Inter IIT Tech-Meet 10.0

DRDO'S UAV-Guided UGV Navigation Challenge

Project Documentation

Team Number 10

22 March 2022

Contents

1 Installation and Setup Instructions

1.1 Pre-Installation Assumptions

The following instructions assume that the system has a working installation of Ubuntu 18.04 with ROS Melodic and Gazebo 9. Further, the Ardupilot Stack and the ardupilot_gazebo have also been properly installed. We will be using the **catkin build** system instead of catkin_make. The following workspace structure was maintained during development:
Instructions followed for installing Ardupilot Stack and ardupilot_gazebo plugins are reproduced below for verification:

```
$ cd $HOME
$ git clone https://github.com/Ardupilot/ardupilot
$ cd ardupilot
$ git submodule update --init --recursive
```

Next, we shift to the recommended firmware branch - 3.6 and build it. Within the ardupilot directory,

```
$ source ~/.bashrc
$ git checkout Copter-3.6
$ git submodule sync
$ git submodule update --init --recursive
$ ./waf configure --board px4-v3
$ ./waf copter
```

Next, we install the ardupilot_gazebo package as follows:

```
$ git clone https://github.com/khancyr/ardupilot_gazebo.git
$ cd ardupilot_gazebo
$ git checkout dev
$ mkdir build
$ cd build
$ cmake ..
$ make -j4
$ sudo make install
```

If mavros is not installed, it can be installed as follows:

```
$ sudo apt-get install ros-melodic-mavros
$ ros-melodic-mavros-extras
```

```
$ cd /opt/ros/melodic/lib/mavros
$ ./install_geographiclib_datasets.sh
```

1.2 Installing Dependencies

First, install the following additional system dependencies:

```
$ sudo apt-get install python-wstool python-catkin-tools \
$ ros-melodic-cmake-modules protobuf-compiler autoconf \
$ libboost-dev libeigen3-dev libgoogle-glog-dev
```

Next, we create a new catkin workspace, if not already done:

```
$ mkdir -p ~/drdo22_ws/src
$ cd ~/drdo22_ws
$ catkin init
$ catkin config --extend /opt/ros/melodic
$ catkin config --cmake-args -DCMAKE_BUILD_TYPE=Release
```

Note: We installed the Ardupilot stack and plugins in home, however this is not necessary and will not affect the working of the submission (provided the paths of the installation are properly set where required) since these are not built with catkin.

Place the provided package folder (i.e. the submission) into the src folder and build after changing the appropriate paths in the launch files for ardupilot and ardupilot_gazebo.

```
$ cd ~/drdo22_ws
$ cd src
$ git clone <repo_link>
```

The submission depends on the following ROS packages. Use the following command to ensure all of them are installed. Most of these will already be installed, however all packages have been mentioned here just to be sure.

```
$ sudo apt-get install ros-melodic-std-msgs ros-melodic-std-srvs \
ros-melodic-geometry-msgs ros-melodic-mavros-msgs ros-melodic-nav-msgs \
ros-melodic-message-generation ros-melodic-message-runtime \
ros-melodic-tf2-ros ros-melodic-cv-bridge ros-melodic-tf \
ros-melodic-tf-conversions ros-melodic-eigen-conversions \
```

```
ros-melodic-sensor-msgs ros-melodic-image-transport \
ros-melodic-cv-bridge ros-melodic-visualization-msgs \
ros-melodic-pcl-ros ros-melodic-pcl-conversions \
ros-melodic-interactive-markers ros-melodic-trajectory-msgs
```

Once all dependencies have been satisifed, build the workspace:

```
cd ~/drdo22_ws/src
catkin build
```

1.3 Running the Simulation

Please refer to the README for the same.

1.4 Version Information for Dependencies

ROS Version: melodic 1.14.10

Gazebo Version: 9.16.0

Pip Version : 20.3.4

Pip3 Version : 21.0.1

Kernel Version: 5.4.0-67-generic

Commit IDs for git installed dependencies: ArduPilot : Copter-3.6
All other repositories use the **main** or **master** branch.

