# MIE443H1S: Contest 1

# Where am I? Autonomous Robot Exploration of an Environment

## 1.1 Objective:

To use the TurtleBot 4 to autonomously drive around an unknown environment while using the ROS 2 SLAM Toolbox package (a grid-based SLAM method) to dynamically create a map using sensory information from the onboard LiDAR sensor. Your team will be in charge of developing an algorithm for robot exploration that can autonomously navigate an environment using sensor data without human intervention. The robot will need to navigate and explore the environment within a time limit. Team scores will be determined based on the percentage of the environment mapped within the time limit.

## 1.2 Requirements:

The contest requirements are:

- The TurtleBot 4 has a time limit of 8 minutes to both explore and map the environment.
- The robot must perform the overall task in autonomous mode. There will be NO human intervention with the robot.
- You must use sensory feedback on the TurtleBot 4 to navigate the environment. The robot cannot use a fixed sequence of movements without the help of sensors.
- There must be a speed limitation on the robot that will not allow it to go any faster than **0.25m/s** when it is navigating the environment and **0.1m/s** when it is close to obstacles such as walls. This is to ensure safety and consistency in mapping, since higher speeds reduce LiDAR scan overlap with adjacent scans and can increase mapping error.
- Once the 8-minute exploration time is completed (or if your robot is done sooner), you will save the map your TurtleBot 4 has generated and provide this map to the Instructor/TAs.
- The contest environment will be contained within a 4.87x4.87 $m^2$ area with static objects in the environment. An example environment is shown in Figure 1 (not the layout on contest day).

Figure 1: Example Environment

- The exact layout of the environment will not be known to your team in advance of the contest. Therefore, please ensure that your exploration algorithm is robust to unknown environments with static obstacles.

### 1.3 Scoring:

- Your team will be given a total of 2 trials. Each trial will have a maximum duration of 8 minutes. The best trial will be counted towards your final score.
- Scoring is based on the percentage of the environment mapped (10 marks) and the mapping of key obstacles (5 marks) within the environment during exploration. Namely, the map generated by your robot will be compared to a ground truth map of the environment. Figure 2 shows an example of a generated map that would be submitted at the end of the contest. In the case of this map the key obstacles are the garbage cans which can be seen at points 1 and 2.

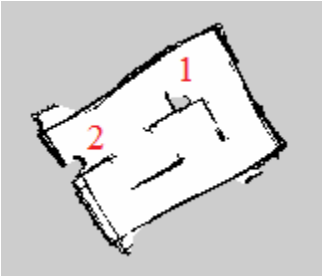


Figure 2: Example Map Generated of the Environment

### 1.4 Procedure:

Code Development - The following steps should be followed to complete the work needed for this contest:

- Download the mie443_contest1.zip from Quercus. Then extract the contest 1 package (right click > Extract Here) and then move the mie443_contest1 folder to the ros2_ws/src folder located in your home directory.
- In terminal, change your current directory to ~/ros2_ws using the cd command. Then run

  ```
  colcon build --packages-select mie443_contest1
  ```

  to compile the code to make sure everything works properly. ****Please note if you do not run this command in the correct directory a new executable of your program will not be created.****
- Refer to the mie443_contest1/src/contest1.cpp file. The majority of development for this contest will be done in the contest1.cpp.
- Code Breakdown – contest1.cpp
  - These are the header declarations that allow you to use the required libraries in this contest:

```cpp
#include <chrono>
#include <memory>
#include <cmath>
#include <map>
#include <vector>
#include <algorithm>

#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/twist_stamped.hpp"
#include "sensor_msgs/msg/laser_scan.hpp"
#include "nav_msgs/msg/odometry.hpp"
#include "irobot_create_msgs/msg/hazard_detection_vector.hpp"
#include "tf2/utils.h"
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>

using namespace std::chrono_literals;
```

- Sensor Implementation
  - These are the callback functions that are run whenever the subscribers that are declared in the main block of code are triggered. The variable msg is an object that contains the data pertaining to the instance of each respective callback press:

```cpp
    void laserCallback(const sensor_msgs::msg::LaserScan::SharedPtr scan)
    {
        // implement your code here
    }

    void odomCallback(const nav_msgs::msg::Odometry::SharedPtr odom)
    {
        // implement your code here
    }

    void hazardCallback(const
irobot_create_msgs::msg::HazardDetectionVector::SharedPtr hazard_vector)
    {
        // implement your code here
    }
```

