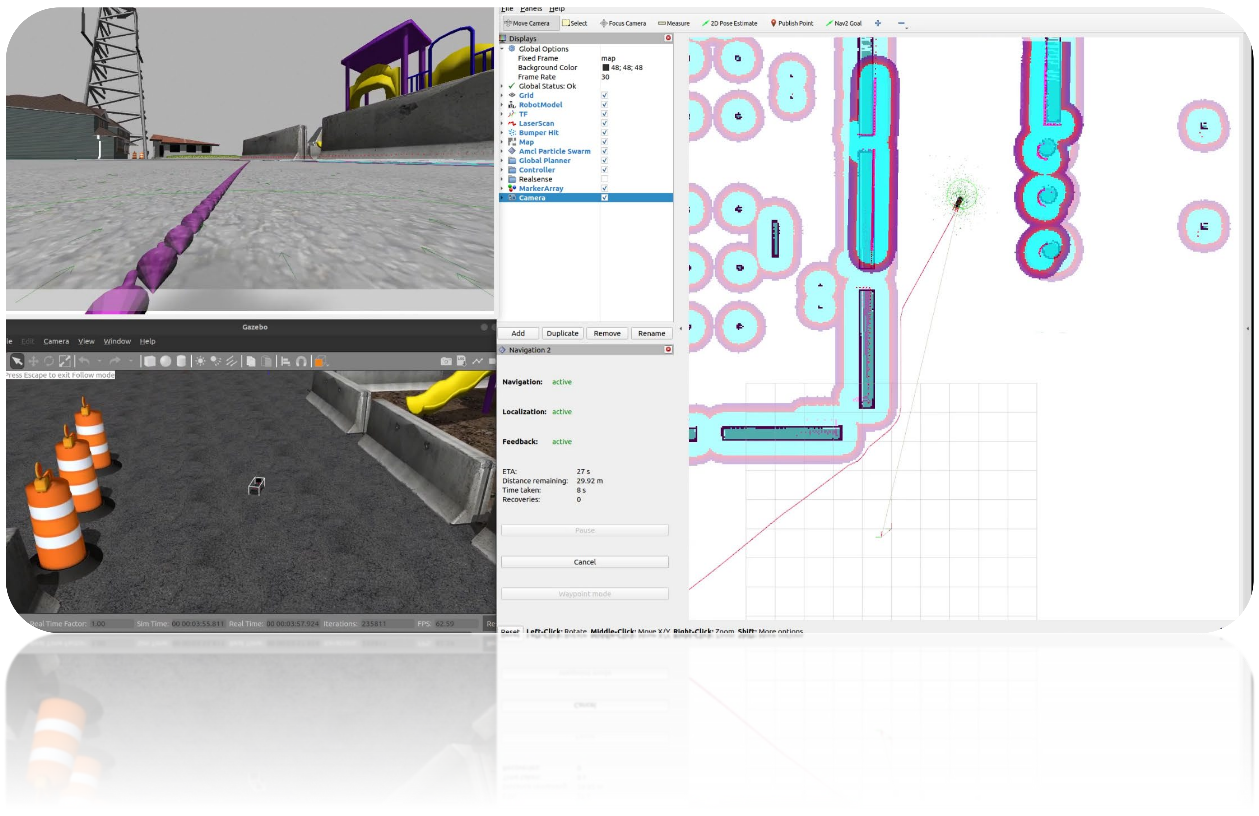
Robot Architecture final assignment – SmartCAR Ros2

In this Assignment, we will use ROS2 and Gazebo with the smartCAR. The goal is to create a fully functional navigation smartCAR for Gazebo simulation and real-life applications.



Global Options
Fixed Frame: map
Background Color: 48; 48; 48
Frame Rate: 30
Global Status: Ok
Grid ✓
RobotModel ✓
TF ✓
LaserScan ✓
Bumper Hit ✓
Map ✓
Amcl Particle Swarm ✓
Global Planner ✓
Controller ✓
Realsense
MarkerArray ✓
Camera ✓

Navigation: active
Localization: active
Feedback: active

ETA: 27 s
Distance remaining: 29.92 m
Time taken: 8 s
Recoveries: 0

Requirements:

- Completed ROS2 Tutorial
- Virtual Box (ROS2_workshop.vbox) or
- Dual Boot Ubuntu22.04

TABLE OF CONTENTS

Version History

VERSION	RELEASE DATE	AUTHOR	CHANGES/UPDATES
1.0.0	2024/9/8	Yanzheng Xiao	- Initial version

1 PREPARATION

1.1 INSTALLATION:

Finish the preparation process that is described in this document.

1.2 ROS2 TUTORIAL

Finish the ROS2 humble tutorial up to the intermediate part (Python) of the ROS2 online tutorial.

