Development of a Autonomous Navigation System for a Wheeled Ground Robot in High Pedestrian Density Environments

Shivang Baveja, Nicholas Crispie, Ritwik Das, Danendra Singh, and Karsh Tharyani

***Abstract*— This paper provides an overview of our project developing a software system for autonomous navigation and obstacle avoidance for a wheeled mobile robot around the Carnegie Mellon campus. The main software subsystems consist of State Estimation and Navigation, a Global Planner, and Perception and Local Planning. The robot hardware used is the Robotanist, a four wheeled mobile robot developed at the Field Robotics Center.**

I. INTRODUCTION

Autonomous navigation is an area of much technological development in the recent years, with self-driving cars occupying much of the development. However, less emphasis has been put on developing robust navigation of a wheeled, ground based robot in an environment with a high degree of unpredictable foot traffic. In this project, we develop a software and perception system to effectively navigate within Carnegie Mellon University, with point to point navigation and obstacle avoidance. Work in this area opens up more possibilities for the application of robots such as autonomous delivery systems. The platform that we use for developing the software system is the Robotanist[1], developed by Tim Mueller-Sim at CMU as part of a Advanced Research Projects Agency Energy project supervised by George Kantor. It is a 4-wheeled robot with skid-steer control and multiple sensors available for perception and navigation.

Each of the 4 wheels are attached to independently controlled motors with Hebi Modules, allowing access to wheel odometry through precise encoders and individual IMUs. The robot itself has state position estimates from a separate, more accurate IMU as well as a GPS module (non-RTK version). For perception, the Robotanist has mounted to the front of the robot a SICK Time of Flight Camera, which creates a point cloud in the direction of robot travel. Additionally, the robot has a Carnegie Robotics MultiSense Stereo Camera, which creates a point cloud at a higher height and wider field of view than the SICK ToF camera. These two sensors are used in the obstacle detection and costmap generation, described in more detail in a later section.

Fig. 1. The Robotanist in the wild, navigating around farmland

The key challenges involved in building such as system are threefold: The most basic one is interfacing with an existing setup of which we have limited familiarity with. The next two related to the actual system itself. Because we do not have an RTK GPS, the localization of the robot in our map will be challenging to do accurately. Additionally, building a proper costmap and planning around a combination of dynamic and static obstacles will be challenging given the unpredictable environment of CMU.

II. SYSTEM ARCHITECTURE

As seen in Figure 2, there are three main software subsystems that we are creating as a part of this project: State Estimation, Global Planner, and Perception/Local Planner. The trajectory execution and motion is well supported by the robot platform, so we do not need to focus on this for the scope of the project.

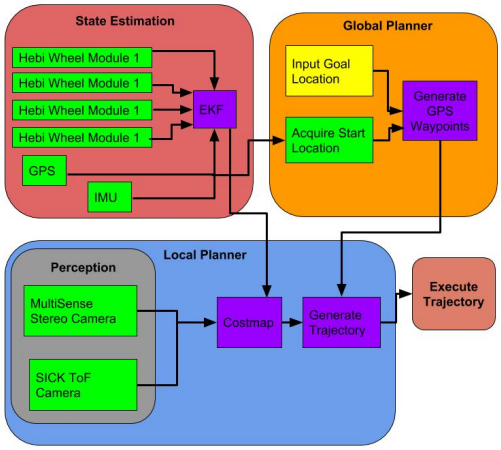

Fig. 2. Cyberphysical Architecture of our software system running on the Robotanist

The state estimation consists of fusing the odometry data from each of the Hebi Wheel modules (which have quadrature encoders and IMUs) with the Robot's IMU and GPS receiver. This gives a more accurate pose estimate for the robot for use in the Global and Local Planner. This fusion of the state data is done with an Extended Kalman Filter, based off of the Robot Localization package in ROS. There is some pre-exisiting code to relate all the frames in the robot, but we have to modify both those frames and the Robot Localization package in order to tailor the output of the EKF for our needs. Ultimately the State Estimation subsystem will output the current robot pose for use in the Global and Local Planners.

The Global Planner consists of an input of the current robot position, and a user defined end goal. This subsystem then computes a path taking advantage of the Google Maps API to generate a set of waypoints for the robot to follow. This only needs to be done once at the beginning of the robot's trip. These waypoints are then sent to the local planner one by one until the goal is reached.

