Lab report 3

Student# 21060007

Task 1#

```
def callback(data):
   y = []
   global value
   rospy.loginfo(data)
   for x in data.ranges:
           if x != float('inf'):
         y.append(x)
   value = (min(y))
   #rospy.loginfo(value)

def listener():
   rospy.init_node('turtlebot_controller', anonymous=True)
   rospy.Subscriber("/scan", LaserScan, callback)
   # spin() simply keeps python from exiting until this node is stopped
   rospy.spin()
if __name__ == '__main__':
   listener()
```

Task 2

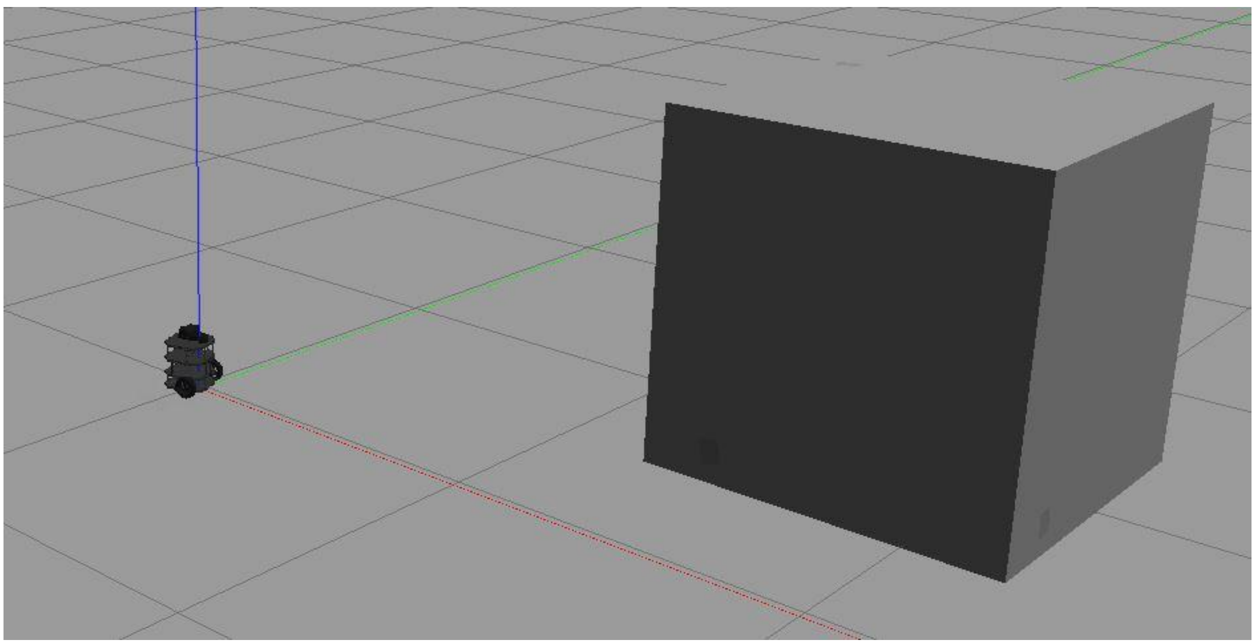
Description: The robot finds the closest point for the obstacle using min lidar data. The robot rotates until the the 0 angle (front face i.e 0^{th} value of the lidar) is equal to minimum distance calculated above

Code:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
import math
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

value =0
value1=0

def callback(data):
   rate = rospy.Rate(10)
   y = []
```

```
w = []
global value
for x in data.ranges:
        if x != float('inf'):
      y.append(x)
value = (min(y))
min_angle=data.angle_min
max_angle=data.angle_max

rospy.loginfo("minimum value in array is %f", value)
bearing=min_angle+ data.ranges.index(value) * max_angle/(len(data.ranges))
rospy.loginfo("corrosponding angle is %f", bearing)

y1 =[]


value1 = (max(y))
rospy.loginfo("maximum value in array is %f", value1)
bearing1=min_angle+data.ranges.index(value1) * max_angle/(len(data.ranges))
rospy.loginfo("corrosponding angle is %f", bearing1)

