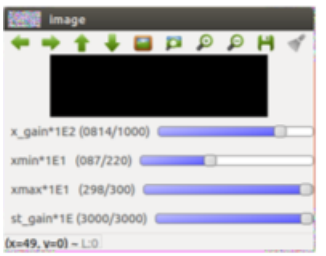


Fig. 21. Gain Setter Slider GUI

to run the GUI on a separate process from the publisher while the gain values were stored on a shared memory space. Each resulted in the GUI preventing the publisher from running. The resulting code however suited the overall purpose in that it was ideal to have the set gains instantly published to

the pure pursuit controller on the TK1. This allow the gains to be tuned and the racer car behavior to be observed in real-time. The analysis of the vehicle behavior allowed the develop of the pure pursuit method used in the final implementation.

—ROS
  —Gazebo
  —R-Viz
—RST Toolbox Matlab
—Hector Slam

4. IMPLEMENTATION

4.1 System Overview

The autonomous system implemented in this project can be broken down into several categories. First there are the sensors which collect information about the environment in the form of data. This includes the IMU, the Lidar, and the structure camera. Next, there are the actuators which physically move the vehicle through the environment, following the requested control output. In this application, the actuators include the vehicles motor and steering systems. Finally, the control system uses the sensor data to make decisions and generate a set of actions to be performed by the actuators. The top level supervisory controller in this case is Nvidias Jetson TK1 embedded system which communicates with the host computer over a wireless network. The lower layer controller is the Teensy. Additionally, the system requires auxiliary equipment such as the Ubiquiti, the USB Hub, and the Energizer. As discussed previously, ROS provides an organized network of communication between the sensors, actuators, and controllers. In doing so, ROS simplifies the task of each sensor and actuator from communicating with a specific entity to communicating with ROS.

4.2 Sensors

4.2.1 *IMU.* The SparkFun 9DoF Razor IMU (Inertial Measurement Unit) M0 collects information about how the vehicle is moving through space. The nine degrees of freedom (DoF) come from three axis of acceleration, three axis of magnetometer, and three axis gyroscopic measurements. The accelerometer measures the magnitude and direction which the integrated circuit (IC) is linearly accelerating in 3D space. Additionally, the gyroscope measures changes in orientation, ie angular velocity. The magnetometer helps counter sensor drift by measuring the direction of magnetic north in 3D space. In this implementation, the magnetometer is critical to the yaw of the vehicle.

The Razor 9 DOF IMU is a plug and play sensor which can have the example firmware installed through the Arduino IDE, and be interfaced through serial communication. Additionally, the sensor can be configured by sending information to the IMU through the serial monitor as well. Certain characters change certain settings on the IMU which is stored with non-volatile memory (assuming you used the proper libraries when uploading the firmware). Because of this, configuring the IMU should only have to be done once, and all of the settings will be saved. Certain data can be omitted or included from being published as a serial message such as quaternion orientation, Euler orientation, magnetometer data, etc. While this sensor interfaces with serial communication out of the box, interfacing this sensor with ROS is slightly more challenging. While there is a supplied ROS package for IMU, it does not seem to support the Razor 9DOF IMU

The zero yaw is represented by orienting the X-axis (mentioned in Figure above) with the north pole. Zero roll is obtained by maintaining the z-axis perpendicular to the ground plane. Zero pitch is observed when the IMU pane is parallel to the ground plane.

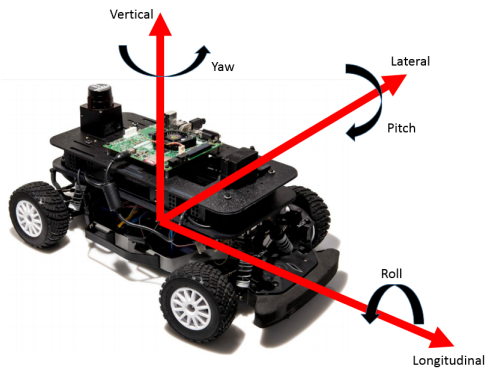


Fig. 22. IMU Coordinates

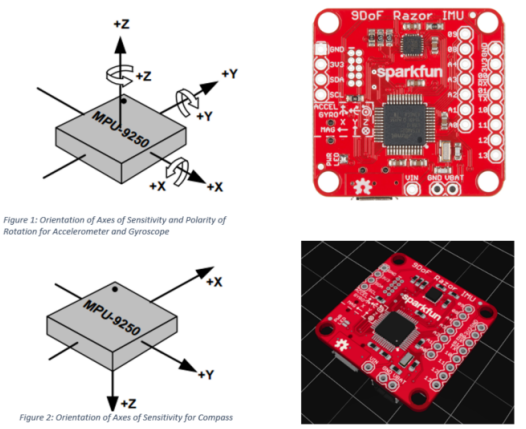


Fig. 23. IMU Axis

Software

Step1: Making the IMU to display all the data in the serial port

Please get the firmware and the config.h file from the link
https://learn.sparkfun.com/tutorials/9dof-razor-imu-m0-hookup-guide

Make sure to have Sparkfun boards and all the libraries mentioned in the above link installed in your Arduino, before continuing with next step. Please install Arduino IDE and open both the firmware and config.h file on the same Arduino window (in separate tabs).

Please open the config.h file and make the following changes You can find the following options below, Default Logging Parameters section.

If you enable the options true, the values of all of them will be printed in the serial port.

You can refer to the firmware program to know about the order in which the data will be sent to serial port.

Step 2: Node to make the serial port data get published in imu topic

```
#define ENABLE_TIME_LOG      true
#define ENABLE_CALCULATED_LOG true
#define ENABLE_ACCEL_LOG     true
#define ENABLE_GYRO_LOG      true
#define ENABLE_MAG_LOG       true
#define ENABLE_QUAT_LOG      false
#define ENABLE_EULER_LOG     false
#define ENABLE_HEADING_LOG   false
```

Fig. 24. IMU Parameters

Please make sure you have all the required tools mentioned in F1tenth website (mentioned under sensor configuration), before proceeding through the next steps.

