MISSION COORDINATION TP1&2

Report

M2 SAAS

Group members:

Farid Azakov 20235679 Cagdas Anil 20235675 Atakan Eliacik 20235691

Project Objective

The objective of this project is to move three simulated robots using ROS 1 and Gazebo from a starting position to the adequate flag (e.g. robot1 going to flag1, robot2 going to flag2 and robot3 going to flag3) while avoiding any collision with another robot. The main goal is to reach the adequate flag as fast as possible.

The simulated robots are defined as follow:

- An ultrasonic sensor is embedded, with a limited range (5 meters)

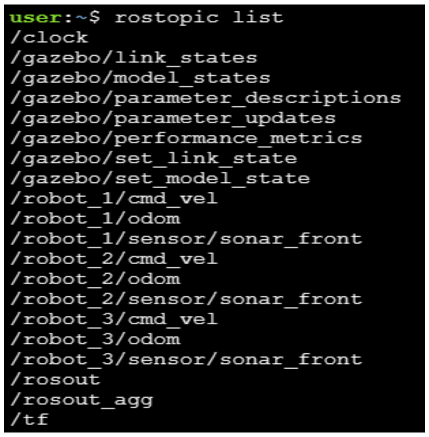
- 2 motorized wheels allow the robot to operate in the environment

- Its current pose in the environment is known (Position and orientation)

Steps and Questions:

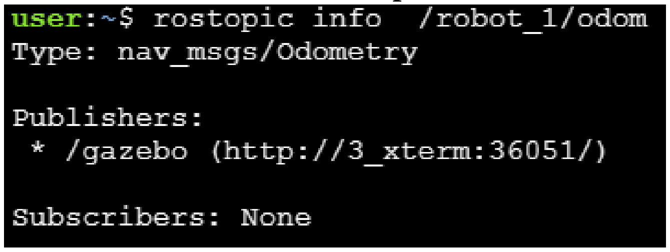
Step 2: Visualization of some concepts

1. **List the topics**
   Q: What is list command used for and what is the result?
   A: It gives the list of all topics for actual nodes running in ROS
   The result is:



```
user:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/robot_1/cmd_vel
/robot_1/odom
/robot_1/sensor/sonar_front
/robot_2/cmd_vel
/robot_2/odom
/robot_2/sensor/sonar_front
/robot_3/cmd_vel
/robot_3/odom
/robot_3/sensor/sonar_front
/rosout
/rosout_agg
/tf
```

2. **Information about Sonar Topic**



```
user:~$ rostopic info  /robot_1/odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://3_xterm:36051/)

Subscribers: None
```

   Q: According to you who are the publisher and subscriber?
   A: Publisher is gazebo, there is no subscriber
   Q: According to you what types of messages are published on this topic?
   A: Type of the message is: nav_msgs/Odometry

3. **Listening to a topic:**
   Q: What is echo command used for and what is the result?
   A: It is used for returning the messages which we are listening to: pose (position, orientation), velocity (linear, angular) and covariance for pose and twist.

Step 3: Moving one robot

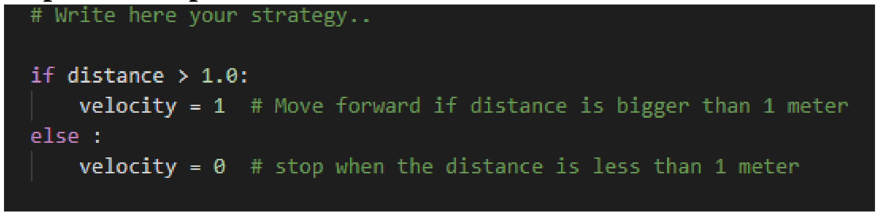
In this step we modify the python script agent.py to see one of the robots is moving.

