Autonomous Robotics: ROS Project 3

Arwed Paul Meinert

January 4, 2025

1. Grade 1

After the files have been copied to their respective destinations, the Robot Operating System (ROS) core must be executed. Next, the ROS console should be opened, and the two commands shown below should be run. First, the correct robot used in the environment is specified. Then, the Gazebo environment is started in the appropriate world.

```
export TURTLEBOT3_MODEL=burger
roslaunch turtlebot3_gazebo turtlebot3_myWorld.launch
```

1.1 Moving Thru the Maze without Touching Walls

The robot is equipped with sensors positioned around it, which scan in all directions in 1° increments. This generates a vector with 360 distance values, starting from 0°, directly in front of the robot. Each value represents the distance from the robot to an object. If there is no object within the sensor's maximum range, the return value is `inf`. Additionally, both the angular and linear speeds of the robot can be controlled.

To retrieve and transmit all necessary information to and from the robot, Matlab is used. The code to control the robot is shown in Lst. 1.1. First, the Internet Protocol (IP) address of the ROS environment is defined. To ensure ROS is not already connected, all connections are shut down first. If no connections exist, this has no effect on the program.

Afterward, the ROS connection is initialized using the IP address. Additionally, the `velPub` publisher is set up to publish velocity data to the correct topic to control the robot. Similarly, the `laserSub` subscriber is defined to listen to the `\scan` topic, which contains the sensor data. Finally, the minimal distance from any object to the robot is defined.

Once all the variables are initialized, the program enters a loop (Lst. 1.1, line 7). This loop continuously executes the following code. The messages are stored in individual variables, including the data structure for the velocity message and the contents of the laser messages. The ranges are extracted in Lst. 1.1, line 10. To ensure the robot does not collide with any objects, only the sensor in front of the robot is considered when it is moving forward. Therefore, the distance of the laser in front of the robot is checked. If the distance is greater than the `min_distance`, the robot moves forward. If the distance is smaller, the robot rotates without moving. To ensure the robot keeps moving forward most of the time, 90° cones on the front left and front right of the robot are evaluated. The sum of all distances within the laser range (values smaller than `inf`) is calculated. If the sum of all values from the front right is smaller than that from the front left, it indicates that objects to the right are closer than those on the left, resulting in a left turn. Conversely, when there are more objects to the left than to the right, the robot turns right. If both values are equal, the robot moves forward without turning. Since the robot stops when the `min_distance` is undercut, forward movement halts. It then only turns until the `min_distance` is no longer undercut, preventing the robot from colliding with objects.

At the end of the loop, the `velMsg` is sent to the `velPub` publisher, which is connected to the robot in the ROS environment. The loop repeats every 0.5 seconds. Since the `min_distance` is 1 m and the velocity is $0.2\,\frac{\mathrm{m}}{\mathrm{s}}$, the robot will never collide with any obstacle.

```matlab
ip = "http://192.168.237.129:11311"; % Modify with your ROS
    Master URI
rosshutdown;
rosinit(ip);
velPub = rospublisher('/cmd_vel', 'geometry_msgs/Twist');
laserSub = rossubscriber('/scan', 'sensor_msgs/LaserScan');
min_distance=1;
while true
velMsg = rosmessage(velPub);
laserMsg = receive(laserSub, 10); % Wait up to 10 seconds for
     a message
ranges = laserMsg.Ranges; % Distance readings from the laser
    scanner

% Filter out 'inf' values before summing
filtered_right = ranges(1:90);
filtered_right = filtered_right(~isinf(filtered_right)); %
    Remove 'inf' values
sum_right = sum(filtered_right);

filtered_left = ranges(270:360);
filtered_left = filtered_left(~isinf(filtered_left)); %
    Remove 'inf' values
sum_left = sum(filtered_left);

if ranges(1)>min_distance
    velMsg.Linear.X = 0.2; % Linear velocity in m/s
else
    velMsg.Linear.X = 0.0; % Linear velocity in m/s
end

if sum_right<sum_left
    velMsg.Angular.Z = -0.2; % Angular velocity in rad/s
elseif sum_left<sum_right
    velMsg.Angular.Z = 0.2; % Angular velocity in rad/s
```

```
else
    velMsg.Angular.Z = 0; % Angular velocity in rad/s
end
% Example: Move the robot forward
send(velPub, velMsg);
pause(0.5);
end
```

Listing 1.1: Moving the Robot thru the maze without touching walls.