Versions for additional system installed dependencies:

python-wstool	: 0.1.17-1
python-catkin-tools	: 0.6.1-1
ros-melodic-cmake-modules	: 0.4.2-0bionic.20201015.021712
protobuf-compiler	: 3.0.0-9.1ubuntu1
autoconf	: 2.69-11
libboost-dev:amd64	: 1.65.1.0ubuntu1
libeigen3-dev	: 3.3.4-4
libgoogle-glog-dev	: 0.3.5-1

The following dependency versions are the default versions.
Version for ROS binary installed dependencies:

ros-melodic-mavros-msgs	: 0.5.12-0bionic.20201017.034204
ros-melodic-std-srvs	: 1.11.2-0bionic.20201017.034224
ros-melodic-geometry-msgs	: 1.12.8-1bionic.20210112.173042
ros-melodic-nav-msgs	: 1.12.8-1bionic.20210112.173737
ros-melodic-message-generation	: 0.4.1-1bionic.20201017.033327
ros-melodic-message-runtime	: 0.4.12-0bionic.20201017.033232
ros-melodic-tf2-ros	: 0.6.5-0bionic.20210112.183245
ros-melodic-cv-bridge	: 1.13.0-0bionic.20210112.181516
ros-melodic-tf	: 1.12.1-1bionic.20210112.183814
ros-melodic-tf-conversions	: 1.12.1-1bionic.20210112.190235
ros-melodic-eigen-conversions	: 1.12.1-1bionic.20210112.180039
ros-melodic-sensor-msgs	: 1.12.8-1bionic.20210112.173755
ros-melodic-image-transport	: 1.11.13-0bionic.20210112.181401
ros-melodic-visualization-msgs	: 1.12.8-1bionic.20210112.173714
ros-melodic-pcl-ros	: 1.7.1-1bionic.20210112.185053
ros-melodic-pcl-conversions	: 1.7.1-1bionic.20210112.181845
ros-melodic-interactive-markers	: 1.11.5-1bionic.20210112.190840
ros-melodic-trajectory-msgs	: 1.12.8-1bionic.20210112.180155

1.5 Changes in provided Model Files

Following changes in the file gimbal_small_2d/model.sdf:

- line 155: **update_rate** tag of depth_camera sensor changed from **10.0 to 20.0**
- line 149: **clip_far** tag changed from **10 to 20** : As the maximum height above the road plane that we can fly our drone with is 20m.
- line 164: **pointCloudCutoffMax** tag under depth_camera sensor changed from **10 to 20** : same reason as above

1.6 Potential Build and Runtime Issues

- No downward camera topic Try deleting gimbal_small_2d model in ∼/.gazebo/models
- On running road_seg_node of the segmentation package, on the first run it gives the error - segmentation fault (core dumped). The possible reason might be that on starting there is no image (or null image) and its pointer is accessed somewhere without check. But once the drone attains certain height and properly detects the image, on second run road_seg_node works fine.

2 Overall Approach and Algorithm Description

2.1 Autonomous Takeoff

We have utilized **QGC** as our mission planner to arm the iris quadrotor. In the world files given, the quadrotor is spawned on top of the Prius which itself is on a slanted road. Since the drone was found to be sliding over the Prius rather than being static, the in-built safety check for the gyroscope was failing, and the drone would refuse to arm. For takeoff, we can either place the drone on the ground some distance away from the Prius and then further operate or alternatively we can manually disable all the safety checks except *GPS lock* in the mission planner. Either of these two solutions successfully arms the drone.

We arm the drone using the command shown below

```
$ sim_vehicle.py -v ArduCopter -f gazebo-iris
```

2.2 Road Segmentation

We primarily use the pointcloud generated by the RGBD camera for road pixel segmentation. We first use the **PCL library** to find the three most prominent planes in the pointcloud using the **RANSAC** algorithm. The next task will be to identify which of these planes is the road. For this we use some heuristics and criteria which will be detailed below.

