UNITED STATES MILITARY ACADEMY

LAB 4 REPORT - SLAM WITH THE TURTLEBOT3

EE487: EMBEDDED SYSTEMS DEVELOPMENT
SECTION T2

COL KORPELA

BY

CDT NICHOLAS E. LIEBERS '23, C-2

WEST POINT, NY 10996

21 MAR 2022

NEL
____ DOCUMENTATION IDENTIFIES ALL SOURCES USED AND ASSISTANCE RECEIVED IN COMPLETING THIS ASSIGNMENT.

____ NO SOURCES WERE USED OR ASSISTANCE RECEIVED IN COMPLETING THIS ASSIGNMENT.

SIGNATURE: LIEBERS.NICHOLAS.EDWARD.1557580448 Digitally signed by LIEBERS.NICHOLAS.EDWARD.1557580448 Date: 2022.03.21 08:15:31 -04'00'

Contents

1. Introduction **3**

1.1. Purpose . . . 3

1.2. Learning Objectives . . . 3

2. Theory **4**

2.1. What is Simultaneous Localization and Mapping (SLAM)? . . . 4

2.2. How does SLAM Work? . . . 4

2.3. Different SLAM Algorithms . . . 5

2.3.1. Gmapping . . . 5

2.3.2. Cartographer . . . 5

2.3.3. Hector . . . 6

3. Experimental Setup and Procedure **7**

3.1. Required Equipment . . . 7

3.2. Software Setup . . . 7

3.2.1. ROS Setup on the TurtleBot3 . . . 8

3.3. Physical Setup . . . 8

4. Results and Discussion **9**

4.0.1. Errors and Issues . . . 9

4.0.2. Solutions to Errors . . . 9

4.0.3. Incremental Software Building Process . . . 9

4.0.4. Algorithm Comparison . . . 10

5. Conclusion and Recommendations **14**

6. References **15**

Appendices

0.1. Python Code 16

1. Introduction

1.1. Purpose

The purpose of this lab is to use the Robot Operating System (ROS) to implement Simultaneous Localization and Mapping (SLAM) using a Turtlebot3. One must use SLAM to map a room manually and autonomously. The purpose was accomplished, and the following learning objectives were met:

1.2. Learning Objectives

1. Utilize the Turtlebot3.
2. Utilize the RViz visualization environment.
3. Learn the principles of SLAM (Simultaneous Localization and Mapping).
4. Research and compare/contrast at least 3 SLAM techniques.
5. Program a single-board Linux computer for an embedded application.
6. Utilize the Linux operating system, including its command line interface (CLI).
7. Leverage the usability of Python and its libraries for the rapid development and prototyping of embedded systems.

2. Theory

In order to complete this lab, we must define and explore different SLAM algorithms. For more information on ROS and the Turtlebot3, please reference Lab3.

2.1. What is Simultaneous Localization and Mapping (SLAM)?

SLAM is a way to draw/map an environment by estimating the current location of a scanning machine in some arbitrary space [3].

2.2. How does SLAM Work?

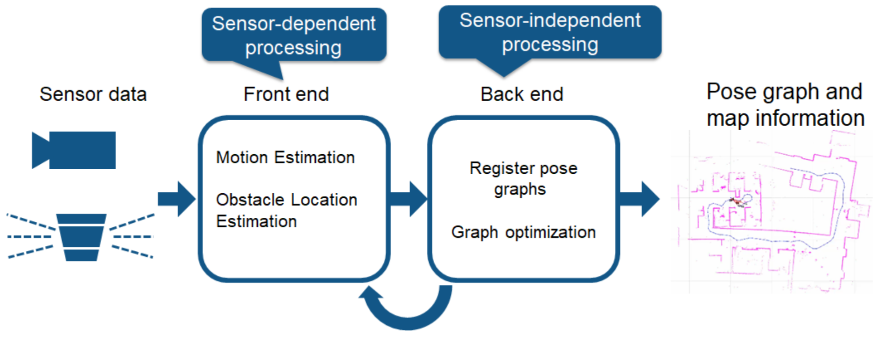
Sensor-dependent processing
Sensor-independent processing
Sensor data
Front end
Back end
Pose graph and map information
Motion Estimation
Obstacle Location Estimation
Register pose graphs
Graph optimization

Figure 1: This diagram describes how a SLAM algorithm receives sensor data, processes it and determines where obstacles are, and then uses algorithms to create pose estimation and a map [9].

