Robotics Club IITKanpur

Summer Projects
Science and Technology Council
IIT Kanpur

Autonomous Navigation in Rough Terrain Environments

Documentation

Last Updated:

August 5, 2020

Contents

1 Acknowledgements

We are immensely grateful to Robotics Club, IIT Kanpur and Science & Technology Council, IIT Kanpur for providing us with the resources and motivation for this Summer Project. We thank our mentor: Ashwin Shenai and the Coordinators of Robotics Club: Prakash Choudhary, Ramyata Pate and Suman Singha for their guidance, constant support and invaluable inputs, without which we would not have been able to do this project in a timely manner.

2 Introduction

2.1 Aim

The objective of this project is to virtually implement an autonomous navigation module for a ground robot that can handle undulated terrain and gain information about its environment as it moves through the course.

Learning Goals:

- Implement ROS architecture for the complete project
- GAZEBO for simulation
- CV for vision-based navigation
- Path planning using depth cameras
- LiDAR-based obstacle avoidance

2.2 Brief Overview

We aim to implement the functionalities of the module by having a ground robot that completes a specifically designed course in simulation.
The navigation is checkpoint-based, guided by GPS and ArUco markers. An obstacle avoidance module is also implemented to allow the robot to navigate the terrain obstacles.

The main parts of the projects are:

- GPS and IMU are used for localisation and navigation.
- A 2D Lidar and Camera are used to detect obstacles and ArUco markers.
- An obstacle avoidance module and path planning is implemented to allow the robot to navigate the terrain obstacle

2.3 Plan of Action

Main aim of the project was the autonomous navigation of robot in rough terrain. The project was divided into several sections which summed up to the main aim. All these sections' documentation are accompanied by tutorials and references.

3 ROS Installation

For this project we have used the ROS Melodic Morenia Distribution.(1)

3.1 WSL on Windows

The most popular method for setting up a ROS system on Windows is via the Windows Subsystem for Linux (WSL) with Ubuntu 18.04. Windows Subsystem for Linux is a compatibility layer for running Linux binary executables natively on Windows 10 and Windows Server 2019.

3.2 Docker on Windows

For users unable to comfortably configure WSL, an alternative approach is to use Docker. Docker can package an application and its dependencies in a virtual container that can run on any Linux server.

3.3 Virtual Machines

The third alternative for Windows and Mac systems is to use VMs. Virtual Machines provide a substitute for a real machine. They provide functionality needed to execute entire operating systems. For Mac, one can install a VM with Ubuntu 18.04 LTS. Using VM with Windows, however might not be able to run Gazebo which is required for the project since one cannot allocate all of the system's resources to the VM.

3.4 Linux

Instead of using VMs or WSL which provides a container on the host OS, one can also Dual Boot Ubuntu 18.04 beside the main operating system. This is the best option, however doing so incorrectly can completely damage the system, hence extreme caution is recommended. Following references were followed to setup Dual Boot on Windows and Mac.

4 ROS Basics

4.1 Creating a workspace

Workspace is a directory which contains all the ROS source files, where you can create, edit and modify ROS packages. For creating a ROS workspace, we first install catkin. Then a workspace is created as per the following steps mentioned in the documentation(2): Creating a Workspace.

4.2 Package Structure

Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. ROS packages are created using catkin_create_pkg command as per the documentation for: Creating a Package.

4.2.1 CMakeLists.txt

The file CMakeLists.txt is the input to the CMake(3) build system for building software packages. Any CMake-compliant package contains one or more CMakeLists.txt file that describe how to build the code and where to install it to. The basic structure and components of CMakeLists.txt are available here.

4.2.2 package.xml

This file defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages. The basic structure of package.xml is defined here.

4.3 Subscribers and Publishers

ROS nodes are executable files within the ROS package which uses a client library to communicate with other nodes. Nodes communicate with each other by sending and receiving ROS messages(composed of ROS data types) on a Topic. A Node can recieve data from a topic by Subscribing to it and can send data to the Topic by Publishing to it. The documentation for ROS provides detailed instructions for creating a Publisher and Subscriber

4.4 Writing Launch Files

Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters. Launch files are of the format .launch and use a specific XML(4) format. roslaunch is a tool for easily launching the launch files from the user terminal. All launch files go in launch directory - inside the package. Launch files must end in .launch. The XML format used for roslaunch .launch files is specified in detail here.

4.5 ROS Parameters

The Parameter Server is a part of the ROS Network. It saves and provides the values of specific variables for the nodes of that package to use during runtime. These variables are known as "Parameters". Parameters are best used to set values of static configuration parameters, therefore they are saved in the config folder of the package. The Parameter Server can store

integers, floats, boolean, dictionaries, and lists. These can be specified in a launch file or, also as separate parameter files. the ROS documentation provides basic tutorial on ROS parameter usage for both C++ as well as Python.

4.6 Useful utilities

Here are some useful utilities which include few commonly used rqt plugins and some terminal commands. These utilities come handy while debugging or using ROS in general. "rqt" is a software framework of ROS that implements the various GUI tools in the form of plugins. rqt_graph, rqt_image_view, rqt_plot, rqt_publisher, rqt_gui are some of the commonly used plugins. rosnode, rostopic, rosparam, rosservice, rosmsg and rossrv are few useful commands. roscd, rosed , rosrun, rospack find, rosls, roscore are few other commonly used commands. Also another commonly used tool is tab completion.

5 Simulation Environment

5.1 Creating a Gazebo Model

Step 1: Meshes
A mesh model consists of vertices, edges, and faces that use polygonal representation, including triangles and quadrilaterals, to define a 3D shape. Unlike solid models, a mesh has no mass properties. Gazebo(5) provides a set of simple shapes: box, sphere, and cylinder, that can be used in combination to create a basic representative model.
Custom meshes can be modelled directly from CAD files using MeshLab(6), Blender or Sketchup. Gazebo requires that mesh files be formatted as STL, Collada or OBJ, with Collada and OBJ being the preferred formats. Meshes created via Blender etc. can be exported in the above formats. **This is an example of a mesh file of a flag that we have used in our project.**

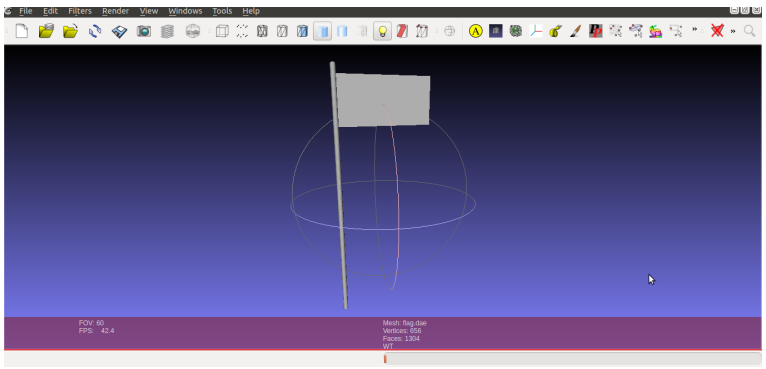


Figure 1: Example of a mesh file of the flag used in the project

Step 2: Creating the model's SDF
Gazebo models are defined using the SDF format. A model is created then by generating an SDF file and including the appropriate meshes to define the visual, collision and other physical properties of the model. Note the model's meshes have to be inside a 'meshes' folder inside the model directory - along with the SDF file.
Following is the SDF file for the flag model using the above flag's mesh:

```
<?xml version="1.0" ?>
<sdf version="1.5">
  <model name="flag">
      <static>true</static>
      <link name='link'>
       <collision name="collision">
        <geometry>
            <mesh>
              <uri>model://flag/meshes/flag.dae</uri>
              <scale>1 1 1</scale>
            </mesh>
        </geometry>
        <surface>
          <contact>
             <collide_bitmask>0xffff</collide_bitmask>
          </contact>
```

```
          <friction>
            <ode>
              <mu>100</mu>
              <mu2>50</mu2>
            </ode>
          </friction>
        </surface>
      </collision>
        <visual name='visual'>
          <geometry>
            <mesh>
              <uri>model://flag/meshes/flag.dae</uri>
              <scale>1 1 1</scale>
            </mesh>
          </geometry>
        </visual>
      </link>
  </model>
</sdf>
```

