Autonomy Science and Systems

Final Project Report

5th May 2017

Team:

Longxiang Guo

Dylan Gelven

Vishwas Powar

Adhiti Raman

Abstract

The interest and need for autonomy in vehicles in growing. A larger number of automobile manufacturers are looking for people with a different set of skills than was traditionally expected from an Automotive engineer. Through the course of this project, we gained a basic introduction into what it takes to make an autonomous car and successfully built an autonomous mobile robot of our own.

Acknowledgments

This project could not have been possible without the assistance and support of a number of people.

Foremost, we would like to thank Professor Krovi for giving us this opportunity to experiment with new tech and learn a new skillset.

We would also like to thank our classmates for all their help and knowledge, whose support and contributions made this a success.

Table of Contents

Introduction

Advanced robotics is beginning to have a major impact on the society we live in, not only in manufacturing but also in all aspects of our lives. Robots are used in many industries today, like military, space, marine, farming, mining, medical, and of course the automotive industry.

Manufacturing robots date back to the 1950s when George Devol invented a robot named, "Unimate" - which was sold to General Motors to lift hot pieces of metal from die casting machines and stack them. Robots in today's manufacturing industry save companies' time and money by completing jobs in a fraction of the time it would take human employees. These manufacturing robots are extremely precise and have repeatability and productivity far above the capacity of any human. Manufacturing robots allow companies to produce parts around the clock with extremely high reliability.

A more recent development in the field of robotics is the research and development of autonomous vehicles. Early self-driving vehicles date back to the 1980s, but it was in the early 2000's with the DARPA challenge, which made the most progress with autonomous vehicles. The DARPA challenge - which began in 2004 with the Defense Advanced Research Project Administration challenging various teams working on autonomous vehicles to compete for a 1$ million-dollar prize - provided a tremendous amount of progress in the field and helped produce some of the hardware and software used in today's R&D of commercial autonomous vehicles.

SAE level	Name	Narrative Definition	Execution of Steering and Acceleration/ Deceleration	*Monitoring* of Driving Environment	Fallback Performance of *Dynamic Driving Task*	System Capability *(Driving Modes)*
***Human driver* monitors the driving environment**						
0	**No Automation**	the full-time performance by the *human driver* of all aspects of the *dynamic driving task*, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a
1	**Driver Assistance**	the *driving mode*-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	Human driver and system	Human driver	Human driver	Some driving modes
2	**Partial Automation**	the *driving mode*-specific execution by one or more driver assistance systems of both steering and acceleration/ deceleration using information about the driving environment and with the expectation that the *human driver* perform all remaining aspects of the *dynamic driving task*	**System**	Human driver	Human driver	Some driving modes
***Automated driving system* ("system") monitors the driving environment**						
3	**Conditional Automation**	the *driving mode*-specific performance by an *automated driving system* of all aspects of the dynamic driving task with the expectation that the *human driver* will respond appropriately to a *request to intervene*	System	**System**	Human driver	Some driving modes
4	**High Automation**	the *driving mode*-specific performance by an automated driving system of all aspects of the *dynamic driving task*, even if a *human driver* does not respond appropriately to a *request to intervene*	System	System	**System**	Some driving modes
5	**Full Automation**	the full-time performance by an *automated driving system* of all aspects of the *dynamic driving task* under all roadway and environmental conditions that can be managed by a *human driver*	System	System	System	**All driving modes**

Table 1: SAE classification of autonomous vehicles

The commercial autonomous vehicle is becoming such a popular idea that there are large-scale efforts being made to standardize and classify them. For instance, the Society of Automotive Engineers released a scale on the level of driving automation a commercial vehicle has. The levels are listed in the *Table 1*.

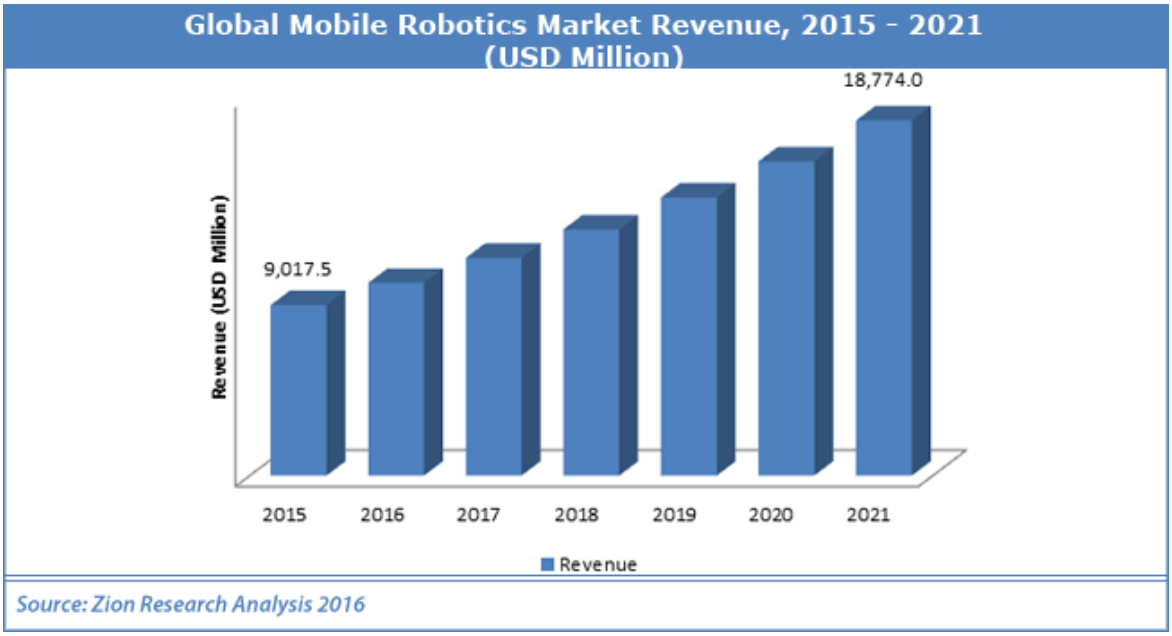
As of early 2017, every major automotive manufacturer is in a very competitive race to be the first to mass-produce a fully autonomous car. Most new cars produced have some form of automation embedded in their systems, from adaptive cruise control, side collision warning, automatic emergency braking, front collision warning, to autonomous lane keeping/departure systems, which are classified as SAE level 2 automation. The goal of the automotive OEMs is to offer a level five, fully autonomous vehicle on every roadway and environmental condition, to the consumers. In can be argued, that Tesla is the front runner in the competition to bring a fully autonomous vehicle to market. Tesla offers an autopilot system in all of its vehicles and provides software updates as validation and testing on its newer versions of autonomous software is complete. However, this autopilot can only be used under certain operating conditions, such as on a divided highway or on local roads under the speed of 35 MPH. The autopilot also still requires the driver to be attentive at the wheel and even touch the wheel every so often to prove the driver is attentive and able to take over if need be.

Many other OEMs are pouring money into the autonomous driving industry to keep up with the competition. This is evident in Ford Motor Company's recent 1$ billion investment in Argo AI, an artificial intelligence company based in Pittsburgh, PA. This partnership will combine Ford's autonomous vehicle development with Argo AI's robotics expertise to further the advancement of their autonomous vehicles. This all comes as Ford tries to stay on their timeline of promising to deliver high volume level four autonomous vehicles by 2021.

The automotive industry plays a big role in the advancement of mobile robotics, but other industries are also ramping up their investment. Zion Market Research, has estimated by 2021 the mobile robotics market revenue will reach over 18$ billion.

This market provides great opportunity for employment in an industry with a great future and job outlook. The industry needs recent college graduates with experience in mobile robotics, which can help further develop the advancement of automation and autonomous vehicles.

Here at Clemson University's International Center for Automotive Research (CU-ICAR), the United States only graduate automotive engineering program, the goal of the graduate program is to create engineers who will seamlessly transition from the academic world into the industry. In an automotive industry which is going through a transition phase with the introduction and implementation of autonomous vehicles and continued automation in manufacturing, the industry needs new college graduates with experience with autonomous systems. CU-ICAR provides the perfect platform to provide graduate students with the knowledge required to be able to transition into the industry and start making an impact in the field of autonomous systems. In the course, Autonomy: Science and Systems, a seminar like course, where students are introduced to various mobile robotics systems and their various sensors, the knowledge learned is a perfect foundation and introduction into the world of autonomous systems. Students are exposed to a variety of wheeled platforms from differential drive to four-wheel car-like systems. Students get hands on experience with the hardware and implementation of the sensors used in these wheeled platforms. Two mobile systems students used this semester were the Turtlebot personal robot and the F1tenth autonomous racing RC car.



The Turtlebot 2 personal robot, a differential drive robot, was used as an introduction to mobile robots for the course. The Turtlebot is controlled by an onboard laptop, which can be accessed remotely by other laptops via SSHing for wireless control. The Turtlebot 2 pictured below includes the following sensors:

Figure 1 Turtlebot 2

SENSORS		
3D VISION SENSOR (ASUS Xtion PRO LIVE)	Color Camera: 640px x 480px, 30 fps.	Depth Camera: 640px 480px, 30 fps
ENCORDERS	25700 cps	11.5 ticks/mm
RATE GYRO	110 deg/s Factory Calibrated	
AUXILIARY SENSORS	3x forward bump, 3x cliff, 2x wheel drop	

These sensors are interfaced with the onboard computer via the Robotics Operating System (ROS). Students learned to communicate between sensors and the controlling laptop by publishing and subscribing to various topics through ROS. These topics have specific functions through the various sensors and allow for accurate data measurement and control. Learning ROS on the Turtlebot provides a great introduction to ROS and mobile path planning and control. The knowledge learned from the Turtlebot transferred over to the next project in the course, F1tenth race car, which also uses ROS.

The F1tenth car was the second mobile platform introduced in the Autonomy course. The F1tenth car is part of an autonomous racing competition designed by students at the University of Pennsylvania. The goal of the F1tenth competition was to provide students around the world with a introductory course in autonomous vehicles, while providing important hands on and interfacing experience with the hardware. In the figure below you can see an image of the F1tenth car and the list of sensors provide:

Sensors
SparkFun 9DoF Razor IMU M0
LIDAR
Structure Sensor (Optional)
ZED Camera (Optional)

One of the many challenges faced in any project, let alone an autonomous mobile robot project, is the challenge of system integration. Any autonomous vehicle will require a number of various different sensors, and these sensors need to communicate efficiently to accurately drive the vehicle. Throughout this course students were faced with integrating the various sensors on both the Turtlebot and F1tenth car. This experience with sensor and vehicle integration is valuable knowledge, which can be applied in the industry as the transition to more driver assistance systems and autonomous control continues.

Upcoming in the rest of this report, we will discuss the background and literature reviewed during this course to help gain the knowledge of mobile robotics, the methods used to control various mobile robotics systems, the implementations of those methods on the final F1tenth group project, and the results achieved by the final F1tenth car.

Literature Review

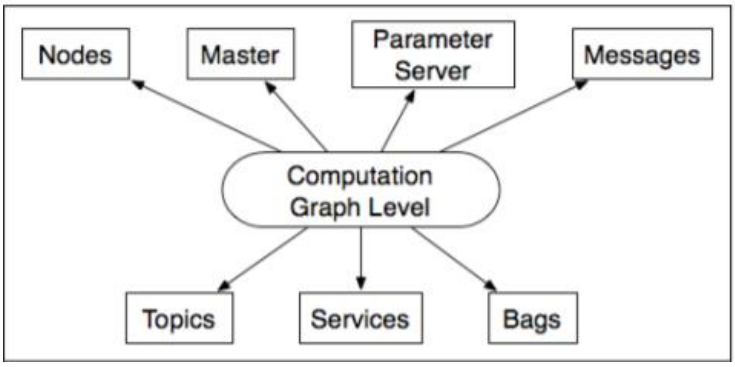
The section covers other projects that set the basis and the inspiration for the autonomous car described in this report. The biggest contribution and help was the F1tenth competition held by CMU last year. However, they were not the first. Here, we discuss the alternatives to F1Tenth. We also highlight other popular ROS-based robots that we used in class and that are commonly used in research. We begin with an overview of ROS - and why this novel system architecture is the next gen tool for implementing autonomy in cars.

ROBOTICS OPERATING SYSTEM (ROS)

Robot Operating System commonly known as ROS is a flexible framework for writing robot software that can be used for hardware abstraction, low-level device control, implementation of commonly-used functionalities, message-passing between processes and package management. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. It is distributed under the term of the BSD license, and is also open source software which is free for development of both non-commercial and commercial projects. ROS was developed in 2007 by Willow Garage which is now known as the **O**pen **S**ource **R**obotics **F**oundation (OSRF). Thereby making ROS a newer growing platform which is considered a global standard for robotic applications. There is already an extensive list of robots that support ROS and the list is still growing. One such robot well-known for educational and research purposes is the turtlebot. The turtlebot is commonly used to solve path planning problems, obstacle avoidance and navigation.

A ROS system is comprised of several independent nodes, each of which communicates with the other nodes using a publish/subscribe messaging model. This is a big advantage of using ROS because the nodes don't have to run on the same system (i.e. they can be distributed in a network) and are very flexible. ROS provides platform-independency by building and executing codes blocks in different languages like C++ and python. ROS is a complete modular platform that is built out of several packages. Packages are the primary unit for managing software in ROS and can contain libraries, tools, executables, configure files, etc. Nodes are the processes which are combined into a graph and communicate with other processes using steaming topics, RPC services, and the Parameter Servers. Every node is declared to the master. The master (also known as roscore) is a kind of a server where nodes are declared and registered. It allows nodes to find each other and exchange data. The roscore starts the rosout topic. It is the console log reporting mechanism used in ROS. The main idea is that the rosout node subscribes to /rosout, logs the messages to file and re-broadcasts the messages to /rosout_agg. Topics are the connections between nodes. They are named buses over which nodes exchange messages with a publish/subscribe paradigm. A node sends out a message by publishing it to a given topic. A node which is interested on a certain kind of data will subscribe to the suitable topic. There are a lot

of different messages to send on a topic. The use of messages depends on the application to be developed. Messages are data structures made of a combination of primitive types like Boolean, integers, floating point. This is a compact and fast way to send data to different systems. A Parameter Server is a shared, multivariable dictionary that is accessible via a network. Nodes use this server to store and retrieve parameters at runtime.