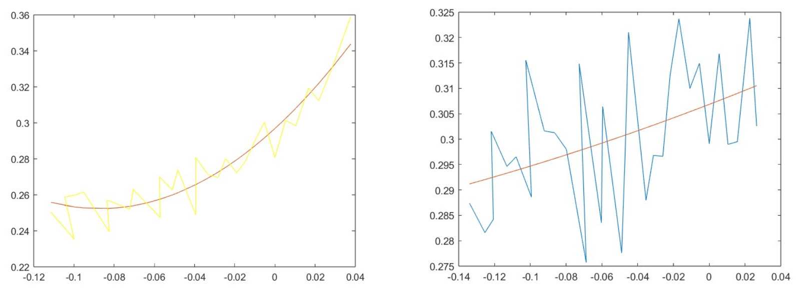
1.2 Finding the Cone

In this task, the robot is required to navigate through the maze and locate a cone-shaped object. For this task, either the solution in Lst. 1.1 or the solution in Lst. 2.1 can be used. Since the robot moves more or less randomly without any specific direction in the code in Lst. 1.1, the function is used with the moving function implemented in Lst. 2.1. As the robot always moves with the maze wall to its right, it reliably encounters the cone within a short time. The complete program for this solution is shown in App. A. The only change to the main moving function is that the function in Lst. 1.2 is called, and it must return true two times in a row. This helps prevent false positives and is necessary due to the noisy sensor data.

The function in Lst. 1.2 receives the sensor values from the right side of the robot, as the robot moves along the left side of the obstacles. Additionally, the number of points to be evaluated is passed to the function. Due to the noisy sensor readings, this number needs to be relatively high.

The function identifies the smallest distance from the robot to any obstacle. It then generates a range where it attempts to fit a curve by taking the index of the smallest distance and adding and subtracting half of the number of points passed to the function. The distances are picked based on the index range. Afterwards, it converts the polar coordinates (the angle and the distance) into Cartesian coordinates. These coordinates are used to create a quadratic polynomial that best describes the Cartesian points.

If the closest object to the robot is a wall, the first constant a_1 of the polynomial $a_1x^2 + a_2x + a_3 = y$ should be small since the points form a straight line (see Fig. 1.1b). If the robot is near a cone, the value of a_1 should be higher, as the surface is curved (see Fig. 1.1a). If the robot is near an edge, the value is very high, as the points form a step. Both cases are shown in Fig. 1.1. The yellow line indicates that the robot has detected a cone two times in a row and displays the sensor readings converted to Cartesian coordinates. The blue line in Fig. 1.1b shows that no cone was detected based on the sensor readings. The red line in both figures represents the second-order polynomial that best describes the sensor readings. It can be observed that, in Fig. 1.1a, the curvature is higher compared to the mostly linear curve in Fig. 1.1b. This distinction is made in Lst. 1.2, line 33. If the



(a) Cone found (b) No cone found

Figure 1.1: Identification algorithm for cones.

curvature is greater than 5 but smaller than 10, the robot is very likely to be near a cone. However, due to noisy sensor values, the object is only detected as a cone when two consecutive sensor readings lead to this conclusion. The detection accuracy could be improved by lowering the curvature threshold and increasing the number of consecutive cone detections required.

```
1  %nComments done by Chat GPT
2  function [isCone] = identify_cone(sensor, points_to_check)
3      %Comments done by chat GPT
4      isCone = false; % Default: not a cone
5      % Find the index of the closest object
6      [~, min_index] = min(sensor);
7  
8      % Extract region of interest (ROI)
9      start_index = max(1, min_index - ceil(points_to_check /
           2));
10     end_index = min(length(sensor), min_index + ceil(
           points_to_check / 2));
11     selected_distances = sensor(start_index:end_index)';
12 
13     % Remove invalid data points
14     if any(isinf(selected_distances))
15         return;
16     end
17 
18     % Convert polar coordinates to Cartesian coordinates
19     angles = deg2rad(start_index:end_index);
20     x = selected_distances .* cos(angles);
```

```
    y = selected_distances .* sin(angles);

    % Analyze geometric properties
    % Fit a second-order polynomial (quadratic curve)
    if length(x) < 3
        % Not enough points for curvature analysis
        return;
    end
    p = polyfit(x, y, 2); % Quadratic fit

    % Check curvature
    curvature = abs(p(1)); % Curvature is proportional to the
        quadratic coefficient
    if curvature > 5 && curvature<10
        isCone = true;
    end
end
```

Listing 1.2: Finding the cone based on the sensor values.

