# 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

* 1. 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

...

A diagram of a diagram

Description automatically generated

* 1. A copy of the rqt\_graph

1. [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

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()

1. [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

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)

1. [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)