TURTLEBOT

The Turtlebot is an out-of-box mobile robot platform that was an excellent beginner's tool to learn ROS. It's open source code structure meant that there are a lot of pre-existing programs and toolbox readily available to make this a fast and easy process.

The robot was developed by Willow garage by Melonee Wise and Tully Foote in November 2010. Turtlebot 2, which was used in our class, ships with a Kobuki robot as a base, an RGBD camera (which in our case was the Astra Pro), and a laptop with ROS. An excellent introduction to Turtlebot and ROS can be found on the ROS wiki page (http://wiki.ros.org/Robots/TurtleBot).

The main features of the Turtlebot are described below:

The Kobuki base

The iClebo Kobuki base is not a standalone robot base. It main strength lies in its flexibility to be operated by any external computation device. The base consists of a cliff detector, gyroscope, bumper sensor, a two-wheel drive, with separate encoders. The base is capable for maximum payload of 5 kg and can operate at a maximum speed of 0.65 m/s with a rotational speed of 180 °/ S. It carefully observers an obstacle clearance of 15 mm (0.6 in).

Orrbec Astra Pro RGBD Camera

The Orrbec camera shipped with the robot is essentially meant to be the eyes of the bot. Capable of viewing in color as well depth, it supposed to be an excellent tool for map building and autonomous navigation. However, the implementation is a new one, and the Orrbec does

not ship with a driver customized for ROS. The *openni2* driver which works on most cameras does not work very well with the Astra Pro, as it not (yet) designed to stream HD images.

ROS implementation on Turtlebot

The Turtlebot ships with it own SDK pre-loaded, but this easy to install on any Ubuntu computer. The most imporatant and useful feature of the software package is it's Gazebo environment. Almost every program that run on the Gazebo environment easily translates to the actual Turtlebot.

It is best to learn to operate the robot locally – by setting up the Turtlebot as a ROS Master and communicating with it from your own PC over WiFi. This is something we will continously encounter with all robot we use. The importance of being able to remotly operate and modify code and commands cannot be overstated.

Turtlebot set as the ROS MASTER

ROS_MASTER_URI=http://localhost:11311

ROS_HOSTNAME=<Turtlebot_IP>

Remote PC

ROS_MASTER_URI=http://localhost:11311

ROS_HOSTNAME=<Turtlebot_IP>

With most applications and their nodes neatly organized into launch files, the Turtlebot requires very little pre-requisites in ROS to implement any commands.

F1TENTH

The F1Tenth platform is a global platform, where students compete twice a year, with fully autonomous 1/10 scale race cars capable of achieving speeds of up to 40mph. With uniform hardware amongst competitors, successful teams demonstrate novel approaches for trajectory

planning, race strategy, and dynamic obstacle avoidance. Solutions to these problems are critical for the deployment of self-driving cars and advancement of mobile robotics. One of the primary reasons for selecting a F1Tenth car for this project is its high-speed capability and on-board sensor technology, that is analogous to many autonomous real sized cars. Its compatibility with ROS (Robot Operating System), high speed processing of sensor readings, and lateral and longitudinal dynamic control parameters, makes it the ideal choice of scalable model to study autonomous vehicle technology.

The **mLab** (Real-Time & Embedded Systems Lab) promotes competition with the autonomous 1/10th scale F1 race cars through the website: http://f1tenth.org. The F1 tenth Race car was developed as a part of intergrated course on Robotics at the University of Pennsylvania which later was develped into a racing competiton between the different teams. The intellectual merit of the F1/10 encompasses the development of computationally efficient and effective, planning and perception algorithms which enable safe autonomy. As its name suggests, it is derived from the Formula One (F1) race and tries to effciently develop algorithms for a similar racing environment with the only exception that the cars are now operated fully autonomously. The computation system consists of a Nvidia Jetson TK1 which runs **R**obot **O**perating **S**ystem (**ROS**). The vehicle chassis is a Traxxas 74076 Rally R/C car (in our case we used the alternate chasis providd on the website). It uses sensors like Hokuyo UST-10LX laser, inertial measurement unit (Sparkfun Razor 9- DOF IMU) and Structure depth camera along-with Teensy Microcontroller to generate PWM to control the steering servos and ESC to control the throttle speed.

The broader impacts of the F1/10 competition include hands on learning and development of practical tools for the analysis of cyberphysical-systems.

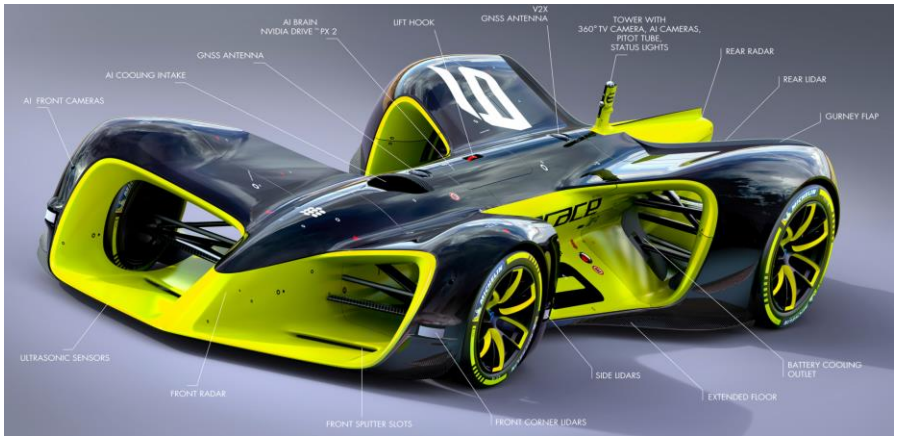
ROBORACE

Similar to the F1tenth car being built in this course, there also exist other autonomous vehicle development programs geared toward autonomous racing. One such organization that is promoting autonomous vehicles is Roborace, who plans to conduct an all autonomous race is

correspondence with formula E for the 2016-2017 racing season. This is extremely fascinating because it is taking the principles used in the development of F1tenth and using them in a full scale and very high speed environment.

Racing events such as NASCAR and Formula 1 are very popular sports with an extensive and established fan base. Racing, in the United States at least, is losing its appeal and its fan base. NASCAR fans are normally part of the older generations. Millennials and the upcoming generations are more focused on environment and the development of technology. This can explain why NASCAR is losing its appeal to the younger generations. It has not evolved at all with technology in recent decades. Roborace is planning to change that.

RoboRace is playing a ten-team race, involving a total of 20 cars, to race during the upcoming Formula E season. Each team will have equally powerful cars; however, each team must design and implement their own control algorithms and deep learning technologies. Roborace has gained a significant amount of attention and popularity, especially within the automotive and tech industries. This is evident through its partnerships with major tire manufacturer, Michelin, and tech giant, NVidia.



The Roborace car incorporates almost every piece of tech that is available on the planet. The robocar is covered with a variety of antennas, radars, lidars, IMUs, and various other sensors to control the vehicle autonomously, and to have the ability to be monitored from a distance. The car also has two navigation systems mounted on top of the car, which help accurately estimate the position of the vehicle on the track. In addition to front facing cameras, the car also has a tower mounted to the top of the vehicle that houses a 360-degreee camera, an AI camera, and status lights. As to be expected, this robocar requires extremely powerful processors to power all the various sensors. These processors need to be cooled, so the car has been design in such a way that allows the processors to be cooled solely from air flow.

This competition is an extremely exciting competition, not only from an entertainment point of view, but also from a technological stand point. It has the ability to drive competition to improve existing technologies and provide better algorithms that could be used on commercial autonomous vehicles. This race could have the effect on the autonomous driving community that the DARPA challenge had in the early 2000s.

AUTORALLY

Another alternative to the F1tenth project created by the University of Pennsylvania students, is a project from Georgia Tech called AutoRally. Students at Georgia Tech have designed an autonomous vehicle one-fifth the scale of a rally cross truck, which can drift and jump just like its full-scale counterpart. The goal and scope of the project is very similar to that of the F1tenth project. It is designed as a learning opportunity to design and implement the algorithms required to make quick and complicated decisions that would enhance the development of autonomous cars for full scale, real world applications. Also, similar to the F1tenth project, Autorally provides a relatively cheap and safe platform for students and researchers to test and implement new algorithms. These vehicles can be used as a proving ground for such algorithms, which could then be transferred to commercial autonomous vehicles for testing. Not only is it a great learning experience and testing platform, but Autorally is very exciting and brings a fun factor to the side of autonomous vehicles.

Some of the hardware used on the Autorally truck are very similar to the hardware used on the F1tenth vehicle. The RC car is one-fifth the size of an actual rally truck and has custom 3D printed mounts and enclosures. The sensors are also very similar, which include a high precision IMU, RTK GPS system, and Hall-effect sensors at each wheel. The computing power of the Autorally car comes from an Intel Skylake Quad-core i7, it has an Nvidia GTX 750ti GPU, and uses Wifi and Xbee communication. Overall, this system is extremely similar to the F1tenth car, and each has its unique hardware and computing capabilities to test various algorithms remotely.

Autorally also uses the Robot Operating System. The team at Georgia tech has created a public github account located at the following URL: https://github.com/AutoRally/autorally. Here you can find all the code and resources used on the Autorally truck. They also provide walk through instructions on how to run the truck in simulation using Gazebo.

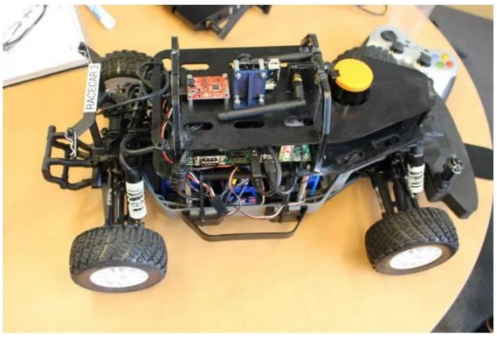
MIT TUNNEL RACECAR

The MIT Tunnel Race is yet another alternative to the F1tenth project which is held at the MIT every Spring Semester. Although the name RACECAR may seem but obvious, it is an acronym that stands for **R**apid **A**utonomous **C**omplex-Environment **C**ompeting **A**ckermann-steering **R**obot that was first developed in the year 2015. On a similar note to F1Tenth this 1:10-scale model cars navigate autonomously through MIT's underground tunnels and compete to finish the dedicated track length in the least amount of time. There are various similarities and differences in F1tenth and The Tunnel Racecar.

As the F1tenth, these cars also run on ROS and have the power of Nvidia Jetson TK1 platform for processing and control. To perceive its motion and the local environment, the race is built with a heterogeneous set of sensors, including a scanning laser range finder (Hokuyo UST-10LX laser), camera (Stereolabs ZED stereo camera), inertial measurement unit (Sparkfun IMU) and Structure depth camera. The ZED camera is used to implement accurate image perception and control algorithms to help the car navigate through the race track without collisions. This acts as the RGB sensor while the Structure camera gives the depth.

The differences include an open-source Electronic Speed Control unit called the VESC, which acts as a wheel odometer. VESC is a custom motor controller having a built in STM32F4 microcontroller, that can provide regenerative braking and odometry data.

MIT Tunnel RACECAR also uses the Robot Operating System. The team has created a public github account located at the following URL: https://github.com/mit-racecar



Methods

ROBOTICS OPERATING SYSTEM (ROS)

Installation of ROS Kinetic

ROS Kinetic supports Wily (Ubuntu 15.10) and Xenial (Ubuntu 16.04).

Step 1. Setup your computer to accept software from packages.ros.org.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc
) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Step 2. Set up your keys

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-ke
y 421C365BD9FF1F717815A3895523BAEEB01FA116
```

Step 3. Installation

First, make sure your Debian package index is up-to-date:

```
sudo apt-get update
```

Then download the full-desktop installation. This includes (but not limited to) - ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators, navigation and 2D/3D perception.

```
sudo apt-get install ros-kinetic-desktop-full
```

Step 4. Initialize rosdep

Before running ROS, we have to check for dependencies and install various system dependencies for the source on which we have to compile. These actually enable some core components in ROS.

```
sudo rosdep init

rosdep update
```

Step 5. Environment Variables

ROS environment variables are automatically added to your bash session every time a new shell is launched:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
```

```
source ~/.bashrc
```

Once all these steps are successfully installed, one can use ROS platform to write and compile code in either C++ or python.

For further detailed instructions, please follow the ROS Wiki for installation on: http://wiki.ros.org/kinetic/Installation/Ubuntu

Getting Familiar with Simulation Environments

We used various simulation environments to analyze and test our robot. We started with a simple 2-D simulator called as TurtleSim and then later went on to implement similar logic in a 3D environment simulated by GAZEBO.

TurtleSim Simulator

Initially an algorithm was developed on turtlesim 2D simulator to cover maximum area in a closed environment using keyboard teleop. The Roomba vacuum cleaner uses simple behaviors to move about a room, until it bumps something and then it adapts (backs up, turns etc.) and starts again. Its goal is to achieve "coverage" of a bounded area without knowing exactly what the space looks like, and without using complex sensors in an "automated" manner. Similar analogous behavior was developed on the turtlesim using manual moving instructions and later in an automated manner. The output was something like this.

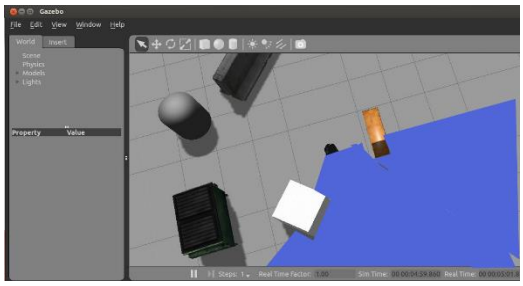
Here we thus got familiar with ROS Nodes and how to use turtlesim. Similar operations were carried on with the turtlebot simulator in gazebo and an autonomous node to move the turtlebot in a square were implemented.