2.2.1 Average slope computation

We expect that the road will be a more even and horizontal surface. To ease the computational load of performing this operation, we adopt the following method. From the very large number of points in every plane, we randomly select a fraction $\boldsymbol{f}$ of the points. We compute the quantity $\frac{|p_{2z} - p_{1z}|}{\sqrt{(p_{2x} - p_{1x})^2 + (p_{2y} - p_{1y})^2}}$ for every two points p_1 and p_2 from the selected points. This value is now averaged over every two pairs from the selected points and this computed average will be called the **Z-value** from now on. If the road is reasonably flat, then the numerator is expected to be small. Since the points are taken randomly, we get a set that can reasonably be expected to cover all parts of the plane.
We compute this quantity for all the three planes and sort them based on the same.

2.2.2 Inclusion of time based filters

A major challenge in the map is the **similarity** in some places of the road to the surrounding mountains. An example would be the RGB image:

Figure (a)

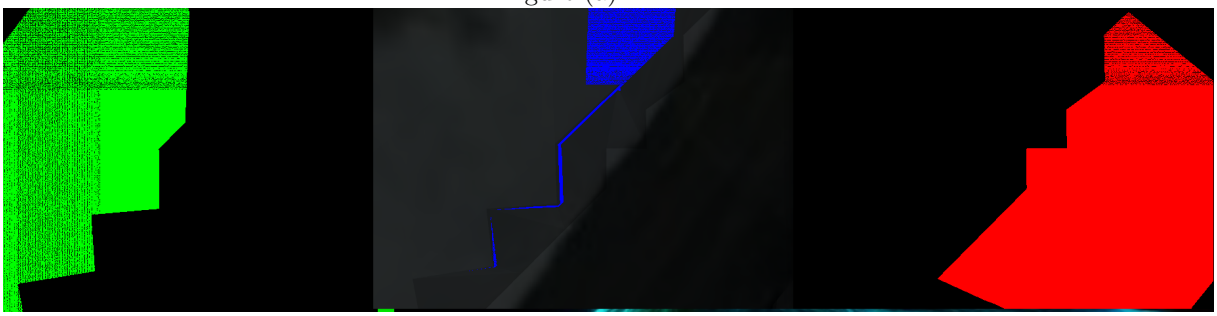


Figure (b)

Figures (a), (b) : Three major planes shown as described in algorithm. Red plane is the final segmented road.

In such a case, the **Z-value** for these two planes will be very similar and not enough to reliably catch the road correctly. Thus we introduced **time-based filtering** to solve this problem. The idea is that the depth of the road plane will not change by a large amount in one timestep. To apply this method we need to know the **average depth** of the planes and a stable and reliable estimate of the previous depth values.

2.2.3 Estimating the previous depth prevZ reliably

We follow a very similar method for the computation of the **Z-value**. The only difference here is that the average is taken of the depth values of the points, which are available from the depth camera. A simple way to compute **prevZ** is to just pick this average depth and use it in the next timestep. However this idea is unreliable. The reason is that if our algorithm picks the wrong plane even once, then *prevZ* value will store its average depth. Thus, the quantity which

we wished to use to stabilise our segmentation will now have the opposite effect.

A better way to compute $prevZ$ is to use a **moving average**. At every timestep, we perform the following update parameterized by γ:

$$prevZ = prevZ \times (1 - \gamma) + (Z - value) \times \gamma$$

Such an update effectively gives us an average of the last few Z-values. Exactly how far back in time does this average go is decided by the parameter γ. The advantages of using this method are multiple:

- The update is cheap. We can take the average as far back in time as we please just by changing γ. There is no need to store the Z-values, thus using lesser space and also being easier to debug and implement.
- When we use this method, one bad segmentation cannot throw off our average. The algorithm can still correctly segment a road in the future timesteps.

2.2.4 Bringing it all together

Here is how we use all the components described above.
First we get the three most dominant planes from the pointcloud. Then we compute the **Z-value** for each of them. Along with this, we count the number of points in each plane. If this number is below a certain threshold then we apply a certain fixed penalty to that plane; i.e. we will add a penalty to that plane's Z-value. The reasoning is that overly small planes are most likely stray planes and not the road.
If the Z-values are sufficiently far apart from each other then we are reasonably certain that the plane with the smallest Z-value is the road. But if that is not the case and the Z-values are very close together, then we use the prevZ value.

