2024 Robotics Assignment
Mobile Robot Challenge

Team RoboRover: Suju Li Shan Jiang Jiayi Wu

Abstract

This report details the execution and outcomes of the Mobile Robot Challenge, undertaken by Team RoboRover, which utilized a robot equipped with a Raspberry Pi processor to navigate and interact with its environment effectively. The challenge comprised two primary tasks: autonomous return to origin and object recognition followed by navigation. The team utilized color detection techniques for enhanced interaction capabilities. This study evaluates the technological implementations, discusses their efficacy, and explores the potential improvements for future iterations of robotic challenges.

1. Introduction

In this assignment, we use a robot equipped with a raspberry chip to complete two challenges. The first challenge was to manually maneuver the robot some distance away from its origin and then the robot was able to automatically return to its origin, as shown in Figure 1.

The second challenge was to manually command the robot to move a distance away from the origin and then return to the origin after the robot was able to recognize a specific object, approach and almost touch the object, as shown in Figure 2.

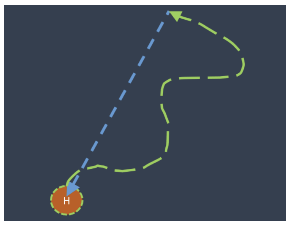

Figure 1. Challenge 1

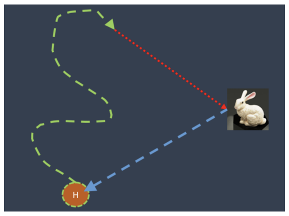

Figure 2. Challenge 2

2. Methodology[1]

2.1. Attempt 1: Based on SLAM

Since this mission only allows a monocular camera mounted on the robot, our team preferred Visual SLAM technology for implementation.

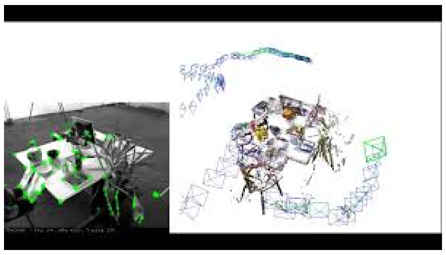

Figure 3. Simple example of Visual SLAM

Visual SLAM (Simultaneous Localization and Mapping) is a technology that utilizes visual information for simultaneous localization and map construction. It captures environmental images through a camera and uses algorithms for environment perception and self-localization. Figure 3 is a simple example of Visual SLAM.

The core components of Visual SLAM include feature extraction, feature matching, motion estimation, map updating, loop closure detection, and optimization. These components are well-suited for the tasks we aim to accomplish. Therefore, we decided to use a Visual SLAM library for our initial attempt to tackle this challenge.

Among the many available Visual SLAM libraries, the ORB-SLAM series is relatively well-known. Since our team members are more familiar with Python, we chose to use third-party libraries based on ORB that support Python syntax, specifically SLAMpy-ORBSLAM, as our experimental configuration library.

[1]All the detailed implementation could be found in our GitHub repository.

2.2. Attempt 2: Based on Color Detection

2.2.1. Movement Logs Maintaining&Backtracking

When the robot performs these four actions (forward, backward, turn left, turn right), we record these actions in the log, as shown in Algorithm 1. When we want to backtrack through all the steps, we change the action of each element inside the array *movement_logs*: change *forward* to *backward*, *backward* to *forward*, *turn_left* to *turn_right*, *turn_right* to *turn_left* ... and then execute from the end of the array to the beginning of it so we can walk back to the origin. This process is demonstrated in Algorithm 2.

Algorithm 1 Movement Logs Maintaining

```
running = True
movement_logs = []
while running do
    key = procedure READKEY()
    if key == q then
        running = false key in [w, s, a, d]
        start_time = time.time()
        action = {w: forward, s: backward, a: turn_left, d: turn_right}[key]
        move(action, 5)
        duration = time.time() - start_time
        movement_logs.append((action, duration))
    end if
end while
```

Algorithm 2 Movement Logs Backtracking

```
procedure MOVE
    if action == forward then
        fc.forward(power_val)
    end if
    if action == backward then
        fc.backward(power_val)
    end if
    if action == turn_left then
        fc.turn_left(power_val)
    end if
    if action == turn_right then
        fc.turn_right(power_val)
    end if
    fc.stop()
end procedure
reverse_commands = {forward: backward, backward: forward, turn_left: turn_right, turn_right: turn_left}
while movement_logs do
    action, duration = movement_logs.pop()
    procedure MOVE(reverse_commands[action], 5)
    time.sleep(duration)
end while
```

2.2.2. Target Searching and Approaching

We are using color to detect the target object. In this case, we are detecting the red color in the image that the camera captures in real time and using OpenCV to find the contours of the target.