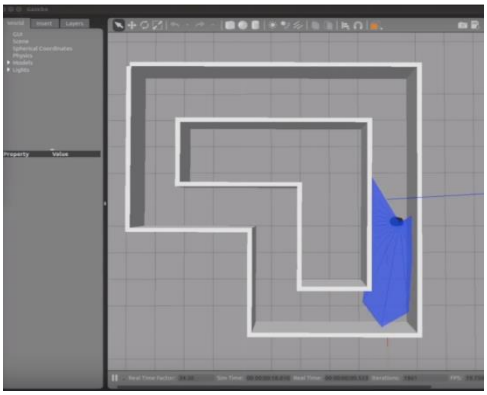
GAZEBO Simulator

GAZEBO is a well-designed simulator that makes it possible to rapidly test algorithms, design robot models, perform regression testing, and train AI system using realistic scenarios. Gazebo was initially developed at the University of Southern California in 2002. However later in the year 2012, Open Source Robotics Foundation (OSRF) overtook the project and now continually maintains and develops Gazebo for diverse robotics simulation.

In this course, Gazebo platform was used extensively to model and develop various automated controls for both the turtlebot and F1Tenth. We first used the simulated environment to test our algorithms like the PD controller using the lidar scan data to navigate through a designed track environment, for doing gmapping, check interactions of various frames (base frame, odom frame etc.) and for various path planning and speed control maneuvers. Initially we performed the keyboard teleop and autonomous navigation in a closed square environment similar to turtlesim for the turtlebot. However, it was observed that the turtlebot tend to drift away from the straight line and could not follow a fixed trajectory.

Next, we used the tutorials on ROS Wiki and learned to spawn a customized URDF (**Unified Robot Description Format** written in XML) robot model on Gazebo, that had various on-board sensors like, camera, lidar, encoders. Later this model was used to navigate through a fixed race track to achieve obstacle avoidance and wall following algorithms using the Lidar scan data. The achieved output is as follows:





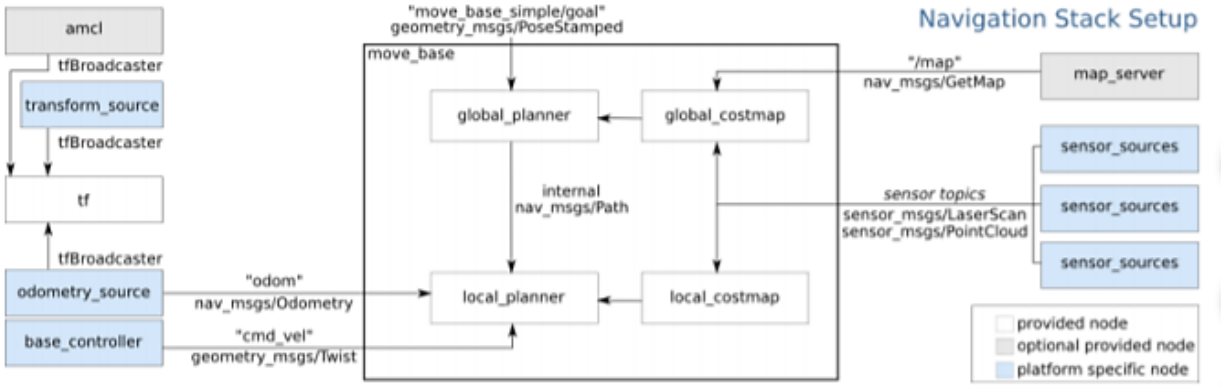
Understanding concepts involved in move_base operation for the Robot

ROS uses the Twist message type for publishing motion commands to be used by the base controller. While we could use almost any name for a topic, it is usually called */cmd vel* which is short for "command velocities". The base controller node subscribes to the */cmd vel* topic and translates Twist messages into motor signals that actually turn the wheels. We can see the components of a Twist message using the following command: *$rosmsg show geometry msgs/Twist*

The Twist message is composed of two sub-messages with type Vector3, one for the x, y and z linear velocity components and another for the x, y and z angular velocity components. Linear velocities are specified in meters per second and angular velocities are given in radians per second.

The Navigation Stack is fairly simple on a conceptual level. It takes in information from odometry and sensor streams and outputs velocity commands to send to a mobile base. there are three main hardware requirements that restrict its use:

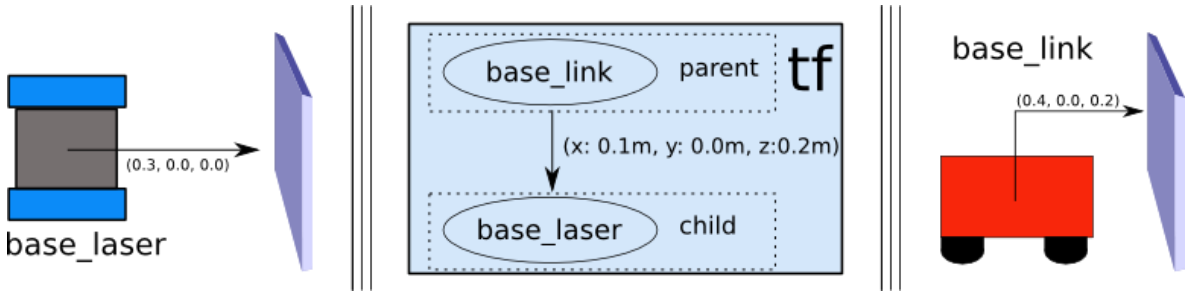
- It is meant for both differential drive and holonomic wheeled robots only.
- It requires a planar laser mounted somewhere on the mobile base.
- The Navigation Stack was developed on a square robot, so its performance will be best on robots that are nearly square or circular.



A general scheme of move_base path planner

- Transform Configuration (tf) : The navigation stack requires that the robot be publishing information about the relationships between coordinate frames using tf. http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF

- Sensor Information : The navigation stack uses information from sensors to avoid obstacles in the world, it assumes that these sensors are publishing either sensor_msgs/LaserScan or sensor_msgs/PointCloud messages over ROS, we followed the tutorials on http://wiki.ros.org/navigation/Tutorials/RobotSetup/Sensors
- Odometry Information : The navigation stack requires that odometry information be published using tf and the nav_msgs/Odometry message. We followed the tutorials on http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom
- Base Controller : The navigation stack assumes that it can send velocity commands using a geometry_msgs/Twist message assumed to be in the base coordinate frame of the robot on the "cmd_vel" topic. This means there must be a node subscribing to the "cmd_vel" topic that is capable of taking (vx, vy, vtheta) <==> (cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z) velocities and converting them into motor commands to send to a mobile base.
- Mapping : The navigation stack does not require a map to operate, but we'll assume you have one. We followed the tutorials on http://wiki.ros.org/slam_gmapping/Tutorials/MappingFromLoggedData



MATLAB's ROBOTICS SYSTEMS TOOLBOX

MATLAB's Robotics System Toolbox provides an interface between MATLAB® and Simulink® and the Robot Operating System (ROS). It allows us to write algorithms for developing autonomous robotics applications with ease, and provides good hardware connectivity for the same. The several inbuilt toolbox algorithms include path planning and path following for differential drive robots, scan matching, obstacle avoidance, and state estimation. We do not experiment with this but the system toolbox also includes algorithms for inverse kinematics, kinematic constraints, and dynamics for manipulator robots.

The toolbox works well with the Gazebo environment, as we will discuss in detail further ahead. It supports C++ code generation and the Simulink External Mode lets you view signals and change parameters while your deployed model is running.

Implementation

Use *rosinit* to initialize ROS. By default, rosinit creates a ROS master in MATLAB and starts a "global node" that is connected to the master. The "global node" is automatically used by other ROS functions.

However, since we primarily communicate with ROS enabled robots over SSH, via remote laptop running MATLAB, it is important that we define the robot as the ROS Master as a MATLAB environment variable as shown below

```
%%%
setenv('ROS_MASTER_URI','http://192.168.1.3:11311')
setenv('ROS_HOSTNAME','192.168.1.3')
setenv('ROS_IP','192.168.1.29')
rosinit
%%%
```

Once ROS core has been initialized, use *rostopic list* to check if the topics being published correctly.

Writing publisher and subscribers in MATLAB is similar to ROS. In fact, it uses the same framework but follows the syntax unique to MATLAB. Most ROS implementations are defined with neat function calls.

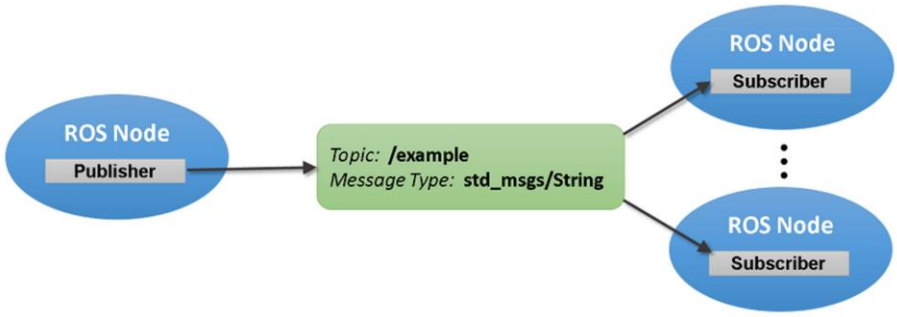


Fig:

For instance, the syntax for defining publishers and sending it its ROS messages are shown below:

```
chatterpub = rospublisher('/chatter', 'std_msgs/String');

chattermsg = rosmessage(chatterpub);

chattermsg.Data = 'hello world');
send(chatterpub,chattermsg);
```

Similarly, interfacing LIDARs and depth cameras with MATLAB were easy to implement.

The following figures show the LIDAR and depth camera results in MATLAB. The codes for the same are in the Appendix

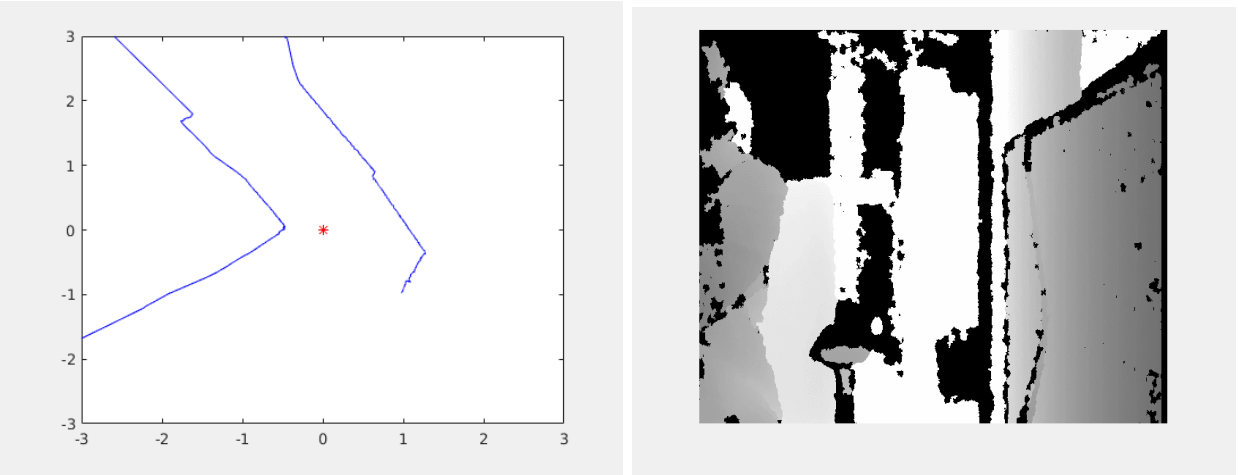
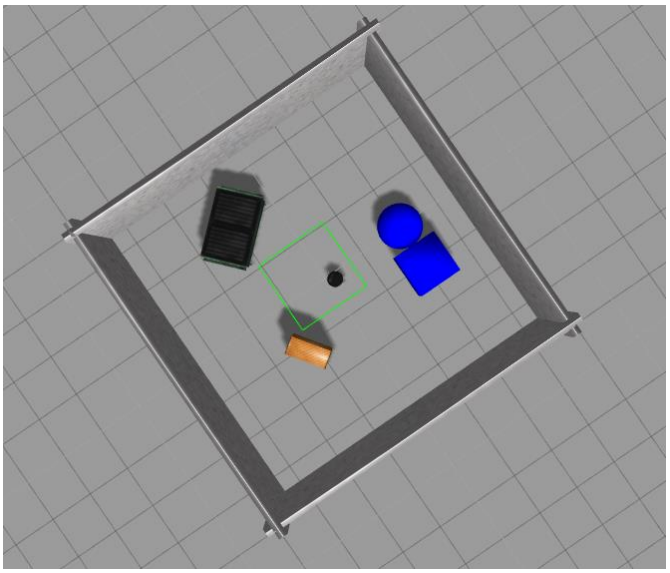


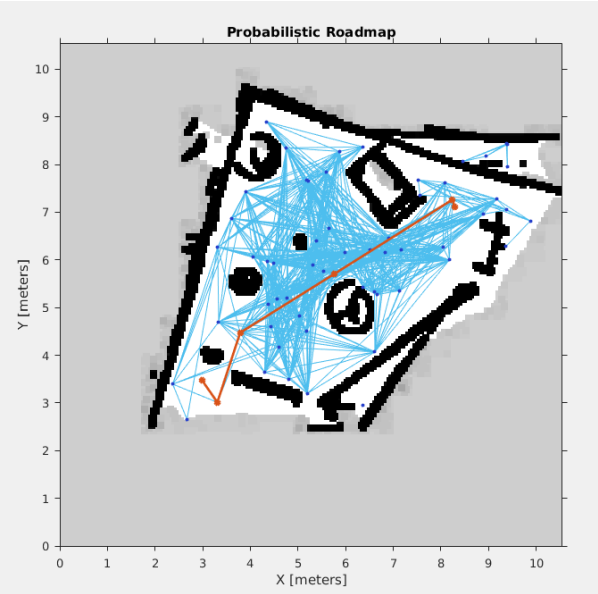
Fig (a) LIDAR in MATLAB. (b) Depth camera in MATLAB

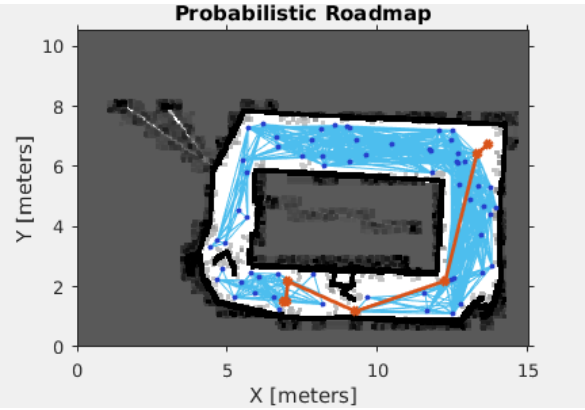
MATLAB can also be used to add and remove objects in Gazebo. Entire Gazebo models can be built through MATLAB alone. The following figure shows an environment that was built solely in MATLAB.