The Local Planner is divided up into two parts: Perception and Trajectory generation. The data from the the SICK time of flight camera and used to generate a three dimensional point cloud. The raw point cloud has ground hits and some inherent noise which is removed using filters from PCL library. The filtered data is then published as ROS laserscan message which when combined with the robot's pose from the State Estimation subsystem, forms a two dimensional obstacle occupancy map based off of the Costmap 2D ROS package.

We also developed custom code for classifying the images from multisense camera into road i.e. traversable and "not road", i.e. not traversable by the robot. This image was then projected onto the ground. We publish laser scan data based on this projected image. This module has not been integrated into the costmap package in the interest of time.

The generated costmap is then used by the Local Planner to compute a trajectory in order to follow the waypoints from the Global Planner while at the same time locally avoiding obstacles in the map. The end output is a trajectory that is sent to the motor controllers. We are using TEB local planner for this purpose.

As mentioned above, we used off-the-shelf software packages and libraries for some of the sub-systems. These code were not plug and play and but took meticulous tuning of parameters which is also described in the following sections.

III. System Development

Each of the sub-systems are now described in detail.

A. State Estimation

We have implemented an instance of an Extended Kalman Filter, or the non-linear version of the standard Kalman Filter, using the ROS Robot Localization package. Figure 3 show the locus of the wheel encoder odometry points of our system. In this test circuit, we ended our path at the same place we started our robot at, closing the loop. However, one can see that the odometry propogated position at the end does not match the beginning, thus the need for a Kalman Filter.

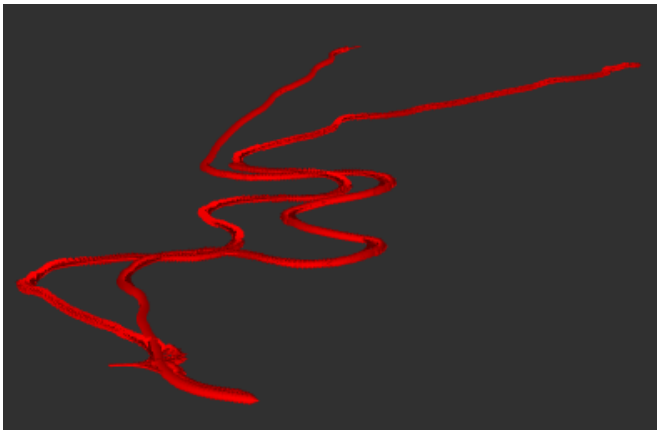

Fig. 3. Odometry data recorded from a test run with the Robotanist

We implemented a Dual EKF from the robot localization package, first fusing the wheel odometry with the robot's IMU and then combining that with a fusion of the wheel odometry and the GPS measurement. This architecture sets up a local EKF and a global EKF respectively, with different uncertainties associated with each. A preliminary result from the EKF output can be seen in Figure.

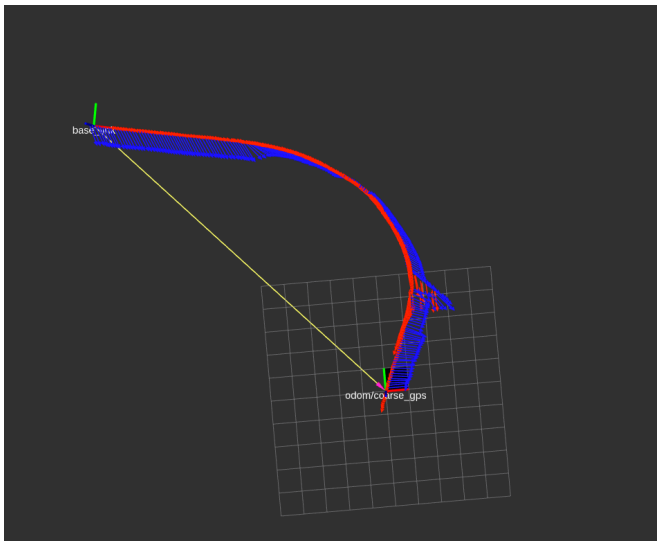

Fig. 4. Output of the Extended Kalman Filter (blue) along with the GPS (red)

One of the big challenges we have faced for the State Estimation is an extremely erratic GPS signal. We included a Mahalanobis distance gate to filter out these jumps. Additionally, tuning of the parameters in the filter has proven difficult. We found that increasing the uncertainty of the velocity of our robot improved performance, but this is still a work in progress in order to optimize the output of the filter.

