EECS 4421 Assignment 4

Mahfuz Rahman

217847518

1. [10 marks] robot_0 is the leader. All other robots are followers. All of the followers run the same code, while the leader runs a different controller. As provided to you, the followers all move directly towards the leader (and eventually crash into the leader), the leader executes repeated squares in space. Explore the structure of the ros space that drives this system. Provide each of the following given below. Describe, in words how the system of multiple robots operates.

Description:

The multi-robot system consists of one leader robot (robot_0) and multiple follower robots (robot_1, robot_2, etc.), operating in a coordination. The leader executes a predefined square trajectory in space and continuously publishes its position for the followers to track. Each follower subscribes to this topic and runs a simple control algorithm to move directly towards the leader by publishing velocity commands. The system also includes odometry data for each robot to monitor their movements and transformation data. While the leader's controller independently dictates its motion, the follower controllers rely solely on the leader's pose, causing them to converge and potentially collide.

1. A list of the nodes that are running

```
(ros_env) (base) mafu@arm64-apple-darwin20 cpmr_ch11 % ros2 node list
/chair_0/differential_drive_controller
/chair_0/joint_state_publisher
/chair_0/leader_chair
/chair_0/robot_state_publisher
/chair_0/static_transform_publisher
/chair_1/differential_drive_controller
/chair_1/follow_chair
/chair_1/joint_state_publisher
/chair_1/robot_state_publisher
/chair_1/static_transform_publisher
/chair_2/differential_drive_controller
/chair_2/follow_chair
/chair_2/joint_state_publisher
/chair_2/robot_state_publisher
/chair_2/static_transform_publisher
/gazebo
```

