Field Scale Autonomy for Vineyards

David Robinson, Harikrishnan Suresh, Matt Swenson, Rahul Ramakrishnan, Vivek Ramaswamy
Advisor: George Kantor

***Abstract*— In this paper, we present an architecture for an autonomous mobile platform for the purposes of navigation within a vineyard. Our system uses on-board sensors to perform localization, perceive non-traversable obstacles in the environment, as well as the target vineyard row to enter. It then uses this information to generate and execute a simple trajectory towards a local goal point, while avoiding static and dynamic obstacles.**

I. INTRODUCTION

A. Motivation

This project prescribes the development of an autonomy architecture for the purpose of navigating a wheeled vehicle safely through a vineyard. The vehicle acts as a platform for an autonomous pruning system, the development of which is motivated by the challenge of the labor intensive task of vineyard pruning in an economic climate where labor costs are high and workers are scarce.

Previous work on the vehicle has involved designing a system for navigating between vineyard rows, as in Fig. 1. However, creating an autonomous solution for the mobile platform to travel between a parking area and a specified vineyard block remains a pertinent challenge. This task is critical for a fully functional system as the vehicle will need to navigate to specific vineyard rows during operation, and will also need to park in a specified area for storage when the system is not being used. Performing this task will require the platform to negotiate less structured terrain than in the vineyard, and to avoid obstacles which may arise on the planned trajectory, such as humans who are sharing the pathways.

Fig. 1: Desired path for robotic system

B. Project Goals

The high-level goals which are required for the system are as follows:

1) The robot shall travel from a known starting location to the start of a vineyard row
2) The robot will avoid static and dynamic obstacles

In order to simulate the target environment of a vineyard within our limited testing facilities, the team set up two traffic cones in place of a vine row entrance, and marked them by attaching April tags in a similar manner to the project's actual test site in Erie, PA. A graphical depiction of the project goals, along with the April tag setup is shown in Fig. 2

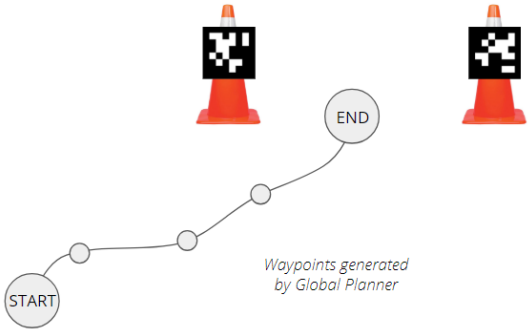


Fig. 2: Testing plan for robotic system

C. Robotic Hardware Platform

The hardware platform and the associated sensors for navigation are outlined in Fig. 3. The system is a re-purposed CMU robotic platform named Cave Crawler, which was used to create maps of underground mines. The upper section of the system is dedicated to the vine imaging system and vineyard pruning manipulator. The vehicle can be controlled with a manual controller, or overridden with a digital input from an bluetooth Arduino configuration.

D. Key Challenges

There are a number of key challenges associated with realizing this autonomous system. One of the most pertinent of these is related to the nature of the platform and sensors which we are working with; the platform is dated and as such, interfacing with its extant suite of hardware will be difficult. Also, there is a large number of available sensors for use, and each of these will require interfacing and calibration, which the team anticipates will be a time-consuming task. Another key challenge that we anticipate is the low level of

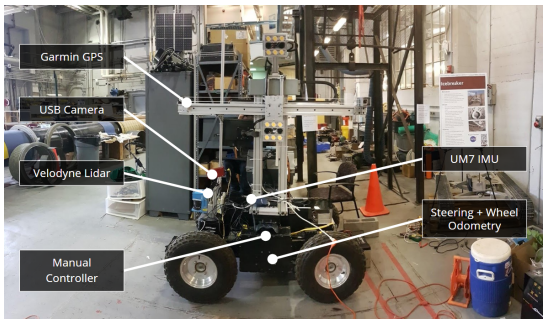


Fig. 3: Robotic platform

global accuracy for the GPS which we are using. In order to address this issue, we are planning to use April Tags for more accurate local planning once the goal destination becomes closer. Also, for our stretch goal of starting the autonomous navigation path inside a parking garage, we anticipate challenges associated with localization accuracy for the robot when it is indoors or close to buildings.