We observed that even though we set the parameter in the EKF to remove gravitational-acceleration from the IMU to true, the base link was constantly drifting away even when the robot was absolutely stationary. Hence we set a relative parameter to true in order to overcome this issue.

B. Global Planner

The Global Planner in the Tartan Bot utilizes the Google Maps API for getting the way-points. As an input, it takes in the start position of Tartan Bot and, from the user, takes in the end position. All the way points are referenced relatively from the start position of the robot, and hence, the start position of the robot becomes the origin of the global frame and, also, such that the base link coincides with the global frame.

The ROS Navigation Stack offers a *move_base* action server which can be issued commands through an action client. The action client, we made, is responsible for querying the way-points to the *move_base* action server and is a node which listens to the pathPublisher node, and in turn, sends a sequence of way-points to Tartan Bot. The pathPublisher node is also responsible for converting the latitude and longitudes to East North Up (ENU) coordinates. A consequent way-point is only sent to the server once the current way-point has been reached. The message type of the way-points is geometry_msgs/Pose. Figure 6 is a schematic representation of the current Global Planner.

The orientation target of the way-point is set such that the robot gets aligned to the vector from that way-point to the next way-point.

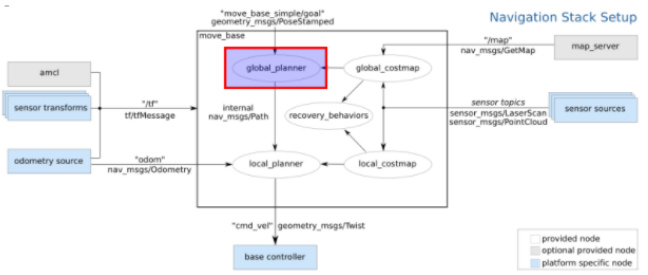

Fig. 5. Overview of Global Planner architecture

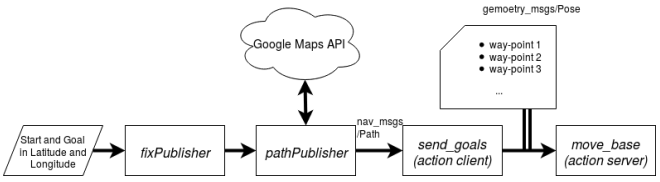

Fig. 6. ROS Architecture of the Global Planner

C. Perception

The robot has multiple perception sensors which were evaluated for our application.

1. MultiSense S7 stereo camera: The sensor provides instantaneous vertical and horizontal fields of view (FOV) along with color information for every range point found. It publishes a point cloud message which can be used for obstacle detection. Figure 7 shows pedestrian detected in the received point cloud. This point cloud published by multisense was found to be very dense and could only be updated once every 3