Step 3: Add to the model SDF file
The SDF file can then be extended following the same procedure as above in order to define more complex models whose individual parts themselves have their own meshes and the links need to be movable between these parts
We followed the following procedure in order to extend our model:

- Add a link.
- Set the collision element.
- Set the visual element.
- Set the inertial properties.
- Go to 1 until all links have been added
- Add all joints (if any).
- Add all plugins (if any).

With each new addition, load the model using the Gazebo GUI to ensure that the model is rendered and behaves as expected
The following tutorial was referred to for this section.

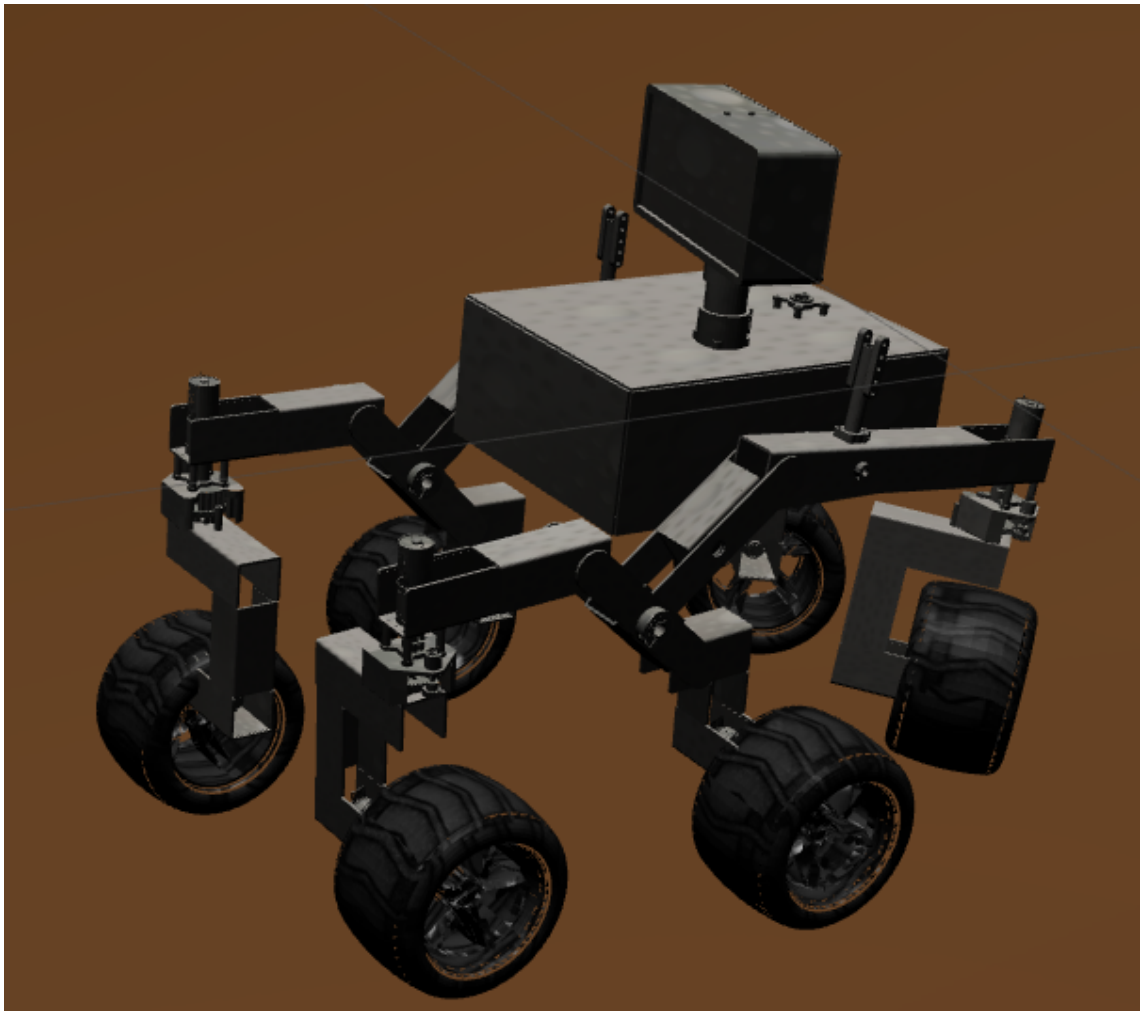


Figure 2: Rover

5.2 Defining the Robot URDF

Step 1: Defining the tree structure

At first, we wrote the basic structure, and stated all the links and their connections with other links.

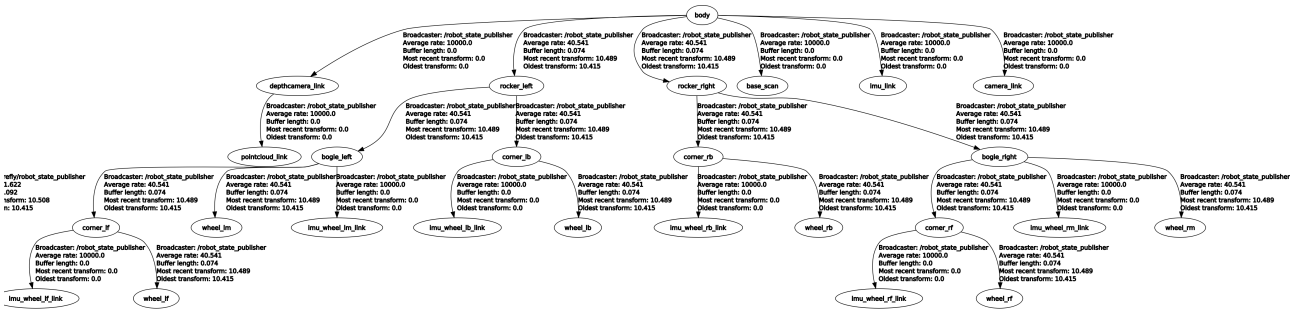


Figure 3: Network of joint links in the rover

A small part of the whole urdf file that we have used for rover description is used as an example below:

```
<link name="bogie_right"/>
<joint name="rocker_right_bogie_right" type="continuous">
  <parent link="rocker_right"/>
  <child link="bogie_right"/>
</joint>
```