II. SYSTEM ARCHITECTURE

An overview of the system architecture is depicted in Fig. 4. and is then described.

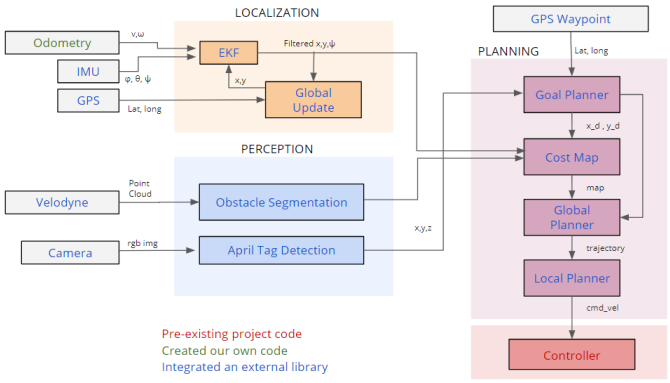


Fig. 4: System Architecture

A. Localization

This subsystem is responsible for actively localizing the robot, so that the position of the robot with respect to the world is known at all times, and this is a very important step towards making the system autonomous. The input to this system are the sensor readings from odometry, IMU and GPS. The odometry provides both the linear and angular velocity v and ω. The IMU gives roll ψ pitch θ and yaw Ψ as an output and the GPS provides latitude and longitude information. The sensor readings from IMU and odometry are then fused using the EKF from the ROS *robot_localization* package, while the GPS readings are sent to the global update to get fused x and y coordinates from the ROS *navsat_transform_node* node which is fed to the EKF as to obtain a final estimate of the robot's position and heading.

B. Perception

The Perception system takes in sensor readings from a LIDAR and a RGB camera. The point cloud generated from the LIDAR is clustered using eigen-clustering algorithm in the *Point Cloud Library*, which essentially detects the moving objects near the robot as an obstacle. The output from this module is the position of the detected object clusters. The RGB camera is used for detection of April Tags so that the robot can safely align itself to the center of the beginning of a row before performing in-row navigation.

C. Motion Planning and Control

The autonomous navigation software is built using the *ROS Navigation Stack*. The planning sub-system takes in the values from both the localization and perception sub-system for generating a plan for the robot to move. The x, y and yaw readings from the localization sub-system and the point cloud of the moving obstacles are used in the generation of a cost-map, which essentially populates the environment with obstacles and the current position of the robot. The planning sub-system also receives a GPS way-point from the user which is sent to the goal planner. Afterwards, the global planner uses the map and the x and y coordinates of the destination and creates a trajectory. This trajectory is then sent to the local planner, Timed Elastic Band (TEB), which essentially takes in this trajectory and converts it into the command velocity. In the final stage, the command velocity is sent to the low-level controller which is responsible for moving the robot to the required destination.

III. SYSTEM DESCRIPTION

A. Hardware Interfacing

The process of obtaining odometry information from the motor drivers and wheel encoders involved significant hardware interfacing. A custom D-Sub connector was made which ported the appropriate connections from the *Encoder Equivalent Output* of the Kollmorgen Servostar motor driver to an Arduino. The encoder signals were interpreted through the Paul Stoffregen's Encoder library [1] and were converted into metric scale velocities. The steering angle was obtained by attaching two connecting wires to the *Linear Pot* + and *Analog GND* terminals of the IDC B8501 Digital Brushless Analog Position Control module and then using the Arduino on-board +5V ADC to interpret the steer angle signals.