Fig

Probablistic Road Mapping is another function call in MATLAB that makes path planning easy. While this is good function call and simple to implement, it is far from perfect. It is best to write one's own code.





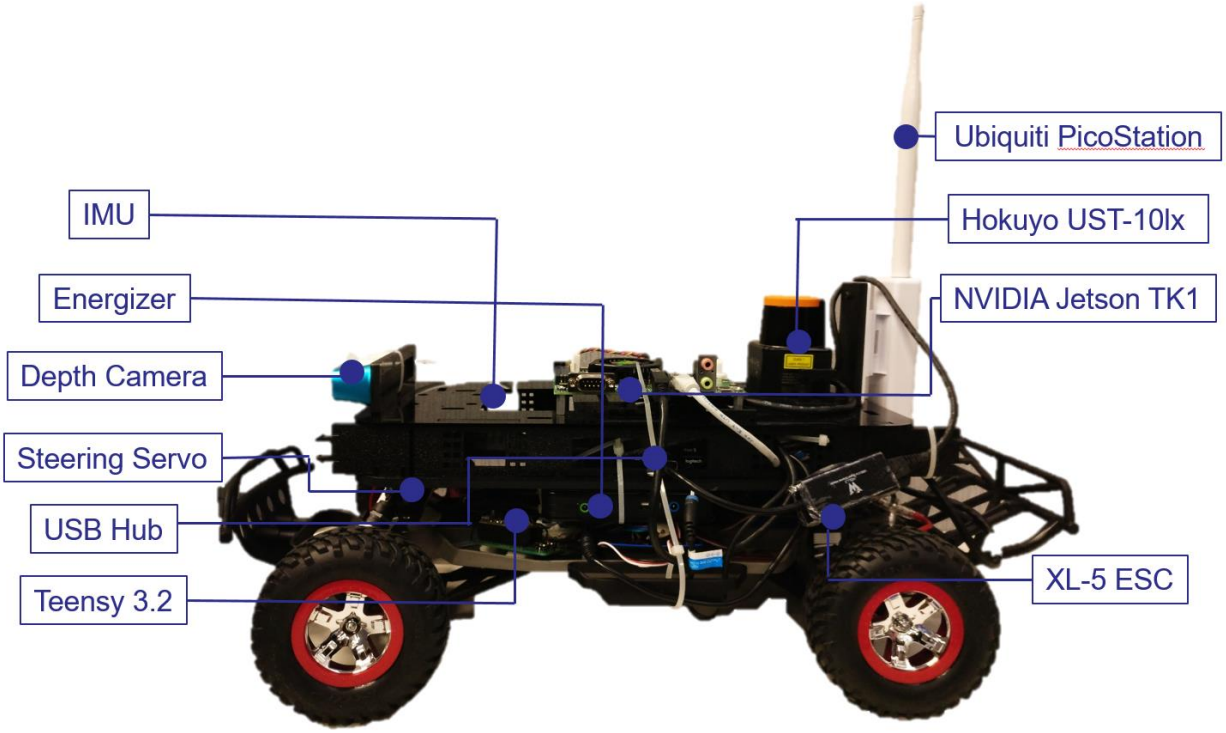
The figures above show the implementation of the same in MATLAB. One is of a Gazebo environment and the second figure is from a real map of the F1tenth track.

Implementation

BUILD

Part list

Hokuyo UST 10lx	IMU
Teensy 3.2	Ubiquiti PicoStation
USB Hub	NVIDIA Jetson TK1
Structure Sensor	XL-5 ESC
Steering Servo	Energizer



The figure above describes the build and layout of the F1tenth car as implements by us. The chassis given on the website was meant for an older version of the car that is now out of production. The new model was smaller and required custom fixtures in the front and rear to fix the chassis firmly to the car.

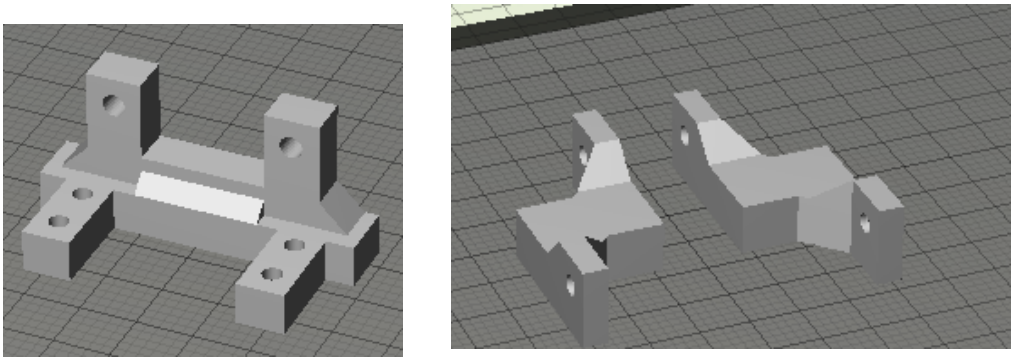


Fig shows (a) rear and (b) front fixtures for the chassis.

SYSTEMS

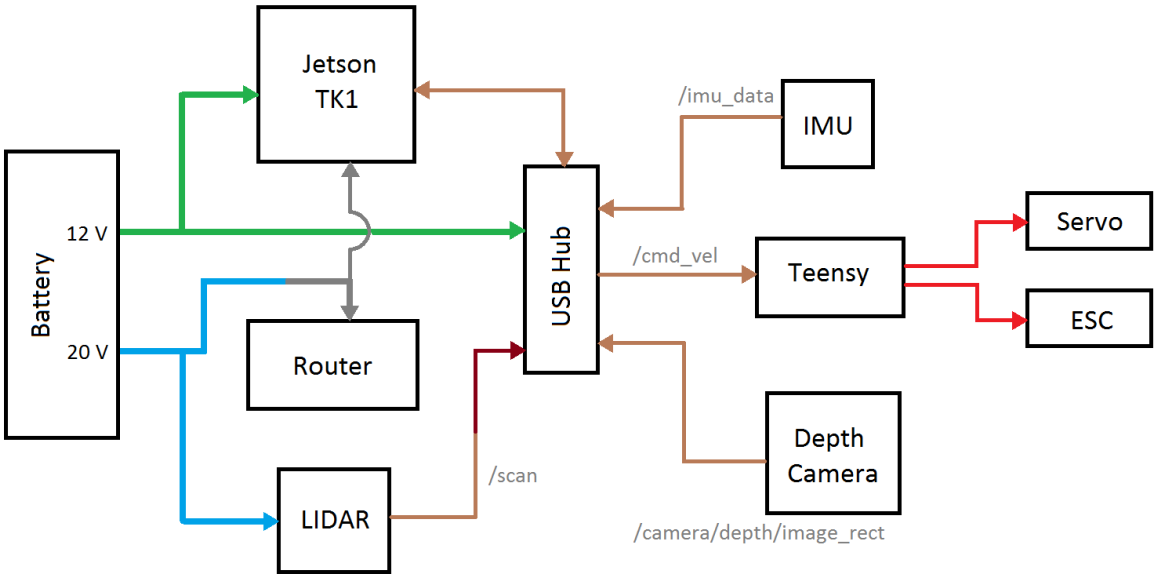


Fig: System architecture and connections of the parts in the F1tenth car.

In the above figure, the bold lines represent power channels, and the thin lines represents data cables. The gray wire is the power over Ethernet cable connecting the Jetson the router along with its power supply. The maroon wire represents the Ethernet to USB converter. The main topics subscribed or published to the sensors are shown in gray. This is not a comprehensive list of topics used, but a good idea of the software and hardware implementation of the robot.

We will now discuss and describe the implementation of each of the parts in detail.

WIRELESS ROUTER

Router used: Ubiquiti PicoStation

The PicoStation is connected to the Jetson TK1 with a PoE cable. The Ethernet port goes to the TK1 and the power cable is connected to the 20 V port of the Energizer via a custom-built cable as shown in the picture above. The router is connected following the instructions given in the f1tenth manual.

BATTERY

Battery used: Energizer

This is a battery pack that supplied power to everything except the motors (which have their own battery). There are two voltage levels – 9V and 16 V and together they power the TK1, PicoStation, LIDAR, IMU, Teensy Depth Camera and the USB Hub.

COMPUTING PLATFORM

Motherboard used: NIVIDIA Jetson TK1

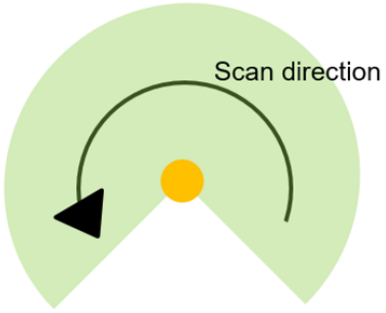
The NVIDIA Jetson TK1 motherboard runs a light version of Ubuntu 14.04. ROS indigo installation process for the TK1 was similar to any other installation on Ubuntu except for some notable exceptions, like the inability to run graphics intensive applications like rviz or Gazebo. The installation of ROS on Ubuntu in the TK1 follows the same process described above.

Fig: The wifi router was connected to the TK1 and Energizer via a power-over-Ethernet cable

LIDAR

LIDAR used: Hokuyo 10LX UST

Specifications:



Input Voltage	12-24 VDC
Current	150 mA (450 mA on startup)
Detection Angle	0 – 270 deg
Resolution	0.25 deg
Measurement steps	1081
Interface	Ethernet
Scan rate	40 Hz

The Energizer battery powers the LIDAR. It draws it's power from the 20 V port in parallel with the Ubiquiti PicoStation. A custom chord is soldered together connecting the three. The blue and gray wires on the Hokuyo is soldered to a barrel jack connector and the remaining wires are ignored for the nonce. The Ethernet cable is connected to an Ethernet to USB adapter that goes to the USB hub. This forms the second Ethernet connection on the TK1.

The IP address of the LIDAR was changed by following the instructions given on the F1Tenth manual. The urg_node was installed and the LIDAR was streamed data successfully as was visible in rviz. The general steps for streaming data over ssh is given in [].

sudo apt-get install ros-indigo-urg-node

rosrun urg_node urg_node _ip_address:=192.168.1.3

For visualizing laser data in rviz change the fixed frame topic from /map to /laser and select /scan topic under LaserScan.

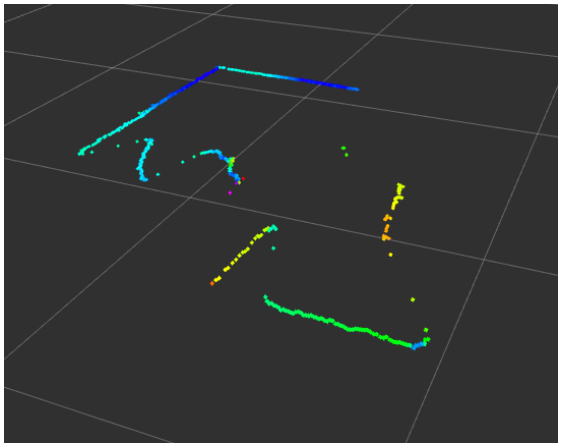


Fig: Laser scan in rviz

Bridging Connections

Two Ethernet ports are connected to the TK1. The LIDAR is connected to eth2 and the PicoStation to eth0. The two hardware in the network need to be bridged together under one IP address so they communicate seamlessly with the TK1. *bridge_utils* is an Ubuntu Application that creates a bridge between eth0 and eth2. Once the bridge is established, any external computer ssh-ing into the TK1 can access the LIDAR.

However, this gives us only a temporary bridge.

For some reason, the TK1 does not allow us to set up a persistent bridge to connect the TK1, Hokuyo and PicoStation by editing the /etc/network/interfaces file. Making any changes to this (empty) file causes the network manager to crash. Therefore, a crude fix to this problem was to edit the /etc/rc.local file and copy the terminal commands for bridging in it. Any commands in rc.local are executed as soon as the system boots. Sometimes this can cause a problem if the bridge is formed before the individual networks are identified by the TK1.

Adding a 45 s sleep command (sleep 45s) before executing the brctl commands solves this issue.

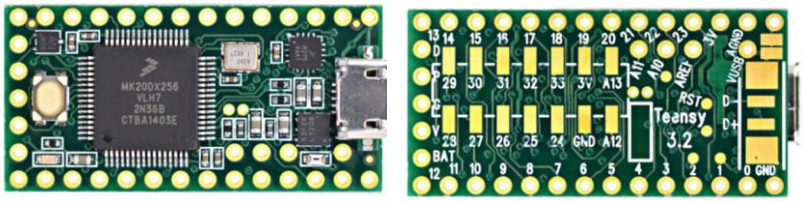
```
rc.local ×
1 #!/bin/sh -e
2 #
3 # rc.local
4 #
5 # This script is executed at the end of each multiuser runlevel.
6 # Make sure that the script will "exit 0" on success or any other
7 # value on error.
8 #
9 # In order to enable or disable this script just change the
  execution
10 # bits.
11 #
12 # By default this script does nothing.
13
14 sleep 45s
15
16 brctl addbr br0
17 brctl addif br0 eth0
18 brctl addif br0 eth2
19 ifconfig br0 192.168.1.3 netmask 255.255.255.0
20
21 exit 0
```

Fig: rc.local edited

ARDUINO/TEENSY

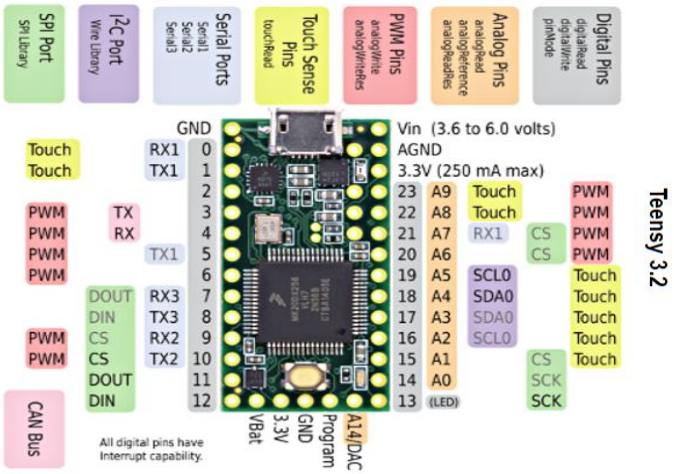
Teensy

Teensy is a complete USB-based microcontroller development system, in a very small footprint, capable of implementing many types of projects. In this project, we are using a 40 pin Teensy 3.2 Micro-controller board that has MK20DX256VLH7 Cortex M-4 ARM Core Processor manufactured by NXP USA Inc. It has a 32-bit core size with 256 KB of FLASH Program memory and a crystal of 72 MHz. The Teensy 3.2 has 12 PWM pins with a RTC that requires a 32.768 kHz crystal & 3V battery. It is compatible with Arduino Software & Libraries and can directly be interfaced with the Arduino IDE after few setup instructions.



