BezosBots

George Boyer[1], Doncey Albin[1], Ben Kraske[1], and Matt Nguyen[1]

***Abstract*— The goal of this project was to set up and program an AWS DeepRacer to autonomously navigate the hallways in the basement of the Engineering Center at CU Boulder. This was to be accomplished via two methods: 1. A simple wall-following algorithm using lidar and Proportional Integral Derivative (PID) control. 2. A full implementation of Simultaneous Localization and Mapping (SLAM) and path planning. The DeepRacer was able to successfully navigate the basement course using the first method and significant progress was made in implementing the SLAM method. Here we provide a summary of our progress on the work and a reference for implementation details using the Amazon DeepRacer platform.**

I. Introduction

Autonomous navigation has long been a challenging frontier in the field of robotics. In order to better understand the practical challenges of implementing autonomous navigation and put into practice the concepts learned in CSCI 5302, team BezosBots elected to implement a hardware-based autonomous navigation system. This system was tested in an indoor corridor navigation setting. This implementation gave insight not only into the process of integrating multiple software components, but also into the practical challenges faced when doing so on hardware, which is prone to noise and errors.

The outline of this paper is as follows: Sections II-VI provide the details of our implementation and design process. Section VII is a presentation and discussion of our results. Section VIII provides a reference for use of the various Robot Operating System (ROS) packages and DeepRacer interfaces used.

II. Problem Definition and Background

The initial goal of this project was to develop an autonomous navigation system which was capable of using Simultaneous Localization and Mapping (SLAM) to generate a map which would be planned over with a path planner. This system would be capable of avoiding static objects and would be fairly robust to any errors in control.

The minimum viable product for this project was to have the system autonomously navigate the hallways in a much less robust manner, simply using some form of odometry or relative localization (wall-following and wall collision avoidance).

The system was tested on a track composed of a series of hallways in the University of Colorado Boulder's Engineering Center. The track is a rectangle of approximately 24 meters by 32 meters, with numerous features, such as doorways and alcoves.

The DeepRacer platform used for this project is an all-wheel drive car platform with onboard compute and sensors [1]. The compute module is an Intel Atom with 4GB of RAM and 32GB storage (there is an SD card slot for additional storage). The vehicle ships with an IMU and camera. The sensor package for the vehicle adds an additional camera (for stereo imaging) and a RPLidar A1. The default operating system is Ubuntu (version varies according to the age of the bot), loaded with Intel OpenVINO toolkit and ROS.

III. Setup

A. Software Setup

DeepRacer initially comes with Ubuntu 20.04 and ros2. This can be rolled back to Ubuntu 16.04 through directions provided by AWS. This is the simplest way to get ros1 working without using a bridge.

To build packages and use msgs to control the DeepRacer. **add source /opt/aws/deepracer/setup.bash** to the bashrc after the source /opt/ros/kinetic/setup.bash. Source the bashrc to be able to get the types. View our how to pages to see how this is done.

The software for DeepRacer is partially open source. Most of the open source elements are for ROS2. Most of the ROS services are started on startup. These start as services and can be controlled by systemctl.

B. Hardware Setup

The hardware included with the DeepRacer platform included quite a few parts that can be found on remote RC (radio-controlled) car platforms, such as a ESC (Electronic Speed Controller) to drive the power to a servomotor for steering and a DC motor for driving, a chassis, and wheels. However, there is a sensor available through Amazon that was also utilized for this project, consisting of an extra camera module and a 2-D lidar scanner. Also, during testing of this device, our team also added a bluetooth dongle for wireless input support and a Phidgets 9-axis IMU to replace the on-board IMU, though it was not utilized for successful completion of our goal.

1) Lidar Sensor
2) Phidgets IMU (external)
3) Bosch BMI160 IMU (internal)
4) Bluetooth dongle
5) Xbox Controller
6) Driving: 7-8V DC Motor
7) MG996 17 gram servo motor

IV. Localization

A. IMU

The DeepRacer has an imu connected to the I^2C bus. The address and which bus it is connected to varies on device. Checking where it is connected to can be found by using the tool *i2cdetect*. There are no public ROS drivers for this and DeepRacer built in packages do not have references to this.

[1]University of Colorado Boulder

This IMU is not calibrated and is fairly noisy. The team ran out of time trying to get this to work and moved onto a phidgets IMU which has ROS drivers and produces better raw results.