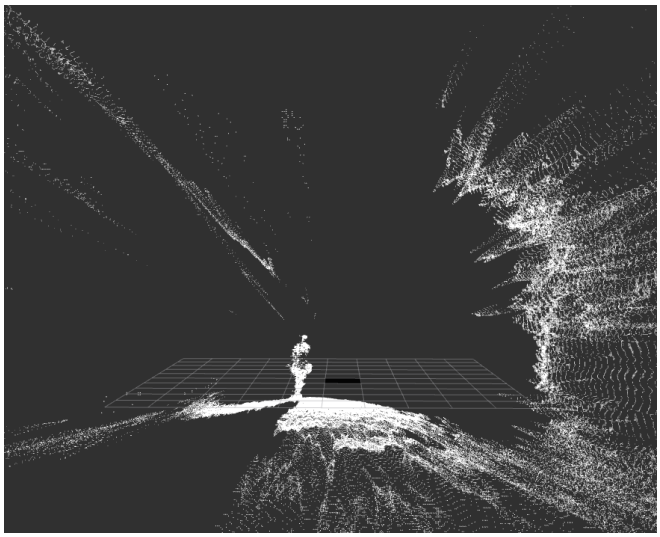

Fig. 7. MultiSense stereo camera seeing a pedestrian

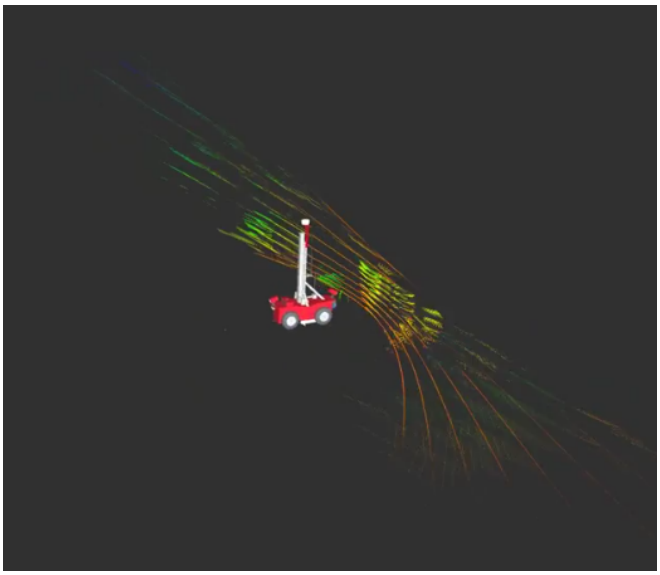

Fig. 8. Laser Scan from the Quanergy LIDAR

seconds. Since this rate of update is not acceptable we decided not to use point cloud from this sensor. The sensor also publishes raw images from both the cameras which were found to be good enough for the purpose of road detection.

2. Quanergy M8 LIDAR: This sensor has 8 lasers with 360 degree field of view and it generates 420,000 points per second. We tested this sensor and found it to be noisy with a lot of distortion in the point cloud data. Also, the sensor has been mounted with a downward tilt which limits its field of view in the forward direction. Since we were not allowed to do any mechanical changes to the robot, we decided to not use this sensor for our purpose. Figure 8 shows the data received from Quanergy LIDAR.

3. Sick Visionary-T time-of-flight (TOF) LIDAR: This sensor records up to 50 3D images per second and generates distance values of 144 x 176 pixels per recording. It also published point cloud data. Figure 9 shows raw point cloud from this sensor.

For navigation two tasks are required which are described now:

- *Obstacle detection:* For obstacle detection we are processing the raw point clouds received from SICK sensor using filters from Point Cloud library. The process is described in the Figure 10.
  - *Down-sampling:* Point Clouds are inherently rich sources of information from the environment. This leads to the issue of noise and other bogus information which is not required. Hence, we down sample the voxel points to obtain accurate obstacle planes (and not false obstacles) for generating a cost map.
  - *Statistical outlier removal:* Despite reducing the density of points in the point cloud from the raw sensor, chances are that some points which dont represent a plane still creep in from the sensor into the obstacle cost map. The statistical outlier removes these anomalies and ensures that only points which can be clustered on a whole to form a plane are only used.
  - *Ground plane filtering:* This is achieved using

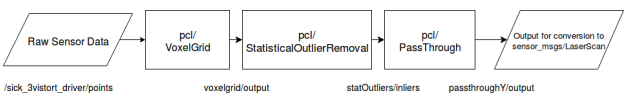

Fig. 10. Filtering the point cloud from SICK Tof

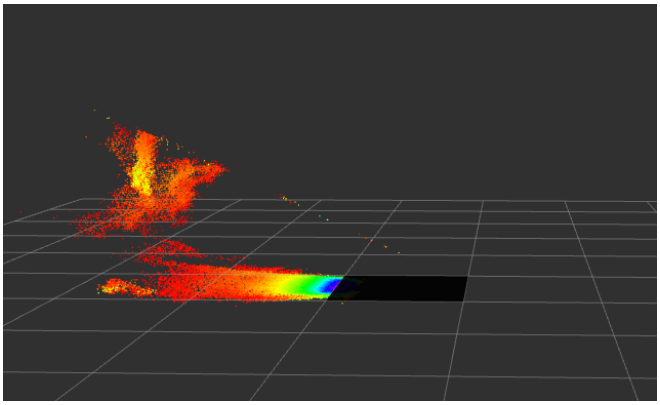

Fig. 9. SICK time of flight camera seeing a pedestrian

a pass through filter from the pcl library. The filter simply crops the points at a height threshold along the Y-axis of the sensor. This means that we only consider data points above a certain Y-value(threshold about the road data points).

