Harikrishnan Suresh

Team C: Fly Sense
Teammates: Shivang Baveja, Nicholas Crispie, Joao Fonseca, Sai Nihar Tadichetty

ILR05
Nov 10, 2017

Individual Progress

For the mapping component of our aerial subsystem, I decided to use the octomap package to generate the 2D obstacle maps, which would then be sent as a stream of images to the AR interface. Considering the limitations of the octomap package when fed with dense point cloud data, I built a pipeline in our flysense_mapping package which would take the raw point cloud data and convert it into a form usable for the octomap package.
The rqt_graph for the flysense_mapping package is shown in Figure 1. For now, it receives the point cloud and tf data from a bag file and processes it. This is then fed into the velodyne_height_map package that outputs only the obstacles in point cloud format. This goes into the octomap package to get the 2D occupancy grid map. The bag file used is from the test conducted outside NSH with our benchtop hardware (DJI matrice and onboard components mounted on a cart).

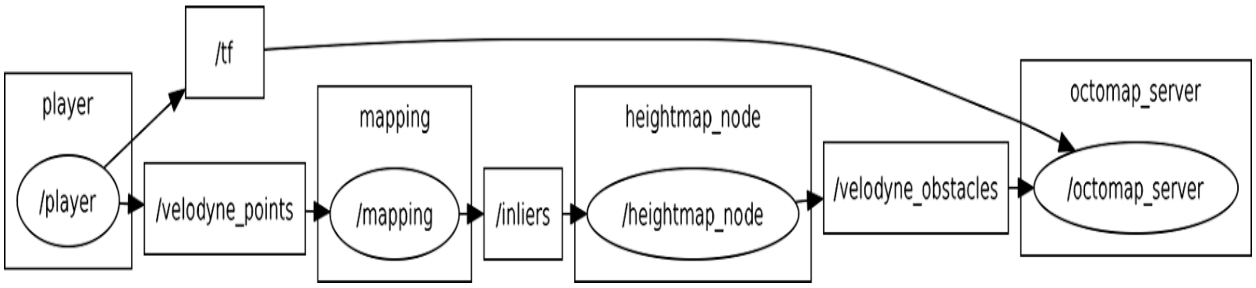

Figure 1: rqt_graph of flysense_mapping package

Expanding the mapping node from the above rqt_graph, we have the pipeline for processing the raw point cloud. The pipeline is described below:

1) Down sampling the point cloud using a voxel grid filter
The algorithm works by dividing the space around into a set of 3D voxels (small boxes), and all the point clouds inside a voxel are replaced by a single point at the centroid of the voxel. The leaf size of the voxel is now set to 10 cm, but can be modified from the launch file.

2) Removing noise using a Statistical Outlier Removal filter
The sparse outlier removal is based on the computation of the distribution of point to neighbors' distances in the input dataset. For each point, the number of neighbors to be considered can be specified and the algorithm computes the mean distance from it to all its neighbors. All the points that lie outside a specified scale of standard deviation from the mean distance are considered as outliers and removed from the dataset. For now, a neighbor threshold of 100 and standard deviation of 1 is considered, but the same can be modified from the launch file.

The output of the pipeline is then fed to the velodyne_height_map package. This package extracts only the point clouds that constitute the obstacles by using a height map algorithm. The minimum height of point cloud to be considered as an obstacle can be changed, which helps to remove all the random point clouds on the ground coming from the velodyne. This eliminates the need to use a separate pass through filter for the point cloud. This algorithm will produce the ideal input for the octomap package as only the relevant point cloud data is obtained. In addition, the input data not being dense improves the dynamic mapping capability of octomap.

Figure 2 shows the output at the 2 stages of the pipeline and velodyne_height_map as seen in Rviz.

