

CSCE 452/752 - Project 4b: 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 and 'world_file' which is used to determine what world description file we need to use. 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 to play the input bags and record to the output bag.

Sample command that we use to run our code:

```
ros2 launch project4b start.launch.py world_file:=project4b/world_files/brick.world  
robot_file:=project4b/robot_files/normal.robot bag_in:=project4b/input_bags/input5  
bag_out:=project4b/output_bags/output5
```

1. start.launch.py

This launches our 3 nodes that come together to create our simulator. The robot's description is loaded from a specified robot file using the load_disk_robot module. If there are input and output bags included in the launch argument it will play the specified input bag and record to the specified output bag.

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. In the same way, we access our laser parameter values to pass to our simulator node. We also have world_file as a launch argument, bag_in and bag_out as launch arguments using the regular DeclareLaunchArgument method.

- **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. This is a built-in node that we didn't have to modify at all.

- **Simulator Node:**

This node we created 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. It also takes parameters related to the laser like rate, count, angles, ranges, and error characteristics. It subscribes to the topics **/vl** and **/vr** taking these to be the velocities of the left and right wheels of the robot and will periodically broadcast the transform of the robot's location based on these values. It will also periodically broadcast the map of the environment from the world file on the topic **/map**. If the robot moves into an obstacle represented on the map then it will stop instead. Additionally, it will periodically simulate laser scans being sent from the robot based on the robot's current orientation in the map and the laser scan specifications from the robot file on the topic **/scan**.

- **Velocity Translator Node:**

This node translates Twist commands into the relevant velocities of the left and right wheel of a differential drive robot for it to move in the same way the Twist command would tell it to. It takes the robot's description and uses that to interpret what the velocities should be.

- **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 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 and then calculate the `vr` and `vl` 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 robot's movement 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_transform** 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. It calls **is_collision** before actually broadcasting the transform to make certain that the move wouldn't cause the robot to enter any obstacles.
- **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.
- **Get_map_data**
Opens the inputted world file and parses the data within. From the data it gets the initial pose and sets the current **self.pose** to that and it stores the resolution of the

map. Then it iterates through the map characters themselves and creates a 1-dimensional array that can be published as an occupancy grid and 2-dimensional array that can be used in other functions to represent the occupancy grid. It also stores the height and width of the map. It then creates a timer to call `publish_map` every 2 seconds.

- **Publish_map**

Publishes the map as an occupancy

- **Is_collision**

This is called before `update_robot_pose` makes an update. It takes in an updated x and y and returns whether the new coordinates would result in a collision. It does this by checking the points in the 360 range around the center of the robot that are a radius' distance away to see if they intersect with an obstacle.

- **Generate_scan_data**

This is the function that simulates laser scan data from the robot. It gets its specific parameters largely from the passed-in robot file. It will generate as many points as specified which it will intentionally fail to generate if the randomly generated percentage is less than the probability of failure. If it doesn't fail for that scan then starting from the center of the robot it will extend a point out along that particular scan's angle until it reaches an obstacle at which point it stores the distance if that was the smallest distance where an obstacle was seen. After this function is done it will return the scan data.

