Table of Contents

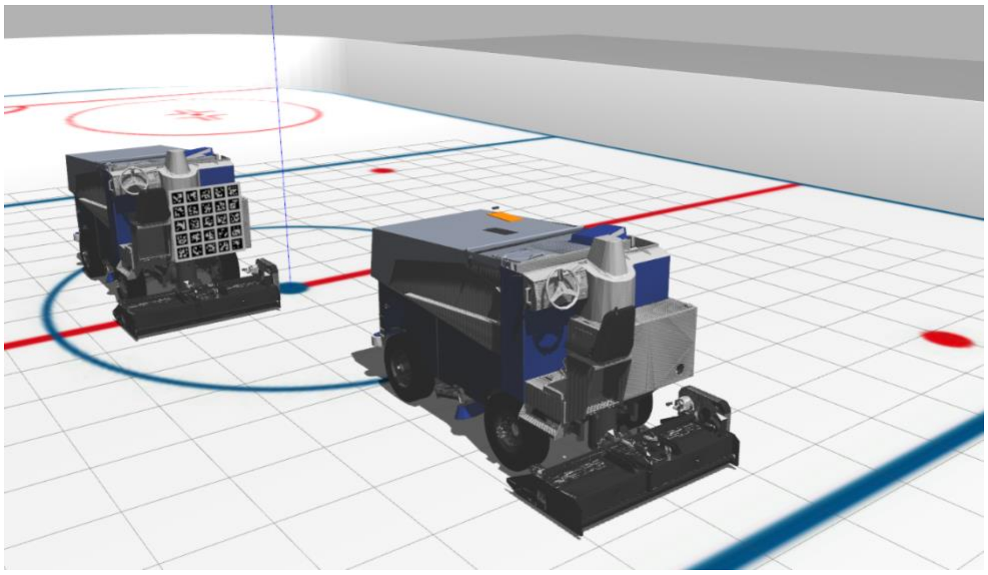
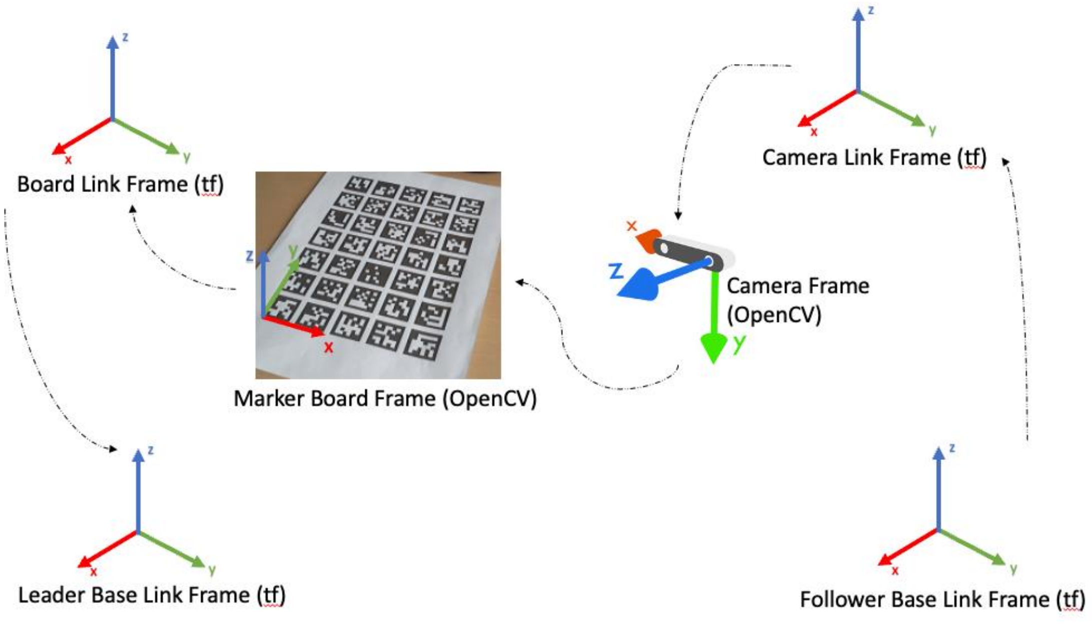


Figure 1. Leader and Follower are Launched in Gazebo

Individual Progress

During the past few weeks, I focused on solving the errors in the transform between board frame and camera frame returned by my pose estimation algorithm. The accuracy of this transform is crucial because all of the following subsystems, such as velocity estimation and waypoint generation (based on leader's position), depend on it. In addition, I calibrated the IMU in RealSense D435i in reality, making sure it is equivalent to the Gazebo IMU plugin currently used by the localization subsystem.