Please clone the github repository mentioned in F1tenth website. The port name has to be changed in the file razor.yaml in the following link. razor_imu_9dof-indigo-devel/config Please get the imu node from the following link

https://github.com/RamrajSegur/Razor_Imu

Code explanation:

The code is written in python. A serial object is created to access the data in serial port and tried to publish in the imu topic by creating the publisher to the topic. Data that you can get from the IMU serial port using the firmware are

—Accelerometer readings along three axes (in g)

—Gyroscope readings with respect to three axes

—Magnetometer readings along three axes

—Quaternion information

—Pitch, Roll and Yaw

—Heading information

We have tried to neglect the effect of gravitational force on the accelerometer values using calibration.

You can find the sample code that uses the quaternions to take out the effect of gravity in the following link

http://www.varesano.net/blog/fabio/simple-gravity-compensation-9-dom-imus

Note: Only the quaternions are used to find the pitch and roll for hector SLAM application. They used the LIDAR scan matching to measure the yaw rate of the vehicle!

—Serial Interface - configuring sensor for proper active data (quaternion, linear acc, etc.) what data to turn on

—Node for serial to ROS - covariance calculation, and other calcs

4.2.2 *Lidar.* In our case, we are using Hokuyo Lidar UST-10LX, one of smallest and the lightest of its kind with a 130g weight. With a Mid-detection range 10m/20m and wide detection angle 270. Fast response in 25msec and high angular resolution 0.25. It is possible to detect size, position and

the moving direction of objects. LIDAR could provide Depth information to tell how far are objects and obstacles around us.

Fig. 25. Lidar

LIDAR start scanning from right to left, as shown in the figure below, as long as there are obstacles in the LIDAR scan right, we can know all measure points distance and angle value from LIDAR.

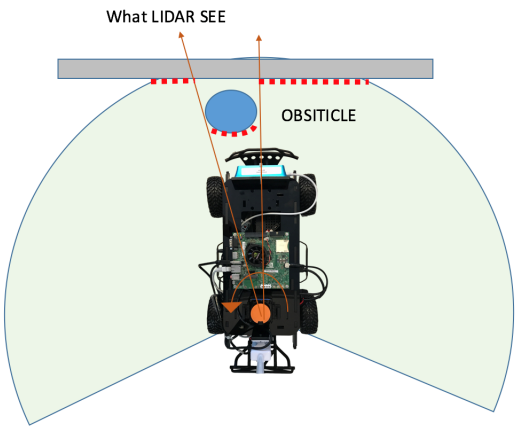


Fig. 26. Lidar Sensing

The fundamental is LIDAR is to send and receive light signal and record the traveled time. The calculation for distance is show as below:

Distance to obstacle = (speed of light * time traveled) / 2

LIDAR setup and configuration

Step 1: Hardware Connection

Setup hardware connection: As shown in the figure above. Hokuyo Lidar should be powered by the 16-20V Output Port of Energizer Battery Pack through the blue wire. The Ethernet port should be connected to the USB hub through USB 3.0 Data port.

Fig. 27. Lidar Sensing

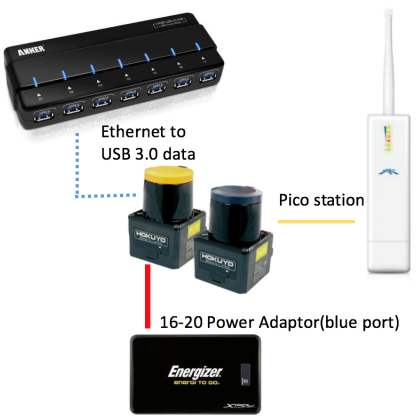


Fig. 28. Lidar - Hardware Connections

Step 2: Manually assign the correct IP Address to LIDAR

Lidar should be setup in windows machine with Benri data viewing tool and Ip changing tool. The results are shown as following:

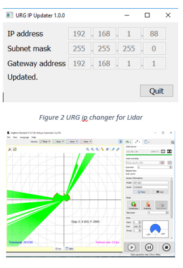


Fig. 29. Lidar - URG Node

Step 3: Setup Bridge Network and Read data through the Jetson

In order to connect to the LIDAR, we need setup a bridge network that allows the 2 Ethernet connections on the Jetson to coexist. Therefore, we need run below install script to install bridge-utils on the Jetson. $ sudo apt-get install bridge-utils Setting up the network bridge by running following commands in the terminal $ sudo brctl addbr br0 $ sudo brctl addif br0 eth0

$sudo brctl addif br0 eth1

$sudo ifconfig br0 192.168.1.2 netmask 255.255.255.0 up After this, we can check Ubiquiti network connection to see different devices connected on your network by running: $ ifconfig

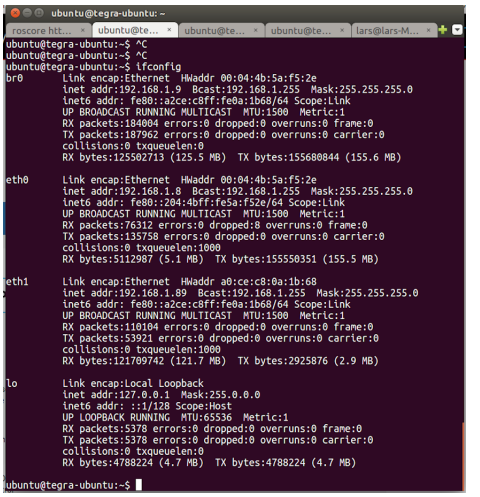


Fig. 30. Lidar - IP COnfigured

Once we see the there is a new connection called br0, and eth0 and eth1 connected to the Ip addresses of the Jetson and the LIDAR (result shown as above), we can install urg node package which is able to publish the Lidar scan message by running follow command:
$ sudo apt-get install ros-indigo-urg-node

