src/turtlebot_controller.py

```
#!/usr/bin/env python3

#Junhyung Park and Ryan Buckton
#turtlebot_controller.py
#This lab will integrate the on-board IMU with the Turtlebot3 controller to turn the
robot 90 degrees left or right.
# 14 March 2023: Changed the P constant and angle that it is detecting
# 12 March 2023: Found 0 division error, added mouseControl, able to turn left 90
degrees when the distance is near, added launch file
# 10 March 2023: Started Lab 3

import rospy, math
from lab1.msg import MouseController
from geometry_msgs.msg import Twist
from squaternion import Quaternion
from sensor_msgs.msg import Imu
#TODO Import the laser message used in ICE8
from sensor_msgs.msg import LaserScan

#TODO Add lamda function
# lambda function to convert rad to deg
RAD2DEG = lambda x: ((x)*180./math.pi)
# convert LaserScan degree from -180 - 180 degs to 0 - 360 degs
DEG_CONV = lambda deg: deg + 360 if deg < 0 else deg

class Controller:
    """Class that controls subsystems on Turtlebot3"""
    #TODO Add Class variables
    DISTANCE = 0.4 # distance from the wall to stop
    K_POS = 1 # proportional constant for slowly stopping as you get closer to the
wall
    MIN_LIN_X = 0.05 # limit m/s values sent to Turtlebot3
    MAX_LIN_X = 0.2 # limit m/s values sent to Turtlebot3

    K_HDG = 0.1 # rotation controller constant
    HDG_TOL = 10 # heading tolerance +/- degrees
    MIN_ANG_Z = 0.5 # limit rad/s values sent to Turtlebot3
    MAX_ANG_Z = 1.5 # limit rad/s values sent to Turtlebot3

    def __init__(self):
        self.curr_yaw = 0
        self.goal_yaw = 0
        self.turning = False

        self.cmd = Twist()

        self.cmd.linear.x = 0.0
        self.cmd.linear.y = 0.0
        self.cmd.linear.z = 0.0
        self.cmd.angular.x = 0.0
        self.cmd.angular.y = 0.0
        self.cmd.angular.z = 0.0
```

```
        self.mouseStatus = False # For mouse control to take over

        #TODO Add instance variables
        self.avg_dist = 0 # Store the average dist off the nose
        self.got_avg = False # Store when an average is calculated

        #TODO A subscriber to the LIDAR topic of interest with a callback to the
callback_lidar() function.
        rospy.Subscriber('scan', LaserScan, self.callback_lidar)

        #A subscriber to the IMU topic of interest with a callback to the
callback_imu() function
        rospy.Subscriber('imu', Imu, self.callback_imu)

        self.pub = rospy.Publisher('cmd_vel', Twist, queue_size = 1)
        rospy.Timer(rospy.Duration(.1), self.callback_controller)

        rospy.Subscriber('mouse_info', MouseController, self.callback_mouseControl)

        self.ctrl_c = False
        rospy.on_shutdown(self.shutdownhook)

    # TODO Add the callback_lidar() function from ICE8, removing print statements and
setting the instance variables, self.avg_dist and self.got_avg
    def callback_lidar(self, scan):
        if not self.ctrl_c:
            degrees = []
            ranges = []

            self.rangeCount = 0 # Reset
            self.rangeSum = 0 # Reset

            # determine how many scans were taken during rotation
            # count = int(scan.scan_time / scan.time_increment) #problem line, divide
by zero issue. Use len(ranges) to find number of scans.
            # Or... Do (anglemax - anglemin)/angleincrement?
            count = len(scan.ranges)

            for i in range(count):
                # using min angle and incr data determine curr angle,
                # convert to degrees, convert to 360 scale
                degrees.append(int(DEG_CONV(RAD2DEG(scan.angle_min +
scan.angle_increment*i))))
                rng = scan.ranges[i]

                # ensure range values are valid; set to 0 if not
                if rng < scan.range_min or rng > scan.range_max:
                    ranges.append(0.0)
                else:
                    ranges.append(rng)