Step 2 : Add the dimensions

After the first part, the second thing is to add appropriate dimensions. So, to add dimensions to our tree, all we specified was the offset from a link to the joint(s) of its children. To accomplish this, we added the field "origin" to each of the joints.
Let's look at the rocker_right_bogie_right joint. It was offset in the all three directions from rocker_right joint,and it was exactly parallel to rocker_right joint. So, we added the following "origin" element:

```
<origin rpy="0  0  0" xyz="-0.19542 -0.08523 -0.0081"/>
```

Then we repeated this for all the elements and our URDF looked like this:

```
<link name="bogie_right"/>
  <joint name="rocker_right_bogie_right" type="continuous">
    <parent link="rocker_right"/>
    <child link="bogie_right"/>
    <origin rpy="0  0  0" xyz="-0.19542 -0.08523 -0.0081"/>
  </joint>
```

Step 3: Complete the Kinematics

What we didn't specify yet was around which axis the joints rotate. Once we add that, we

actually had a full kinematic model of this robot! All we need to do was add the ¡axis¿ element to each joint. The axis specifies the rotational axis in the local frame.
So, if you look at rocker_right_bogie_right joint, you see it rotates around the positive Z-axis. So, we simply added the following xml to the joint element:

```
<axis xyz="0 0 1" />
```

Note that it is a good idea to normalize the axis.
After adding this to all the joints of the robot, our URDF looks like this:

```
<link name="bogie_right"/>
<joint name="rocker_right_bogie_right" type="continuous">
<parent link="rocker_right"/>
<child link="bogie_right"/>
<origin rpy="0  0  0" xyz="-0.19542 -0.08523 -0.0081"/>
<axis xyz="0  0  1"/>
<limit effort="1.0" lower="0.0" upper="0.0" velocity="0.0"/>
</joint>
```

After adding more details to these links and joints , it looked like this:-

```
<link name="bogie_right">
   <inertial>
     <mass value="0.621260139018214"/>
     <origin rpy="0  0  0" xyz="0  0  0"/>
     <inertia ixx="0.00453008397919647" ixy="0.00490350467843443"
     ixz="0.000635833755568301" iyy="0.00836086403351998"
     iyz="0.000607789161715965" izz="0.0121912838568506"/>
   </inertial>
   <collision name="bogie_right_collision">
     <origin rpy="0  0  0" xyz="0.25249  0.02578 -0.0854"/>
     <geometry>
       <mesh filename="package://rover_sim/models/rover/meshes/bogie_right.STL"
       scale="1 1 1"/>
     </geometry>
   </collision>
   <visual name="bogie_right_visual">
     <origin rpy="0  0  0" xyz="0.25249  0.02578 -0.0854"/>
     <geometry>
       <mesh filename="package://rover_sim/models/rover/meshes/bogie_right.dae"
       scale="1 1 1"/>
     </geometry>
   </visual>
 </link>
 <joint name="rocker_right_bogie_right" type="continuous">
   <parent link="rocker_right"/>
   <child link="bogie_right"/>
   <origin rpy="0  0  0" xyz="-0.19542 -0.08523 -0.0081"/>
   <axis xyz="0  0  1"/>
```

```
  <limit effort="1.0" lower="0.0" upper="0.0" velocity="0.0"/>
</joint>
```

The following tutorial was referred for the implementation of this section.

5.3 The rover_sim package

Software in ROS is organised in packages. The rover_sim package contains ROS nodes, configuration files, urdfs, worlds, launch files, CmakeLists.txt and package.xml file.

After building your rover_sim package and using roslaunch rover_sim package default.launch, you'll see a chunk of mars-like terrain, a crater and a 6-wheeled rover. Using rqt_graph in a separate terminal, you can see all the nodes running and the topics that they publish. Mainly there's:

1. A normal camera publishing images on /rover/camera/image_raw
2. A depth camera publishing in /rover/depthcamera
3. LiDar data on /rover/scan
4. Odometry data on /rover/odom
5. A central imu on /rover/imu and an imu on each wheel
6. A bumper sensor on /rover/robot_bumper

5.3.1 Rover Joint Description

Our Mars rover has six wheels and uses a rocker-bogie suspension system to drive smoothly over bumpy ground. There is one rocker-bogie assembly on each side of the rover. The "rocker" part of the suspension comes from the rocking aspect of the larger, body-mounted linkage on each side of the rover. These rockers are connected to each other and the vehicle chassis through a differential(a gear train with three shafts). Relative to chassis, the rockers will rotate in opposite directions to maintain equal wheel contact. The chassis maintains the average pitch angle of both rockers. One end of a rocker is fitted with a drive wheel and the other end is pivoted to the bogie.

The rocker-bogie design has no springs or stub axles for each wheel, allowing the rover to climb over obstacles (such as rocks) that are up to twice the wheel's diameter in size while keeping all six wheels on the ground. The system is designed to be used at slow speed of around 10 centimetres per second (3.9 in/s) so as to minimize dynamic shocks and consequential damage to the vehicle when surmounting sizable obstacles.

5.3.2 Sensors on the Rover

1. LiDAR(Light Detection and Ranging) is an active remote sensing system that can be used to to capture information about a landscape and record things that we can use to estimate conditions and characteristics. LiDAR is a method of measuring distances by illuminating the target with laser light and measuring the reflection with a sensor. Differences in laser

return times and wavelengths can then be used to make digital 3-D representations of the target. And so the reason why it is commonly used to make high-resolution maps which we have used.

2. IMU(Inertial Measurement Unit) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the orientation of the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers.

3. Bumper Sensor lets our robot to sense contact. These are digital sensors which can detect and avoid obstacles and provide feedback to microcontrollers. This sensor works by acting like a SPST switch. When the "whisker" bumps into a foreign object it will make contact with a nut next to it, closing the connection and, by default, turning off the motor. By attaching these mechanical bumpers to your robot the whisker will bump something before your robot crashes into it. Note that the bumper sensor is not used in our project, instead we use LiDAR data for local obstacle avoidance.

4. The camera on the Rover publishes images on /rover/camera/image_raw. The image_raw topic gives raw data from the camera driver.
   Camera parameters are available at /rover/camera/camera_info. It is used for box detection and marker detection in this project.

5. Depth Camera provides depth maps that are used in the path planning section of the project.

6. Odometry sensor provides history-based sensor mappings, which indicate how far the robot has travelled, based on what amount that the wheels have turned. This is based on an EKF implemented on the IMU data, which is provided as an out-of-the-box Gazebo plugin.

7. GPS receiver is used so that our rover is capable to navigate to given targets.GPS provides the required position data and a current heading. This data in conjunction with a list of way-points obtained from the map is used to calculate the required correction in heading. This is then used to navigate the vehicle. The GPS data is implemented as a Gazebo plugin added to the rover URDF.