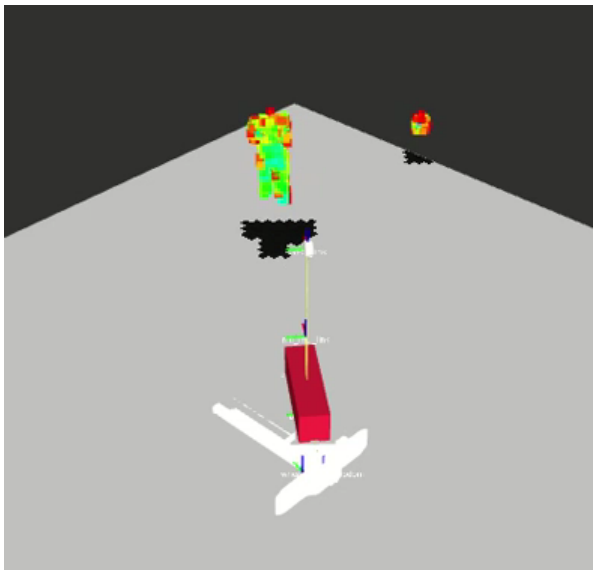

Fig. 11. Filtered output of the SICK LIDAR

  - *Point Cloud to Laser Scan:* We were originally using the point cloud data directly from the SICK sensor to generate the costmap. This was good for registering the obstacles but made it really difficult to clear obstacles from the voxel grid of the costmap. Same obstacles got registered in different layers of the costmap while the obstacle clearing was done only in one layer. To tackle this, we converted the point cloud data to 2D laser scan data using ROS package pointcloud-to-laserscan.
- *Road Detection:* The images received from multisense camera are used to identify traversable regions. For this purpose, we are using colour thresholding to segment out the gray regions from the image. The characteristics of the noise in the image closely resembled salt and pepper noise. Therefore a median filter is used after obtaining the binary mask. A top view of the resultant image is then taken so that we get the projected road. We finally obtain a binary mask of 0s and 1s representing non-traversable and traversable regions respectively. This mask is then used to publish a laser scan message which can be used with costmap 2d ROS package. In the interest of time, we havent integrated this module with our overall navigation pipeline. A final result of this is shown in Figure 12.

Segmentation.png

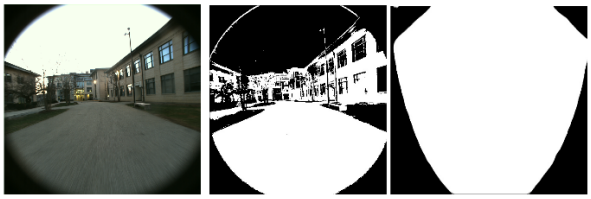

Fig. 12. Binary road classifier derived from SICK images

The SICK Time of Flight Sensor (SICK Tof) is the primary sensor that is used for making the cost map. During the mid semester report we were using a Y-axis pass through filter, accompanied by a statistical outlier filter, to filter the noise from detecting the road. Doing this we observed that the publishing rate from the final output of the filter was dramatically slower than that from the sensor itself. Given this, the update loop in the cost map was throwing out warnings stating that it was using old data to update its cost map. This, in turn, led to the problem of jerky motion while executing the plan output from the planner. In order to avoid this issue, we instead incorporated a voxel grid filter and stacked the outlier and pass through filter on top of it (in the same sequence as stated). Figure 10 shows the final filtering sequence on the raw point cloud data from SICK Tof. The original output is seen in Figure 9.

D. Local Planner

The local planning module receives the immediate goals from global planner in Cartesian coordinates and orientation and uses them to plan a feasible path for the robot accounting surrounding obstacles. The generated path is in the form of velocity linear and angular velocity commands published in the form of twist message.

There are two separate components which are run concurrently in order to plan a path:

1) Costmap generation: The laser scan published after filtering the SICK point cloud is used to update the cost map. We are using costmap_2d ROS package for this purpose. The obstacles are inflated based on inflation radius parameter and are projected on a rolling occupancy grid map where each cell is assigned a cost. The cost

assigned to an obstacle depends on the robot footprint and the inflation radius parameter set. The costs are divided into different levels based on possibility of collision depending upon robot position relative to the obstacle. These levels are lethal, inscribed, circumscribed, free-space and unknown.

Costmap 2d package has a lot of parameters which affect the accuracy and rate at which costmap is updated. Following parameters were tuned to generate the right behavior from the local planner:

a) *Obstacle range:* This was set equal to the range of sick sensor.
b) *Raytrace range:* This was set slightly higher than the max range of sick sensor so that obstacles can be cleared effectively when they are no longer in the field of view.
c) *Inflation radius:* It was reduced from 0.5 to 0.2. This was done to reduce the radius of turn when the robot sees an obstacle.
d) *Obstacle persistence:* It was set to 0 as we want the obstacle to get cleared as soon as it is out of the field of view of the sensor. This was required as are operating in an environment with lot of pedestrians. If we keep the obstacle for some time, the robot will have a tough time planning around them.