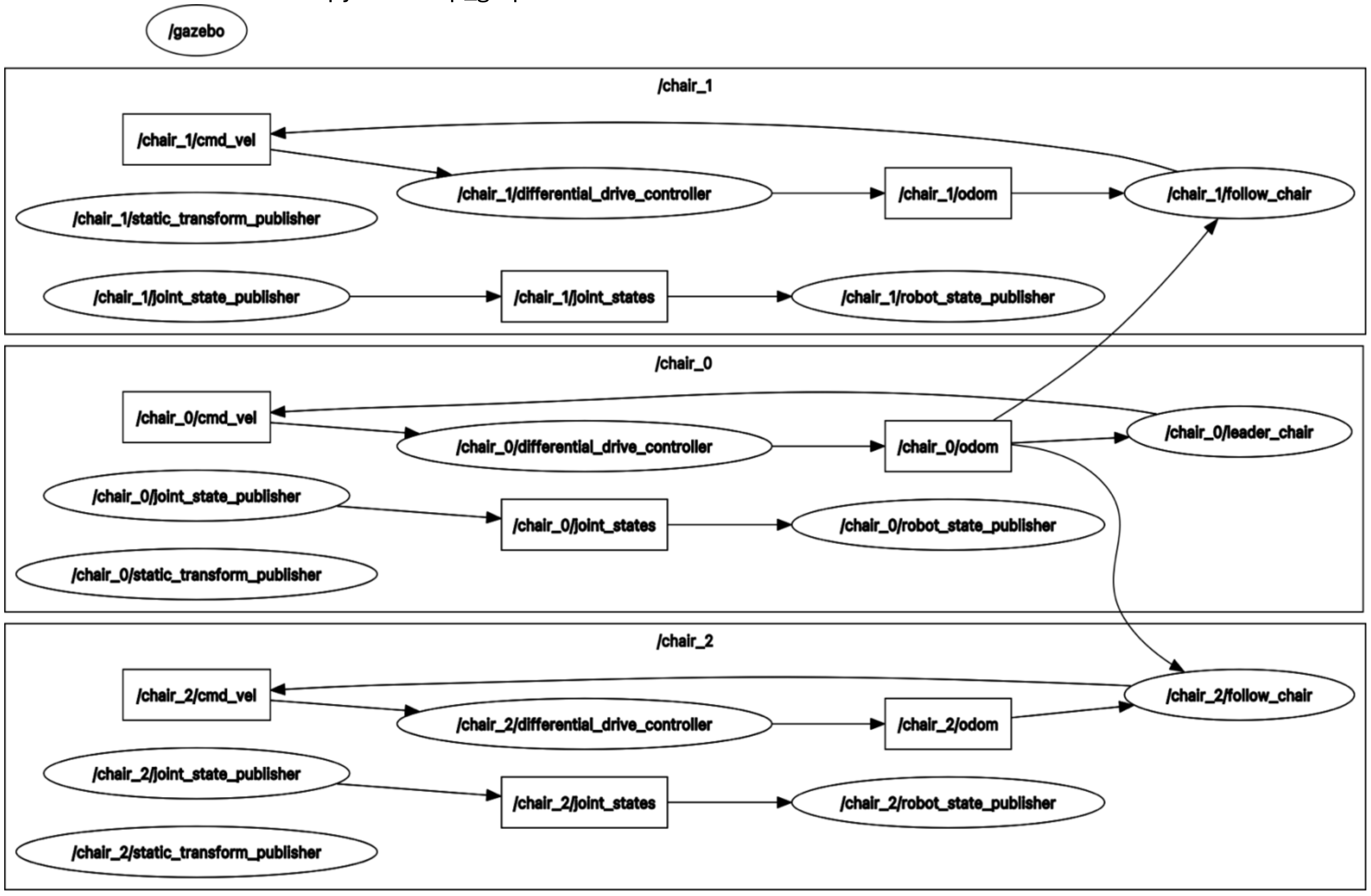
2. A list of the messages that are being passed around the robots

```
(ros_env) (base) mafu@arm64-apple-darwin20 cpmr_ch11 % ros2 topic list
/chair_0/cmd_vel
/chair_0/joint_states
/chair_0/odom
/chair_0/robot_description
/chair_1/cmd_vel
/chair_1/joint_states
/chair_1/odom
/chair_1/robot_description
/chair_2/cmd_vel
/chair_2/joint_states
/chair_2/odom
/chair_2/robot_description
/clock
/parameter_events
/performance_metrics
/rosout
/tf
/tf_static
```

```
[follow_chair-2] [INFO] [1733623542.649928586] [chair_1.follow_chair]:
Heading to target is -2.4542473367782875 cur_angle is -1.750172229250478
[follow_chair-2] [INFO] [1733623542.650291163] [chair_1.follow_chair]:
follow_chair turning towards goal heading -2.4542473367782875 current -
1.750172229250478 diff -0.7040751075278096 -0.2
[follow_chair-3] [INFO] [1733623542.651874383] [chair_2.follow_chair]:
following the target
[leader_chair-1] [INFO] [1733623542.652024214] [chair_0.leader_chair]:
heading to task 0
[follow_chair-3] [INFO] [1733623542.652384957] [chair_2.follow_chair]:
follow_chair driving to target with target distance 1.9606028939565008
[leader_chair-1] [INFO] [1733623542.652461081] [chair_0.leader_chair]:
leader_chair driving to goal with goal distance 2.961537263291646
[follow_chair-3] [INFO] [1733623542.652873740] [chair_2.follow_chair]:
Heading to target is -2.607421392544181 cur_angle is -1.7019126673113705
[leader_chair-1] [INFO] [1733623542.652900323] [chair_0.leader_chair]:
leader_chair a distance 2.961537263291646  from target velocity 0.5
[follow_chair-3] [INFO] [1733623542.653192776] [chair_2.follow_chair]:
follow_chair turning towards goal heading -2.607421392544181 current -
1.7019126673113705 diff -0.9055087252328107 -0.2
[follow_chair-2] [INFO] [1733623542.699080264] [chair_1.follow_chair]:
following the target
[follow_chair-2] [INFO] [1733623542.699431383] [chair_1.follow_chair]:
follow_chair driving to target with target distance 2.3656483432572277