B. *Laser Scan Matching*

With lidar being available and fairly computationally light compared to processing camera data, we decided to use scan matching [2] in order to improve our localization. ROS scan tools provides laser scan matcher, a convenient ROS package for the Canonical Scan Matcher (CSM) using Iterative Closest Point Matching [2]. This package uses a series of lidar scans to determine the odometry and pose of a vehicle, published as 2D pose. It optionally takes additional input in the form of IMU data (in the fixed frame) for updating the pose of the robot more accurately and odometry or velocity data (in the fixed frame) for more accurate information on vehicle movement between scans.

The laser scan matcher package provided a demo with a pre recorded bag and launch file for the demo. Since we were working with recorded ROS bag files in order to test localization after completing a manual lap around the course, we were able to leverage this demo launch file. Replacing the demo bag with our bag files and running the launch file allowed for convenient testing of the laser scan matcher. Additional parameters, such as static frames from the lidar to the base link (vehicle) frame and world to map were added to this launch file. Here, additional parameters, such as what sensors to incorporate, were also included in this file.

As discussed below (in GMapping SLAM), the demo launch file also allowed for GMapping SLAM to be launched at the same time as laser scan matching. As such, most testing of the laser scan matching was run simultaneously with SLAM. Initial results with only the lidar data were decent, with the general shape of the path matching that of the course, but not being dimensionally consistent with the course and having some curved segments. Adding IMU data provided more accurate information on when turns occurred, generally providing a roughly rectangular route, which was still dimensionally inaccurate. This is likely due to the lack of distinct features in some portions of the hallway. Likewise, there were many specular surfaces (reflective at undesirable angles) in the form of door kick plates and windows. These factors in combination with the lidar being 1-D led to poor matches in some areas.

We attempted (at various points in the project) to add either odometry or velocity data to the laser scan matcher (see Dead Reckoning section). This data should have helped correct the inaccurate dimensions of the route. However, adding odometry and velocity generally seemed to do more harm than good. Paths were longer, but generally did not match the true shape of the path. This is likely due to inaccuracy in the estimated odometry and velocity. The laser scan matching package does not appear to account for the variance in these inputs, meaning noisy estimates may be taken as ground truth. Maps/routes generated with both constant and variable velocity movement were both inaccurate. Although some minor tuning, such as the number of ICP cycles, was performed on the scan matcher, lack of tuning on the scan matching and frame issues may have contributed to this behavior.

C. *Hector SLAM*

The first attempt at SLAM was to use an algorithm that allowed for solely using the lidar scanner. If this worked well, we did not have to use the on-board IMU, as it was troublesome to work with. This is where Hector SLAM came into play. It uses the lidar sensor and its placement on the vehicle to build 2D pose estimates and odomotry transformations. Unfortunately, the Hector SLAM algorithm was not able to accurately build a reliable map for us during testing. This was shown to be most problematic when the vehicle was turning corners. Our team attributed these errors to likely being caused by the tilting of the lidar during turning, as the DeepRacer has a suspension mechanism. If this is the case, a filtering algorithm may be used to process (or transform) this noise before publishing it to the "/scan" topic, though this is not for sure. Ultimately, our team decided that the best approach would be to use IMU data in conjunction with lidar for best results.

D. *GMapping SLAM*

The GMapping ROS package provides Fast Volumetric SLAM in the ROS environment based on work from Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard [3]. This package subscribes to frames for the lidar, odometry, and base link, in addition to the lidar data itself, and publishes a map. The laser scan matching demo script provided a convenient means to load a ROS bag and the scan matching results. The scan matching package also came with a demo script incorporating both scan matching and GMapping. This script was used as the basis for our testing of GMapping with laser scan matching on our data, as described in Section VIII.

A large portion of the tuning for GMapping with Laser Scan Matching was focused on returning correct localization data with laser scan matcher. As such, many of the parameters for GMapping were left as they were in the demo file. Parameters related to the lidar range were adjusted in GMapping, but these had little impact on its performance. The quality of the maps returned largely depended on the odometry/frame data provided by laser scan matcher, with the issues discussed in the previous subsection, b, manifesting in the maps generated. Maps generated without odometry input to laser scan matcher were fairly representative of the space, but dimensionally inaccurate, likely due to reflections and lack of features in some portions of the hall. When odometry was added to laser scan matcher, path length increased to some extent, but there were other issues, likely with frames or the odometry estimate fed to laser scan matcher, which caused these maps to diverge from the true course, often in catastrophic ways which made the map unrecognizable. More tuning and adjustments to the frame parameters are needed. Furthermore, use of a multidimensional lidar or placing the

lidar at a different height (above reflective door kickplates) may have helped obtain better mapping results.

