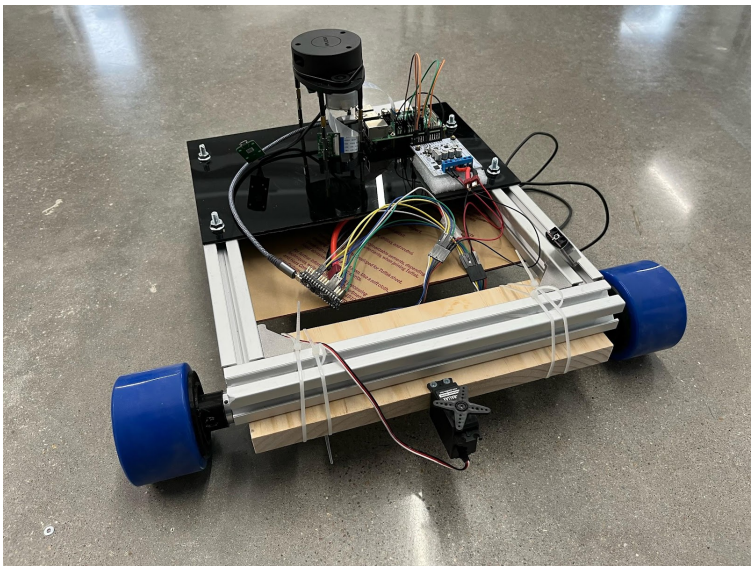
Robotics 2 Project 2: Autonomous Navigation

Brandon Donaldson, Austin Miller, and Will Ward

UCA Department of Physics & Astronomy

Date: May 6, 2022



Abstract:

In Project 1, we built a mobile ground robot that responded to teleop_twist_keyboard /cmd_vel messages and implemented proportional gain control in limited situations. In this project, we implement proportional control on /cmd_vel messages, publish robot odometry using an inertial measurement unit (IMU), and create a map of the robot's surroundings with LiDAR and SLAM (simultaneous localization and mapping).

Introduction:

In the previous project, we used teleop_twist_keyboard messages to remotely navigate our robot around obstacles in the classroom. Our goal in this project, however, was to design a vehicle that navigates on its own through a pre-mapped area using SLAM, Simultaneous Localization And Mapping. The robot was designed to map an area using a LiDAR sensor, a rotating assembly that emits laser pulses to measure the proximity of physical objects from the robot. As the robot moves through an area, the LiDAR continues to take 2D snapshots of the

robot's surroundings. However, since the robot's orientation (pose) changes as it moves, the orientation of these instantaneous maps must be adjusted to the orientation of the robot at the start. Robots use odometry, a measure of how far the robot has traveled/rotated away from its starting position, to transform the robot's current position (base_link position) into the original (odom) frame [1]. This allows the robot to create coherent maps of its surroundings. Our project was successful in mapping the classroom, but our future work includes loading the map and navigating to waypoints inside of it.

Remote Control Drive with Proportional Gain:

In Project 1, our robot listened to /cmd_vel messages from the teleop_twist_keyboard node, but its response was not proportional to the magnitude of the speed. The robot always moved forward, backward, left, or right at the same speed, no matter how fast /cmd_vel was set to. The first step towards autonomous navigation in this project was to modify our subscriber and use proportional gain control on the motor duty cycle to match the speed in meters per second given by the teleop_twist_keyboard command. The odom_pub.py program initializes a velocity subscriber to monitor Twist messages sent to /cmd_vel from teleop_twist_keyboard.

Once a Twist message is received the /odom_publisher topic will record the target velocity to compare it with the current wheel speed of the left and right wheels. While the target speed and current velocity are being compared, an error value between the real and current velocities are recorded within the proportional gain function. The velocity of the robot will incrementally increase or decrease until the target speed it has received from /cmd_vel has been reached and maintains an error of zero.

Transformations and Mapping:

We used the slam_toolbox package and navigation2 to create a map of the robot's surroundings. In order to create a map, we needed three transformations: base_link to sensors on the robot, odom to base_link, and map to odom. The base_link frame is the body of the robot. Each of the sensors used to create the map (IMU and LiDAR) also have individual frames. The odom frame is located at the position where the robot starts, and the map frame is fixed global frame that provides a coordinate axis for the room. Each transformation provides a way to move from one frame (coordinate axis) to another through translations and rotations [2].

We achieve the base_link to sensor transformation by using a URDF (Universal Robot Descriptor File) file and a robot_state_publisher node that is launched when running the project_2 package using the "bringup_thebot.launch.py" launch file. Inside the URDF file, the base_link and all of the sensor links are created with defined physical dimensions. The sensor links are also localized relative to the origin of the base link using physically measured distances from the sensor mounted on the robot to the center of the robot.