Let α be the smallest Z-value obtained. Now we pick the planes whose Z-value is smaller than $\delta \times \alpha$ where δ is a parameter. For these planes we compute the average depth. Finally, we pick the plane whose average depth is the closest to the prevZ value.

Figure : The unrefined algorithm was fluctuating between green part(mountain part) and red part(actual road part). The described refinement solved the ambiguity.

2.3 Mean Path Detection

We primarily used the pointcloud of the main plane detected above and converted it into RGB image for Mean Path Segmentation. We used OpenCV to find the mean path using majorly the following algorithms - MinAreaRect and Principal Component Analysis (PCA). MinAreaRect bounding rectangle is drawn with minimum area, so it considers the rotation also. Principal Component Analysis (PCA) is a statistical procedure that extracts the most important features of a dataset. These two algorithms along with some heuristics are used in concert with each other to determine mean path, the details of which are given below.

2.3.1 Condition when road occupies major part of Image

We converted the above obtained image into a Binary Image(road is in white and the background is black) and then checked whether the road occupies a major part of the image. Let total number of pixels in the image be N. We then count the number of white pixels(N_0) and check if the ratio $\frac{N_0}{N}$ is greater than a set threshold. If it is greater, then the drone is given waypoints to move straight on the road.

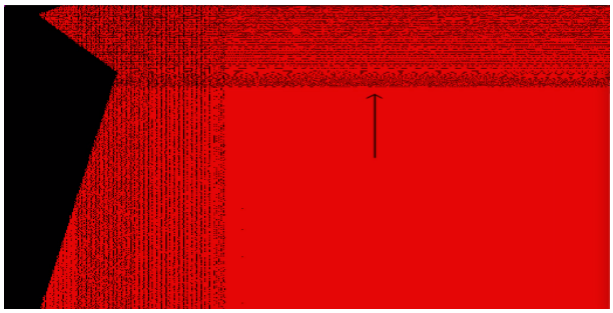


Figure: Arrow shows the direction of movement.

2.3.2 Next Waypoint Calculation using MinAreaRect

If the above condition is not satisfied then we move to our second approach. Here we computed the contours of the above image and selected the contour with the maximum area(**max_area_contour**) to remove any unwanted noise. Further processing is now done on the max_area_contour. We try to fit a rectangle with the minimum area around the above contour to get the orientation of the road using the predefined OpenCV function cv::MinAreaRect(). The **center** of the road is taken as the center of the rectangle. If the ratio $\frac{Rectangle\_Area}{max\_contour\_Area}$ is greater than a given threshold, we consider the direction calculated by the minAreaRect which is the forward direction in the image.

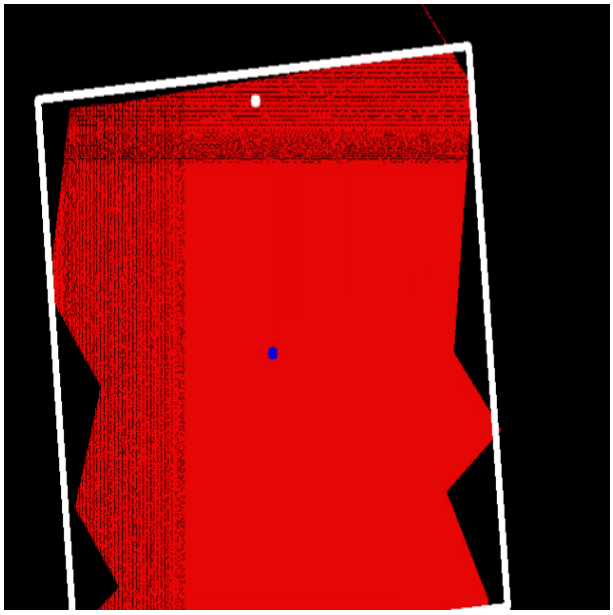


Figure: Blue point is the center of the rectangle and white point represents next waypoint in image frame.