Some of the questions can be answered before this.

2 ASSIGNMENT DESCRIPTION

It is highly recommended that you start the assignment from the third lesson after you have finished the intermediate part (Python) of the ros2 online tutorial.

The local localization system and global localization system mean Relative position measurements and Absolute position measurements respectively from the class.

The following requirements are needed for this system:

1. General requirements
   a) Each Ros2 package must have a readme file that has the description of the package, compile instructions, and running instructions.
   b) Each Ros2 package must be able to successfully compile using colcon build.
   c) Each Ros2 package launch file must be well-written, commented on, and operational.
   d) Each Ros2 node within the Ros2 package must be commented on and operational.
   e) The system must have a readme file that describes the process of bringing up itself.
2. Robot definition:
   a) The robot URDF file is defined as described in 2.1.1.
   b) The robot description is correctly published in /tf and /tf_static.
   c) The robot model can be displayed in rviz2.
   d) The robot interface is done as described in 2.1.2
3. Gazebo simulation
   a) The sensor of the robot is defined as described
   b) The simulated robot must be able to output all the sensor topics.
   c) The simulated robot must be able to receive and act on the /cmd_vel command.

   d) The simulated robot must be able to report its status on the topic /smart_car/vehicle_status
   e) The odometry for the simulated robot is defined correctly using the kinematic model.
   f) The coordinates reference frame is defined as described in REP-105.
   g) The simulated robot localization system should be working at the local level.
   h) The navigation2 stack is properly integrated into the robot simulation.
      i. The global localization system must be able to correct the local localization system
      ii. The simulated robot must be able to drive from location to location
      iii. The simulated robot must be able to avoid objects during navigation
4. Real-life application (group assignment)
   a) The robot must be able to receive and act on the cmd_vel command.
   b) The robot must be able to report its status on the topic /smart_car/vehicle_status
   c) The sensor drivers are all properly installed and functional.
      i. The IMU sensor
      ii. The Lidar sensor
   d) The robot must be able to output all the sensor topics.
   e) The odometry for the robot is defined correctly using the kinematic model.
   f) The coordinates reference frame is defined as described in REP-105.
   g) The robot localization system should be working at the local level.
   h) The navigation2 stack is properly integrated into the robot.
      i. The global localization system must be able to correct the local localization system
      ii. The robot must be able to drive from location to location
      iii. The robot must be able to avoid object during navigation

2.1 DEFINING THE SMARTCAR

2.1.1 Creating the Ros2 packages

Create a Ros2 package named smart car in your chosen ros2 workspace. This should be a hybrid package that uses a C++ package and has a CMakelist.txt.

Write down your name and email address in the package.xml file.

2.1.2 Defining the Robot Structure

It is required to use the Xacro file to define your robot URDF. Install the xacro ros2 package using

```
sudo apt install ros-humble-xacro
```

Create an Xacro file named smartcar.urdf.xacro in your ROS 2 package .

Define the robot's structure using the following specifications:

wheel_diameter: 0.064 meter

wheel_width: 0.025 meter

wheelbase_length: 0.257 meter

wheelbase_width: 0.17 meter

Define the robot's base_link, wheel_links, and Chassis_link and base_footprint.

For your wheel joint use the names:

back_left_wheel_joint

back_right_wheel_joint

front_left_wheel_steer_joint

front_left_wheel_joint

front_right_wheel_steer_joint

front_right_wheel_joint

Add appropriate mass, inertia, and center of mass for each component this could be assumed.

Validate your Xacro file by converting it to URDF using the xacro command.

Ensure the URDF file is valid and without errors.

Add your xacro file in to the folder 'urdf' within the package modify the cmakelist file to install the folder in the shared folder. For Python launch file function to search easily.

2.1.3 Creating launch file with Robot state publisher

Install joint stat publisher gui with

```
sudo apt install ros-humble-joint-state-publisher-gui
```

Create a launch file named 'smartcar.launch.py'.

Configure it to launch the robot_state_publisher node with the SmartCar URDF file.