(We did not include the script, as it is easy)

Step 4: Move one robot to the corresponding flag

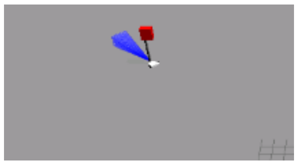
Now as we know how to start and stop the robot in the environment we will answer following questions:

1. **Q: to modify your program to move one robot safely to its corresponding flag and stop it at this position**

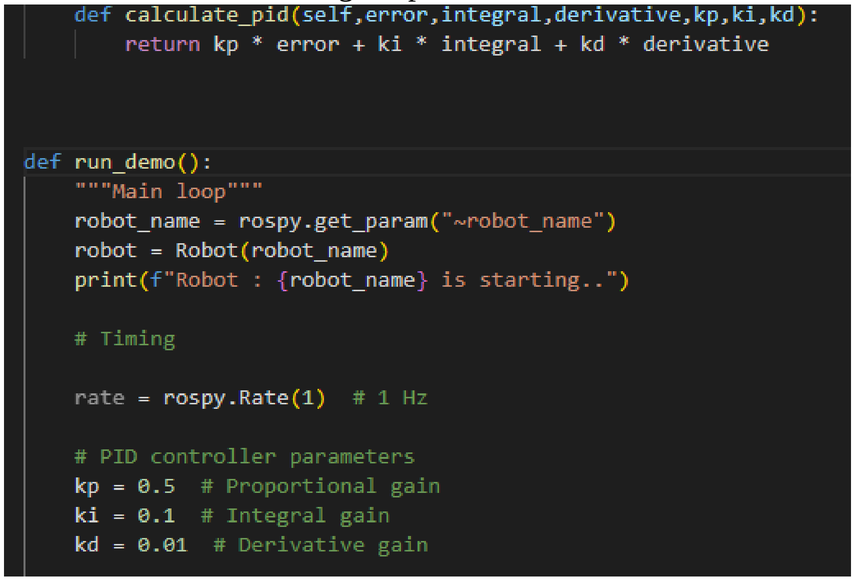


```
# Write here your strategy..

if distance > 1.0:
    velocity = 1  # Move forward if distance is bigger than 1 meter
else :
    velocity = 0  # stop when the distance is less than 1 meter
```

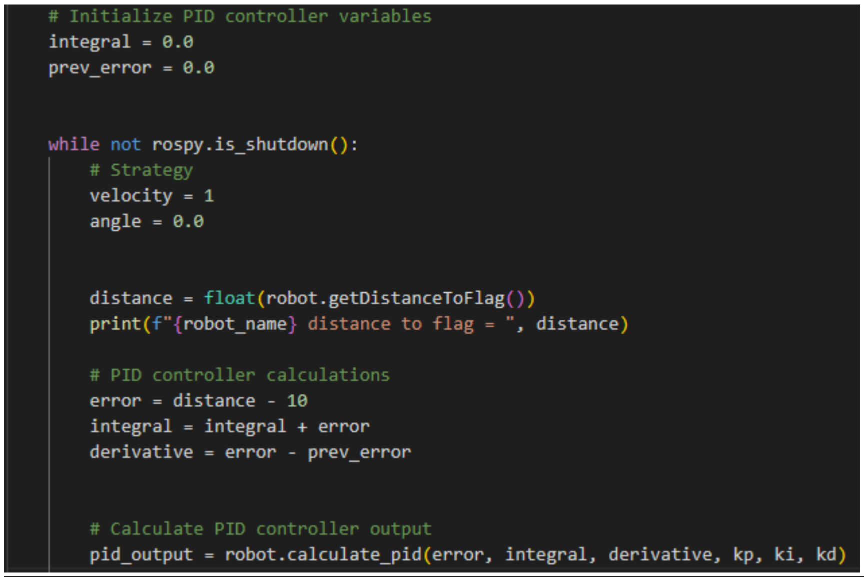
   We see that the robot has stopped:



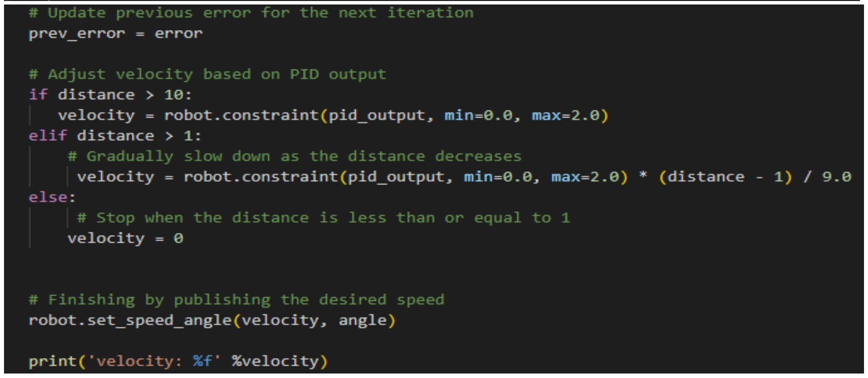
2. **Q: to adapt it to real life. For this purpose, you can implement a PID controller. It means that, when the robot is far from its goal, it moves with the highest velocity values and as it gets closer, it slows down to a stop.**
   We can write the following script:



```python
    def calculate_pid(self,error,integral,derivative,kp,ki,kd):
        return kp * error + ki * integral + kd * derivative



def run_demo():
    """Main loop"""
    robot_name = rospy.get_param("~robot_name")
    robot = Robot(robot_name)
    print(f"Robot : {robot_name} is starting..")

    # Timing

    rate = rospy.Rate(1)  # 1 Hz

    # PID controller parameters
    kp = 0.5  # Proportional gain
    ki = 0.1  # Integral gain
    kd = 0.01  # Derivative gain
```

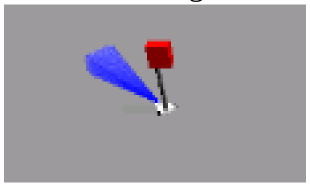


```python
    # Initialize PID controller variables
    integral = 0.0
    prev_error = 0.0


    while not rospy.is_shutdown():
        # Strategy
        velocity = 1
        angle = 0.0


        distance = float(robot.getDistanceToFlag())
        print(f"{robot_name} distance to flag = ", distance)

        # PID controller calculations
        error = distance - 10
        integral = integral + error
        derivative = error - prev_error


        # Calculate PID controller output
        pid_output = robot.calculate_pid(error, integral, derivative, kp, ki, kd)
```



```python
# Update previous error for the next iteration
prev_error = error

# Adjust velocity based on PID output
if distance > 10:
   velocity = robot.constraint(pid_output, min=0.0, max=2.0)
elif distance > 1:
    # Gradually slow down as the distance decreases
     velocity = robot.constraint(pid_output, min=0.0, max=2.0) * (distance - 1) / 9.0
else:
     # Stop when the distance is less than or equal to 1
    velocity = 0


# Finishing by publishing the desired speed
robot.set_speed_angle(velocity, angle)

print('velocity: %f' %velocity)
```

We will see that when the robot gets closer it starts to slow down and stops when it reaches the flag:

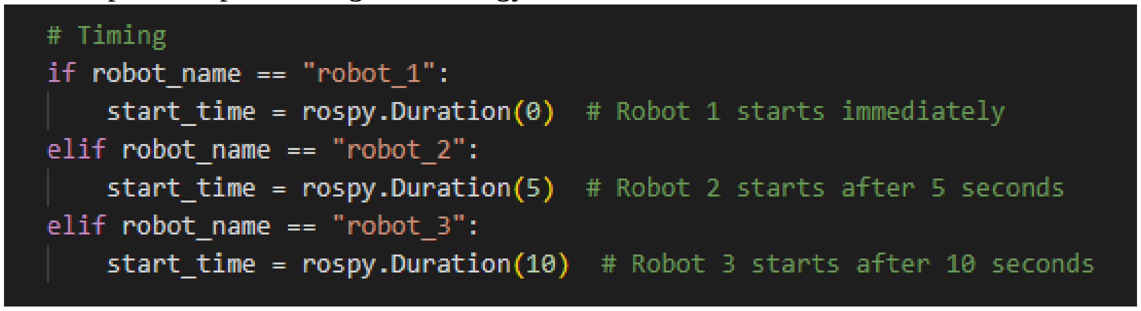


(You can see it in the video that we will upload separately.)

