Software

Robot Operating System (ROS)

The need for a simplified approach for localization sparked the switch to using the Robot Operating System (ROS.) ROS greatly simplifies complex localization, mapping, and pathfinding algorithms; all of which are incredibly essential to this competition, although, heed was taken at first because of the vast learning curve coupled with ROS. A very layman's explanation of ROS is that sensor data is input nodes, which house all the vital information for the sensors such as programs, readme files, or launch files. Nodes can talk to each other and request information, process the information, and send to another node. A graphical explanation of how our sensor nodes communicate is shown in Figure 1 below. We have four sensors; LIDAR, Camera, IMU, and encoders that create four ROS nodes. The LIDAR, IMU, and Encoder nodes all input into the localization node, this node coupled with the camera node create what we call the decision maker or essentially a path planner. The decision maker then outputs motor control commands to move the robot.

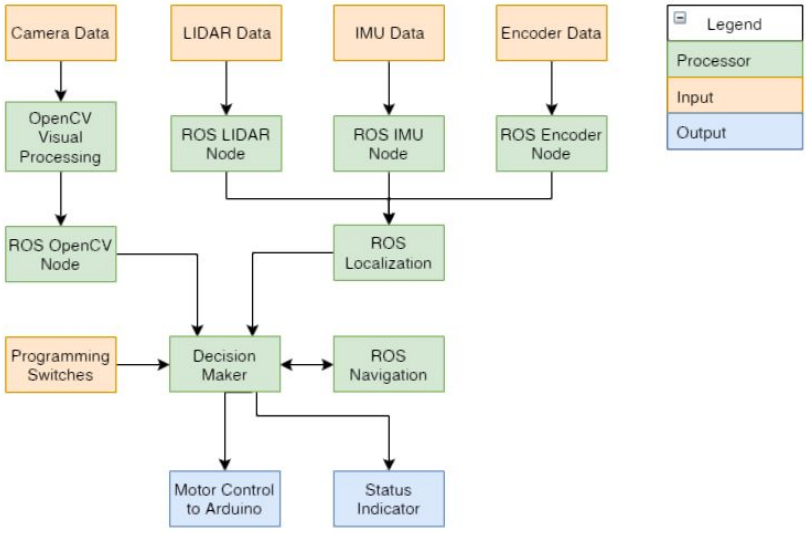

Figure 1: ROS Node Diagram

Figure 2 shows the resulting ROS graph of our implementation. After testing, we determined the IMU was not adding any additional helpful information for localization so it was omitted from the final design.

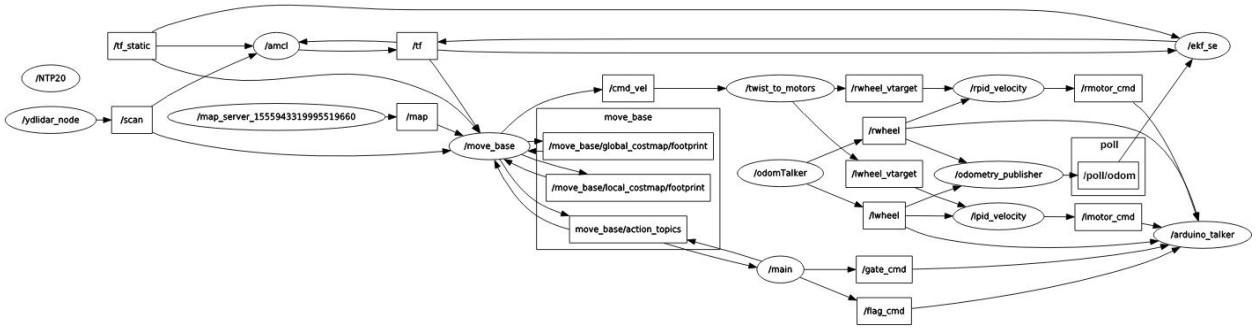

Figure 2: RQT Graph

LIDAR and Localization

Figure 3: YDLIDAR

The LIDAR sensor is one of the main components of localization; this sensor acts sort of like a GPS for the robot. The YDLIDAR (Figure 3) has a 360-degree scanning range with a 10-meter range. As stated previously and shown in Figure 7 the LIDAR will be a node, and this is one of two inputs into the localization node. ROS greatly simplifies the localization process as it already has the algorithms needed, one being AMCL. Adaptive Monte Carlo Localization is the process of tracking the pose of a robot against a known map, which is our case. AMCL takes in a map, LIDAR scan, and transform messages, and outputs pose estimates. As stated one of the first things we need to implement the AMCL algorithm is a known map, which is the competition field. Our original process of creating a map was creating a jpeg image and feeding it thru a python program launched within ROS to return a map. However, errors were given when adding the LIDAR scan data to the map. To solve this issue, we had to pre-map the field with actual data using simultaneous location and mapping (SLAM). This proved to be very effective, especially when there was a large skew in the field shape. See Figure 4 for a view of the robot's orientation and position plotted on a map in ROS.

