## CSCE 452/752 - Project 4c: 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', ‘velocity_translator_node' and ‘navigation_controller_node’.

Sample command that we use to run our code:

```
ros2 launch project4c start.launch.py world_file:=src/project4c/world_files/cave.world
robot_file:=src/project4c/robot_files/normal.robot
```

### 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 is an output bag included in the launch argument it will 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 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.

- **Navigation Controller Node**
  This node subscribes to the laser scans published by the simulator node and moves the robot based off of that. It uses the laserscan to figure out where it should move. It does this reactively and will publish a twist message of where the robot should go.
- **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_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. 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 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 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.

4. **navigation_controller_node.py**
   This node subscribes to the /scan topic and publishes Twist messages on the /cmd_vel topic. It uses these laser scans to reactively get the next move the robot should make. It then publishes the twist message telling the robot where to move.

   Key components:

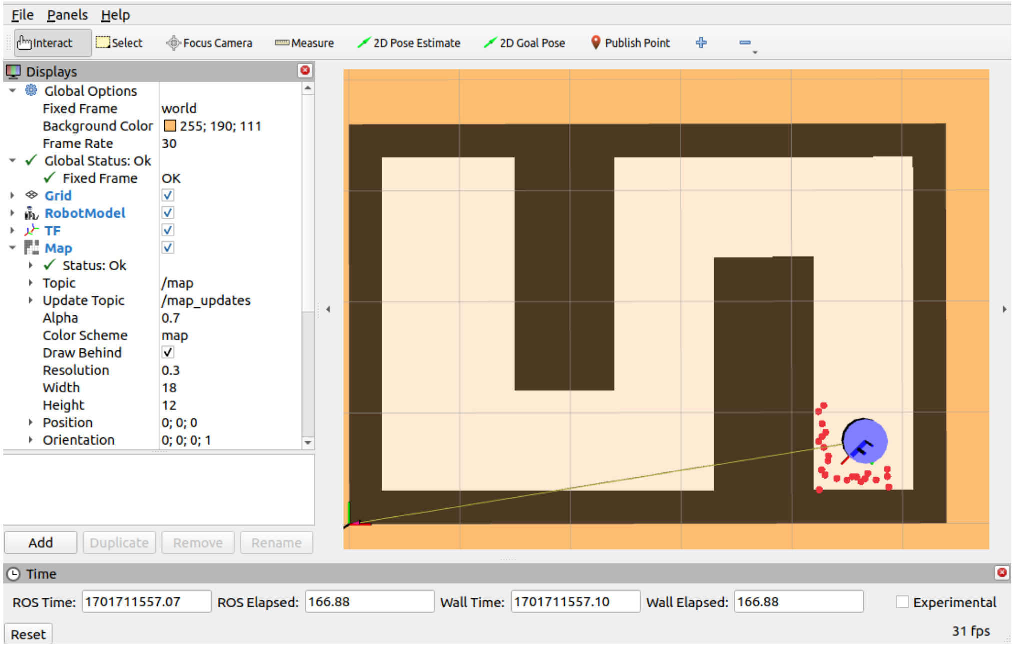
   - Subscribes to LaserScan messages on the /scan topic when it receives them it calls the laser_callback function.
   - It also declares the distance and radius parameters that are passed in from the launch file.
   - We then initialize a safe distance the robot is allowed to be from obstacles as well as scalar multipliers for linear and angular velocities.
   - **Laser_callback**
     The only function in the node.

It loops through all the scans and adjusts the vectors for movement according to whether the robot is at a safe distance from the scans or not. Since it does this for all scans then it avoids the large lines of points we take to be walls. It then publishes the twist message.

