山东大学

崇新学堂

2025－2026 学年第一学期

实 验 报 告

课程名称：电子信息工程导论

实验名称：Controlling Robots

专 业 班 级　崇新学堂

学 生 姓 名　高子轩，钱竹玉，吕思洁，徐亚骐

实 验 时 间　2025 年 10 月 12 日

Basis (Driven: Make a robot brain that can follow a path that is composed of a sequence of straight line segments)

Step1: Build Easy system

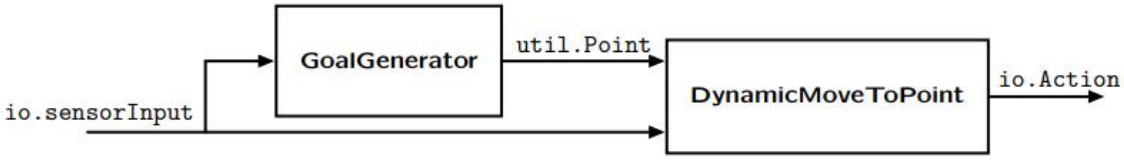

Figure 1 Preliminary signal system diagram

Solving logic Aiming to complete the task of building a simple system (As shown in Figure 1), this project will utilize sm.Cascade, sm.Parallel, and similar modules.

• First, we observe that the final state machine, DynamicMoveToPoint, accepts two inputs. This indicates that the preceding system is a parallel structure.

• Upon examining this parallel structure, we find that the first branch is processed by a GoalGenerator, while the second branch is a direct pass-through. This entire configuration is implemented as a parallel state machine.

• Finally, as per the requirements, we use sm.constant to represent the GoalGenerator and sm.wire() for the direct pass-through path. Then cascade them as a whole.This completes the implementation of the state machine.

Here is our answer

```
1. mySM = sm.Cascade(sm.Parallel(sm.Constant(util.Point(1.0, 0.5)), sm.Wire()),

dynamicMoveToPointSkeleton.DynamicMoveToPoint())
```

Odometry

Each drive wheel of the robot is equipped with an encoder for calculating the number of rotations of the wheel. We can use them to estimate the robot's pose under ideal conditions.

Utilities

We can use the util module to determine the position of the robot in two-dimensional space.

Driving to a point

Step 2. Determine the action that should be output by the DynamicMoveToPoint state machine for the following input conditions.

Strategy for move to point:

• If the vehicle's heading is not aligned with the goal point, it will first perform an in-place rotation until its orientation is approximately correct.

• After that, it will then proceed straight, stopping once it is sufficiently close to the target.

Current robot pose	goalPoint	Action
(0.0, 0.0, 0.0)	(1.0, 0.5)	Rotate left
(0.0, 0.0, π/2)	(1.0, 0.5)	Rotate right
(0.0, 0.0, $\tan^{-1}$ 0.5)	(1.0, 0.5)	Move forward
(1.0001, 0.4999, 0.0)	(1.0, 0.5)	Stop

Table 1 Answer for step2

Step 3. The implementation of the strategy

Code logic

- First, we define the goal conditions using angular and distance errors. The robot is considered to have arrived when both the heading and distance errors fall below their predefined thresholds.
- During the adjustment process, we use a proportional controller for velocity. The robot's speed is calculated by multiplying the error by a proportional gain. This causes the robot's rotational and translational movements to slow down as it gets closer to the target's orientation and position, ensuring a fine-tuned final alignment.
- A critical final point: when calculating the angular error, it is essential to use the fixAnglePlusMinusPi function to normalize the angle . Otherwise, out-of-bounds angle values can lead to incorrect behavior.

