ROBT502 – ROBOT PERCEPTION AND VISION

Laboratory Report 8

Z. Kuangaliyev, A. Amangeldi, S. Omirbayev and K. Kantoreyeva

18 April 2024

1. Introduction

This report presents the processes and results of controlling our turtlebot in the frame of cameras we set up in Gazebo simulation. For the purpose of this lab we have set up 9 cameras that capture and stream the movement of the simulated turtlebot.

2. Setting up the cameras in Gazebo

To get the best coverage of the labyrinth we decided to settle on the 3x3 grid of cameras. In order to make the simulation somewhat similar to the actual space of the testing site we have set the following rules to all the cameras.

- **Height:** 3m of the floor
- **FOV:** 50 degrees
- **Image dimensions:** 800x563
- **Update rate:** 30

The cameras were spread equidistant from each other to cover the full area of the labyrinth.

```
Unset
Example of one of the cameras:
<model name="camera_0">
                   <static>true</static>
                   <pose>1.3 -0.865 3 0 1.5708 1.5708</pose>
                   <link name="link">
                    <sensor type="camera" name="camera_0_sensor">
                      <camera>
                        <horizontal_fov>0.8726</horizontal_fov>
                        <image>
                          <width>800</width>
```

```
                              <height>563</height>
                              <format>R8G8B8</format>
                            </image>
                            <clip>
                              <near>0.1</near>
                              <far>100</far>
                            </clip>
                          </camera>
                          <always_on>1</always_on>
                          <update_rate>30</update_rate>
                          <visualize>true</visualize>
                          <plugin name="camera_controller"
filename="libgazebo_ros_camera.so">
                            <alwaysOn>true</alwaysOn>
                            <updateRate>0.0</updateRate>
                            <cameraName>camera_0</cameraName>
                            <imageTopicName>/camera_0/image_raw</imageTopicName>
           <cameraInfoTopicName>/camera_0/camera_info</cameraInfoTopicName>
                            <frameName>camera_0_link</frameName>
                          </plugin>
                        </sensor>
                      </link>
                </model>
```

3. Stitching the images from the cameras

After setting up the cameras in the Gazebo environment, we stitched the obtained images from the cameras. For this, we have created a CameraStitch class, in which we initialize the ROS node "camera_stitcher" and a list of images containing the images from nine cameras. We then set up ROS subscribers for nine camera image topics. Each subscriber uses a callback method passing an index to identify the camera.

The "stitch_images" resizes each image to a fixed size and then horizontally concatenates images in three groups, then vertically stacks these rows to form a single composite image.

The code can be found below:

```
Python
#!/usr/bin/env python3
import rospy
import cv2
import numpy as np
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

class CameraStitcher:
    def __init__(self):
        rospy.init_node('camera_stitcher', anonymous=True)
        self.bridge = CvBridge()
        self.images = [None] * 9  # List to store images from six cameras
        self.image_subs = [
            rospy.Subscriber("/camera_0/image_raw", Image, lambda msg, idx=0:
self.image_callback(msg, idx)),
            rospy.Subscriber("/camera_1/image_raw", Image, lambda msg, idx=1:
self.image_callback(msg, idx)),
            rospy.Subscriber("/camera_2/image_raw", Image, lambda msg, idx=2:
self.image_callback(msg, idx)),
            rospy.Subscriber("/camera_3/image_raw", Image, lambda msg, idx=3:
self.image_callback(msg, idx)),
            rospy.Subscriber("/camera_4/image_raw", Image, lambda msg, idx=4:
self.image_callback(msg, idx)),
            rospy.Subscriber("/camera_5/image_raw", Image, lambda msg, idx=5:
self.image_callback(msg, idx)),
            rospy.Subscriber("/camera_6/image_raw", Image, lambda msg, idx=6:
self.image_callback(msg, idx)),
            rospy.Subscriber("/camera_7/image_raw", Image, lambda msg, idx=7:
self.image_callback(msg, idx)),
            rospy.Subscriber("/camera_8/image_raw", Image, lambda msg, idx=8:
self.image_callback(msg, idx))
        ]

    def image_callback(self, msg, idx):
        try:
            cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
            self.images[idx] = cv_image
            self.stitch_images()
        except CvBridgeError as e:
            rospy.logerr(e)

    def stitch_images(self):
        if all(image is not None for image in self.images):  # Check if all images are
received
            # Resize images to fit in a 3x3 matrix
            resized_images = [cv2.resize(img, (400, 280)) for img in self.images]
            # Row 1 (first 3 images)
            row1 = np.hstack(resized_images[:3])
            # Row 2 (next 3 images)
            row2 = np.hstack(resized_images[3:6])
            # Row 3 (next 3 images)
            row3 = np.hstack(resized_images[6:])
            combined_image = np.vstack((row1, row2, row3))
            cv2.imshow("Stitched Cameras", combined_image)
            cv2.waitKey(1)

if __name__ == '__main__':
    stitcher = CameraStitcher()
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")
    cv2.destroyAllWindows()
```