A UM7 IMU was interfaced via a Sparkfun FTDI ROS driver breakout board, which allowed direct connection via USB to obtain IMU sensor data in a *sensor_msgs/Imu* format using *ros_serial*. A housing was then created to securely contain the sensor related electronics, as in Fig. 6.

B. Localization Subsystem

The localization subsystem uses wheel odometry for the forward velocity and heading velocity, an IMU for yaw velocity and a GPS for global x,y position corrections. The raw wheel odometry is received as a step pulse which is converted to a velocity and the values were calibrated

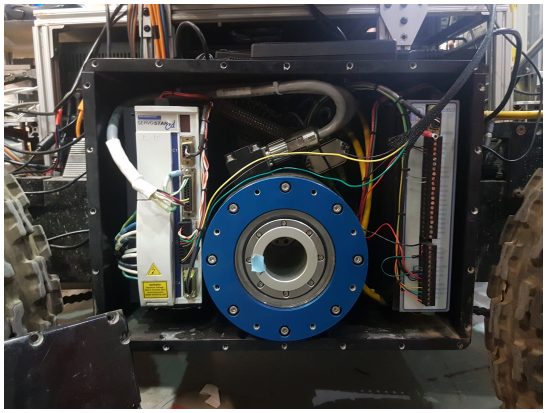


Fig. 5: Odometry data acquisition

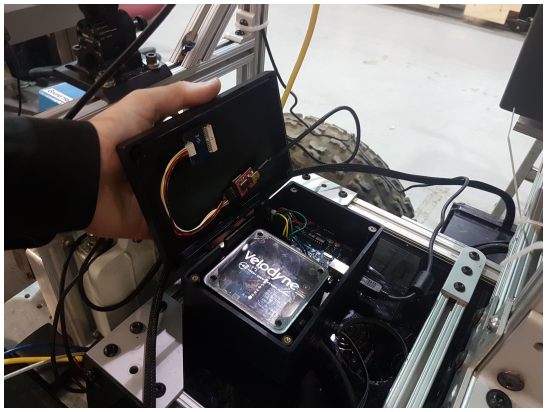


Fig. 6: IMU, Arduino and Velodyne housing

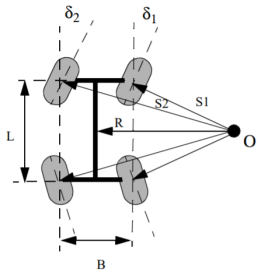


Fig. 7: 4-wheel steer motion model [2]

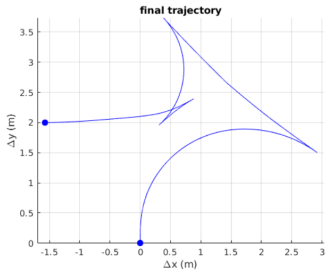


Fig. 8: Sample trajectory

by driving the vehicle a known distance and adjusting a scaling factor in order to achieve the appropriate distance for an integrated trajectory. The steering angle is received as a raw ADC signal which was then calibrated to steering angles by gathering a dataset of wheel angles and voltage values and performing a linear regression fit. The calibrated odometry is converted to a forwards velocity and steering angle through the 4-wheel steer motion model in Fig. 7 and through Equations 1, 2 and 3.

$$v_t = (v_{left,t} + v_{right,t})/2 \tag{1}$$

$$O = \frac{L}{2} \times (tan(\delta_{1,t}) + tan(\delta_{2,t})) \tag{2}$$

$$\omega_t = atan2(v_t, \sqrt{O^2 - v_t^2}) \tag{3}$$

A known trajectory was then plotted using MATLAB, as in Fig. 8, and the path accuracy was validated qualitatively by comparison to a video of the robot motion. The IMU and odometry data have been successfully integrated into the *robot_localization* EKF package in ROS to produce a local EKF.