2.3.3 Next Waypoint Calculation using PCA

If the above condition is not satisfied then we move to the final algorithm which uses PCA analysis. A key point of PCA is Dimensionality Reduction. Dimensionality Reduction is the process of reducing the number of the dimensions of the given dataset. PCA takes array of 2D points and finds the best fitted line. This helps in getting the orientation of the road. So we take the binary image(road is in white and the background is in black) and store white pixels(road part) in **Points** array. Then we apply PCA on **Points** which gives us two **Eigen Vectors** in the direction of best fitted line and perpendicular to it. The **center** of the road is also calculated by an OpenCV function, **pca_analysis**. We pick the vector with the larger eigen value as our drone movement direction. Here another condition check was applied. If the eigen values of both

the vectors are approximately equal, it means that we are not able to distinguish between the two directions of movement. Hence an average of the two Eigen Vectors was used as the final direction.

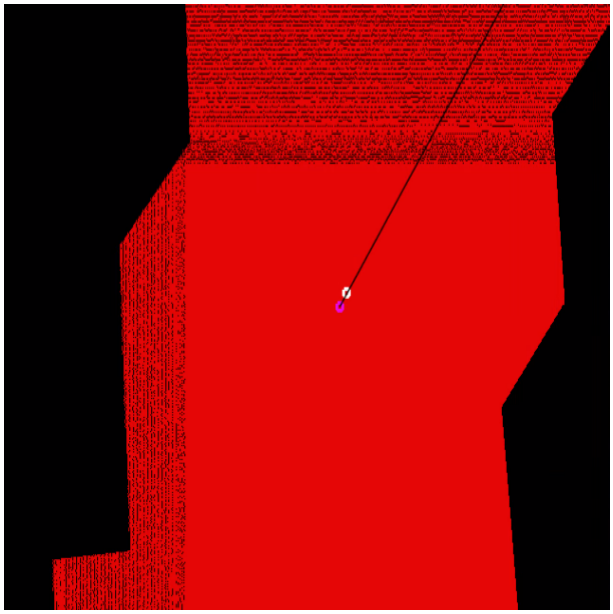
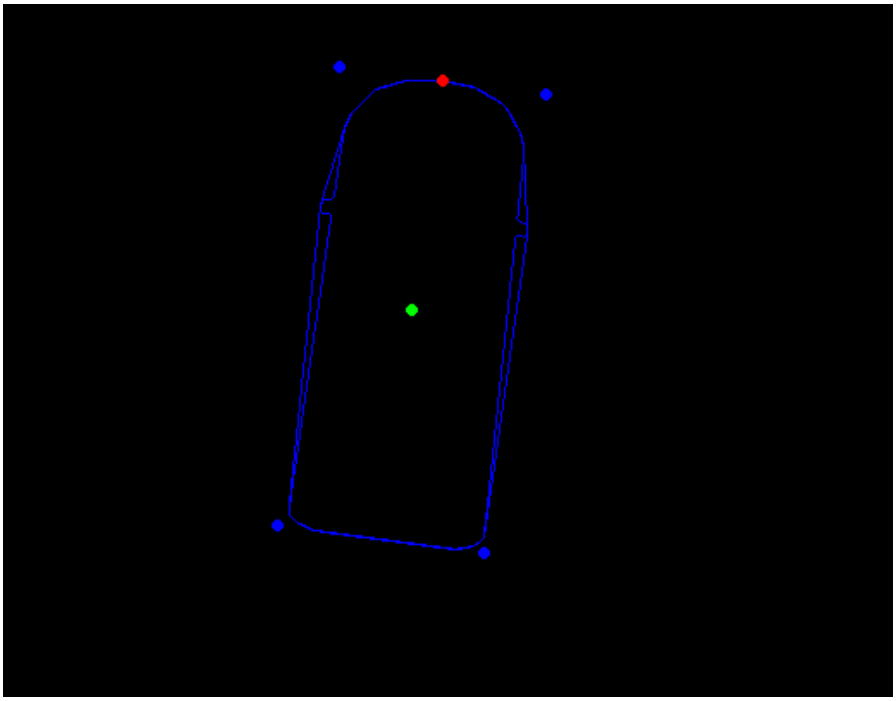


Figure: Black line detects the direction of movement of drone in the next timestamp. Pink point is the center detected by PCA and white point represents next waypoint in image frame.