Here is the key code

```
class DynamicMoveToPoint(sm.SM):
    def getNextValues(self, state, inp):

        #  Unpack the goal point and sensor data from the input
        goalPoint, sensors = inp
        robotPose = sensors.odometry

        # Initialize the output actions
        fvel = 0.0
        rvel = 0.0

```

```
12.         # Calculate the distance error between the current position and the goal point
13.         dist_error = robotPose.point().distance(goalPoint)
14.
15.         # Check if the goal point has been reached
16.         if dist_error < 0.005:
17.             # If close enough, stop
18.             fvel = 0.0
19.             rvel = 0.0
20.         else:
21.             #  Calculate the angle error between the current orientation and the direction to the
goal
22.             angle_to_goal = robotPose.point().angleTo(goalPoint)
23.             angle_error = util.fixAnglePlusMinusPi(angle_to_goal - robotPose.theta)
24.
25.             #  Decision logic: prioritize rotation, then move forward
26.             if abs(angle_error) > 0.02:
27.                 # If the angle deviation is too large, rotate in place to align with the target
28.                 # The turning speed is proportional to the angle error
29.                 fvel = 0.0
30.                 rvel = 0.9 * angle_error
31.             else:
32.                 # If already aligned, move forward
33.                 # The forward speed is proportional to the distance, to automatically decelerate
when approaching the target
34.                 fvel = 0.7 * dist_error
35.                 rvel = 0.0  # Maintain straight-line movement
```

Results under idle and soar

```
>>> ================================ RESTART ================================
>>>
[SMCheck] the start state of your state machine is 'None'
The actual inputs are (target, sensorInput) pairs;  here we
are just showing the odometry part of the sensorInput.

Input: (point:(1.000000, 0.500000), pose:(0, 0, 0))
Output: Act: (fvel = 0.000000, rvel = 0.324553)

Input: (point:(1.000000, 0.500000), pose:(0, 0, 1.570796))
Output: Act: (fvel = 0.000000, rvel = -0.775004)

Input: (point:(1.000000, 0.500000), pose:(0, 0, 0.463648))
Output: Act: (fvel = 1.006231, rvel = 0.000000)

Input: (point:(1.000000, 0.500000), pose:(1.000100, 0.499999, 0))
Output: Act: (fvel = 0.000000, rvel = 0.000000)
```

Figure 2 Results of operation under idle

In soar,when we set verbose to True, the following information will appear on the console.

