ECE 350/450 Intro to Robotics, Lab 2

**ROS Tutorial and Racecar Simulator**

Nicholas LaRosa and Nolan Sornson

Department of Electrical & Computer Engineering

Lehigh University

{nil318, nls221}@lehigh.edu

Date: 3/11/2021

**Abstract**

In this lab, our team continued learning to work with Robot Operating System (ROS) and hardware associated with the F1/10 race car. Using our personal installations of ROS on our host computers, both team members completed the remaining ROS tutorials. These tutorials taught the team how to create publisher and subscriber nodes in python and this knowledge allowed the team to immediately start making custom code to run in the f1tenth simulator. The first script written took lidar data as an input and output the nearest and farthest points within each lidar scan. The second script implemented autonomous emergency braking, an algorithm to prevent the car from colliding with a wall. Finally, the team setup and tested the lidar module on the car, to ensure that it was working properly.

**Introduction**

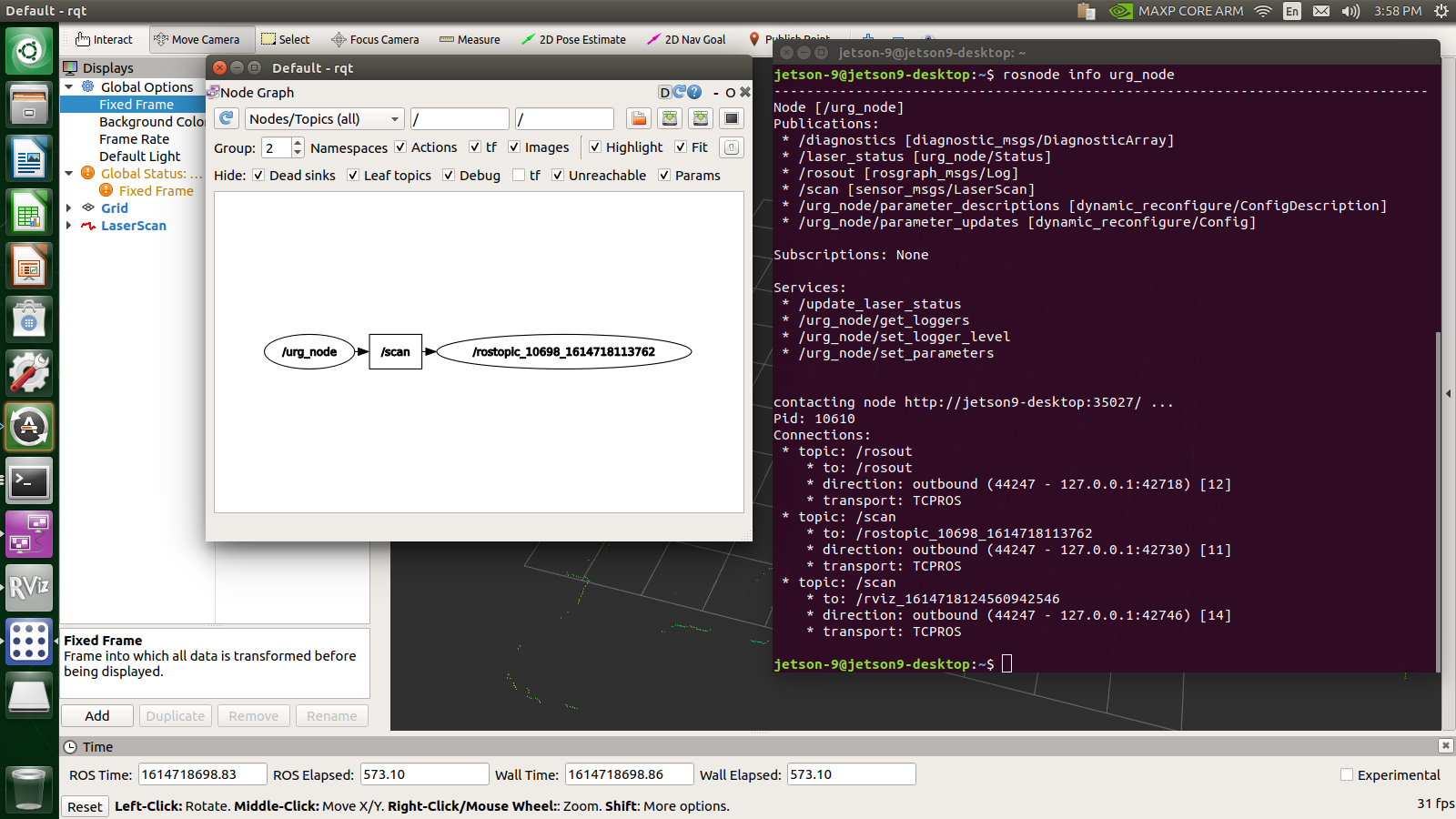
* Part 1:
  + Rqt\_console - this terminal command opens a small GUI window that allows the team to see messages sent to /rosout. This is a very useful debugging tool.
  + Roslaunch - this terminal command is used to launch packages using a .launch file. The .launch files are XML files that list which ROS elements need to be started for the desired environment. The XML code can be modified to launch a custom set of windows and many ROS elements have a parameter YAML file that can be modified to set the initial conditions.
  + Rosed - Instead of navigating to a files location and opening a file using a chosen text editor, rosed allows the team to access files such as python scripts by specifying the package they are in and then the name of the file. This saves a lot of time navigating the linux filesystem in the command line. The default editor is vim, but can be changed using an environment variable.
  + Rosmsg - this command tool displays information about the numerous ros messages. By navigating to the desired message, the team can see what data is sent in the message and the structure required to access that data from their own custom python scripts.
  + roscp - ROS has provided modified basic command line tools that allow the team to specify a package and the file name instead of the full file path. This saves time when files will need to be moved from the host computer to the Jetson.
  + Publisher - a publisher is a node that connects to a specified topic and sends a message. Other code can be included in the script that starts the publisher to process data before it is sent.
  + Subscriber - a subscriber is a node that connects to a specified topic and looks for messages that are sent on that topic. Other code can be included in the script that starts the publisher to process data after it is received.
