SUNG KYUN KWAN UNIVERSITY

# AUTONOMOUS PARKING
## EME3071-41

Using a mobile robot with a lidar sensor to achieve autonomous parking


Authors

Ekrem Yüksel
Trym Nygård
Sverre Fagerås

*Supervisors*

Prof. Hyungpil Moon
TA, Hong-ryul Jung

*Sungkyunkwan University*
*성균관대학교*


Date
08/11/2019

# Introduction

Parking can be a difficult maneuver to achieve. It requires precision, distance estimation and the ability to plan ahead. Parking in a tight spot or between two cars (parallel parking) can be a difficult task even for humans. In order for us humans to park a car we have to use our natural sensors like our eyes. We as humans also have the ability to plan ahead and foresee consequences.

Robots on the other hand don't have this ability, but with the help of sensors and good implemented software we can achieve a similar effect. A robot needs to adapt itself to the environment, but in order to do that it has to know where it's in relation to other objects in the environment. This can be achieved by using a LiDAR sensor and Inertial Measurement Unit (IMU). The robot can determine the location of itself via IMU and via LiDAR sensor, it can create a map of the environment. With the position data and map, the robot can make a decision of what it should do or where it should go. In this report we will look at how we can use simultaneous localization and mapping (SLAM) to create a map that the robot can use to navigate itself in real time through an environment.

# Table of contents

# 1 Problem Formulation

The robot should be able to navigate through an environment on its own without any external help. If the environment change it should be able to adapt and adjust itself accordingly. The robot has to be able to:

- Move to a specific point
- Follow a predefined trajectory
- Use the LiDAR sensor to achieve autonomous parking

In order to achieve this the robot has to be able to use its LiDAR sensor to create a map to navigate through and park in a test space, such as parking area. Topics like planning, pose estimation and tracking control are also going to be discussed in more detail through this report.

# 2 The robot

## 2.1 TurtleBot

The TurtleBot is a low cost, modular robot ran with ROS. Made for small projects. For this project it will be made to park autonomously.

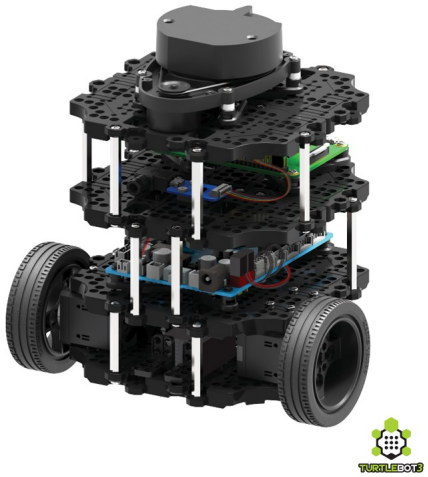


Figure 1: Turtlebot3 (1)

## 2.2 The first layer

The first layer (the lowest one) has space for the battery and contains the dynamixels. The dynamixels are smart actuators. This means that they contain DC motor, reduction gearhead, motor controller, motor driver and network adapter, all in one module.

## 2.3 The second layer

The second layer contains the Arduino based openCR. This card is used to connect to the battery, deliver power to the wheels and everything above.

## 2.4 The third layer

The third layer contains the brains of the robot, the Raspberry Pi 3. This is a single board computer, meaning it's a functioning computer in it's own right. It will receive and transmit the program to and from the robot, through the USB connected WI-FI access point.

## 2.5 The fourth layer

The fourth and top layer contains the LiDAR sensor. This is a laser sensor, sending out a laser and measuring the time it takes to return. Used to find out distances. This will return one point for each degree for a total of 360 point around the robot. It has a range from 12cm to 350cm.

## 2.6 robot parameters

The robot parameters are (2):

- $v_{max} = 0.22^{m}/_{s}$
- $\dot{\phi}_{max} = 2.84^{rad}/_{s}$
- $r_{wheel} = 32.5mm$
- $L = 79mm$

To avoid uncertainties, maximum linear velocity is decreased to 0.16 m/s and angular velocity is decreased to 2.5 rad/s.