6 Controllers

6.1 Velocity Controller

6.1.1 Implementation

The Mars rover has six velocity_ controllers of type "**effort_controllers/JointVelocityController**". They are as follows:

1. corner_lf_wheel_lf_controller
2. bogie_left_wheel_lm_controller
3. corner_lb_wheel_lb_controller
4. corner_rb_wheel_rb_controller
5. bogie_right_wheel_rm_controller
6. corner_rf_wheel_rf_controller

The velocity controller mainly uses a **differential drive system** where to attain a linear speed **v** and angular speed **w** we use **-(v - wheelSep*w/2)** for the right wheels and **v + wheelSep*w/2** for the left wheels, where **wheelSep** is the separation between the left and right wheels. The **Position Controller** heavily depends on the **Velocity Controller**.

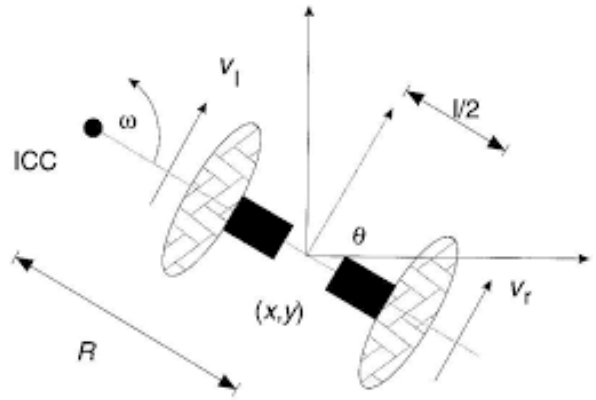


Figure 4: Description of the differential drive system implemented

The following code snippet implements the **move** function for the controller:

```
wheelSep=0.3; // model dimension
w =50*angle;
a.data = v + wheelSep*w/2;
pub_vel_lf.publish(a);
pub_vel_lm.publish(a);
pub_vel_lb.publish(a);
a.data =-(v  -  wheelSep*w/2);
pub_vel_rf.publish(a);
pub_vel_rm.publish(a);
pub_vel_rb.publish(a);
```

6.2 Position Controller

6.2.1 Implementation

For the rover reach a particular position setpoint, we first give some angular speed to rover and turn its direction using velocity controllers until the yaw of rover is nearly equal to the required angle , i.e.

$$\tan^{-1}\frac{y}{x}$$

where y is the y-coordinate of desired position and x is the x-coordinate of desired position as shown in the figure below.
To calculate yaw of rover, we subscribe to the **/rover/odom** topic and then convert current position's orientation quarternion to roll,pitch,yaw angles.

```
tf::Quaternion q(quat.x, quat.y, quat.z, quat.w);
tf::Matrix3x3 rot(q);
double roll, pitch, yaw;
rot.getRPY(roll, pitch, yaw);
return yaw;
```

Once the yaw is close enough to the required angle, we use the velocity controllers again to give a forward velocity proportional to the distance between desired position and current position, and once the rover is close enough to the desired position, we stop the velocity controllers and hence rover will be at the desired position.

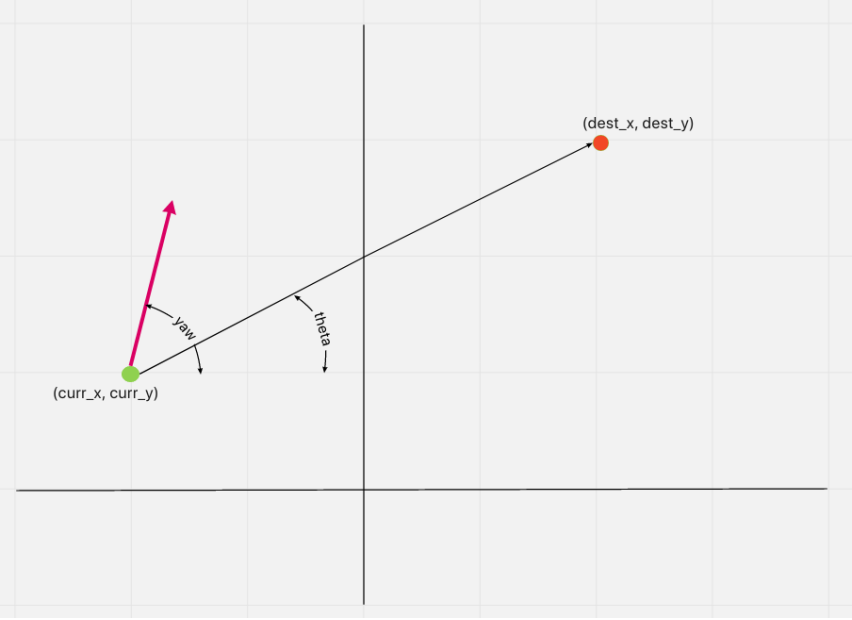


Figure 5: Graphical representation of orientation of the robot and the destination

Following code snippet implements the position controller:

```
dist = sqrt((dest_x-curr_x)*(dest_x-curr_x)
            + (dest_y-curr_y)*(dest_y-curr_y)
            + (dest_z-curr_z)*(dest_z-curr_z));
angle = yaw + atan(dest_y-curr_y)/(-(dest_x-curr_x));
if((curr_x-dest_x)<0){
    if(yaw<0) angle += M_PI;
    else angle -= M_PI;  // yaw correction
}
if(dist<0.3){
    speed = 0;
    angle = 0; // stop
}
else{
    if(abs(angle)<0.07){
        speed = dist*10;
        if(abs(angle)<0.035) final_angle = 0; // forward
    }
    else{
        speed = 0;
        final_angle = angle; // rotate
    }
}
geometry_msgs::Twist msg;
msg.linear.x = speed;
msg.angular.z = final_angle;
pub.publish(msg);
```

7 GPS Navigation

7.1 Frame Transformations

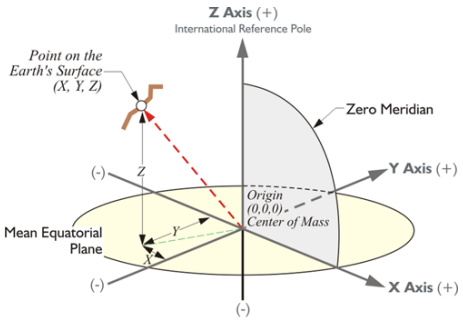


Figure 6: The Cartesian Coordinate system

We are using GPS coordinates for navigation of the rover. For that we included a GPS plugin(7) **hector_gazebo** in the rover urdf which will publish the rover's current odometry using **sensor_messages/Navsatfix message**. We are using Frame Transformations to convert Cartesian-coordinates into GPS-coordinates and vice versa. We use Geographic Library's

UTMUPS::Forward (and similarly UTMUPS::Inverse) functions for carrying out this transformation.

7.2 GeographicLib

GeographicLib(8) is a small set of C++ classes for performing conversions between geographic, UTM, UPS, MGRS, geocentric, and local Cartesian coordinates, for gravity , geoid height, and geomagnetic field calculations, and for solving geodesic problems. The link to the official site for installation and setup can be found here

The Project has used to library to obtain GPS coordinates of the rover in the gazebo world and for navigation as well to travel to at a location of known GPS coordinates.