  + Rosbag - this specialized tool records ros messages that are sent between all nodes and saves them to a .bag file. These bagfiles can be replayed in the simulator to simulate user input and demonstrate car behavior.
* Part 2:
  + LIDAR - a method for measuring distance and range using the time it takes for a laser to be reflected back. Using the data, a 3D representation of the area seen by the sensor can be constructed. The LIDAR module used in this class is the Hokuyo UST-10LX which has a detection range of 10 meters and 1081 measurements per scan over a field of view of 270 degrees.
* Part 3:
  + MUX - a software component that takes multiple sets of input data (in our case multiple ROS topics representing the keyboard, sensors, and planner scripts) and a selector input. Depending on the state of the selector input, it then sets the single output data stream to one of the input data streams.
  + Behavior Controller - This accepts various inputs from the keyboard or joystick controller and sets the state of the MUX. In this way, the behavior controller determines the inputs and resultant behavior of the simulated car by changing the feedback mode.
  + Driver Modes (B, J, K, R, N)
    - Represents the different selector states of the MUX and the resultant operation mode of the behavior controller.
    - B - enables or disables the automatic emergency brake
    - J - enables/disables joystick control mode
    - K - enables/disables keyboard control mode
    - R - enables/disables random walk mode
    - N - enables/disables navigation control
* Part 4:
  + Autonomous Emergency Braking - an algorithm that uses LIDAR and Odometry data to calculate a time to collision factor. If that TTC is determined to be less than a predetermined threshold, the AEB algorithm forces the vehicle to brake to avoid a collision.
  + Time to Collision (TTC) - a factor calculated using the distance away from an object and the relative velocity. The range from the object is divided by the relative velocity projected onto the distance vector. This factor can then be compared to a threshold that would make stopping the vehicle possible prior to collision.
  + LaserScan message - a message that contains all data sent from the LIDAR module. This includes constants for the minimum and maximum values for range and angles as well as arrays containing all measurements collected during each scan.
  + Odometry message - a message that sends all data collected by the inertial measurement unit contained in the VESC. It contains positioning data in cartesian and quaternion coordinates and velocity data for both linear and angular motion.
  + AckermanDriveStamped Message - a message that can be sent to modify the behavior of the powertrain. The steering angle, the steering angle velocity, the speed, the acceleration, and the jerk can be set to desired values by sending this message.