Board Link Frame (tf)
Marker Board Frame (OpenCV)
Camera Link Frame (tf)
Camera Frame (OpenCV)
Leader Base Link Frame (tf)
Follower Base Link Frame (tf)
x
y
z

Figure 2. Coordinate Systems (Right-Handed)

Improving Accuracy of Leader Pose Estimation

To avoid the snowball effect when error accumulate along the system architecture, the pose estimation of the leader must be accurate enough so that there will be more leniency for the following subsystems that depend on the output of it, such as leader velocity estimation and waypoint generation. Therefore, when we found the pose returned by my algorithm was not satisfactory enough, I tried the following measures to debug its inaccuracy:

1. I started with reading multiple documentations to confirm the unconventional coordinates used by OpenCV for the camera and the board. This was important when we tried to chain the transform from the follower base to the leader base. The coordinate frames of different links are drawn in Figure 2. If the position of the leader relative to the follower is defined as "front", all of the links in ROS tf tree (as consistently defined by our URDFs) have x facing forward, y facing left and z facing upward. These include follower base link, camera link, board link, and leader base link. The camera frame

used in OpenCV has x facing right, y facing down and z facing front. The marker board used in OpenCV has the origin of its frame at the bottom left corner of the board, with x facing right, y facing up, and z facing out of the board. The correct chain of transforms to estimate the leader position based on the follower position (obtained from localization subsystem) and the camera-board transform (obtained from perception subsystem) is as illustrated by the arrows.

2. I tested the same algorithm using the camera to estimate an ArUco board in reality to be sure if the error is caused by the sensor plugin in simulation. The accuracy was satisfactory in reality but it was not convincing because the ArUco board was printed on an A4 paper, which made it too small to be recorgnized when placing it 2 meters away the camera. This was far from representative of the simulation, so I printed a marker board that is 42 inch by 42 inch with the help of SCS Poster Printing. And by next PR, we will use this to test the accuracy of the pose estimation algorithm by placing it in front of the camera with a distance that mimics a leader-follower setting.
3. I published the tf between the camera and the board using the estimated pose, so I could visualize it in RViz and compared it with the RobotModel. The result was very different from the ground truth, which was supposed to be at the bottom left corner of the marker board.
4. I assumed the estimated translation vector was scaled incorrectly from pixels to meters, so I normalized the translation vector, and scaled it using the interpolated average depth value (explained in the previous ILR, which is essentially a mask over the depth image to get the average depth in the mask, where the mask is generated by interpolating detected markers in four corners in the RGB image). This did not solve the error either.

We thought the error was caused by some unknown issues of the RealSense Gazebo plugin we used, so I decided to include *Velodyne* Puck, VLP-16, into simulation to estimate the translation vector instead (explained in next section). However, on the day before this PR, I found we made a mistake when interpreting the transform between the board and the camera. The OpenCV documentation for ArUco stated that the returned pose is a transform from the board frame to the camera frame, which is equivalent to the board's position in the camera frame. We interpreted it as the other way around, the camera's position in the board frame. Therefore the transform drawn in RViz was never close to the bottom left of the board no matter what we tried. After we corrected this, the tf of the estimated board frame was exactly drawn at the bottom left of the board in RViz, as shown in Figure 3. The top right image is the axis drawn using the drawAxis function in OpenCV, which directly takes the estimated pose and the camera intrinsics/distortions as input. The tf drawn in RViz is after applying the chain of transforms starting from the follower base link as explained before.