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

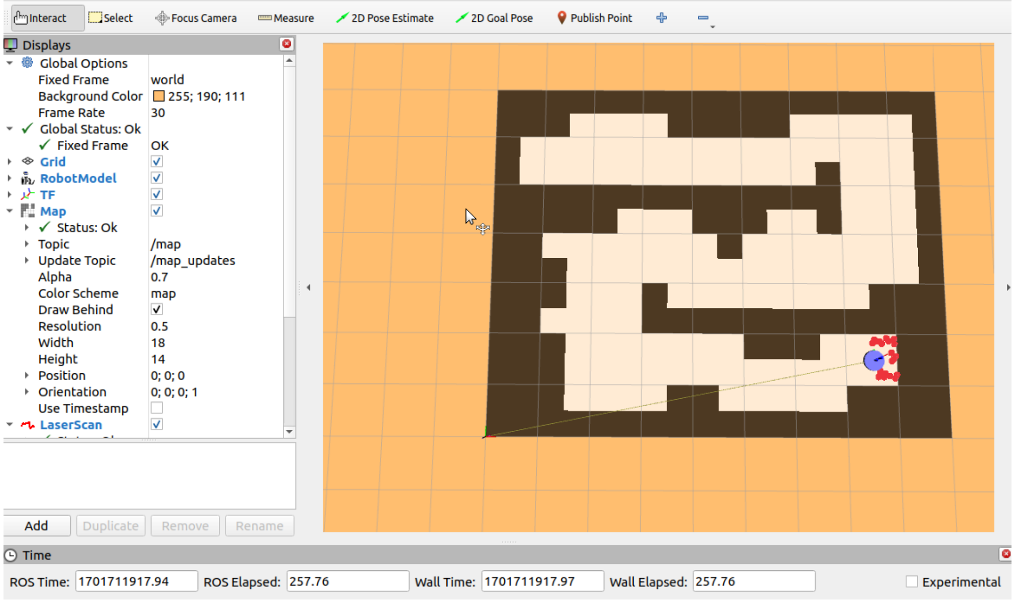
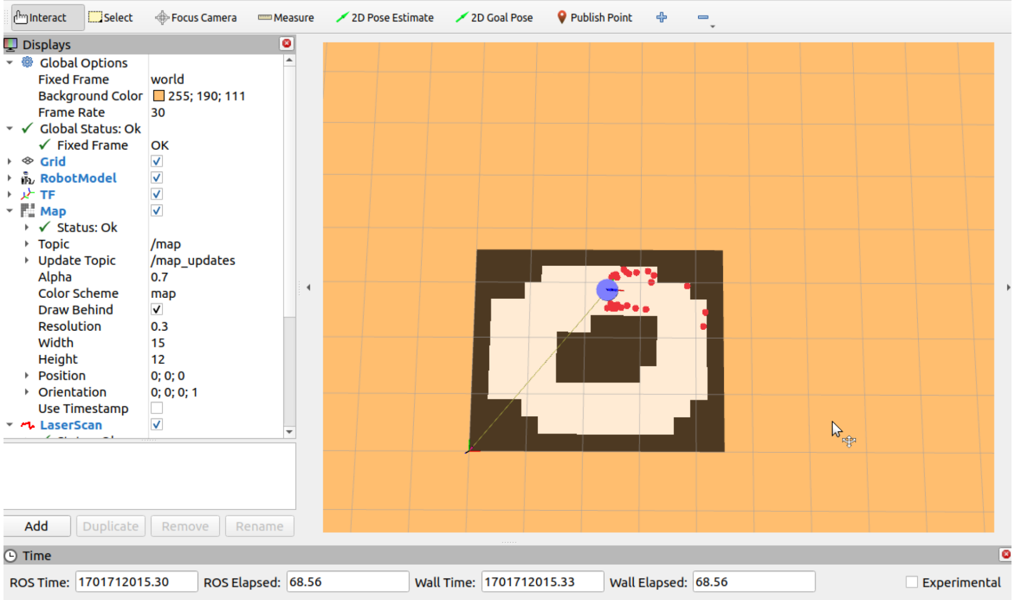
The results did meet our expectations! We expected to be able to complete the reactive controller successfully, to be able to navigate the robot in 3D space what a reactive controller that depended only on the laser scans. It was more straightforward than we had expected and we are proud of the results we obtained.