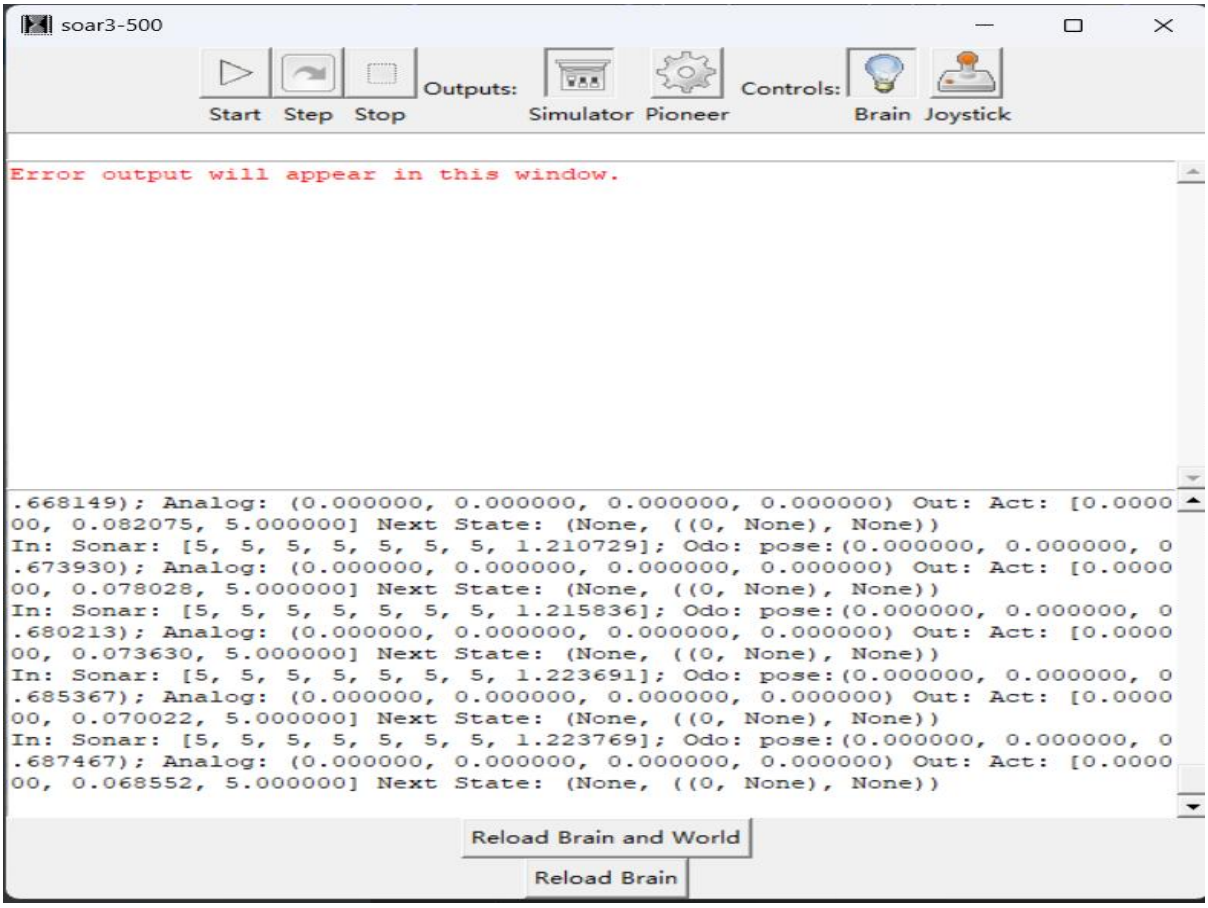

Figure 3 Printed message of operation under soar

Hip to be Square Make the robot move in a square. by defining a new state machine class FollowFigure, which serves the role of being a GoalGenerator

Solving logic The first target point is (0.5, 0.5). Since neither of the first two currentPose values are sufficiently close to this destination, the system remains in State 0, which is dedicated to navigating to the first point. The same logic applies to the next phase; the system never reaches the second target, and as a result, it stays in State 1.

Current robot pose	State	Target point
(0.0, 0.0, 0.0)	0	(0.5, 0.5)
(0.0, 1.0, 0.0)	0	(0.5, 0.5)
(0.499, 0.501, 2.0)	1	(0.0, 1.0)
(2.0, 3.0, 4.0)	1	(0.0, 1.0)

Table 2 Answer for question below the step4

Step 4. Test ffSkeleton.py in idle by running testFF.py.

Code logic

- The FollowFigure class acts as a sequential waypoint manager. Its primary role is to process a list of target points and output the active target for the robot to pursue.
- The machine's state represents the index of the current target waypoint. It will only transition to the next state (i.e., the next waypoint's index) when the

robot's current position is within a predefined proximity of the current target.

• A key feature is its handling of the final waypoint. Once the robot reaches the last point in the list, the state machine ceases to advance. It will continuously output the final waypoint as the target, effectively instructing the robot to hold its position there.

Here is the key code

```
class FollowFigure(sm.SM):

    def __init__(self, waypoints):

        self.waypoints = waypoints
        # The initial state is the index of the first waypoint: 0
        self.startState = 0

    def getNextValues(self, state, inp):

        currentTarget = self.waypoints[state]
        robotPoint = inp.odometry.point()

        is_near_target = robotPoint.isNear(currentTarget, WAYPOINT_EPSILON)

        if is_near_target:
            # If the target is reached, advance to the next waypoint.
            # Use min() to ensure the index does not go out of bounds.
            next_state = min(state + 1, len(self.waypoints) - 1)
        else:
            # If not yet at the target, the state remains the same.
```

```
22.             next_state = state
23.
24.         # The output is always the current target being pursued.
25.         output = currentTarget
26.          return (next_state, output)
```

Results under idle

```
>>> ================================ RESTART ================================
>>>
[SMCheck] the start state of your state machine is '0'
The actual inputs are whole instances of io.SensorInput;  here we
are just showing the odometry part of the input.

Input: pose:(0, 0, 0)
Output: point:(0.500000, 0.500000)

Input: pose:(0, 1, 0)
Output: point:(0.500000, 0.500000)

Input: pose:(0.499000, 0.501000, 2)
Output: point:(0.500000, 0.500000)

Input: pose:(2, 3, 4)
Output: point:(0.000000, 1.000000)
>>>
```

Figure 4 Results of operation under idle

Step 5 Move Brain Skeleton

Checkoff Wk.3.2.2: Show the slime trail resulting from the simulated robot moving in a square or other interesting figure

Through the previous learning, we have refined the goalGenerator state machine and can truly and perfectly combine the state machine mySM mentioned in step1.