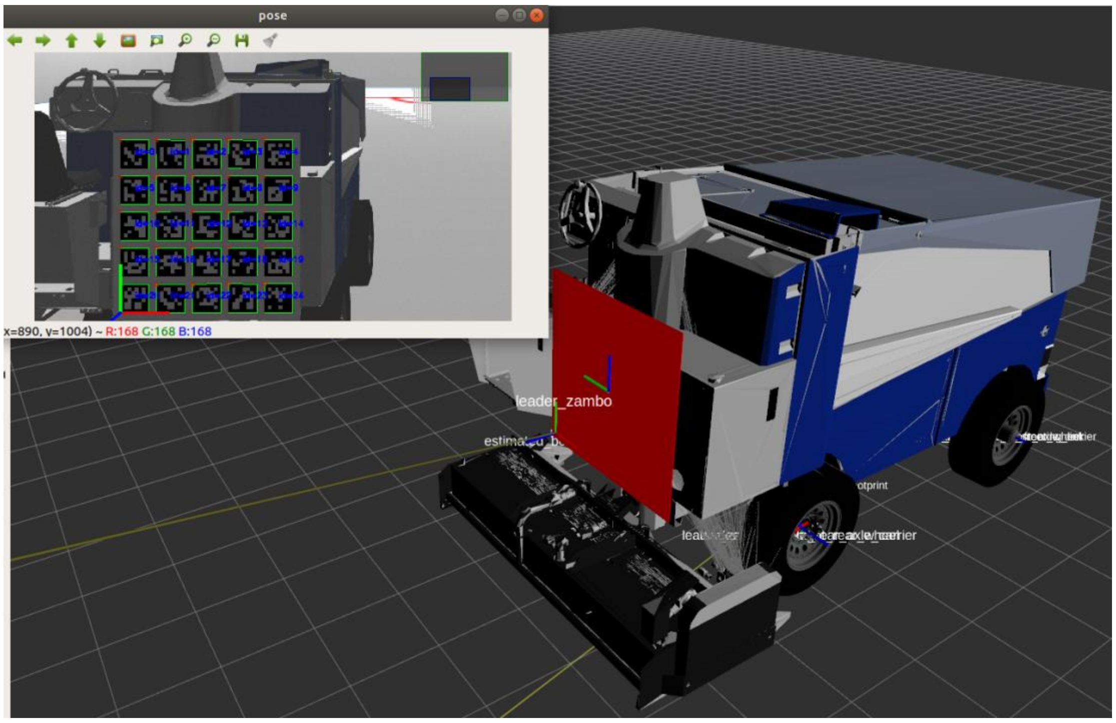


Figure 3. Estimated Marker Board Frame

Pose Estimation Enhancement using Velodyne Puck

The initial motivation to use Puck in our project was due to the translation errors, but it turned out to be a mistake on our end when calculating the transform between the camera and the marker board. The other motivation to use Puck is because when the leader is taking a turn, the marker board disappears from the FoV of the camera on the follower. This can be slightly mitigated by placing the camera to the rear of the follower but the trade-off will be half of the image is occluded by the snow tank at the front of the follower. Using a 360-degree LiDAR will solve this problem because it will continually output point cloud of the leader even if it's making a U-turn.

Our team will use Velodyne's Puck LiDAR sensor, VLP-16, in our project, so I started off by adding the Puck into our simulation environment. I used Velodyne Simulator package to insert a Gazebo plugin of Puck into the follower's URDF, which publishes ROS PointCloud2 message with the same structure (x, y, z, intensity, ring, time) as the real sensor. To manipulate the point cloud in preparation for leader pose estimation, I made a pipeline as follows:

1. Conversion: The ROS PointCloud2 message is first converted to a Point Cloud Library (PCL) XYZRGB format, with the “ring” field converted to a corresponding RGB value.
2. Filtering: The point cloud is downsampled using voxel grid filter in PCL. Then it will be filtered based on three different axes using the passthrough filter in PCL. In our case, I filter it along z from $-h$ to 0 where h is the height of the Puck from the ground, along x from -5 to 10 which approximates the farthest longitudinal distance between the leader and the sensor, and along y from -5 to 5 which approximates the farthest lateral distance.
3. Plane Segmentation (Optional): if the filtering along z still gives a lot of points on the ground, then the point cloud is segmented using a RANSAC plane segmenter in PCL. The inliers will be the plane while the outliers will be the follower, the leader, and any random obstacles.
4. Clustering: The PCL XYZRGB point cloud is converted to a XYZ point cloud first because Euclidean clustering does not need RGB information. Then an Euclidean clustering is applied to the points, which in the backend uses a Kd-tree to find the nearest neighbors (further details are well explained here). After getting a list of clusters, I assign a color to each cluster of points.
5. Conversion: The downsampled, filtered, and clustered point cloud is converted back to a ROS PointCloud2 message which includes the corresponding cluster color as a PointField. The RViz can now show the clustered data with colors.

The current progress has been summarized in Figure 4. The left image shows the original point cloud published by the Puck, while the right image shows the point cloud after processing, where there’re only two cluster of points remaining. One (in yellow) belongs to the leader and the other (in cyan) belongs to the follower.