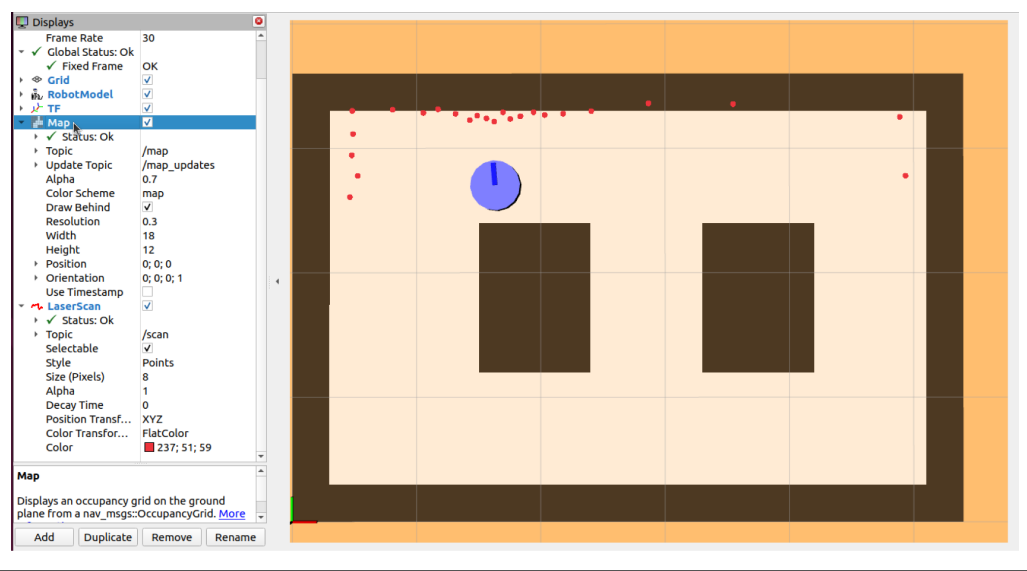
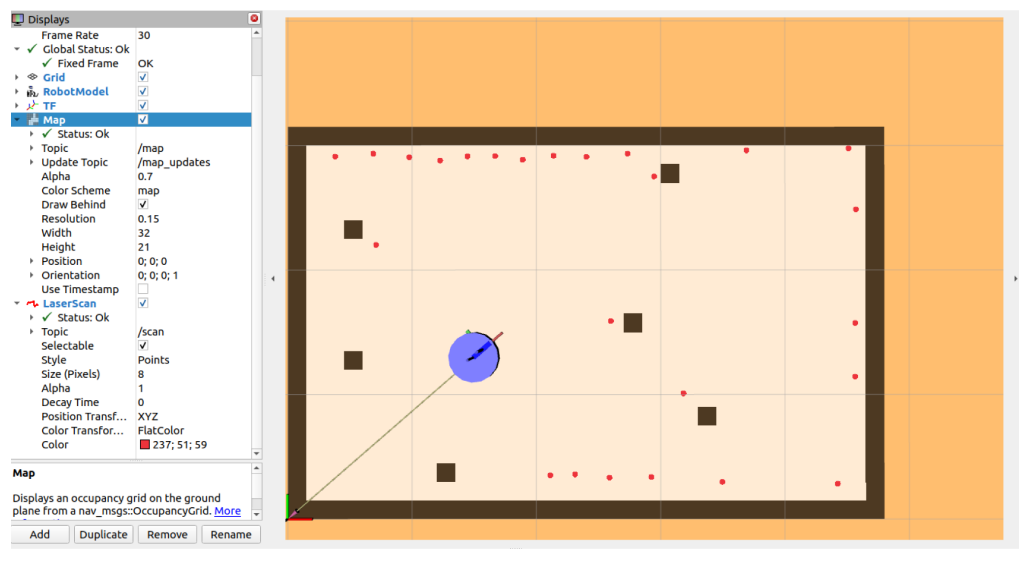
- **Publish_laser_scan**

This calls `generate_scan_data` to get the laser scan data and publish it.

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

The results did meet our expectations. We expected that we would be able to implement all the required functionalities and that publishing the map and detecting collisions would be straightforward while simulating laser scanning would be difficult. All of this proved to be true. We were able to implement all the requirements and sim