This script efficiently handles real-time image processing from multiple cameras in a robotic application, demonstrating integration of ROS and OpenCV for practical tasks.

4. Using Reinforcement Learning

We tried to implement the reinforcement learning and create the best possible way to navigate the labyrinth and have settled on the following script.

Python
```
#!/usr/bin/env python3
import rospy
import numpy as np
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

class TurtleBot3Agent:
    def __init__(self):
        rospy.init_node('turtlebot3_rl_agent', anonymous=True)
        self.velocity_publisher = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
        self.laser_subscriber = rospy.Subscriber('/scan', LaserScan, self.laser_callback)
        self.current_scan = None

    def laser_callback(self, data):
        # Process laser scan data
        self.current_scan = data.ranges

    def choose_action(self, state):
        # Implement your action policy here
        return np.random.choice(['FORWARD', 'LEFT', 'RIGHT'])

    def update_environment(self, action):
        twist = Twist()
        if action == 'FORWARD':
            twist.linear.x = 0.2
            twist.angular.z = 0.0
        elif action == 'LEFT':
            twist.linear.x = 0.0
            twist.angular.z = 0.3
        elif action == 'RIGHT':
            twist.linear.x = 0.0
            twist.angular.z = -0.3
        self.velocity_publisher.publish(twist)

    def run(self):
        print(f"Running agent type: {type(self)}")
        rate = rospy.Rate(10)  # 10 Hz
        prev_state = None
        action = None
        while not rospy.is_shutdown():
            if self.current_scan is not None:
                state = self.process_scan_data()
                if prev_state is not None and action is not None:
                    reward = self.compute_reward(action)
                    self.learn(prev_state, action, reward, state)

                action = self.choose_action(state)
                self.update_environment(action)
                prev_state = state

            rate.sleep()

    def process_scan_data(self):
        # Process self.current_scan to create a state representation
        return np.mean(self.current_scan)

class TurtleBot3QAgent(TurtleBot3Agent):
    def __init__(self):
        super().__init__()
        self.learning_rate = 0.1
        self.discount_factor = 0.95
        self.epsilon = 0.1  # Exploration rate
        self.num_actions = 3
        self.num_states = 10  # Example discretization
        self.q_table = np.zeros((self.num_states, self.num_actions))


    def discretize_observation(self, data):
        # Simplified example: discretize the distance to the closest obstacle
        min_distance = min(data)
        state = int((min_distance // 0.1) * 0.1 * self.num_states)
        return min(state, self.num_states - 1)

    def choose_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice(['FORWARD', 'LEFT', 'RIGHT'])
        else:
            return np.argmax(self.q_table[state])

    def learn(self, state, action, reward, next_state):
        old_value = self.q_table[state, action]
        future_optimal_value = np.max(self.q_table[next_state])
        new_value = old_value + self.learning_rate * (reward + self.discount_factor *
future_optimal_value - old_value)
        self.q_table[state, action] = new_value

    def process_scan_data(self):
        state = self.discretize_observation(self.current_scan)
        return state

    def compute_reward(self, action):
        # Placeholder reward function
        if action == 'FORWARD':
            return 1.0  # Encourage moving forward
        return -1.0  # Penalize turning (simplistic assumption)

if __name__ == '__main__':
    agent = TurtleBot3QAgent()
    try:
        agent.run()
    except rospy.ROSInterruptException:
        pass
```

The video of operating the robot in the frames of our cameras can be found following this link:
Video Demo.mp4