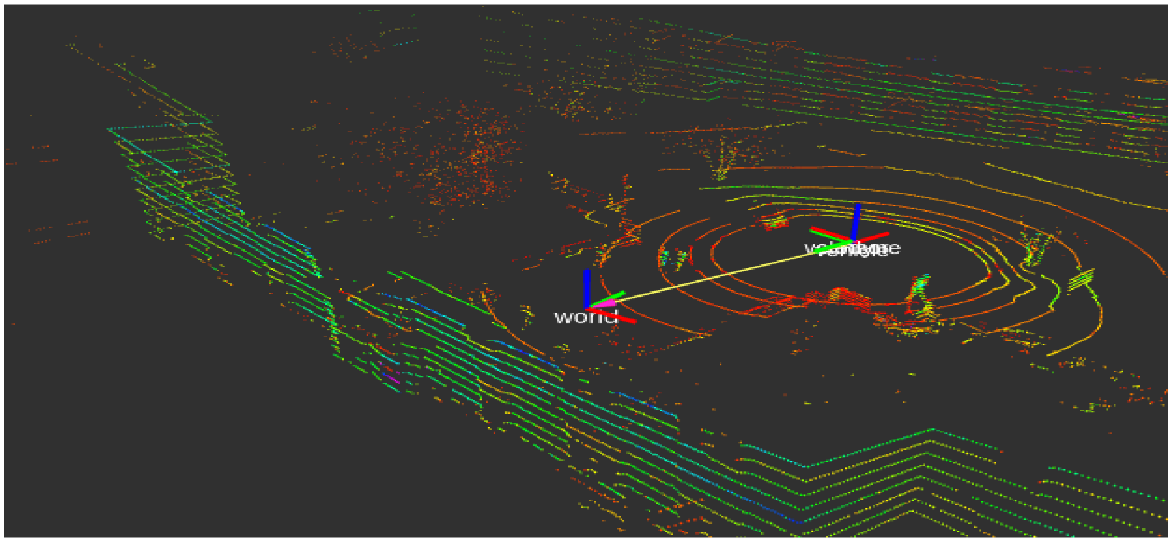

Figure 2a: Raw point cloud

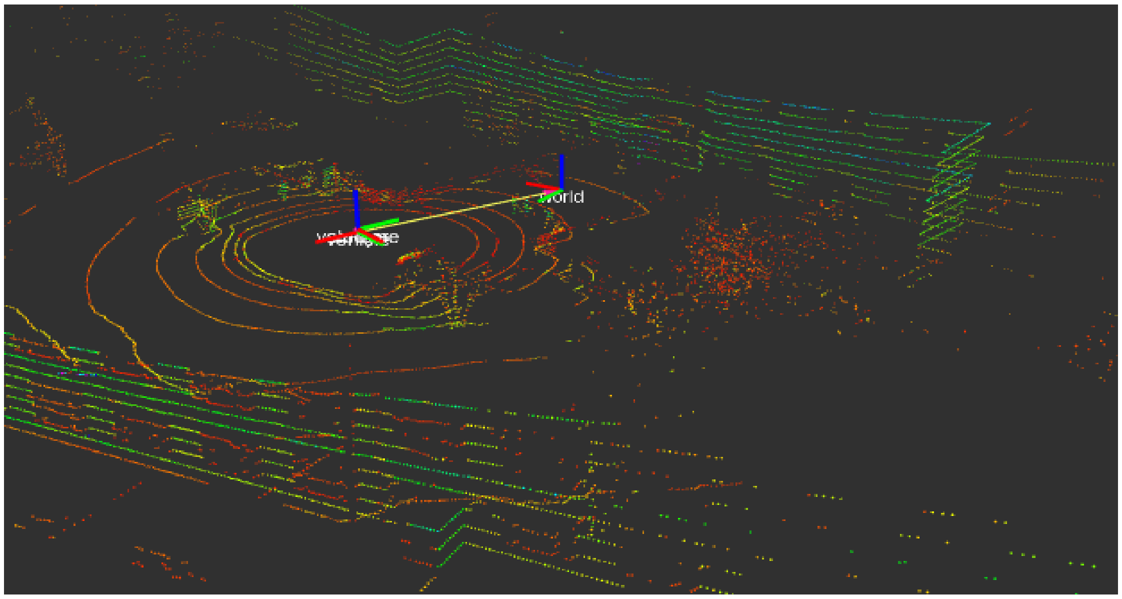

Figure 2b: After down sampling

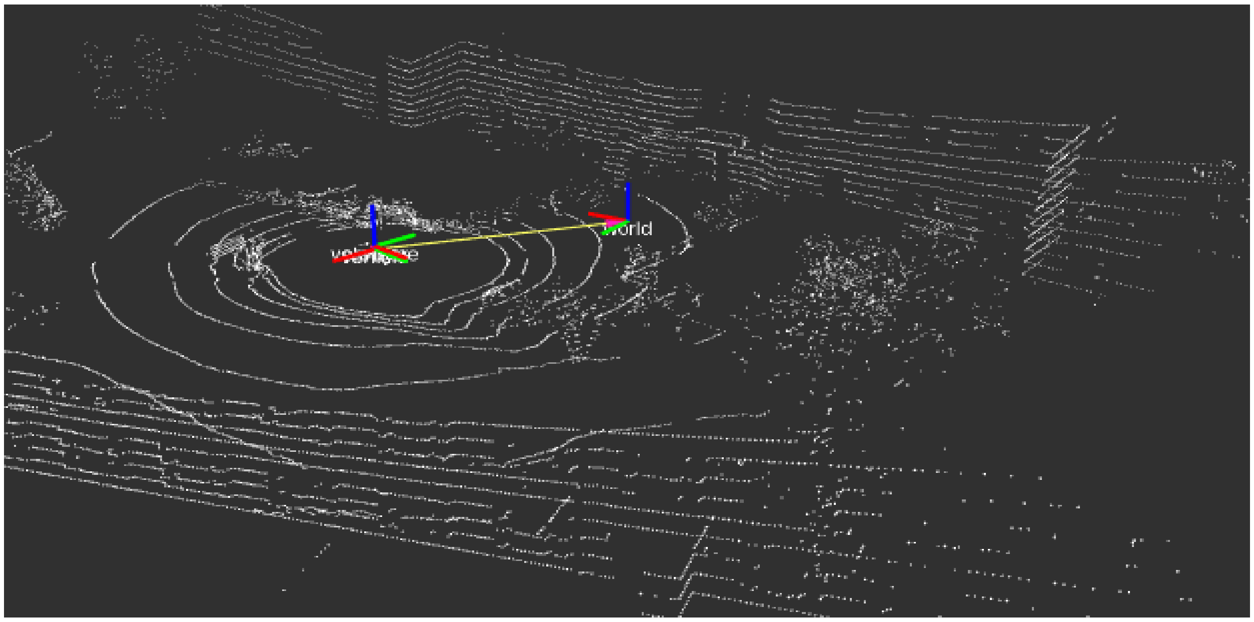

Figure 2c: After removing outliers

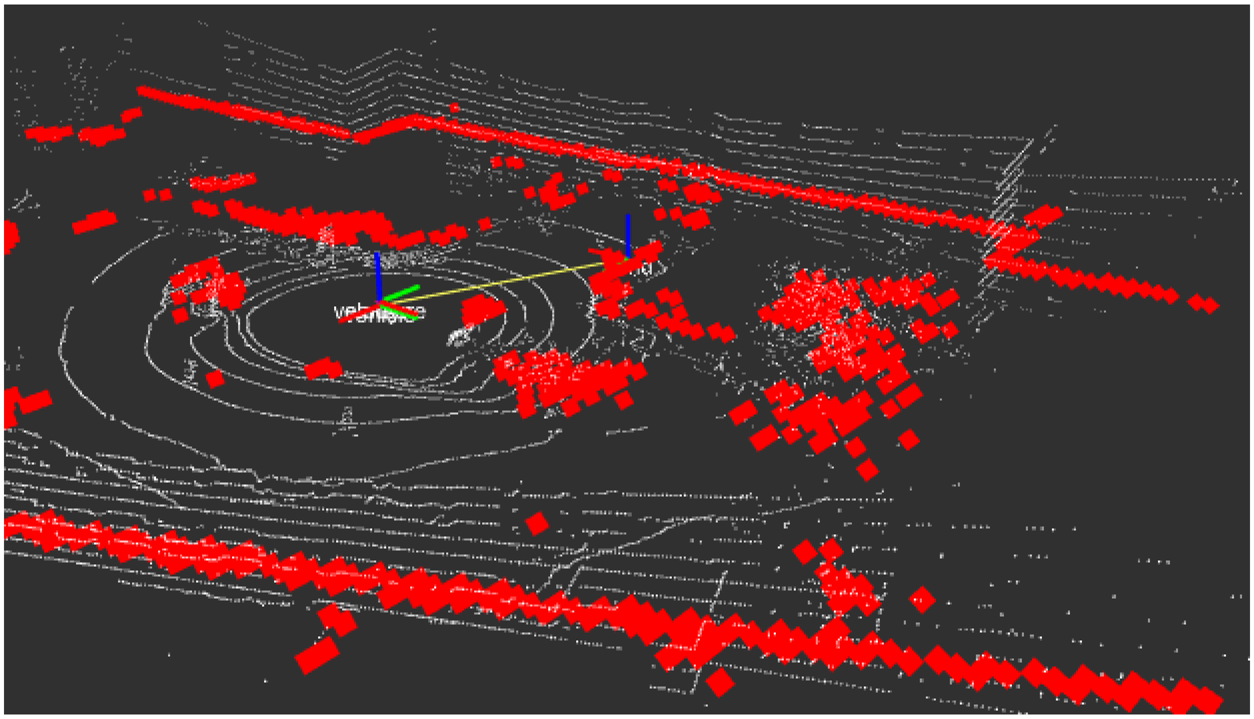

Figure 2d: After velodyne_height_map

The issue however is the underlying concept of height map where the obstacles are projected to a 2D plane. For a flying vehicle, the obstacles