Screenshots for test cases

Test case 1	
Test case 2	

Test case 3

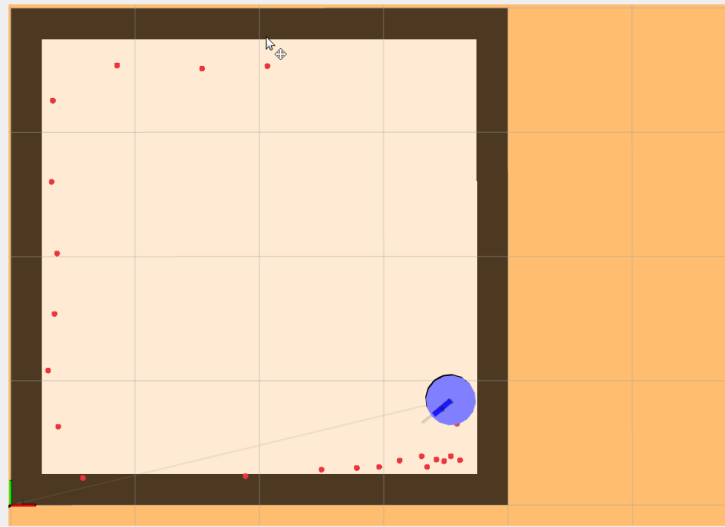
Displays

- Frame Rate: 30
- Global Status: Ok
- Fixed Frame: OK
- Grid: ☒
- RobotModel: ☒
- TF: ☒
- Map: ☒
 - Status: Ok
 - Topic: /map
 - Update Topic: /map_updates
 - Alpha: 0.7
 - Color Scheme: map
 - Draw Behind: ☒
 - Resolution: 0.25
 - Width: 16
 - Height: 16
 - Position: 0; 0; 0
 - Orientation: 0; 0; 0; 1
 - Use Timestamp: ☐
- LaserScan: ☒
 - Status: Ok
 - Topic: /scan
 - Selectable: ☒
 - Style: Points
 - Size (Pixels): 8
 - Alpha: 1
 - Decay Time: 0
 - Position Transf...: XYZ
 - Color Transfor...: FlatColor
 - Color: 237; 51; 59

Map

Displays an occupancy grid on the ground plane from a nav_msgs:OccupancyGrid. [More](#)

Add Duplicate Remove Rename



Test case 4

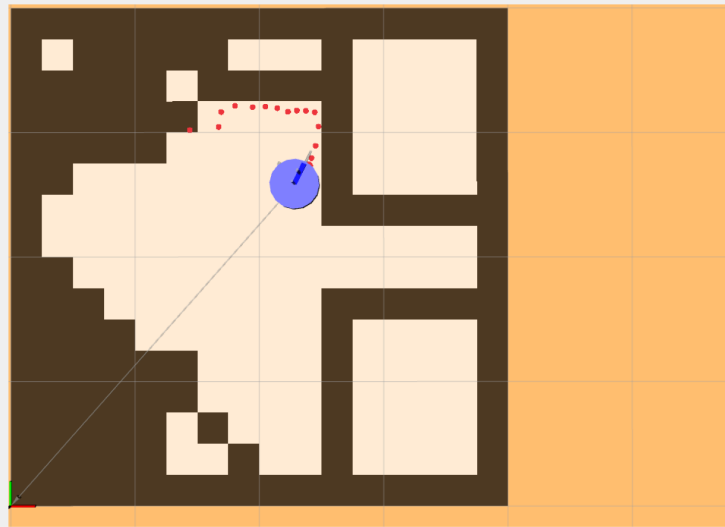
Displays

- Frame Rate: 30
- Global Status: Ok
- Fixed Frame: OK
- Grid: ☒
- RobotModel: ☒
- TF: ☒
- Map: ☒
 - Status: Ok
 - Topic: /map
 - Update Topic: /map_updates
 - Alpha: 0.7
 - Color Scheme: map
 - Draw Behind: ☒
 - Resolution: 0.25
 - Width: 16
 - Height: 16
 - Position: 0; 0; 0
 - Orientation: 0; 0; 0; 1
 - Use Timestamp: ☐
- LaserScan: ☒
 - Status: Ok
 - Topic: /scan
 - Selectable: ☒
 - Style: Points
 - Size (Pixels): 8
 - Alpha: 1
 - Decay Time: 0
 - Position Transf...: XYZ
 - Color Transfor...: FlatColor
 - Color: 237; 51; 59

Map

Displays an occupancy grid on the ground plane from a nav_msgs:OccupancyGrid. [More](#)