Step 4: SSH into Jetson from your PC

SSH into Jetson:
$ ssh X Ubuntu@192.168.1.x (IP of Jetson, in our case, x = 8)
$ export ROS_HOSTNAME=192.168.1.X (IP of Jetson)
$ roscore
launch the node which is able to publish the Lidar scan using:
$ rosrun urg_node urg_node _ip_address: = 192.168.1.88
Finally, we could read streaming from scan topic by following script:
$ rostopic echo /scan
The result is shown as below:
We can also run RVIZ to visualize the scan result by following commands:
$ export ROS_MASTER_URI=http://192.168.1.X:11311/
$ export ROS_IP=192.168.1.Y (IP of your PC)
$ rosrun rviz rviz
Result is shown as below:

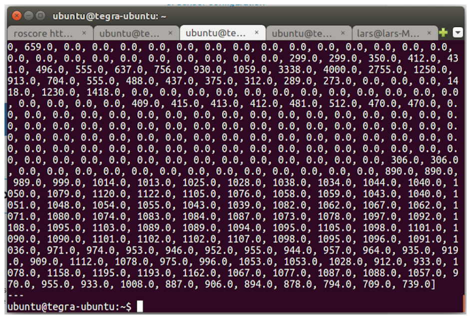


Fig. 31. Lidar - Scan Data

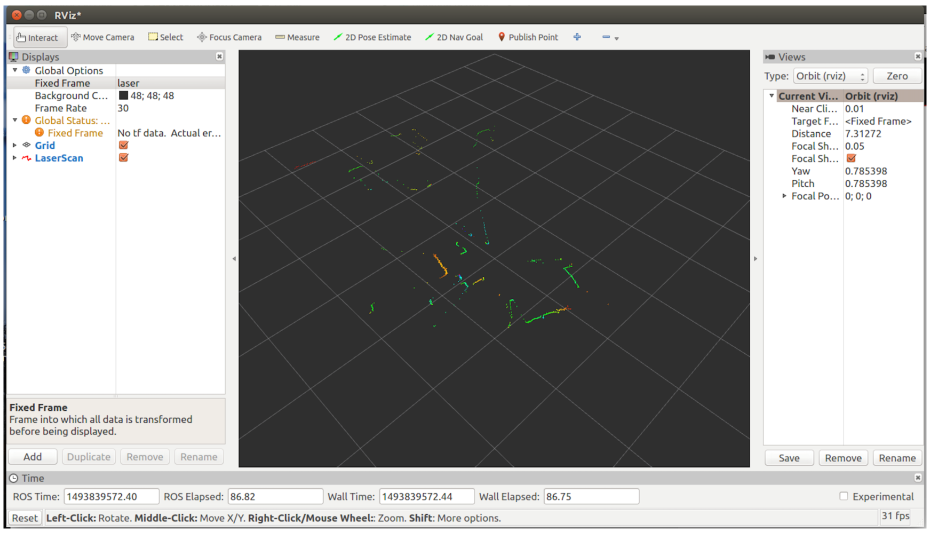


Fig. 32. Lidar - Scan Data on RViz

LIDAR ROS Message Structure

the message contains LIDAR data is: sensor_msgs/LaserScan.msg The detailed message structure and descriptions are shown below:

Message structure	Description
float32 angle_min	start angle of the scan [rad]
float32 angle_max	end angle of the scan [rad]
float32 angle_increment	angular distance between measurements [rad]
float32 time_increment	time between measurements [seconds]
float32 scan_time	time between scans [seconds]
float32 range_min	minimum range value [m]
float32 range_max	maximum range value [m]
float32[] ranges	range data [m] (Note: values < range_min or > range_max # should be discarded)
float32[] intensities	intensity data [device-specific units].

Fig. 33. Lidar - Scan Data on RViz

Matlab Implementation

Results from Matlab:

```
%% Subscribe LIDAR data
% subscribe '/scan' topic
laser_sub = rossubscriber('/scan','sensor_msgs/LaserScan');
% receive data
laser_msg = receive(laser_sub);
% identify data we want
laser_data = laser_msg.Ranges;
% for each step, get all scan points, if measure distance beyond max range,
% set it as range.max
    for i = 1:length(laser_data)
        if laser_data(i) == Inf

            mag = laser_msg.RangeMax;

        else

            mag = laser_data(i);

        end
    end
```

Fig. 34. Lidar - Matlab Implementation

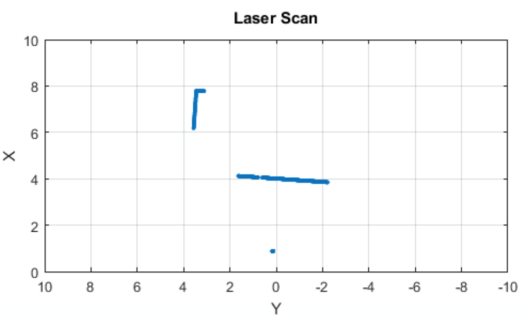


Fig. 35. Lidar - Matlab Results

4.2.3 *Structure Camera.* The vehicle that we built also had a structure sensor to generate a depth scan.

Structure sensor is a depth camera which is able to detect the distance to the certain points in the camera

For read data from structure sensor, we need install OpenNI2 package and node on Jetson TK1, so we can subscribe certain data.