A U-Blox Precise-Point-Positioning GPS unit was sourced in order for us to obtain global position information, and a second EKF was created in the same manner with as the first, but with filtered GPS data from the *navsat_transform_node* to create a global estimate of the robot's position. The team considered alternative methods for localization which involved building a map of the environment and localizing from it, but have chosen the Dual-EKF approach in the interest of simplicity and scalability.

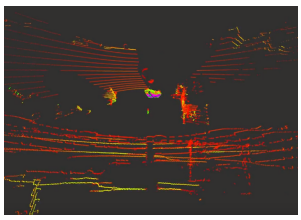
The most significant challenge associated with localization was obtaining the odometry sensor data from the robots motor drivers. This required reading outdated online manuals and a lot of experimentation with the output signals from numerous ports, cable making and creating housing for the final Arduino which runs the *ros_serial* node to publish the odometry data.

C. Perception Subsystem

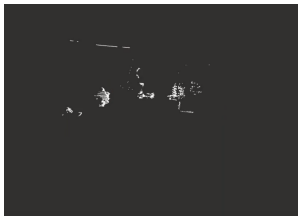
Most of the points from the incoming point cloud are not necessary for developing our costmap and also consume significant processing time. Hence we considered points that correspond to obstacles within 5m and 270 degrees of the field of view of the LIDAR. After filtering the points based on these conditions, we used euclidean clustering to associate points to objects and segment them. A clustering method divides an unorganized point cloud into smaller parts so that the overall processing time significantly reduced.

A simple data clustering approach which uses a Euclidean distance metric was implemented by making use of a 3D grid subdivision of space using fixed width boxes, or more generally, an octree data structure. This particular representation is very fast to build and is useful for situations where either a volumetric representation of the occupied space is needed, or the data in each resultant 3D box (or octree leaf) can be approximated with a different structure. In a more general sense however, we made use of nearest neighbors and implemented a clustering technique that is essentially similar to a flood-fill algorithm. Fig. 9a shows the raw velodyne data containing all the points whereas Fig. 9b shows the filtered and clustered points that is required for building costmaps.

April tags are used for goal sensing. Once the robot reaches the rows of the vineyard from its start location, April tags attached to start of the vineyard rows are used to center



(a) Raw velodyne data



(b) After clustering

Fig. 9: Euclidean clustering results

the robot between the vineyard rows. The current algorithm identifies the two April tags and finds the transform for the mid-point between them. These transforms are to be given to the robot as way-points towards the goal.The detection of April tags is done using the *ros-apriltags* package and a point grey chameleon camera which is interfaced through the *pointgrey_camera_driver* ROS package. The transforms for each April tags are obtained by subscribing to the detections topic from the ROS package. These transforms are manipulated to find the mid-point between the two April tags and find the corresponding transform with respect to the robot. Fig. 10 shows the results of initial April tag detections using the April tags ROS package.

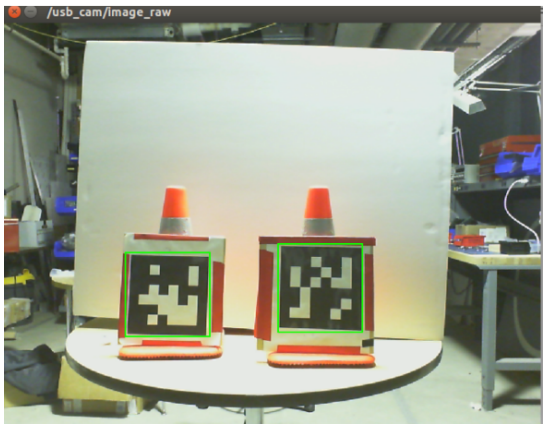


Fig. 10: Indoor April tag detection test results

D. Motion Planning and Control Subsystem

1) Costmap generation: The ROS navigation stack contains a global and local costmap. We started the global costmap with an empty space (no prior occupancy grid map) since we wanted our robot to adapt to new environments. The local costmap was then built incrementally as the robot's sensors discovered new regions of the environment, and updated to the global costmap. The output of the LIDAR perception module was used to discover the traversable region free of obstacles and build a local costmap. The segmented point cloud was seen to contain significantly lower noise thereby enabling generation of clean costmaps. All the obstacles were given an inflation radius of 0.1m to account for factor of safety while the robot navigates around them. The robot was assumed to be circular with a footprint padding of 0.1 given for safety. Fig. 11 shows the costmap generation using the segmented point cloud.

