SC 627 Motion Planning

Third Assignment Project

<u>Omar Kashmar</u>

<u>213011004</u>

<u>Group 4</u>

03/May/2023

<u>Under Guidance of</u>

<u>Prof. Arpita sinha</u>

Multi-Agent Swarm Control for 4 Turtle-Bots Robots: Synchronization and Balanced Consensus

Abstract: This report presents an implementation of a multi-agent control algorithm for a swarm of four Turtlebot robots in a ROS and Gazebo simulation environment. The code achieves both synchronization and balanced consensus behaviors using a unicycle model. The robots communicate and adjust their motions to achieve the desired swarm behavior.

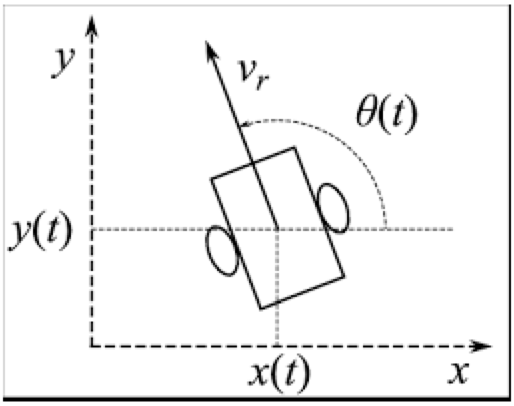

Figure 1: unicycle model

1. Introduction: Swarm robotics involves coordinating the actions of multiple robots to achieve collective behaviors. The code presented in this report demonstrates how a swarm of four Turtlebot robots can be controlled to perform synchronization and balanced consensus tasks using the Robot Operating System (ROS) and the Gazebo simulation environment.

2. Methodology: The code employs the following key components:

- **Odometry Callbacks:** The positions and orientations of each robot are retrieved from the odometry messages. Quaternion orientations are converted to Euler angles for easier calculations.
- **Control Loop:** The control loop calculates control signals `u1` through `u4` based on the difference in orientations between the robots. In synchronization mode, negative control gains are used, while positive gains are used for balanced consensus.
- **Control Commands:** The calculated control signals are applied to the angular velocities of the robots, effectively controlling their movements. The control commands are published to the `/cmd_vel` topics for each robot.
- **Rate Control:** The control loop operates at a defined rate to ensure consistent execution.

3. Results: The code successfully achieves both synchronization and balanced consensus behaviors in the simulated swarm of Turtlebot robots. Depending on the chosen condition (`condition = 0` for synchronization, `condition = 1` for balanced consensus), the robots move in a coordinated manner to achieve the desired behavior.

4. Discussion: The code demonstrates the principles of multi-agent control in a simulated environment. It showcases how robots can collaborate and adjust their actions based on their relative positions and orientations. The choice of control parameters (`k`, `radius`, `n`, etc.) affects the behavior of the swarm and can be adjusted for different scenarios

The code

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

import rospy
from math import sin
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
# Odometry is given as a quaternion, but for the algorithm we'll need to find the orientaion
theta by converting to euler angle
from tf.transformations import euler_from_quaternion


def odometryCb1(msg):
    global tur1_x, tur1_y, tur1_theta

    tur1_x = msg.pose.pose.position.x
    tur1_y = msg.pose.pose.position.y

    quat = msg.pose.pose.orientation
    (tur1_roll, tur1_pitch, tur1_theta) = euler_from_quaternion(
        [quat.x, quat.y, quat.z, quat.w])

def odometryCb2(msg):
    global tur2_x, tur2_y, tur2_theta

    tur2_x = msg.pose.pose.position.x
    tur2_y = msg.pose.pose.position.y

    quat = msg.pose.pose.orientation
    (tur2_roll, tur2_pitch, tur2_theta) = euler_from_quaternion(
        [quat.x, quat.y, quat.z, quat.w])

def odometryCb3(msg):
    global tur3_x, tur3_y, tur3_theta

    tur3_x = msg.pose.pose.position.x
    tur3_y = msg.pose.pose.position.y

    quat = msg.pose.pose.orientation
    (tur3_roll, tur3_pitch, tur3_theta) = euler_from_quaternion(
        [quat.x, quat.y, quat.z, quat.w])

def odometryCb4(msg):
    global tur4_x, tur4_y, tur4_theta

    tur4_x = msg.pose.pose.position.x
    tur4_y = msg.pose.pose.position.y

    quat = msg.pose.pose.orientation
```

```
    (tur4_roll, tur4_pitch, tur4_theta) = euler_from_quaternion(
        [quat.x, quat.y, quat.z, quat.w])