that are below will be projected right in front of the vehicle. So, even if the quad is flying straight and does not have to bother about this obstacle below, our flysense system will show it as a dangerous obstacle in front. This calls for a more detailed analysis of the same algorithm. However, for our FVE experiment with benchtop hardware, this will not cause any problems as all the obstacles are in the relatively in the same plane as the cart and do not appear below the cart. Hence, we might use the algorithm and define our operating conditions accordingly.

The 2D occupancy grid map produced by the octomap package is shown in Figure 3 for the two tests we conducted- static LIDAR test outside MRSD lab and moving LIDAR test outside NSH.

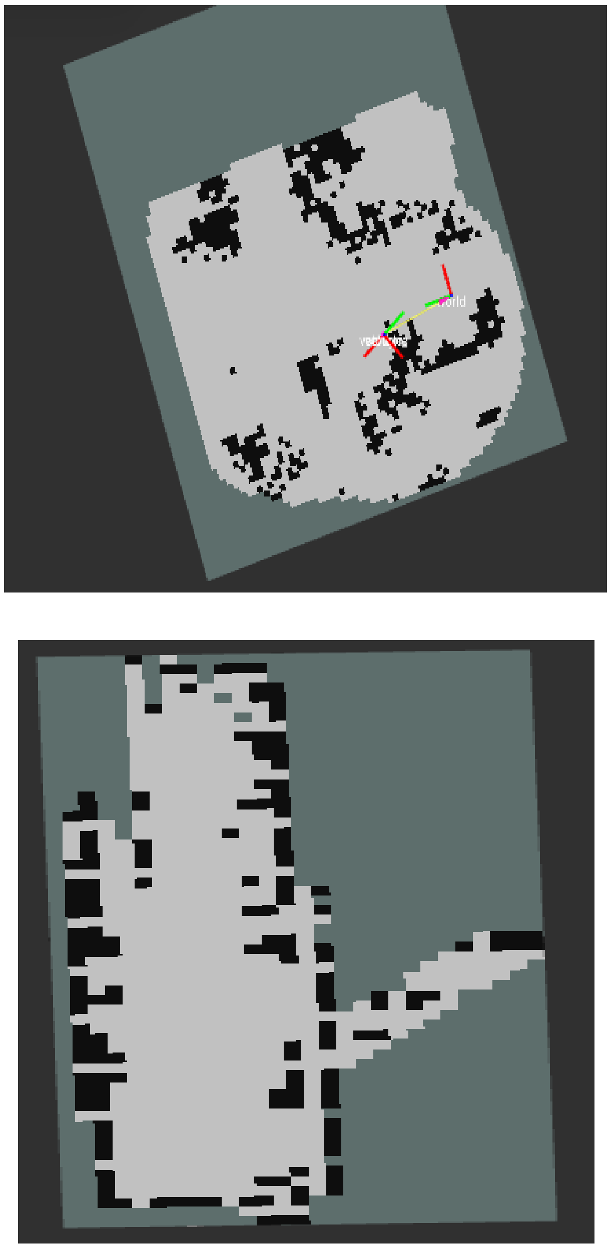

Figure 3: 2D occupancy grid map. left: test outside NSH. right: Static LIDAR test