E. Dead Reckoning

In an effort to better localize, control inputs were incorporated into a simple dynamics model to predict the odometry and later the velocity of the vehicle. To accomplish this, scripts were written which listened to the manual_control topic being published either by remote control or by a controller. A steering angle in the range of 1 (full left) to -1 (full right) and throttle percent [0,1] are in each manual control message. Using these values, along with (tuned) estimated conversion rates to steering angle in radians and velocity in meters/second, respectively, estimated kinematics were determined using a simple car model[8]. Kinematics (x- and y-velocity, and vehicle angle) were published to the odometry topic. During some tests, angle was replaced with IMU orientation data to see if this would lead to better results. See the dead_reckoning.py script in the localization folder of the attached zip for more details. Note that y-velocity may be set to zero as the frame the data needed to be in for laser scan matcher was initially unclear. It is also worth noting that laser scan matcher discards away the angle provided by either the odometry or velocity inputs in favor of the IMU angle. As noted above, the addition of odometry data did not help reduce error in the odometry/pose resulting from the scan matching.

Additionally, a velocity-only dead reckoning script (vel_reck.py) was used to provide velocity to the laser scan matcher node in hopes that this would alleviate any error induced by managing the timing of kinematics in a separate node. In this case, only the x- and y- velocities were provided, using instantaneous orientation from the IMU to determine the direction of each. This also did not improve scan matching accuracy. It may be the case that frame definitions were incorrect and that better definitions would have led to more effective incorporation of odometry or velocity data, although the laser scan matcher does not appear to directly use frame information from these sources.

E. Visual Odometry

Visual odometry was attempted due to a lack of odometry from the car. The DeepRacer has 2 built in cameras. The cameras however are not calibrated and there are no public or well documented examples of using them not on the DeepRacer so calibration happened on device with the checkerboard.

Visual odometry was done with viso2_ros. This package is supported in kinetic but noetic and above there is no support so testing was difficult. The speed at this would run is in the singles of fps with frames being dropped. Hardware acceleration was attempted

DeepRacer runs on an intel atom cpu that supports OpenVINO. OpenVINO is Intel's machine learning and vision deployment framework for its hardware (cpus, fpgas and soon gpus). Openvino is powerful but a pain. Using AWS tools building a RL model and deploying it to openvino to stay in the middle of the hallway with just cameras should work.

Viso first finds the homography H matrix from point correspondences using RANSAC with 8 points. It then computes E from H with the camera calibration. It then transforms the 2D corresponded points into 3D points. Those 3D points are then used to compute the ground plane and camera height and pitch are used to scale the ground plane.

We propose using a Raspberry Pi 4 with its more powerful cores and better community should allow for students not being stuck in compiler complications and instead focusing on algorithms.

V. Path Planning

For our path planning, we had our global planner take in our position obtained in localization in order to check proximity to our target points. Our local planner would provide our controller with a path to our next target to follow.

A. Global Planner

For our global planner, we were planning on having a set start point defined as (0,0) and having our target points defined in terms of displacement from the set starting point. For our targets, we generally selected points 5 meters away from each other with some exceptions at or around turns.

The main function of our global planner was checking our vehicles proximity to the current target point and switching to the next target when within a proximity of 0.2 meters. The proximity threshold was decreased to 0.1 meters when the vehicle was trying to reach the final goal point. When the vehicle has reached the current target pose, it will publish the next target pose to the local planner to plan a path to it.

B. A Path Planner*

Our first implementation of our local planner used an A* path planning algorithm that had the euclidean distance between points as the heuristic. In order to avoid collision with obstacles and have a gap between the vehicle and the wall, we attempted to pad the obstacles in the occupancy grid.

One of the issues that we ran into with the A* path planner was runtime. Initially, we used a list for the open and closed sets with a separate f-score dictionary which led to higher runtime of around 5 seconds due to looking through the open set for the point with the lowest f-score during each iteration. We were able to reduce this runtime to approximately 0.5 seconds by having the open and closed sets as dictionaries with the points as keys and the f-scores as values. Despite being able to reduce the runtime of the A* path planning itself, we found that padding the obstacles caused us to have a runtime that was still too high.