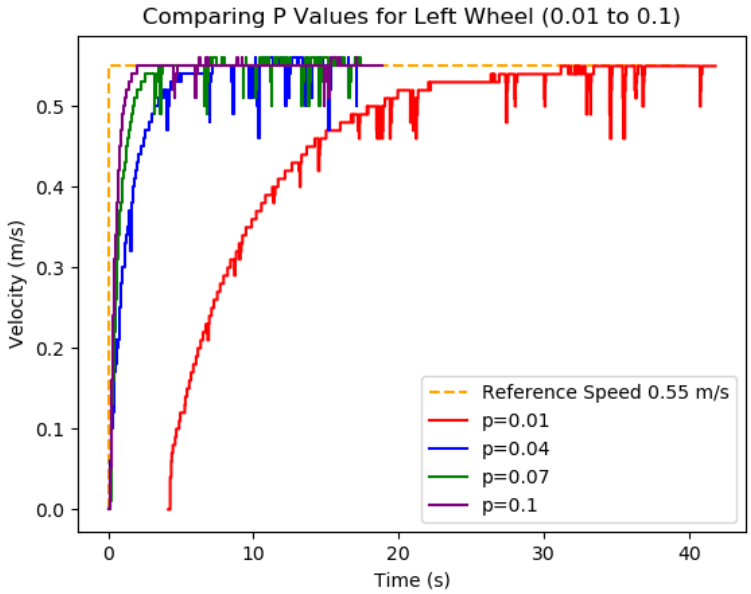


Figure 1: Encoder readings are unstable.

Next, the odom to base_link transformation is provided by the Inertial Measurement Unit (IMU) and velocity messages from the /cmd_vel topic. While some robots can use real velocity measurements from motor encoders, we do not use encoders to calculate odometry because our velocity readings are unstable. For instance, Figure 1 is a graph from our previous project report showing the recorded wheel velocity for different proportional gain constant values. These plots have frequent dips in velocity due to unstable encoder readings. The velocity messages from /cmd_vel, however, are clean and only change when the user inputs a new command with the keyboard. In our program odom_publisher.py, a callback function calculates the transformation from odom to base_link by calculating the x and y translation and the quaternion transformation. The program then broadcasts this transformation and publishes the odometry over the /odom topic. In a separate program, imu_pub.py, the program reads linear acceleration and angular velocity messages from the IMU over the I2C communication channel. The program then creates

a publisher node to publish the messages over an /imu topic every 0.01 seconds. The timer callback function monitors Imu messages to determine the robots real angular and linear velocities. Finally, in the launch file of the project, an ekf_filter node is launched that combines the /imu and /odom messages into a new /filtered_odom message and broadcasts a new transformation. In theory, these two sensors combined results in higher accuracy.

The third transformation between the map frame and odometry frame is provided by slam_toolbox. When this package is launched, slam_toolbox accepts odometry messages and /scan messages in order to stitch all of the scan messages together as the robot travels. If the odometry is not working properly, the individual scans won't stack properly to create an accurate map. Instead, the image will look like several maps rotated and skewed. Figure 2 below shows the map we created using LiDAR and our /odom messages.

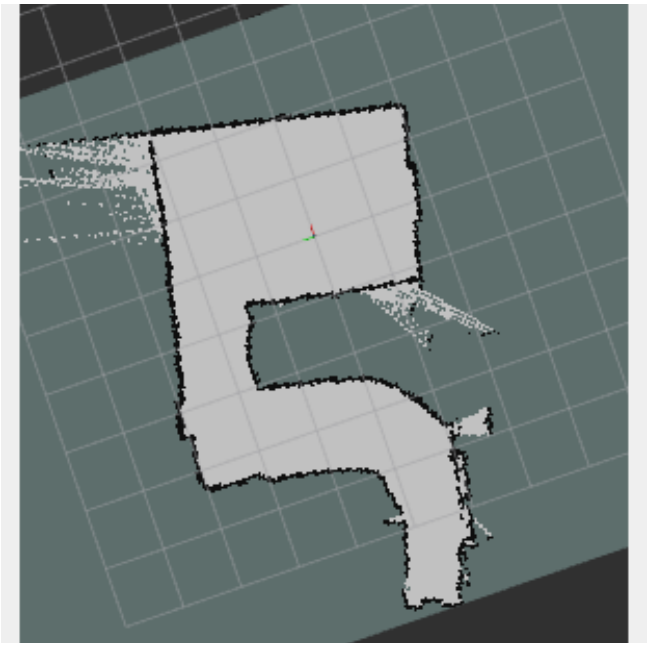


Figure 2: Map created by the robot.