if __name__ == "__main__":

    global tur1_x, tur1_y, tur1_theta,tur2_x, tur2_y, tur2_theta,tur3_x, tur3_y,
tur3_theta,tur4_x, tur4_y, tur4_theta
    tur1_x, tur1_y, tur1_theta,tur2_x, tur2_y, tur2_theta,tur3_x, tur3_y, tur3_theta,tur4_x,
tur4_y, tur4_theta = [0]*12


    rospy.init_node('controller')
    rospy.loginfo('My node has been started')
    publish_to_cmd_vel1 = rospy.Publisher('/tb3_0/cmd_vel', Twist, queue_size=10)
    publish_to_cmd_vel2 = rospy.Publisher('/tb3_3/cmd_vel', Twist, queue_size=10)
    publish_to_cmd_vel3 = rospy.Publisher('/tb3_4/cmd_vel', Twist, queue_size=10)
    publish_to_cmd_vel4 = rospy.Publisher('/tb3_5/cmd_vel', Twist, queue_size=10)
    # A subscriber to call the orientation function
    sub1 = rospy.Subscriber("/tb3_0/odom", Odometry, odometryCb1)
    sub2 = rospy.Subscriber("/tb3_3/odom", Odometry, odometryCb2)
    sub3 = rospy.Subscriber("/tb3_4/odom", Odometry, odometryCb3)
    sub4 = rospy.Subscriber("/tb3_5/odom", Odometry, odometryCb4)



    # create an object of Twist data

    move_the_bot1 = Twist()
    move_the_bot2 = Twist()
    move_the_bot3 = Twist()
    move_the_bot4 = Twist()

    condition = 1              # 0 for Synchronization and 1 for balanced

    k = 3
    radius = 0.4
    n = 4
    if condition ==0 :
        k = -1
        bias = 0
        linear = 0.1
    elif condition == 1:
        k = 1
        bias = 0.2
        linear = radius * bias
    rate = rospy.Rate(10)
```

```
    move_the_bot1.linear.x = linear
    move_the_bot2.linear.x = linear
    move_the_bot3.linear.x = linear
    move_the_bot4.linear.x = linear

    publish_to_cmd_vel1.publish(move_the_bot1)
    publish_to_cmd_vel2.publish(move_the_bot2)
    publish_to_cmd_vel3.publish(move_the_bot3)
    publish_to_cmd_vel4.publish(move_the_bot4)


    while not rospy.is_shutdown():
        # Synchronizing mode - Negative k
        # print(tur1_theta,tur2_theta,tur3_theta,tur4_theta)

        # Balance configuration - Positive k
        print((tur1_x+tur2_x+tur3_x+tur4_x)/n, (tur1_y+tur2_y+tur3_y+tur4_y)/n)

        u1 = -k/n*(sin(tur2_theta-tur1_theta)+sin(tur3_theta-tur1_theta)+sin(tur4_theta-tur1_theta))
        u2 = -k/n*(sin(tur1_theta-tur2_theta)+sin(tur3_theta-tur2_theta)+sin(tur4_theta-tur2_theta))
        u3 = -k/n*(sin(tur1_theta-tur3_theta)+sin(tur2_theta-tur3_theta)+sin(tur4_theta-tur3_theta))
        u4 = -k/n*(sin(tur1_theta-tur4_theta)+sin(tur2_theta-tur4_theta)+sin(tur3_theta-tur4_theta))
        #u1 = -k/n*(sin(tur1_theta-tur2_theta))
        #u2 = -k/n*(sin(tur2_theta-tur3_theta))
        #u3 = -k/n*(sin(tur3_theta-tur4_theta))
        #u4 = -k/n*(sin(tur4_theta-tur1_theta))
        move_the_bot1.angular.z = u1 + bias
        move_the_bot2.angular.z = u2 + bias
        move_the_bot3.angular.z = u3 + bias
        move_the_bot4.angular.z = u4 + bias

        publish_to_cmd_vel1.publish(move_the_bot1)
        publish_to_cmd_vel2.publish(move_the_bot2)
        publish_to_cmd_vel3.publish(move_the_bot3)
        publish_to_cmd_vel4.publish(move_the_bot4)

        rate.sleep()

    rospy.spin()
```

Demo:

Provide google drive link for Implementation in the real world and simulation (ARMAS Lab) Syscon department:

https://drive.google.com/drive/folders/1Gg0PJ28U47C_sUtsRL8fssxPrjtp3Njv?usp=sharing.