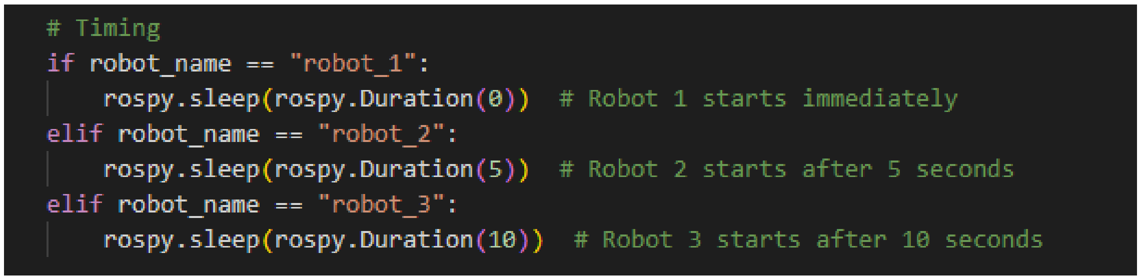
Step 5: Implementation of one strategy – timing solution

For this step we will write an approach that allows all three robots to reach their corresponding flags without colliding with each other.

1. **Q: We will implement one of the simplest strategy: timing strategy. The timing strategy consists on starting each robot at different time. In this way, we will avoid collision.**
   The script for implementing this strategy:

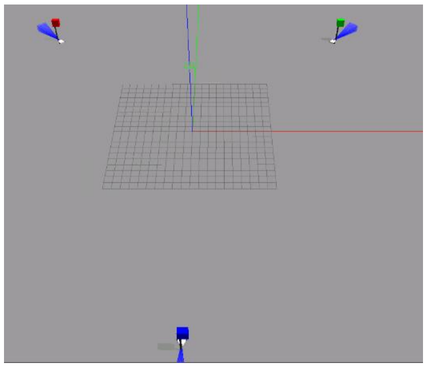


```
# Timing
if robot_name == "robot_1":
    start_time = rospy.Duration(0)  # Robot 1 starts immediately
elif robot_name == "robot_2":
    start_time = rospy.Duration(5)  # Robot 2 starts after 5 seconds
elif robot_name == "robot_3":
    start_time = rospy.Duration(10)  # Robot 3 starts after 10 seconds
```



```
# Timing
if robot_name == "robot_1":
    rospy.sleep(rospy.Duration(0))  # Robot 1 starts immediately
elif robot_name == "robot_2":
    rospy.sleep(rospy.Duration(5))  # Robot 2 starts after 5 seconds
elif robot_name == "robot_3":
    rospy.sleep(rospy.Duration(10))  # Robot 3 starts after 10 seconds
```

When we simulate we see that all robots reach their destination without colliding.



(It will also be available as video)

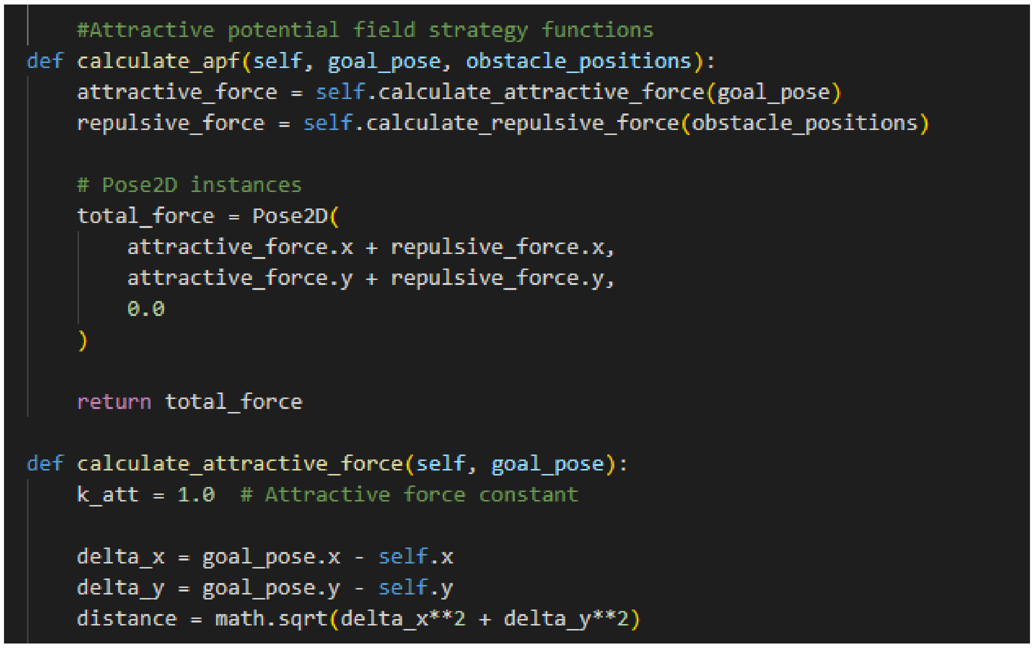
This strategy is efficient, but not robust, because it can be used only in this case and it is not that fast time.