7.3 Implementation

The library was used from here refer here.
Configuring the **CMakeLists.txt** is required to use it in the ROS world. The changes to use GeographicLIb in CMakeLists.txt should be to add these in the file:

```
find_package(GeographicLib REQUIRED)

include_directories(${GeographicLib_INCLUDE_DIRS})

add_executable(programme src/file_name.cpp)
target_link_libraries(programme ${GeographicLib_LIBRARIES})
```

After configuring CMakeLists.txt the next part is including/importing header files in respective .cpp/.py files for using it. For Rover the part was to convert Lat Long to UTM coordinates. So,

```
#include <GeographicLib/UTMUPS.hpp>
```

was useful.**UTMUPS::Forward()** is used here for conversion of lat long to UTM
example:

```
int zone;
bool northp;
double x, y, g, k;
void callBack(const sensor_msgs::NavSatFixConstPtr& msg){
    UTMUPS::Forward(msg->latitude, msg->longitude, zone, northp, x, y, g, k);
    string zonestr = UTMUPS::EncodeZone(zone, northp);
    ROS_INFO_STREAM("X: "<<x<< "Y: "<<y);
}
```

8 Fiducial Marker-based Navigation

8.1 ArUco Markers

An ArUco(9) marker is a synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier (id). The black border facilitates its fast detection in the image and the binary codification allows its identification and the application of error detection and correction techniques. The marker size determines the size of the internal

matrix. For instance a marker size of 4x4 is composed by 16 bits.It must be noted that a marker can be found rotated in the environment, however, the detection process needs to be able to determine its original rotation, so that each corner is identified unequivocally. This is also done based on the binary codification. Each detected marker includes the position of its four corners

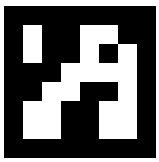


Figure 7: Aruco Marker

in the image (in their original order) and the id of the marker.

8.2 Algorithm Description

So in ArUco based navigation we have a world in which there's one red flag, a black circle for the goal zone and an ArUco marker hovering in the air. Our task was to navigate inside the black circle. GPS coordinate of the flag was given and we could get to that point using the GPS navigator. Once we reach that point, the marker pad is in view. After that, we rely on the marker's estimate to get to the goal.
We have also written a basic box detection node that detects the centre white flickering rectangle hovering in the air. For that we threshold the image to segment the markers. Then contours are extracted from the threshold Image and those that are not convex or do not approximate to a square shape or are too small or big are discarded.
Now we have to detect the marker using **aruco_ros**.The basic marker detection with box detection is as follows :-

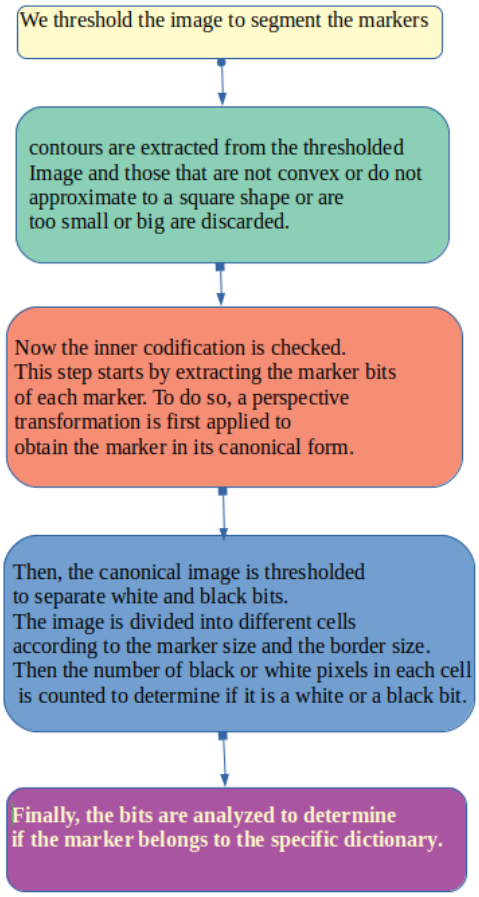


Figure 8: Flowchart for marker detection with box detection Algorithm

We had to convert the centre coordinates from the image frame to the global frame using the scaling factor so that we could use it as a setpoint(10). Camera parameters are available at */rover/camera/camera_info*

```
geometry_msgs::Pose converted;
sensor_msgs::CameraInfo camera_info
float cx =cy =fx =fy=0 ;   //camera parameters
void callback1(sensor_msgs::CameraInfo info){
    cx=info.K[0][2];
    cy=info.K[1][2];
    fx=info.K[0][0];
    fy=info.K[1][1];
}
void callback2(aruco_controller::Centre point){
    float u=point.x;
    float v=point.y;
    if(cx==0 && cy==0 && fx==0 && fy==0)
    {
        converted.position.x=0
        converted.position.y=0
    }
    else
    {
        converted.position.x=(u-cx)/fx;
        converted.position.y=(v-cy)/fy;
    }
    cout<<"("<<u<<",\t"<u<<")\n";
}
```

8.3 Implementation

The library was installed from here refer here.
Configuring the **CMakeLists.txt** is required to use it in the ROS world. The changes to use aruco in CMakeLists.txt should be to add these in the file:

```
find_package(aruco REQUIRED)
find_package(OpenCV 3 REQUIRED)

include_directories(
# include
${catkin_INCLUDE_DIRS}
${OpenCV_INCLUDE_DIRS}
${aruco_INCLUDE_DIRS}
)

add_executable(node_name src/file_name.cpp)
target_link_libraries(node_name ${catkin_LIBRARIES} ${OpenCV_LIBS} ${aruco_LIBRARIES})
```

If you are creating a separate node for scaling the camera parameters then add that executable in the same manner. You can omit linking **{aruco_LIBRARIES}** to this node. No need to add executable for python files.
Now include/import header files in respective .cpp/.py files for using it. Use,

```
#include <aruco/aruco.h>
#include <aruco_ros/aruco_ros_utils.h>
#include <cv_bridge/cv_bridge.h>
```

9 LiDAR-based Obstacle Avoidance

9.1 Sensor Data Description

We used a LiDAR-based sensor to avoid obstacles. The rover has a 2D LiDAR sensor which publishes a LaserScan message on the /rover/scan topic. The message consists of many useful attributes. The sensor sends out light rays along angles from *angle_min* to *angle_max* with angular difference in two adjacent rays being *angle_increment*. Then it measures the time taken for each ray to return, based on which the distance of the nearest obstacle along that ray is calculated and populated in *ranges*. *range_min* and *range_max* are lower and upper bounds of the ranges,with values lying outside of it would be discarded.

9.2 Algorithm Description

The goal of the *Decision Making Algorithm* is to provide the best decision possible for the vehicle at any given time in order to reach its destination. It takes into account multiple factors, such as:vehicle direction, motor state, direction of Goal Position and surrounding obstacles. The algorithm drives the vehicle to the direction of the Goal Position until an obstacle is detected. When the vehicle cannot keep going forward due to a detected obstacle, the algorithm then checks left and right to determine the possibility of a turn. If both left and right directions are free of obstacles, the algorithm favors the direction closer to the Goal Position.