2. Grade 2 and 3

2.1 Finding the Exit by Staying at the Right Wall

Lastly, the robot should exit the maze without using mapping algorithms. The simplest way to achieve this is by selecting a wall (e.g., the left or right wall) at the entrance of the maze and then consistently staying near that wall. This guarantees that an exit will eventually be found.

The code used to implement this task is shown in Lst. 2.1. The initial structure is the same as in the other tasks, where the topics and variables are defined. The control in this case is achieved using a proportional controller. The parameters for it are defined before the control loop, including the desired distance to the wall, the minimal distance, and the proportional factor `k_p`. This factor is used to scale the angular velocity while the robot moves next to a wall.

Additionally, the cones are defined. The front cone spans from 315° to 45°. The minimal distance in front is determined, and the same is done for the distance to the right. This is defined by a cone of 90°, starting at 225° from the robot.

To scale the angular velocity, the error between the desired distance and the distance to the right is calculated. If the error is 0, the robot moves straight without turning. If the error is negative, the robot turns right to decrease the distance. Conversely, if the robot is too close to the object, it turns in the other direction. This logic is implemented in Lst. 2.1, lines 30-46, with the addition of some edge cases. One of these cases handles the situation when the distance in front of the robot is less than the minimal distance. In this case, the robot turns until the distance becomes safe again. This prevents the robot from colliding with walls when the distance is too short for the proportional controller to correct the heading.

The parameters for the minimal distance and desired distance might need to be adjusted depending on the maze. Since the robot tries to maintain the desired distance, if the maze contains passages narrower than the desired distance, the robot will not move through those passages. This can be seen with the cone: the robot will not move between the cone and the wall, as this distance is too small. In this case, this behavior is beneficial because it indicates the robot is not on the correct path. In other cases, however, it may be necessary to adjust some parameters.

```
1 %nComments done by Chat GPT
2 ip = "http://192.168.237.129:11311";
3 rosshutdown;
4 rosinit(ip);
```

```
velPub = rospublisher('/cmd_vel', 'geometry_msgs/Twist');
laserSub = rossubscriber('/scan', 'sensor_msgs/LaserScan');
% Desired wall distance
desired_distance = 0.5; % Adjust based on your environment (in meters)
min_distance = 0.2; % Minimum safe distance to avoid collisions

while true
velMsg = rosmessage(velPub);
laserMsg = receive(laserSub, 10); % Wait up to 10 seconds for a message
ranges = laserMsg.Ranges; % Distance readings from the laser scanner
% Parameters
desired_distance = 1; % Desired distance from the wall (meters)
min_distance = 0.4; % Minimum safe distance to avoid collisions (meters)
k_p = 1.0; % Proportional gain for angular velocity (tune as needed)
k_p_linear = 0.5; % Proportional gain for linear velocity (tune as needed)

% Get distances from laser scan
front_indices = [1:45, 315:360]; % Combine indices for front cone (0 to 45 and 315 to 360)
front_distance = min(ranges(front_indices)); % Minimum distance in the front region
[right_distance, right_index] = min(ranges(225:315)); % Left sensor (225 to 315)

% Calculate the error (distance to wall deviation)
error = desired_distance - right_distance;

% Logic for wall-following with proportional control
if front_distance < min_distance
    % Obstacle ahead: Stop and turn left to avoid collision
    velMsg.Linear.X = 0.0; % Stop
    velMsg.Angular.Z = 0.5; % Turn left
elseif right_distance < desired_distance && right_distance > min_distance
    % Wall is within an acceptable range: move forward and adjust turning rate
    velMsg.Linear.X = 0.2 + k_p_linear * error; % Adjust forward speed proportionally
    velMsg.Angular.Z = -k_p * error; % Adjust turning rate proportionally
elseif right_distance >= desired_distance
```

```
    % Wall is too far: Turn left to get closer
    velMsg.Linear.X = 0.2; % Move forward
    velMsg.Angular.Z = k_p * error; % Turn left
else
    % Wall is too close: Turn right to move away
    velMsg.Linear.X = 0.2; % Move forward
    velMsg.Angular.Z = -k_p * error; % Turn right
end
send(velPub, velMsg);
pause(0.1);
end
```

Listing 2.1: Moving the Robot thru the maze without touching walls.

