HAN_UNIVERSITY
OF APPLIED SCIENCES

MMR ROS2 Smart_car

Robot Architecture final assignment

THIJS COLBERS, 1659068

MINOR: MOBILE ROBOTICS

DATE: 19-12-2024

Contents

Introduction

The Smartcar ROS2 project focuses on developing an autonomous navigation system for a robotic car using ROS2 and Gazebo simulation. The project's main goal is to create a simulated vehicle capable of navigating autonomously through a virtual environment using wheel odometry and IMU and LIDAR sensors for localization. The created system solves three robotics challenges: accurate self-localization through sensor fusion, path planning around obstacles, and autonomous navigation to set target locations.

The solution is developed through a structured approach, starting with defining the basic robot. Later IMU and LIDAR sensors will be added, as well as wheel odometry and a joint state publisher. These components feed into an EKF node that fuses sensor data to provide accurate localization. The final stage integrates the Nav2 stack to enable the autonomous navigation capabilities. All components are individually tested in the Gazebo simulation before being integrated into the final complete system, resulting in a fully self-driving car system.

Functional requirements

The requirements can be divided into three main categories: General requirements, Robot definition and Simulation & Navigation requirements.

General requirements:

- All packages must successfully be able to build with colcon.
- The final simulation should be launchable using a single launch file.

Robot definition:

- The robot model must follow the structure requirements from section 2.1.2.
- The robot's URDF file must implement the correct dimensions.
- The robot's URDF file must implement the required links:
    - Base and chassis link with mass and inertia.
    - Four wheel links.
    - Two front wheel mount links for steering.
    - IMU and LIDAR link.
- The robot's URDF file must implement the joints correctly:
    - Four wheel joints and Two steering joints.
    - Fixed joints for base, IMU, and LIDAR.
- The IMU and LIDAR sensors must be configured correctly.
- The robot must have successful visualization in both Rviz2 and Gazebo.

Simulation & Navigation:

- The robot must output data from IMU and LIDAR sensors.
- The robot must calculate and publish wheel odometry.
- The robot must use an EKF node to fuse sensor data.
- The robot must be able to:
    - Navigate autonomously between points.
    - Detect and avoid obstacles.
    - Follow a given route.

Code Structure and Design

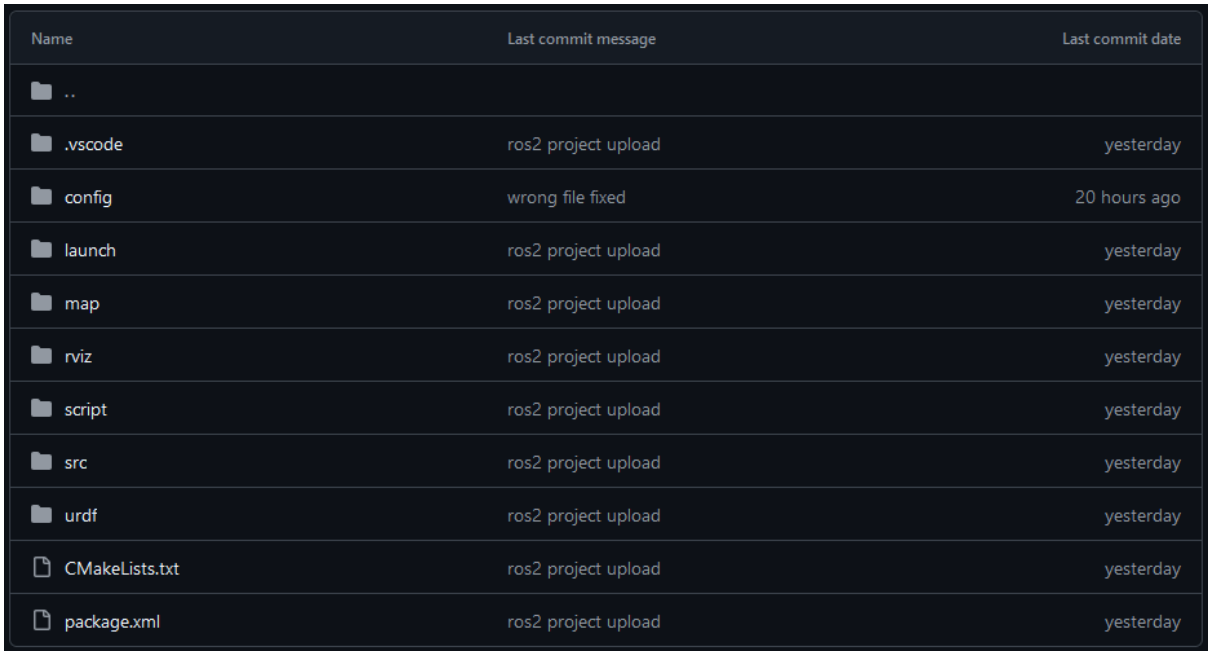
Repository Overview:

The repository mentioned in this chapter, and in all other parts of the report can be found in the supplied zip file on Handin. It can also be found here: https://github.com/ThijsCol/Ros2-MMR

The image below shows an overview of the repository. The repository contains several folders, each folder here is a separate ros2 node. With exception for the 'video' and 'report' folders, which contain the example video and this report. The repository contains the following folders:

- **Ackermann_msgs:** This package was provided for this assignment. It contains the message definitions for Ackermann steering.
- **Car_gazebo_plugin:** This package was provided for this assignment. It contains the smartcar plugin.
- **Gazebo_world:** This package was provided for this assignment. It contains the smalltown world used in Gazebo. A cmakelist and package.xml has been added to build this as a separate node.
- **Nav2_map:** This package was provided for this assignment. It contains the map files for Rviz2. This package will not be built, the map files are copied to the smart_car package.
- **Smart_car:** This package was created for the assignment. It contains the core robot structure and simulation logic. In the next subchapter a more detailed view of this node will be given, as well as an explanation about the key scripts and files.
- **Smartcar.msgs:** This package was created for the assignment. It contains the message definitions for the status of the smartcar.

Smart_car package:



Name	Last commit message	Last commit date
..		
.vscode	ros2 project upload	yesterday
config	wrong file fixed	20 hours ago
launch	ros2 project upload	yesterday
map	ros2 project upload	yesterday
rviz	ros2 project upload	yesterday
script	ros2 project upload	yesterday
src	ros2 project upload	yesterday
urdf	ros2 project upload	yesterday
CMakeLists.txt	ros2 project upload	yesterday
package.xml	ros2 project upload	yesterday

In this image an overview of the smart_car folder can be seen. These folders contain several important files. Some folders are not important and will not be mentioned in the overview. An overview of the folders:

- Config: This folder contains the EKF node .yaml file and nav2 configuration file. These files will be further explained in a next chapter.
- Launch: This folder contains all the launch scripts used during the assignment. The nav2.launch.py is the final launch file used to complete the assignment and launches the complete simulation as required. The other launch files will only launch part of the simulation, for example only the localization.
- Map: This folder contains the map files used for Rviz2.
- Rviz: This folder contains the config file used for Rviz2 during several parts of the assignment.
- Script: This folder contains the joint state publisher and the wheel odometry nodes.
- Urdf: This folder contains the urdf and xacro files. The urdf file is generated from the xacro file. The simulation and nodes use this urdf file.

In the next chapter I will go into detail about the functions of the smart_car package.

Function Documentation:

In this chapter I will go into detail about the nodes and scripts created for the smart_car package.

Joint_state_publisher.py

The **joint_state_publisher.py** node is responsible for managing the wheel positions and steering angles of the smartcar. The node serves as a bridge between the car's control system and its visual representations in Rviz and Gazebo. The node subscribes to the vehicle status message **/smartcar/vehicle_status** and publishes the joint states to **joint_states.** The core functionality is implemented in the **update_wheel_position** function, which handles the wheel rotation and steering updates. The code of this function can be seen below:

```
def update_wheel_position(self, engine_speed, steering_angle):
    wheel_rotation_rate = (engine_speed * 2.0 * 3.14159) / 60.0  # Convert
RPM to rad/s
    current_time = self.get_clock().now()
    dt = 0.02  # 50Hz update

    self.wheel_position += wheel_rotation_rate * dt

    # Create joint state message
    self.msg.header.stamp = current_time.to_msg()
    self.msg.position = [
        self.wheel_position,  # back left wheel
        self.wheel_position,  # back right wheel
        self.wheel_position,  # front left wheel
        self.wheel_position,  # front right wheel
        steering_angle,  # front left steering
        steering_angle  # front right steering
    ]

    # Publish
    self.publisher.publish(self.msg)
```

The function performs several operations:

- Converting the engine speed from RPM to radians per second.
- Updates the wheel positions based on the rotation rate and time.
- Creates a joint state message containing the position of all four wheels and the steering angles of the front wheels.
- It publishes the message so it can be used within ROS2.

The function makes sure that all wheel's actual rotation data gets published and also publishes the steering data. It does this with a 50hz update rate to provide smooth motion and accurate simulation data.