In both MinAreaRect and PCA method of finding next waypoint, a corrective measure is applied. If the center detected by any of algorithms is too far from the center of the image (which means that the drone is at the edge of the road), wavepoints are given such that the drone moves to the center of the road.

2.4 Prius Detector

The details of the orientation of the ground vehicle are necessary for modulating and fine tuning the control system. The rectangular top of the ground vehicle is detected, along with its corner points and centre point. The centre point allows the aerial vehicle to track the ground vehicle and the orientation of the rectangular box is used to establish a feedback loop to control the steering of the ground vehicle. The depth image is extracted from the gimbal's depth camera and then processed into a single channel gray scale image. Contours are then constructed upon the gray channel image.

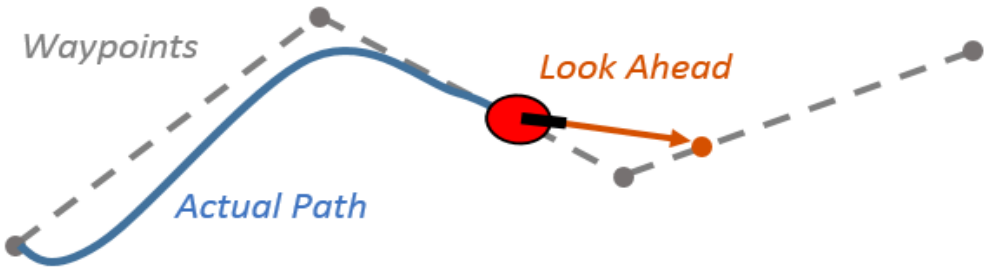


Next we used the approach of filtering the raw image on the basis of its RGB value. Gaussian blur was performed on the image to filter out irregularities. In order to avoid other objects from disturbing the outcome, we ensured that the number of points in the convex hull of the prius lied within a certain range, as an additional check. After detecting the prius, we then evaluate the center of the prius along with center of the line connecting the front wheels in order to get the orientation of prius.

2.5 UGV Controller

We have used the Pure Pursuit path tracking algorithm for UGV controller. It computes the steer angle that moves UGV from its current position to reach some look-ahead point in front of the UGV. The algorithm then moves the look-ahead point on the path based on the current position of the UGV until the last point of the path is reached. You can think of this as the UGV constantly chasing a point in front of it. The property *look ahead distance* decides how far the UGV would look for the next waypoint.

Look ahead distance property is the main tuning parameter for the controller. The figure below shows the UGV and the look-ahead point. Note that the actual path does not match the direct line between way-points, as is evident in this image.



The effect of changing this parameter can change how your UGV tracks the path and there are two major goals: regaining the path and maintaining the path. In order to quickly regain the path between way-points, a small *look ahead distance* will be beneficial, as it will cause your robot to move quickly towards the path. However, as can be seen in the figure below, the robot overshoots the path and oscillates along the desired path. In order to reduce the oscillations along the path, a larger look ahead distance can be chosen, however, it might result in larger curvatures near the corners.