There are two parts to how SLAM is functioning: the front-end processing and the back-end processing. Figure 1 illustrates how data is collected using some sensor, the signal data is processed to estimate motion and determine where obstacles are, a back-end uses algorithms to determine location, and then a map of the user's environment is created [9].

SLAM can use cameras or a LiDAR sensor. In this lab, only LiDAR SLAM was used. A LiDAR sensor creates a point cloud of the environment, and the lasers in a LiDAR sensor are more precise than cameras. Due to this feature, LiDAR tends to be used in self-driving cars or drone applications. However, if there are few obstacles in the environment, LiDAR SLAM can struggle to find its own location since there are fewer references to approximate its location.

2.3. Different SLAM Algorithms

There are three SLAM algorithms that will be reviewed in this lab: Gmapping, Cartographer, and Hector. A high level review of each will occur below:

2.3.1. Gmapping

Gmapping is the most popular of SLAM algorithms. In 2005, the early versions of Gmapping were published [5]. Gmap takes raw laser range data and can create a grid map [7]. Gmapping is based on a technique called Rao-Blackwellized particle filter that allows the required number of laser samples to be reduced, and it reduces the required re-sampling while scanning [6].

2.3.2. Cartographer

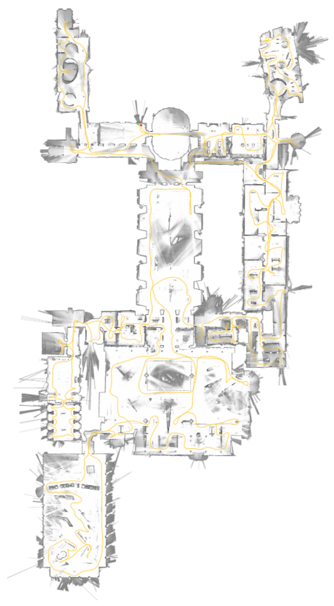

Figure 2: A Cartographer map [8]

Cartographer is an algorithm developed by Google researchers in 2016. It contains a ROS implementation which allows it to be use on robots running ROS [2] [8]. Figure 2 is an example Cartographer map.

2.3.3. Hector

Hector is the last SLAM algorithm to be shortly reviewed. The main difference between Hector and the other algorithms is that Hector does not use odometry–meaning it does not require motion sensor data from the robot [1]. Therefore, in open environments with little obstacles, Hector would perform worse since it would lack the data necessary to localize itself.

3. Experimental Setup and Procedure

3.1. Required Equipment

1. Computer running Linux with proper software installed to interface with Turtlebot3[1].
2. A Turtlebot3.
3. Turtlebot3 battery.

3.2. Software Setup

This lab runs ROS on a computer and on the Turtlebot3. The following command will install required packages and Gmapping.

```
$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
  ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
  ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
  ros-noetic-rosserial-python ros-noetic-rosserial-client \
  ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
  ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
  ros-noetic-compressed-image-transport ros-noetic-rqt* ros-noetic-rviz \
  ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-markers
```

[3]

To run install Cartographer, run the following commands:

```
$ sudo apt update
$ sudo apt install -y python3-wstool python3-rosdep ninja-build stow
$ cd ~/catkin_ws/src
$ wstool init src
$ wstool merge -t src https://raw.githubusercontent.com/cartographer-project/
    cartographer_ros/master/cartographer_ros.rosinstall
$ wstool update -t src
$ sudo rosdep init
```

[1]see `https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/#pc-setup` for required packages.

```
$ rosdep update
$ rosdep install --from-paths src --ignore-src --rosdistro=noetic -y
$ src/cartographer/scripts/install_abseil.sh
$ sudo apt remove ros-noetic-abseil-cpp
$ catkin_make_isolated --install --use-ninja
```

[3]

To install Hector, use the following commands:

```
$ sudo apt-get install ros-noetic-hector-mapping
```

[3]

3.2.1. ROS Setup on the TurtleBot3

The Turtlebot3 comes fully equipped with ROS; however, some configuration was required in order to get full functionality.

The TurtleBot3 Setup guide on the West Point Robotics Github offers configuration details for how we setup our robot [4].