2) Planning: For local planning we were initially using `base_local_planner` ROS package. It is based on trajectory rollout and dynamic window approach which is a common approach for local control. This planner worked well with static obstacles but gave problems for dynamic obstacles. Specifically the planning generation was slow which caused the robot motion to be jerky.

We decided to switch to Time Elastic Band (TEB) local planner which provides fast re-planning and generates smoother trajectories. This planner locally optimizes the robot's trajectory with respect to trajectory execution time, separation from obstacles and compliance with kinodynamic constraints at runtime.

Following parameters were tuned to generate the desired obstacle avoidance behavior from the robot:

a) *dt ref:* Desired temporal resolution of the trajectory. It was increased from 0.3 to 0.5 to increase the planning speed. This forces the planner to take bigger steps while simulating the trajectory.
b) *dt hysteresis:* This is set to 10% of det ref. So it was set to 0.05.
c) *max vel x:* Max velocity in x direction was set to 1.0 m/s.
d) *max vel x backwards:* Max backward velocity in x direction. This was set to 0.2 m/s as we don't want the bot to move fast in backward direction as there are no sensors facing backwards.
e) *max vel theta:* Max turn rate was set to 1 radian per second.
f) *acc lim x:* Max acceleration in x direction was set to 1 m/s2.
g) *acc lim theta:* Max turn acceleration was set to 1 radian per sec2.

Figure 13 shows the path generated by TEB local planner around the obstacle.

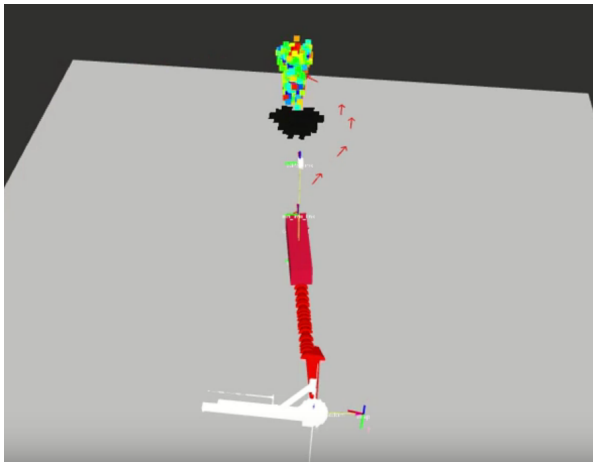

Fig. 13. Results of the costmap and planner that was implemented in our navigation stack

A big challenge here is the robot is skid steer (non-holonomic). Those constraints, in addition to the high wheel base to track-width ratio, make turning in place difficult, let alone moving sideways. Factoring in those limitations to the trajectory planning makes finding a solution for the local planner more difficult. Moreover, we anticipate autonomous navigation around CMU to be tough, given the large number of pedestrians that could get in the way. This presents a challenge for the costmap update and planning. We are considering moving to a Time Elastic Band (TEB) planner that allows fast re-planning in the event of dynamic obstacles.

IV. Evaluation and Results

As we were developing the system, we evaluated each subsystem as we went along to verify individual correctness before integrating and evaluating the system as a whole. One of our first subsystems to get validated was of EKF module testing how well the state estimation would work. We did this by taking the robot out of a predefined track looping around the plaza between Newell-Simon and Smith Halls and completed one full loop with loop closure. Based on the map of the known trajectory and the error from the loop closure, we were able to meet our standard of localization within 1 meter over the course of the entire trajectory.

Next, we verified that our costmap was rendering correctly by checking what the output was from a recorded bag file in which we placed obstacles at specific points in the route. We tested that the costmap was working correctly when we saw that the obstacles would be generated in the costmap at the appropriate time in the bag file data. We validated this functionality by doing a live test of the costmap with a dynamic obstacle by having one of us walk in front of the robot while it was driving forward. In this test we were able to meet our requirement of the obstacle appearing in the costmap within 1 second, and clearing correctly within 1 second, as well as placing the obstacle to within 1m accuracy within the map by checking the output of the map vs what we measured and observed in the real world.