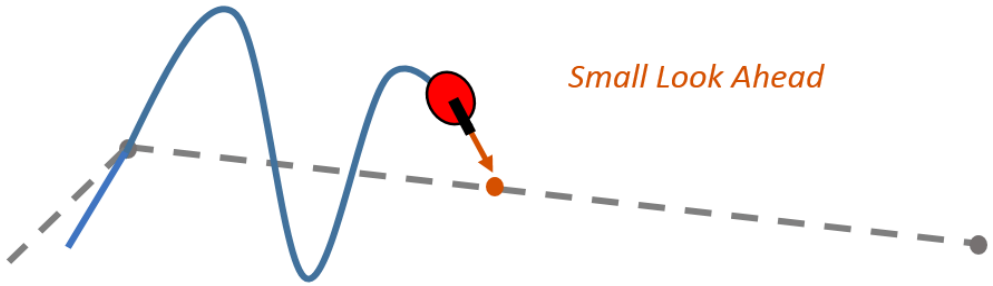


Figure : Small Look-Ahead

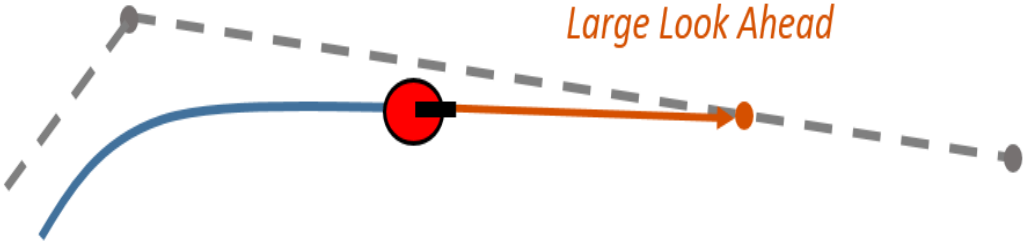


Figure : Large Look-Ahead

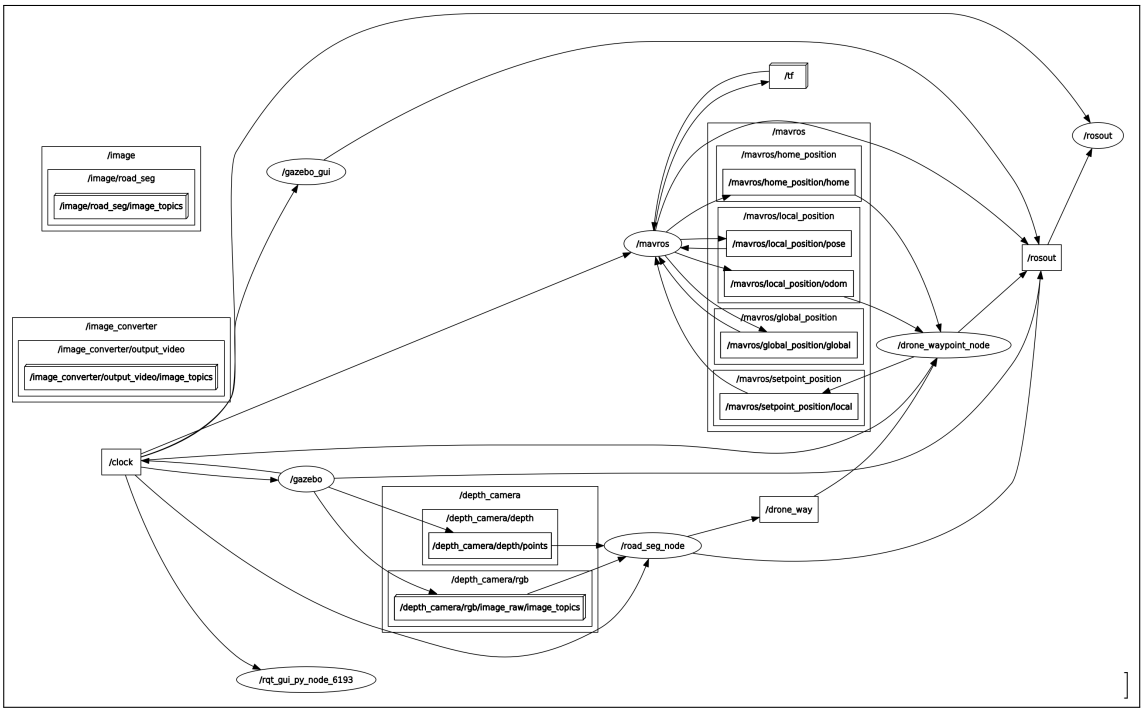
A sharp change in curvature of the path of UGV with high speeds will cause the rear end to skid. To overcome the skidding of the UGV at high speeds we have used a PID velocity controller and made the target speed as a function of steer angle so as to slow down at the turns.