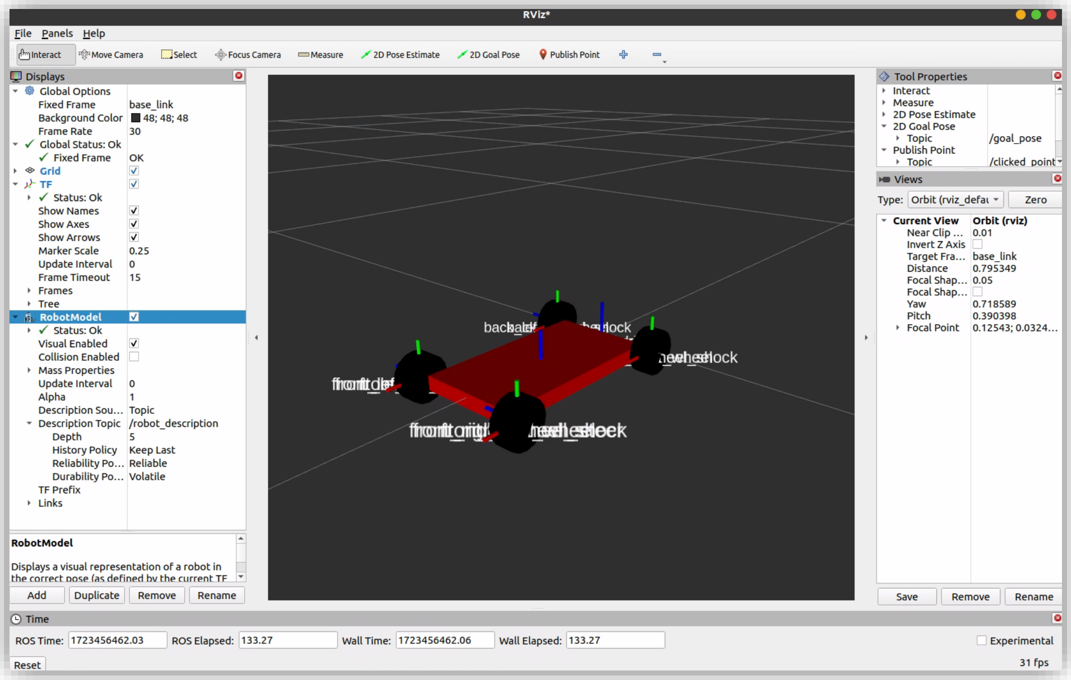
Add the joint_state_publisher_gui to the launch file to allow for manual adjustment of joints.

Compile your ros2 packages using colcon build.

Run the launch file to ensure that the robot state is being published correctly.

Add your launch file in to the folder 'launch' within the package modify the cmakelist file to intall the folder in the shared folder. For enable ros2 launch command line interface.

2.1.4 Setup basic Rviz2 visualization



Displays

Global Options	
Fixed Frame	base_link
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0.25
Update Interval	0
Frame Timeout	15
Frames	
Tree	
RobotModel	✓
✓ Status: Ok	
Visual Enabled	✓
Collision Enabled	☐
Mass Properties	
Update Interval	0
Alpha	1
Description Sou…	Topic
Description Topic	/robot_description
Depth	5
History Policy	Keep Last
Reliability Po…	Reliable
Durability Po…	Volatile
TF Prefix	
Links	

RobotModel

Displays a visual representation of a robot in the correct pose (as defined by the current TF

Tool Properties

Interact	
Measure	
2D Pose Estimate	
2D Goal Pose	
Topic	/goal_pose
Publish Point	
Topic	/clicked_point

Views

Type: Orbit (rviz_defau

Current View	Orbit (rviz)
Near Clip …	0.01
Invert Z Axis	☐
Target Fra…	base_link
Distance	0.795349
Focal Shap…	0.05
Focal Shap…	☐
Yaw	0.718589
Pitch	0.390398
Focal Point	0.12543; 0.0324…

Time

ROS Time: 1723456462.03 ROS Elapsed: 133.27 Wall Time: 1723456462.06 Wall Elapsed: 133.27 ☐ Experimental

31 fps

Set up an RViz2 configuration file (smartcar.rviz) that includes visualizing the robot model.

This can be done by opening up rivz2 by command line and add TF and Robot Model Display in the rivz2 window.

Ensure that the robot appears correctly in the 3D view, with all links and joints properly displayed.

Ensure that the TF and RobotModel displays are properly configured.

2.1.5 Defining the robot ros2 interface

Creating a new ros2 interface package called 'smartcar_msgs'.

Add folder msg and put the create a new file called Status.msg.

Put the following content in:

int32 battery_voltage_mv

int32 battery_current_ma

float64 battery_percentage

float64 steering_angle_rad

int32 engine_speed_rpm

properly configured the cmakelist and make sure it can compile.

2.2 THE SMARTCAR IN GAZEBO SIMULATION

2.2.1 Setup sensor configuration

For basic navigation, A robot should have the following two sensors imu and lidar. Then we must define then in gazebo simulation

IMU Sensor:

Position: The IMU should be placed at the center of the robot (half the wheelbase length along the X-axis).

Plugin: Use the 'gazebo_ros_imu_sensor' plugin.

The sensor data should be published to a topic named /imu_data.

The IMU should be linked to the frame named imu_link.