Once we detect the target in the image, we attempt to touch it. We achieve this by computing the target's width percentage in the image and its position. If the target is placed on the left side of the image, it means the robot is facing the right side of the target, so we have the robot turn left slightly. Similarly, the robot turns right when the target is on the right side. The robot loops to adjust its angle until the target is at the center, indicating that the robot directly faces the target; then, it moves forward.

Since every movement of the robot may introduce an error, we take several measures to eliminate the influences of these errors. First, when the robot is far from the target, a slight change in the robot's angle can significantly affect the position of the target in the image, thus making it challenging to find the exact angle that places the target at the center. To overcome this, we set a larger tolerance value in the function that detects if the target is centered, allowing the robot to move closer to the target before we reduce the tolerance value. This adjustment enables the robot to locate the target more accurately as it gets closer. Second, to eliminate the effects of floor reflections and other interference factors, we can adjust the range in the $cv2.inRange$ function.

When the width of the target in the image is greater than 70% (we can adjust this value to control the closest distance between the robot and the target), we assume that the robot has touched the target. Then, we use the same method as in Challenge 1 to return the robot to the origin.

For more comprehensive understanding, please refer to the Algorithm 3.

3. Experiment

3.1. Indoor Environment

Lighting In all experiments, we closed the curtains and used two indoor light bulbs as light sources, which ensured the stability of the brightness and avoided the influence of outdoor sunlight.

Floor During the experiment (shown in the example videos), the robot walks on a regular wooden floor, which will have a much different coefficient of friction than the ceramic floor of the classroom in the demo show. Therefore it may lead to some errors.

Algorithm 3 rotate_move_rotate_touch Algorithm

```
procedure ROTATE_MOVE_ROTATE_TOUCH
    Initialize camera with 640x480, RGB888
    Start camera
    while $\neg is\_touch$ do
        Capture image $img$
        $img \leftarrow color\_detect(img, 'red')$
        Calculate $is\_item\_present$, $center\_state$, $width\_percentage$ from $img$ and $contours$
        if $is\_item\_present$ then
            if $width\_percentage < 0.2$ then
                Check center state, set tolerance = 60
                Adjust rotation    ▷ turn left, right, or forward
            else if $width\_percentage > 0.7$ then
                Stop the Robot
                Set $is\_touch = $ True
            else
                Check center state, set tolerance = 20
                Adjust rotation
            end if
        else
            Turn right    ▷ turn right to search for the target
        end if
    end while
    Destroys window and close camera
    return $movement\_log$
end procedure
```

Target Object For the second challenge, we used a red rectangular cardboard box with a length, width and height of about 5*5*9cm.

3.2. Hardware&Software Environment

3.2.1. Hardware

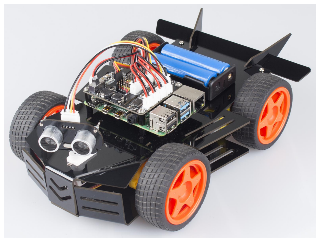

Figure 4. SunFounder PiCar-4wd Car Kit for Raspberry Pi

Regarding hardware, we chose the PiCar-4wd Car Kit produced by SunFounder, equipped with a Raspberry Pi 3 and a monocular camera. In addition, the package includes a Web terminal operation interface and some Python codes that support obstacle avoidance and line tracking functions. From the outside, it looks like a four-wheel-drive car equipped with a Raspberry Pi board with a hardware interface. A monocular camera is installed on the front of the car, and its pitch angle can be adjusted manually to adapt to different experimental needs. Figure 4 shows a simple example of the appearance of the SunFounder PiCar-4WD Car Kit for Raspberry Pi.

In addition, during our experiment, to install the required dependent environment, we used a network cable to connect the Raspberry Pi to the router to achieve the Internet access function.

3.3. Software Environment

In order to avoid environmental chaos and the need to reinstall the Raspberry Pi operating system, we built an isolated experimental environment in the Raspberry Pi. In this virtual environment, we use the Python that comes with the system as the interpreter. The robot component has a built-in picar_4wd library to control motor operation and implement basic operations such as forward and backward. For our first challenge, we tried writing our keyboard control code to capture keypress times. However, since the Keyboard library in Python requires root permissions to run, and the library is installed in our virtual environment when we call it with root permissions, calling the Python interpreter in the system root directory will cause packet loss errors. Therefore, we decided to adapt based on the code that comes with the machine instead of rewriting this part of the code.

