AARHUS UNIVERSITET

Department of Electrical and Computer Engineering

Assignment 1

Implementing robot circling with RL and measuring individual vs. group performance

Group 4

Alba Arcos I Ribas
Student no. 202503224

Birnir Þór Árnason
Student no. 202503220

Martin Skelde Krøjmand
Student no. 201710685

Wouter Jonathan Oudijk
Student no. 202109059

Zsombor Krisztián Szöke
Student no. 202411164

Teacher: *Mirgita Frasheri and Lukas Esterle*

Number of characters: *7070 (3 standard pages)*

Aarhus, October 7, 2025

Contents

List of Figures

List of Tables

Listings

1 Implementation

1.1 Hard-coded solution

To get familiar with the IR-Sim library, we first hard-coded a robot to move in a circle. To do this we first found the source code for the *Dash* behaviour when using *Diff* kinematics[1]. This basic behaviour simply makes the agent move directly towards the goal, slowing down to turn if necessary. Since the agent wants to move directly towards the goal, the desired angle of the agent should be equal to the angle between the goal and the agent. So the DiffDash behaviour starts by getting the angle of the agent and the desired angle (the angle between the agent and the goal). The difference between these angles is calculated and then the agent simply turns to reduce this difference to zero, making it's angle equal to the desired angle.

To make the agent instead move along a circle, we simply add 90 degrees to the desired angle. This makes the agent move along the circle tangent and assuming infinetly small steps, this would result in a perfect circle. However since we don't have infinetly small steps, the agent will move further and further away from the circle with each step. To account for this, we simply take the distance from the circle center and add this to the equation, increasing the angle when the distance is less than the desired radius and decreasing the angle when the distance is more than the desired radius. This results in the code seen in listing 1.1.

```python
# Get the distance and the angle to the goal (Circle center):
distance, radian = relative_position(state, goal)

# Calculate the distance correction amount:
distanceCorrection = (circle_radius - distance) * correction_multiplier

# Add 90 degress + the distance correction to the desired angle
radian += np.pi/2 + distanceCorrection

# Calculate the difference between the agents angle and the desired angle
diff_radian = WrapToPi(radian - state[2, 0])
```

Listing 1.1: Implementation of getting the desired angle for the circle following agent

1.2 Reinforcement Learning

Having gotten a deeper understanding of the IR-Sim library, we started implementing the circle following behaviour using reinforcement learning. In our group we implemented RL in 3 slightly different ways, which allows us to later compare the different solutions, including the hard-coded solution, and determine which is the best for creating circle following agents.

1.2.1 Alba

1.2.2 Q learning with outside table and error shaped reward

This solution uses predefined actions and states to fill out an outside Q-table by learning throughout multiple episodes. The predefined states are determined by using data binning, meaning we sort the current state into categories. These categories are determined by computing a relative distance and angle from the robot's pose (x, y, θ). From these we get a radial and an angle error value, from which we can discretize a 3×3 state matrix using the bins:

- Radius bins: $(-\infty, -0.2]$, $[-0.2, 0.2]$, $[0.2, \infty)$ (inside / on / outside).
- Angle bins: $(-\pi, -\pi/4]$, $[-\pi/4, \pi/4]$, $[\pi/4, \pi)$ (left / aligned / right).

The robot's actions are defined by 4 simple actions. These were originally meant to be used for test purposes, but the final results were accurate enough that we didn't need to implement more actions. The actions work by changing the linear and angular velocities of the robot (v, ω). The actions are built from the robot's limits ($v_{\max}, \omega_{\max}$): fast means 80% of the max linear speed, slow means 40%, and turning actions use 80% of the max angular speed:

- fast forward + turn left
- fast forward straight
- fast forward + turn right
- slow forward straight

The learning is a Q-learning algorithm, where the learning rate is 0.5, the discount is 0.9, and the ε-greedy rate is 0.2. After each transition the update is:

$$Q^{\Pi}(s,a) \leftarrow Q^{\Pi}(s,a) + \alpha\big(R + \gamma max_a Q^{\Pi}(s',a') - Q^{\Pi}(s,a)\big).$$

The reward is shaped by penalising radial and heading errors, while rewarding progress along the orbit. Weights are added for normalisation, as you can see in 1.2:

```python
def shaped_reward(e_r: float, e_psi: float, v_cmd: float) -> float:
    alpha = 2.0    # weight radial error
    beta  = 1.0    # weight heading error
    tan   = 0.4    # weight tangential
    return float(-alpha * abs(e_r) - beta * abs(e_psi) + tan * (v_cmd * math.cos(e_psi)))
```

Listing 1.2: Shaped reward function

We used a CSV for the Q table so the learning progress can be saved throughout iterations, and so the model can be trained throughout multiple epochs.

1.2.3 Wouter Jonathan

For this version we defined an environment with states based on the distance to the circle center and actions

1.3 Subsumption Architecture and Obstacle Avoidance

1.3.1 Alba

1.3.2 Birnir

To improve the robot's behavior, an additional layer was introduced to handle collision avoidance. This enhancement expanded the original state space by implementing a binary indicator for detecting threats, wheter a robot is within the safety distance, which indicates a threat, or not, thereby doubling the number of states. The action space was also extended with a new "stop" action, which allows the robot to remain stationary when it detects a collision threat. A new reward function was designed to penalize proximity to other robots, unnecessary stopping, and crashes, while still encouraging progress towards the goal of circular movement. By applying a layered approach, it becomes possible to retain the original navigation behavior while adding a simple safety mechanism that activates only when necessary.
In safe conditions, the robot behaves similarly to the original behavior, navigating around the target in a circular movement. However, when another robot comes within the safety distance, the robot modifies its policy by stopping. This could be expanded by introducing some sort of turning-away behavior to avoid the collisions completely. The behavior develops naturally from the Q-learning updates and the enhanced rewards. When structuring such a system in layers, the robot benefits from a modular implementation where the collision avoidance layer acts as an override mechanism, enhancing robustness in multi-agent scenarios without disrupting the original core objectives.

2 Testing methodology

To measure the quality of our solutions, we measure based on the following metrics:

1. **Radius Error**: Every simulated step, we add the current distance between the agent and the desired circle radius, adding together the error of all the agents to get one value per solution. A lower value would equal the agent having spent more time closer to the circle it is meant to follow.
2. **Angle Error**: Every simulated step, we add the amount the desired angle for the agent has changed, adding together the error of all the agents to get one value per solution. While following a circle, the angle should stay almost constant, so a lower value would equal the agent having followed a smooth path.
3. **Speed**: The amount of time the agents on average spend doing one lap around the circle. A higher speed is better.

And to measure the quality of the solutions using the subsumption architecture, we also add the following metrics:

4. **Collisions**: The amount of agents that collide.

3 Results

First we will go through every metric, showing a graph of how the different solutions compare on the specific metric throughout the simulation time. In section 4 *Discussion* we will collect the results in table 4.1 and discuss our findings.

3.1 Radius Error

3.2 Angle Error

3.3 Speed

3.4 Collisions

4 Discussion

Solutions	Radius Error	Angle Error	Speed	Collisions
Hard-coded				
Alba				
Krisztián				
Wouter Jonathan				
Subsumption Alba				
Subsumption Birnir				

Table 4.1: Overview of all the test results

5 Conclusion

Bibliography

Websites

[1] Hanruihua. *IR-SIM DiffDash documentation*. Last visited: 02-10-2024. URL: https://ir-sim.readthedocs.io/en/stable/_modules/irsim/lib/behavior/behavior_methods.html#DiffDash.