  - Main function and ROS initialization are provided below:

```cpp
int main(int argc, char** argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<Contest1Node>();
```

- Here we create one publisher and three subscribers. The publisher vel_pub will be used to advertise robot velocity commands to the ROS 2 framework. The three subscribers are used to return sensor data from (1) the laser scan information from the LiDAR sensor, (2) the bumper on the robot base and (3) the odometry from the wheeled encoders and inertial measurement unit (IMU).

```
        vel_pub_ =
this->create_publisher<geometry_msgs::msg::TwistStamped>("/cmd_vel", 10);

        laser_sub_ = this->create_subscription<sensor_msgs::msg::LaserScan>(
            "/scan", rclcpp::SensorDataQoS(),
            std::bind(&Contest1Node::laserCallback, this,
std::placeholders::_1));

        hazard_sub_ =
this->create_subscription<irobot_create_msgs::msg::HazardDetectionVector>(
            "/hazard_detection", rclcpp::SensorDataQoS(),
            std::bind(&Contest1Node::hazardCallback, this,
std::placeholders::_1));

        odom_sub_ = this->create_subscription<nav_msgs::msg::Odometry>(
            "/odom", rclcpp::SensorDataQoS(),
            std::bind(&Contest1Node::odomCallback, this, std::placeholders::_1));
```

- Declaration of speed variables and the TwistStamped Class. TwistStamped is a type of descriptor that defines the velocities of a robot in 6 Degrees-of-Freedom. Three degrees for linear vectors XYZ and 3 degrees for vectors for Yaw, Pitch, and Roll. For TurtleBot 4, we only need to define the linear X and angular Yaw velocities. The current time is also required to be set for each TwistStamped message.

```
        geometry_msgs::msg::TwistStamped vel;
        vel.header.stamp = this->now();
        vel.twist.linear.x = linear_;
        vel.twist.angular.z = angular_;
```

- Controller Design and Implementation
  - The following starts the timer that runs at 10 Hz for the main control loop.

```
        // Timer for main control loop at 10 Hz
        timer_ = this->create_wall_timer(
            100ms, std::bind(&Contest1Node::controlLoop, this));
```

  - Inside the control loop, the first few lines keep track of the total amount of time the robot has spent exploring the environment.

```
    void controlLoop()
    {
        // Calculate elapsed time
        auto current_time = this->now();
        double seconds_elapsed = (current_time - start_time_).seconds();
```

- The program remains within this loop until 480 seconds have elapsed. You should implement your exploration code here to decide a linear and angular velocity for the robot at each timestep. The angular and linear values are used to set the respective speeds in the TwistStamped class declared in the previous block. The velocity command is then published to the ROS 2 framework and is used by the TurtleBot 4 base to drive the robot.

```
        // Check if 480 seconds (8 minutes) have elapsed
        if (seconds_elapsed >= 480.0) {
            RCLCPP_INFO(this->get_logger(), "Contest time completed (480 seconds).
Stopping robot.");

            // Stop the robot
            geometry_msgs::msg::TwistStamped vel;
            vel.header.stamp = this->now();
            vel.twist.linear.x = 0.0;
            vel.twist.angular.z = 0.0;
            vel_pub_->publish(vel);

            // Shutdown the node
            rclcpp::shutdown();
            return;
        }

        // Implement your exploration code here

        // Set velocity command
        geometry_msgs::msg::TwistStamped vel;
        vel.header.stamp = this->now();
        vel.twist.linear.x = linear_;
        vel.twist.angular.z = angular_;

        // Publish velocity command
        vel_pub_->publish(vel);
```

- Every time any file in your project is changed, you must compile it from terminal in the ros2_ws directory using the colcon build command. If not, the executable will not contain your modifications.