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



rospy.loginfo("0 value is at %f", data.ranges[0])
w.append(data.ranges[0])
vel_msg.angular.x = 0
vel_msg.angular.y = 0
vel_msg.angular.z = -1
velocity_publisher.publish(vel_msg)


rospy.loginfo(w)
while round(data.ranges[0],1) == round(value,1):
   rospy.loginfo("0 value is at %f", data.ranges[0])
   vel_msg.angular.z = 0
   velocity_publisher.publish(vel_msg)


   rate.sleep()
```

```
    #rospy.loginfo(value)

def listener():
    rospy.init_node('turtlebot_controller', anonymous=True)
    rospy.Subscriber("/scan", LaserScan, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```



Task 3

Description: The code ask for the user for a minimum distance, Once the robot reaches the minimum distance it rotates until the lidar 270th element is equal to the minimum distance. PI controller is used for linear and angular velocity error calculation

```
#!/usr/bin/env python
# license removed for brevity
import rospy
import math
```

```
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from math import pow, atan2, sqrt
import time

import numpy as np
import matplotlib.pyplot as plt

# PID variable

error_angle = 0
error_linear = 0

sumError_angle = 0
sumError_linear = 0

rateError_linear = 0
rateError_angle = 0

currentTime = time.time()
previousTime = time.time()

lastError_linear = 0
lastError_angle = 0

y = []
value =0
required_yaw = 0


class TurtleBot:

    def __init__(self):
        # Creates a node with name 'turtlebot_controller' and make sure it is a
        # unique node (using anonymous=True).
        rospy.init_node('turtlebot_controller', anonymous=True)

        # Publisher which will publish to the topic '/turtle1/cmd_vel'.
        self.velocity_publisher = rospy.Publisher('/cmd_vel',
                                                  Twist, queue_size=10)

        # A subscriber to the topic '/turtle1/pose'. self.update_pose is called
```

```
    # when a message of type Pose is received.
    self.Laserscan_subscriber = rospy.Subscriber("/scan", LaserScan, self.callback)

    self.pose = LaserScan()

    self.pose_subscriber =  rospy.Subscriber("/odom", Odometry, self.update_pose)

    self.pose_odom = Odometry()
    self.rate = rospy.Rate(10)


def update_pose(self, data):

    self.pose_odom = data
    self.pose_odom.pose.pose.position.x= round(self.pose_odom.pose.pose.position.x, 4)
    self.pose_odom.pose.pose.position.y = round(self.pose_odom.pose.pose.position.y, 4)


    self.pose_odom.pose.pose.orientation.x = round(self.pose_odom.pose.pose.orientation.x,4)
    self.pose_odom.pose.pose.orientation.y = round(self.pose_odom.pose.pose.orientation.y,4)
    self.pose_odom.pose.pose.orientation.z = round(self.pose_odom.pose.pose.orientation.z,4)
    self.pose_odom.pose.pose.orientation.w = round(self.pose_odom.pose.pose.orientation.w,4)


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

        self.pose.angle_min=self.pose.angle_min
        self.pose.angle_max=self.pose.angle_max

def min_distance (self):

        for x in self.pose.ranges:
            if x != float('inf'):
         y.append(x)
        value = (min(y))
    return value

def bearing_angle(self,value):
    bearing=self.pose.angle_min+ value * self.pose.angle_max/(len(self.pose.ranges))
```