# 3 Implementation

## 3.1 Move to a point

When implementing the code its a good idea to start at the basics. For instance, making the robot move to a given point. The robot will place itself in a global coordinate system, and have its own position defined by the coordinates [X, Y]. Given a point, its should then be able to get there by itself. Making the robot move to a point is done via function moveTo which consists of two steps:

- Conversion to robot frame R from inertial frame I
- Moving the robot forward

Function first calculates the error and the rotational matrix $R_z(\theta)$ to convert coordinates from inertial frame I to robot frame R. The heading error is calculated via the relation $\dot{\theta_e} = tan^{-1}(\frac{y_e}{x_e})$.

```
iD = [(iP(1)-iQ(1)); (iP(2)-iQ(2))]; % iP - iQ, position error

 ri_R = [ cos(iQ(3)), sin(iQ(3));
        -sin(iQ(3)), cos(iQ(3))];    % Rotation matrix "R" to "I"

 rP = ri_R * iD;                     % We need rP!

 theta = atan2(rP(2), rP(1));        % arctan2(Y,X)
```

The function is a implementation of the equations written in the 31st and 32nd page of slide *"LAB5_ Turtlebot_ with_ MATLAB"* (refer Appendix A for the slide pages). The function takes the coordinates of current goal point, current position according to the world frame, PID gains as parameters, and returning the values of velocities. The function is explained step by step in the lines below.

To control the robot, proportional controller is used. Implementation is done according to the LAB5 notes (refer to Appendix A for the lecture slide).

```
%=========== PID Controller ===========
% P-control Example
linear_velocity = PID(1,1) * rP(1);     % Kx_p * rP_x
angular_velocity = PID(2,1) * theta;    % Kh_p * rP_theta
```

To avoid harming the robot, following constraints are added to both rotational and linear velocities.

```
% Constraints
%limiting from + side
linear_velocity = min(linear_velocity, max_linear_vel);
%limiting from - side
linear_velocity = max(linear_velocity, -max_linear_vel);
%limiting from + side
angular_velocity = min(angular_velocity, max_rotational_vel);
%limiting from - side
angular_velocity = max(angular_velocity, -max_rotational_vel);
```

Left and right wheel velocities are determined by the following relations:

$$r\dot{\phi}_r = \dot{x_R} + l\dot{\theta_R}$$

$$r\dot{\phi}_l = \dot{x_R} - l\dot{\theta_R}$$

```
 %=========== Velocity -> Commend ===========
left_wheel = linear_velocity - (L * angular_velocity);
right_wheel = linear_velocity + (L * angular_velocity);
```

Turtlebot will move to the point until it is within the 0.02 m range. If it is within the *Look Ahead* it stops the robot, if not it will update the velocity_msg.

```
%=========== Publishing ===========
distance = sqrt( rP(1)*rP(1) + rP(2)*rP(2) ); % Using 'square
    root'

if distance < Look_Ahead % Termination condition
    vel_msg.X = 1;
    vel_msg.Y = 0;
    vel_msg.Z = 0;
    return
else
    vel_msg.X = 1;
    vel_msg.Y = left_wheel;
    vel_msg.Z = right_wheel;
end
```

## 3.2 Follow a predefined trajectory

The next step is to determine a trajectory and making the robot follow it. First a path needs to be implemented. This could either be done by describing it with a equation, or making it with a series of points. As the equation will become extremely complex, its common to use the points.

When using points, it's not that complicated to move the robot. It's the same principle as making it move to a single point, using the moveTo function just described. Then, once it arrives at the goal position, it will stop and conduct a scan of the area for a small moment. This will be used to compare the robot's measured position to it's real position. When that's done, the robot will update the goal position for the moveTo function and start driving again.