**Procedures**

1. Part 1. ROS Tutorials
   1. Both team members have completed ROS tutorials 7, 8, 10, 12, 13, 15, and 16-18
2. Part 2. Lidar Installation
   1. The Lidar sensor used on the team’s F1/10 racecar is the Hokuyo UST-10LX. This module connects to the Jetson via Ethernet and the setup involves configuring the local ethernet connection to allow ROS and RVIZ to access the data from the sensor.
   2. First click on “Edit Connection” under the internet connection settings in Ubuntu. In the IPv4 tab, set up a static IP address for the lidar module of 192.168.0.10 with a Netmask of 255.255.255.0 and Gateway of 192.168.0.1. If you ping 192.168.0.10 now, the packets should be returned.
   3. Next the “ros-melodic-urg-node” package must be installed using apt
   4. Start ros with roscore and enter “rosrun urg\_node urg\_node \_ip\_address:=”192.168.0.10” This starts the node for the Lidar module and the messages it is transmitting can be seen by entering “rostopic echo /scan” in another terminal window
   5. Finally, to visualize the lidar data, start RVIZ by entering “rosrun rviz rviz” in a terminal window. Here the LaserScan topic under /scan must be added to the displays on the left side and the Fixed Frame must be changed to “laser” instead of “map”. This displays a collection of points where each point represents the point where the laser reached and was reflected back to the lidar module. The lidar module has a range of 10 meters and therefore is able to map almost the entire room, as long as no objects are standing in the way.
3. Part 3: Implementing a publisher and subscriber
   1. First we installed the F1Tenth simulator from the Github repository [1]. Following the instructions given from the class slides and the project page, we were able to successfully install the package in the team workspace. Using another repository, we ran into the issue that there was no “msg” directory, but the revised repository source solved the issue without any workaround.
   2. Next we ran the simulator using the provided launch file. It was invoked using the command “roslaunch f1tenth\_simulator simulator.launch” This spawned a new instance of the simulator, and it also launched a new RViz instance showing the car, the illustrated LIDAR data, and a maze for testing.
      1. “rostopic list” gives all topics created by the simulator, including the “/scan” and “/odom” topics. “rosmsg show sensor\_msgs/LaserScan” provides a simulated interface to the LIDAR instrumentation, and “rostopic echo /scan” provides an output array of the simulated sensor scanning data. Examples of the outputs of these commands are provided in our code submissions.
   3. Following the rosbag tutorial given [2], we then captured several seconds of the simulation using the command “rosbag record -a”. While this command was running, we performed some actions on the simulated car to drive it around.
      1. Using the command “rosbag info”, we can see that all messages being published while the record command is run are captured into the bagfile. When “rosbag play” is issued, these captured commands are re-published.
   4. Next we wrote our first Python program to interact with the ROS nodes and messages. It subscribes to the “/scan” topic to access the simulated LIDAR data, filters out the erroneous data, determines the closest and farthest points detected by the LIDAR sensor, and publishes the two values to the new topics “/closestpoint” and “/farthestpoint”.
      1. This for the most part went smoothly, but the main issues we encountered were related to declaration of the publishers and subscribers. If they are not declared in the correct location, they can go out of scope when they are attempted to be used by the callback function and will throw an error. The code for this program is submitted with our code submission.
4. Part 4: Implemented Autonomous Emergency Braking
   1. To begin writing code for the AEB system, the team cloned the f1tenth\_labs package from github and copied the lab2 python script. This was moved to the safety package that we created in the same workspace as f1 tenth simulator. The “safety” package needed to be created with a few ROS dependencies in order to create the src folder.
   2. The lab2 template was used to write the AEB algorithm. NumPy was used to avoid for loops and accelerate the processing of the data from the LIDAR. NumPy allows for powerful array operations to be completed and simplifies the code significantly. The team made use of np.clip to clean up the range data, np.full to instantiate arrays, and np.arange to create an array of indices. NumPy’s default arithmetic schema is element wise meaning the team was able to apply addition, multiplication and division operations to each and every element without much hassle.
   3. Beyond the algorithm, many messages had to be imported from their respective locations within ROS. LaserScan, AckermannDriveStamped, Odometry, and Bool all come from different locations in the filesystem and the team learned to use rosmsg info to determine which dependencies need to be imported.