LIDAR Sensor:

Position: The LIDAR should be mounted at the front of the robot (at the wheelbase length along the X-axis) with a slight elevation.

Plugin: Use the 'gazebo_ros_ray_sensor' plugin.

Sensor output should be of type sensor_msgs LaserScan.

The data should be published to a topic named /scan.

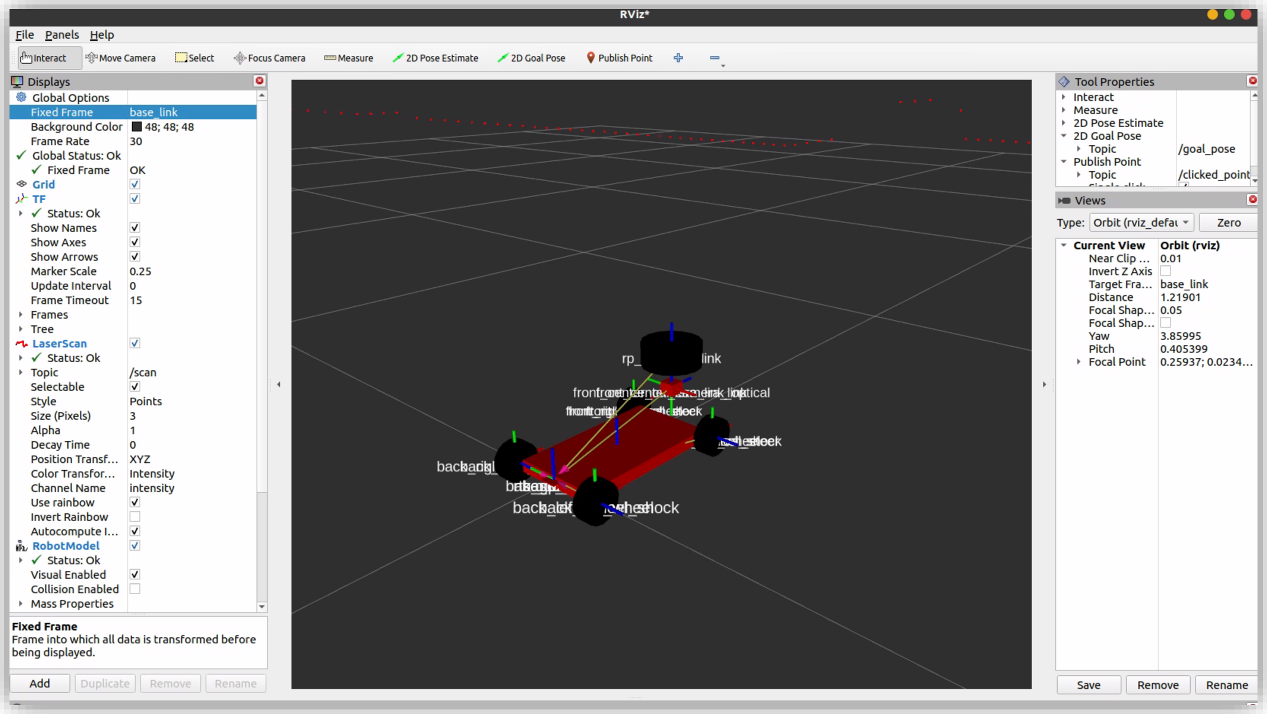
The sensor should be linked to the base_link of the robot.

The LIDAR should be linked to the frame named sllidar_base_link.

Updated smartcar.urdf.xacro file with sensor configurations in the smartcar package.

Optional: make separate file and pass argument to it for setup the sensor.

Rerun the rivz2 with the updated urdf file.



Global Options	
Fixed Frame	base_link
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
TF	✓
✓ Status: Ok	
Show Names	✓
Show Axes	✓
Show Arrows	✓
Marker Scale	0.25
Update Interval	0
Frame Timeout	15
Frames	
Tree	
LaserScan	✓
✓ Status: Ok	
Topic	/scan
Selectable	✓
Style	Points
Size (Pixels)	3
Alpha	1
Decay Time	0
Position Transf...	XYZ
Color Transfor...	Intensity
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	
Autocompute I...	✓
RobotModel	✓
✓ Status: Ok	
Visual Enabled	✓
Collision Enabled	
Mass Properties	

Fixed Frame
Frame into which all data is transformed before being displayed.

Interact	
Measure	
2D Pose Estimate	
2D Goal Pose	
Topic	/goal_pose
Publish Point	
Topic	/clicked_point

Type: Orbit (rviz_defau

Current View	Orbit (rviz)
Near Clip ...	0.01
Invert Z Axis	
Target Fra...	base_link
Distance	1.21901
Focal Shap...	0.05
Focal Shap...	
Yaw	3.85995
Pitch	0.405399
Focal Point	0.25937; 0.0234...

