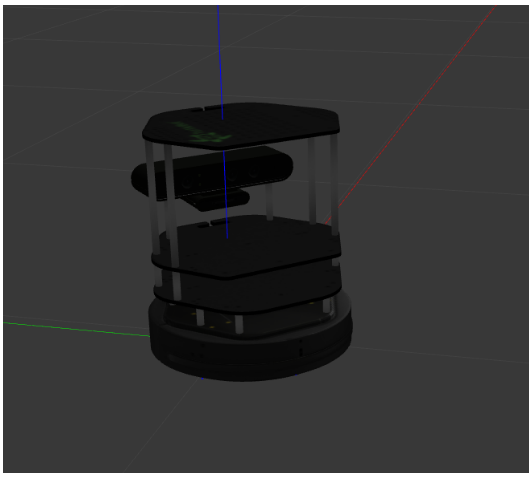
# Robotics Lab 3-4 Post Lab Assignment

Arthur Gomes
190929090
Batch A1
Roll No. 22, Section A

## Aim:

Move a robot in a Gazebo simulation in the shape of a square 10 times. Use 2 nodes, one to control the robot's velocity and the other to track and visualize its position. Repeat the same for the Turtlebot2 in the Robotics Lab and mark final and initial positions.

Square Dimensions: 3m x 3m (centred at (0,0) )
Simulation Robot: Turtlebot2 with kobuki base and hexagon stacks



## Programs:

## Position Controller

A closed loop controller is implemented to move the robot. PI gains are used for linear motion and P gain is used for angular motion. The node receives position feedback from wheel odometry and publishes planar twist messages to the desired velocity topic.

```
#! /usr/bin/env python
# created by: Arthur Gomes

import rospy
import math
from geometry_msgs.msg import Twist
```

```python
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion

class position_controller:

   '''
   A position controller for kobuki mobile base.
   Takes desired position in world frame as in input
   outputs velocity commands
   assume the world frame is always situated at robot starting position
   '''

   def __init__(self, node_name, velocity_topic, position_topic):

       # load gains from param server
       self.linear_pid = rospy.get_param('~linear_pid', [0.35, 0.001, 0.0])
       self.angular_pid = rospy.get_param('~angular_pid', [1.5, 0.0, 0.0])

       # load max speed for base from server
       self.linear_max = rospy.get_param('~linear_max',0.9)
       self.angular_max = rospy.get_param('~angular_max', 3.0)

       # load goal tolerances for base from server
       self.linear_tolerance = rospy.get_param('~linear_tolerance', 0.01)
       self.angular_tolerance = rospy.get_param('~angular_tolerance', 0.01)

       # set initial configuration of bot
       self.current_config = [0,0,0]

       # create twist msg
       self.vel_msg = Twist()

       # initialize some node stuff
       rospy.init_node(node_name)
       self.vel_pub = rospy.Publisher(velocity_topic, Twist, queue_size=1)
       rospy.Subscriber(position_topic, Odometry, self.odom_callback)
       self.rate = rospy.Rate(20)

   def odom_callback(self,msg):
       self.current_config[0] = msg.pose.pose.position.x
       self.current_config[1] = msg.pose.pose.position.y
       quat= msg.pose.pose.orientation
       _,_,self.current_config[2] = euler_from_quaternion([quat.x, quat.y, quat.z, quat.w])
       # print(math.degrees(self.current_config[2]))

   def goto(self,x,y):

       print('moving to ',x,y)
       linear_EA = 0
       angular_EA = 0
       prev_linear_E = 0
       prev_angular_E = 0

       self.vel_msg.linear.x = 0
       self.vel_msg.angular.z = 0

       Ea = self.angular_error(x,y)

       while abs(Ea) > self.angular_tolerance and not rospy.is_shutdown():
```

```
            Ea = self.angular_error(x,y)
            self.vel_msg.linear.x = 0
            w = self.angular_pid[0]*Ea + self.angular_pid[1]*angular_EA + self.angular_pid[2]*(Ea -
prev_angular_E)
            prev_angular_E = Ea
            angular_EA += Ea

            # cap angular velocity and publish
            self.vel_msg.angular.z = self.cap_angular_velocity(w)
            self.vel_pub.publish(self.vel_msg)
            self.rate.sleep()
            # print(self.vel_msg)

        El = self.linear_error(x,y)
        while abs(El) > self.linear_tolerance and not rospy.is_shutdown():

            Ea = self.angular_error(x,y)
            El = self.linear_error(x,y)
            w = self.angular_pid[0]*Ea + self.angular_pid[1]*angular_EA + self.angular_pid[2]*(Ea -
prev_angular_E)
            prev_angular_E = Ea
            angular_EA += Ea
            v = self.linear_pid[0]*El + self.linear_pid[1]*linear_EA + self.linear_pid[2]*(El -
prev_linear_E)
            prev_linear_E = El
            linear_EA += El

            # cap velocities and publish
            self.vel_msg.angular.z = self.cap_angular_velocity(w)
            self.vel_msg.linear.x = self.cap_linear_velocity(v)
            self.vel_pub.publish(self.vel_msg)
            self.rate.sleep()
            # print(self.vel_msg)

        rospy.loginfo('reached')

    # def execute_trajectory(self, trajs):
    #     rospy.loginfo('starting trajectory')
    #     for traj in trajs:
    #         pass

    def angular_error(self,x,y):
        cx = self.current_config[0]
        cy = self.current_config[1]
        yaw = self.current_config[2]

        a = x*math.cos(yaw) + y*math.sin(yaw) - cx*math.cos(yaw) - cy*math.sin(yaw)
        b = -x*math.sin(yaw) + y*math.cos(yaw) + cx*math.sin(yaw) - cy*math.cos(yaw)
        return math.atan2(b,a)

    def linear_error(self,x,y):
        return math.sqrt((x - self.current_config[0])**2 + (y - self.current_config[1])**2)

    def cap_linear_velocity(self, v):
        if v > self.linear_max:
            return self.linear_max
        else:
            return v

    def cap_angular_velocity(self, w):
```

```
        if w > self.angular_max:
            return self.angular_max
        else:
            return w


o = position_controller('kobuki_velocity_controller', '/cmd_vel', '/odom')

# add trajectory here
o.goto(1.5,1.5)

for i in range(10):
   o.goto(1.5,-1.5)
   o.goto(-1.5,-1.5)
   o.goto(-1.5, 1.5)
   o.goto(1.5,1.5)

o.goto(0,0)

rospy.spin()
```