C.. Hybrid A Path Planning*

In order to circumvent the issues of the A* algorithm, we looked into switching to a hybrid A* path planner [5] that uses Dubin's path as a heuristic. One of the differences

between the regular and hybrid A* path planners is that, while the regular A* algorithm just returns the shortest path, hybrid A* will simulate the movements of an actual vehicle, creating a path that is more viable for the vehicle to follow.

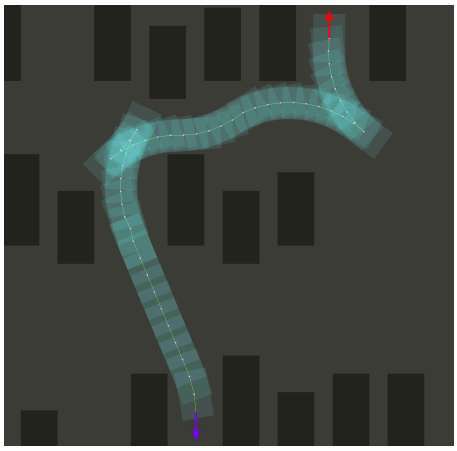

Figure 1: Example of Map Produced Using Hybrid A*

VI. Control and Basic Autonomous Navigation

A. Wall-Following

In order to develop a minimum viable solution, capable of navigating the course without outside input. We implemented a simple wall-following algorithm. This algorithm used the lidar to determine the distance from the left wall and use this as an error signal in a basic PID controller.

The course posed several challenges to successfully implementing this method. There are various entryways which are offset from the hallway by roughly one meter. The vehicle would tend to dive into these alcoves due to the increased error and would not correct in time to avoid collision with the end of the alcove. Furthermore, the reflective kick plates on these doors made determining the true distance from these doors difficult due to their specular nature. Most corners had a wall directly in front of the current path of travel, making them a potentially useful feature for indicating the need to turn. However, one corner did not, forcing the turn method to be more general.

A number of variations on the wall-following algorithm were attempted. The base implementation considered on the left hand lidar inputs constrained to some range, symmetrical about the short axis of the vehicle. The cosine of each of these points was taken and averaged to give an approximate distance from the wall. Additional implementations included the mirror image of this range about the long axis, capturing data on the right as well. This was done in an effort to have the vehicle center on the hallway, rather than using only the left side offset, in hopes this would be more robust to the alcoves and kick plates. Additionally, a small front-facing range was incorporated in order to detect when a wall was approaching and allow for a higher gain/tighter turn radius. The issue with both these methods is that they were not robust to cases where the vehicle was not parallel to the hallway and corners. These methods often caused aggressive turns which became circles after failing to find the walls.

Another challenge was how to process infinite returns, especially those returned by specular surfaces, such as the kick plates. In some cases, these were set to some high value in order to bias the mean to a higher value. In other cases we threw away these values and only used returns within the range of the lidar. Likewise, we also attempted to use the minimum of these range returns rather than mean in some cases.

The implementation of this system which was successfully able to complete the course was fairly simple. In one script, the lidar range was clipped from roughly 15 degrees left from center to 80 degrees left from center, such that the returns from the front right were considered and averaged into a distance from the wall. Infinity-value returns were not considered in this average. Using this value minus an offset (1.6 meters) as error, a basic proportional controller script was used to maintain position. Throttle was set to a fixed value. The offset from the wall, throttle, and proportional term were tuned until the vehicle was able to successfully navigate the course. This implementation was successful as it provided positive feedback in the case of the alcoves. This is due to more forward-looking returns being incorporated, more accurately predicting what was to come by the time the actuation had taken place and avoiding front-left impacts by not considering distances to features that had already been passed. Likewise, since the lidar was incorporating data from further in front of the vehicle, turns toward an upcoming alcove were likely corrected by the presence of a wall on the immediate left of the vehicle as it began its turn. The scripts which ran this algorithm are in the attached code folder. One script returns

A. PID Controller

A PID controller was designed to aid with the wall-following solution. This required us to find a deviation (error) from our desired distance from the wall to the measured distance from the wall using the lidar scanner. One of the variations of the PID controller that we tried implemented the formula, incorporating all three PID terms:

$$\Theta = K_{\mathrm{p}} e(t) + K_{\mathrm{i}} \int_0^t e(t)\,dt + K_{\mathrm{d}} \frac{de}{dt}$$

While we had some success with this controller and came close to completing the full loop, the controller was unstable and we were never able to find values that caused the DeepRacer to complete the track.

Although a number of PID control methods were tested, the implementation of wall-following above only utilized a proportional term for simplicity, as the error term was sufficiently clean to limit oscillations around equilibrium conditions at the desired offset distance. Speed likely played a factor in this stability, as features differing from the nominal wall distance moved out of frame quickly.