            # python way to iterate two lists at once!
            for deg, rng in zip(degrees, ranges):
                # TODO: sum and count the ranges 20 degrees off the nose of the robot
                if deg >= 350 or deg <= 10:
```

```
                    self.rangeCount += 1
                    self.rangeSum += rng

            if self.rangeCount != 0:
                self.avg_dist = self.rangeSum / self.rangeCount # There are instances
when there is no object detected, and rangeCount would equal to 0, causing error

    # The IMU provides yaw from -180 to 180. This function
    # converts the yaw (in degrees) to 0 to 360
    def convert_yaw (self, yaw):
        return 360 + yaw if yaw < 0 else yaw

    def callback_imu(self, imu):
        if not self.ctrl_c:
            # create a quaternion using the x, y, z, and w values
            # from the correct imu message

            # w, x, y, and z is whithin orientation of imu
            imu_q = Quaternion(imu.orientation.w, imu.orientation.x,
imu.orientation.y, imu.orientation.z)

            # convert the quaternion to euler in degrees
            imu_e = imu_q.to_euler(degrees=True)

            # get the yaw component of the euler
            yaw = imu_e[2]

            # convert yaw from -180 to 180 to 0 to 360
            self.curr_yaw = self.convert_yaw(yaw)

    #TODO When not turning and you have an average LIDAR reading, calculate the
distance error (actual dist - desired dist)
    #TODO use that to drive your robot straight at a proportional rate (very similar
to how we calculated the turn rate in lab 2).
    def callback_controller(self, event):
        # local variables do not need the self
        yaw_err = 0
        ang_z = 0
        lin_x = 0

        distanceError = self.avg_dist - self.DISTANCE

        if not self.mouseStatus: # If the mouseControl did not take over
            if (distanceError == -0.4 or distanceError > 0) and not self.turning:
                lin_x = self.K_POS * distanceError

                # Limit the linear speed of the robot to MIN_LIN_X and MAX_LIN_X.
                if lin_x > self.MAX_LIN_X or distanceError == -0.4:
                    lin_x = self.MAX_LIN_X
                if lin_x < self.MIN_LIN_X:
                    lin_x = self.MIN_LIN_X

            else:
                if not self.turning:
                    #TODO If within DISTANCE of a wall, then stop and start turning
(left or right, you decide).
```

```
                    self.turning = True

                    lin_x = 0
                    self.goal_yaw = self.curr_yaw + 90

                elif self.turning:
                    # turn until goal is reached
                    yaw_err = self.goal_yaw - self.curr_yaw

                    # determine if robot should turn clockwise or counterclockwise
                    if yaw_err > 180:
                        yaw_err = yaw_err - 360
                    elif yaw_err < -180:
                        yaw_err = yaw_err + 360

                    # proportional controller that turns the robot until goal
                    # yaw is reached
                    ang_z = self.K_HDG * yaw_err

                    #Add negative test
                    if abs(ang_z) < self.MIN_ANG_Z and ang_z > 0:
                        ang_z = self.MIN_ANG_Z
                    elif abs(ang_z) < self.MIN_ANG_Z and ang_z < 0:
                        ang_z = -self.MIN_ANG_Z
                    elif ang_z > self.MAX_ANG_Z:
                        ang_z = self.MAX_ANG_Z
                    elif abs(ang_z) > self.MAX_ANG_Z:
                        ang_z = -self.MAX_ANG_Z


                    # check goal orientation
                    if abs(yaw_err) < self.HDG_TOL:
                        self.turning = False
                        ang_z = 0

            # set twist message and publish
            # TODO Save the linear x and angular z values to the Twist message and
publish.
            self.cmd.angular.z = ang_z
            self.cmd.linear.x = lin_x
            # Publish cmd
            self.pub.publish(self.cmd)

    def callback_mouseControl(self, mouseInfo):
        if mouseInfo.status.data: # If mouseControl was activated
            self.mouseStatus = True # Let mouseControl take over

            #Scale xPos from -1 to 1 to -.5 to .5
            scaled_xPos = -(mouseInfo.xPos)/2

            # Set angular z in Twist message to the scaled value in the appropriate
direction
            self.cmd.angular.z = scaled_xPos

            # Scale yPos from -1 to 1 to -.5 to .5
```

```
            scaled_yPos = -(mouseInfo.yPos)/2

            # Set linear x in Twist message to the scaled value in the appropriate
direction
            self.cmd.linear.x = scaled_yPos

            # 8 publish the Twist message
            self.pub.publish(self.cmd)
        else:
            self.mouseStatus = False # mouseControl is not being used

    def shutdownhook(self):
        print("Controller exiting. Halting robot.")
        self.ctrl_c = True
        # 9 force the linear x and angular z commands to 0 before halting
        self.cmd.linear.x = 0
        self.cmd.angular.z = 0

        self.pub.publish(self.cmd)

if __name__ == '__main__':
    rospy.init_node('controller')
    c = Controller()
    rospy.spin()
```