Better Autonomous Driving

Report on the updated hardware and changed software for autonomous driving.

Thomas Schenk

Contents

[Context 3](#_Toc133355479)

[Problem 3](#_Toc133355480)

[The launch files 4](#_Toc133355481)

[How-to use the launch files for startup 4](#_Toc133355482)

[Tunability of the remote-jackal.sh script 4](#_Toc133355483)

[Changes made 6](#_Toc133355484)

[Occurring errors 7](#_Toc133355485)

[Results 8](#_Toc133355486)

### Context

The JACKAL was able to drive autonomously. With the old sensory it managed to traverse buildings. With the new and improved hardware, we aim to not only replicate this; but also improve where possible on this feature.

### Problem

The current configuration on the robot still assumes old hardware is in place. As we have seen before, new hardware is present on the robot to improve many things; amongst which is autonomous driving. The robot currently does not know however, that this hardware is there and still assumes old hardware is in place. Because of this, the robot cannot drive autonomously. Due to this being a major part and requirement of the robot, we need to rewrite the robot to accommodate for this change; which in turn provides us with the correct basis to re-enable autonomous driving.

**Note: This document assumes the correct packages for either the camera, Lidar or both, are installed. Mainly because it is a trivial thing, but also since it has been discussed numerous times in other, more domain specific, documents.**

## The launch files

The robot executes all its real-life functions via launch files. These are files which determine what is executed but most importantly in which order these executions are done. For autonomous exploration you will mainly use two of these launch files.

* Exploration.launch

This launch file launches all the software connections between hard- and software. This is the core of the autonomous exploration filesystem, which links everything together. In strict terms, this is the only launch file you need to launch in order for the robot to explore.

* Visualization.launch

While practically it is true that the robot only needs the *Exploration.launch* file, for us developers (or end users) it is nice that we can see what the robot sees. This is where the visualization file comes in. It takes every datapoint the robot publishes and maps these in Rviz; a visualization program for ROS based applications. This tool is nice for when you don’t have a direct line of sight to the robot, or to “dashboard” all the robot outputs in a kind off GUI like environment.

### How-to use the launch files for startup

To get the robot up and running the autonomous exploration package, these are the steps you need to follow from scratch:

//Terminal 1

1. ssh administrator@'robotIp' // password: clearpath

2. roslaunch autonomous\_exploration exploration.launch

// Terminal 2 (visualization only)

1. cd go/to/working/folder/of/the/script

2. source remote-jackal.sh

3. roslaunch autonomous\_exploration visualization.launch

### Editing the remote-jackal.sh script

The *remote-jackal.sh* script offers a wireless connection to the robot on a ROS level. No files can be transmitted or moved, this connection is solely based on the ROS architecture level and is also only used for that. Because it relies on Ip adderresses, it is very likely that each user needs to correctly set their host machine’s Ip before the script will work. Below is a short explanation of how the script works and explains what the Ip’s are used for.

This script sets two very important variables:

The *ROS\_IP* and *ROS\_MASTER\_URI*, these two variables determine the place of a computer in the ROS network.

The *host\_ip* refers to the computer this script is ran on, you could say your local ip.   
The *current\_uri* switches dynamically based on if a connection is possible with the given Ip for the JACKAL.   
If a connection is possible: The ROS master will be set to the JACKAL.  
If a connection is NOT possible: The ROS master will be set to the local Ip.

The whole script is very short:

host\_ip="192.168.131.50"

host\_uri="http://localhost:11311"

echo $host\_ip

echo $host\_uri

# get Jackal IP and save URI

jackal\_ip="192.168.131.1"

jackal\_uri="http://cpr-j100:11311"

echo $jackal\_ip

echo $jackal\_uri

# ping Jackal and save exist status

ping -c1 $jackal\_ip 2>/dev/null 1>/dev/null

connected="$?"

# set ROS master to Jackal if connected to same network

if [[ $connected -ne 0 ]]

then

    echo "Jackal not connected to the same network as host, host is set as ROS master"

    current\_uri=$host\_uri

else

    echo "Jackal is set as ROS master"

    current\_uri=$jackal\_uri

fi

export ROS\_IP=$host\_ip

export ROS\_MASTER\_URI=$current\_uri

The robot uses tailscale to use the internet. This comes with a great benefit of having connectivity all over the world. A downside to this is that the script should be adjusted according to the tailscale Ip of your machine. You most likely have to tune this and find out what your tailscale iP is. The variable *$host\_ip* should then be edited accordingly.

