2D Map of The Robot's Environment

Batuhan Ozgur BASAL

P2681345

MSc Intelligent Systems and Robotics

Faculty of Technology, De Montfort University, Leicester

Introduction

Mankind is always navigating various settings. We can recognize things, organize behavior, and analyze environment, such as the temperature and humidity, in a matter of seconds. All of this is feasible thanks to the sensitivity of our biological senses and our brain's capacity to combine data in real time.
We must actively construct a comparable skill into our inventions as we develop intelligent robots and program them to learn their surroundings. This is referred to as "Mapping" in robotics.
A robot must create "map" of its surroundings as it explores its area, utilizing data from accessible cameras and sensors. This map is the robotic version of what we might think of as a visual image. However, whereas a human's "map" is based on sensory input, a robot's "map" is based on data fusion from depth cameras and/or LIDAR.

Environmental modelling is required for a variety of commercial robotic activities. Learning how to read maps necessitates the completion of a task: mapping. The task of mapping is to combine the data collected by the robot's sensor systems into more of a particular description. The modeling of the surroundings and the processing of sensor information are critical parts of mapping.
The aim of this study is to create a 2D map using ROS with turtlebot3 robot, gazebo simulation and rviz visualization tool.

Map Construction Technique

The chosen method for mapping the robot's area is the occupancy grid mapping. In this method, the robot's environment is represented using occupancy grids. Environmental data can be acquired in instantaneously via sensors or imported using previous data. To locate objects in the robot's surroundings**,** various types of sensors and cameras can be employed. In this study, the lidar sensor of the turtlebot3 (360° Laser Distance Sensor LDS-01) was used. Although it seems like a very advantageous situation for the lidar sensor to have a 360-degree viewing angle, it should be said that it has a few disadvantages (further explanation provided at the Testing and Comparison section). Figure 1 demonstrates how information from a laser beam may be used to describe a section of an area inside an occupancy grid. The black cells indicate items identified by the lidar sensor, the white cells indicate unoccupied areas or areas that are not inhabited by obstacles, and the grey cells indicate areas that have not yet been identified.

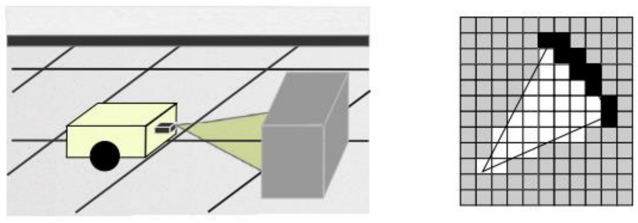
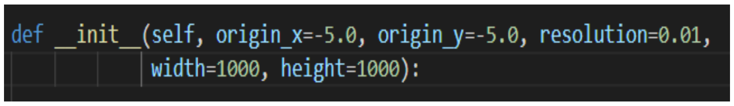


Figure 1 – Occupancy grid and sonar cone

Software Implementation

2 python files were run to use the robot in map drawing. These are mapper.py and wandering_robot.py. wandering_robot.py allows the robot to navigate indefinitely within the gazebo simulation without colliding with objects. This is very important because for the robot to draw the map properly and accurately, it must reach many coordinate points within the environment.
The python file named mapper.py is one of the most important parts of the study. mapper.py has 2 classes. These are the Map and Mapper classes. The Map class stores an occupancy grid as a two dimensional NumPy array and the Mapper class creates a map from laser scan data.

The values of objects created inside the Map class are important. These values determine the size of the occupancy grid. The size of the resolution value determines the size of the cells in the grid (further explanation provided at the Appendix).

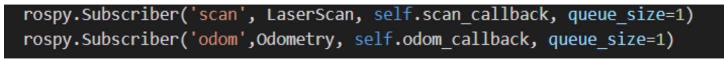


```
def __init__(self, origin_x=-5.0, origin_y=-5.0, resolution=0.01,
             width=1000, height=1000):
```

Figure 2 - The __init__ method

After that, It is necessary to subscribe to 2 topics.