Add Duplicate Remove Rename



Test case 5

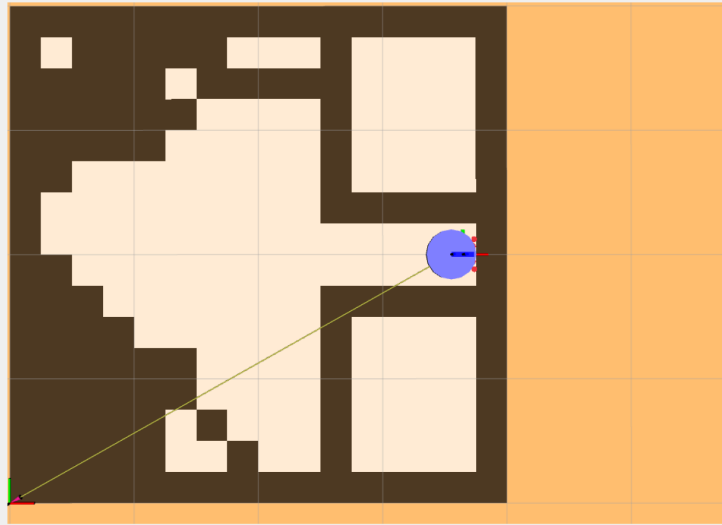
Displays

- Frame Rate: 30
- Global Status: Ok
- Fixed Frame: OK
- Grid: ☒
- RobotModel: ☒
- TF: ☒
- Map: ☒
 - Status: Ok
 - Topic: /map
 - Update Topic: /map_updates
 - Alpha: 0.7
 - Color Scheme: map
 - Draw Behind: ☒
 - Resolution: 0.25
 - Width: 16
 - Height: 16
 - Position: 0; 0; 0
 - Orientation: 0; 0; 1
 - Use Timestamp: ☐
- LaserScan: ☒
 - Status: Ok
 - Topic: /scan
 - Selectable: ☒
 - Style: Points
 - Size (Pixels): 8
 - Alpha: 1
 - Decay Time: 0
 - Position Transf...: XYZ
 - Color Transfor...: FlatColor
 - Color: 237; 51; 59

Map

Displays an occupancy grid on the ground plane from a nav_msgs:OccupancyGrid. [More](#)

Add Duplicate Remove Rename



Test case 6

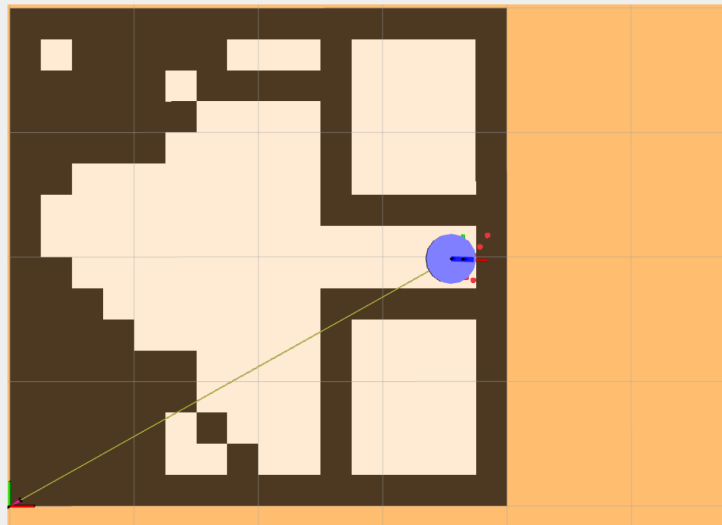
Displays

- Frame Rate: 30
- Global Status: Ok
- Fixed Frame: OK
- Grid: ☒
- RobotModel: ☒
- TF: ☒
- Map: ☒
 - Status: Ok
 - Topic: /map
 - Update Topic: /map_updates
 - Alpha: 0.7
 - Color Scheme: map
 - Draw Behind: ☒
 - Resolution: 0.25
 - Width: 16
 - Height: 16
 - Position: 0; 0; 0
 - Orientation: 0; 0; 1
 - Use Timestamp: ☐
- LaserScan: ☒
 - Status: Ok
 - Topic: /scan
 - Selectable: ☒
 - Style: Points
 - Size (Pixels): 8
 - Alpha: 1
 - Decay Time: 0
 - Position Transf...: XYZ
 - Color Transfor...: FlatColor
 - Color: 237; 51; 59

Map

Displays an occupancy grid on the ground plane from a nav_msgs:OccupancyGrid. [More](#)

Add Duplicate Remove Rename



Team Members:

Jonas Land

Ayushri Jain