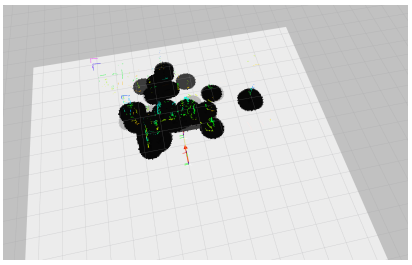


Fig. 11: Costmap generation results

2) Global Planning: Using the global costmap, the global planner generates path for the robot from the start position to the specified goal position in order to avoid obstacles. As the global costmap updates, the global planner re-plans the path to ensure there are no collisions with the obstacles.

For our project, we started by using a custom global planner written using the Open Motion Planning Library framework [2]. RRT* was chosen as the global planner since it generates an optimal path over time. This planner was added as a library, which the move_base package of the navigation stack calls to generate the path. The collision checking is performed by calling the updated global costmap and comparing the footprint of the robot with the location of the obstacle to calculate a footprint cost. A potential collision would result in a negative footprint cost thereby avoiding that location in the global plan. One thing to note is that we did not consider the kinematic constraints of the robot for the global planner, since the local planner was written to give velocity commands based on the global plan respecting the constraints. Fig. 12 shows the output of the RRT* planner in RViz, with the global plan in green.

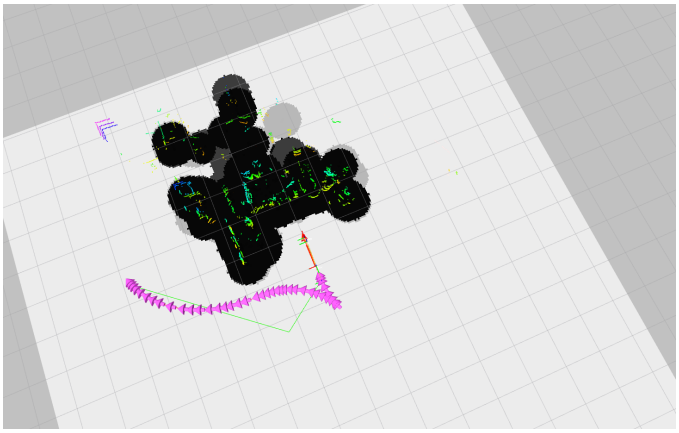


Fig. 12: RRT* output

We tested our system with the default global planner of the navigation stack that used a discrete search algorithm, navfn, as the global path planner. The navfn planner

provides a fast interpolated navigation function that can be used to create plans for a mobile base. The planner assumes a circular robot and operates on a costmap to find a minimum cost plan from a start point to an end point in a grid. The navigation function is computed with Dijkstra's algorithm. Fig. 13 shows the output of default planner in Rviz.