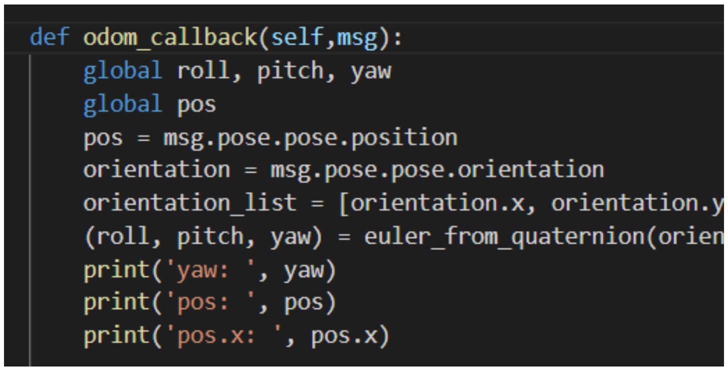
These are provided by ROS and many values can be reached, from the position of the robot to the laser values.



```
rospy.Subscriber('scan', LaserScan, self.scan_callback, queue_size=1)
rospy.Subscriber('odom',Odometry, self.odom_callback, queue_size=1)
```

Figure 3 – Subscribing the ROS Topics

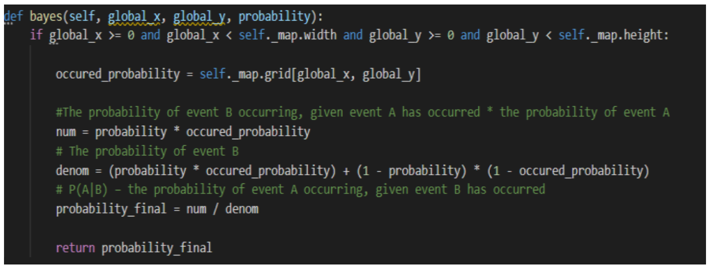
As seen in the code above, there is a message type and callback function next to the subscribed ROS topic. The position of the robot (on the x and y axes) and orientation can be reached by this function. Each time this function is called, it will run the lines of code it contains. The reason why only the yaw value is printed, because only the yaw changes when the turtlebot3 rotates around because it couldn't fly.



```
def odom_callback(self,msg):
    global roll, pitch, yaw
    global pos
    pos = msg.pose.pose.position
    orientation = msg.pose.pose.orientation
    orientation_list = [orientation.x, orientation.y
    (roll, pitch, yaw) = euler_from_quaternion(orien
    print('yaw: ', yaw)
    print('pos: ', pos)
    print('pos.x: ', pos.x)
```

Figure 4 – Odom_callback function

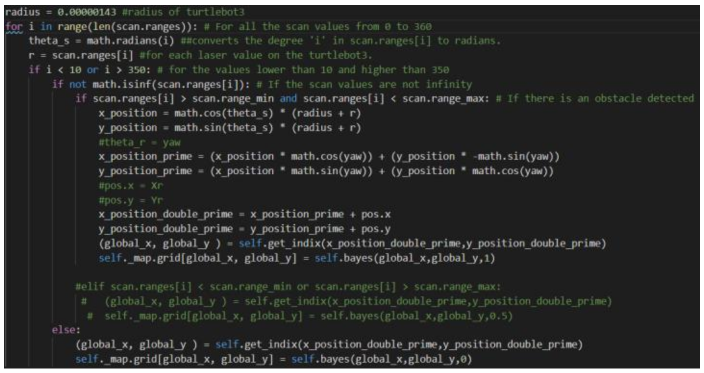
Next, we should update the estimated probabilities of an event. Bayesian rule was used to do this. The Bayes theorem is a scientific equation that may be utilized to calculate the conditional probability of an occurrence. The Bayes' theorem, in its most basic form, defines the likelihood of an occurrence depending on prior awareness of the circumstances that may be important to the occurrence.



```
def bayes(self, global_x, global_y, probability):
    if global_x >= 0 and global_x < self._map.width and global_y >= 0 and global_y < self._map.height:

        occured_probability = self._map.grid[global_x, global_y]

        #The probability of event B occurring, given event A has occurred * the probability of event A
        num = probability * occured_probability
        # The probability of event B
        denom = (probability * occured_probability) + (1 - probability) * (1 - occured_probability)
        # P(A|B) - the probability of event A occurring, given event B has occurred
        probability_final = num / denom

        return probability_final
```