Here is the key code

```
1. goalGenerator = ffSkeleton.FollowFigure(squarePoints)

2. figureFollower = sm.Cascade(sm.Parallel(goalGenerator,

sm.Wire()),dynamicMoveToPointSkeleton.DynamicMoveToPoint())

3. myST = figureFollower
```

Square trajectory We completed the following slime diagram by passing in the dot columns required for the square through the FollowFigure class.

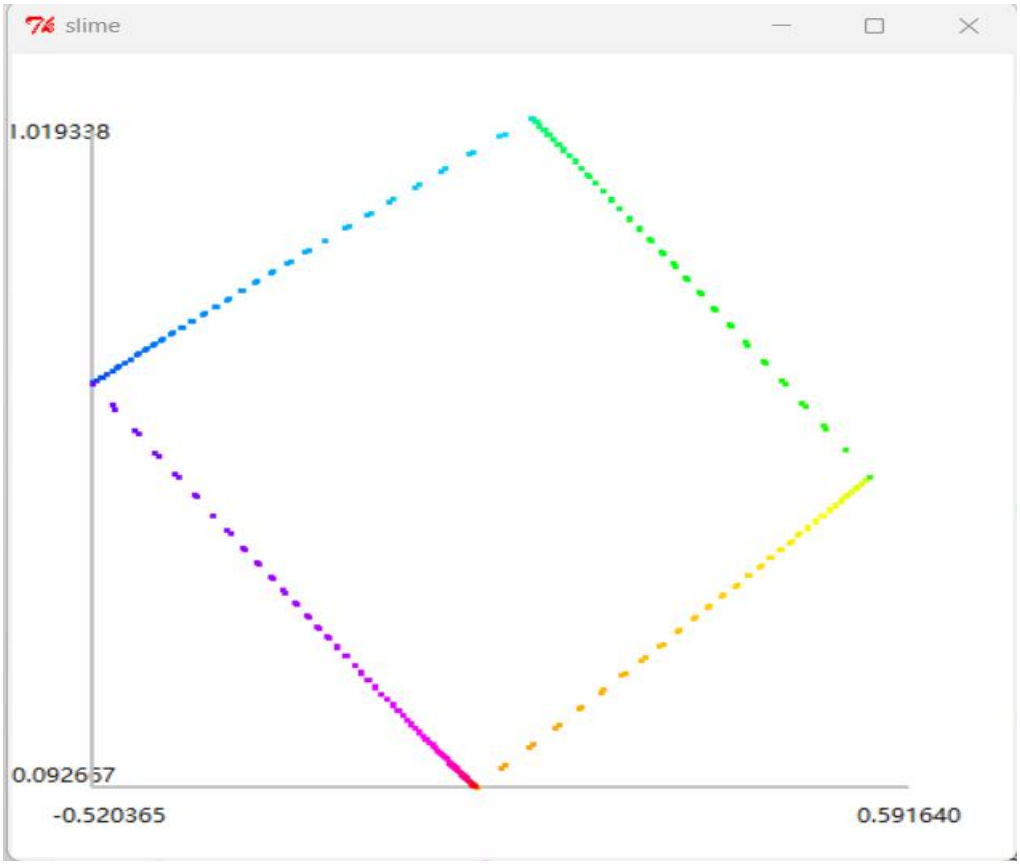

Figure 5 Square slime map

Customized trajectory plot We accomplished the trajectory tracking of the mountain element from the Shandong University logo by constructing an appropriate point sequence in moveBrainSkeleton and passing it to FollowFigure, resulting in the following slime trail.