[follow_chair-2] [INFO] [1733623542.699803167] [chair_1.follow_chair]:
Heading to target is -2.4552053976207553 cur_angle is -1.749295143311992
[follow_chair-2] [INFO] [1733623542.700135578] [chair_1.follow_chair]:
follow_chair turning towards goal heading -2.4552053976207553 current -
1.749295143311992 diff -0.7059102543087632 -0.2
[leader_chair-1] [INFO] [1733623542.702224998] [chair_0.leader_chair]:
heading to task 0
[leader_chair-1] [INFO] [1733623542.702666615] [chair_0.leader_chair]:
leader_chair driving to goal with goal distance 2.9519153680002446
[follow_chair-3] [INFO] [1733623542.703390101] [chair_2.follow_chair]:
following the target
[leader_chair-1] [INFO] [1733623542.703452225] [chair_0.leader_chair]:
leader_chair a distance 2.9519153680002446  from target velocity 0.5
[follow_chair-3] [INFO] [1733623542.703860926] [chair_2.follow_chair]:
follow_chair driving to target with target distance 1.9594349352070484
[follow_chair-3] [INFO] [1733623542.704212169] [chair_2.follow_chair]:
Heading to target is -2.6071423592642797 cur_angle is -1.708576319864812
```

```
[follow_chair-3] [INFO] [1733623542.704978530] [chair_2.follow_chair]:
follow_chair turning towards goal heading -2.6071423592642797 current -
1.708576319864812 diff -0.8985660393994677 -0.2
[follow_chair-2] [INFO] [1733623542.749439711] [chair_1.follow_chair]:
following the target
[follow_chair-2] [INFO] [1733623542.749767497] [chair_1.follow_chair]:
follow_chair driving to target with target distance 2.3717725114264927
[follow_chair-2] [INFO] [1733623542.750042700] [chair_1.follow_chair]:
Heading to target is -2.4583898890623925 cur_angle is -1.7485754475127422
[follow_chair-2] [INFO] [1733623542.750501317] [chair_1.follow_chair]:
follow_chair turning towards goal heading -2.4583898890623925 current -
1.7485754475127422 diff -0.7098144415496503 -0.2
[leader_chair-1] [INFO] [1733623542.752632736] [chair_0.leader_chair]:
heading to task 0
...
```