In order to turn on the LiDAR sensor and node, the following command is run:

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

[3]

To manually control the Turtlebot3's movement, the following command can be run:

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

[3]

To launch gMapping, the following command was run:

```
$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

[3].

To run the other SLAM algorithms, simply replace "gmapping" with the other algorithm name–such as cartographer or hector.

3.3. Physical Setup

No modification are required to the TurtleBot3.

4. Results and Discussion

The Turtlebot3 was able to map its environment manually using two different SLAM algorithms. Additionally, the Turtlebot3 was able to map its environment autonomously using gMapping and a custom python script.

4.0.1. Errors and Issues

The distance measurements that the LiDAR sensor outputted were erratic when I was attempting to read the distance measurements.

4.0.2. Solutions to Errors

To fix my issue, I conducted numerous tests to determine why the LiDAR would sometimes not see a wall. I ”echoed” the laser data and determined that the LiDAR is not very consistent. Sometimes it would receive good readings while other times it would not. However, it was good enough to do a basic job of mapping the room.

4.0.3. Incremental Software Building Process

This project consists of two parts:

1. The Computer.

2. The Turtlebot3.

I first looked back to Lab2 where I built a custom publisher and subscriber that allowed a turtle to spell the letter ”N.” Since this script already established a ”publisher/subscriber” framework, I only needed to add this code to my current project to add a publisher and subscriber to the project.

Next, I began to test the script on the actual Turtlebot3 robot. While some small bugs arose, I was able to quickly make the Turtlebot3 draw the letter ”N.” I then changed the code so that it subscribed to the LiDAR sensor. Using a tutorial from the internet, I was able to build code to effectively interpret the LiDAR data. See Appendix A for the python code.