4. Results

4.1. Attempt 1: Based on SLAM

During the process of configuring the environment, we encountered significant difficulties. Although, as mentioned above, we tried to configure two third-party libraries based on ORB-SLAM - system and SLAMPy-ORBSLAM, we encountered major challenges during the configuration process. These third-party libraries were last updated 4 to 7 years ago, resulting in serious environment compatibility issues. Some dependent versions have stopped being maintained and cannot even be downloaded. Ultimately, our installation attempt failed because we could not successfully configure these dependencies.

Subsequently, we decided to install and configure ORB-SLAM. However, the Raspberry Pi 3 mounted on the car cannot meet the computing power requirements of Visual SLAM. The car travels slowly, consumes abnormal power, and the machine is seriously hot, which clearly shows that it is unrealistic to configure the library on the car. Usually, in this case, we can deploy the environment locally, transfer the images captured by the camera to the computer for calculation and analysis, and then transmit the action decision back to the robot. However, we ultimately gave up on using Visual SLAM for this task because this approach was too cumbersome, and we were limited on time.

4.2. Attempt 2: Based on Color Detection[2]

4.2.1. Challenge 1

As we change our strategy to backtracking the path, we achieve obvious progress. After we operate the robot with commands, the robot can accurately return to the origin, as shown in Figure 5.

4.2.2. Challenge 2

In challenge 2, the robot runs with the command input in the terminal. The user can press the ”T” key to make the robot start detecting the target object. The robot can approach and touch the target and then return to the origin with a slight distance error. The more time the robot takes to find the target, the greater the distance error when it returns to the origin. This happens because each move of the robot, even with the same interval and angle, differs slightly. The more the robot moves, the more these differences accumulate.

Besides that, we also eliminate the interference of the target object’s reflection on the floor by adjusting the color detection parameters. By tuning the saturation and brightness of the detection function, we can eliminate the effect of other elements.

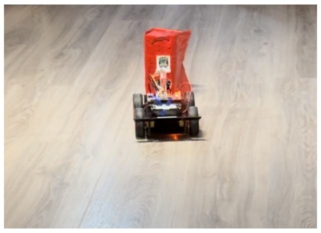

Figure 5. The robot touched the target object.

5. Analysis

The methodologies adopted by Team RoboRover, particularly the use of color detection, represent a significant stride toward integrating complex computer vision techniques with real-time robotic control. Color detection algorithm facilitated the robot’s interaction with objects based on their color attributes while simultaneously tracking its own movements. This approach was crucial for the robot’s ability to return to its starting point autonomously.

[2]Test videos for both challenges could be found here.

6. Limitations

Despite the successes reported, several limitations were evident in the study:
Hardware Constraints The reliance on a monocular camera limits depth perception, which is critical for more nuanced navigation and interaction tasks.
Environmental Dependency The performance of color detection algorithms is highly contingent on environmental conditions such as lighting and the presence of distinct visual features.
Error Accumulation Inherent in our movement backtracking technologies, error accumulation over time can lead to significant deviations in mapping and localization, affecting long-term reliability.
Generalization The algorithms’ ability to generalize to different environments or tasks without extensive reconfiguration remains questionable.

7. Conclusion and Future Work

The Mobile Robot Challenge showcased the effective application of advanced algorithms in a practical robotics context, with Team RoboRover demonstrating significant proficiency in employing Visual SLAM and color detection techniques. The robot successfully navigated to and from a designated origin point and interacted with specific objects based on visual cues. However, the challenges highlighted crucial areas for improvement, particularly in enhancing the robustness and adaptability of the systems used.

To address the current limitations and enhance future implementations, several areas of development are recommended:
Enhanced Sensory Hardware Integrating stereo cameras or LIDAR could improve depth perception and environmental mapping accuracy.
Algorithm Optimization Developing less resource-intensive algorithms could allow for smoother operation on limited-capacity processors.
Adaptive Lighting Solutions Incorporating adaptive algorithms that can adjust to varying lighting conditions would make the system more robust to environmental changes.
Error Correction Mechanisms Implementing advanced error correction techniques in SLAM could reduce cumulative mapping errors.
Cross-Environment Generalization Future studies should focus on creating more adaptable models that can perform across a broader range of environments and tasks without needing extensive retraining.

8. Appendix

All dependencies and instructions for running the code could be found in the README file in our GitHub repository. You can click **here** to access it.