VII. Results & Discussion

With the basic wall-following implementation, the DeepRacer was able to successfully complete several

(roughly 10) successful laps around the course at a quick pace, achieving an estimated lap time of approximately 45 seconds. There was some variance on the success of navigating corners, especially when the vehicle dynamics were altered with additional payload, such as cameras. There was one successful case of unexpected obstacle avoidance when a pedestrian approached a blind corner. However, this caused the robot to miss the corner.

Ideally, this minimal solution would have been implemented first, in order to get a sense of how the vehicle and sensors perform. A full SLAM and path planning solution would be much more robust to deviations from the ideal path and would be able to better account for obstacles. However, the simple solution, while not robust to environmental changes, is effective at navigating the course.

During this project, we learned a lot about ROS and setting up hardware. We also learned about different path planning algorithms and the different effects that each algorithm would have on a project.

We would have liked to have gotten our original implementation working, and would have probably started the project earlier in order to do so. Also, adding GPS to our bot could have helped with localization, but the location of the course would have probably decreased the accuracy of the GPS.

VIII. Implementation Reference/How-to

A. Web Portal

In order to connect to the web portal, you first need to connect a laptop to the DeepRacer via the micro USB port. Afterwards, you can connect your DeepRacer to the internet by navigating to the DeepRacer homepage and logging in with the password provided on the bottom side of the DeepRacer. The web portal can be accessed by clicking on or navigating to the IP address provided at the top of the page after connecting to the internet. From the web portal, you can enable SSH connections by going to the Settings tab, scrolling down to the SSH section and selecting enable. SSH will need to be done after reflashing as well.

B. ROS Versions and Reflashing

For our project, we decided to use an Ubuntu 16.04 image with ROS Kinetic which can be downloaded using this link. Instructions on how to reflash your DeepRacer can be found at this link.

C. Manual Drive with Controller

In order to control the DeepRacer manually with a controller or a joystick, you will need to use the AWS DeepRacer Gamepad Control. The instructions for setting this up can be found in the README file of their github repository.

For our manual drive, we decided to use an Xbox controller which was having trouble connecting via bluetooth. If your bluetooth devices are having trouble connecting as well, enter the following commands in the terminal:

>> bluetoothctl

>> power off

>> power off (just for good measure)

>> power on

Wait until the controller stops flashing.

>> exit

If controller is connecting and disconnecting repeatedly:

>> bluetoothctl

>> scan on (look for xbox controller MAC address)

>> trust <MAC_ADDRESS>

D. Battery Lockout

It is important to note that if the vehicle's actuators are not moving after publishing to the "/maual_drive" topic, it may be due to the 7.7-V battery being locked-out. This battery is intended to solely be used for the DeepRacer vehicle and instructions to unlock the batter can be found here.

E. Restarting Core Services

This should be done if you have unplugged the cameras or the lidar and need them back. This can be done with systemctl to restart the service. This service is a launch file that starts the rplidar, cameras and control node.

>> sudo systemctl restart deepracer.service

Might break when you connect through the web portal. (Don't do this. Get ssh setup as fast as possible and never look at the web portal.)

F. Publishing with Built-in IMU

The DeepRacer has a Bosch BMI160 IMU built in. This IMU is connected to the DeepRacer via an I^2C bus. An example Python script to pull data from the device's IMU is available from the reply on this AWS forum. This script can be used as a ROS node to publish to the "/scan" topic. The data looks valid from this IMU, however with limited time being a major constraint, this sensor proved difficult to functionally integrate.

If you are having trouble with finding what address the imu is on the I^2C bus: use the I^2C detect tool in linux! This will scan all possible device hardware addresses and try to identify if something is on that address.

F. Publishing with Fidgets IMU

The phidgets IMU was added to the DeepRacer, as the Bosch IMU had very little documentation, whereas the Phidgets IMU has a lot of resources for integration with ROS. These resources include getting necessary drivers and how to use already-tested filters to work with the data from the Phidgets IMU. An example of using the Phidgets IMU with ROS can be seen in Figure 2 below.

As for installation, the Phidgets IMU was mounted beneath the RP LiDAR scanner with $\hat{x}$ going forward, $\hat{y}$ to the left of the vehicle and $\hat{z}$ pointing towards the ground.

These coordinate directions were made such that we could implement SLAM using *Gmapping*.