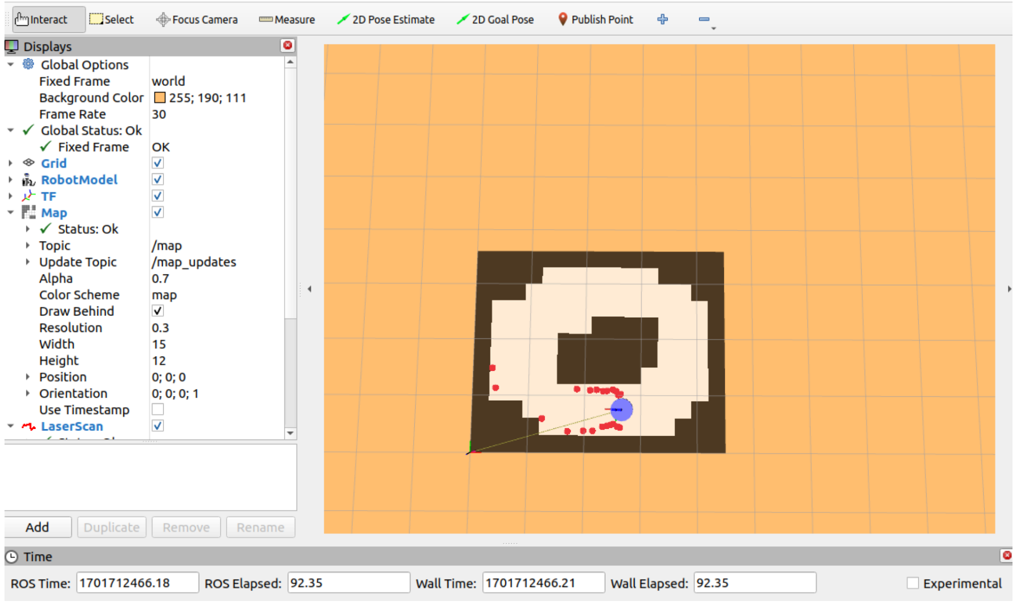
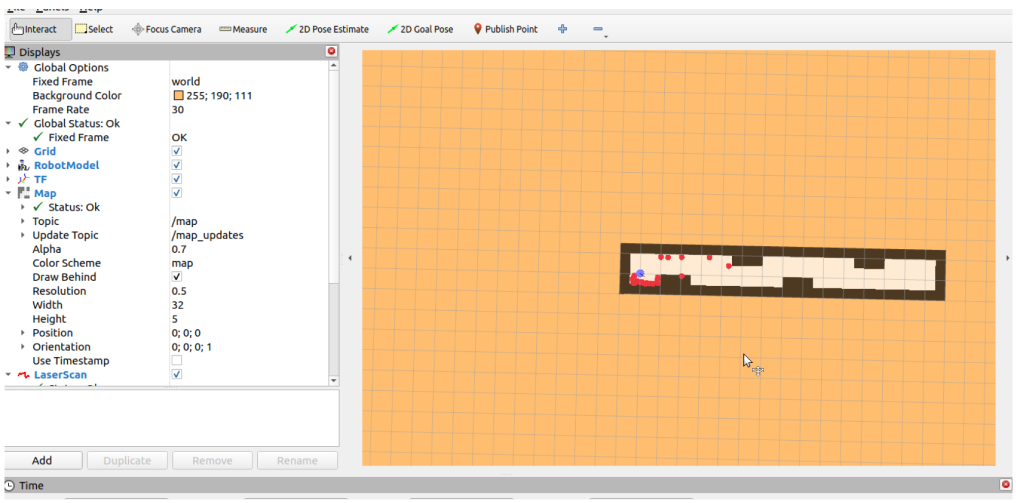
## Screenshots for test cases

| | |
|---|---|
| **Test case 1** |  |

```
Displays
  Global Options
    Fixed Frame        world
    Background Color   255; 190; 111
    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         0.3
    Width              18
    Height             12
    Position           0; 0; 0
    Orientation        0; 0; 0; 1

Time
ROS Time: 1701711557.07   ROS Elapsed: 166.88   Wall Time: 1701711557.10   Wall Elapsed: 166.88   □ Experimental
31 fps
```