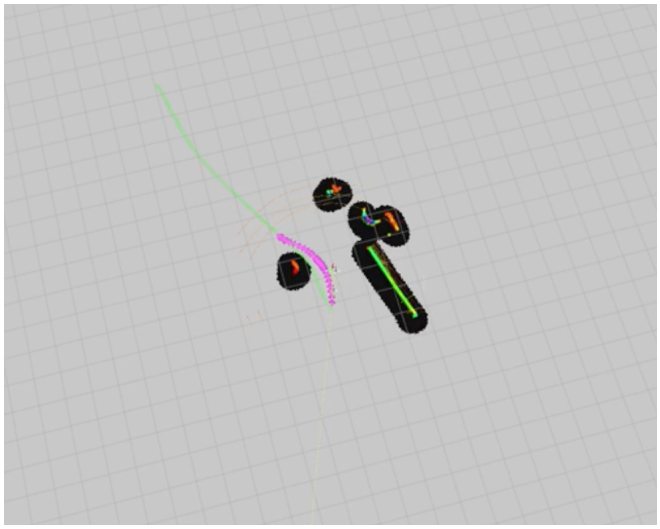


Fig. 13: navfn output

3) Local Planning: The robot uses TEB local planner [3] to generate Ackermann-drive feasible paths irrespective of the feasibility of the global plan. TEB performs dynamic optimization of over a dozen different control parameters, which required a significant tuning period to achieve proper performance on the robot. Figs. 12 and 13 contain the output of the TEB planner displayed in pink.

4) Control: The original CaveCrawler platform was designed to operate autonomously using hardware that has been since removed from the robot, or via a USB joystick. The other teams working on the robot have adapted the USB interface into a platform that (tenuously) supports commands sent via ROS. Control velocities received by a ROS node are sent over a *ros_serial* node to an Arduino, which communicates via XBee radio to another Arduino connected to the original USB connection.

The remaining control system on the robot translates forward and rotational velocity commands into forward movement and wheel angles, so it was not necessary to transform the output of the TEB planner into a form more closely related to the double-ackermann drive the robot uses.

IV. SYSTEM EVALUATION

In this section we will discuss how both the sub-system components as well as the system as a whole performed. We will describe what we have achieved through the project and highlight limitations and future work.

A. Subsystem-Level Evaluation

The Local EKF performed well, as the IMU and wheel odometry data was relatively noise free and accurate. However, when we attempted to integrate the GPS with the global EKF, there were some issue with the noise in the GPS signal. Given the physical constraints of the robot and the roads leading from NSH, the only feasible outdoor testing environment is the asphalt area directly outside of the Field Robotics Center. We tested the GPS unit in the CMU Cut, as in Fig. 14, a wide open space with good satellite visibility, and obtained a mean covariance for Latitude and Longitude of 1.04m. A similar test was performed outside of the FRC and it gave a mean covariance of 3.42m.

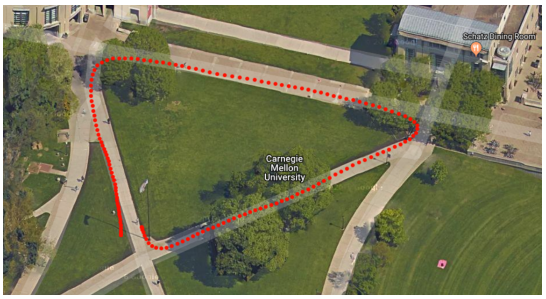
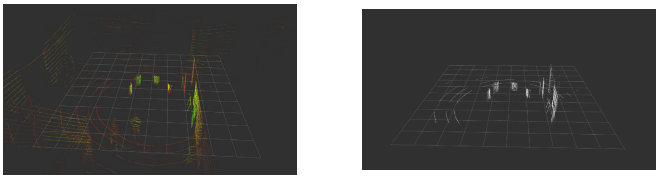


Fig. 14: GPS testing results

The algorithm for filtering and clustering was tested using a bag file recorded using a Clearpath Husky. The input point cloud at each time-step contained around 23,000-40,000 points including outliers, points corresponding to obstacles and noise. After filtering with our algorithm, the points corresponding to obstacles required for developing the costmap reduced significantly to 5,000-8,000 points, thus decreasing the processing time from 36-43 ms to 14-19 ms. This makes the update of obstacles in the costmap much faster and closer to real-time. Fig. 14 shows the testing results obtained using the bag file recorded from the Husky robot.



(a) Raw velodyne data (b) After clustering

Fig. 15: Euclidean clustering evaluation

Testing of the April Tag perception system was performed independently of the robot and was demonstrably able to accurately determine the center-point of the two tags. The system was validated by performing tests both in an indoor and outdoor environment.