Factors affecting map building:

Creating a digital map using RPLidar is a great way to effectively see the surroundings of the robot. However, there are a few drawbacks to this type of set up. Since lidar uses light in the form of a pulsed laser to measure distance ranges the robot will not be able to identify any glass or clear objects as light passes through them. When creating the map the robot speed must be very slow in order to get an accurate mapping, otherwise the odometry would get off track and

think the robot is located at a different spot than it actually is. The odometry used for localization is based on the /cmd_vel topic (the target speed in our proportional control). For slow speeds, the robot is able to reach the target speed quickly, and the localization is accurate. For fast speeds, however, the robot takes longer to reach the target speed, and the robot has not traveled as far as the odometer claims. Another issue with Lidar can be caused by changing the surroundings that are getting mapped. If the robot creates a map and then there is a change in the surroundings the map will not be correct and this can cause confusion in the path the robot must take. The walls of the path we created in the classroom were made of cardboard and spare boxes that were easily bumped and moved. This could have caused problems in the future if we were able to navigate using a map that is no longer precise.

Vision:

As we developed the mapping process, we also began working with a camera mounted at the front of our robot. Our ultimate goal was to use computer vision to find an object in the room while the robot is autonomously navigating, navigate close enough to the object to reach it, and then remotely control a robotic arm to pick up the object and store it on board the robot. While we did not get to this point, we were able to set up computer vision for object recognition.

The first task to tackle was how to interface ROS 2 with OpenCV, a popular computer vision library. To begin we needed a module to run object detection with an open source TensorFlow Lite model. To achieve this we first created a publisher node to publish webcam data to a topic, as well as an image subscriber node that subscribes to that topic. The publisher will publish an image to the video_frames topic. The queue size is 10 messages and will publish every 0.1 seconds. To convert our image from OpenCV to ROS 2 we used self.br = CvBridge(), and self.publisher_.publish(self.br.cv2_to_imgmsg(frame)). Once the publisher was created the subscriber will receive an image from the video_frames topic. After verifying the publisher and subscriber were communicating properly we needed to combine the publisher and subscriber in one code that we can include in our Launch file. Once these were combined we needed to add the Object Detector TensorFlow, the example tensor flow lite was downloaded and tested. We then added the module ‘efficientdet_lite0.tflite’ and the object detection class in tandem with object detector options class to our combined publisher and subscriber code.

Once this was done we ran the code and the image viewer popped up with a black image and an error saying our message was empty. Testing if the ROS 2 topics were being published correctly we could see the message was in fact correctly publishing. Our URDF file did not include our camera, so we added it to the base_link in the correct location. This still didn't fix our error, we then realized that we were not sending the time stamp with it and this might have caused the problem. We then added the utils code from TensorFlow lite because importing it from OpenCV was not working. After fixing this our error message was: "Message Filter dropping message: frame 'camera_link' at time 1651168479.031 for reason 'discarding message because the queue is full'". After doing some google searching we found "ros2 run tf2_ros static_transform_publisher 0 0 0 0 0 0 map camera_link" is the command to correctly publish our transformation for us to see our image displayed in rviz2. The camera_pub_node was then added to our launch file and we used colcon build to correctly assemble the work space. This allows our robot to see the field in front of it and detect objects while predicting what they are and giving a percentage of correctness. We will use this to find our object and grab it with the robotic arm.

Conclusion:

In this project, we successfully used slam_toolbox and navigation2 to map the robot's surroundings. We used LiDAR and the RPlidar package to scan the room, and we used velocity readings from /cmd_vel and acceleration/angular rotation from an IMU to provide transformations. These two combined helped the robot create a large 2D map by stitching together instantaneous maps as it navigated using remote control. Unfortunately, we did not complete our original project goal of navigating autonomously through the saved map using waypoints. The idea is to use ACML (Adaptive Monte Carlo Localization) to determine where the robot is located within the map, but this localization technique is currently not working. In the future, we will use Navigation2 documentation and Nav2's turtlebot simulation examples to determine how to implement the ACML algorithm and how to successfully navigate using waypoints.

References:

[1] Setting Up Odometry. Navigation 2 Documentation website: https://navigation.ros.org/setup_guides/odom/setup_odom.html#robot-localization-demo

[2] Setting Up Transformations. Navigation 2 Documentation website: https://navigation.ros.org/setup_guides/transformation/setup_transforms.html#transforms-introduction