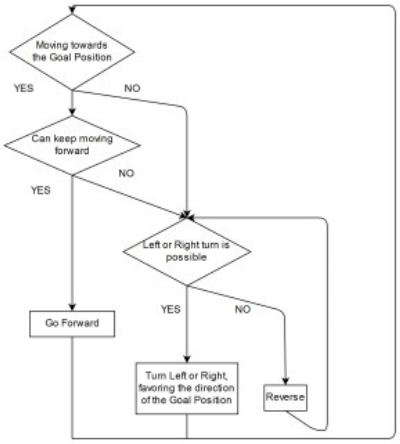


Figure 9: Flowchart of the Decision Making Algorithm

9.3 Implementation

For implementing the Lidar based obstacle avoidance -:
1- We used the Position and Velocity controller we had previously implemented.
2- Using the library

```
#include<sensor_msgs/LaserScan.h>
```

to get the Sensor Data from the Lidar Scanner. Using the *ranges* and *range_max* attributes of the LaserScan message,we got the distances of all nearby obstacles whose distances are less than a small value(some fixed value around 3-6) and set a flag variable to 1 signifying nearby obstacle for avoiding by changing direction.
3- If the value of flag is 1, we published commands for rover to turn either left or right. Else the Rover moves in the direction towards the Goal Position, specified by Position Controller.

Reference implementation for the local obstacle avoidance algorithm:

```
void lidar_callback(sensor_msgs::LaserScan laser_data){// Receiving the LaserScan message
    flag.data=0;
    for(int i=0;i<360;i++){
        if((laser_data.ranges[i]>0.40)&&(laser_data.ranges[i]<4.0)){
            flag.data=1;
        }
    }
}
```

10 Path Planning with Depth Maps

10.1 Setup and Installation

The simulation environment now includes a quadrotor equipped with its own depth camera. The aim was to use the quadrotor to survey the path to the goal, and then generate a global plan that the rover can follow.

10.1.1 Dependencies Required

1. For simulation of Quadrator: RotorS(11) Simulator from here
2. For control of the Quadrotor(12) : from here
3. Voxblox(13) Library :Voxblox(14) is a volumetric mapping library based mainly on Truncated Signed Distance Fields (TSDFs). from here
4. Existing Voxblox Planners, only for the handy PlanningPanel it provides. from here

10.2 Algorithm Description

Implementation of the path planning mainly involves two algorithms-
Planner Algorithm and Collision Check Algorithm.

1.Planner Algorithm: A* on graph generated using randomly sampled points.
A* algorithm is a heuristic function based algorithm for proper path planning. It calculates heuristic function's value at each node on the work area and involves the checking of too many adjacent nodes for finding the optimal solution with zero probability of collision.

2. Collison Check Algorithm:
Our overall system functions in two parts: first, incorporating incoming sensor data into a TSDF, and then propagating updated voxels from the TSDF to update the ESDF.

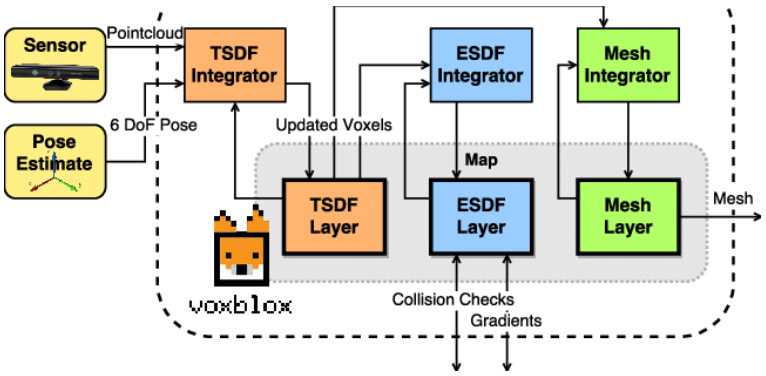


Figure 10: Flowchart for Mesh formation function

Truncated Signed Distance Field (TSDF) is an implicit surface representation, where the zero-crossings represent locations of surfaces, positive values indicate free space, and negative values indicate occupied space behind a surface. The distances in a TSDF are taken along each sensor ray, and truncated to a small radius around the surface boundary. Each voxel in a TSDF also has an associated weight, which allows more accurate merging of multiple scans into a single field.

Euclean Signed Distance Field (ESDF) which contains the true Euclidean to the nearest surface. This representation is needed or useful in many planning applications, as collision checks are significantly sped up by knowing global distances, and collision cost gradients are available everywhere in the map, not only on object boundaries.

The *key difference between* the ESDF and the TSDF is how the distances are calculated –

TSDFs used projective distances, which significantly speed up computation time, and make a good approximation to Euclidean distances very near to the surface boundaries

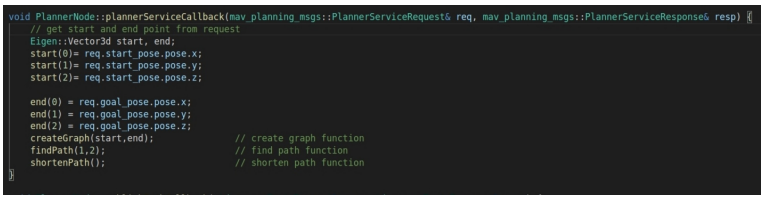
10.3 Implementation

The various steps we followed in the implementation are:

1. Implemented the main planner node with a service Server on a service topic that responds to

```
mav_planning_ msgs::PlannerService
```

requests to get the start point and goal point from the request.



2. Defined a function which determines if a given point is free or occupied: Using the EsdfServer.

```
double PlannerNode::getMapDistance(const Eigen::Vector3d& position){
    if (!server_.getEsdfMapPtr()) {
      return 0.0;
    }
    double distance = 0.0;
    if (!server_.getEsdfMapPtr()->getDistanceAtPosition(position,
                                                        &distance)) {
      return 0.0;
    }
    return distance;
  }
```

If distance is less than the robot_radius, point is occupied else it's not.

3.*Point Sampling*: We can't run A* on the whole complete ESDF because that's a whole lot of points so it's computationally very expensive. Instead, we uniformly sampled out points from the free region between the start and the goal point, thus greatly reducing the size of our graph. We defined the sampling region to be centered around the midpoint of the start and goal point, and then used a random number generator to generate points in this frame which were then converted to the global frame via a simple affine transformation.

4.*Implementing a graph*: Graph is the collection of all node structs, also it provides us some utility function such as add neighbour, delete neighbour etc.

5.*RTree(Rectangle Trees)(15)*: We had the nodes of our graph, we needed to connect them. This completed the graph by telling us which nodes was queried node connected to. For this, we need to find the k-nearest neighbours for each point and then connect them to each other.