Structure Sensor Interfacing

We can visualize the structure sensor message depth/image_raw through MATLAB

We need initiate node first by running: rosinit('http://10.42.0.225:11311')

Matlab Code:

4.2.4 *Actuator.* **Traxxas Car** - The actuators in this project, motor and servo, are equipped with the supplied Traxxas RC vehicle. This vehicle features an AC??? motor which applies torque to the rear wheels and propels the vehicle up to 35 mph in its stock configuration (before adding weight to the vehicle). The torque is transmitted to the rear wheels through an open differential allowing the vehicle to make sharp maneuvers with less resistance from the ground. Because the vehicle is meant to be operated via a handheld remote controller, the vehicle is already equipped with a motor controller. This motor controller functions like a servo; it sets the motor to a speed until a new speed is requested in the form of a pulse width modulation (PWM) signal. The other actuator is the steering system which

Fig. 36. Structure Sensor and sample scan

```
image_sub = rossubscriber('/camera/depth/image');

figure
while 1
    imgdata = receive(image_sub);
    img = readImage(imgdata);
    imshow(img)
end
```

Fig. 37. Structure Sensor - Matlab Implementation

uses a servo to manipulate the steering angle of the front wheels. This vehicle uses Ackerman steering to improve the handling and reduce resistance from the ground caused by steering.

Like the motor controller, the steering servo also sets the steering angle as requested by the incoming PWM signals.

4.2.5 *The Supervisory Controller.* **TK1 (the brains) supervisory controller**

Specs - The Nvidia Jetson TK1 embedded development board is the primary controller (the brains) of the vehicle. The role of the Jetson is to collect data from the Teensy, structure camera, IMU, and Lidar. The Jetson also functions as a server, using a wireless network through the PicoStation to communicate with the supporting host computer. During operation, the Jetson runs the all of the interfacing nodes; nodes which publish sensor data to a ROS topic, or nodes that send signals to actuators as per a ROS topic (rosserial, rostopic). The Jetson is operated remotely with the Secure Socket Shell (SSH) protocol from the host computer. The host computer handles the more computationally expensive processes such as Hector SLAM, and communicates through ROS.

Installation - The TK1 is preinstalled with Linux Ubuntu 14.04 for Tegra L4T out of the box. While the Jetson is suitable for a desktop application, it does not come with sufficient capabilities for many external peripherals and is also restricted in terms of configuration parameters. To address these limitations, the Grinch Kernel is installed (Jetson hacks). With the Grinch Kernel installed, the bare

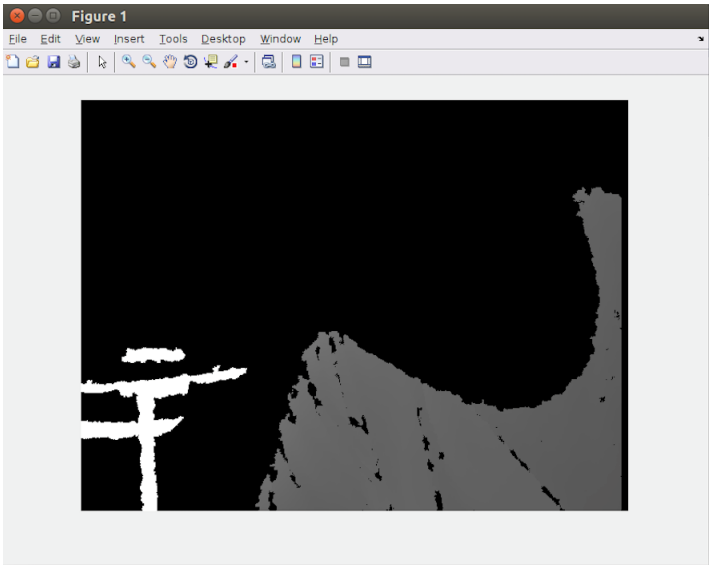


Fig. 38. Structure Sensor - Matlab Implementation

bones version of ROS indigo is also installed with several select packages such as rosserial (maybe only need rosserial-python), openni, etc.

All of the nodes that it runs ...

PID Control

sensor/actuator interfacing

Teensy the low level controller

The Teensy is responsible for relaying the ROS twist messages (velocity and steering commands) into PWM signals which is compatible with the factory installed controllers on the car. By using Teensyduino, and custom firmware can be loaded onto the Teensy through Arduino IDE. The ROS subscriber is created in the firmware and is always running on the Teensy after it has been flashed. The firmware operates in the subscriber callback, so that it only operates when it is getting new messages. Additionally, the firmware converts a twist message (linear and angular velocity) into a corresponding PWM signal. Due to variances between battery life, the lack of encoder feedback, and the rolling resistance, converting a velocity command into a corresponding PWM duty cycle is not easy and imposes a lot of error.

4.2.6 *Auxiliaries.* The Device Interfaces between the Jetson, a Laptop and the Lidar are facilitated with a Pico Station.

The device interface diagram is depicted below:

4.3 Control Algorithm

We have employed and modified the 'Pure Pursuit' Algorithm as the control Algorithm for this implementation.

