

CSCE 452/752 - Project 4a: Group 11

How the program works

Our launch file is used to launch the nodes that allow our project to run. It takes in the command line argument 'robot_file' which is used to determine what robot file we need to get the information about the current robot from. It runs the built-in ROS2 node robot_state_publisher as well as our custom nodes 'simulator_node' and 'velocity_translator_node'. For creating output bag files, we use 'bag_in' and 'bag_out' as additional launch arguments.

Sample command that we use to run our code:

```
ros2 launch project4a start.launch.py robot_file:=project4a/robot_files/bad.robot  
bag_in:=project4a/input_bags/input4 bag_out:=project4a/output_bags/output5
```

1. start.launch.py

The launch file uses the ROS2 launch system to orchestrate our three nodes that contribute to the simulation. The robot's description, including its URDF (Unified Robot Description Format), body parameters, wheel details, and error characteristics, is loaded from a specified robot file.

Key components:

- **Launch Arguments:**

We declare robot_file as a launch argument so that we can use it within the simulator node to run load_disc_robot. Additionally, we were able to access this value within the launch file without using DeclareLaunchArgument. We used sys.argv to grab that value so that we can use it in our robot state publisher node without having to hardcode it. We also have bag_in and bag_out as launch arguments.

- **Load_disc_robot**

We call this function to get the robot's information from the correct robot file so that we can pass it to the other nodes.

- **Robot State Publisher Node:**

This node publishes the robot's state to the ROS2 ecosystem. It utilizes the robot's URDF description for this purpose.

- **Simulator Node:**
This node is responsible for simulating the robot's behavior. It takes parameters such as the robot's description, radius, height, wheel distance, and error characteristics that we get from the robot file.
- **Velocity Translator Node:**
This node translates velocity commands, possibly transforming them to match the simulation's requirements. It also takes parameters related to the robot's description, radius, and wheel distance.
- **play_bag:**
This process is used to play a ROS bag file. It uses the `ros2 bag play` command and takes the `bag_in` argument as input.
- **record_bag:**
It is used to record a ROS bag file using the `ros2 bag record` command and takes the `bag_out` argument as the output file. We record all the topics.

2. `velocity_translator_node.py`

This node subscribes to twist messages with topic `/cmd_vel` and uses the angular and linear velocities received to calculate what the velocities of the left and right wheels of the differential drive robot should be. Once it calculates them it publishes them as `Float64` messages to the `/vl` and `/vr` topics respectively.

Key components:

- The node declares two parameters, 'distance' and 'radius', representing the distance between the wheels of the robot and the radius of the wheels, respectively. Which we get from the robot file.
- The node **subscribes** to the `/cmd_vel` topic, for messages of type **Twist**. When it receives them it calls the **`cmd_vel_callback`** function.
- In the **`cmd_vel_callback`** function, we get the linear and angular velocity then we calculate the `vr` and `vl` that we need. We then publish these on the topics `/vl` and `/vr` with the type **Float64**.

3. `simulator_node.py`:

This node receives the `/vl` and `/vr` topics and uses them to simulate the movement of the robot using `tf` transforms.