## Changes made

To accommodate for the change in hardware, we need to edit the *Exploration.launch* file like stated before.

The only changes really needed are found in the sensor category, we need to swap out the working directory which is used when pinging the sensor.

The following lines are changed:

<include file="$(find realsense2\_camera/launch/rs\_camera.launch"/>   
<include file="$(find velodyne\_pointcloud)/launch/VLP16\_points.launch">

These lines set the working directories for the ROS packages which interface with the hardware. If these are wrong, the robot cannot use the hardware. In the current version, the robot uses a realsense camera and a velodyne lidar. Above lines should work if the same hardware setup is present.

**Please pay attention to the different setup variables in the exploration.launch file. If the wrong kind is selected, the program could potentially pick a wrong sensor configuration.**

<!--  
Valid robot\_config values (default value can be changed for easy launch command; more configurations can be added if needed):  
- jackal\_astra\_rplidar: Clearpath Jackal robot with front-mounted Intel camera and RPLIDAR-A1 2D laser scanner.  
- jackal\_astra\_velodyne: Clearpath Jackal robot with front-mounted Intel camera and Velodyne VLP-16 3D LiDAR.  
→  
<arg name="robot\_config" value="jackal\_astra\_rplidar"/>

This code example sets the configuration of the *Exploration.launch* file to use the intel x RP sensory setup.

## Occurring errors

During the process of integrating the new sensory, we were blocked for a very long time on a nasty problem. The robot could see obstacles (remember, we can see this using the visualization tool); but the software link which told the robot that it shouldn’t move there; didn’t work.

This piece of functionality relied on a ROS package called costmap\_2d. There is lots of information on the internet about this, so this will be kept short. There is a global and a local costmap, both are identical. Objects are of course identical in all worlds or dimensions. But the robot only uses the local costmap. This local costmap however stayed empty, while the global costmap on which it relies; was updating properly.

This was a problem which bugged us for a very long time, even old project members which we asked for help; couldn’t figure it out.

Finally, we found the problem.

To find the problem, I retrached every possible connection to the *costmap\_2D* package, which is the ROS package that implements the global and local costmaps. After a very long time of testing and experimenting, I found what was blocking me for so long.

The topics and obstacle height *costmap\_common\_params.yaml* weren’t configured correctly. You need root acces to find this file at the following location:

*/opt/ros/noetic/share/jackal\_navigation/params*

We initially didn’t think that topic remapping would be the problem, since topics were remapped at the very beginning of every execution; meaning that the topic essentially only was remapped once and used the same everywhere else.

Obstacle height is a tricky one to figure out, since it requires some fiddling around. If the obstacle height is too low or high in relation to your lidar height; the lidar wont register your obstacles as valid. Dismissing them, in other words. A baseline for this is that the minimum value should never reach below zero. The max height is more tricky to find out and also relies on how high or low you lidar is mounted. You should finetune this to get optimal results.

The current configuration of this file is as follows:

*‘map\_type: costmap*

*origin\_z: 0.0  
z\_resolution: 1  
z\_voxels: 2*

*obstacle\_range: 2.5  
raytrace\_range: 3.0*

*publish\_voxel\_map: true  
transform\_tolerance: 0.5  
meter\_scoring: true*

*footprint: [[-0.21, -0.165], [-0.21, 0.165], [0.21, 0.165], [0.21, -0.165]]  
footprint\_padding: 0.1*

*plugins:*

*- {name: obstacles\_layer, type: "costmap\_2d::ObstacleLayer"}  
- {name: inflater\_layer, type: "costmap\_2d::InflationLayer"}*

*obstacles\_layer:*

*observation\_sources: scan*

*scan: {sensor\_frame: velodyne, data\_type: LaserScan, topic: scan, marking: true, clearing: true, min\_obstacle\_height: 0.0, max\_obstacle\_height: 4.0, obstacle\_range: 2.5, raytrace\_range: 3.0}*