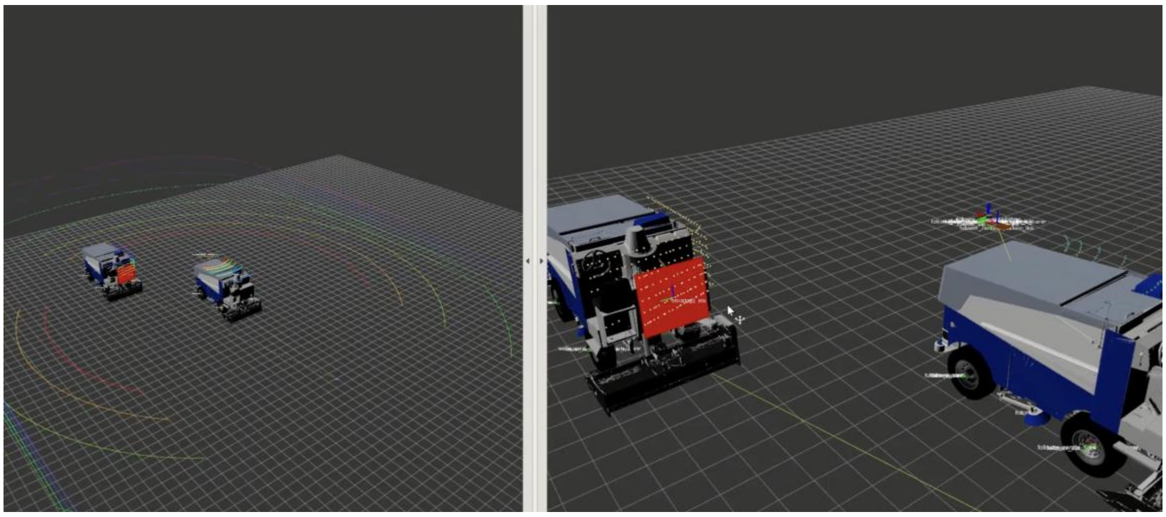


Figure 4. Point cloud data before and after processing

IMU Calibration

I, together with Rathin, calibrated the IMU on RealSense D435i by following the steps in the manual.

Challenges

The major challenges I have encountered when implementing the tasks above include:

1. Figuring out the correct transform between elements
   As mentioned in the previous section, the conventions for the coordinate system are different in ROS tf tree and in OpenCV. Therefore, to make sure we get the correct chain of transforms from the follower base to the leader base, we need to figure out the coordinate systems of each link defined in tf, as well as the coordinate systems of the camera and the marker board used in OpenCV. Most importantly, the pose returned by the ArUco library is the transform from the board frame to the camera frame, which is the board's position viewed in the camera frame. The mistake we made on this led to hours spent on debugging the algorithms which are actually correct.
2. Coarse collision box of the leader causes inaccurate point cloud
   The size of the mesh file for the leader is extremely large, which makes it impossible to be loaded as a collision geometry in Gazebo. Hence, we use a single box geometry as both ice resurfacers' collision element. This will not cause any problem in most scenarios, but when we add the LiDAR that essentially shoots lasers and receives them after they bounce off a "collidable" surface, the problem occurs. The point cloud around the leader becomes a single cuboid which doesn't tell the actual translation between the Puck and the marker board. The solution is to manually add more collision boxes in the URDF. However, note that there will be no such a problem when testing everything in real life because we no longer to need to approximate the collision box.
3. PCL requires a steep learning curve
   This starts off by setting up the correct PCL library for a specific Python and Ubuntu version. Since there's no active maintainer of the Python binding to PCL, different approaches have to be used to install python-pcl in different environments. To install a working PCL library in my Python 2.7 in Melodic, I went through dozens of posts and issues in the GitHub community with trial and error. In addition, because the visualization tool kit (VTK) in my Ubuntu is inconsistent with the PCL installed, I can't visualize PCL data using the

PCL visualization library. The solution is to convert the point cloud to a ROS PointCloud2 message and visualize it in RViz when debugging the algorithm.

Teamwork

- Nick continued to work on the hardware. He installed the encoder onto the RC car and tuned the wheels so that the car can move straight. He also completed waypoint testing with Rathin
- Rathin spent a lot of efforts in the PDS assignment which will be used for the RC car power distribution. He validated his Stanley controller on the RC car.
- Yilin managed to control two Zamboni ice resurfacers in Gazebo independently with keyboard. He also developed wheel odometry and fused it with IMU accordingly, all in simulation.
- Jiayi managed to visualize both paths of the leader and the follower in RViz (the former is the ground truth while the latter is calculated based on leader's pose and velocity). She also got familiar with the teb_local_planner and move_base packages in ROS navigation stack.

Plans

Before next progress review, I plan to solve the issue that the marker board disappears from the camera FoV when the leader is taking a turn. I can think of two ways approaching this: (1) continue to use the LiDAR and find the alignment pose of a predefined leader point cloud in the scene, which can be accomplished by the PCL library shown here (this requires the calibration between camera and lidar, which is provided by the ROS package lidar_camera_calibration) (2) add another RealSense D435i and form a "Λ" with the current one so that both fields of view can add up to 174° (this also requires the calibration between two cameras, which is instructed in this article).

In addition, I will work together with the team to set up Jetson, validating all the connections between Jetson and sensors, and the ones between Jetson and Arduino.