## 3.3 Simultaneous Localization And Mapping (SLAM)

Simultaneous localization And mapping (SLAM) can be achieved by using a LiDAR (Light Detection and Ranging) sensor. The LiDAR sensor works by firing a laser at nearby objects and measuring the reflected light with its sensor. The robot is using the LiDAR sensor to scan the area around it and then generating a point every time the laser bounces of an object. After a certain amount of iterations the robot will have enough points to generate a complete map of the area as seen in the example in figure 2.

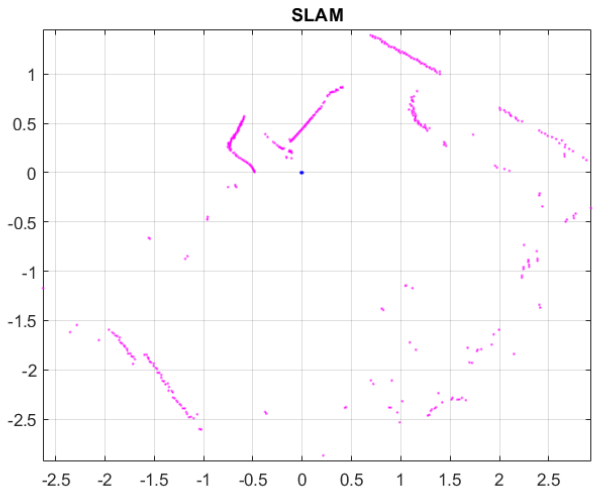



Figure 2: Map of environment

The robot can then use this map to navigate itself into the parking spot. Since ROS automatically add all of the points into an array we can simply go through all of the points in the list and find the point with the smallest distance from the robot as long as the point is within the robot's field of view.

```
function min = closest_point(scan)
    %min.dist = the point with the max. dist. from the robot.
    min.dist = max(scan.Ranges);

    %Go through the list and find the point with the min. dist.
    for j = 1:1:size(scan.Ranges)
        %Limit the fov to 180deg,skip points outside the fov area.
        if(scan.Angles(j) > -pi && scan.Angles(j) < 0)
            if(scan.Ranges(j) < min.dist && scan.Ranges(j) ~= 0)
                min.dist = scan.Ranges(j);
                min.x = scan.Cartesian(j,1);
                min.y = scan.Cartesian(j,2);
            end
        end
    end
end
```

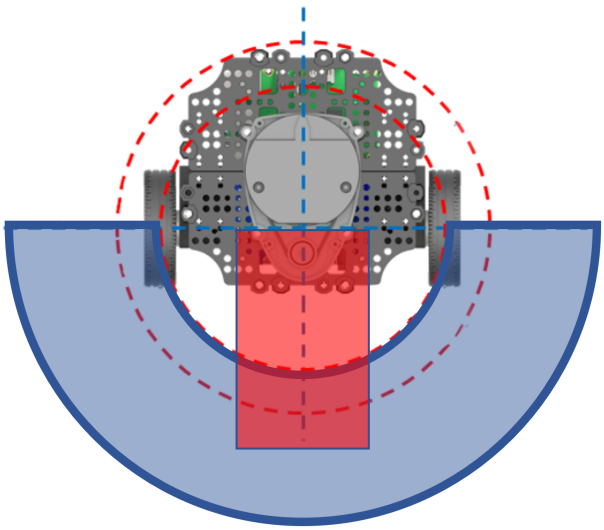


Figure 3: Sketch of the robot

We have defined an area in front of the robot as in the figure 3. This area basically restricts the robots field of view to 180 deg. Thus, it doesn't take into account the objects that are behind it. We have also defined a rectangular search space in front of the robot. The search space area is used to detect if the object is on the right or left side of the robot. If the object is on the right side of the robot it should add an offset to left side and vice versa.