Part 2 of the lab work

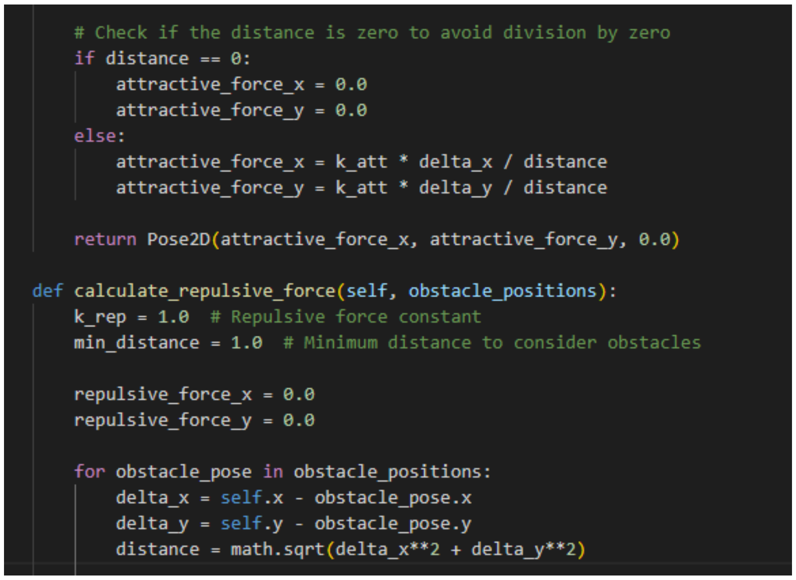
Now we will try to implement robust strategy.

We can use Artificial Potential Field strategy for path planning and obstacle avoidance.

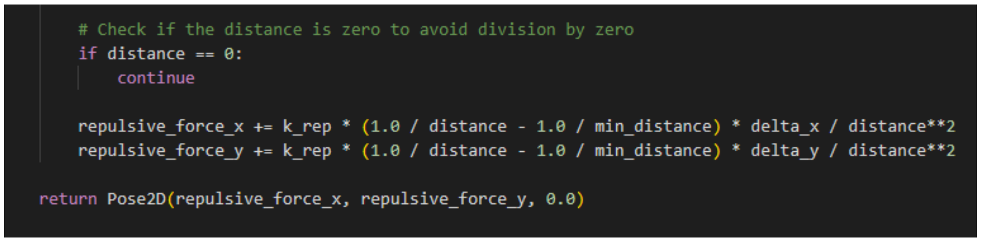
We can define following functions:



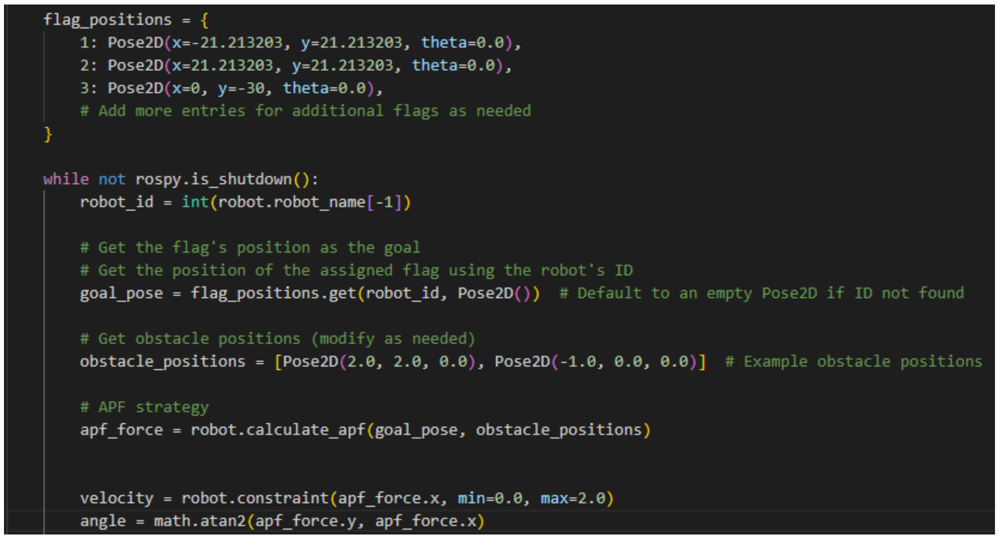
```
    #Attractive potential field strategy functions
def calculate_apf(self, goal_pose, obstacle_positions):
    attractive_force = self.calculate_attractive_force(goal_pose)
    repulsive_force = self.calculate_repulsive_force(obstacle_positions)

    # Pose2D instances
    total_force = Pose2D(
        attractive_force.x + repulsive_force.x,
        attractive_force.y + repulsive_force.y,
        0.0
    )

    return total_force

def calculate_attractive_force(self, goal_pose):
    k_att = 1.0  # Attractive force constant

    delta_x = goal_pose.x - self.x
    delta_y = goal_pose.y - self.y
    distance = math.sqrt(delta_x**2 + delta_y**2)
```



```
    # Check if the distance is zero to avoid division by zero
    if distance == 0:
        attractive_force_x = 0.0
        attractive_force_y = 0.0
    else:
        attractive_force_x = k_att * delta_x / distance
        attractive_force_y = k_att * delta_y / distance

    return Pose2D(attractive_force_x, attractive_force_y, 0.0)

def calculate_repulsive_force(self, obstacle_positions):
    k_rep = 1.0  # Repulsive force constant
    min_distance = 1.0  # Minimum distance to consider obstacles

    repulsive_force_x = 0.0
    repulsive_force_y = 0.0

    for obstacle_pose in obstacle_positions:
        delta_x = self.x - obstacle_pose.x
        delta_y = self.y - obstacle_pose.y
        distance = math.sqrt(delta_x**2 + delta_y**2)
```



```
        # Check if the distance is zero to avoid division by zero
        if distance == 0:
            continue

        repulsive_force_x += k_rep * (1.0 / distance - 1.0 / min_distance) * delta_x / distance**2
        repulsive_force_y += k_rep * (1.0 / distance - 1.0 / min_distance) * delta_y / distance**2

    return Pose2D(repulsive_force_x, repulsive_force_y, 0.0)
```



```
    flag_positions = {
        1: Pose2D(x=-21.213203, y=21.213203, theta=0.0),
        2: Pose2D(x=21.213203, y=21.213203, theta=0.0),
        3: Pose2D(x=0, y=-30, theta=0.0),
        # Add more entries for additional flags as needed
    }

    while not rospy.is_shutdown():
        robot_id = int(robot.robot_name[-1])

        # Get the flag's position as the goal
        # Get the position of the assigned flag using the robot's ID
        goal_pose = flag_positions.get(robot_id, Pose2D())  # Default to an empty Pose2D if ID not found

        # Get obstacle positions (modify as needed)
        obstacle_positions = [Pose2D(2.0, 2.0, 0.0), Pose2D(-1.0, 0.0, 0.0)]  # Example obstacle positions

        # APF strategy
        apf_force = robot.calculate_apf(goal_pose, obstacle_positions)


        velocity = robot.constraint(apf_force.x, min=0.0, max=2.0)
        angle = math.atan2(apf_force.y, apf_force.x)
```

However, when we run the code robots create circles around themselves, we could not solve this problem.