Figure 5 – Bayes Function

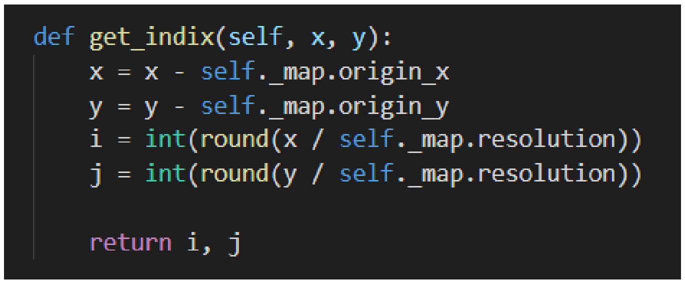
We saw that the lidar sensor has a 360 degree angle of view. Thanks to the for loop, each angle of the robot's lidar sensor can be accessed individually. Cells are formed according to the value corresponding to each degree of the sensor. It is known that the lidar sensor takes value for each degree, for the robot to draw the map more accurately, the robot needs to have a sharper point of view. Therefore, there is a additional if statement to give the robot a sharper and accurate view (20 degrees). Then, the if else statement is used to understand whether the laser beam corresponding to these degrees hit the object or not (checks the value if it is infinity or not). The code under the if statement (scan.ranges[i] > scan.range_min and scan.ranges[i] < scan.range_max:) simply finds the barrier's local location. The barrier's local coordinate system is then mapped to the global coordinate system. The rotation matrix is then multiplied. Finally, the coordinates are translated to obtain the barrier's location in the global system.



```
radius = 0.00000143 #radius of turtlebot3
for i in range(len(scan.ranges)): # For all the scan values from 0 to 360
    theta_s = math.radians(i) ##converts the degree 'i' in scan.ranges[i] to radians.
    r = scan.ranges[i] #for each laser value on the turtlebot3.
    if i < 10 or i > 350: # for the values lower than 10 and higher than 350
        if not math.isinf(scan.ranges[i]): # If the scan values are not infinity
            if scan.ranges[i] > scan.range_min and scan.ranges[i] < scan.range_max: # If there is an obstacle detected
                x_position = math.cos(theta_s) * (radius + r)
                y_position = math.sin(theta_s) * (radius + r)
                #theta_r = yaw
                x_position_prime = (x_position * math.cos(yaw)) + (y_position * -math.sin(yaw))
                y_position_prime = (x_position * math.sin(yaw)) + (y_position * math.cos(yaw))
                #pos.x = Xr
                #pos.y = Yr
                x_position_double_prime = x_position_prime + pos.x
                y_position_double_prime = y_position_prime + pos.y
                (global_x, global_y ) = self.get_indix(x_position_double_prime,y_position_double_prime)
                self._map.grid[global_x, global_y] = self.bayes(global_x,global_y,1)

            #elif scan.ranges[i] < scan.range_min or scan.ranges[i] > scan.range_max:
             #   (global_x, global_y ) = self.get_indix(x_position_double_prime,y_position_double_prime)
              #  self._map.grid[global_x, global_y] = self.bayes(global_x,global_y,0.5)
        else:
            (global_x, global_y ) = self.get_indix(x_position_double_prime,y_position_double_prime)
            self._map.grid[global_x, global_y] = self.bayes(global_x,global_y,0)
```

Figure 6 – For loop of the scan_callback function

To determine the likelihood for the cells, the proper 'I' and 'j index values for self. map.grid[i, j] must be passed. For this grid, the indexes 'I' and 'j' must be integers in the range [0,1000]. Therefore, depending on the x and y values, we must calculate the matching 'I' and 'j'. By giving x and y to the Mapper class, the index values corresponding to the relevant cell in the self. map.grid[i, j] array can be received.



```
def get_indix(self, x, y):
    x = x - self._map.origin_x
    y = y - self._map.origin_y
    i = int(round(x / self._map.resolution))
    j = int(round(y / self._map.resolution))

    return i, j
```

Figure 7 – get_indix function

Finally, if the scan value has an unlimited external value, it means that there is an object at that point. Therefore, its probability is set to 1. If the scan value is unlimited, it is known that there is no object at this point. That's why it is equated to 0

Testing and Results

Finally, by running the mapper.py code, the map is obtained through rviz visualization tool. There are very important factors in obtaining this map. Some of them are map size, map resolution, robot radius and sonar cone of the lidar sensor. Increasing or decreasing these values ensures the drawing time and smoothness of the map.

First, the size of the map can be changed by accessing the input values on the __init__ function under the Map(object) class. If the values entered are smaller or larger than the normal value, the problem of matching with the grid arises. In order to prevent this situation, the most optimal values were found by using the trial and error method.

Map resolution is another important parameter. Reducing this value to 0.01 will cause the cells on the map to shrink. The smaller these cells are, the longer the robot takes to draw the map.

Another important factor is the radius of the robot. This value plays a role in finding the local position of the barrier. The smaller this value is, the more accurately the map is drawn.

The angle of the robot's sonar cone is the last parameter that needs to be discussed. Normally, since the robot has a 360-degree angle lidar sensor, it doesn't matter which way it looks to obtain the part of the map. This causes the robot to detect the same place over and over while drawing the map. That's why it's overlapping. In order to prevent this situation and create the map more accurately, it is necessary to give a certain angle to the robot like a sonar cone.