Finally, I began testing the Turtlebot3 in the environment. As the Turtlebot3 ran into walls, I would adjust certain distance constants in my code and retest. Eventually, I found some settings that worked well enough that the Turtlebot3 could semi-effectively map the room.

4.0.4. Algorithm Comparison

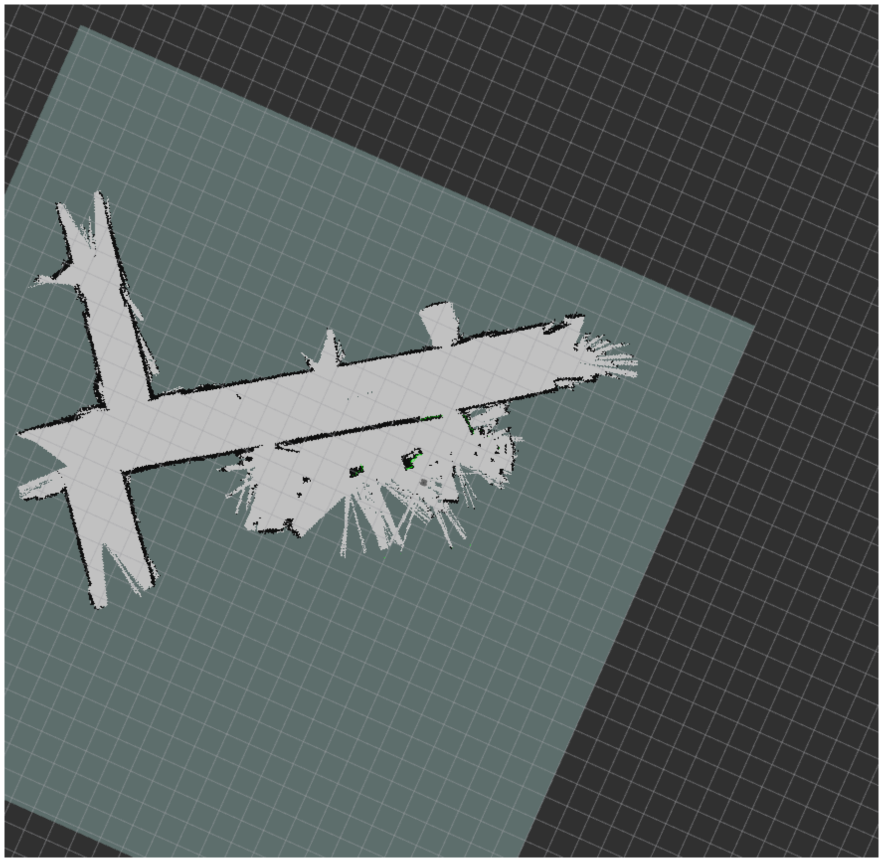

Figure 3: The EECS Robotics Lab and EECS hallways mapped manually using the Gmapping algorithm.

Gmapping's use of LiDAR and on-board sensors proved to be the best algorithm for mapping the robotics lab in the EECS department. Figure 3 illustrates how well Gmapping performed. It was able to accurately map most of the robotics lab and the outside hallways. The Turtlebot3 was not in ”explorer” mode, but this is still an impressive result.

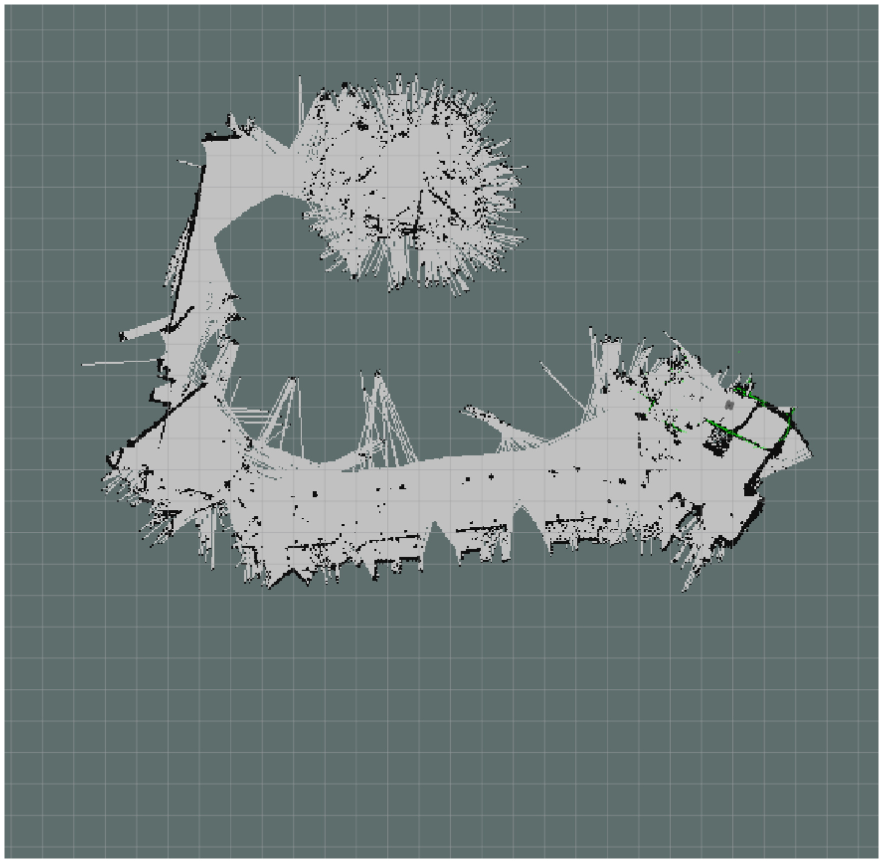

Figure 4: The EECS Robotics Lab mapped manually using the hector algorithm.

As seen in Figure 4, the Hector algorithm performed okay, but struggled to map most of the open area in the middle of the room. However, since Hector does not use any motion data from the Turtlebot3 sensors, this conclusion makes sense. When there were walls/obstacles present, Hector did well and was able to map them with great detail; however, in the absence of these obstacles, Hector appeared lost.

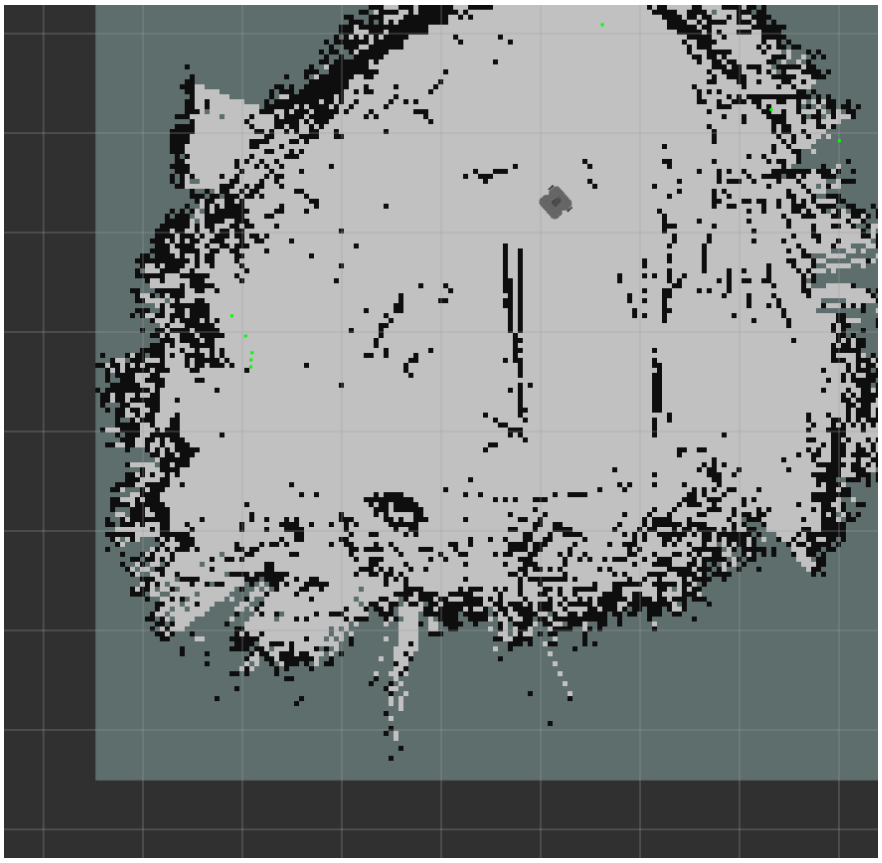

Figure 5: The EECS Robotics Lab mapped manually using the Karto Algorithm

In Figure 5, the Turtlebot3 had no sense of its surroundings. It was unable to localize its location with respect to its surroundings. As a result, the final map shows little to no similarity to the robotic lab. This algorithm may require additional tuning from the operator to make it more effective; however, I did not conduct any additional tuning. While I tried to see if Karto would offer any improvements, it is clear that Karto performed exceptionally worse than both Gmapping and Hector.

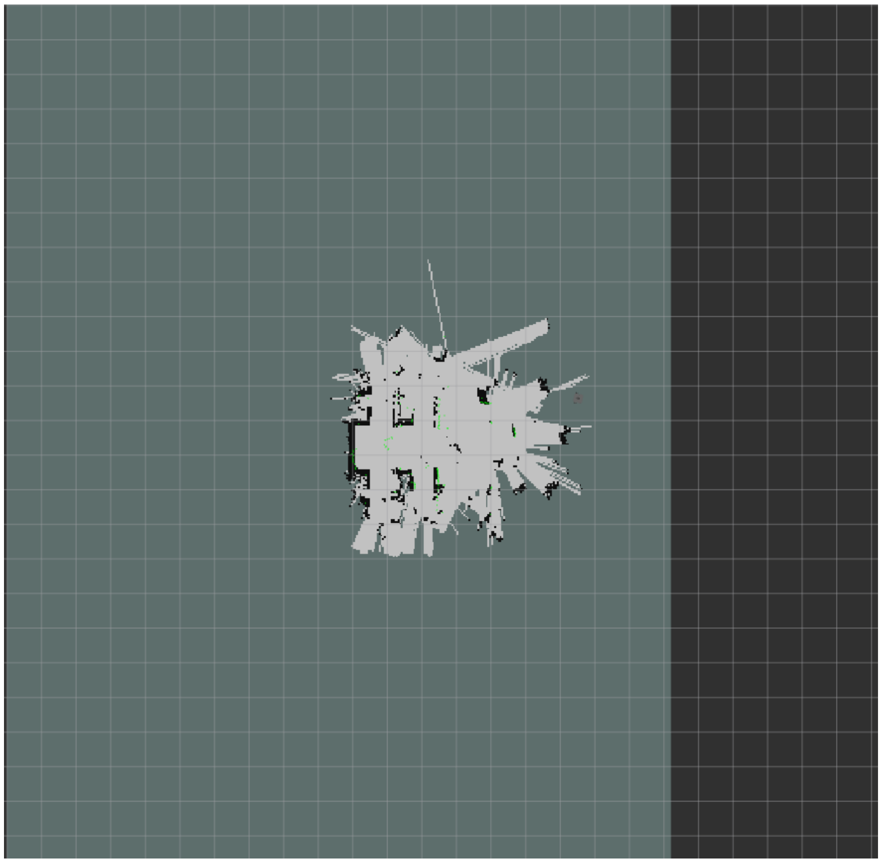