The costmaps and planner were tested first with simulated data and then the actual robot. The costmaps were initially generated using the raw point cloud data which resulted in noisy sensor readings being treated as obstacles. This was fixed by feeding only the segmented and clustered point cloud for costmap generation. This results in a clean costmap containing inflations only at the obstacles. In addition, the point cloud was converted to a laser scan in the plane of the LIDAR before feeding it as into the costmap, because the point cloud source was seen to cause slow clearing of the costmap for dynamic obstacles.

Regarding the custom global planner, we faced many challenges since it was sometimes producing plans which would give collisions with obstacles. We spent a significant amount of time attempting to debug the issue, and finally arrived at the conclusion that there is some issue with the way the costmap is being interpreted by the planner. The planner identifies the obstacles at a certain offset from the actual location in the costmap. As such, the planner generates the path even when an obstacle is given as the goal but causes collision warnings when a slightly offset location is given. The custom planner is not robust enough and was not deployed on the final robot. Fig. 16 shows some results of the RRT* planner. The default planner proved to be robust enough and the final test results are obtained using it in the navigation stack.

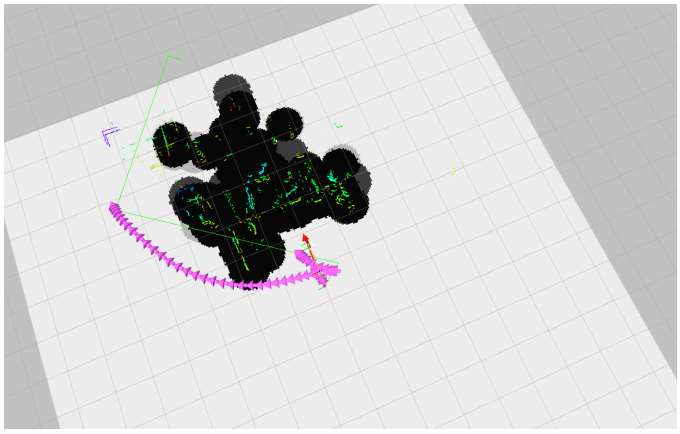


Fig. 16: failed RRT* output

Local planning, after calibration, performed relatively well. The TEB Local Planner reliably generated feasible paths and demonstrated very little of the frequent small-order rocking common in other implementations of TEB. However, there was a notable tendency for TEB to attempt to switch between multiple paths around smaller obstacles. In practice, hysteresis in the physical system prevented TEB from actually switching between the two paths.

B. System-Level Evaluation

The full system evaluation was conducted in a predefined environment outside the Field Robotics Center. The robot

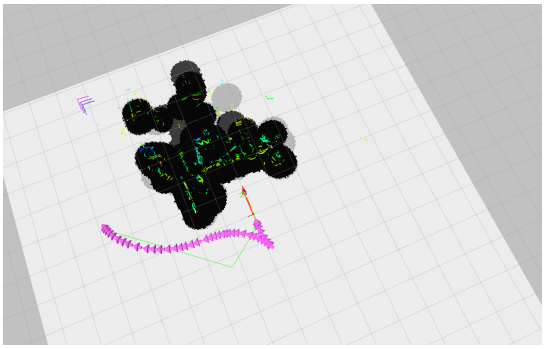


Fig. 17: Depiction of the integrated cost-map, global, and local planners.

was made to move towards the local goal position that was between two cones containing the April tags to simulate the ultimate navigation goal. The robot was assessed over four direct tests out of which it performed the autonomous navigation successfully three times. The system was also assessed in response to a static obstacle, a trash bin, and was able to negotiate the obstacle two out of four times, and never collided with the object. Finally, the system was tested with a dynamic obstacle, a moving trash bin, and was able to stop and re-plan it's trajectory without colliding with the object. However, in the case of the dynamic obstacle, the robot was unable to reach its goal.