| | |
|---|---|
| **Test case 2** | <br> Interact Select Focus Camera Measure 2D Pose Estimate 2D Goal Pose Publish Point<br>Displays<br>Global Options<br>Fixed Frame world<br>Background Color 255; 190; 111<br>Frame Rate 30<br>✓ Global Status: Ok<br>✓ Fixed Frame OK<br>Grid ☑<br>RobotModel ☑<br>TF ☑<br>Map ☑<br>✓ Status: Ok<br>Topic /map<br>Update Topic /map_updates<br>Alpha 0.7<br>Color Scheme map<br>Draw Behind ☑<br>Resolution 0.5<br>Width 18<br>Height 14<br>Position 0; 0; 0<br>Orientation 0; 0; 0; 1<br>Use Timestamp ☐<br>LaserScan ☑<br> Add Duplicate Remove Rename<br>Time<br>ROS Time: 1701711917.94 ROS Elapsed: 257.76 Wall Time: 1701711917.97 Wall Elapsed: 257.76 ☐ Experimental |
| **Test case 3** | Not sure what will the furthest point be in this case as it is is circular, so included 2 screenshots<br>- When robot is directly opposite to where it started from<br>- When robot completed the round and reaches near starting point again<br><br> Interact Select Focus Camera Measure 2D Pose Estimate 2D Goal Pose Publish Point<br>Displays<br>Global Options<br>Fixed Frame world<br>Background Color 255; 190; 111<br>Frame Rate 30<br>✓ Global Status: Ok<br>✓ Fixed Frame OK<br>Grid ☑<br>RobotModel ☑<br>TF ☑<br>Map ☑<br>✓ Status: Ok<br>Topic /map<br>Update Topic /map_updates<br>Alpha 0.7<br>Color Scheme map<br>Draw Behind ☑<br>Resolution 0.3<br>Width 15<br>Height 12<br>Position 0; 0; 0<br>Orientation 0; 0; 0; 1<br>Use Timestamp ☐<br>LaserScan ☑<br> Add Duplicate Remove Rename<br>Time<br>ROS Time: 1701712015.30 ROS Elapsed: 68.56 Wall Time: 1701712015.33 Wall Elapsed: 68.56 ☐ Experimental |

| | |
|---|---|
| | <br> Interact<br>Select<br>Focus Camera<br>Measure<br>2D Pose Estimate<br>2D Goal Pose<br>Publish Point<br>Add<br>Duplicate<br>Remove<br>Rename<br>Displays<br>Global Options<br>Fixed Frame world<br>Background Color 255; 190; 111<br>Frame Rate 30<br>✓ Global Status: Ok<br>✓ Fixed Frame OK<br>Grid ☑<br>RobotModel ☑<br>TF ☑<br>Map ☑<br>✓ Status: Ok<br>Topic /map<br>Update Topic /map_updates<br>Alpha 0.7<br>Color Scheme map<br>Draw Behind ☑<br>Resolution 0.3<br>Width 15<br>Height 12<br>Position 0; 0; 0<br>Orientation 0; 0; 0; 1<br>Use Timestamp □<br>LaserScan ☑<br>Time<br>ROS Time: 1701712466.18 ROS Elapsed: 92.35 Wall Time: 1701712466.21 Wall Elapsed: 92.35 □ Experimental |
| **Test case 4** | <br> Interact<br>Select<br>Focus Camera<br>Measure<br>2D Pose Estimate<br>2D Goal Pose<br>Publish Point<br>Add<br>Duplicate<br>Remove<br>Rename<br>Displays<br>Global Options<br>Fixed Frame world<br>Background Color 255; 190; 111<br>Frame Rate 30<br>✓ Global Status: Ok<br>✓ Fixed Frame OK<br>Grid ☑<br>RobotModel ☑<br>TF ☑<br>Map ☑<br>✓ Status: Ok<br>Topic /map<br>Update Topic /map_updates<br>Alpha 0.7<br>Color Scheme map<br>Draw Behind ☑<br>Resolution 0.5<br>Width 32<br>Height 5<br>Position 0; 0; 0<br>Orientation 0; 0; 0; 1<br>Use Timestamp □<br>LaserScan ☑<br>Time |

**Team Members:**

Jonas Land
Ayushri Jain