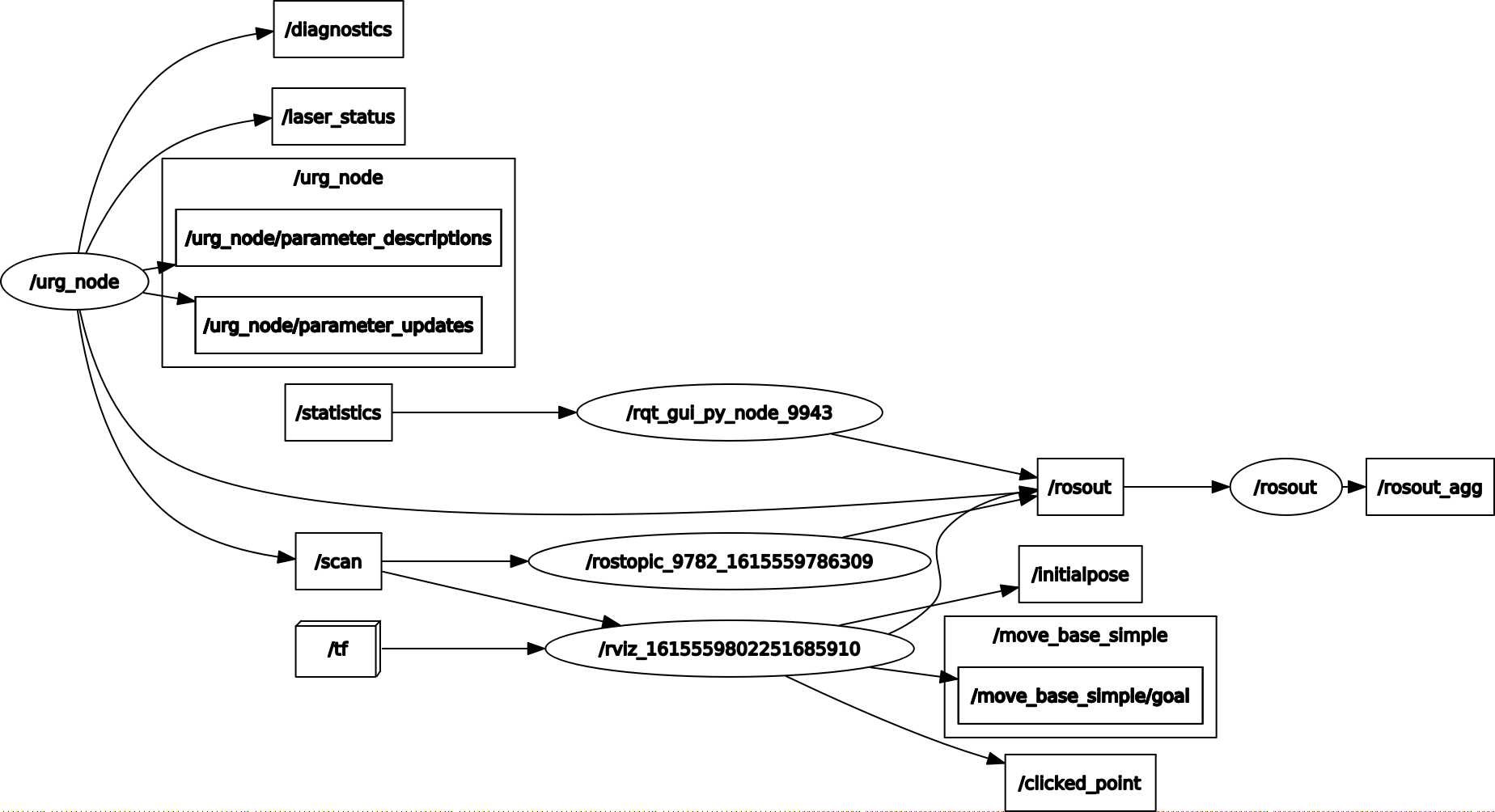
**Results and Analysis**

1. Part 1
   1. Both team members completed all required parts of ROS tutorials 1-18 [3] and are familiar with the operation of the ROS tools for manipulation, recording/playback, and visualization.
2. Part 2
   1. What type of node (publisher of subscriber) is the urg\_node? What channel does it communicate on? What type of message does it have?
      1. The urg\_node is a publisher, and it publishes data primarily to the /scan topic. Additional diagnostic messages are published to other nodes. In the /scan topic, urg\_node publishes LaserScan messages which contain a lot of information. In the std\_msgs header, there is a timestamp and a message id. In the rest of the message there are values for the minimum and maximum lidar scan angle, the angle step between lidar scans, the time between scans, the time it takes to scan at each point, the minimum and maximum distance possible with this lidar module, and finally an array for the 1080 lidar points that were scanned in each pass and an array for the intensities of the lidar scans. These last two arrays are the actual data that is sent with each message and the rest of the values given allow for post processing of the data to ensure it is valid.
   2. What other nodes does the Lidar communicate with? How do they show on the rqt\_graph
      1. The urg\_node primarily communicates LaserScan messages to the /scan topic as seen in Figure 1.



*Figure 1: rqt\_graph with default visibility settings*

* + 1. However, the urg\_node also publishes messages on other topics such as /diagnostics, /laser\_status, and /rosout as seen in Figure 2.