## Position Tracker And Visualizer

This node subscribes to wheel odometry and publishes the path taken by the robot as well as the ideal to RVIZ for visualization.

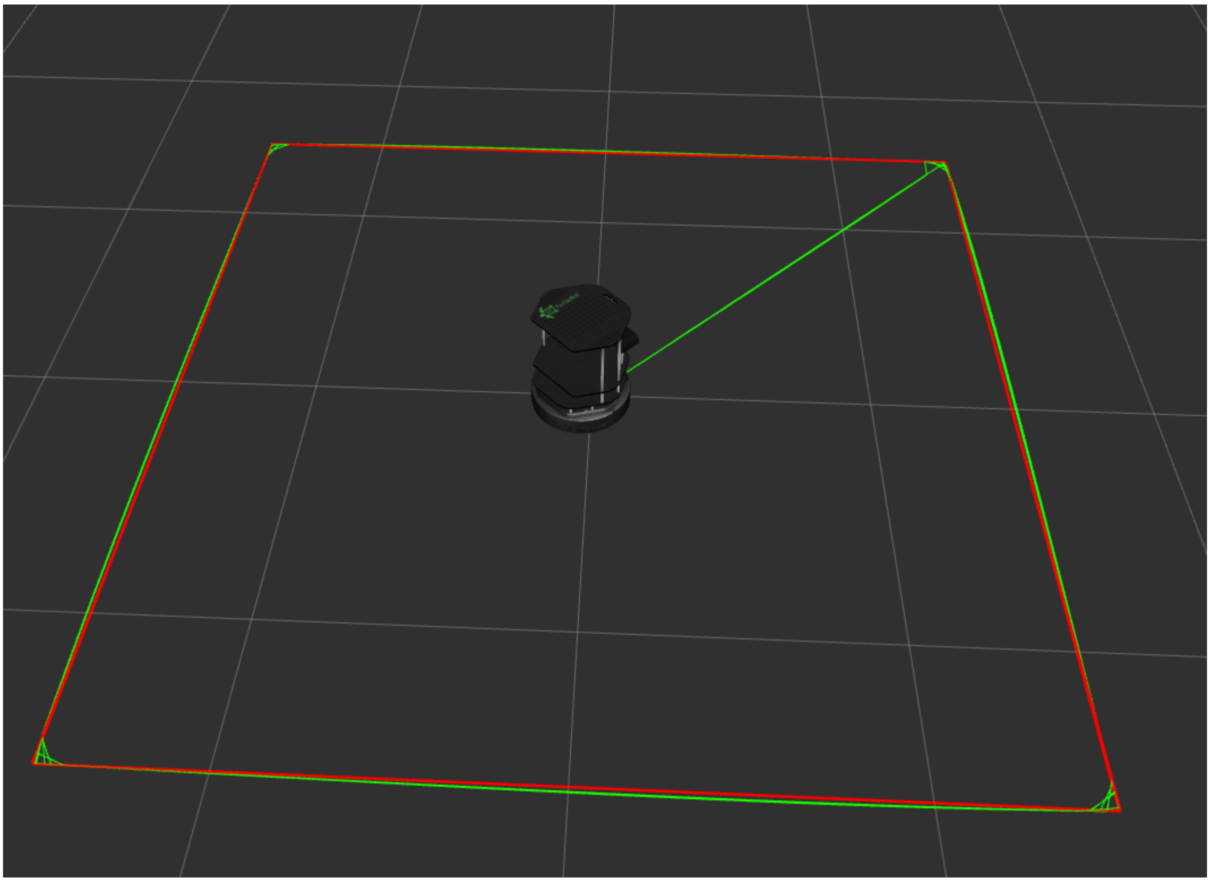
```
#! /usr/bin/env python
# Created by: Arthur Gomes

import rospy
import csv
import math
from nav_msgs.msg import Odometry, Path
from std_msgs.msg import ColorRGBA
from visualization_msgs.msg import Marker
from geometry_msgs.msg import Point, PoseStamped

class tracker:

   '''
   This class is written for monitoring robot odometry by publishing rviz markers
   '''
   def __init__(self, position_topic):
       self.prev = [0,0]
       rospy.init_node('tracker')
       rospy.Subscriber(position_topic, Odometry, self.odom_callback)
       self.marker_pub = rospy.Publisher('/visualization_marker', Marker, queue_size=10)
       self.path_pub = rospy.Publisher('/robot_path', Path, queue_size=1)

       # define marker msg object
       self.define_marker_object(1, 0, 0, 1.0)

       #define path msg
       self.path_msg = Path()
       self.path_msg.header.frame_id = '/odom'

       #print ideal path
       self.publish_line( 1.5, 1.5, 1.5,-1.5, 1)
       self.publish_line( 1.5,-1.5,-1.5,-1.5, 2)
       self.publish_line(-1.5,-1.5,-1.5, 1.5, 3)
```

```
        self.publish_line(-1.5, 1.5, 1.5, 1.5, 4)

        rospy.loginfo('listening to odom')
        rospy.spin()

    def define_marker_object(self,r,g,b,a):
        self.marker = Marker()
        self.marker.scale.x = 0.01
        self.marker.scale.y = 0.01
        self.marker.scale.z = 0.01
        self.marker.action = Marker.ADD
        self.marker.header.frame_id = '/odom'
        self.marker.type = Marker.LINE_STRIP


        # self.line_color = ColorRGBA()
        # self.line_color.r = r
        # self.line_color.g = g
        # self.line_color.b = b
        # self.line_color.a = a