Wheel_odometry_node.py

The **wheel_odometry_node.py** is responsible for tracking the smartcar's position and orientation using measurements from the wheel. The node calculates odometry data by processing the car's engine speed and steering angle. The node subscribes to the **/smartcar/vehicle_status** topic and publishes the odometry data to **/smartcar/wheel/odom.** The core functionality is implemented in the **vehicle_status_callback** and **timer_callback** functions, which handle velocity calculations and position updates. The first part of the function is shown below:

```
def vehicle_status_callback(self, msg):
    # Calculate velocity from RPM
    rpm = float(msg.engine_speed_rpm)
    self.linear_velocity = (rpm * math.pi * self.wheel_diameter) / 60.0

    self.steering_angle = msg.steering_angle_rad

    # Calculate angular velocity
    self.angular_velocity = (self.linear_velocity / self.wheelbase) *
math.tan(self.steering_angle)


def timer_callback(self):
    current_time = self.get_clock().now()
    dt = (current_time - self.prev_time).nanoseconds / 1e9

    # Update position and orientation
    self.phi = self.phi + self.angular_velocity * dt
    self.x = self.x + self.linear_velocity * math.cos(self.phi) * dt
    self.y = self.y + self.linear_velocity * math.sin(self.phi) * dt
```

This function performs the following operations:

- Converting engine RPM to linear velocity using the wheel diameter.
- Calculating the steering angle from radians.
- Calculating the angular velocity based on the steering angle and the car's wheelbase.
- It determines the time intervals for accurate position updates.
- It updates the vehicle's position (x, y) and orientation (phi) using the velocity and time data.

Once the position and orientation are calculated, the data is formatted into an odometry message and then published, this is shown in the following code:

```
# Create and populate odometry message
odom_msg = Odometry()
odom_msg.header.stamp = current_time.to_msg()
odom_msg.header.frame_id = 'odom'
odom_msg.child_frame_id = 'base_link'

# Set position and orientation
odom_msg.pose.pose.position.x = self.x
odom_msg.pose.pose.position.y = self.y
odom_msg.pose.pose.position.z = 0.0

# Convert orientation to quaternion
q = Quaternion()
q.x = 0.0
```

```
q.y = 0.0
q.z = math.sin(self.phi / 2.0)
q.w = math.cos(self.phi / 2.0)
odom_msg.pose.pose.orientation = q

# Set velocity data
odom_msg.twist.twist.linear.x = self.linear_velocity
odom_msg.twist.twist.angular.z = self.angular_velocity
```

This part of the code performs the following operations:

- Creates a timestamped odometry message.
- Sets the calculated position in 3D space (x, y, z).
- Converts the orientation angle to quaternion format for ROS2.
- It includes both the linear and the angular velocity data.
- It prepares the message for publishing.

Ekf.yaml

The **ekf.yaml** configuration file is needed for the Extended Kalman Filter implementation for the smartcar. This file configures how the robot fuses the data from the wheel odometry and IMU sensor to achieve accurate localization. The config file sets the EKF to run at 30Hz and enables 2D mode since the smartcar operates in a two-dimension navigation plane.

The file needs two main sensor inputs:

- The wheel odometry **/smartcar/wheel/odom**, which provides position and velocity data.
- The IMU sensor data **/imu_data**, which provides orientation and acceleration data.

For the wheel odometry the configuration enables X & Y position tracking, Yaw tracking and linear & angular velocity tracking.

For the IMU sensor the configuration enables Roll, pitch and yaw orientation tracking, angular velocity measurements, linear acceleration measurements and it removes the gravitational acceleration from the measurements.

The output data is published to **/odometry/filtered.**

Nav2_params.yaml

The **nav2_params.yaml** configuration file sets up the core parameters for Nav2. It contains the configuration settings for the navigation related functions. The main components configured within this file are:

- AMCL for robot positioning using particle filters.
- Behavior Tree Navigator that forms the main framework of the navigation, using various plugins and actions.
- Controller Server which limits velocities of the smartcar.
- Local and global cost maps for obstacle detection and avoidance.
- Recovery behaviors including spin, backup and wait actions.

The configuration makes sure that the Nav2 stack functions properly. This stack enables the smartcar to navigate autonomously while detecting and avoiding obstacles.