2.2.2 Setup the gazebo simulation

It is required to use gazebo to simulate the robot in a virtual environment, the following packages should be installed:

```
sudo apt install gazebo

sudo apt install ros-humble-gazebo-ros-pkgs
```

For simulating the system properly a custom gazebo plugin is developed, please download the package from link. Decompress it and put it in your workspace.

In addition there are also a map including in the zip file decompress it and put it in you main ‘smartcar’ package create a folder world and modify your cmakelist file properly to include it in the shared folder.

Now you are ready to set up the gazebo simulation.

Implement the custom plugin to control the robot using ‘libcar_gazebo_plugin.so’ plugin. Add it to your urdf file.

Note if you did not follow the naming conventions from defining your robot in chapter 2.1.2. Please change the joint controller pointer in the source code accordingly.

Make a new launch file called ‘gazebo.launch.py’, and define your simulation content:

a. Start with copying something from the ‘smartcar.launch.py’

b. Adding the map path for gazebo to read.

c. Include ‘gazebo_ros’ package ‘gazebo.launch.py’ to run the gazebo in a launch file

   This is including a launch file in a launch file

d. Include ‘gazebo_ros’ package ‘spawn_entity’ node to spawn your robot in the gazebo world.

Compile your ros2 packages using colcon build. And launch the gazebo.launch.py.

If everything goes correctly you would see the following topics:

/clock

/cmd_vel

/imu_data

/joint_states

/robot_description

/scan

/smart_car/cmd_ackermann

/smart_car/odo_fl

/smart_car/odo_fr

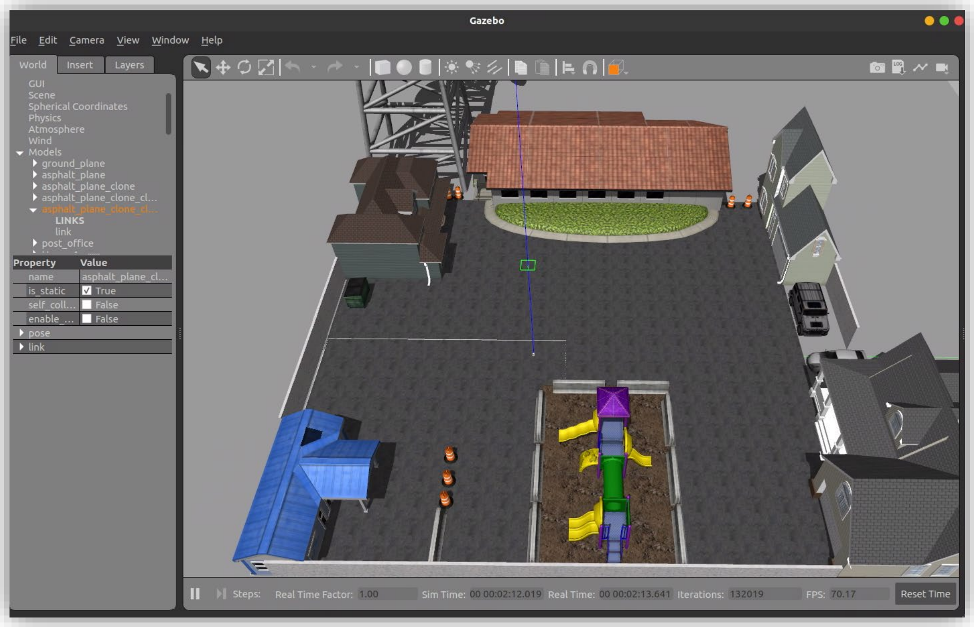
/smart_car/odom

/smart_car/pose

/smart_car/vehicle_status

/tf

/tf_static



Now you can look at each topic and see what's inside. From this moment you have a complete simulation that your can work with.

2.2.3 Setup keyboard control node

Use the teleop_twist_keyboard package to publish velocity commands (/cmd_vel).

ros2 run teleop_twist_keyboard teleop_twist_keyboard

Optional: write your own python keyboard node in package 'smartcar' to control your robot. Colcon build and run it.

2.2.4 Setup wheel odometry

Once the simulation is running you can see the wheel encoder information from the topic /smart_car/vehicle_status. Then you have the following feedback from the robot:

int32 battery_voltage_mv

int32 battery_current_ma

float64 battery_percentage

float64 steering_angle_rad

int32 engine_speed_rpm

Then you can obtain the linear velocity in robot frame using:

$$\mathrm{v} = \frac{\mathrm{RPM} \cdot \pi \cdot \mathrm{D}}{60}$$

Where:

v is the Linear Velocity of the Robot

RPM is the rotation per minute of the wheel

D is the wheel diameter

Then use the simple kinematic model to calculate the angular velocity.

$$\omega = \frac{v}{L} \times \tan(\delta)$$

Where:

v is the Linear Velocity of the Robot

ω Angular Velocity of the Robot

L is the distance between the front and rear axles

δ is the steering angle of the front wheels.