Figure 6: This map illustrates how the Gmapping algorithm in "explore" mode was able to map parts of the room autonomously.

In Figure 6, this Gmapping map of the robotics lab was created autonomously using a custom built python script to control and help the Turtlebot3 navigate. While the Turtlebot3 got stuck in a corner at one point, it made exceptional progress on its own. It mainly explored desk areas, but it failed to navigate tighter areas effectively.

5. Conclusion and Recommendations

In conclusion, this lab was successful. I was able to complete the required objectives for this Lab. An important take way for this lab is SLAM technology is still not perfect. While accurate maps were able to be created using Gmapping, it took a long time to create these maps. When navigating autonomously, the Turtlebot3 would sometimes get stuck, and for those two reasons alone, this technology present in this lab–in this current form–would not be ready for a field environment. If this lab were to be conducted again, more time should be spent on learning how to fine-tune the SLAM algorithms. Just using them ”out of the box” is not as effective as it could be. Also, incorporating a range finder to the front of the Turtlebot3 would be an interesting alternative to using the LiDAR to keep track of obstacle to avoid.

6. References

[1] Hector_Mapping - ROS Wiki.

[2] Cartographer ROS Documentation The Cartographer Authors. 2021.

[3] TurtleBot3 Setup and Tutorials, 2022.

[4] Turtlebot3 Setup West Point Robotics, 2 2022.

[5] Pieter Abbeel. gMapping.

[6] Giorgio Grisetti and Cyrill Stachniss. Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling.

[7] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. GMapping.

[8] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2D LIDAR SLAM. *Proceedings - IEEE International Conference on Robotics and Automation*, 2016-June:1271–1278, 6 2016.

[9] MathWorks. What Is SLAM (Simultaneous Localization and Mapping), 2022.

Appendix A - Code

0.1. Python Code