        self.marker.color.a = a
        self.marker.color.r = r
        self.marker.color.g = g
        self.marker.color.b = b
        self.start_point = Point()
        self.end_point = Point()

    def publish_line(self, x1,y1, x2,y2, id):

        self.marker.pose.orientation.x = 0
        self.marker.pose.orientation.y = 0
        self.marker.pose.orientation.z = 0.0
        self.marker.pose.orientation.w = 1.0
        self.marker.id = id
        self.marker.pose.position.x = 0
        self.marker.pose.position.y = 0
        self.marker.pose.position.z = 0.0

        self.marker.points = []
        #start point
        self.start_point.x = x1
        self.start_point.y = y1
        #end point
        self.end_point.x = x2
        self.end_point.y = y2

        self.marker.points.append(self.start_point)
        self.marker.points.append(self.end_point)

        while self.marker_pub.get_num_connections() == 0:
            pass
        self.marker_pub.publish(self.marker)

    def odom_callback(self, msg):
        temp = [msg.pose.pose.position.x, msg.pose.pose.position.y]
        if abs(math.sqrt( (temp[0]-self.prev[0])**2 + (temp[1]-self.prev[1])**2 )) > 0.1:

            pose_msg = PoseStamped()
            pose_msg.header.frame_id = '/odom'
```

```
            pose_msg.pose.position.x = temp[0]
            pose_msg.pose.position.y = temp[1]

            self.path_msg.poses.append(pose_msg)
            self.path_pub.publish(self.path_msg)

            self.prev = temp[:]

if __name__ == '__main__':
    obj = tracker('/odom')
```

## Simulation Results

The square trajectory was completed 10 times by the robot with minimal diversion



Red line - Ideal trajectory
Green line - Actual trajectory

## Hardware Test

The same test was run on the turtlebot 2 in the Robotics lab.
The deviation was much more significant.