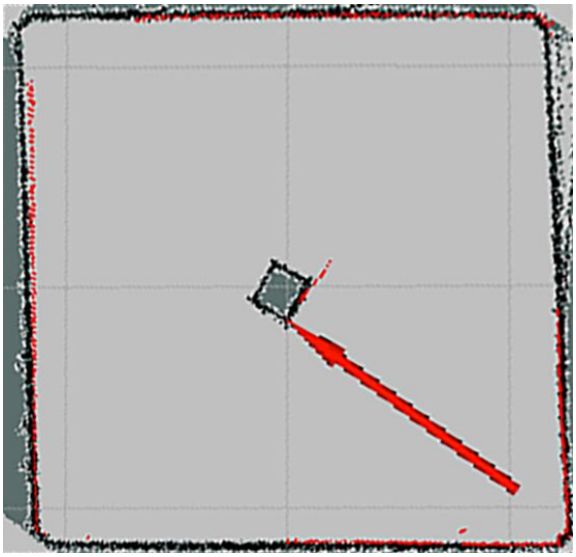

Figure 4: RVIZ map with localized position and orientation

Rotary Encoder and Odometry

Figure 5: Quadrature Encoders

In order to have more precise measurements in movement, we will use two capacitive encoders attached to the motors. The encoders measure and count the turns of the motors, allowing us to know how far the robot will travel with one motor turn. Using this alongside the localization and object detection, we would be able to move the robot the exact distance towards any detected object to ensure the robot picks up the object. This will also prevent the robot from pushing any object into a position that would make it difficult to pick up.

The encoders are wired and read using a Teensy microcontroller. The Teensy features built-in interrupts that allow regular updates from the encoders. This is required as the encoders measure rotations in ticks, these ticks update at a very fast frequency, requiring processing just as quickly. The Teensy reads the encoder ticks and calculates the angular acceleration, which is then sent to the Raspberry Pi via serial.

Visual Detection

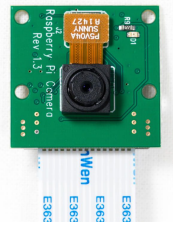

Figure 6: Raspberry Pi Camera Module

We are using the Raspberry Pi Camera Module (Figure 6) to detect the debris objects. By using the open-source computer vision library, the Pi can extrapolate the debris's pixel location, pixel size, the angle from center, color, type, and approximate distance from the robot. It uses this data to make decisions on where the robot should go. For E-Day, we demonstrated that the robot can turn towards an object, drive until the object is right in front of it and then turn and find a different object. The program worked really well. The biggest issue is that the camera was pointed too far up and needed to be mounted at a lower angle. The next step was integrating visual detection with ROS. A custom message group was created to pass the object information to the main node for navigation.

Navigation

Once the robot is capable of finding itself on a map with localization, it becomes necessary to plan the steps from point A to point B. There are several ROS nodes available for this that we attempted to use listed below.

- DWA (Dynamic Window Approach)
    - Smoother paths
    - Requires more computation power
- FTC (Follow the Carrot)
    - More simplistic approach, less computation
    - Does not avoid obstacles well
- Move_Basic
    - Simplest planner
    - Turn to goal, move until goal is reached
    - Not always accurate
    - Does not avoid obstacles

Ultimately, our design settled on Move_Basic, as it provided the simplest approach with parameters that were in our scope of design. Aside from Move_Basic, most path planning algorithms use a form of costmaps. These costmaps determine what path would "cost" the least

in terms of time and distance. Most algorithms also use both a local costmap and a global costmap.

The local costmap will calculate where the robot should go in its immediate vicinity, this is usually calculated in real time as the range of the map is shorter and in relation the position of the robot. The global costmap will calculate a path using algorithms such as A* and output something like what is seen in Figure 7. Using both of these costmaps, the robot can efficiently navigate its space avoiding obstacles it can see.

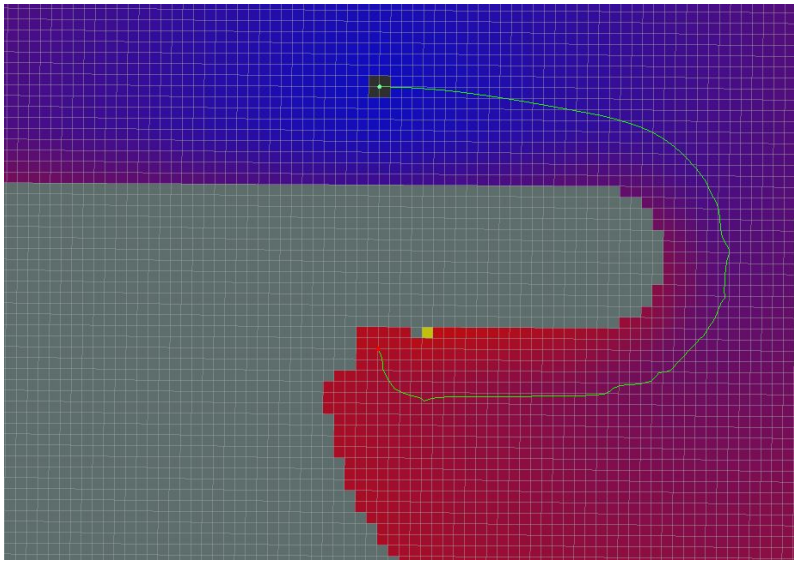

Figure 7: Example visualization of navigational pathfinding [2]