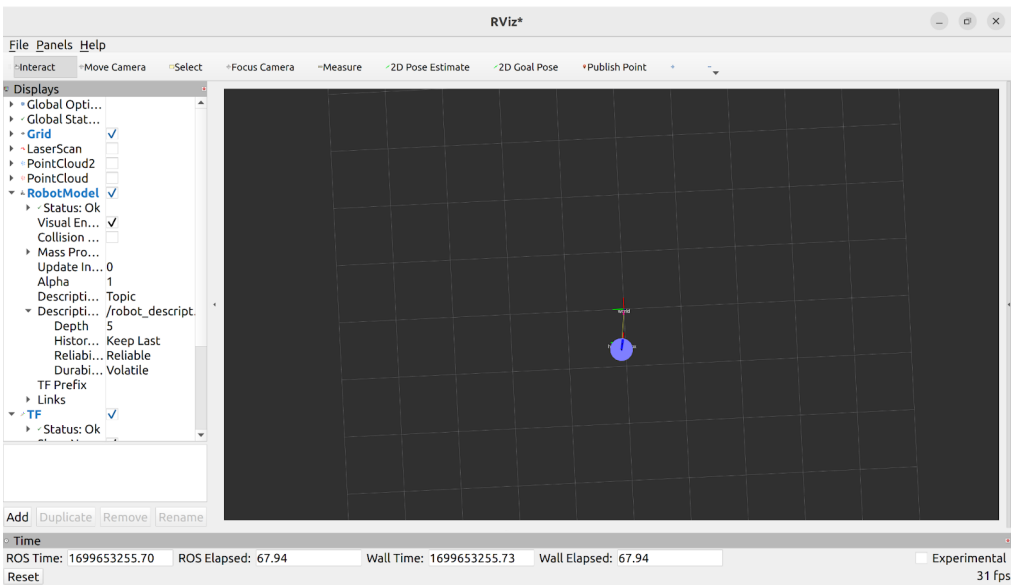
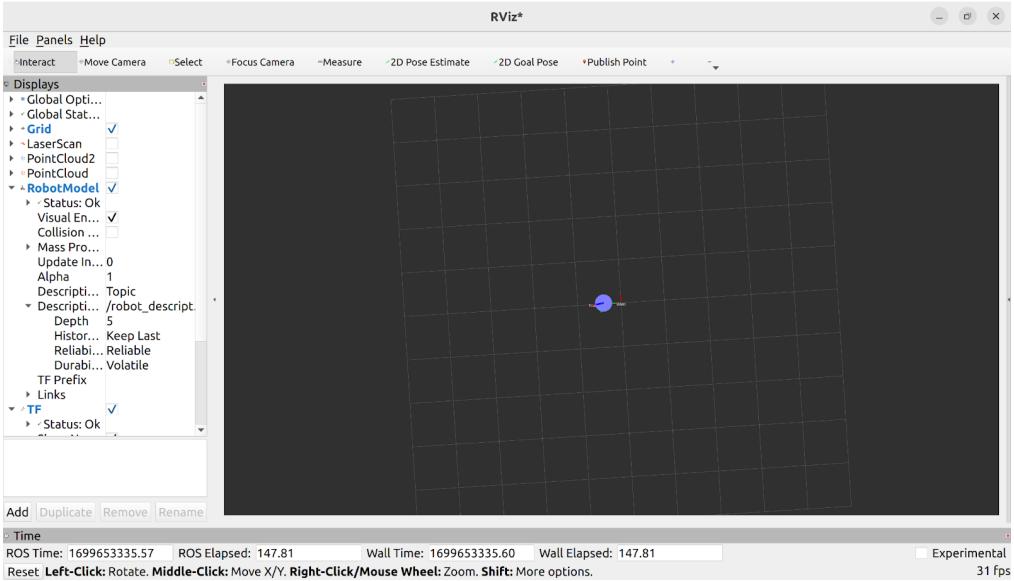
Key components:

- The node subscribes to `/vl` and `/vr` and has a timer with a period of 0.1 that has a callback called **timer_callback**.
- It also declares various parameters for the robot information that are passed in from the launch file.
- When the vl or vr callbacks are triggered they update the **self.vl** and **self.vr** values with the received message's value. If the error is not zero they multiply the velocities by their respective multipliers. Additionally, they update the **self.velocity_update_time**
- The **timer_callback** function handles calling all the other functions within the code. It gets the current time and if it has been more than a second since **self.velocity_update_time** then it sets the velocities to 0. If the time since the last error update is greater than the error update rate it calls **update_errors**. It will then call **update_robot_pose** to get the pose after the current velocities are applied. Then it calls **broadcast_tranform** to actually update the change.
- **Update_errors** calculates the error multipliers from a random gaussian function with the variance from the robot file.
- **Update_robot_pose** figures out the change in pose that would occur from the vl and vr velocities using equations from the differential drive section of our course.
- **Broadcast_transform** calculates the translation and rotation needed to move the robot into its new position and then calls the tf2 broadcaster to do this.

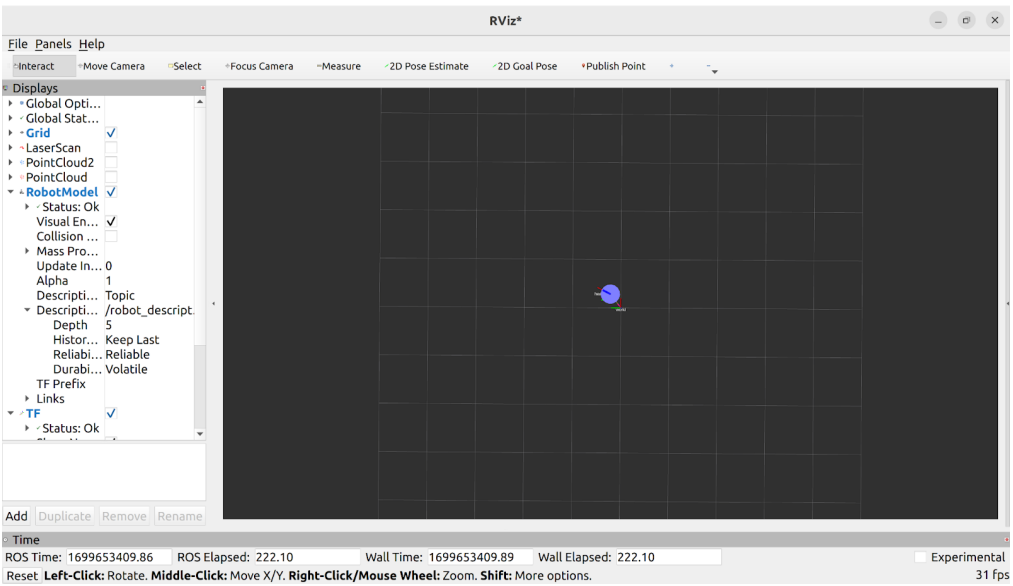
Did the results meet your expectations? Why or why not?