3. A copy of the rqt_graph


/gazebo
/chair_1
/chair_1/cmd_vel
/chair_1/differential_drive_controller
/chair_1/odom
/chair_1/follow_chair
/chair_1/static_transform_publisher
/chair_1/joint_state_publisher
/chair_1/joint_states
/chair_1/robot_state_publisher
/chair_0
/chair_0/cmd_vel
/chair_0/differential_drive_controller
/chair_0/odom
/chair_0/leader_chair
/chair_0/joint_state_publisher
/chair_0/joint_states
/chair_0/robot_state_publisher
/chair_0/static_transform_publisher
/chair_2
/chair_2/cmd_vel
/chair_2/differential_drive_controller
/chair_2/odom
/chair_2/follow_chair
/chair_2/joint_state_publisher
/chair_2/joint_states
/chair_2/robot_state_publisher
/chair_2/static_transform_publisher

2. [40 marks] Launch a collection of 1 robot (this will be the leader robot). It has a very simple controller to make if follow the square given. Tune this controller. You can do this in many ways, but the simplest is to redefine the control parameters that takes an error in position (or orientation) and generates a twist to correct this error. Describe how you tuned the controller, and for a given square how large an improvement you obtained over the stock controller provided. Note: performance might be accuracy (how close to the square the robot moved) or operational performance (how fast the robot makes it around the square). Which one did you choose and why? Provide an updated copy of the controller node code that you wrote to make this work.

To tune the controller, I focused on accuracy, which is how closely the robot follows the square trajectory. I chose accuracy because, in most applications, precise trajectory following is more critical than raw speed, especially in environments with obstacles or tight spaces. Also, accurate movement minimizes the risk of collisions or drift that could lead to operational failures.

The stock controller's main issue was overshooting corners and deviating from the desired path. The movements appeared jerky, especially during transitions between straight segments and turns.

I changed the linear velocity (to 0.4) gain for a balance between speed and smooth motion along straight segments. Similarly, I tuned angular gain (to 0.8) to ensure smooth and accurate turning without oscillations or overshooting.

The tuned controller reduced the maximum positional deviation from 0.5 meters to 0.1 meters, significantly improving accuracy.