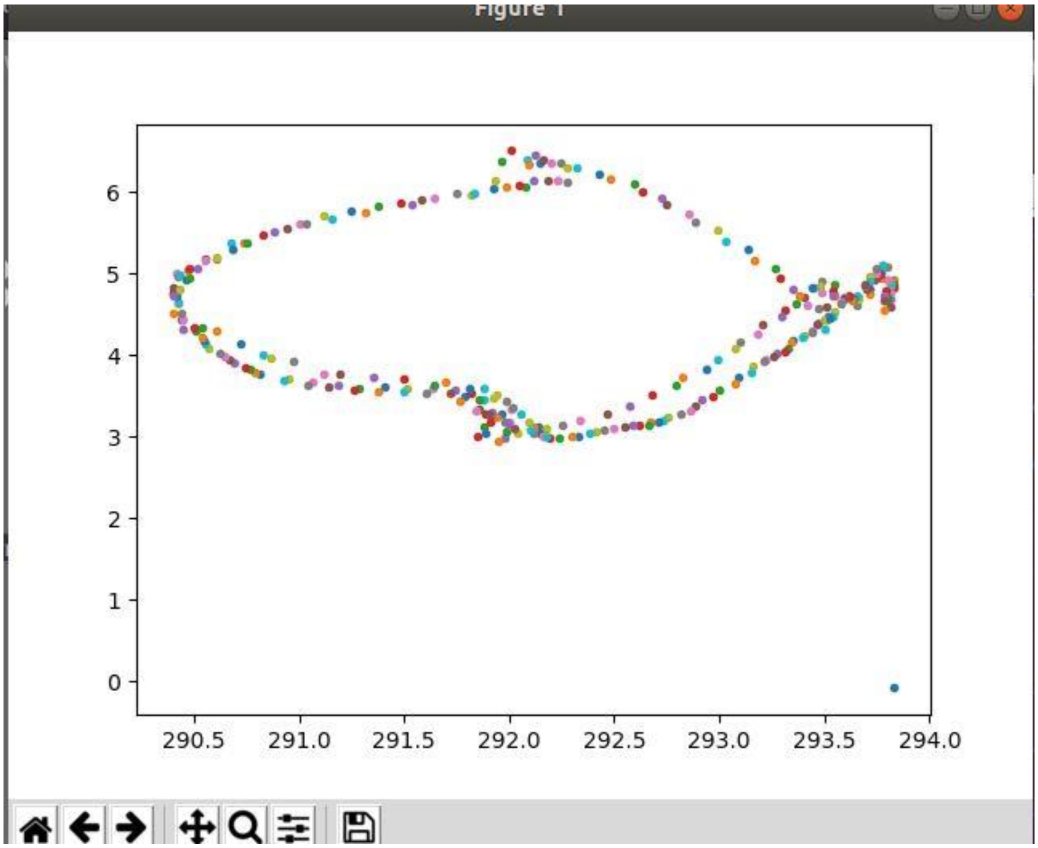
```
        return bearing


    def angular_vel(self):
        """See video: https://www.youtube.com/watch?v=Qh15Nol5htM."""

        orientation_list = [self.pose_odom.pose.pose.orientation.x,
self.pose_odom.pose.pose.orientation.y,self.pose_odom.pose.pose.orientation.z,self.pose_odom.pose.
pose.orientation.w]
        (roll, pitch, yaw) = euler_from_quaternion(orientation_list)
        #rospy.loginfo('yaw %f',yaw)

        return  (yaw)

    def closeset_angle_degree(self):

        clospt=min(self.pose.ranges)
            closang=self.pose.angle_min + np.argmin(self.pose.ranges) *
self.pose.angle_max/len(self.pose.ranges)

            if closang>=0 and closang <= math.pi:
                        closang=closang
            if closang>math.pi and closang <= 2*math.pi:
                        closang=-(2*math.pi-closang)
            closang = math.atan2(math.sin(closang),math.cos(closang))
        return closang



    def move2goal(self):

        global currentTime, previousTime, error_linear, sumError_linear, rateError_linear, lastError_linear,
error_angle, sumError_angle, rateError_angle, lastError_angle, yaw

        global x
        global y
        global yaw
        global clospt
        global closang
            x=0
            y=0
            yaw=0
            nearest_point=0
            close_ang=0
```

```
# Get the input from the user.

distance_min = input("Set your min distance: ")

vel_msg = Twist()

while self.pose.ranges[0] > distance_min:

        vel_msg.angular.z = 0
        vel_msg.angular.x = 0
        vel_msg.angular.z = 0

        vel_msg.linear.z = 0
        vel_msg.linear.y = 0
   vel_msg.linear.x = 0.22

   self.velocity_publisher.publish(vel_msg)
   rospy.loginfo(self.pose.ranges[0])
   self.rate.sleep()

   # Publishing our vel_msg
   self.velocity_publisher.publish(vel_msg)

   # Publish at the desired rate.
   self.rate.sleep()

# Stopping our robot after the movement is over.
rospy.loginfo("reached")
rospy.loginfo(len(self.pose.ranges))
vel_msg.linear.x = 0
vel_msg.linear.y = 0
vel_msg.linear.z = 0

vel_msg.angular.x = 0
vel_msg.angular.y = 0
vel_msg.angular.z = 0
self.velocity_publisher.publish(vel_msg)
self.rate.sleep()

while round(self.pose.ranges[270]) > round(distance_min):

        vel_msg.angular.z = 0.5
        vel_msg.angular.x = 0
```

```
        vel_msg.angular.y = 0

        vel_msg.linear.z = 0
        vel_msg.linear.y = 0
    vel_msg.linear.x = 0

    self.velocity_publisher.publish(vel_msg)
    rospy.loginfo("270 is %f", self.pose.ranges[270])
    rospy.loginfo("270 angle is %f", self.bearing_angle(270))
    x = self.pose.ranges.index(min(self.pose.ranges))