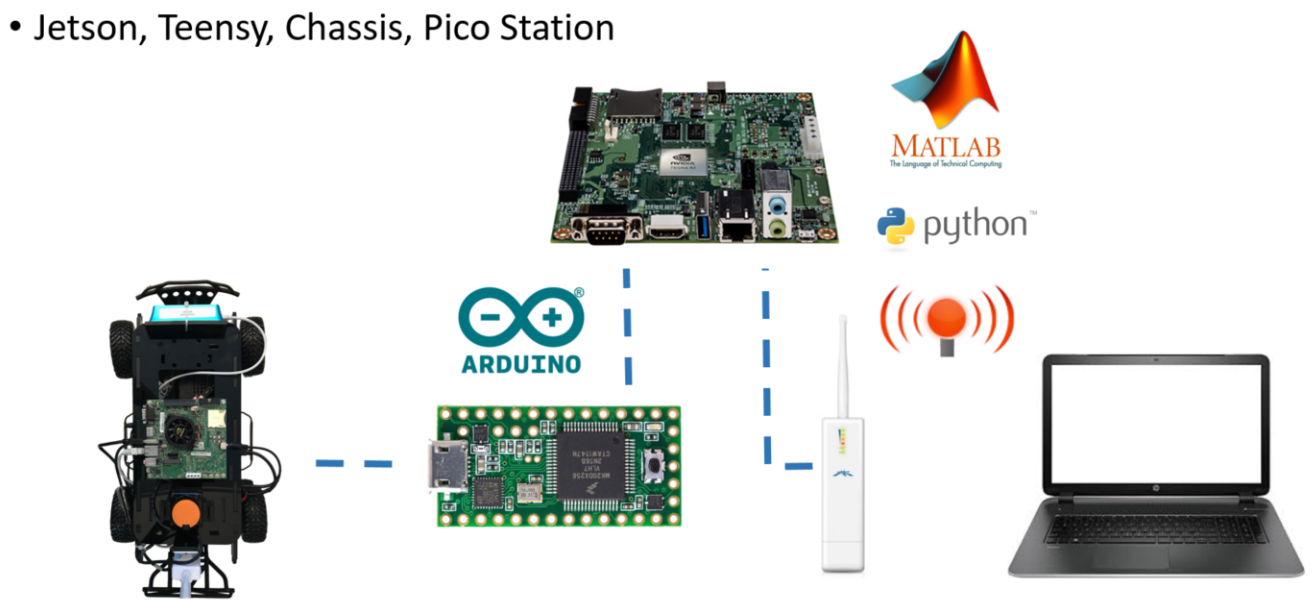


Fig. 39. Device Interfaces

4.3.1 *Motion Control.* Motion control is used to guide the vehicle along the track with a controlled trajectory and velocity. In terms of racing, motion control offers an interesting tradeoff of going fast at the risk of having an unstable system which crashes into the wall. For this implementation, a novel derivative of the pure pursuit family was. Unlike the Stanley method, or another PID application for lane keeping, pure pursuit does not directly set a desired state (position and angle) or require using an error function. Instead, pure pursuit works from the idea of determining a goal point in the vehicle frame, and then calculating the necessary steering angle and velocity inputs to reach this goal point. The idea of pure pursuit can be expressed by drawing an analogy to holding a carrot in front of a donkey, where carrot keeps evading the donkey down the hallway. Likewise, the F1tenth car chases after this goal point which is continuously evading the car down the track. This pure pursuit problem is broken down into three sections: calculating the goal point, calculating the required steering inputs to get to the goal, and calculating the appropriate velocity commands based on the goal point.

4.3.2 *Goal Point Determination.* This formulation of a goal point is slightly unconventional. Typically, the first step is to generate a path within an environment a priori. Then, the goal point is found by searching for the point on this path which is some look ahead distance from the rear axle of the vehicle. However, for a real-time control, another solution must be used.

In this implementation, the first step in pure pursuit control is to generate a goal point based on the information we have about the surroundings. In this case, the information of the surroundings is local information taken from the Hokuyo Lidar sensor. As previously discussed, the Lidar data is a series of depth measurements which are taken at a constant angle increment interval sweeping right to left. This data is transformed into the x-y plane with an origin at the center of the Hokuyo sensor which also happens to be nearly over the rear axle (the convenience of this will be illustrated later).

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = r_i \begin{bmatrix} \cos(\theta_{start} + i * \Delta\theta) \\ \sin(\theta_{start} + i * \Delta\theta) \end{bmatrix}$$

where, subscript, i, is the index of the sensed range; r is the corresponding sensed range; $_start$ is the initial angle; and is the angle increment of the Lidar sensor.

With the data in the x-y plane, one means of calculating the goal point is to take the centroid of all the data points. This is done by taking the average values.

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n} x_i$$
$$\bar{y} = \frac{1}{n}\sum_{i=0}^{n} y_i$$

where n is the number of ranges which are considered. The goal point is thus located at the centroid, ie. the goal point, (g_x, g_y), is equal to $(\bar{x}, \bar{y})$ relative to the Hokuyo sensor location which is above the rear axle.

4.3.3 *Steer Control.* With the goal point located relative to the rear axle, the steering control can be implemented. For simplicity, the Ackerman steered vehicle can be modelled as an equivalent bicycle. Essentially, this means that instead of considering the four wheels, the motion of the vehicle is represented by the motion of bicycle with a front wheel located at the center of the vehicle between the two front wheels, and a rear wheel located at the center of the vehicle between the two rear wheels. This reduces the complexity, and is equivalent from a geometric standpoint. The measured wheelbase of the vehicle for this implementation is about 34 cm.

Using the goal point, the look ahead distance (fig XX) is now calculated by taking the hypotenuse of the triangle which is formed by the goal point, and the vehicles heading and position.

$$l_d = \sqrt{g_x^2 + g_y^2}$$

Where l_d is the look ahead distance. This look ahead distance is then used to calculate the curvature of the desired trajectory (arc) from the rear axle to the goal point.

$$\frac{1}{R} = k = \frac{2\sin(\alpha)}{l_d} = \frac{2g_x}{l_g^2}$$

Where is the curvature of the arc, R is the radius of the arc, and is the angle between the vehicle heading and the line formed by the rear axle and the goal point. By incorporating the wheelbase of the vehicle, the desired steering angle is also calculated.

$$\phi = a\tan(kL)$$

Where ϕ is the required steering angle to follow the desired trajectory.