Nav2.launch.py

The **nav2.launch.py** is the main launch file responsible for starting all necessary nodes and components for the smartcar simulation and navigation. The launch file sets up the environment, loads configuration files, and launches various nodes required for navigation. The following section shows how some of the nodes are launched and configured:

```
ld.add_action(Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    name='robot_state_publisher_smartcar',
    output='screen',
    parameters=[{'use_sim_time': True}],
    arguments=[LaunchConfiguration('model')]
))

# joint state publisher
ld.add_action(Node(
    package='smart_car',
    executable='joint_state_publisher.py',
    parameters=[{'use_sim_time': True}]
))

# wheel odometry node
ld.add_action(Node(
    package='smart_car',
    executable='wheel_odometry_node.py',
    parameters=[{'use_sim_time': True}]
))

# Nav2
ld.add_action(IncludeLaunchDescription(
    PythonLaunchDescriptionSource([
        PathJoinSubstitution([nav2_bringup_path, 'launch',
'bringup_launch.py'])
    ]),
    launch_arguments={
        'map': LaunchConfiguration('map_config'),
        'use_sim_time': 'true'
    }.items()
))
```

This section launches the following nodes:

- Robot state publisher
- Joint state publisher
- Wheel odometry node
- Nav2 navigation stack

It also ensures all nodes use the same simulated time for synchronization. And configures the necessary parameters and arguments for each node.

Program demonstration

In order to determine if the smartcar works correctly, a test has been conducted. The user will select an estimated start position, and then set an end position in Rviz. This should result in the car automatically navigating to the end point in Gazebo and Rviz. The car should detect any possible objects in its path and avoid them. This test has been performed and recorder. The video of this test can be found in the repository in the folder 'video'.

Video Explanation

The video can be found in the 'video' folder in the zip file and Github.

The video starts at the Ubuntu desktop with the terminal open. The workspace has been cleaned, meaning that the build, install and log folders are empty. The project is then built using the colcon build command, demonstrating that it builds successfully. Once the build has been completed, the setup files are sourced so they are available for ROS2. Then the nav2.launch.py file is executed, starting all the nodes and parameters. This also opens Rviz and Gazebo. Once everything is loaded, the test can begin.

The test starts in Rviz, where a 2D Pose Estimate is placed on the approximated position and orientation of the smartcar. Then a Nav2 Goal is placed, the goal is placed such that the car will have to navigate around obstacles on the map to reach its destination. As soon as the goal is placed you can see that the car starts driving, this can be seen in both Rviz and Gazebo at the same time. The camera in Gazebo is moved around and zoomed in so that the movement of the car can be seen in more detail. In Rviz the Lidar dots can be seen, scanning and mapping the environment.

While the car is driving towards its destination, a block is placed on its path in gazebo. In Rviz can be seen that the Lidar starts scanning this block, but it is still far away. Once the car gets closer to the block, you can see that the Lidar visualizes the block in Rviz, and the navigation is automatically adjusted to drive around the obstacle. To further confirm the working of the navigation, a cylinder is then placed close in front of the car. You can then immediately see that the navigation gets adjusted, avoiding the obstacle. Finally, you can see the car drive the remaining distance to the Nav2 goal, and stopping once its reached it.

Testing process

In the video explained above, most of the core functionality has been shown working together in a single system. During the project the individual nodes have also been continuously tested while developing. Some of the components can still be tested separately by using the different launch files:

- **nav2.launch.py:** This is the launch file used to test everything at once, as seen in the video.
- **smartcar.launch.py:** Can be used to test the basic visualization of the smartcar, it also launches a 'joint_state_publisher_gui' so you can manually move the joints.
- **gazebo.launch.py:** Can be used to test the correct spawning of the smartcar and world file in gazebo.
- **localization.launch.py:** Can be used to test the localization system, the wheel odometry node, and EKF nodes are used here. As well as spawning in gazebo of the car and world. It will also show the results in Rviz.

Challenges and Solutions

During this assignment I encountered various challenges. In general I had some issues with ROS2, because I didn't have any prior experience with it. Sometimes I would get errors when trying to launch my new launch files, and to me it wasn't clear if the issue was within the launch file or in some other file of the node. This caused me to sometimes spend quite some time trying to fix something, only to later find out the issue was somewhere else and the first version of the file was fine.

Another issue I had was in the beginning with my urdf file. I created an urdf file that was quite modular, it used some of the definitions across the wheels, instead of writing them for every wheel. And this seemed to work great within Rviz. But then later, in Gazebo it didn't work correctly, the wheel and mainly steering joints didn't work as they should. At first, I didn't know that the issue was within the urdf file, causing me to spend some time trying different fixes. Once I found out that the urdf file was the issue, I created a completely new one. This new one is much simpler, but less compact. In this file some of the joint definitions are manually written for every wheel joint. But since this file does work correctly, I decided it is fine, and I could continue with the assignment.