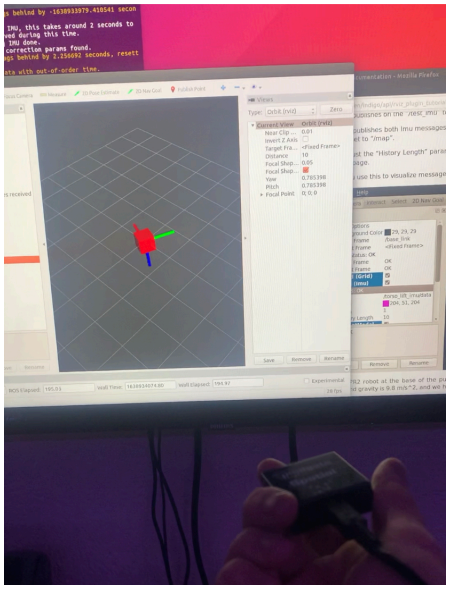

Figure 2: Example of using Phidgets IMU with *rviz* to see coordinate rotations.

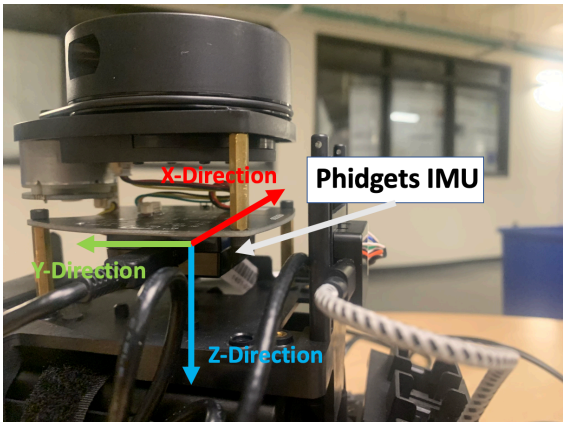

Figure 2: Phidgets IMU placement on DeepRacer with $\hat{x}$, $\hat{y}$, $\hat{z}$ directions labeled.

G. Laser Scan Matching and GMapping with Bag Files

In order to run local packages with the demo script which launches both the laser_scan_matcher and the GMapping node (in RVIZ) do the following:

1. Follow the installation steps for Scan Tools.
2. Locate the demo launch file referenced in this tutorial. It should be in the "share" folder of the ROS installation directory in the "/opt" folder in Ubuntu (/opt/ros/noetic/share/laser_scan_matcher/demo) .
3. Either edit or replace the demo.launch (for just scan matching) or the demo_gmapping.launch file.
4. The rosbag playback node should be edited to reference the location of the local bag i.e. *<node pkg="rosbag" type="play" name="play" args="/<bagfilelocationhere>/bagnamehere.bag --delay=5 --clock --start=30"/>* where the delay adds delay to the replay after launching RVIZ, the clock argument sets the parameter use_sim_time, and the start argument selects the startpoint of the bag in seconds.

Below this there are sections for publishing fixed frames, starting RVIZ, starting laser scan matcher, and starting GMapping. Additionally, arguments for the packages can be added in each section using the format shown in the script.

Once changes are saved, the script can be run using the following command: *roslaunch laser_scan_matcher demo.launch*

I. Collecting ROS Bags

See Recording and playing back data **Note: We don't commend rosbagging the camera topics as it significantly decreases performance and leads to dropped frames.** This is due to slow disk write speeds and low device storage capacity.

J. Spare Parts

Spare parts for the DeepRacer can be found through this link. This is a great compliment to the platform, as parts are sure to break. If the part is not in stock, there are many other alternatives on the market- just be sure to verify that the part will be compatible with the DeepRacer.

References

[1] https://aws.amazon.com/deepracer/

[2] A. Censi, "An ICP Variant Using a Point-to-Line Metric", IEEE International Conference on Robotics and Automation. Pasadena, CA, May 2008.

[3] G. Grisetti, C. Stachniss, and W. Burgar, "Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters", IEEE Transactions on Robotics, Volume 23, pages 34-46, 2007

[4] S. M. LaValle, "13.1.2.1 a simple car," Planning Algorithms. [Online]. Available: http://planning.cs.uiuc.edu/node658.html.

[5] K. Kurzer, "Path Planning in Unstructured Environments : A Real-time Hybrid A* Implementation for Fast and Deterministic Path Generation for the KTH Research Concept Vehicle," 2016. [Online]. Available: https://github.com/karlkurzer/path_planner/.