In the offset function we define the search space area, basically if an object is within the area the robot should go in either in the left or right direction depending on the position of the obstacle. The parking mode is turned on when the robot is close to the final goal. The only thing that the park mode does is basically to adjust the offset value so that the parking becomes a little bit more smooth. This offset function will return a value that will then be added to a predefined trajectory. We can then use this predefined trajectory to lead the robot safely into its parking spot area.

```
function const = offset(scan, th, parking_mode)

    const = 0;
    min = closest_point(scan);
    if ~parking_mode
        if((min.x < 0) && (min.x > -th.x))
            if((min.y > 0) && (min.y < th.y))
                const = -(0.15);
            elseif((min.y < 0) && (min.y > -th.y))
                const = +(0.15);
            end
        end
    elseif parking_mode
        if((min.x < 0) && (min.x > -th.x))
            if((min.y > 0) && (min.y < th.y))
                const = -(0.08);
            elseif((min.y < 0) && (min.y > -th.y))
                const = +(0.08);
            end
        end
    end

end
```

# 4 Discussion and results

## 4.1 Following a trajectory

To demonstrate the robots ability to follow a given trajectory, it was made to run in a square. This is because it is easy to compare the the measured trajectory with the real. In figure 4 you can see how the robot didn't get a perfect square but rounded corners.

**There's a video demonstrating it here:**
`https://drive.google.com/open?id=1bLsZeUhwNvizUGOAMjieBo6A5qX_a9Iv`

### 4.1.1 The robots real pose vs the estimated pose

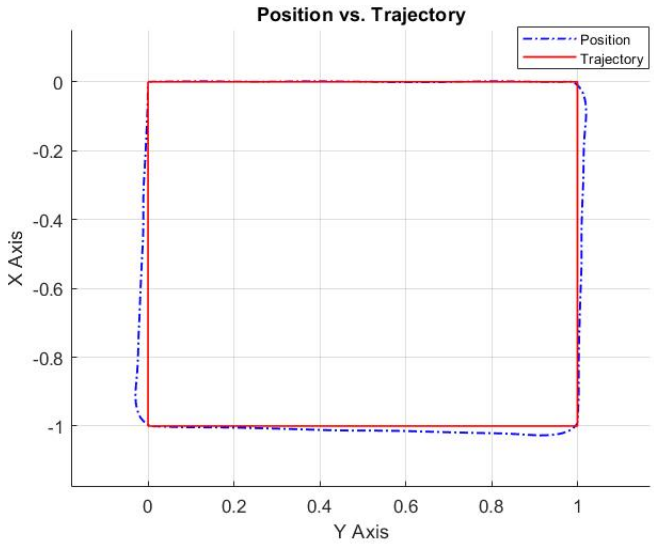



Figure 4: The robots real pose vs the estimated pose, following a square

Looking at the figure above, one can get an idea about the performance of the robot that tries to follow a predetermined trajectory. According to this figure, it is possible to say that robot follows the trajectory adequately.

## 4.2 Parking

The scenario was constructed with the robot going along a wall, with a small obstacle it has to turn past before it has to park itself in a tight space. Here the path of the robot is set to drive into the box. This was chosen to show that it firstly has the ability to follow the wall without crashing, and then autonomously turn itself around objects in its path.

**There's a video demonstrating it here:**
`https://drive.google.com/open?id=1bLsZeUhwNvizUGOAMjieBo6A5qX_a9Iv`

Here you can see that the robot starts turning towards the box, stops and scans the area, and then turns away because there's an obstacle in the way. Then, when it's adjusting back towards the end goal, it finds the black obstacle and avoids that as well. In the end it manages to park itself neatly in the area without crashing.