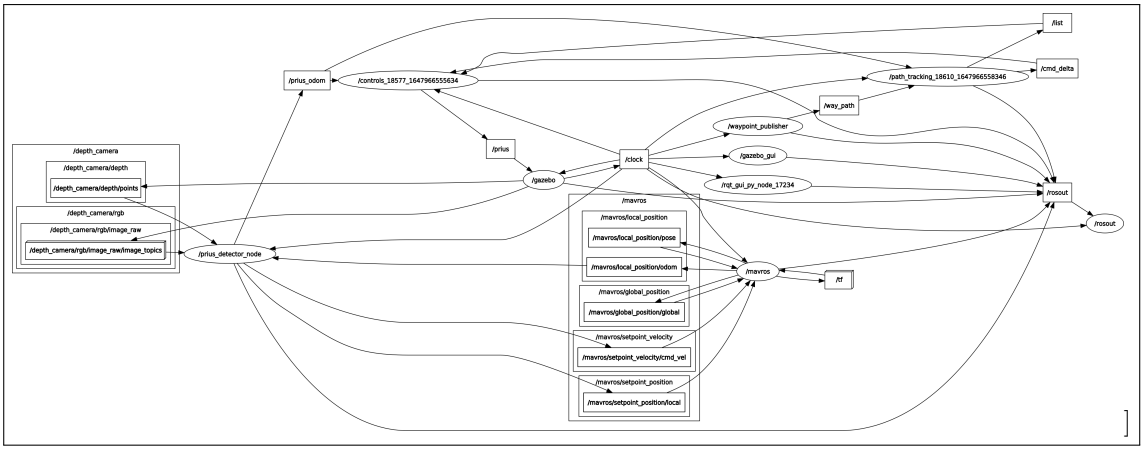
$$v = max\_vel - (0.5 * \theta)$$

where v is the speed of the UGV and θ is the *steer angle in degree*. We tried to improve the pure pursuit model by adding a feedback and a Low pass filter for smoother steer but it lead to more oscillations, so it was later removed.

3 Software Architecture Description

3.1 rqt_graph





3.2 Parameter Files

The parameter files being used are:-

3.2.1 segmentation

- *params.yaml* - camera parameters and transformation matrices

3.2.2 prius_detector

- *params.yaml* - camera parameters, rotation matrices and other general cv parameters

3.3 Nodes

The nodes being run are as follows:-

3.3.1 road_seg_node

Subscribed topics:-

- /depth_camera/rgb/image_raw: Gets the raw RGB image from the camera
- /depth_camera/depth/points: Gets the pointcloud from the depth camera

Published topics:-

- /image/road_seg : Image of the segmented road
- /drone_way : The next waypoint in the drone frame

3.3.2 mapping_fsm_node

Subscribed topics:-

- /drone_way : Gets the next waypoint in the drone frame.
- /mavros/local_position/odom : Odometry of the drone
- /mavros/home_position/home : Odometry of the home position

Published topics:-

- /mavros/setpoint_position/local : For publishing waypoints in the ground frame.

3.3.3 prius_detector

Subscribed topics:-

- /depth_camera/depth/points : Gets the pointcloud from the depth camera
- /clock : Simulation timing
- /mavros/local_position/odom : Odometry of the drone

Published topics:-

- /prius_odom : Publishing the odometry of Prius
- /mavros/setpoint_position/local : For publishing waypoints in the ground frame.
- /mavros/setpoint_velocity/cmd_vel : Providing velocity to the drone in the map frame

3.3.4 controls_18577_1647966555634

Subscribed topics:-

- /prius_odom : calcuated odometry of prius
- /clock : Simulation time
- /cmd_delta : Calculated steering angle

Published topics:-

- /prius : Publishes the throttle, steer and brake

3.3.5 path_tracking_18610_1647966558346

Subscribed topics:-

- /prius_odom : Subscribimg the odometry of Prius
- /clock : Simulation time
- /way_path : Subscribing mean path

Published topics:-

- /cmd_delta : Calculated Steering angle

3.3.6 waypoint_publisher

Subscribed topics:-

- /clock : Simulation time

Published topics:-

- /waypath : Publishing mean path

4 References

- For Takeoff : px4-docs
- Pure Pursuit
- openCV-docs