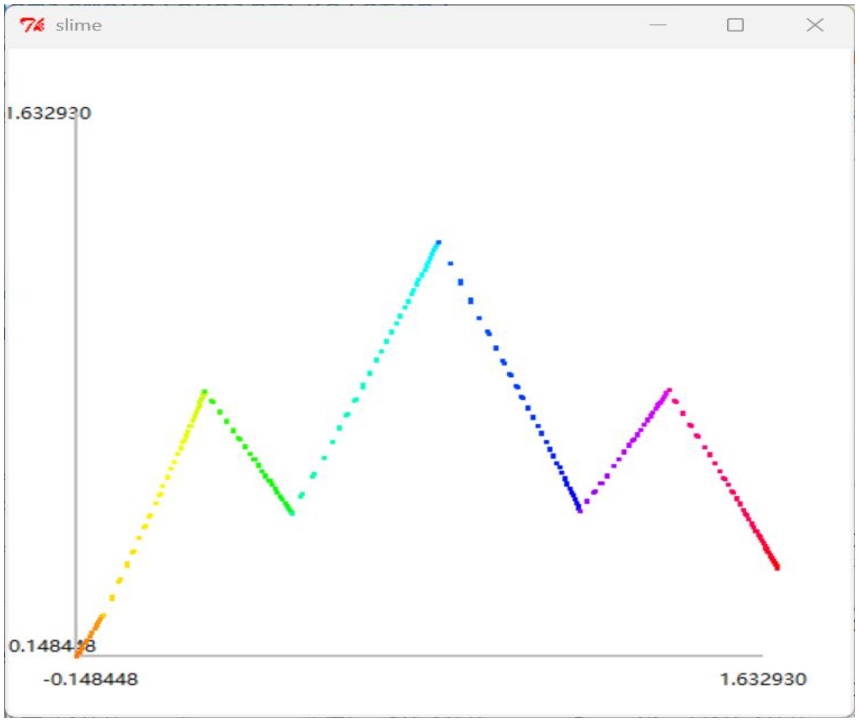

Figure 6 The slime map of mountain from Shandong University Logo

Here is the key code

```
1. # Adjusting to an Appropriate Point Sequence
2. SDU_Dotlist = [util.Point(0.1, 0.2),util.Point(0.3,0.8),
util.Point(0.5,0.3),util.Point(0.8,1.2),util.Point(1.1,0.3), util.Point(1.3, 0.8),util.Point(1.5,
0.2)]
3. goalGenerator = ffSkeleton.FollowFigure(SDU_Dotlist)
4. figureFollower = sm.Cascade(sm.Parallel(goalGenerator,
sm.Wire()),dynamicMoveToPointSkeleton.DynamicMoveToPoint())
5. myST = figureFollower
```

Step 6 Maintain the sonar obstacle avoidance function while moving forward

Code logic We defined a Boolean type variable, isObstacle, and used sm.constant(io.Action()) to constantly output an empty state to achieve the effect of stopping. Use sm.switch() as the final state machine, set the first element to isObstacle. If it is True, run the stop state machine; otherwise, run the original state tracking state machine.

Here is the key code

```
1. isObstacle = lambda inp: min(inp.sonars[3], inp.sonars[4]) < 0.3

2. stopMachine = sm.Constant(io.Action())

3. mySM = sm.Switch(isObstacle, stopMachine, figureFollower)
```

experimental result

By taking the smaller value of the front sonar reading as the basis for judgment, when the car is about to hit the wall, it will stop running first to achieve the purpose of obstacle avoidance. We successfully achieved that the car will only track the target point when there are no obstacles in front of

Figure 7 The car is going along the square road

Figure 8 The car stopped next to a sudden obstacle

Summary: We have successfully designed and implemented a robot control system with path tracking and obstacle avoidance functions. The core of the system adopts a state machine architecture. By integrating the GoalGenerator target point generator and the DynamicMoveToPoint dynamic movement module, the robot can move along the preset straight path. The experiment further achieved the movement of the robot along geometric trajectories such as squares. The path point sequence was orderly managed by the FollowFigure state machine, and the mucus trajectory in the simulation verified the tracking accuracy. In addition, the system integrates an obstacle avoidance function based on sonar. When an obstacle is detected ahead, it automatically triggers the stop mechanism, ensuring operational safety.

General Thanks:

For the help of AI, we completed some difficult translation tasks and Make clear some function's usage by the powerful search capabilities of AI(such as sm.Wire(),sm.Switch()) .

Thanks to MIT and SDU for providing detailed guidance notes to complete this experiment.