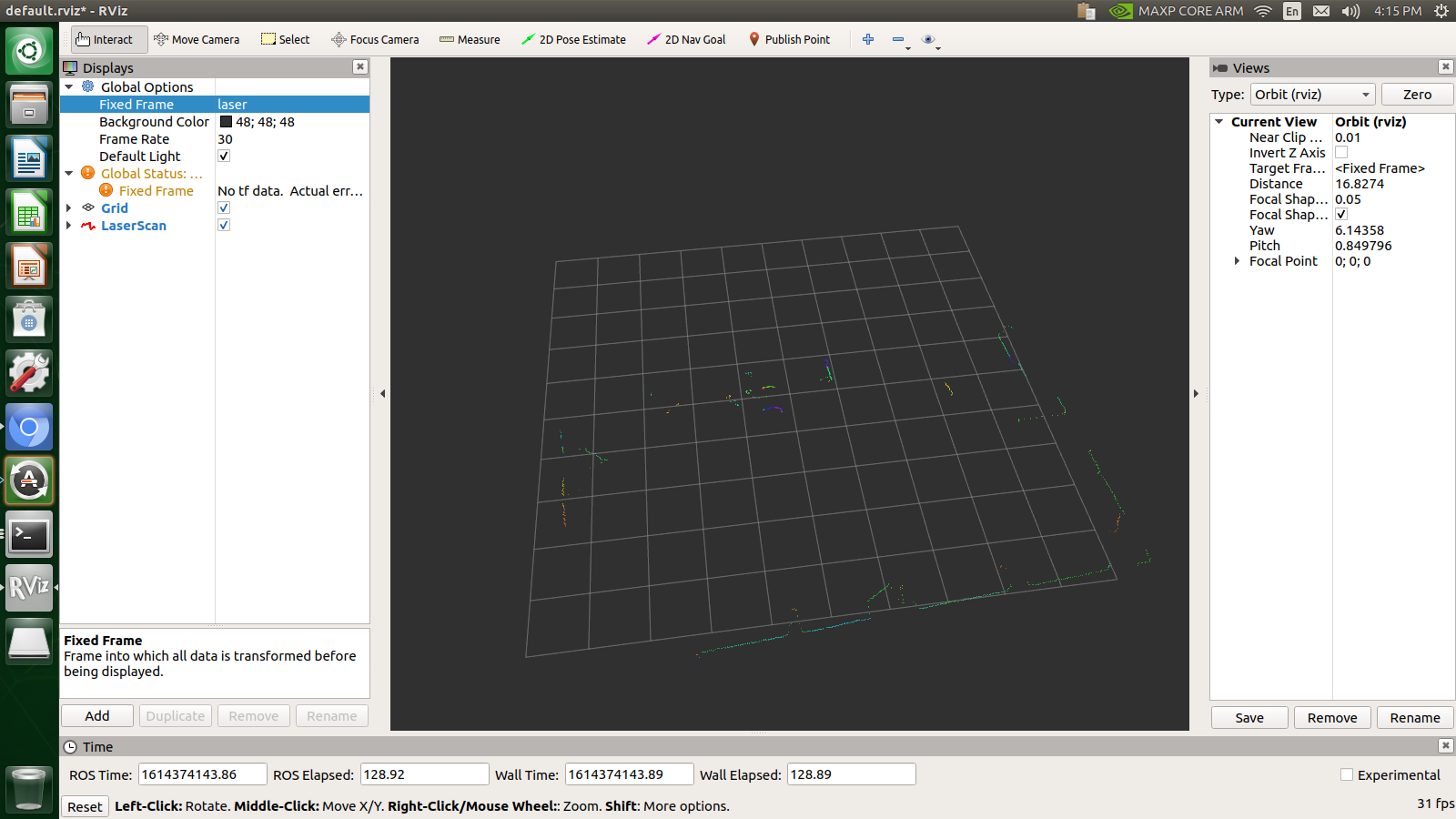


*Figure 2: rqt\_graph with all hidden nodes shown*

* 1. If you have tried to change the IP address of the Lidar, report your experience and the new IP address
     1. This was not performed by the team
  2. How sensitive is the lidar to the obstacles in terms of location, size, and transparency of the obstacle?
     1. The Lidar module has surprisingly fine resolution and is able to accurately map out the outlines of irregularly shaped objects within its range (Figure 3). Objects that are opaque reflect the laser well and prevent anything behind them from showing up on the scans. Lastly, objects beyond the 10 meter range are not able to be picked on the scan. In Figure 4, the opposite side of the room is able to be mapped, but the lower left corner is blank as it was not fully within the lasers range.



*Figure 3: Zoomed in view of Lidar RVIZ with annotations*



*Figure 4: Zoomed out view of Lidar, note bottom left corner was not mapped by Lidar*