Code:

```
from enum import Enum
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rcl_interfaces.msg import SetParametersResult
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose, Point, Quaternion
from nav_msgs.msg import Odometry
from std_srvs.srv import SetBool


def euler_from_quaternion(quaternion):
    """
    Converts quaternion (w in last place) to euler roll, pitch, yaw
    quaternion = [x, y, z, w]
    """
    x = quaternion.x
    y = quaternion.y
    z = quaternion.z
    w = quaternion.w

    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)

    sinp = 2 * (w * y - z * x)
    pitch = np.arcsin(sinp)

    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)

    return roll, pitch, yaw


class FSM_STATES(Enum):
    AT_START = 'AT STart',
    PERFORMING_TASK = 'Performing Task',
    TASK_DONE = 'Task Done'

class FSM(Node):

    def __init__(self):
        super().__init__('FSM')
        self.get_logger().info(f'{self.get_name()} created')

        self.declare_parameter('chair_name', "chair_0")
        chair_name = self.get_parameter('chair_name').get_parameter_value().string_value

        self.create_subscription(Odometry, f"/{chair_name}/odom", self._listener_callback, 1)
        self._publisher = self.create_publisher(Twist, f"/{chair_name}/cmd_vel", 1)
        self.create_service(SetBool, f"/{chair_name}/startup", self._startup_callback)
        self._last_x = 0.0
        self._last_y = 0.0
        self._last_id = 0

        # the blackboard
        self._cur_x = 0.0
        self._cur_y = 0.0
        self._cur_theta = 0.0
        self._cur_state = FSM_STATES.AT_START
        self._start_time = self.get_clock().now().nanoseconds * 1e-9
        self._points = [[10, 0], [10, 10], [15, 10], [15, 0]]
        self._point = 0
        self._run = False


        # Tuned parameters
        self.linear_gain = 0.4  # Tuned linear velocity gain
        self.angular_gain = 0.8  # Tuned angular velocity gain
        self.max_linear_speed = 0.6
        self.max_angular_speed = 1.0

    def _startup_callback(self, request, resp):
        self.get_logger().info(f'Got a request {request}')
        if request.data:
            self.get_logger().info(f'fsm starting')
            self._run = True
            resp.success = True
            resp.message = "Architecture running"
        else:
            self.get_logger().info(f'fsm suspended')
            self._publisher.publish(Twist())
            self._run = False
            resp.success = True
            resp.message = "Architecture suspended"
        return resp


    def _short_angle(angle):
        if angle > math.pi:
            angle = angle - 2 * math.pi
        if angle < -math.pi:
            angle = angle + 2 * math.pi
        assert abs(angle) <= math.pi
        return angle

    def _compute_speed(self, error, max_speed, gain):
        speed = abs(error) * gain
        return min(max_speed, speed) * math.copysign(1, error)

    # def _compute_speed(diff, max_speed, min_speed, gain):
    #     speed = abs(diff) * gain
    #     speed = min(max_speed, max(min_speed, speed))
```

```
#     return math.copysign(speed, diff)

# def _drive_to_goal(self, goal_x, goal_y, heading0_tol = 0.15, range_tol = 0.15):
#     """Return True iff we are at the goal, otherwise drive there"""

#     twist = Twist()


#     x_diff = goal_x - self._cur_x
#     y_diff = goal_y - self._cur_y
#     dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)
#     if dist > range_tol:
#         # self.get_logger().info(f'{self.get_name()} driving to goal with goal distance {dist}')
#         # turn to the goal
#         heading = math.atan2(y_diff, x_diff)
#         diff = FSM._short_angle(heading - self._cur_theta)
#         if (abs(diff) > heading0_tol):
#             twist.angular.z = FSM._compute_speed(diff, 0.1, 0.05, 0.5)
#             # self.get_logger().info(f'{self.get_name()} turning towards goal heading {heading} current {self._cur_theta}
diff {diff} {twist.angular.z}')
#             self._publisher.publish(twist)
#             self._cur_twist = twist
#             return False

#         twist.linear.x = FSM._compute_speed(dist, 0.5, 0.2, 0.2)
#         self._publisher.publish(twist)
#         # self.get_logger().info(f'{self.get_name()} a distance {dist}  from target velocity {twist.linear.x}')
#         self._cur_twist = twist
#         return False

#     self.get_logger().info(f'at goal pose')
#     self._publisher.publish(twist)
#     return True

def _drive_to_goal(self, goal_x, goal_y):
    twist = Twist()

    # Calculate errors
    x_diff = goal_x - self._cur_x
    y_diff = goal_y - self._cur_y
    distance_error = math.sqrt(x_diff ** 2 + y_diff ** 2)

    # Heading calculation
```

```python
        desired_heading = math.atan2(y_diff, x_diff)
        heading_error = desired_heading - self._cur_theta
        heading_error = (heading_error + math.pi) % (2 * math.pi) - math.pi

        if distance_error > 0.1:  # Position tolerance
            # Correct heading
            if abs(heading_error) > 0.1:  # Orientation tolerance
                twist.angular.z = self._compute_speed(heading_error, self.max_angular_speed, self.angular_gain)
            else:
                twist.linear.x = self._compute_speed(distance_error, self.max_linear_speed, self.linear_gain)

        self._publisher.publish(twist)
        return distance_error <= 0.1


    def _do_state_at_start(self):
        self.get_logger().info(f'in start state')
        if self._run:
            self.get_logger().info(f'Starting...')
            self._cur_state = FSM_STATES.PERFORMING_TASK

    def _do_state_performing_task(self):
        if not self._run:
            return
        self.get_logger().info(f'heading to task {self._point}')
        # if self._drive_to_goal(self._points[self._point][0], self._points[self._point][1]):
        #     self._point = self._point + 1
        #     if self._point >= len(self._points):
        #         self._point = 0
        if self._drive_to_goal(*self._points[self._point]):
            self._point = (self._point + 1) % len(self._points)

    def _state_machine(self):
        if self._cur_state == FSM_STATES.AT_START:
            self._do_state_at_start()
        elif self._cur_state == FSM_STATES.PERFORMING_TASK:
            self._do_state_performing_task()
        else:
            self.get_logger().info(f'bad state {state_cur_state}')

    def _listener_callback(self, msg):
        pose = msg.pose.pose
        d2 = (pose.position.x - self._last_x) * (pose.position.x - self._last_x) + (pose.position.y - self._last_y) *
(pose.position.y - self._last_y)

        roll, pitch, yaw = euler_from_quaternion(pose.orientation)
        self._cur_x = pose.position.x
        self._cur_y = pose.position.y
        self._cur_theta = FSM._short_angle(yaw)
        self._state_machine()



def main(args=None):
    rclpy.init(args=args)
    node = FSM()
    try:
        rclpy.spin(node)
        rclpy.shutdown()
    except KeyboardInterrupt:
        pass

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

3. [40 marks] Currently all of the robots follow the leader. Instead, have the n+1'th robot follow the one prior to it. (This is a simple change to the controller launch file.) Modify the code in the follower controller code so that the robot does not run into the robot it is following. There are many ways of doing this, (The option of setting the follower robot's speed to zero is not considered a valid solution.) Describe your solution, provide the code, and include in the video submission an example of the robots executing the command

I implemented a safe distance mechanism for the follower robot to ensure it doesn't collide with the leader. My solution dynamically adjusts the follower's speed based on its distance to the leader, slowing down smoothly as it approaches a defined safe distance, such as 1.5 meters. If the robot gets too close, it reduces its speed accordingly without abruptly stopping. When the robot is beyond the safe distance, it resumes normal behavior, aligning with the leader's direction and moving toward its position.

Video is provided. Filename**: Video_3.mp4**

Code for **follow_chair.py**:

```
from enum import Enum
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rcl_interfaces.msg import SetParametersResult
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose, Point, Quaternion
from nav_msgs.msg import Odometry
from std_srvs.srv import SetBool




def euler_from_quaternion(quaternion):
    """
    Converts quaternion (w in last place) to euler roll, pitch, yaw
    quaternion = [x, y, z, w]
    """
    x = quaternion.x
    y = quaternion.y
    z = quaternion.z
    w = quaternion.w

    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)
    sinp = 2 * (w * y - z * x)
    pitch = np.arcsin(sinp)

    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)

    return roll, pitch, yaw


class FSM_STATES(Enum):
    STARTUP = 'Waiting',
    SLEEPING = 'Sleeping',
    FOLLOWING = 'Following'

class FollowChair(Node):
    """This will make one chair (chair_name) move to whever another chair (target_name)
       is right now. They will crash, of course, unless the chairs work to avoid this."""

    def __init__(self):
        super().__init__('FollowChair')
        self.get_logger().info(f'{self.get_name()} created')

        self.declare_parameter('chair_name', "chair_1")
        self._chair_name = self.get_parameter('chair_name').get_parameter_value().string_value
        self.declare_parameter('target_name', "chair_0")
        self._target_name = self.get_parameter('target_name').get_parameter_value().string_value
        self.get_logger().info(f'Chair {self._chair_name} is following {self._target_name}')

        self.create_subscription(Odometry, f"/{self._chair_name}/odom", self._self_callback, 1)
        self.create_subscription(Odometry, f"/{self._target_name}/odom", self._target_callback, 1)
        self._publisher = self.create_publisher(Twist, f"/{self._chair_name}/cmd_vel", 1)

        self.create_service(SetBool, f"/{self._chair_name}/startup", self._startup_callback)

        # the blackboard
        self._target_x = None
        self._target_y = None
        self._cur_state = FSM_STATES.STARTUP
        self._start_time = self.get_clock().now().nanoseconds * 1e-9
        self._run = False

    def _startup_callback(self, request, resp):
```

```
    self.get_logger().info(f'Got a request {request}')
    if request.data:
        resp.success = True
        resp.message = "Architecture running"
        self._cur_state = FSM_STATES.FOLLOWING
    else:
        if self._cur_state == FSM_STATES.STARTUP:
            self.get_logger().info(f'fsm suspended but not yet running?')
            resp.success = False
            resp.message = "In startup state"
        else:
            self._cur_state = FSM_STATES.SLEEPING
            self._publisher.publish(Twist())
            self.get_logger().info(f'fsm suspended')
            resp.success = True
            resp.message = "Architecture suspended"
    return resp


def _short_angle(angle):
    if angle > math.pi:
        angle = angle - 2 * math.pi
    if angle < -math.pi:
        angle = angle + 2 * math.pi
    assert abs(angle) <= math.pi
    return angle

def _compute_speed(diff, max_speed, min_speed, gain):
    speed = abs(diff) * gain
    speed = min(max_speed, max(min_speed, speed))
    return math.copysign(speed, diff)



def _drive_to_target(self, heading0_tol=0.15, range_tol=0.5, safe_distance=1.5):
    """
    Drive to the target while maintaining a safe distance.
    Return True iff we are at the goal, otherwise drive there.
    """
    twist = Twist()

    # Calculate the distance to the target
    x_diff = self._target_x - self._cur_x
    y_diff = self._target_y - self._cur_y
    dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)

    # Check if we are within the safe distance
    if dist < safe_distance:
        # Reduce speed proportionally to the distance
        twist.linear.x = FollowChair._compute_speed(dist - safe_distance, 0.5, 0.05, 0.5)
        self.get_logger().info(f'Too close! Slowing down. Distance: {dist:.2f}, Speed: {twist.linear.x:.2f}')
        self._publisher.publish(twist)
        return False

    # Check if we are within range tolerance to consider reaching the target
    if dist > range_tol:
        # Turn towards the target
        heading = math.atan2(y_diff, x_diff)
        diff = FollowChair._short_angle(heading - self._cur_theta)
        if abs(diff) > heading0_tol:
            twist.angular.z = FollowChair._compute_speed(diff, 0.5, 0.2, 0.2)
            self._publisher.publish(twist)
            self._cur_twist = twist
            return False

        # Move towards the target
        twist.linear.x = FollowChair._compute_speed(dist, 0.5, 0.05, 0.5)
        self._publisher.publish(twist)
        self._cur_twist = twist
        return False

    self.get_logger().info(f'At target location. Distance: {dist:.2f}')
    self._publisher.publish(twist)
    return True



def _do_state_at_start(self):
    # self.get_logger().info(f'waiting in start state')
    pass

def _do_state_following(self):
    # self.get_logger().info(f'following the target')
    if self._target_x is not None:
        self._drive_to_target()
        self._target_x = None
```