We encountered significant issues with the costmap and rover model inflation, causing obstacles extremely close to the rover to fall off the costmap and cease being planned around. In combination with the large turning radius of the rover, this resulted in several tests being aborted just before the robot hit anything. In addition to the large turning radius, steering locking was a ubiquitous problem, and the motor reset function used to clear the steering-lock often disrupted planning, slowing the system down and reducing the quality of results. The April tag goal detection was performed separately due to software compatibility issues during integration.

C. Major Limitations

The current performance of the system is severely limited by battery life. The unpredictable nature of the steer locking problem means it is unlikely the system as it is would be able to perform truly autonomously. In addition, obstacle detection is performed using only a 2D slice of the full Velodyne data. Because of this, the system only detects obstacles approximately 1.5m off of the ground plane, which is insufficient for a wide variety of commonly encountered obstacles. The lack of full GPS integration and testing means we are unsure of how the robot performs over large distances.

D. Future Work

The following are the things which we plan to carry out in the future in order to capture all of the desired system functionality and to make it more robust.

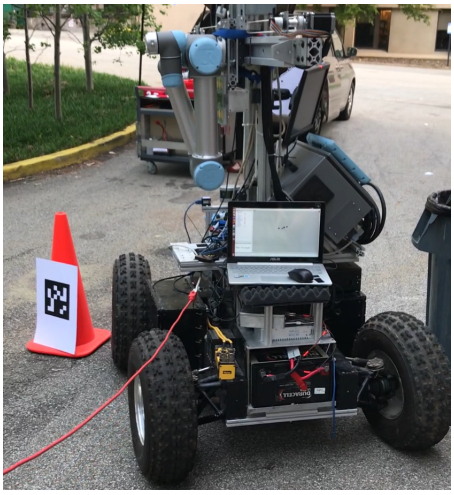


Fig. 18: Robot moving around static obstacles

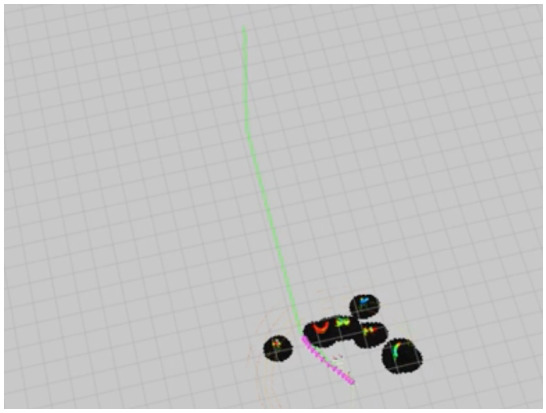


Fig. 19: Rviz output for robot following plan

- Expand the obstacle detection algorithm to be able to use the entire Velodyne point cloud.
- Perform ground plane fitting with the Velodyne data to improve planning and obstacle detection performance.
- Perform path shortening on the RRT planner's output paths.
- Integrate the global EKF for GPS way-point following capability.
- Integrate the April Tag to set the goal in the local planning frame.
- Replace and possibly upgrade the batteries used to power the robot.

A video of the robot in operation can be found at `https://youtu.be/KkF2rET7Z3c`, and the project code can be found at `https://github.com/harjatinsingh/CaveCrawler/tree/global_planning`

REFERENCES

[1] Encoder Library. (n.d.). Retrieved from `https://www.pjrc.com/teensy/td_libs_Encoder.html`

[2] Shamah, B. (1999). Experimental Comparison of Skid Steering vs. Explicit Steering for Wheeled Mobile Robot M. Sc.

[3] Wiki.ros.org. (2018). teb_local_planner - ROS Wiki. [online] `http://wiki.ros.org\/teb_local_planner`

[4] Ioan A. ucan, Mark Moll, Lydia E. Kavraki, The Open Motion Planning Library, IEEE Robotics & Automation Magazine, 19(4):7282, December 2012. `http://ompl.kavrakilab.org`, `https://ompl.kavrakilab.org/classompl_1_1geometric_1_1RRTstar.html#details`