The steering angle of the vehicle is not precisely known. Therefore, instead of using this steering angle directly, a gain factor is introduced to tune the responsiveness of the steering. A steering bias is also introduced to tune the steering trim and compensate for misaligned wheels.

$$Twist.Angular.Z = k_{st}\phi + b_{st}$$

Where the steering command takes the form in an angular twist message, k_{st} is the steering gain, and b_{st} is the steering bias.

4.3.4 *Velocity Control.* While conventional pure pursuit determines a goal point as a function of the vehicle velocity, in this implementation the velocity is determined from the goal point.

$$Traditionally : l_d = f(v)$$
$$ThisImplementation : v = f(l_d)$$

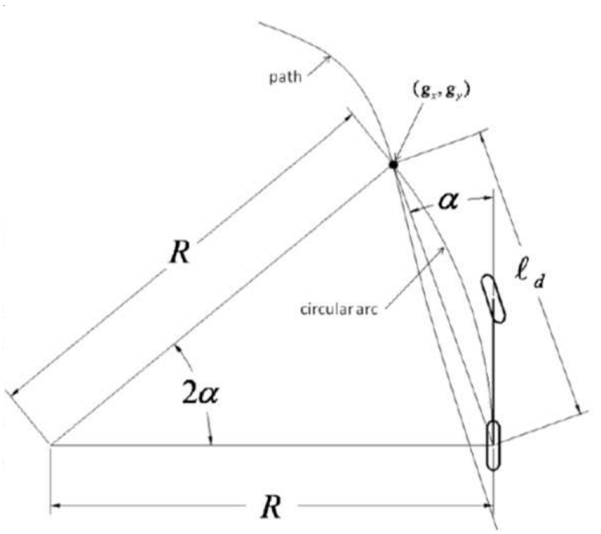


Fig. 40. Look Ahead Description

Not only does the goal point shift laterally as the vehicle navigates the track, but it also shifts longitudinally. In the sections of the track where there is a clear path far from the vehicle, the goal point is further away thus the look ahead distance is greater. Much like human drivers, when the vehicle can calculate a trajectory to a greater distance, the vehicle should go faster. Conversely, if the vehicle can only calculate the trajectory which is very close, the vehicle should slow down. Using this methodology, the velocity can be calibrated to a given look ahead distance linearly.

$$Twist.Linear.X = k_{vel} l_d + b_{vel}$$

Where the velocity command takes the form of a linear twist message, k_{vel} is a velocity gain which increases sensitivity of the look ahead distance, and b_{vel} is another tuning parameter introduced to shift the velocity performance to have a nominally faster or slower speed. Additionally, if the velocity ever exceeds some maximum velocity, the velocity is saturated at this maximum velocity, vel_{max}. One of the limitations of this control regime is that the look ahead distance is not necessarily an optimal indicator of how fast the vehicle can go. For example, in straight narrow passages, the centroid of the scan data is closer to the vehicle than in the case of straight wide passages. This means the vehicle will reduce its velocity in narrow passages. This inherent behavior of the control algorithm is useful for navigating narrow passages. However, in a racing environment, the observed behavior is over-cautious.

To make the car go faster, while maintaining the functionality to reduce speeds in slow passages, a normal weighting method was implemented. This idea of weighting the Lidar scan is analogous to a driver focusing on different parts of the path. In this implementation, it is advantageous to have the car look down the path. For example, when the look ahead distance of the vehicle is reduced because of a narrow passage, the vehicle should weight the importance of what lies ahead more than what is to the sides, when considering the vehicle speed. To weight the importance of the intended path, a normal curve is used to weight the important part of the scan data.

First the important part of the track must be decided. This is determined by the previous means and predicting the direction of the goal point, . Next, the Lidar data which is sent in that direction must be weighted; the importance of the Lidar data at is the Lidar data which receives the greatest weight.

The corresponding index, can be thought of as the mean of a normal distribution. Next the window of weighting should be considered. How narrow or wide the window of importance is (how far it deviates from the maximum importance), can be represented by the variance of the normal distribution. This variance is a constant tuning parameter which can be used to weight a more narrow or wide range of data toward the goal point. Finally, the weighted data can be calculated as follows

$$\vec{d_w} = \vec{d_{uw}} N(\alpha_{goal}(\vec{d_{uw}}), \sigma)$$

Where $\vec{d_w}$ is the distance vector after being weighted, $\vec{d_{uw}}$ is the distance vector before being weighted, α_{goal} $\vec{d_{uw}}$ is angle toward the goal point as defined previously, σ is the variance of the normal distribution, and is the normal distribution notation indicating a normal distribution with a mean of $(\alpha_{goal}\vec{d_{uw}},)$ and variance of σ . Equivalently, the element-wise equation is shown below.

$$d_{wi} = \frac{d_{uwi}}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(\alpha_i - \alpha_{goal})^2}{2\sigma}\right)$$

Where the subscript, i, indicates the ith index. Figure 41 illustrates the effect that the weighting function has on an approaching corner to the left where the angle sweeps from right to left as discussed in the Lidar section.

After weighting the scan data, the centroid is calculated again with the weighted depths. The new centroid will be more representative of the trajectory; it will influence the vehicle to go almost as fast in a narrow passage as it does in a wide passage. Additionally, the car is quicker exiting the corners because it is better at recognizing a long straight away and thus accelerates much sooner. An example of the effect that weighting has on the laser range data in the scenario of an approaching left corner is shown in Figure 42

The new look ahead distance is a better metric for gauging the appropriate speed which the vehicle should travel. The weighted centroid is used to control the vehicle speed while the unweight centroid is used to calculate the desired steering inputs of the vehicle. However, applying the filtered centroid to control the steering of the vehicle could potentially be advantageous in obstacle avoidance as well as taking a more aggressive racing line (getting closer to the apex of the turn).