The results did meet our expectations. It was difficult to figure out how to broadcast transforms and how to perform the calculations to find the robot's new location. However, once we were able to get the functionality working it was very straightforward to get the robot to move correctly. We got the robot to move correctly with the input we needed, so we are proud of our final result.

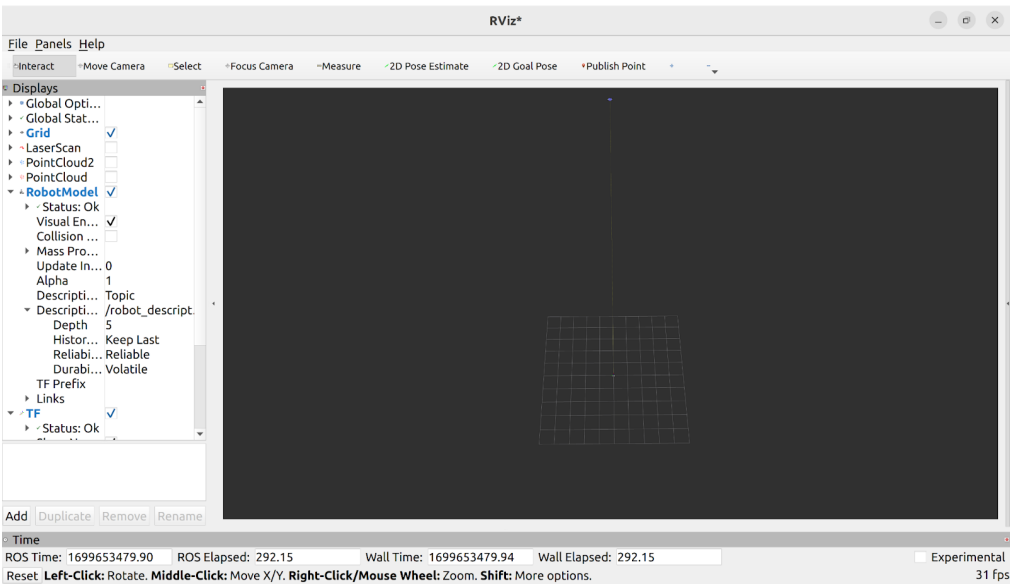
Screenshots for test cases

Test case 1	
Test case 2	

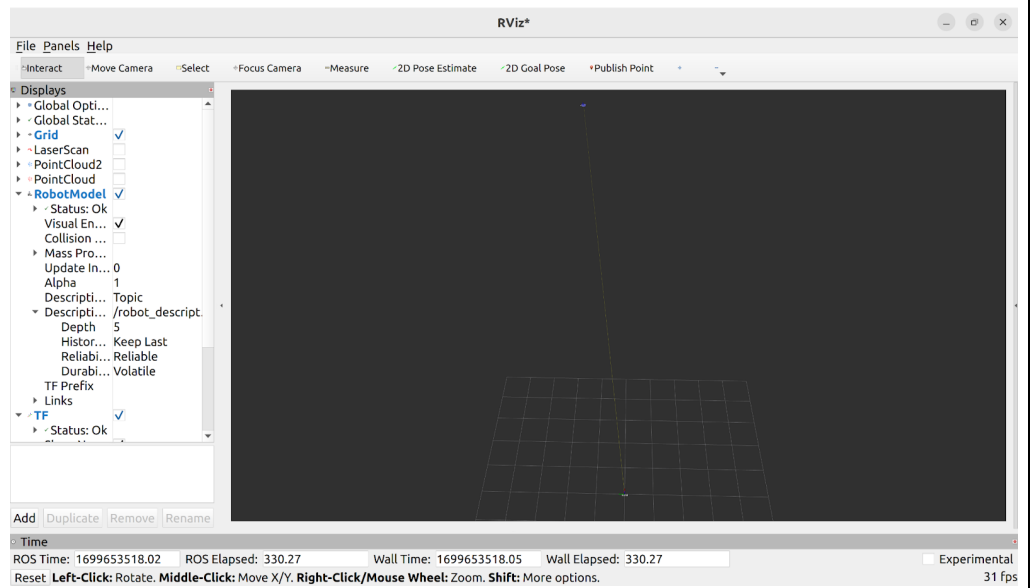
Test case 3



Test case 4



Test case 5



Team Members:

Jonas Land

Ayushri Jain