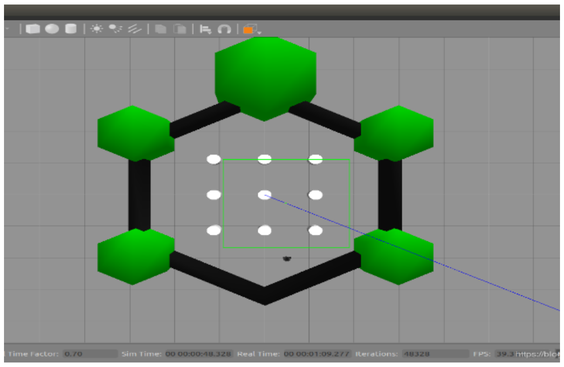


Figure 8 – Turtlebot_world map

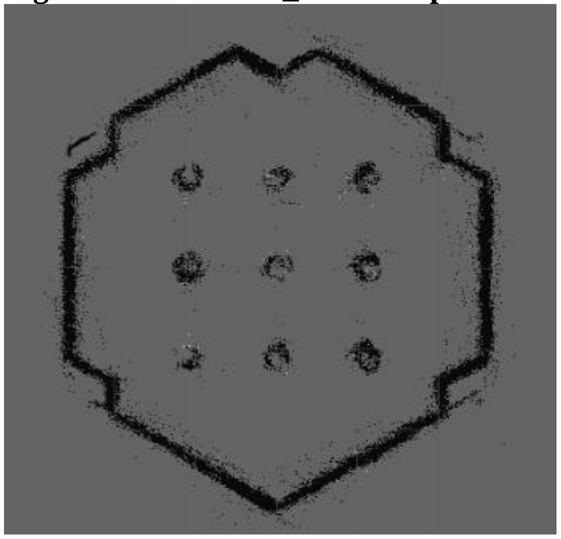


Figure 9 – Final Result of the map

Conclusion

In the software development process, many different methods have been tried and applied. The parameters were optimized by trial and error testing. However, this situation made the drawing process of the map very slow. Although a successful map has been achieved, it can always be done better. More different algorithms can be implemented, and the map can be created more creatively.

In conclusion, I would like to say that many programs were used in this study (Python, ROS, Gazebo, rviz). Each of them has a great importance and it is not possible to get the map without one. Therefore, the biggest positive impact that this work brought to me was to control and communicate with different programs in harmony. Finally, I would like to say that even an event that seems very simple for humans is just as detailed in robotics. To give an example, a lot of calculations are required to calculate the location of the barrier in the global system.

References

[1] How to convert quaternions to Euler angles, [Online] https://www.theconstructsim.com/ros-qa-how-to-convert-quaternions-to-euler-angles/ Week 7 Lab sheet- assignment 1 IMAT5233-Inteligent mobile robotics De Montfort University 2021

[2] nav_msgs/OccupancyGrid Message [Online] http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/OccupancyGrid.html

[3] [RDS] 007 - ROS Development Studio, [Online], Howto use RViz and other ROS Graphical Tools in RDS https://www.youtube.com/watch?v=xkS_OrMN2ag

[4] ROS Navigation, [Online], https://risc.readthedocs.io/1-ros-navigation.html#mapping

[5] tf tutorial, [Online], http://wiki.ros.org/tf/Tutorials

[6]Random moving [Online], http://www2.ece.ohio-state.edu/~zhang/RoboticsClass/docs/ECE5463_ROSTutorialLecture3.pdf

[7] rviz User Guide [Online] https://wiki.ros.org/rviz/UserGuide

[8]How to Output Odometry Data [Online] https://www.theconstructsim.com/ros-qa-196-how-to-output-odometry-data/

[9] Santana, A., Aires, K., Veras, R. and Medeiros, A., 2011. *An Approach for 2D Visual Occupancy Grid Map Using Monocular Vision*.

[10] Ahn, S., K. Lee, W. Chung and S. Sang-Rok Oh (2007), 'SLAM with visual plane: Extracting vertical plane by fusing stereovision and ultrasonic sensor for indoor environment', IEEE International Conference on Robotics and Automation.

[11] Turtlebot3 Specifications, [Online] https://emanual.robotis.com/docs/en/platform/turtlebot3/features/

Acknowledgment

We thank Dr. Aboozar Taherkhani for his help throughout this study. This work was fully supported by him.