4.3.5 *Human Driver Model.* This control scheme is novel from a controls perspective, however, it also shows potential in the context of modelling the human driver. This controller is also a relatively simple implementation of a human driver model. For example, a human driver will go slower through a narrow passage just as the controller demonstrated. In a wide-open area, the controller is comfortable with higher speeds. By implementing weighting functionality, the controller focus on the trajectory of which it is going (much like a human driver would). Much like the human driver, when the controller cannot detect the environment very far in front of it, it is less comfortable at speed and thus decreases its speed.

By varying the constants of the controller, different driving personalities can be modelled. For example, by reducing the k_vel gain, introducing a moderately low minimum speed, and increasing the parameter, the personality of the controller replicates that of a cautious driver. With these settings, the vehicle does not accelerate excessively on the straight sections of the track. Additionally, the vehicle will slow down for narrow passages. Contrarily, these parameters can be tuned to model an aggressive driver - a racecar driver

4.3.6 *Gains and Tuning Parameters.* The tuning of this controller is much easier with the real-time tuning GUI discussed previously. The python script, which is running real time on the TK1, subscribes to not only the laser scan data, but also the gains which are being published by the GUI. This makes

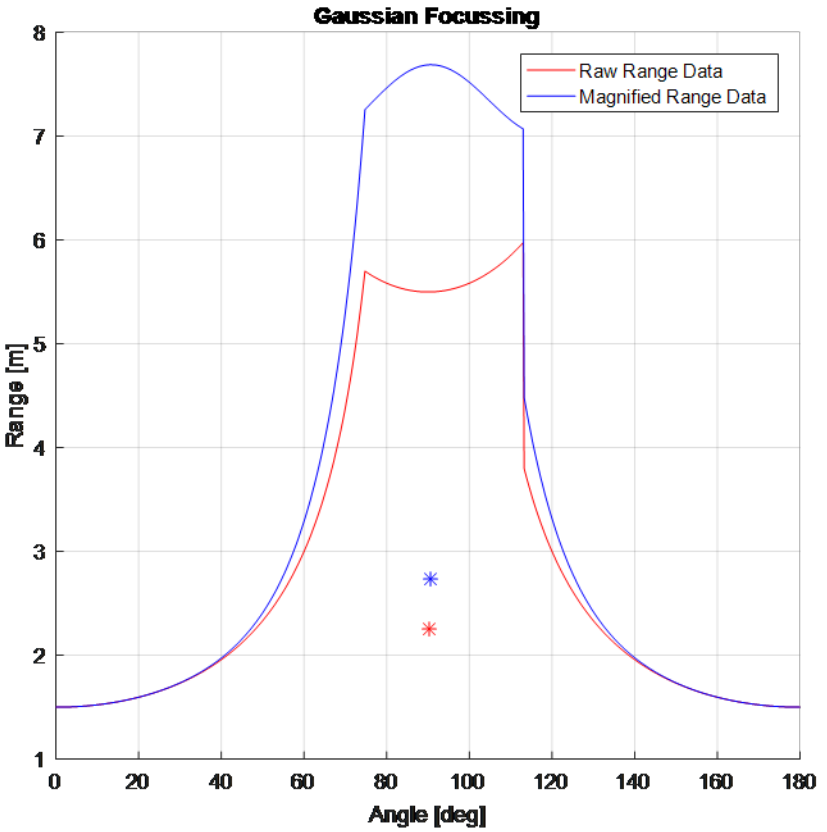


Fig. 41. Gaussian Focusing

the tuning process much less tedious as the code does not need to be shut down for any reason, and quick subtle adjustments can be sent on the fly. Furthermore, with the tuning of the gains, the vehicle speed can be shut off from the client computer for any reason.

Throughout the tuning process, there were several trends that were observed and used to tune with. To make sure that the vehicle does not damage itself, the first parameter that was set in the tuning process was the vel_{max}. The manipulation of this tuning parameter was completely based on whether a higher speed was desired throughout the section of the track which the velocity was saturated.

The tuning of the k_{vel}, b_{vel}, were done simultaneously, while vel_{max} was tuned separately. vel_{max} simply limits the maximum speed, so if the vehicle ever is reaching to high of a top speed this value was reduced. If we ever wanted to achieve a higher top speed this value was increased. This parameter was only relevant to the long straight sections of the track where vehicle can saturate at the limited speed. Tuning k_{vel} and b_{vel} was more complicated because they are coupled together, meaning their values must be considered simultaneously during optimization. Increasing b_{vel} essentially shifts the speed which the vehicle attempts to travel throughout the entire track; it will go faster on corners and straightaways (assuming the speed is not saturated at the maximum value). However, throughout the optimization process, it was rarely the case where the vehicle speed could be uniformly increased