I also worked on a node that converts the point cloud output of the pipeline into a set of clusters representing the obstacles. For this, the Euclidean cluster extraction code from point cloud library was used. The algorithm first performs a planar model segmentation to find the largest plane model in the dataset, and uses a Kd-Tree data structure to generate the clusters at every instant. The idea was to collect all the point clouds representing the obstacles and determine one optimal point in each cluster (preferably centroid) to calculate the distance and time to impact. However, the output was not satisfactory, and the algorithm must be tuned significantly. The clusters were not generated correctly, and the plane model segmentation failed. This algorithm seems to be a good replacement for the velodyne_height_map especially in the scenarios mentioned above, and will be considered in the spring semester.

In addition to the mapping work, I also worked on an NEA quad dataset that we were provided with. I developed a node that subscribes to the odometry information and publishes the tf information. The quadcopter's path as seen in Rviz in shown in Figure 4. As explained by our mentor at NEA, the quad moves towards a region surrounded by crates and lands in the free space in between. The scenario is ideal to test our complete system, and we are planning to use it for FVE if we solve a few issues associated with the dataset (mentioned in the challenges).

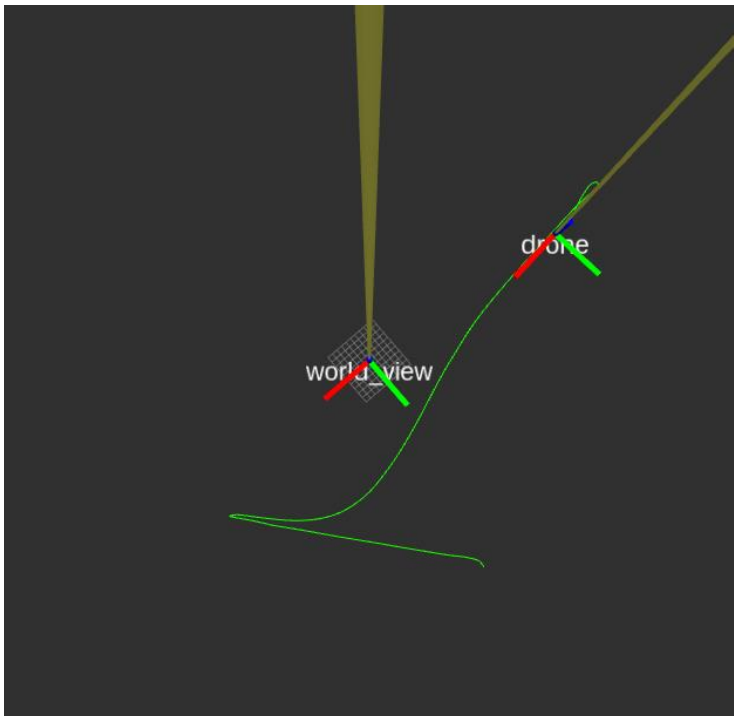

Figure 4: Path of the drone seen in world_view frame

Challenges faced

1. The NEA quad data set contains point cloud in the form of scan lines and not points. It does not work with the pipeline or the octomap package. Some kind of conversion needs to be done. We might experiment with the Air Lab code as it takes in scan lines from hokuyo to build map, or we might test our system with a helicopter data set from NEA we received first.

2. The point cloud data from our test outside NSH was very jerky, and too noisy to build a map. The data was very different from the data sets we received from NEA which had registered point cloud. However, it was found that the velodyne manual provided wrong information about the orientation of the axis. The velodyne was mounted in the wrong way on top of the quad. Applying a static tf with 90deg offset made the data a lot smoother.

3. The point cloud data is still not as smooth as the NEA datasets we received, and we will have to add a point cloud registration node at the beginning of the pipeline.