Appendix

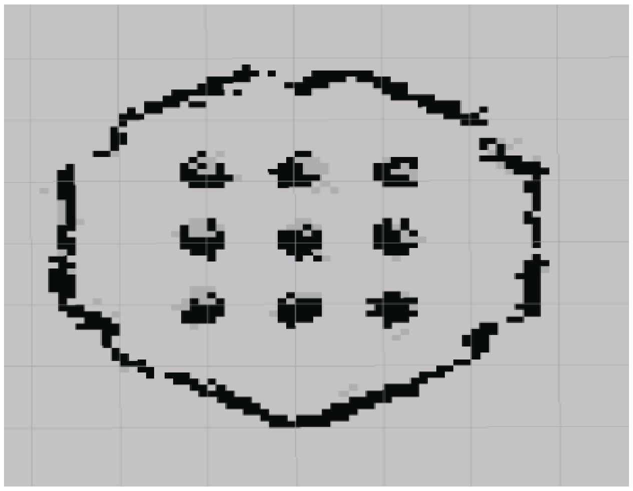


Figure 10 – When the Resolution set at .1

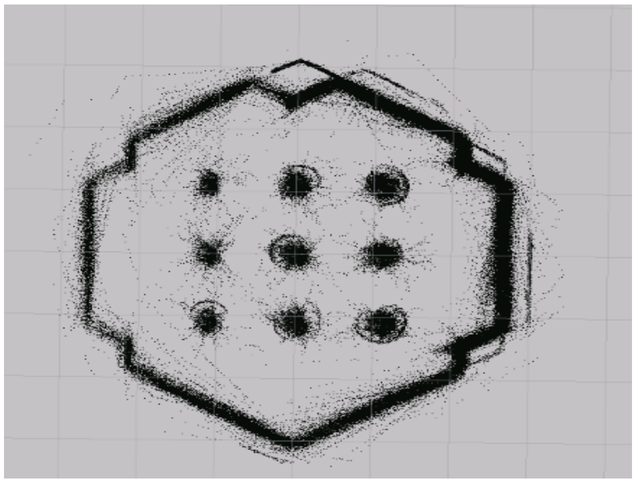


Figure 11 – When all the scan values were used (360 degress)

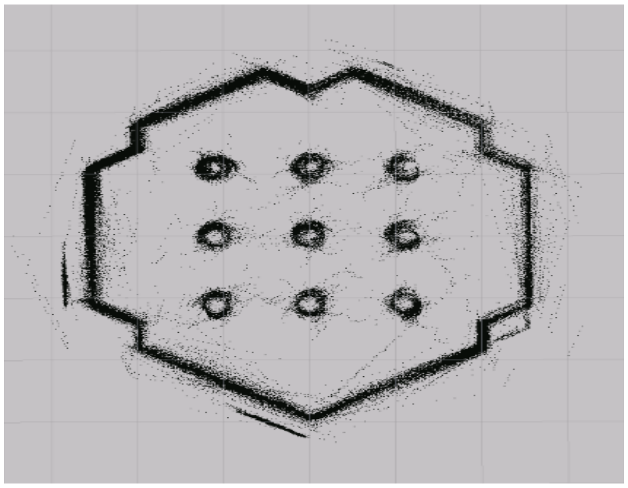


Figure 12 – When the solar cone of the robot set at 20 degrees

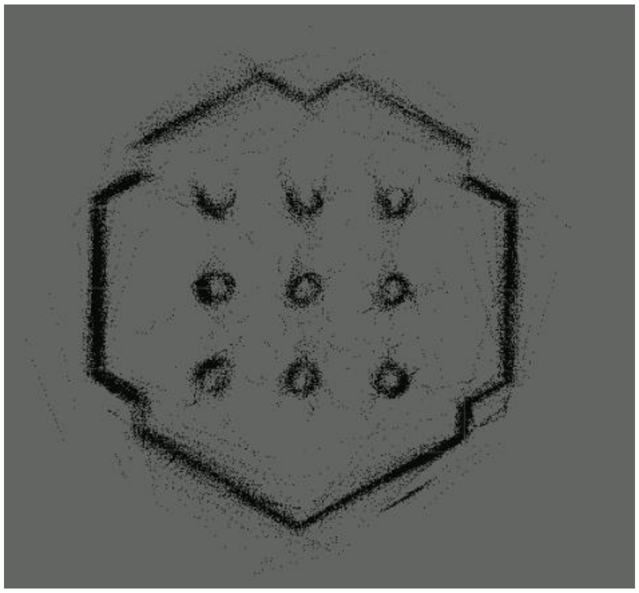


Figure 13 – When the radius of the robot sets as 0.143