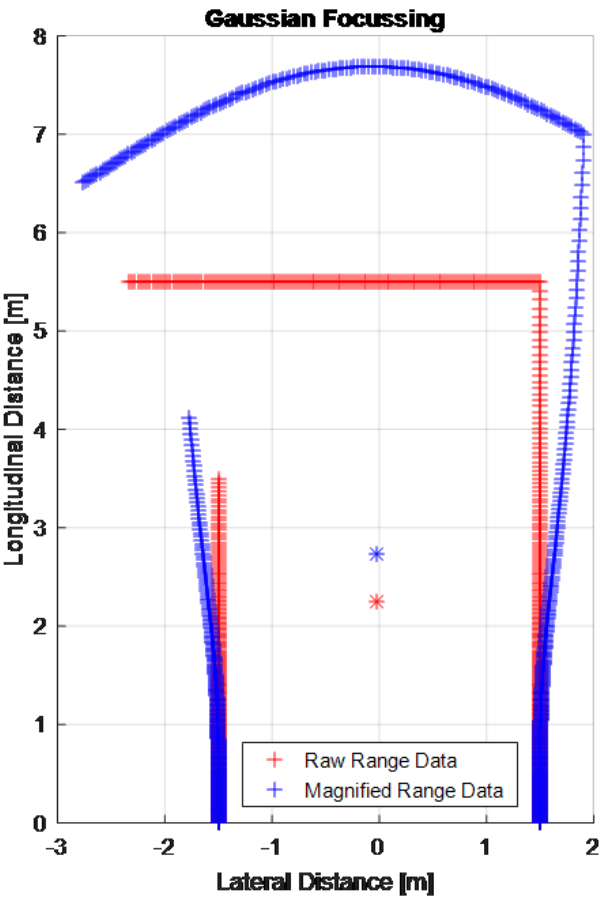


Fig. 42. Gaussian Focusing Data Points

throughout the entire track. What was more common was a scenario where a faster straight away speed was desired. For this, the velocity gain, k_{vel}, must be used. Like the velocity shift parameter, b_{vel}, the gain also increases the speed throughout the entire track where the increase is more pronounced at the straight sections. To manage this, increasing the overall lap time (making the driving more aggressive) was accomplished by increasing gain and simultaneously decreasing the shift.

Having a negative b_{vel} and a high k_{vel}, allowed for the vehicle to brake leading up to corners and drift around the corners. As discussed previously, cautious behavior can be achieved with the opposite settings: having a positive b_{vel} and a low k_{vel}. The variance of the weighting effect was set to some reasonable value initially and kept the same after words. This is because changing the variance considerably, would drastically effect the b_{vel} and k_{vel} parameters. Instead of managing three interdependent tuning parameters, the variance was set and held constant.

The steering tuning was done mostly separate from the velocity tuning, however, it was found that it could be advantageous to make the steering parameters be dependent on either the lookahead distance or the vehicle speed. Regardless, for high speed and low speed, the tuning process was essentially the

same. The b_{st} parameter can be tuned statically before even running the vehicle. To account for any steering bias which is existent on the vehicle the steering shift factor can correct this. Before running the vehicle, a neutral steering command is sent to the vehicle which tells the vehicle to turn the wheels straight ahead. This steering command was varied until the vehicle physically tracked the straight position. The steering command which physically sets the wheels straight on the vehicle was set as the steering bias parameter. The steering gain must be tuned while the vehicle is moving as it is a dynamic parameter. Whenever the vehicle takes turns to widely, the steering gain, k_{st}, is increased. However, if the vehicle starts to oscillate during the straight section or if it seems generally too sensitive in terms of steering, then the steering gain is decreased.

Parameter	Value (cautious)	Value (aggressive)
k_{vel}	7935	9728
b_{vel}	0	-17.9
vel_{max}	22.5	29.2
k_{st}	214	424.6
b_{st}	5	5
σ	100	100

The above values might vary depending on the battery charge.

5. RESULTS

Here articulated are the results of the objectives of the course. The objectives are divided into three broad classes, a mapping objective, a racing objective and an additional obstacle avoidance objective.

5.1 LIDAR-based PID

The vehicle was expected to have:

—Fully Automatically Control

—Prevent hitting boundaries

—Smooth Ride quality

The tasks for the this objectives were:

—Low Speed LIDAR

—Low Speed Obstacle Avoidance

—High Speed LIDAR based PID

—High Speed Obstacle Avoidance

The vehicle was able to achieve the above tasks, with a combination of different values of gains.

5.2 Mapping Objective - LIDAR based SLAM

Real-Time Hector mapping of the track was an objective.

To read the data from the TK1, a wireless connection is established with the ubiquity station and by exporting the ROS_MASTER variable we can read the scan values at real time. These scan values are used by the Hector-Slam node to create required transfer functions and subsequently map the surroundings.

The map obtained by running Hector-Slam on real time is displayed below

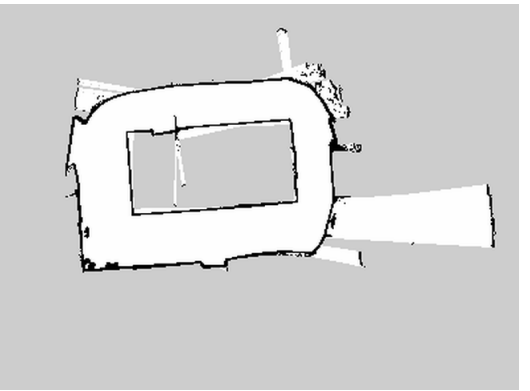


Fig. 43. Hector Slam map

5.3 Speed Objective

The gain values used in this task were equal to the values described as aggressive in the fore-mentioned table.

Also, the gain values might vary based on the level of charge in the battery.

6. CONCLUSION AND FUTURE WORK

The project has provided a great opportunity to learn and implement the ROS tools. It also enabled to explore the sensors and Jetson TK1. The device interface techniques provided a brief knowledge about the benefits of ROS framework and its ease of operation in clustered sensor environments. It also enabled us to gain knowledge about the reliability of sensors and different algorithms.

For further studies, we like to enable the visual odometry using the stereo camera/ structure camera to drive through the long hallway which is a real challenge for LIDAR based odometry.

REFERENCES

[1] uber picking up passengers https://www.theverge.com/2017/2/21/14687346/uber-self-driving-car-arizona-pilot-ducey-california
[2] google waymo since 2009 https://waymo.com/journey/
[3] apple gets permit to test self-driving cars in california https://www.nytimes.com/2017/04/14/technology/apple-self-driving-car-permit.html?_r=0
[4] samsung partinering with hyundai to develop self-driving cars outfitted with sensors and machine-learning systems from samsung https://www.theguardian.com/technology/2017/may/02/samsung-self-driving-car-challenge-google-waymo-apple-uber
[5] Lecture Notes - AuE 8930, Dr Jia, Spring '17 - Look ahead distance determination