- Robot Simulation in Gazebo – You can use Gazebo to implement and test your TurtleBot 4 code prior to integrating it onto the physical robot. The following steps should be followed in order to do this:

    - Begin by launching the simulated world through the following command:

      ```
      ros2 launch turtlebot4_gz_bringup turtlebot4_gz.launch.py
      model=:lite world:=maze
      ```

    - The SLAM Toolbox is a 2D SLAM algorithm that performs both localization and mapping simultaneously, providing features such as loop closure, pose graph optimization, and continuous map updates as the robot moves. To run the SLAM Toolbox algorithm, use the following command in a new terminal:

      ```
      ros2 launch turtlebot4_navigation slam.launch.py
      ```

      or terminate the previous Gazebo simulation by pressing CTRL+ C, and run the following command:

      ```
      ros2 launch turtlebot4_gz_bringup turtlebot4_gz.launch.py
      model=:lite world:=maze slam:=true
      ```

      All other commands to view and save the map are the same as in the mapping tutorial.

    - To visualize robot information (i.e., position, sensor reading), run the following command in a separate terminal window:

      ```
      ros2 launch turtlebot4_viz view_navigation.launch.py
      ```

      or terminate all running commands and only run the following command:

      ```
      ros2 launch turtlebot4_gz_bringup turtlebot4_gz.launch.py
      model=:lite world:=maze slam:=true viz:=true
      ```

    - Finally, to run your code in simulation, it is the same as running it on the physical TurtleBot 4:

      ```
      ros2 run mie443_contest1 contest1
      ```

- Robot Execution - When you are ready to run your program on the physical TurtleBot 4, the following steps should be followed:
    - Place the TurtleBot 4 on the dock if it is in storage mode (when all lights are off) to turn on the robot.
    - Open a terminal, run the undock command below or manually remove the robot from the dock. Note, if you run the undock command, the robot would undock itself.

      ```
      ros2 action send_goal /undock irobot_create_msgs/action/Undock {}
      ```

    - When you are ready to begin mapping, run the following command:

      ```
      ros2 launch turtlebot4_navigation slam.launch.py
      ```

- This command opens a plugin that visualizes the data that the robot is publishing and subscribing to.

```
ros2 launch turtlebot4_viz view_navigation.launch.py
```

- Finally, to execute your code, run the following line in a new terminal:

```
ros2 run mie443_contest1 contest1
```

- If all is working correctly, your robot will begin navigating the environment and collecting map data.
- At the end of the allotted contest time or if you are finished before the contest time, **save the map** you have created by running the following command in a new terminal (press tab to auto-complete the command):

```
ros2 service call /slam_toolbox/save_map
slam_toolbox/srv/SaveMap "name:

  data: 'map_name'"
```

- When you are done, cancel all processes by pressing Ctrl-C in their respective terminals, prior to closing them.

### 1.5 Report:

- The report for each contest is worth half the marks and should provide detailed development and implementation information to meet contest objectives (15 marks).
- Marking Scheme:
  - The problem definition/objective of the contest. (1 mark)
    - Requirements and constraints
  - Strategy used to motivate the choice of design and winning/completing the contest within the requirements given. (2 marks)
  - Detailed Robot Design and Implementation including:
    - Sensory Design (4 marks)
      - Sensors used, motivation for using them, and how are they used to meet the requirements including functionality.
    - Controller Design, both low-level and high-level control (including architecture and all algorithms developed) (5 marks)
      - Architecture type and design of high-level controller used (adapted from concepts in lectures)
      - Low-level controller
      - All algorithms developed and/or used
      - Changes and additions to the existing code for functionality
  - Future Recommendations (1 mark)
    - What would you do if you had more time?
    - Would you use different methods or approaches based on the insight you now have?

    - Full C++ ROS 2 code (in an appendix). Please do not put full code in the text of your report. (2 marks)
    - Contribution of each team member with respect to the tasks of the particular contest (robot design and report). You can use an Attribution Table for this. (Towards Participation Marks)
    - The contest package folder containing your code will also be submitted alongside the softcopy (PDF version) of your report. (Towards the Code Marks Above)
- Your report should be a maximum of 20 pages, single spaced, font size 12 (not including appendix).

**1.6 Reference Materials for the Contest**:

The following materials in Appendix A will be useful for Contest 1:

- A.1.1
- A.1.2