Another issue I faced was the output topic of the EKF node, I was expecting it to be **/odom.** But for some reason it only published to **/odometry/filtered.** No matter what changes I made in the config, I couldn't successfully change the output topic. So, I decided to just rewrite the other files to listen to **/odometry/filtered** instead of **/odom.** This solved the issue.

Performance and Optimization.

The current system has some components where performance could be improved. Mainly the wheel odometry and joint state publisher nodes. These are written in python, and while this is easier for me to write and use, writing/converting these to C++ would be beneficial for performance. C++ should offer better performance without any functional downsides. Since these nodes run continuously, this could result in a relatively big performance gain.

Some nodes a timer of 50Hz has been used, this was chosen as a balance between performance and smooth motion. This timer could be changed to a higher value, like 100Hz. This would result in more frequent updates and smoother motion. This should improve the accuracy of the whole system but will require more CPU load. This value can be changed to find an optimal balance between performance and accuracy.

The system has some optimizations in place for efficiency. The joint state publisher node reuses message instances instead of creating new ones continuously. And the wheel odometry node uses simplified covariance calculations rather than more complex dynamic ones, trading some accuracy for better performance. If better accuracy is required, this could be an area to improve on. Also, the launch file used is structured to load components modularly, which should optimize resource usage during startup.

Future Work and Improvements

As mentioned in the previous chapter some future improvements to the code can be made regarding Performance and Optimization. Looking at more possible code improvements, some nodes have configuration parameters hardcoded in them, like wheel dimensions and update frequency. These parameters can be moved into a separate configuration file, making future adjustments easier and quicker.

Another improvement could be within the wheel odometry node. The kinematic calculations are performed within the timer callback function. This can be improved by creating a separate class to handle the odometry calculations. This would make the code better organized and easier to understand.

Aside from that, one addition that can be made is the use of the ROS2 slam toolbox, adding real-time mapping capabilities. This should improve the accuracy of the car's position within the map in Rviz. Currently, during long runs of the simulation the accuracy slowly decreases as the map and lidar scans drift slowly apart and are not corrected quickly enough. The addition of the slam toolbox could help maintain better alignment between the map and lidar scans.

The speed of the car can also be improved, right now it drives quite slowly. When increasing the speed, it might encounter some issues if objects are placed on its path. The object only gets mapped once it is within a certain distance of the car, if the car moves very fast it might not have enough time to avoid it. This scan distance could then also be increased, to give the car more time to avoid the obstacle and drive around it.

Conclusion

The final smartcar system successfully achieved the primary goal of fully functional autonomous navigation for the car in Rviz and Gazebo. The car successfully detects and avoids obstacles on its path. The system successfully demonstrated the requirements set:

General requirements:

- All ROS2 packages are successfully buildable using colcon build.
- A single launch file launches everything needed for the final simulation.
- The robot is successfully configured with the correct dimensions and structure.
- All the links and joints are correctly configured.
- The car is successfully visualized in both Rviz and Gazebo.

Simulation & Navigation:

- The car integrates both IMU and Lidar data.
- The car has a functional wheel odometry node that calculates the robot's position using kinematic models.
- A correctly working joint state publisher that manages the wheel rotations and steering angles.
- Successful sensor fusion using the EKF node, combining the wheel odometry with the IMU data.
- Correct autonomous navigation with obstacle avoidance.
- Use of the Nav2 stack.

These requirements, and the requirements set in the assignment document have been met by the system.

Before this assignment, I had no prior experience with ROS2. I did have some Python experience which helped with some parts of the assignment. During the project I learned a lot about how the ROS2 system functions, and what is needed to successfully use multiple nodes to achieve a single working system. I learned how to create a car structure using the urdf file, and how to launch the nodes with a launch file. Things like the EKF node and Nav2 stack were also new for me. As well as how to create a working wheel odometry system. At the end I have an increased understanding of the complete system.

References

Robot Architecture final assignment Smart CAR Ros2.pdf from Onderwijsonline

Ros2 gazebo simulation.zip from Onderwijsonline

https://docs.ros.org/en/humble/index.html

https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/URDF-Main.html

https://docs.ros.org/en/humble/Tutorials/Intermediate/RViz/RViz-User-Guide/RViz-User-Guide.html

https://docs.ros.org/en/humble/Tutorials/Intermediate/Launch/Launch-Main.html

https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Tf2-Main.html

https://docs.ros.org/en/humble/Tutorials/Advanced/Simulators/Gazebo/Gazebo.html

http://docs.ros.org/en/melodic/api/robot_localization/html/index.html

https://docs.nav2.org/getting_started/index.html

https://github.com/ThijsCol/Ros2-MMR