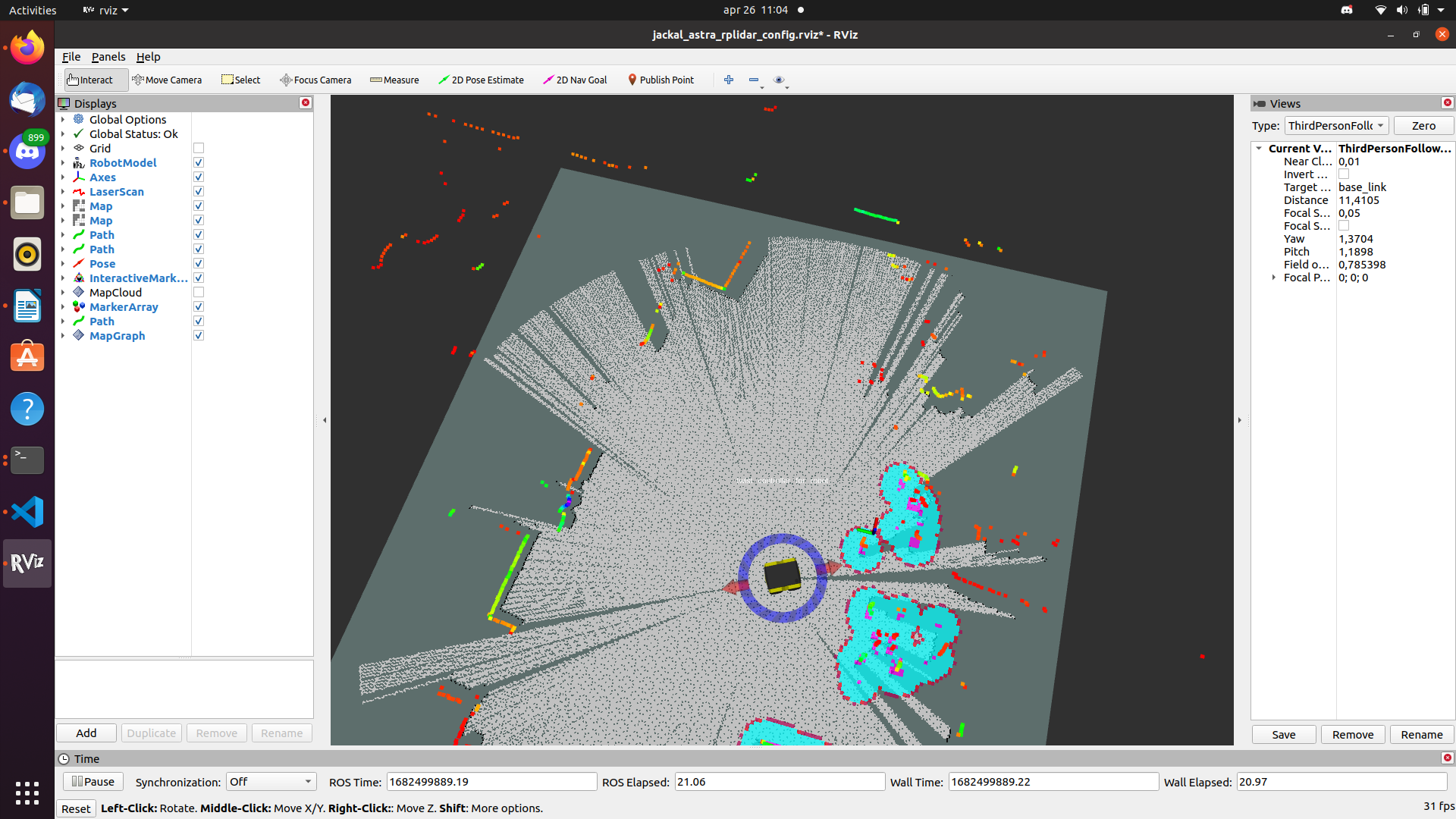
*inflater\_layer:*

*inflation\_radius: 0.30’*

**Please pay attention to this when upgrading or changing hardware! The above mentioned files should NEVER be altered in the current setup.**

## Results

As you can see below, the visualization perfectly represents what the robot sees. Which is, everything! Everything works and the robot can navigate a building all on its own!!

Every gray part is the map, this is the area the robot can ‘see’. Every blue part is an obstacle, the robot sees these and determines it cannot move through the blue part. This is to avoid collisions.

Everything that is not blue or purple, in other words: is not an obstacle; the route planner can plan around. The robot will constanly plan a route to explore more parts of a room, until it determines everything is scanned and moves on.