```python
            self._target_y = None

    def _state_machine(self):
        if self._cur_state == FSM_STATES.STARTUP:
            self._do_state_at_start()
        elif self._cur_state == FSM_STATES.FOLLOWING:
            self._do_state_following()
        elif self._cur_state == FSM_STATES.SLEEPING:
            pass
        else:
            self.get_logger().info(f'Bad state {state_cur_state}')

    def _target_callback(self, msg):
        """Update from target received"""
        pose = msg.pose.pose
        self._target_x = pose.position.x
        self._target_y = pose.position.y

    def _self_callback(self, msg):
        """We got a pose update"""
        pose = msg.pose.pose

        roll, pitch, yaw = euler_from_quaternion(pose.orientation)
        self._cur_x = pose.position.x
        self._cur_y = pose.position.y
        self._cur_theta = FollowChair._short_angle(yaw)
        self._state_machine()



def main(args=None):
    rclpy.init(args=args)
    node = FollowChair()
    try:
        rclpy.spin(node)
        rclpy.shutdown()
    except KeyboardInterrupt:
        pass

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

Code for **chair_controllers.py**

```
import os
import json
import sys
import math
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription, LogInfo
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
import xacro


def generate_launch_description():
    nchairs = 5
    for arg in sys.argv: # there must be a better way...
        if arg.startswith('nchairs:='):
           print(arg.split('chairs:=', 1)[1])
           nchairs = int(arg.split('chairs:=', 1)[1])
        elif ':=' in arg:
           print(f"Unknown argument in {arg}")
           sys.exit(0)
    print(f"Controlling {nchairs}")

    nodelist = []
    nodelist.append(
        Node(
            namespace = "chair_0",
            package='cpmr_ch11',
            executable='leader_chair',
            name='leader_chair',
            output='screen',
            parameters=[{'chair_name' : "chair_0"}])
        )
    print(f"leaderchair done")


    for chair in range(1, nchairs):
        name = f'chair_{chair}'
        #target_name = f"chair_0"
        target_name = f'chair_{chair - 1}'
        print(f"Processing {chair}")
        nodelist.append(
            Node(
                namespace = name,
                package='cpmr_ch11',
                executable='follow_chair',
                name='follow_chair',
                output='screen',
               parameters=[{'chair_name': name, 'target_name': target_name}]) # use chair_{chair-1}
        )

    return LaunchDescription(nodelist)
```

4. [10 marks] Make a new controller launch file that does not launch the leader controller. Instead control the leader robot from the keyboard (you can remap the normal output topics of the teleop_keyboard node to do this. Use your solutions to (2) and (3) above to generate a convoy of the robots following the leader. Provide a video of your system running. Demonstrate a successful run in which the robots stay in the convoy. Also show a run in which the robots crash into each other and (essentially) get stuck in the traffic jam generated.

I made another launch file that does not launch the leader controller. I then use the teleop_twist_keyboard to publish keyboard messages to that node.

Commaned:
ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -r /cmd_vel:=/chair_0/cmd_vel

Video for both the cases is showed in one video as max file limit is 3 on eClass.
Filename**: Video_4.1.mp4**

Code **chairs_keyboard.launch.py**:

```
import os
import json
import sys
import math
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, IncludeLaunchDescription, LogInfo
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node
import xacro


def generate_launch_description():
    nchairs = 5
    for arg in sys.argv: # there must be a better way...
        if arg.startswith('nchairs:='):
            print(arg.split('chairs:=', 1)[1])
            nchairs = int(arg.split('chairs:=', 1)[1])
        elif ':=' in arg:
            print(f"Unknown argument in {arg}")
            sys.exit(0)
    print(f"Controlling {nchairs}")

    nodelist = []
```

```
for chair in range(1, nchairs):
    name = f'chair_{chair}'
    #target_name = f"chair_0"
    target_name = f'chair_{chair - 1}'
    print(f"Processing {chair}")
    nodelist.append(
        Node(
            namespace = name,
            package='cpmr_ch11',
            executable='follow_chair',
            name='follow_chair',
            output='screen',
           parameters=[{'chair_name': name, 'target_name': target_name}]) # use chair_{chair-1}
    )

return LaunchDescription(nodelist)
```