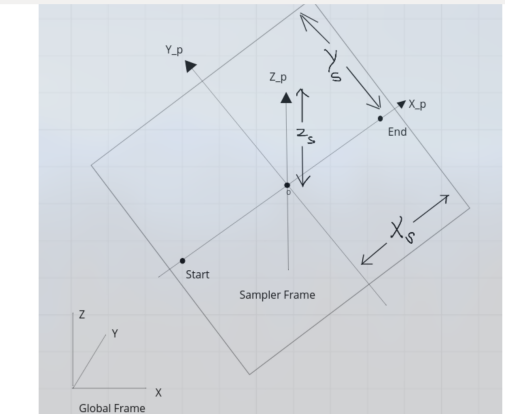


Figure 11: Point Sampling

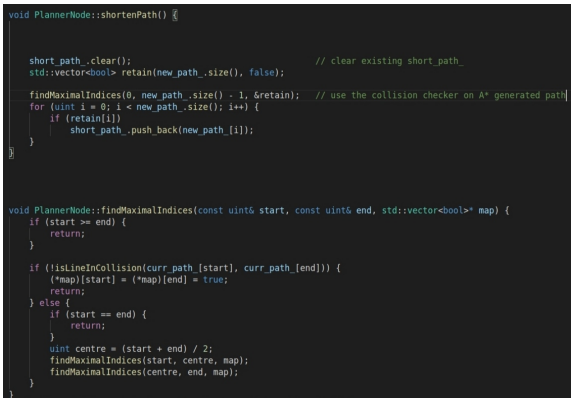
We used RTrees to find the neighbours of each point, which are a bit more efficient than KD-Trees since they partition only the subset region encompassing the points under consideration.

```cpp
void PlannerNode::createGraph(const Eigen::Vector3d& start, const Eigen::Vector3d& end) {
    graph_.clear();
    sampler_.init(start, end);
    double robot_radius = robot_radius_;

    uint max_samples = 10 * (start - end).norm() * sampler_.getWidth() / voxel_size_;
    uint num_sample = 0;
    graph_.push_back(Node(new GraphNode(start, 0)));
    graph_.push_back(Node(new GraphNode(end, 1)));

    uint node_id = 2;
    while (num_sample++ < max_samples) {
        Eigen::Vector3d sample = sampler_.generateSample();
        double distance = 0.0;
        if (getMapDistance(sample, distance) && distance >= robot_radius) {
            graph_.push_back(Node(new GraphNode(sample, node_id++)));
        }
    }

    tree_.clear();
    for (auto& node : graph_) {
        Point pt = Point(node->pos_.x(), node->pos_.y(), node->pos_.z());
        tree_.insert(std::make_pair(pt, node->id_));
    }

    for (auto& node : graph_) {
        std::vector<Value> neighbours;
        Point pt = Point(node->pos_.x(), node->pos_.y(), node->pos_.z());
        tree_.query(boost::geometry::index::nearest(pt, (unsigned) num_neighbours_ + 1), std::back_inserter(neighbours));
        for (auto& neighbour : neighbours) {
            if (neighbour.second != node->id_) {
                node->addNeighbour(graph_[neighbour.second]);
            }
        }
    }
}
```

6. Then we implemented the A* algorithm on the generated graph.

```
void PlannerNode::findPath(const uint& start_index, const uint& end_index) {
if (graph_.empty()) {
        return;
    }
    curr_path_.clear();     // clear existing new_path_
    Eigen::Vector3d end_pos = graph_[end_index]->pos_;
    typedef std::pair<double, uint> f_score_map;
  // implement A*, keep track of parents in a separate vector
    std::priority_queue<f_score_map, std::vector<f_score_map>, std::greater<f_score_map>> open_set;
    std::vector<double> g_score(graph_.size(), DBL_MAX);
    std::vector<uint> parent(graph_.size(), INT_MAX);
    open_set.push(std::make_pair((end_pos - graph_[start_index]->pos_).norm(), start_index));
    g_score[start_index] = 0.0;
    while (!open_set.empty()) {
        uint curr_index = open_set.top().second;
        Eigen::Vector3d curr_pos = graph_[curr_index]->pos_;
        open_set.pop();
 // once end node is reached, backtrack and push all nodes into path
        if (curr_index == end_index) {
            Path new_path;
            while (parent[curr_index] != INT_MAX) {
                new_path.push_back(graph_[curr_index]->pos_);
                curr_index = parent[curr_index];
            }
            new_path.push_back(graph_[start_index]->pos_);
            std::reverse(new_path.begin(), new_path.end());
            curr_path_ = new_path;
            return;
        }
        for (auto& neigh : graph_[curr_index]->getNeighbours()) {
            uint neigh_index = neigh->id_;
            Eigen::Vector3d neigh_pos = neigh->pos_;

            double score = g_score[curr_index] + (neigh_pos - curr_pos).norm();
            if (score < g_score[neigh_index]) {
                g_score[neigh_index] = score;
                parent[neigh_index] = curr_index;
                open_set.push(std::make_pair(score + (end_pos - neigh_pos).norm(), neigh_index));
            }
```

7. Then we shortened our path since our points were randomly sampled there may exist a shorter path between the points of the path. Collision detection between two points was required here. We used ESDF data provided by the *Voxblox Library* for collision detection(in *isLineInCollision* function).



```
void PlannerNode::shortenPath() {



    short_path_.clear();                                // clear existing short_path_
    std::vector<bool> retain(new_path_.size(), false);

    findMaximalIndices(0, new_path_.size() - 1, &retain);   // use the collision checker on A* generated path
    for (uint i = 0; i < new_path_.size(); i++) {
        if (retain[i])
            short_path_.push_back(new_path_[i]);
    }
}



void PlannerNode::findMaximalIndices(const uint& start, const uint& end, std::vector<bool>* map) {
    if (start >= end) {
        return;
    }

    if (!isLineInCollision(curr_path_[start], curr_path_[end])) {
        (*map)[start] = (*map)[end] = true;
        return;
    } else {
        if (start == end) {
            return;
        }
        uint centre = (start + end) / 2;
        findMaximalIndices(start, centre, map);
        findMaximalIndices(centre, end, map);
    }
}
```

```
bool PlannerNode::isLineInCollision(const Eigen::Vector3ds start, const Eigen::Vector3ds end) {


double distance = (end - start).norm();
    if (distance < voxel_size_) {
        return false;
    }
    Eigen::Vector3d direction = (end - start).normalized();

    Eigen::Vector3d curr_pos = start;
    double cum_dist = 0.0;
// check for collisions in the line joining start and end
    while (cum_dist <= distance) {
        voxblox::EsdfVoxel* esdf_voxel_ptr = server_.getEsdfMapPtr()->getEsdfLayerPtr()->getVoxelPtrByCoordinates(curr_pos.cast<voxblox::FloatingPoint>());
        if (esdf_voxel_ptr == nullptr) {
            return true;
        }
        if (esdf_voxel_ptr->distance < robot_radius_) {                    // using getMapDistance and robot_radius
            return true;
        }

        double step_size = std::max(voxel_size_, esdf_voxel_ptr->distance - robot_radius_);     // in increments of voxel size

        curr_pos += direction * step_size;
        cum_dist += step_size;
    }
    return false;
```

Overall Algorithm Execution:

```
start_point // get from request
end_point // get from request
std::vector<GraphNode> graph;
num_sample = 0;
while num_sample < max_samples {
    point <- generateSample(start_point, end_point)
    if getDistance(point) > robot_radius {
    graph.push_back(// new node at point)
    }
}
graph.push_back(// node at start);
graph.push_back(// node at end);

// initialize RTree
for node in graph {
    rtree.insert(...)
}
for node in graph {
    neighbours = rtree.query(node,k, ...)
    addNeighbours(node, neighbours)
}
path = findPath(graph, start_id, end_id); // run A* here
short_path = shortenPath(path); // shorten path
```