3. Acronyms

IP	Internet Protocol. 1
ROS	Robot Operating System. 1, 2

3.2 Use of Generative AI

AI was used for spelling and grammar checks as well as for the creation of tables and debugging in the LaTeXsyntax. In the Matlab code, it was used for debugging and comments in the code.

A. EKF and Mapping Implementation

```
%nComments done by Chat GPT
clc
clear all
ip = "http://192.168.237.129:11311";
rosshutdown;
rosinit(ip);
velPub = rospublisher('/cmd_vel', 'geometry_msgs/Twist');
laserSub = rossubscriber('/scan', 'sensor_msgs/LaserScan');
% Desired wall distance
desired_distance = 0.7; % Adjust based on your environment (
    in meters)
min_distance = 0.4; % Minimum safe distance to avoid
    collisions

cone_prev=false;

while true
velMsg = rosmessage(velPub);
laserMsg = receive(laserSub, 10); % Wait up to 10 seconds for
     a message
ranges = laserMsg.Ranges; % Distance readings from the laser
    scanner
% Parameters
desired_distance = 1; % Desired distance from the wall (
    meters)
min_distance = 0.4; % Minimum safe distance to avoid
    collisions (meters)
k_p = 0.6; % Proportional gain for angular velocity (tune as
    needed)
k_p_linear = 0.5; % Proportional gain for linear velocity (
    tune as needed)

% Get distances from laser scan
front_indices = [1:45, 315:360]; % Combine indices for front
    cone (0 to 45 and 315 to 360)
front_distance = min(ranges(front_indices)); % Minimum
    distance in the front region
[left_distance, left_index] = min(ranges(225:315)); % Left
    sensor (225 to 315)

% Calculate the error (distance to wall deviation)
error = desired_distance - left_distance;

% Logic for wall-following with proportional control
if front_distance < min_distance
    % Obstacle ahead: Stop and turn left to avoid collision
```

```matlab
    velMsg.Linear.X = 0.0; % Stop
    velMsg.Angular.Z = 0.5; % Turn left
elseif left_distance < desired_distance && left_distance > min_distance
    % Wall is within an acceptable range: move forward and adjust turning rate
    velMsg.Linear.X = 0.4 - k_p_linear * error; % Adjust forward speed proportionally
    velMsg.Angular.Z = -k_p * error; % Adjust turning rate proportionally
elseif left_distance >= desired_distance
    % Wall is too far: Turn left to get closer
    velMsg.Linear.X = 0.2; % Move forward
    velMsg.Angular.Z = k_p * error; % Turn left
else
    % Wall is too close: Turn right to move away
    velMsg.Linear.X = 0.2; % Move forward
    velMsg.Angular.Z = -k_p * error; % Turn right
end
send(velPub, velMsg);
cone=identify_cone(ranges(180:360), 30);
if cone_prev &&  cone
    disp('Cone Found')
end
cone_prev=cone;
pause(0.1);
end
rosshutdown;


function [isCone] = identify_cone(sensor, points_to_check)
    isCone = false; % Default: not a cone
    % Find the index of the closest object
    [~, min_index] = min(sensor);

    % Extract region of interest (ROI)
    start_index = max(1, min_index - ceil(points_to_check / 2));
    end_index = min(length(sensor), min_index + ceil(points_to_check / 2));
    selected_distances = sensor(start_index:end_index)';

    % Remove invalid data points
    if any(isinf(selected_distances))
        return;
    end

    % Convert polar coordinates to Cartesian coordinates
    angles = deg2rad(start_index:end_index);
    x = selected_distances .* cos(angles);
    y = selected_distances .* sin(angles);


    % Analyze geometric properties
    % Fit a second-order polynomial (quadratic curve)
    if length(x) < 3
        % Not enough points for curvature analysis
        return;
    end
    p = polyfit(x, y, 2); % Quadratic fit
    %Plot used for debugging
   %plot(x,y)
   %hold on
   %plot(x,polyval(p,x))

    % Check curvature
    curvature = abs(p(1)); % Curvature is proportional to the
        quadratic coefficient
    if curvature > 5 && curvature<10
        isCone = true;
        %plot(x,y,"Color",[1 1 0])
    end
    hold off
end
```