Then the position of a robot in given odom frame can be described as:

$$\varphi = \varphi_{pre} + \omega \cdot dt$$

$$x = x_{pre} + v \cdot \cos(\varphi) \cdot dt$$

$$y = y_{pre} + v \cdot \sin(\varphi) \cdot dt$$

Where:

x, y and φ are the current position of the robot

x_{pre}, y_{pre} and φ_{pre} are previous position of the robot

v is the Linear Velocity of the Robot

ω Angular Velocity of the Robot

dt is the time step.

The integration should always be done in a timer if the dt is fixed, or you can calculate the dt using system clock.

The node should be a single input single output system where the input topic is "/smart_car/vehicle_status", and the output topic is '/smart_car/wheel/odom'

For covariance matrix calculation you can just assume the following for the diagnose component :

position_variance = 1.0

orientation_variance = 0.5

unused_variance = 100000.0

linear_velocity_variance = 0.1

angular_velocity_variance = 0.1

Optional you can calculate the covariance using different statistical method to assume the wheel encoder uncertainty and steering sensor uncertainty.

Add the node in your package smartcar, adding the folder 'script' in your package and modified the cmakelist accordingly to make sure it installed as executable.

Compile the package using colcon build.

Test and verify the node.

2.2.5 Setup the joint state publisher

Set up your joint state publisher python node in your package smartcar. The node should estimate the robot wheel 3-D location using steering_angle_rad and engine_speed_rpm and publish the transformation using TF broadcaster.

Add the node in the folder 'script' of your package smartcar.

Compile the package using colcon build.

Test and verify the node.

Now if you run the rviz2 you should see the joint and if you control your robot you can see the wheel turning.

2.2.6 Setup the local localization system

Set up the robot_localization package to fuse the IMU and odometry data to provide a more accurate estimate of the robot's pose. Read the documentation for robot_localization: https://docs.ros.org/en/melodic/api/robot_localization/html/index.html

Configure an EKF (Extended Kalman Filter) to process the input data /smart_car/wheel/odom and /imu_data and publish the robot's fused pose to /odom

Create a new folder called 'config' to store the yaml file for configuration. Edit your *cmakelist file to put the folder in the shared folder.*

Make a new launch file called 'localization.launch.py', and define your algorithm content:

a. Adding the node that you wrote earlier

    the joint state publisher and wheel odometry.

b. Include 'robot_localization' package 'ekf_node' to run the Extended Kalman Filter

c. Point the yaml file to the as the parameter file

Compile your ros2 packages using colcon build. And launch the gazebo.launch.py.

Launch the localization system and verify that the robot's pose in RViz2 is stable and accurate.

2.2.7 Setup the Nav2

Set up the ROS 2 Navigation Stack (Nav2) to enable autonomous navigation for the SmartCar.

For this you should read: https://docs.nav2.org/ first. To include the navigation 2 software, you could either install the binary version by simply:

```
sudo apt install ros- humble -navigation2

sudo apt install ros-humble-nav2-bringup
```

or following the development guide to build it from source.

It is highly recommended to build the nav2 stack from source, so you have more control for every aspect of the algorithm.

Make sure your system has a stable odometry output, and start by Make a new launch file called 'nav2.launch.py', and define your algorithm content. This can be very versatile such as reused from the nav2's existing launch file, or define everything completely yourself.