Features of Teensy 3.2

The Pinout diagram for the Teensy 3.2 with various features is as shown



We mainly use Teensy 3.2 to send PWM control signals to the Servo motors for setting steering and throttle commands for the F1Tenth Car. The steering control is directly applied to the servo while the throttle is controlled using the ESC. Pins 4 and 5 on the Teensy 3.2 are used to generate the PWM signals for speed and steering control respectively. This approach is used by UPenn F1Tenth car, on similar lines the MIT tunnel Race car uses VESC to control the entire motion of the car, thereby eliminating the need of teensy controller.

Connections to the Teensy

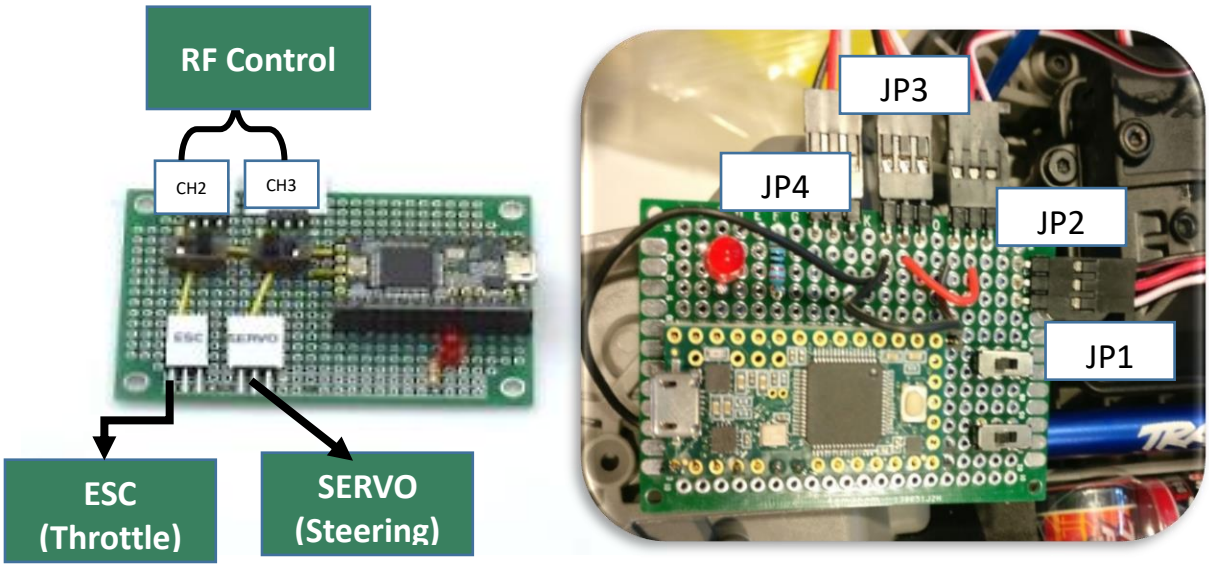


Fig: (a) connections to the teensy, (b) connection on out teensy

The connections on the Teensy allow one to switch from manual (radio controlled) mode and autonomous (TK1) mode. The connections of the Servo and ESC from channel 2 and 3 on the radio are fed into JP1 and JP2 of the Teensy board respectively. In our model, channel 2 controlled the steering servo and channel 3 controlled the ESC. Connections on JP3 and JP4 were fed back into Channel 2 and 3 respectively. The signal connection of JP3 and JP4 were connected to pin 4 and 5 of the Teensy.

Interfacing Teensy 3.2

Setting up the Teensy for use on the TK1 took several steps. The new Teensy board was not compatible with the version of Arduino IDE the TK1 could run. The TK1 is meant to run Arduino 1.5 and lower while the Teensy 3.2 is compatible with Arduino IDE 1.6 and higher. However, the Teensy works just fine when it is programmed separately. So first, we manually downloaded and installed (./install.sh) the Arduino 1.8.2 on the group laptop. It needs to be installed in a place without root access. The installation will throw an error due to a minor bug in the installation file.

Edit the install.sh fine and replace "RESOURCE_NAME=cc.arduino.arduinoide" to "RESOURE_NAME=arduino-arduinoide"

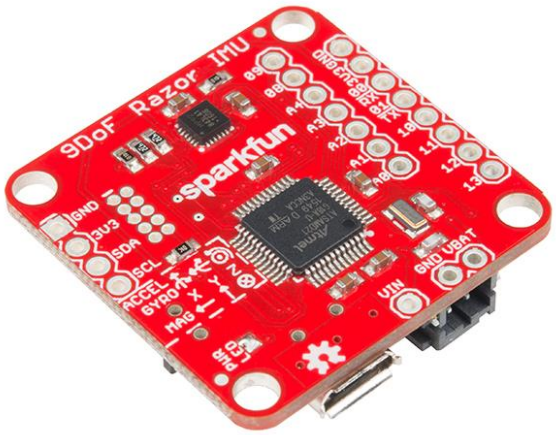
Download TeensyDuino from (https://www.pjrc.com/teensy/td_download.html) and follow the steps given for the install. Note that the TeensyDuino needs to be converted to an executable first. **Teensyduino** is a software add-on for the Arduino, to run sketches on the Teensy and Teensy ++. Most programs written for Arduino work on Teensy. All the standard Arduino functions (digitalWrite, pinMode, analogRead, etc) all work on Teensy. Teensyduino is also compatible with many Arduino libraries.
Install rosserial (http://wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup) and rosserial_arduino on the laptop and TK1. Follow the instructions given for ros_lib in the link above for the laptop. Rosserial is a package that converts Arduino commands to ROS commands. We then add the teensy board to Arduino boards and checked if the right board is being displayed and connected.

Once the installation was complete we uploaded Sagar's code on the Teensy[x].

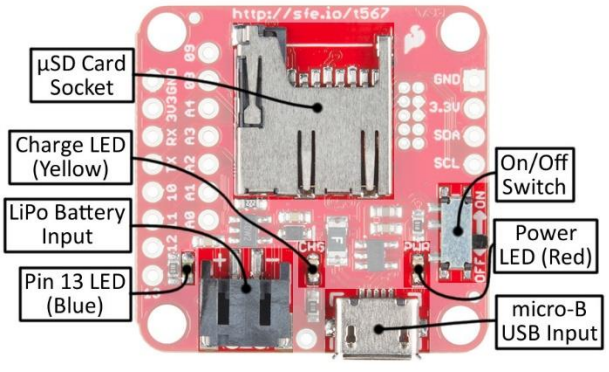
Teleoperation: Install the ros_teleop_keyboard on the TK1 and use it to send commands to the Teensy. SSH into the TK1 and execute the following steps:

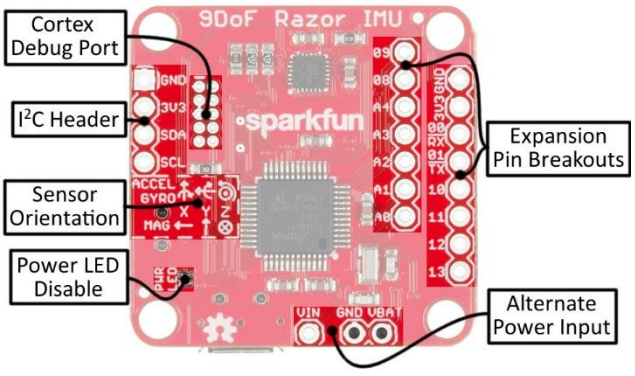
```
roscore

rosrun rosserial_python serial_node.py /dev/ttyACM1

rostopic pub /cmd_vel geometry_msgs/Twist [0,0,0] [0,0,0] (this will cause the blinking green light on the ESC to turn red)

rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

INERTIAL MEASUREMENT UNIT (IMU) :



The IMU sensor used in the F1tenth project was the SparkFun 9 Degree of Freedom Razor IMU MO SEN-14001. This IMU is a combined 9DoF IMU and a SAMD21 microprocessor. This combination allows for reprogramming of the IMU and allows for memory to be stored directly on the sensor. The sensor features three 3-axis sensors, an accelerometer, gyroscope, and a magnetometer. These have the ability to supply measurements of the linear acceleration, angular rotation velocity, and orientation based on magnetic field vectors. The IMU comes with preprogrammed example firmware that can transmits data over a serial port. There also exist multiple Arduino libraries and firmware on the internet that can be uploaded on to the IMU and customized to perform specific tasks and output certain data. In the images below, you can observe the assembly of the sensor and where the various components are located:





I. IMU Setup

For use through ROS and on the F1tenth car, the IMU needed to be configured through the Arduino IDE. Before uploading any firmware to the sensor, some support libraries needed to be added to communicate with the Razor IMU. The first library was the Arduino SAMD Boards library for communication between the SAMD21 processor and the IMU. This can be found using the board manager. The path to the board manager is a follows:

Tools -> Board -> Boards Manager

Search Tool bar: Arduino SAMD Boards(32-bits ARM Cortex-MO+)

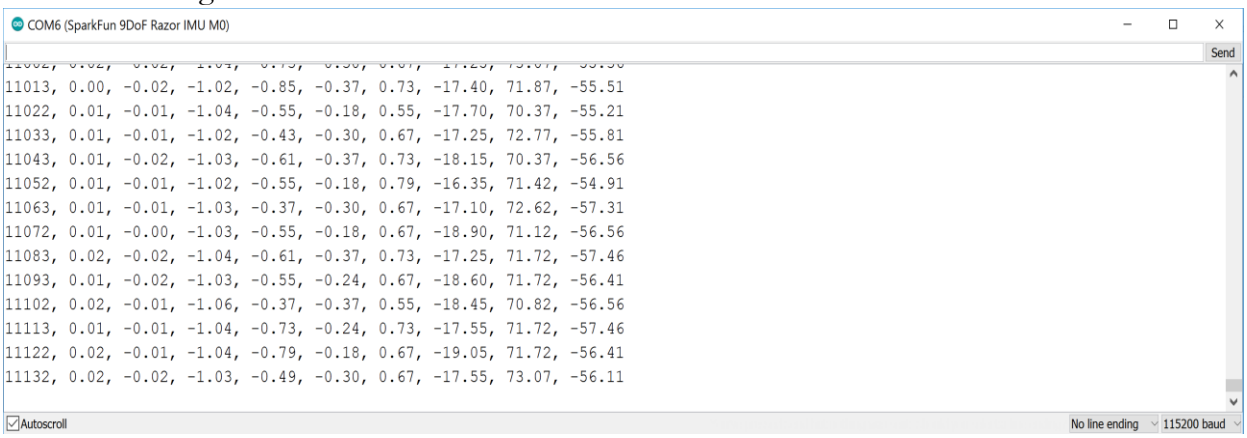
The last part of setup for the SAMD board is to open the preferences under the file tab and locate the Addition Board Manager URLs textbox, and the following URL needs to be copied and pasted in:

https://raw.githubusercontent.com/sparkfun/Arduino_Boards/master/IDE_Board_Manager/package_sparkfun_index.json

The Razor IMU also needs libraries to be installed to run the example firmware uploaded to the sensor. These two libraries are the SparkFun MPU-9250 Digital Motion Processing (DMP) library and the FlashStorage Arduino library. The DMP library configures the sensors and features quaternion calculation, gyroscope calibration, and orientation detection, all of which are useful in the F1tenth project. The flash storage library is optional but allows the SAMD21 to store the serial commands output by the sensor.

Once the necessary libraries are downloaded in Arduino, the firmware needs to be downloaded so it can be uploaded to the IMU. The download for the libraries and the firmware, as well as a step by step tutorial for setting up the IMU can be found at the following on learn.sparkfun.com. When the firmware is downloaded, it is important to make sure that both the firmware .ino file and the config.h file are both open when the uploading to the IMU.

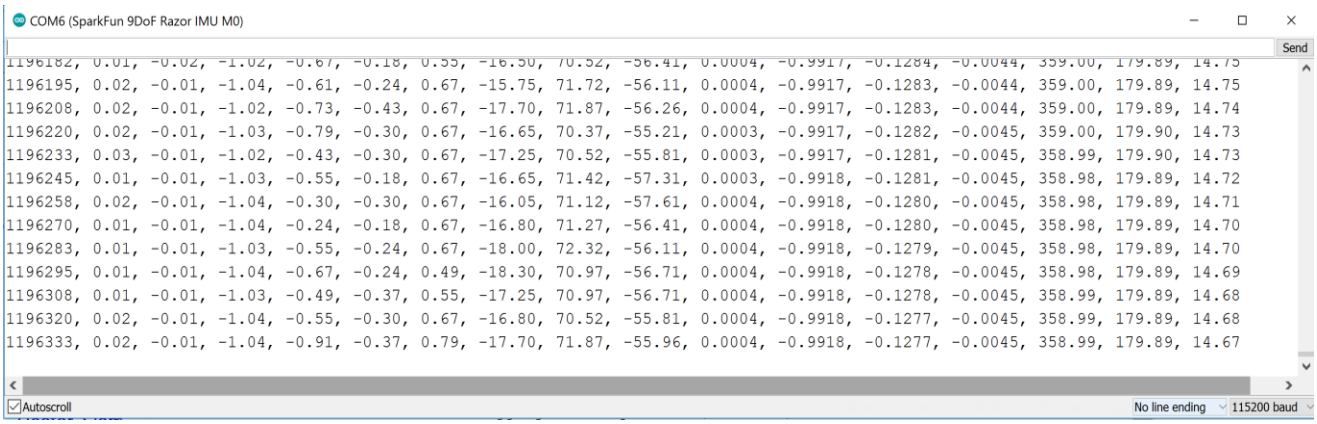
Now that the libraries have been downloaded and the firmware has been uploaded to the IMU, the sensor can now output data to the serial plotter in the Arduino IDE. Without making any adjustments to the code or the serial plotter, along with the current time step, the only data that is displayed are the readings from the accelerometer, gyroscope, and the magnetometer, as seen in the image below.



In the top portion of the serial plotter is a text box where commands can be given. The firmware uploaded to the boards has corresponding commands that will enable various readings to be printed in the serial plotter. The image below shows the commands to display the specific data you require.

- (SPACE) – Pause/resume serial port printing
- `a` – Turn accelerometer readings on or off
- `g` – Turn gyroscope readings on or off
- `m` – Turn magnetometer readings on or off
- `q` – Turn quaternion readings on or off (qw, qx, qy, and qz are printed after mag readings)
- `e` – Turn Euler angle calculations (pitch, roll, yaw) on or off (printed after quaternions)
- `c` – Switch to/from calculated values from/to raw readings
- `r` – Adjust log rate in 10Hz increments between 1-100Hz (1, 10, 20, …, 100)
- `A` – Adjust accelerometer full-scale range. Cycles between ± 2, 4, 8, and 16g.
- `G` – Adjust gyroscope full-scale range. Cycles between ± 250, 500, 1000, 2000 dps.
- `s` – Enable/disable SD card logging

A command not listed here, which is unique to the IMU being used (SEN 14001) is the 'h' command, which displays the heading. If all calculations are enabled, minus the heading measurement, a total of 17 numbers are displayed on the serial plotter, and they follow the order of the commands above. The full serial plotter with all calculations enabled can be viewed in the image below.



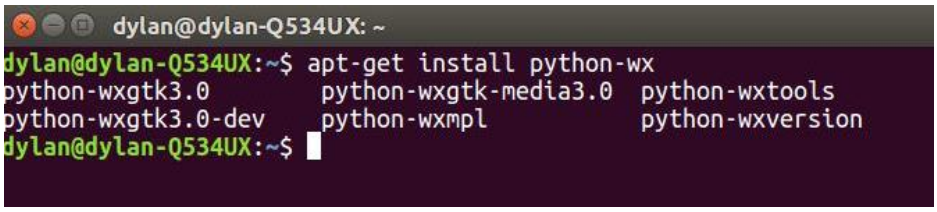
Once the desired calculations are displayed in the serial plotter, the next challenge is to use this data with ROS and publish the information on the necessary topics so they can be easily accessed.

II. Communication through ROS:

The IMU sensor used (Sen 14001) is not the same sensor used for the F1tenth tutorial when interfacing the IMU sensor because it is a newer model than previously used. However, the packages used in the tutorials still need to be downloaded, and the tutorials can be followed

for the installation. The main package used was the razor_imu_9dof package. This package contains all the files needed to create a customized script to integrate our sensor and communicate through ROS. The razor_imu_9dof package contains a subdirectory named, nodes. Contained inside this directory is a file called, 'imu_node.py', and this is the script that needs to be edited so the serial data can be observed through ROS. The customizable script will read the serial output that was seen in the serial plotter in the Arduino IDE. Therefore, it is important to specify which calculations are desired for ROS communication in the serial plotter before leaving the Arduino IDE. The IMU, if using the flash storage library will retain the serial plotter commands and always display the commanded calculations. For example, if the Euler angles are desired, which are values in columns 15 – 17, and the commands were not given to display those values in the serial plotter, they will not be available to use in ROS communication using a python script.

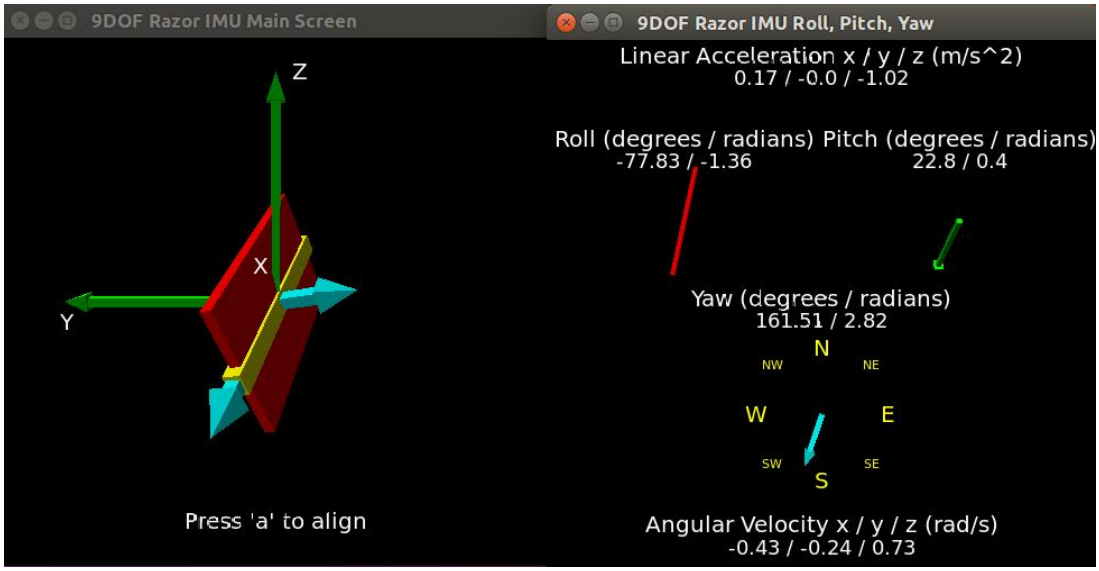
The second package that needs to be installed is the 'python-visual' package. This package opens a graphic interface when the launch file, 'razor-pub-and-display.launch' is ran. On the first attempt to run this launch file, an error was returned. In the display_3D_visualization.py file, there is a library imported called, 'wx', which needs to be installed in order to launch the interface. The following commands will install the necessary software to run this library and return no errors:



apt-get install python-wxversion

apt-get install python-wxgtk3.0

Once these packages are installed, the launch file can be run and will help visualize the orientation and respective measurements from the various IMU orientations. This node would be extremely helpful while SSH'ed into the F1tenth vehicle. You would be able to get an accurate visualization of the forces applied to the vehicle as it is driving around the racetrack. In the image below you can see an example of the interface used for this node.



III. Python Script:

To solve the problem of reading the serial data from the IMU, a python script can be used. The most important part of the code is specifying the port at which the IMU is connected. If the firmware and libraries were downloaded in Windows, this will have a different port name than the port on Ubuntu. Arduino can be used on Ubuntu to determine which port is used for the connection. The port used on group B's computer and for the F1tenth car was, '/dev/ttyACM0'.

This same port value needs to be edited in the razor_imu_9dof package in the file my_razor.yaml. This is a simple task once the port is determined. The second line of the my_razor.yaml file is the only line that needs to be edited.

```
default_port='/dev/ttyACM0'
port = rospy.get_param('~port', default_port)

# Check COM port and baud rate
rospy.loginfo("Opening %s...", port)
try:
    ser = serial.Serial(port=port, baudrate=115200, timeout=1)
except serial.serialutil.SerialException:
    rospy.logerr("IMU not found at port "+port + ". Did you specify the correct port?")
    #exit
    sys.exit(0)
```

Once the port is initialized, a serial object can be declared, which will help read the displayed data from the IMU, as seen in the sample code above.

The next important piece of the python script is reading the serial data. Three important lines of achieve this task.

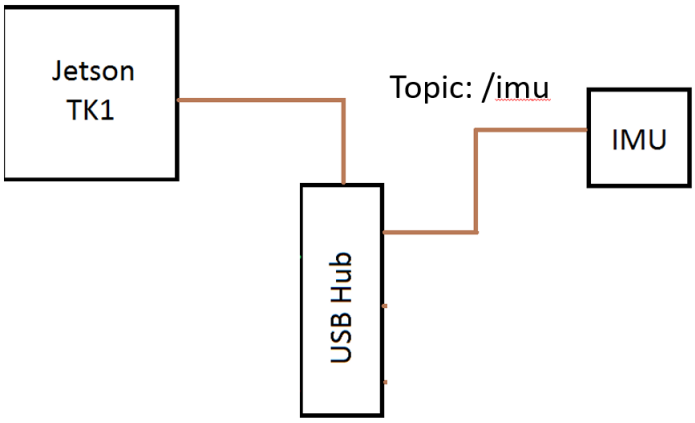
```
line = ser.readline()
line = line.replace("#YPRAG=","")   # Delete "#YPRAG="
words = string.split(line,",")    # Fields split
```

The first line of this code reads the entire row from the serial data. The second line deletes the, '#YPRAG= ', from the serial data row, and the third line separates each of the values by the comma in between each of the values. This means the length of words, assuming all calculations are displayed, will be 17. We can now use the data contained in the 'words' variable to publish data to the IMU topic. The code for publishing this data would look like the following:

```
imuMsg.linear_acceleration.x = float(words[1])
imuMsg.linear_acceleration.y = float(words[2])
imuMsg.linear_acceleration.z = float(words[3])

imuMsg.angular_velocity.x = float(words[4])
imuMsg.angular_velocity.y = float(words[5])
imuMsg.angular_velocity.z = float(words[6])

imuMsg.orientation.x = float(words[7])
imuMsg.orientation.y = float(words[8])
imuMsg.orientation.z = float(words[9])
```

Now, subscribing to the /imu topic in ROS, will allow the F1tenth car and its onboard TK1 to receive data from the IMU. The, 'float(words[x]) can be changed to any of the 17 calculations displayed in the serial plotter from the Arduino IDE. This shows why it is important to display all the calculations in the Arduino serial plotter, this way you can see all the calculations using the python script without having to go back to the Arduino IDE to display more calculations.

IV. IMU Integration:



The IMU is plugged into the USB hub, which is then directly connected to the Jetson TK1. The IMU communicates on ROS through the /imu topic where the data is transmitted through geometry_msgs. In the image below, the message type that will be available on the /imu topic is displayed:

```
geometry_msgs/Quaternion orientation
float64[9] orientation_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance # Row major x, y z
```

V. Implementation on the F1tenth Car:

The IMU has multiple sets of data that could be useful in several different scenarios. The initial thought was to use the IMU to determine the dead reckoning/odometry data of the vehicle. This posed some challenges as there is a significant amount of noise in the sensor, and since the most useful data when it comes to calculating position would be the linear acceleration and angular velocity, the double integration and single integration that needed to occur to determine the vehicles position would produce significant errors as time increased. In order to integrate the IMU into the F1tenth vehicle and provide accurate odometry data for the vehicle position on a certain racetrack would require interfacing with other sensors, or the use of a wheel encoder. Therefore, in the scope of this project, we are using the IMU sensor purely for data measurement as the car completes its time trails and obstacle avoidance around the race track.

AUTONOMY

CONTROLLER

1. Basic information of the controller

1.1. Task Description

The controller needs to be properly designed so that the vehicle can perform the following tasks during the test:

1. The vehicle can finish the test track shown in Figure 1 at both low speed and high speed.
2. The vehicle should not hit the walls.
3. When there are obstacles in the track, the vehicle should be able to dodge them.
4. The vehicle should stop when the track is fully blocked.

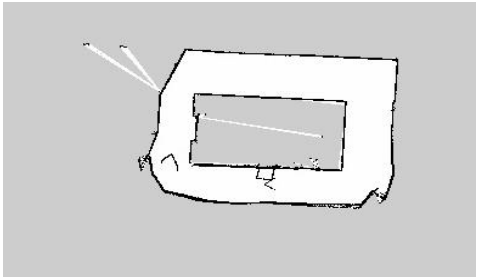


Figure 2 Test track sketch

To deal with those tasks, we implemented a vehicle motion controller with Python, and run the controller on TK1. The controller takes measurement from the Hokuyo LIDAR (subscribe to topic '/scan', message type 'LaserScan'), and output the steering and throttle value to the vehicle (publish to topic '/cmd_vel', message type 'Twist').

1.2. Controller structure

The structure of the controller is shown in Figure 2. The laser scan data from the LIDAR is first pre-processed and the control indices are calculated. The control indices are fed into two controllers: the speed controller and the steering controller.

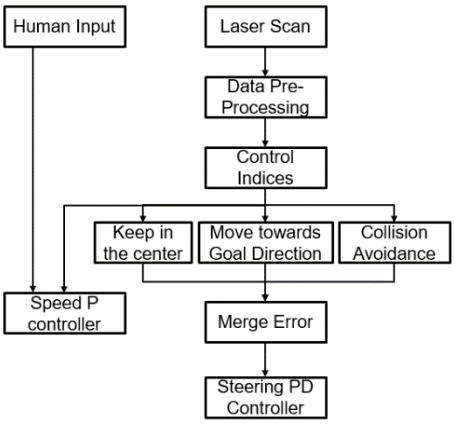


Figure 3 Controller structure

The speed controller is a simple proportional controller. It takes human input from keyboard (subscribe to topic '/keyboard/keyup', message type 'Key', needs additional rospackage 'keyboard') to set the minimum speed, and adjust the speed based on distance information.

The steering controller is a PD controller. The input error consists of three components. The most important component is the 'Move Towards Goal Direction (MTGD)' module. This component is generated by finding the area that has the most area (potential field), and is the footstone of the steering controller. To compensate for the deficiencies of the simple MTGD component, two other components are introduced. One is to help the vehicle avoid obstacles under certain situations, and the other one is to make the vehicle stay in the center of the track.

1.3. Trade-off between front mount and rear mount LIDAR

Before start with detailed control design, we need to decide the positon of the LIDAR first, since that may affect the control strategy a lot. Two different LIDAR mounting positions are shown in Figure 3.

Putting the LIDAR on the front axis will be good for braking and avoiding obstacles in the front because it will get even readings from obstacles at all directions. However, the price is that the readings will be very unstable since the LIDAR is affected by both the orientation of the vehicle and the lateral movement of the front axis when the vehicle is turning. Unlike the front axis, the rear axis will not have lateral movement when the vehicle is turning. So the stability of the readings is better if the LIDAR is on the rear axis.

The characteristics described above is very critical to our controller. As mentioned before, our controller utilizes potential fields algorithm and the vehicle will move to the direction that has the most available area. The R/C vehicle is an Ackerman drive vehicle so the rotation

Figure 4 Front mount vs. rear mount radar

center is always on the same line with the rear axis. That means by mounting the LIDAR on the rear axis, we can get a more accurate desired moving direction. What's more, if the LIDAR is mounted on the front axis, the vehicle will decide to make a turn once the front wheels have passed the obstacle, then the rear wheels may have a risk of hitting the edges of the obstacle.

Considering all those factors, we finally decided to mount the LIDAR on the rear axis of the vehicle.

2. Data pre-processing

The Hokuyo LIDAR we are using will return distance data in the range from -135°(right) to +135°(left), as shown in Figure 3. The scan increment is 0.25°, so there will be 1080 laser points in total. The maximum detecting distance of the LIDAR is 10 meter.

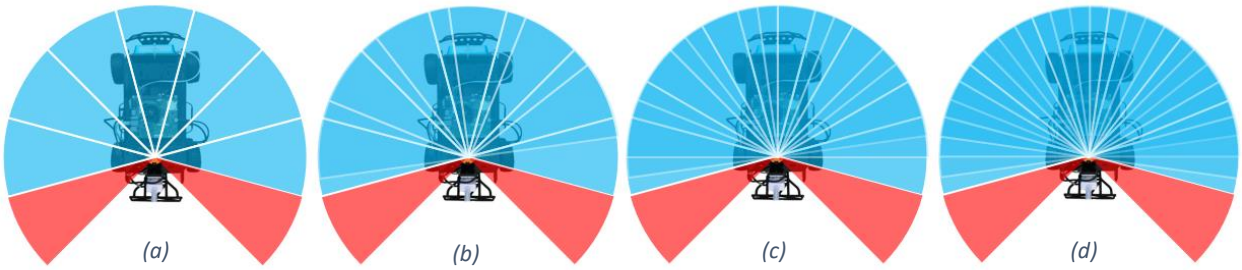


Figure 5 Dividing LIDAR scan data into sectors

Since we need to find the area that has the largest area to navigate the vehicle, we need to divide the LIDAR readings into sectors. First we trim off the data from the rear side and only consider data from the front between -105°(right) and +105°(left), as shown in Figure 4 (a). In our first attempt we divided the front 210° area into 7 sectors, each sector is 30° wide and there are 7 sectors in total. But in the test we found that only 7 sectors lead to a very poor control accuracy and the vehicle will hit the wall or obstacle easily. So we

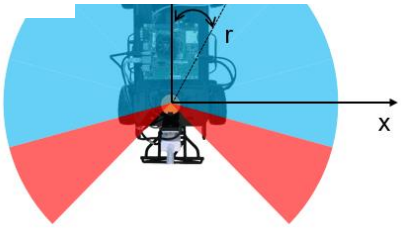


Figure 6 LIDAR data conversion

implemented overlapped sections instead, as shown in Figure 4 (b) – (d). We kept the sector's width to be 30°, while reducing the sector step from 30° to 7.5°, resulting in 25 sectors in total.

Since the LIDAR's effective detecting range is 10 meter while the track's width is no wider than 2 meters, we limit the upper bound of the laser reading to be 3 meter to avoid the extraordinary large readings causing confusion to the controller. The useful control indices are then calculated based on the laser scan data. The control indices are listed in Table 1 and the data format and conversion are described in Figure 5.

Table 1 Control indices

Index Symbol	Description
r_{avg}	Average range of each sector. Represents the available area in a sector
r_{min}	Minimum range among all laser points.
θ_{min}	The corresponding position of the minimum range.
$\|x\|$	Distance along X axis of each laser point.
$\|x\|_{min}$	The minimum distance along X axis.
$\theta_{\|x\|min}$	The corresponding position of the minimum X distance.
$r_{\|x\|min}$	The corresponding range of the minimum X distance

3. Speed Controller

As described above, we used a simple P controller to control the vehicle speed. The structure of the speed controller is described in Figure 6. The speed controller takes input from keyboard by using the keyboard rospackage. The up and down arrow key will adjust the lower speed limit spd_{min} and the controller will calculate the vehicle speed in proportion to $|x|_{min}$. The vehicle will stop if the space key is pressed or the average range of the middle sector $r_{avg}(middle)$ is below the minimum allowed threshold $r_{avg-critical}$. The functionality of the speed controller can be described by the following equation:

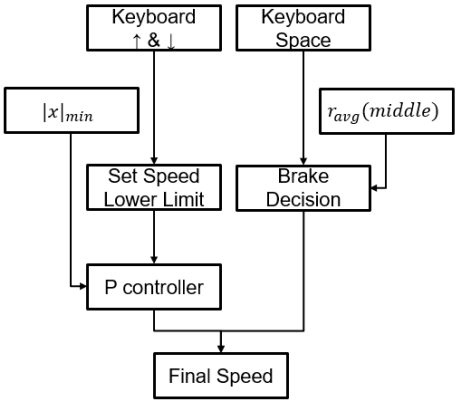


Figure 7 Speed controller

$$spd = \begin{cases} 0, & space\ pressed\ or\ r_{avg}(middle) < r_{avg-critical} \\ spd_{min} + k|x|_{min}, & else \end{cases}$$

4. Steering Controller

As described before, the steering controller has three functionality components: move towards target direction, keep in the center of the track, and avoid obstacles.

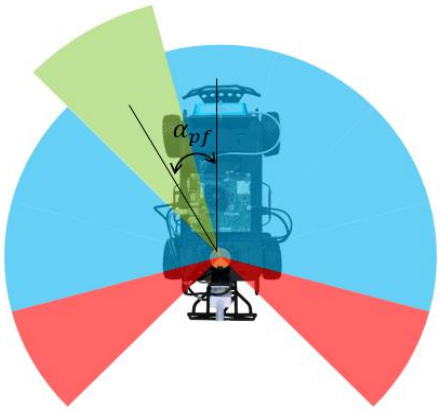


Figure 8 Find the potential field

4.1. Move to the potential field

The potential field is found by finding the sector that has the largest average distance, as shown in Figure 7. The angle between the target sector and the center line α_{pf} is the first component of the input error to the steering PD controller. Since the step size of the sectors is 7.5°, the resolution of α_{pf} is 7.5° also, which is accurate enough for the control of the vehicle.

4.2. Collision avoidance

The potential field component alone is not enough to drive the vehicle to dodge obstacles. Figure 8 (a) and (b) shows two situations where the vehicle may hit the obstacles due to the imperfections in the potential field algorithm.

In Figure 8 (a), the vehicle's front axis has passed the corner of the wall and the left front sector has the largest area. In this case the vehicle will decide to turn left. However the rear axis of still hasn't passed the corner, if the vehicle turns left now, the left rear wheel may hit the corner of the wall.

In Figure 8 (b), the obstacle is only overlapped with less than 10% of the vehicle's front area. The center sector fails to detect the obstacle and is having the largest area so the vehicle will decide to go straight. In this case the left front wheel will hit the obstacle.

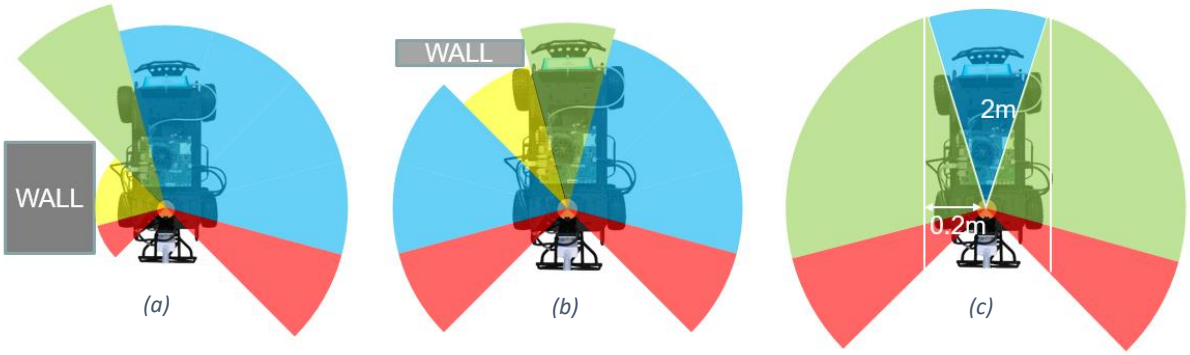


Figure 9 Collision avoidance

It will be hard to use raw distance data to avoid those two collision situations since the critical distance to avoid obstacle varies as the direction of the obstacle changes. However, the width that the vehicle needs to pass through is always fixed. If the vehicle can always monitor the area within a very short distance on each side, forming a narrow 'path', and move towards the side that has more space when there's obstacle intruding in to the path, then the obstacle can be avoided.

In Figure 8 (c), the M shaped area is the 'path' that we selected. First we trimmed the laser scan range down from 3 meter to 2 meter, and limit the calculated $|x|$ to be no larger

than 0.2m. We then calculate the sum of $|x|$ in the left sector, S_{left}, and the sum of $|x|$ in the right sector, S_{right}. The second error component can be calculated based on the difference between S_{left} and S_{right}:

$$\alpha_{avoid} = \sqrt[4]{S_{left} - S_{right}} \frac{|S_{left} - S_{right}|}{S_{left} - S_{right}}$$

To reduce the sensitivity of this algorithm, the lateral distance is only considered between -105° and -10°, and between +10° and +105°. Finally, to increase the stability of the

4.3. Keep the vehicle in the center

With the existing two components, the vehicle can meet the basic requirement of the test. However, the vehicle cannot pass through a corner smoothly at low speed. The vehicle will first move very close to the corner due to the potential field component,

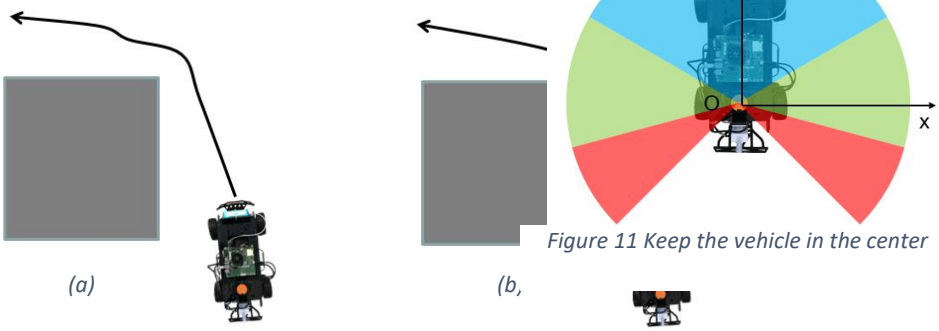


Figure 11 Keep the vehicle in the center

Figure 10 Smoothen the path

then move away from the wall due to the collision avoidance component, generating a jitter path shown in Figure 9 (a). The ideal path would be the one shown in Figure 9 (b). To achieve this, we need to make the vehicle move at the center of the track before it engage corners so that the vehicle can have more space in the corner.

Figure 10 shows how this component works. The 4 sectors between -105° and -60° are used as right section, and the 4 sectors between +105° and +60° are used as left section. The average distances along X axis and Y axis are calculated based on the average range of each sector. The distances on each side are then merged into a straight line separately by using line fit. With the two fit lines, the distance D between the vehicle center O and the track center can be estimated. The centering error can then be calculated with the following equation:

$$\alpha_{cen} = \text{atan}(\frac{3D}{spd})$$

4.4. Combined steering controller

The three steering error components are merged together and fed into the PD controller. The combined error can be described with the following equation:

$$\alpha_{total} = \alpha_{pf} - 0.8 \times \alpha_{avoid} + 0.5 \times \alpha_{cen}$$

The gains are tuned in such a way that that α_{pf} is dominating over α_{cen}, and α_{avoid} is dominating over α_{pf}. To make the controller more stable in straight line, extra condition is added to engage obstacle avoidance:

$$\alpha_{avoid} = \begin{cases} \sqrt[4]{S_{left} - S_{right}} \dfrac{|S_{left} - S_{right}|}{S_{left} - S_{right}}, \ if\ |x|_{min} < 0.25\ and\ r_{|x|min} < 1.2 \\ 0, \ else \end{cases}$$

To prevent the centering force prevent the vehicle from dodging obstacles, extra condition is added to engage centering:

$$\alpha_{cen} = \begin{cases} \text{atan}\left(\dfrac{3D}{speed}\right), \ if\ r_{center-min} > 1.5 \\ 0, \ else \end{cases}$$

where $r_{center-min}$ is the minimum range reading from the center sector.

Hector SLAM Mapping

Created by TU Darmstadt in the RoboCup Rescue League competition, Hector SLAM is unique in that it does not necessarily require odometry information to perform SLAM. It can build a map with scan matching alone. First, we installed hector-slam on ROS. Then we edited the tutorial.launch file under hector_slam_launch/launch, so the sim_time is set as "false". This is required when using a real LIDAR. This value must be changed to "true" when using Gazebo or a bag file for generating maps. Then we edited the mapping_default.launch file and set both the odom transform and base transform to base_link. Then we run the urg_node and the

tutorial.launch

```
<?xml version="1.0"?>

<launch>

 <arg name="geotiff_map_file_path" default="$(find hector_geotiff)/maps"/>

 <param name="/use_sim_time" value="true"/>

 <node name="static_tf0" pkg="tf" type="static_transform_publisher" args="1 0 0 0 0 0  /world /map 100"/>