Team work

I collaborated with Shivang and Nihar to conduct tests outside NSH (our FVE test location) and collect data in bag files. This will now be used to integrate all our subsystems together. I also coordinated with Shivang to integrate the flysense_mapping package with the flysense_onboard package into one flysense_sensing stack.
A picture of our benchtop hardware captured during our test outside NSH is shown in Figure 5.

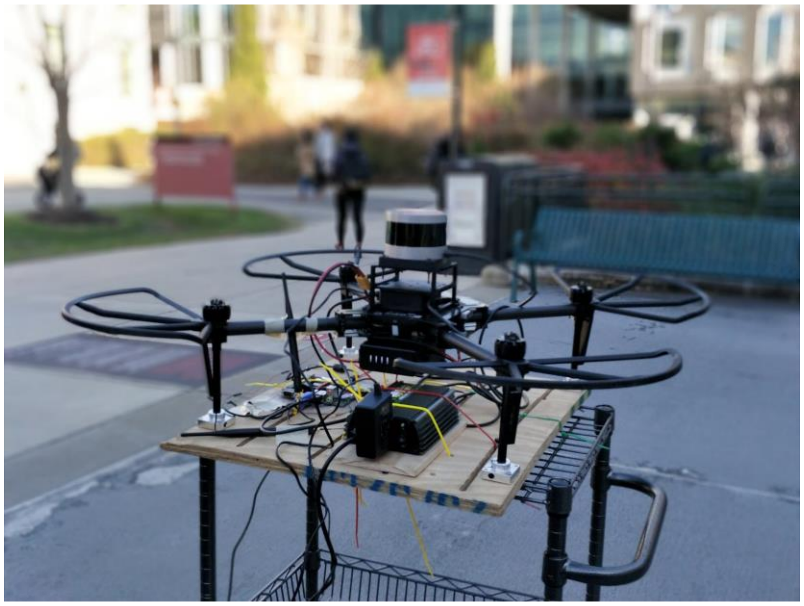

Figure 5: Benchtop hardware

Due to the extent of work left in the mapping portion, the work was divided between myself, Nick and Shivang - Shivang handling the preprocessing node and Nick handling the postprocessing node.

Shivang implemented the dynamic window code developed by Joao in ROS and integrated it with the crop box filter. He also added a new crop box filter to remove the cart and the person pushing the cart from the point cloud stream. Shivang also worked on integrating the onboard computer with the DJI matrice and the AR headset. He set up a Wi-Fi network to communicate with the AR, and to enable us to work remotely with the onboard computer.

Nick used the 2D occupancy grid which was generated for the static LIDAR test, and developed a node to publish it as a stream of images. These images will then be sent to the AR. Nick assembled all the sensing components on the DJI matrice and mounted it on top of the cart. Nick also assembled the Power Distribution Board, and is trying to rectify some of the issues in the PDB.

Joao worked on the sound warning and voice commands in the user subsystem. He developed a code in android to implement sound warnings (with varying frequency based on obstacle location). He implemented voice commands using Google API and did reliability tests for the same. He completed an algorithm to classify obstacles based on time to impact and determine if its left or right.

Nihar worked with Shivang on updating the ROS interface in the AR and configuring it to communicate reliably with the onboard computer over WIFI network. He also worked on sorting out issues with the Image reception. Since the voice commands implemented in Google API could not be integrated with the android platform in the AR headset, he implemented voice commands using Pocket Sphinx and integrated it. He also integrated the audio warnings with the AR headset.

Future work

As we are approaching the Fall Validation Experiment, we are now focusing on integrating various subsystems together. We also plan to do more tests outside NSH to validate all the components of our system.

In the sensing subsystem, the point cloud registration and real-time bird’s eye view image generation must be completed. The algorithms running on the mapping pipeline must be optimized after analyzing its performance on various data sets. This entire subsystem has to be integrated with the user subsystem, with sound warnings generated based on the obstacle location and the bird’s eye view printed seamlessly on the AR interface.

I will be focusing on the point cloud registration first. I will also be working on a node that subscribes to the 2D occupancy grid, and determines the location of the vehicle and the obstacle in the grid at every instant. These grid coordinates will be sent to the post processing node that generates bird’s eye view images based on the dynamic window. I will also work on optimization of all the codes in the mapping pipeline.