    rospy.loginfo("min index is %f", x)
    rospy.loginfo("min angle is %f", self.bearing_angle(x))

    #Srospy.loginfo("required yaw is %f", self.angular_vel())
    required_yaw = self.angular_vel()

    self.rate.sleep()


rospy.loginfo("check")
vel_msg.linear.x = 0
vel_msg.linear.y = 0
vel_msg.linear.z = 0


vel_msg.angular.x = 0
vel_msg.angular.y = 0
vel_msg.angular.z = 0

self.velocity_publisher.publish(vel_msg)
true =1

      while (true):
       nearest_point = min(self.pose.ranges)
       yaw = (self.angular_vel())
                dm=np.matrix([[x],[y],[0]])
       yaw = math.atan2(math.sin(yaw),math.cos(yaw))


       close_ang = self.closeset_angle_degree()
       rm=np.matrix([[math.cos(yaw),-1*math.sin(yaw),0],[math.sin(yaw),math.cos(yaw),0],[0,0,1]])

      Pm=np.matrix([[nearest_point*math.cos(close_ang)],[nearest_point*math.sin(close_ang)],[0]])
                P_i=(rm*Pm)+dm
```

```
currentTime = time.time()
elapsedTime = (currentTime - previousTime)

y = self.pose.ranges.index(min(self.pose.ranges))

        goal = self.bearing_angle(x)

y = self.bearing_angle(y)
        error_angle = y-goal
        #error_angle = (required_yaw - (self.angular_vel()))
error_position = (distance_min - self.pose.ranges[270])


sumError_angle = sumError_angle + (error_angle * elapsedTime)
sumError_linear = sumError_linear + (error_position * elapsedTime)

        rospy.loginfo("error angle is %f", error_angle)
        rospy.loginfo("error position is %f", error_position )

vel_pid = (1.2 * error_position) + (0.0001* sumError_linear)
ang_pid = (0.5 * error_angle) + (0.0001* sumError_angle)

if  error_position > 0 and error_position < 0.01:
   vel_msg.linear.x = vel_pid
          vel_msg.linear.y = 0
          vel_msg.linear.z = 0
          vel_msg.angular.x = 0
          vel_msg.angular.y = 0
          vel_msg.angular.z = ang_pid
          self.velocity_publisher.publish(vel_msg)
else:
   vel_msg.linear.x = vel_pid
          vel_msg.linear.y = 0
          vel_msg.linear.z = 0
          vel_msg.angular.x = 0
          vel_msg.angular.y = 0
          vel_msg.angular.z = ang_pid
          self.velocity_publisher.publish(vel_msg)
```

```
            self.rate.sleep()


                    xaxes=P_i[0]
                    yaxes=P_i[1]
                    plt.plot(xaxes,yaxes,".")
                    plt.draw()
                    plt.pause(0.00000000000000001)
                    #rate.sleep()





        # If we press control + C, the node will stop.
        rospy.spin()


if __name__ == '__main__':
    try:
        x = TurtleBot()
        x.move2goal()
    except rospy.ROSInterruptException:
        pass
```