### 4.2.1 The robots real pose vs the estimated pose

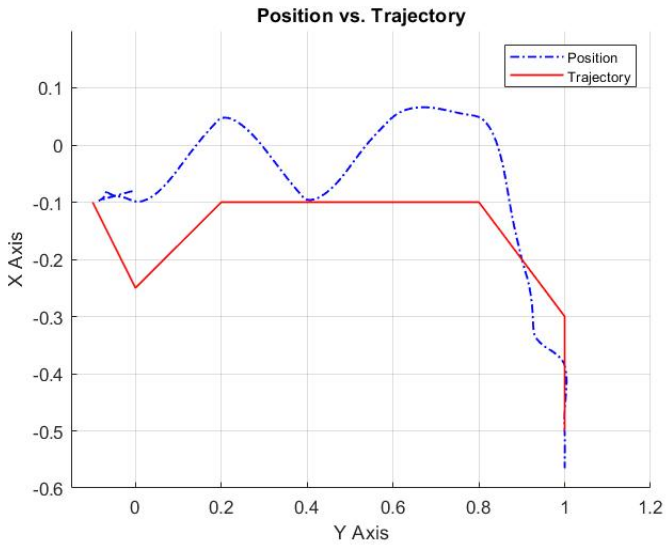



Figure 5: The robots real pose vs the estimated pose

The figure above shows the difference between the real position and desired position.The reason of the large divergence along the y axis is the robot is adding constant to the predetermined trajectory when it sees an obstacle within the threshold. Therefore, it is possible to say that robot diverged from the path in order to avoid obstacles.

To decrease amount of divergence, instead of having a constant offset for each obstacle, difference between distance and the threshold along the y axis. This could have been accomplished via changing the following lines:

- $const = -(0.15) \rightarrow const =$ *-(th.y-min.y)*
- $const = +(0.15) \rightarrow const =$ *+(th.y+min.y)*
- $const = -(0.08) \rightarrow const =$ *-(th.y-min.y)/2*
- $const = +(0.08) \rightarrow const =$ *+(th.y+min.y)/2*

# 5 Sources

[1] ROS components. Turtlebot 3 burger. URL https://www.roscomponents.com/en/mobile-robots/214-turtlebot-3.html. (Found: 06.11.2019).

[2] Robotis. Turtlebot 3, (2019). URL http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/#hardware-specifications. (Found: 08.11.2019).

# 6 Appendices

## Appendix A

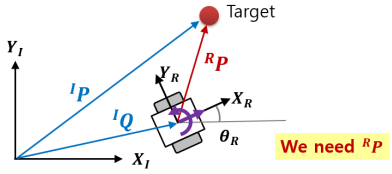
### Coordinate system

- The coordinates of the target point are given in the global coordinate system.

*Transformation*

$${}^{R}P = {}^{R}_{I}T\,{}^{I}P = {}^{R}_{I}R({}^{I}P - {}^{I}Q)$$

unknown — given — given — Estimated by wheel odometry

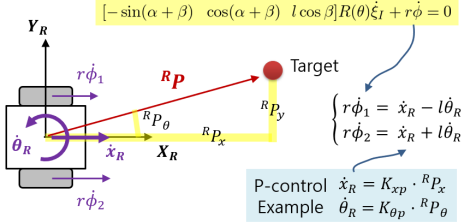



We need ${}^{R}P$

### Mobile Robot Kinematics (refresh)

- To command the robot, you need to know the local coordinates.

$$[-\sin(\alpha+\beta)\quad \cos(\alpha+\beta)\quad l\cos\beta]R(\theta)\dot{\xi}_I + r\dot{\phi} = 0$$




$$\begin{cases} r\dot{\phi}_1 = \dot{x}_R - l\dot{\theta}_R \\ r\dot{\phi}_2 = \dot{x}_R + l\dot{\theta}_R \end{cases}$$

P-control $\dot{x}_R = K_{xp} \cdot {}^{R}P_x$

Example $\dot{\theta}_R = K_{\theta p} \cdot {}^{R}P_\theta$

$r\dot{\phi}_1$ : wheel velocity, $l$ : wheel interval/2