1. Part 3
   1. What is the ROS Header msg? What are the data types used in the LaserScan message?
      1. The data types used in the LaserScan message are a Header structure, 32-bit floats for the maximum and minimum ranges and angles, and 32-bit float arrays for the LIDAR range data. The Header structure contains an integer sequence number, a timestamp structure containing the time the message was generated, and a frame\_id string.
   2. What did you observe when you run rosbag play in the turtlesim example and the f1tenth\_simulator example? Where are the bag files saved when you run rosbag? How do you change the location of the bag files when saving them?
      1. In the f1tenth\_simulator example, “rosbag play” caused the car to perform the exact same commands as when the bag was recorded using “rosbag record”. This is because the bag file contains all messages published at the time of the recording, and playing back the bag file causes all messages to be re-published in exactly the same way. If the starting point was the same, the ending point was also the same. Similarly, performing “rosbag play” on the turtlesim produces the same result. When “rosbag record” is run, the bag files are stored in the current working directory, or the directory specified by the “-o” command line argument. To change the location of the stored bag files, we can either change the working directory or explicitly set a bagfile path using “rosbag record -a -o <filepath>”.
   3. (For ECE450 students) If you are to rosbag record only the LaserScan data and extract a specified number of sequences, how do you do this? Each ECE450 student shall submit the answer and the captured data file.
      1. To record only a specific number of sequences using “rosbag record”, we use the “-l <number of sequences>” option. For example, running “rosbag record -l 5 /scan” will capture exactly 5 messages from the /scan topic and store them in a bag file in the current directory. The output of this command with f1tenth\_simulator running has been included in Nick’s submission.
   4. What are the launch files used in the original f1tenth\_simulator? How do the multiple launch files work together?
      1. The launch files in the f1tenth\_simulator are racecar\_model.launch and simulator.launch. The racecar\_model.launch file initiates a car model and publishers for the transformations on the state.
      2. The simulator.launch file initializes subscribers for the various input control surfaces, sensors, and planners, the MUX and behavior controller, and then calls the racecar\_model.launch file to spawn the car.
   5. What does rospy.init node() do in the program? What happens if you do not include the option “anonymous=True” in rospy.init node()?
      1. Rospy.init\_node tells ROS that the Python script is a node, and it allows ROS to set up the new node within the framework and register all publisher and subscriber declarations.
      2. The option “anonymous=True” tells ROS to initialize the new node with a unique identifier appended to the node name in the hierarchy. If we do not include the “anonymous=True” option, we would not be able to run more than one instance of the node at a time since there would arise a naming conflict.
   6. How do you control callbacks for the subscriber?
      1. When a new subscriber is created using rospy.Subscriber, a pointer to a callback function is passed as an argument to the subscriber definition. Whenever a new message is received on the subscribed topic, the callback function is called along with a pointer to the data in the message, and the callback function can perform processing on the data or publish processed data to new topics.
   7. Rate, time or duration are important in robotics. How do you use rospy.rate() in this example to make best use of the Lidar data?