```
#!/usr/bin/env python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import pow, atan2, sqrt
import math
from sensor_msgs.msg import LaserScan
import time
import random
class TurtleBot:
    def __init__(self):
        rospy.init_node('turtlebot_controller', anonymous=True)

        self.pose_subscriber = rospy.Subscriber('/scan',
                                                LaserScan, self.update_scan)



        self.velocity_publisher = rospy.Publisher('/cmd_vel',Twist, queue_size=10)

        self.pose = Pose()
        self.rate = rospy.Rate(10)

        self.vel_msg = Twist()

        self.obstacle = False

    def update_pose(self, data):
        """Callback function which is called when a new message of type Pose is
        received by the subscriber."""
        self.pose = data
        self.pose.x = round(self.pose.x, 4)
        self.pose.y = round(self.pose.y, 4)

    def update_scan(self, data):
        self.checkObstacle(data.ranges[0:15])
        self.checkObstacle(data.ranges[344:359])

    def checkObstacle(self, data):
        for i, point in enumerate(data):
            print(point)
            if point > 0 and point <= .8:
                self.obstacle = True
                # print(point)
                break

    def rotate(self, angle):
        # Setting the current time for distance calculus
        print("X location: {}, Y location: {}".format(self.pose.x, self.pose.y))
        self.vel_msg.angular.z = .25
        t0 = rospy.Time.now().to_sec()
        current_angle = 0
        relative_angle = angle*2*math.pi/360

        while(current_angle < relative_angle):
                self.velocity_publisher.publish(self.vel_msg)
                t1 = rospy.Time.now().to_sec()
                current_angle = self.vel_msg.angular.z*(t1-t0)

        self.vel_msg.angular.z = 0
        self.velocity_publisher.publish(self.vel_msg)
    def moveLaser(self):
        self.vel_msg.linear.x = .15
        while True:
            self.vel_msg.linear.x = .15
            self.velocity_publisher.publish(self.vel_msg)
```

```python
            if self.obstacle:
                self.vel_msg.linear.x = -1.0
                self.velocity_publisher.publish(self.vel_msg)
                time.sleep(.5)
                self.vel_msg.linear.x = 0.0
                self.velocity_publisher.publish(self.vel_msg)
                self.rotate(random.randint(10, 180))
                self.obstacle = False

if __name__ == '__main__':
        try:
                x = TurtleBot()
                x.moveLaser()

        except rospy.ROSInterruptException:
                pass
# Used https://www.theconstructsim.com/read-laserscan-data/ for sensor_msg ROS package
```

NOTE: This code was heavily adapted from Lab2. While large modifications were used, this code originally came from Lab2 and its respective citations.