 <node pkg="tf" type="static_transform_publisher" name="base_frame_to_laser"
  args="0.0 0.0 0.0 0.0 0.0 0.0 /base_link /laser 10" />

 <node pkg="rviz" type="rviz" name="rviz"
  args="-d $(find hector_slam_launch)/rviz_cfg/mapping_demo.rviz"/>

 <include file="$(find hector_mapping)/launch/mapping_default.launch"/>

 <include file="$(find hector_geotiff)/launch/geotiff_mapper.launch">
  <arg name="trajectory_source_frame_name" value="scanmatcher_frame"/>
  <arg name="map_file_path" value="$(arg geotiff_map_file_path)"/>
 </include>

</launch>
```

hector_slam_launch node to view the map on rviz.

mapping_default.launch

```
<?xml version="1.0"?>

<launch>
 <arg name="tf_map_scanmatch_transform_frame_name" default="scanmatcher_frame"/>
 <arg name="base_frame" default="base_link"/>
 <arg name="odom_frame" default="base_link"/>
 <arg name="pub_map_odom_transform" default="true"/>
 <arg name="scan_subscriber_queue_size" default="5"/>
 <arg name="scan_topic" default="scan"/>
 <arg name="map_size" default="2048"/>
…
</launch>
```

We were initially unable to visualize map over SSH. This was caused due to a time stamp mismatch between systems. The TK1 is by default set to the year 1979 every time during startup. This was fixed by manually setting time in TK1 (over ssh) with

sudo date MMDDhhmmYYYY.ss

where the format of the date is shown above and must the closest approximation to the time on the remote PC. This estimation can then be corrected with

sudo date set="ssh labpc@192.168.1.3 date"

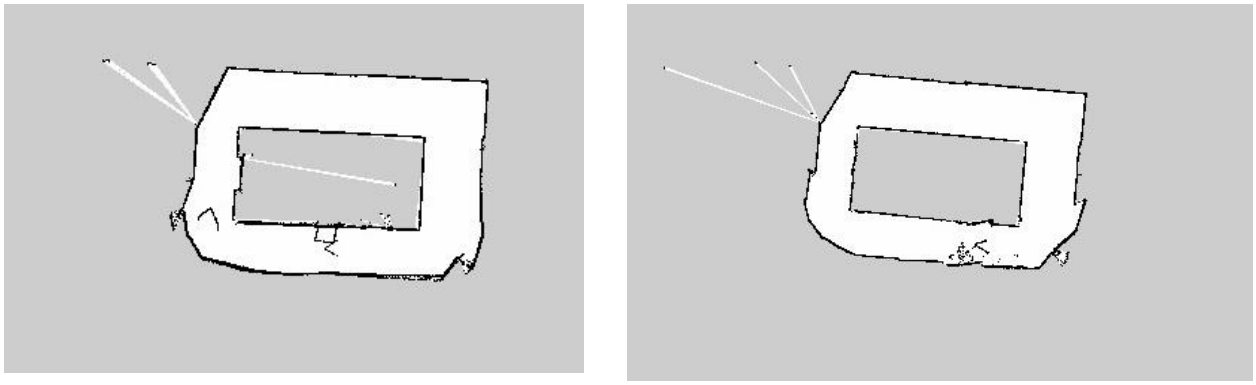


Fig: (a) *Map built when car moved slowly (b) Map built when car was fast*

Results

In the final test, our vehicle successfully avoided all obstacles at both low and high speeds, and managed to finish 3 laps in 26.9 seconds. However, our controller could still be improved in the following aspects:

- When the controller is dodging obstacles it responds a bit late. It is OK for low speed, but when the vehicle is moving at high speed and the available gap is small, the vehicle may still hit the obstacle. This can be solved by fine tune the parameters of the avoiding obstacle component.
- At high speed the vehicle tends to tilt precariously at turns. Lowering the center of gravity will help the vehicle go faster. This can be solved by relocating the battery to below the chassis.
- At high speed the vehicle tend to understeer and may hit the outside wall at sharp corners. That's probably because the vehicle tends to drift to the right after a few crashes. To compensate for that, the steering controller output was biased by 4.5. Since the maximum possible steering command on each side is 30, we lost 16% of maneuverability to the left. This can be solved by adjusting the geometry of the front suspension of the vehicle.
- The speed controller we are using now is too simple. A more complex speed controller can be introduced to make the car be able to slow down when it is approaching obstacles, and come to a full stop at high speed when there's no possible path in the front.

The Hector maps from the final exam are shown below:

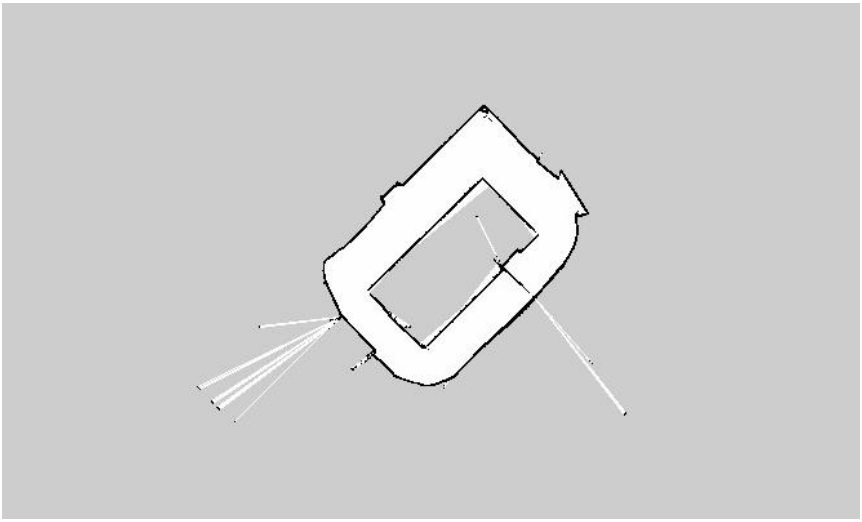


Fig 1: Mapping in slow speed

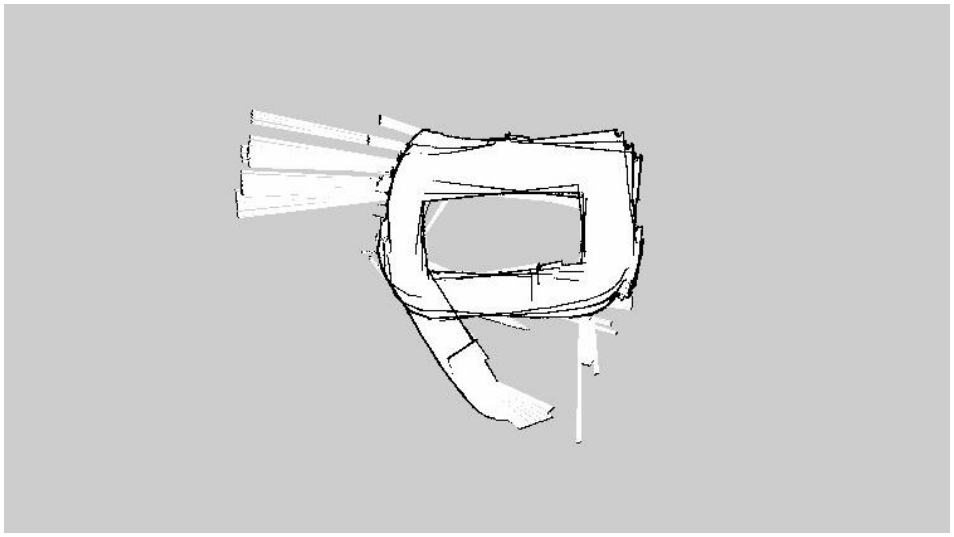


Fig 1: Mapping in fast speed

Discussion and Future Work

Being the first class of students to complete the AuE 893 Autonomy: Science and Systems course, we had to overcome a variety of challenges. As a class, we were warned that we were the guinea pigs for the course, and we all willingly accepted the challenge. Throughout the course there were numerous unknowns and challenges that needed to be overcome, even in order to complete trivial tasks. From the different versions of Ubuntu installed on the team members computers, to the various distros of ROS download and installed, challenges were faced the entire way.

The Turtlebot and F1tenth project were no exceptions to the challenges faced in the class. The Turtlebot had its issue with the bump sensor, where it would work and interface well with the Indigo version of ROS but not the Kinetic version. The interesting part of the course, and perhaps the portion where the most was learned, was the hard coding work-arounds used to implement and problem solve issues that occurred while trying to interface the various sensors and systems used on the mobile robots. Through these hard coding experiences, the students learned very quickly the Linux based system of Ubuntu and how an open source operating system really works.

The F1tenth project posed a variety of issues and frustrating troubleshooting opportunities. The F1tenth project was supposed have an easy to build platform, and simple interfacing all of the sensors since tutorials had been written by the University of Pennsylvania students. However, many of the sensors and hardware used on the bill of materials provided by the F1tenth team were out of date or no longer produced. Therefore, for our project, different vehicles had to be purchased and a few different sensors were used. The vehicle used had a different frame setup, so the laser cut chassis provided by the F1tenth group did not fit correctly on the frame, and custom 3D printed mounts need to be made. The IMU sensor used in the F1tenth tutorials was no longer produced and was not available for purchase, so a new model had to be used. So, the tutorials for IMU integration had to be modified to interface the newer version of the IMU. Overall, a lot of time was spent interfacing the various sensors since many were newer models and had little direction on how to communicate with the specific sensors through ROS and/or the Jetson TK1. However, the hands on experience interfacing the various sensors and diving deeper into the firmware and code involved with them, is some invaluable experience that the students would not have if we just read and followed directions from a tutorial.

Perhaps, if less time was spent interfacing the sensors and working on the various communication aspects, more time could have been spent on the actual algorithms and controls for the autonomous racing portion of the F1tenth project. Given the time we did have to work on these algorithms and simulate in Gazebo, then test in a real world scenario, the team did a tremendous job controlling the vehicle. Given more time, the control parameters of the vehicle's LIDAR PID controller could be further tuned. This could have increased our

maximum accelerations and angular velocities achieved throughout the test course, which could have increased the time for our fastest three laps. Also, further tuning of the PID controller could have provided better obstacle avoidance at both low and high speeds.

Future work would include incorporating both the structure camera and the IMU into the control algorithms of the vehicle. Using the IMU and implementing its data into SLAM could provide important odometry data and produce higher fidelity mapping and localization of unknown environments. The IMU can be incorporated into SLAM using filtering by and extended Kalman Filter or a particle filter. The structure camera or another camera could be interfaced with the TK1 to perhaps provide a live video feed of the vehicle while it is racing around the track. Overall, the project was an invaluable learning experience for all the students involved and provides future students of the course with a solid foundation to start their own F1tenth projects.

Appendix

#1

Streaming data from the Jetson to your PC

On one terminal window
ssh -X ubuntu@192.168.1.3
export ROS_HOSTNAME=192.168.1.3
roscore

another tab
ssh -X ubuntu@192.168.1.3
export ROS_MASTER_URI=http://192.168.1.3:11311/
export ROS_HOSTNAME=192.168.1.3
roslaunch openni2_launch openni2.launch (or the urg_node given above for lidar)

Open another terminal
For rviz:
export ROS_MASTER_URI=http://192.168.1.3:11311/
export ROS_IP=192.168.1.YY (your pc's ip)
rosrun rviz rviz

#2

```
emptyscan = rosmessage('sensor_msgs/LaserScan');
scansub = rossubscriber('/scan');

while(1)
        scandata = receive(scansub, 10);
        data = scandata.Ranges;
        xy = readCartesian(scandata);
        figure(1);
        plot(-xy(:,2),xy(:,1),'color','blue');
        hold on;
        plot(0,0,'*','color','red');
        axis([-3 3 -3 3]);
        hold off;
End
```

#3

Setting up the network bridge
Open the Jetson and connect it to peripherals.
Run ifconfig. Is eth0 and eth2 connected to the ip addresses 192.168.1.2 and 192.168.1.15?

If not go to network settings and switch from Hokuyo to Ubiquiti or vice versa.
Refer to the document “on_startup” on the desktop. Copy paste commands in the terminal.
The commands are here:

sudo brctl addbr br0
sudo brctl addif br0 eth0
sudo brctl addif br0 eth2
sudo ifconfig br0 192.168.1.3 netmask 255.255.255.0 up

EDIT: you no longer need to do this.

> /etc/rc.local was edited with the following commands added before exit 0.
>
> sleep 90s
> brctl addbr br0
> brctl addif br0 eth0
> brctl addif br0 eth2
> ifconfig br0 192.168.1.3 netmask 255.255.255.0 up
>
> This runs the commands in the terminal on startup. The 90 second delay is to allow the eth0 and eth2 to be recognized before the bridge is made.

Run ifconfig again to see if there is a new connection called br0.

SSHing from your PC

ssh -X ubuntu@192.168.1.3

```
%%%

emptyscan = rosmessage('sensor_msgs/LaserScan');
scansub = rossubscriber('/scan');

while(1)
    scandata = receive(scansub, 10);
    data = scandata.Ranges;
    xy = readCartesian(scandata);
    figure(1);
    plot(-xy(:,2),xy(:,1),'color','blue');
    hold on;
    plot(0,0,'*','color','red');
    axis([-3 3 -3 3]);
    hold off;
end

%%%
```

```
%%%

emptyscan = rosmessage('sensor_msgs/Image');
imsub = rossubscriber('/camera/depth/image_rect');
figure(1)

while(1)

    imdata = receive(imsub, 10);
    imageFormatted = readImage(imdata);
    imshow(imageFormatted)

end

%%%
```

Sources:

http://www.robotics.org/joseph-engelberger/unimate.cfm

http://www.computerhistory.org/atchm/where-to-a-history-of-autonomous-vehicles/

https://www.sae.org/misc/pdfs/automated_driving.pdf

https://media.ford.com/content/fordmedia/fna/us/en/news/2017/02/10/ford-invests-in-argo-ai-new-artificial-intelligence-company.html

http://www.clemson.edu/cecas/departments/automotive-engineering/academic-programs/index.html

Turtlebot.com

F1tenth.org

http://docs.ros.org/api/sensor_msgs/html/msg/Imu.html