      1. In this example, rospy.rate() is used to set the maximum generation interval of the talker to 10 messages per second. Since the publisher runs in a while loop, it would simply produce messages as fast as the CPU could generate them, which could potentially lock up the system or cause undesirable behavior and power consumption. The rospy.rate() function causes the process to sleep for a set period of time in between message generation to prevent this from happening.
2. Part 4
   1. What are the variables and functions defined in your Safety class?
      1. In the Safety class, the functions included are the constructor, odom\_callback, and scan\_callback. The constructor function is called whenever a new instance of the class is created, and it generates two publishers and two subscribers. The first subscriber listens to LaserScan (i.e. the LIDAR data) and calls scan\_callback as its callback function. The second subscriber listens to Odometry (which provides the velocity data from the car) and registers odom\_callback as its callback function. The first publisher, brake\_pub, takes an AckermannDriveStamped message and publishes it to the “brake” topic to actuate the brakes. Similarly, the second publisher brake\_bool\_pub takes a boolean as its argument and publishes it to the “brake\_bool” topic.
      2. The “odom\_callback” method takes the data stored in “twist.twist.linear.x” (i.e. the current velocity of the vehicle) and stores it in a Float32 class variable “speed”
      3. The purpose of the “scan\_callback” function is to calculate the time to collision (TTC) and determine whether the automatic emergency braking should be activated. First, a threshold is defined, and the Float32 “range\_min” and “range\_max”, along with the Float32 array “ranges”, are stored in local variables. Next the value of the data in “ranges” is clipped to fall within the range defined by “range\_min” and “range\_max” and stored in “valid\_ranges”. Another array “ttc” is then defined that is the same size as the “ranges” array, but instead contains 0 or 1 depending on whether the corresponding element is greater than the threshold. The angle is determined by the indices of the “ranges” element and stored in the “angle” array. The next part of the algorithm calculates the TTC for each of the LIDAR data elements. If the car is moving (i.e. |speed|>0), project the speed onto the LIDAR range on that vector, take the absolute value, and store in a variable “denominator”. Next, divide the “valid\_ranges” array by “denominator” (we have already guaranteed that it is greater than zero), and store in the “ttc” array. Finally, if any element of “ttc” is less than the threshold, publish an AckermannDriveStamped message with a speed of 0 to brake\_pub and True to brake\_bool\_pub to activate the emergency brakes.
   2. What are the differences between the two layers of twist and pose in the Odometry message in topic “/odom”? What are the fields in the lower twist and pose of Odometry?
      1. In “pose”, there are two subtopics called “position” and “orientation”. The “position” topic provides the current position of the vehicle in space, while the “orientation” topic provides the orientation in quaternion format
      2. Similarly, the “twist” topic contains two subtopics “linear” and “angular”. All messages in the “twist” topics are the rates of the position and orientation respectively, so the “linear” topic provides the velocity in all three axes, and the “angular” topic provides the body rates (i.e. the rates of change in orientation)
   3. How do you find the angle of each data entry in “ranges” of LaserScan in topic “/scan”?
      1. In the topics “angle\_min”, “angle\_max”, and “angle\_increment”, we are given the minimum and maximum angle of the sensor readings, and, particularly, the “angle\_increment” provides the angle difference between each element of the “ranges” array. So, the angle of a given “ranges[i]” can be straightforwardly found by evaluating “angle\_min+(i\*angle\_increment)” for any array index i.
   4. What is your experience of running the simulator with the AEB? How does the AEB when you drive backwards? What difficulties did you encounter in Part 4 of Lab 2.