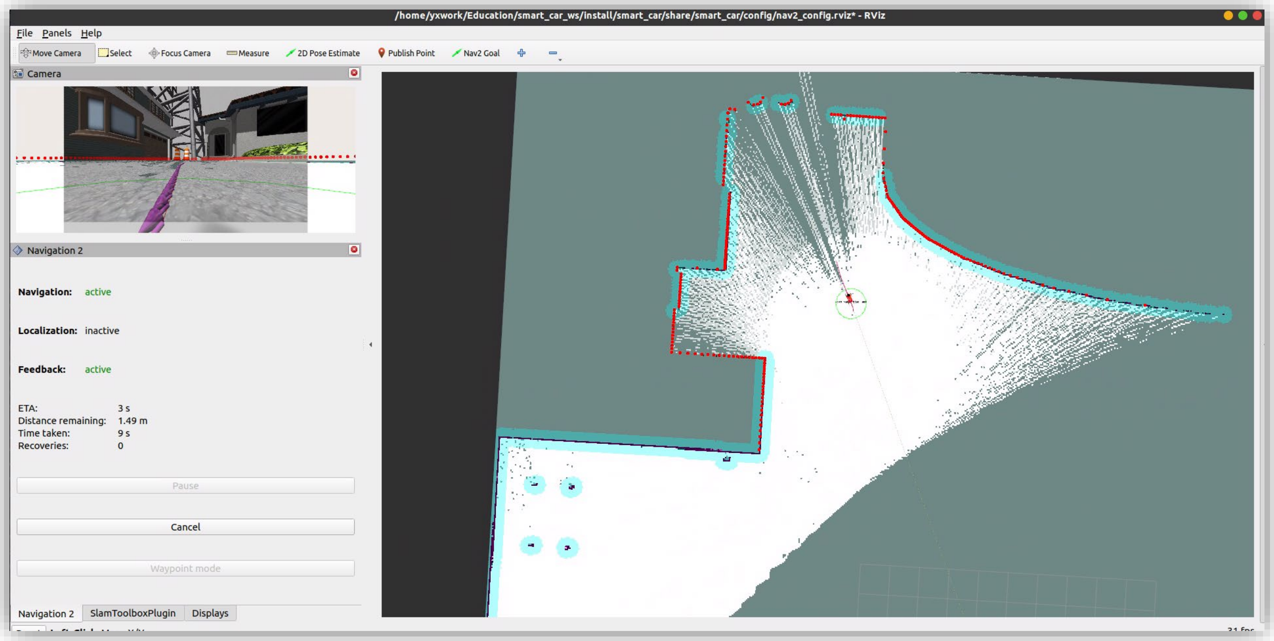
Configure the necessary components, including costmaps, planners, and controllers using the parameter file.

Use the robot's local localization system's output as inputs to the navigation stack.

Download the map files from this links, decompress it, Create a new folder called 'map' to store the yaml and pgm file for map. Edit your cmakelist file to put the folder in the shared folder.

Optional: *Mapping the environment*

Using slam:= true as launch argument or run the slam toolbox separately to start mapping the environment. This could be very difficult depending on the geometry of the map and the accuracy of the localization system. Be slow and always go along the wall.



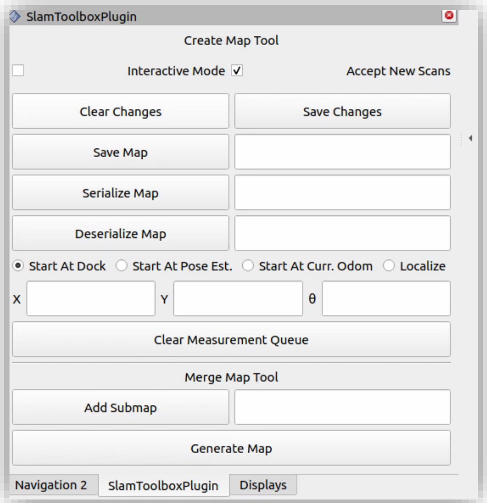
Navigation: active

Localization: inactive

Feedback: active

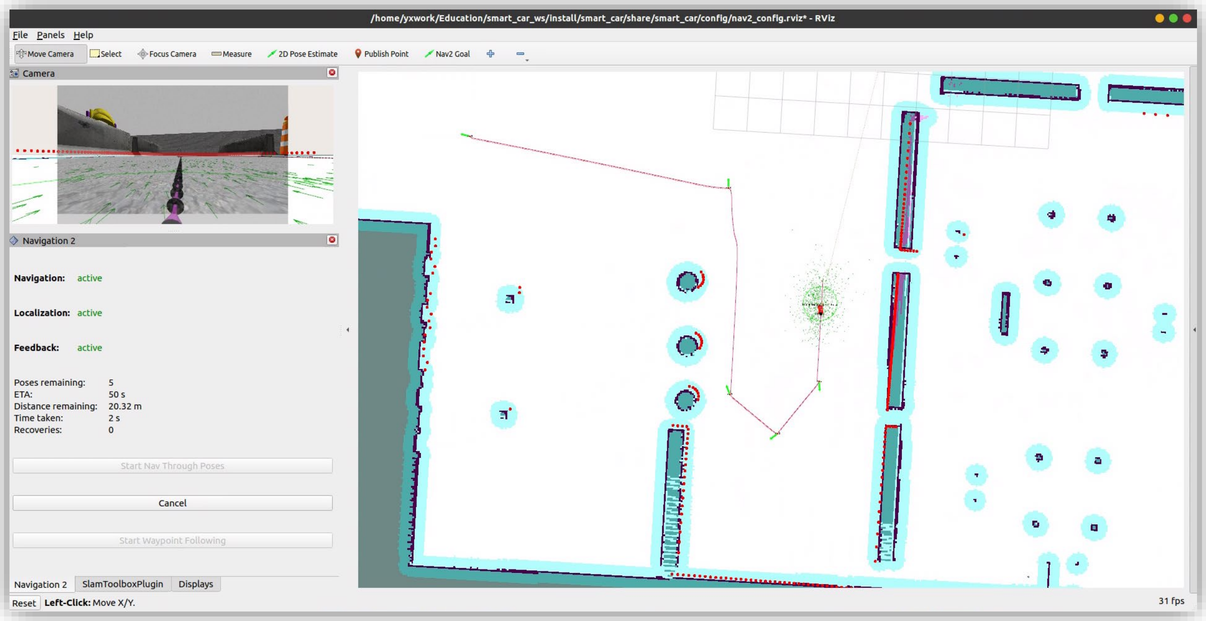
ETA: 3 s
Distance remaining: 1.49 m
Time taken: 9 s
Recoveries: 0