Task 4:

Simple PID controller to reach the goal point

```
#!/usr/bin/env python
# license removed for brevity
import rospy
import math
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from math import pow, atan2, sqrt
import time


# PID variable

error_angle = 0
error_linear = 0
```

```
sumError_angle = 0
sumError_linear = 0

rateError_linear = 0
rateError_angle = 0

currentTime = time.time()
previousTime = time.time()

lastError_linear = 0
lastError_angle = 0


class TurtleBot:

    def __init__(self):
        # Creates a node with name 'turtlebot_controller' and make sure it is a
        # unique node (using anonymous=True).
        rospy.init_node('turtlebot_controller', anonymous=True)

        # Publisher which will publish to the topic '/turtle1/cmd_vel'.
        self.velocity_publisher = rospy.Publisher('/cmd_vel',
                                                  Twist, queue_size=10)

        # A subscriber to the topic '/turtle1/pose'. self.update_pose is called
        # when a message of type Pose is received.
        self.pose_subscriber =  rospy.Subscriber("/odom", Odometry, self.update_pose)

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

    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.pose.pose.position.x = round(self.pose.pose.pose.position.x, 4)
        self.pose.pose.pose.position.y = round(self.pose.pose.pose.position.y, 4)

        self.pose.pose.pose.orientation.x = round(self.pose.pose.pose.orientation.x, 4)
        self.pose.pose.pose.orientation.y = round(self.pose.pose.pose.orientation.y, 4)
        self.pose.pose.pose.orientation.z = round(self.pose.pose.pose.orientation.z, 4)
        self.pose.pose.pose.orientation.w = round(self.pose.pose.pose.orientation.w, 4)

    def euclidean_distance(self, goal_pose):
```

```
        """Euclidean distance between current pose and the goal."""
        return sqrt(pow((goal_pose.pose.pose.position.x - self.pose.pose.pose.position.x), 2) +
                    pow((goal_pose.pose.pose.position.y - self.pose.pose.pose.position.y), 2))

    def linear_vel(self, goal_pose, constant=1.5):
        """See video: https://www.youtube.com/watch?v=Qh15Nol5htM."""
        return self.euclidean_distance(goal_pose)

    def steering_angle(self, goal_pose):
        """See video: https://www.youtube.com/watch?v=Qh15Nol5htM."""
        return atan2(goal_pose.pose.pose.position.y - self.pose.pose.pose.position.y,
goal_pose.pose.pose.position.x - self.pose.pose.pose.position.x)

    def angular_vel(self, goal_pose, constant=6):
        """See video: https://www.youtube.com/watch?v=Qh15Nol5htM."""

        orientation_list = [self.pose.pose.pose.orientation.x,
self.pose.pose.pose.orientation.y,self.pose.pose.pose.orientation.z,self.pose.pose.pose.orientation.w]
        (roll, pitch, yaw) = euler_from_quaternion(orientation_list)
        rospy.loginfo('roll %d pitch %d yaw %d', roll, pitch, yaw)


        return (self.steering_angle(goal_pose) - yaw)

    def move2goal(self):

        global currentTime, previousTime, error_linear, sumError_linear, rateError_linear, lastError_linear,
error_angle, sumError_angle, rateError_angle, lastError_angle
        """Moves the turtle to the goal."""
        goal_pose = Odometry()

        # Get the input from the user.
        goal_pose.pose.pose.position.x = float(input("Set your x goal: "))
        goal_pose.pose.pose.position.y = float(input("Set your y goal: "))

        # Please, insert a number slightly greater than 0 (e.g. 0.01).
        distance_tolerance = input("Set your tolerance: ")

        vel_msg = Twist()

        while self.euclidean_distance(goal_pose) >= distance_tolerance:

            currentTime = time.time()
            elapsedTime = (currentTime - previousTime)
```

```
        error_linear = self.linear_vel(goal_pose)

        error_angle = self.angular_vel(goal_pose)

        sumError_linear = sumError_linear + (error_linear * elapsedTime)
        rateError_linear = (error_linear - lastError_linear)/elapsedTime

        sumError_angle = sumError_angle + (error_angle * elapsedTime)
        rateError_angle = (error_angle - lastError_angle)/elapsedTime


        vel_pid = (0.15 * error_linear) + (0 * sumError_linear) + (0 * rateError_linear)

        ang_pid = (0.1 * error_angle) + (0 * sumError_angle) + (0 * rateError_angle)

        # Porportional controller.
        # https://en.wikipedia.org/wiki/Proportional_control

        # Linear velocity in the x-axis.
        vel_msg.linear.x = (vel_pid)
        vel_msg.linear.y = 0
        vel_msg.linear.z = 0

        # Angular velocity in the z-axis.
        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = ang_pid

        # Publishing our vel_msg
        self.velocity_publisher.publish(vel_msg)


        previousTime = currentTime
        lastError_angle = error_angle
        lastError_linear = error_linear

        # Publish at the desired rate.
        self.rate.sleep()

    # Stopping our robot after the movement is over.
    rospy.loginfo("reached")
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
```

```
        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = 0
        self.velocity_publisher.publish(vel_msg)

        # If we press control + C, the node will stop.
        rospy.spin()

if __name__ == '__main__':
    try:
        x = TurtleBot()
        x.move2goal()
    except rospy.ROSInterruptException:
        pass
```