      1. To run to the AEB the ‘b’ key must be hit in the terminal window for the simulator. Additionally the python script needs to be run in a separate terminal window. This could be made one step if we created a custom launch file. When running the AEB script in the simulator, the car drives at a wall, it is able to detect the collision and stop itself based on a constant threshold that we set in the script. Tuning this threshold was accomplished with trial and error, and we found that a value of 0.5 worked well for the purposes of demoing the behavior and even driving around within the walls of the building in the simulator map where the threshold for AEB was much easier to achieve. In the simulator the AEB works when colliding forward as well as backwards because the lidar module in the simulator actually rotates a full 360deg. In real life this will not be the case because the current lidar module we have only operates with a 270deg field of view. We had a couple of difficulties with writing the script to implement AEB during the lab. First, we initially wrote the program using for loops to process the lidar data and to compute time to collision. The program would constantly run into index errors and was difficult to debug as the lidar provides so much data so quickly. We switched over to using NumPy arrays which simplified our code significantly. Second, we initially calculated TTC with a negative r dot as suggested by UPenn, however as the updated Lab 2 slides pointed out, we have a different definition of angles than UPenn. From running the AEB we could tell there was a problem with the angle definition, the AEB would work when you drove at a shallow approach angle to the wall, and worked less effectively the more perpendicular to the wall the approach became. Removing the negative sign fixed all of our problems.
   5. In real-world AEB systems, what are the two types of errors in false detection (wrong classification)? What are the impacts of the two types of errors?
      1. The first type of error is a false positive, which is when a collision is not about to occur, but the AEB system is triggered anyway. This will cause the car to stop even though there is no impending danger. Constant false positives will wear out the brakes and provide an uncomfortable riding experience. While this is seen as the less serious problem, this is still problematic. Most drivers expect cars on the road to behave rationally, and a false positive behavior would appear irrational. This could cause rear end collisions if the other drivers aren’t able to react fast enough if another car’s AEB system trips early.
      2. The other type of error is the false negative, where a collision is about to occur, but the AEB system does not detect it and stop the car. This is a very serious error because it can cause serious injury or damage if the car is about to collide with another person or a car or a building. Products with high false negative rates will lose the public’s trust and hamper self driving car progress in the future.

**Conclusion**

This lab provided the team, firstly, with an introduction on how to use the LIDAR sensor on the F1/10 vehicle. In addition, it showed us how to use Python to create publisher and subscriber nodes and interact with the data provided in ROS, as demonstrated in the use of the f1tenth simulator. This simulator will certainly be helpful in the future as we develop new algorithms to be run on the car, as it will help us to test quickly without having direct access to the car itself. Finally, we implemented our skills in developing a script to provide automatic emergency braking based on data coming from the LIDAR sensor. In addition to providing a useful safety feature for the operation of our car, it also allows us to become much more familiar with the ROS Python API and other tools (including rosbag) that we will need to use for future development and debugging.

**References**

[1]: <https://github.com/f1tenth/f1tenth_simulator>

[2]: <http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data>

[3]: <http://wiki.ros.org/ROS/Tutorials>