Continuing with our subsystem testing, we verified the output of the global planning by setting waypoints for the robot to drive to (without obstacles) and checking to see how closely the robot would follow that. The skid-steer of the robot made it difficult to plan directly to the waypoint, so the robot had to plan multiple steps to achieve its goal. once it was verified that the robot could drive to each of these waypoints and get to the desired pose to within 1 m translational position, we tested the system local planner by placing an obstacle in front of the robot and plotting the planned trajectory in rViz to verify that the output of the planner was reasonable to achieve the desired goal configuration.

To evaluate the entire system, we structured a test as a combination of our subsystem tests by setting up a number of waypoints for the robot to plan to, and a couple of static obstacles (people standing in place at predefined positions). We then had the robot drive along the planned route, navigating around the static obstacles as it progressed towards the end goal. A setup of this test is shown in Figure 14. We added a couple of random dynamic obstacles when we walked in front of the robot at random times during the execution of the trajectory. Our criteria for success was the robot needed to navigate to each waypoint accurately within 1m and get to the desired goal without running into either the static or dynamic obstacles. By the end of our development, the robot was able to complete several trials which met these criteria.

Fig. 14. Validating the full system in a live test

V. Organization and Reflection on Project

A. Division of Work

The work was divided based on interest of the individual team members. Ritwik was responsible for the Google maps API integration with the global planner and developing road detection routing. Danendra was responsible for generating the 2D costmap from the SICK time of flight sensor. Karsh was responsible for state estimation, filtering noise in SICK sensor and developing the ROS Navigation stacks action client program. Nick worked with Karsh on data collection and state estimation. Shivang worked on the configuring local planner and integration of all the sub-systems.

B. Time Analysis of Individual Components

Costmap and Local Planner were the two most time consuming components. For costmap we had to filter the noise, publish laser scan from the point cloud and tune a lot of parameters. For local planner we had to tune parameters which took a lot of time as we had to run the bot every time to see the effect of the changed parameter.

C. Key Challenges

This robot is not designed to navigate in a campus like environment with dynamic obstacles. The robot cannot turn in place which becomes a problem if you are in a obstacle rich environment. The sensors were not configured properly and It took us a lot of time to set up everything correctly. There was no freedom to move the sensors which made it difficult to get the desired configuration of sensors which provides full 360 degree coverage. SICK sensor is very noisy especially in daylight scenarios. Even at night there is some fixed noise which had to be filtered using statistical outlier removal filter. IMU ROS node kept crashing when using the bot. We couldnt find the reason for that as all the wirings were not accessible. Scheduling work among the team was a major challenge as everyone has been really busy with the coursework this semester. To work with the robot GPS reception was required, so we had to take the robot out every single time.

D. Lessons Learned

Initially we were using the code provided with the bot. So we gave a good chunk of time to understand that code only to find that that code is configured to work in fields which is totally different from our application. The code had a lot of hard-coded hacks which were difficult to work with. After struggling with this we decided to instead work with ROS navigation stack. In hindsight, We would prefer to not use any undocumented code. Our project objectives should have been calibrated according to the time we have to complete the project and with amount of field testing required to make stuff work. At times we took a black box approach to solve certain issues which lead to a waste of time. In hindsight we could have studied the ROS packages we were using before trying to tune the parameters.

If you had to do the project again, what would you change? We would prefer building our robot or at-least testing the robot before finalizing the project plan. We would spend some time on simulation to tune the navigation parameters instead of directly tuning everything in field.

VI. Conclusions

We were able to achieve our goal of developing software for autonomously navigating a wheeled mobile robot while accomplishing obstacle avoidance. The system works with static and dynamic obstacles. We have tested the system outside wean hall and tried multiple times to suddenly walk in front of the bot. It stooped every single time and replanned its path around the obstacle.

One of the possible improvements could be integrating the road segmentation as part of costmap generation to ensure the bot stays within the road boundaries.

APPENDIX

Code is available for public viewing at https://github.com/tartanBot.

Video is available at https://www.youtube.com/watch?v=EVAJZjyaTGo.

ACKNOWLEDGMENT

We would like to thank Tim Mueller-Sim for his assistance in helping us get set up with the Robotanist and answering our questions about how the system works as we continue to develop our software.

References

[1] T. Mueller-Sim, "Development of a ground-based robot for high-throughput plant phenotyping," Master's thesis, Carnegie Mellon University, Pittsburgh, PA, August 2017.