Save the result using the rviz2 slam toolbox plugin.



> *Create a new folder called ‘map’ to store the yaml and pgm file for map. Edit your cmakelist file to put the folder in the shared folder.*
>
> *Point your map path in the parameter file.*
>
> *Now you can use the AMCL as the localization method for global localization.*

Set navigation goals in RViz2 and observe the robot’s ability to navigate autonomously.



```
Navigation:   active
Localization: active
Feedback:     active

Poses remaining:     5
ETA:                 50 s
Distance remaining:  20.32 m
Time taken:          2 s
Recoveries:          0
```

2.3 The SmartCAR in the lab (Group assignment)

2.3.1 Setup the real-life robot interface and calibration

To set up the real-life robot interface please refer to the document on onderwijsonline.

2.3.2 Setup the real-life robot sensor

To set up the real-life robot sensors please refer to the document on onderwijsonline.

2.3.3 Setup keyboard control node

In this step, you could reuse the ros2 node that is in 2.2.3. And try to use it in real-life with the jetson orin.

2.3.4 Setup the real-life wheel odometry

In this step, you could reuse the ros2 node that is in 2.2.4. Use the simple rviz2 visualization to see the performance of the system.

2.3.5 Setup the joint state publisher

In this step, you could reuse the ros2 node that is in 2.2.5. Use the simple rviz2 visualization to see the performance of the system.

2.3.6 Setup the local localization system

In this section, you could reuse the localization system that is in 2.2.6. Use the simple rviz2 visualization to see the performance of the system.

2.3.7 Setup the Nav2

In this section, you could reuse the nav2 system that is in 2.2.7. and repeat the mapping process in the lab.

3 HAND-IN INSTRUCTION.

After finishing the assignment, a complete code repository, a demonstration video and a short report are required as deliverables. In the code repository, each line of code and function must be committed, with functions committed at the front. The demonstration video must show that the program meets all the functional requirements stated in the previous chapter. In the report, you must contain the following sections:

1. Introduction

- Briefly describe the goal of the project and its overall purpose. Mention the specific problem or task the program solves, and how the solution was developed through the assignment.

2. Functional Requirements

- Summarize the functional requirements as outlined in the previous chapter. Clearly list the features that were expected to be implemented and how they guide the development of the solution.

3. Code Structure and Design

- **Repository Overview**: Provide a general description of the code repository's structure. Include how it is organized, the location of the key scripts, and the purpose of each directory or file.
- **Function Documentation**: Provide examples of key functions in the code, detailing what they do, their parameters, and their return values. Mention any important libraries or dependencies used in the code.

4. Program Demonstration

- **Video Explanation**: Give an overview of the demonstration video that accompanies the report. Briefly outline the steps covered in the video, ensuring it highlights how the program meets each functional requirement.
- **Testing Process**: Describe how the program was tested to ensure it performs as expected. Include any specific test cases or scenarios demonstrated in the video.

5. Challenges and Solutions

- Discuss any challenges encountered during development, such as technical obstacles, debugging issues, or design considerations. For each challenge, describe the solution implemented and how it helped improve the final product.

6. Performance and Optimization

- Analyze the performance of the program. If applicable, provide insights on how the code was optimized for better efficiency or scalability. Mention any trade-offs between performance and functionality, and how these decisions impacted the final solution.

7. Future Work and Improvements

- Suggest potential improvements or extensions to the project. What additional features could be implemented? How could the code be optimized further or scaled for larger applications?

8. Conclusion

- Summarize the key takeaways from the project, including its overall success in meeting the functional requirements and any lessons learned throughout the process. Reflect on how the project's outcomes align with the original goals set out in the assignment.

9. References

- List any resources or references that were used in the development of the project, including external libraries, tutorials, or documentation.