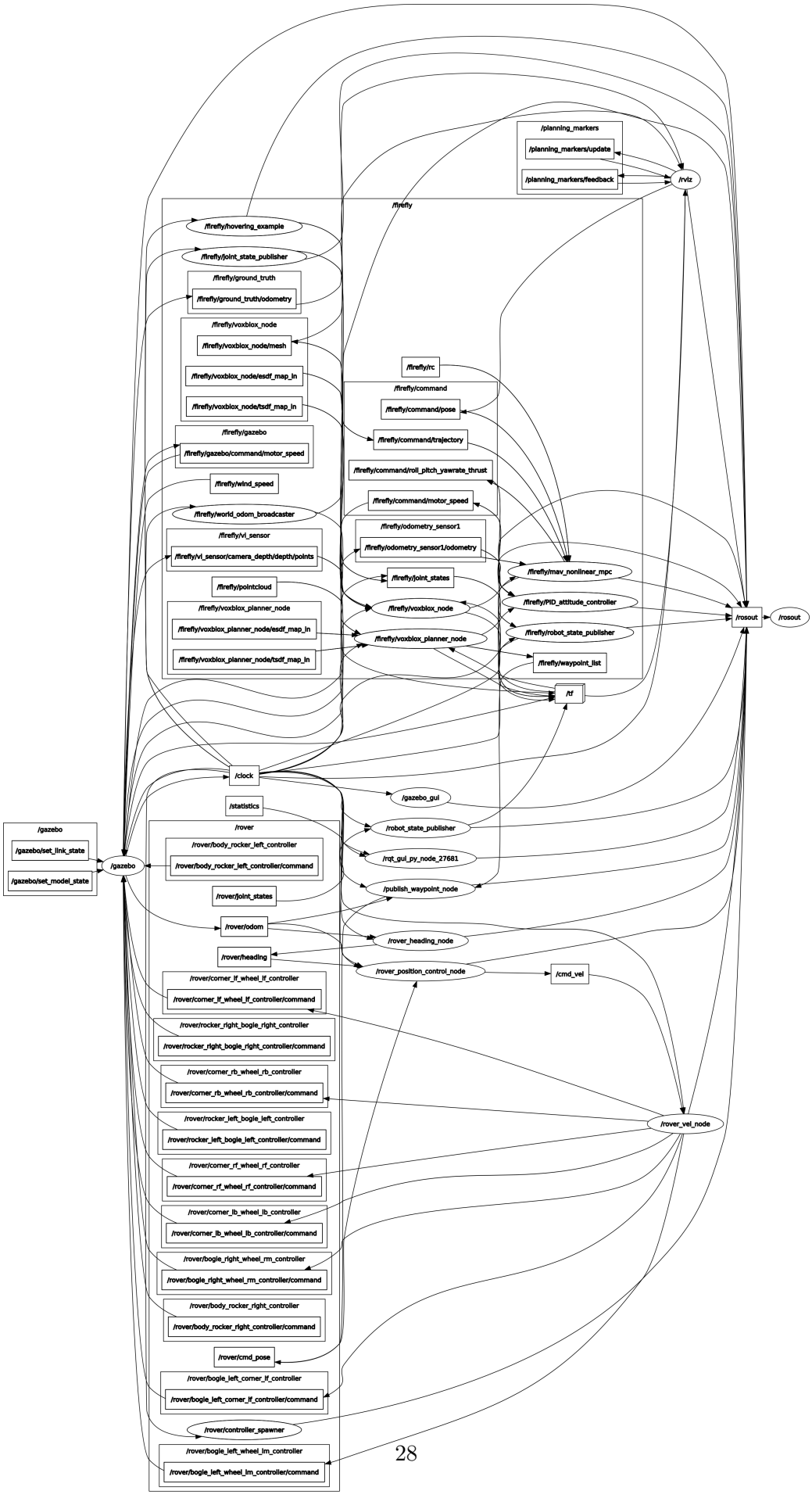


Figure 12: rqt_graph for path planner

11 Troubleshooting

11.1 WSL Issues

Some faced errors while using GUI applications on WSL. We were not able to use Gazebo or play the bag file. The problem found was that the VcXsrv was not properly installed.
Therefore we downloaded the VcXsrv and then we ran the Xlaunch and in the bash terminal -:

```
echo "export DISPLAY=localhost:0.0" >> ~/.bashrc
```

and to save the changes -:

```
. ~/.bashrc
```

11.2 Gazebo Issues

Sometimes model doesn't get launched due to some errors in world files or model SDF files.
For the GPS Controller task we needed to add a GPS Plugin to our model. It wasn't getting launched in the Gazebo simulation with the model due to some incorrect tags in urdf file.

```
=>Properly add GPS Plugin to the sdf file by specifying all tags value.
```

It took a while for us to know that launching big worlds with low gpu power can lead to lesser lighting in the environment and also a bit laggy. This can be solved with a good gpu only or using a smaller world

11.3 Build Issue

We faced major problems in library setup. We needed these libraries for different tasks.
1. Geographic Library
2. ArUco library
3. Voxblox Library
4. Eigen3 Library

Issues came up in libraries installation and linking them to our package by incorporating them properly in CMakelists.txt file.
Proper way to include libraries in our package.Add following line to CMakeLists.txt file:

```
target_link_libraries(${PROJECT_NAME} ${BOOST_LIBRARIES} ${Eigen3_LIBRARIES})
```

12 Future Work

1. Implement a global planner by using Depth Camera on the Rover instead of the Quadrotor.
2. Use Frontier-Based Exploration to make the planning more autonomous.
3. Explore and compare other path searching algorithms like AO* and D* to improve efficiency and reduce time complexity for the planning.
4. Explore other drive systems instead of the current turn-and-move algorithm.
5. Explore more improved and involved methods for LiDAR based local obstacle avoidance.
6. Use the concepts learnt in this project to implement a complete end-to-end solution including the hardware for the URC Challenge.

Bibliography

[1] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[2] ROS Wiki. Website. [Online] https://wiki.ros.org.

[3] CMake. Website. [Online] cmake.org.

[4] eXtensible Markup Language. Website. [Online] https://www.w3schools.com/xml/.

[5] Gazebo Tutorials. Website. [Online] http://gazebosim.org/tutorials.

[6] MeshLab. Website. [Online] www.meshlab.net.

[7] Hector Gazebo for GPS Plugin. Website. [Online] http://wiki.ros.org/hector_gazebo_plugins.

[8] Geographic Library. Website. [Online] https://geographiclib.sourceforge.io/html/.

[9] ArUco Library. Website. [Online] https://github.com/pal-robotics/aruco_ros.

[10] Camera Calibration. Website. [Online] https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[11] Rotor Simulator. Website. [Online] https://github.com/ethz-asl/rotors_simulator.

[12] Mav Controllers. Website. [Online] https://github.com/ethz-asl/mav_control_rw.

[13] Voxblox: Incremental 3D Euclidean Signed Distance Fields for On-Board MAV Planning. Website. [Online] http://helenol.github.io/publications/iros_2017_voxblox.pdf.

[14] Voxblox Library,Voxblox Planners. Website. [Online] https://github.com/ethz-asl/voxblox, https://github.com/ethz-asl/mav_voxblox_planning.git.

[15] RTree of Boost Library. Website. [Online] https://www.boost.org/doc/libs/1_65_1/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost__geometry__index__rtree.html.