EECE 4376

Final Project Report

Zakariyya Al-Quran
Edwin Campbell
Michael Delaney

December 9, 2022

User's Manual

Introduction:

This project uses a Raspberry Pi robot car to traverse a room and measure its dimensions. The main program follows the walls of a room until it has reached the starting point, then terminates. The program stores its readings of points to the local file system to allow processing later. The data is also streamed over the local network to any listeners running the second optional program to create a 2D mapping of the room in real time.

Program Overview:

The main python program runs on the robot and uses multiple concurrent threads to enable the robot to navigate the room while reading its surroundings. The second python program runs on another device on the network and listens to the data being published by the robot, plotting the points on a simple GUI for the user to visualize. The programs are available on GitHub at: https://github.com/zakarlyya/mapping-room-robot/

Configuring the Raspberry Pi and Robot:

To set up a Raspberry Pi to run the program in this project, you can use the following steps:

1. Install Python 3 and the necessary libraries by running the following command:

```
sudo apt-get install python3 && pip3 install pyzmq pyqt5 pyqtgraph numpy
```

2. Construct the robot including the motors, pan-tilt servo mechanism, and ultrasonic sensor according to the manufacturer's instructions. Documentation can be found here: https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for_Raspberry_Pi
3. Configure the Raspberry Pi by setting up Wi-Fi connectivity. Ensure the Pi and the computer are on the same Wi-Fi network.
4. Obtain the Pi's wireless IP address and connect to the Pi over SSH.

Starting the Program:

1. Download or clone the main program (main.py) to the Raspberry Pi. Also, download the server program (server.py) to another computer.
2. Start the server program on the computer and enter the Pi's address when prompted:

```
python3 server.py
```

3. Run the main program, using the command below:

```
python3 main.py
```

   The main program should now begin running on the robot. The server program on the other device will be listening for data to be sent over and graphed.

Running the Program:

Before mapping, the robot must calibrate itself. After starting the program, use the following steps to begin the calibration and mapping process:

1. Place the robot facing directly towards a wall in the room to map at about 1-3 feet away.
2. Enter "START" to calibrate. The robot will then calibrate the sensor and motors, then move closer to the wall and prompt the user with “GO” when it is ready to proceed.
3. Enter "GO" to start the mapping process. The robot will turn left to become parallel to the wall and begin mapping the room. The head will pan, collecting data on the nearby walls. While the robot is panning, the data collected is converted to absolute points that are stored and broadcast to any devices running the server program.

   The points will be displayed continuously on the server program throughout the mapping process showing a 2D view of the room from above as it is being explored.
4. After the ultrasonic sensor has panned once entirely, it will decide to either move forward or turn depending on how far away the robot is from the nearest wall in front of it. This process will repeat until all of the room’s walls have been traversed.
5. The program will either terminate upon completion of the mapping or if the user sends the “STOP” signal manually from the main program terminal.

Troubleshooting:

To troubleshoot potential issues with the robot or program, consider the following points:

- The robot's turning values may need to be adjusted if the robot does not turn properly (90 degrees around walls). How much the robot turns depends on the motors, wheels, surface, batteries, etc. The default value has been tested on both hardwood and carpet flooring.
- Increasing the number of sensor measurements can improve the accuracy of the sensor readings but will slow down the robot’s total traversal time.
- Erroneous data may be measured due to ultrasonic sensor interference or nearby objects. Errors in the robot's position can lead to an accumulation of errors over time. To avoid this, regularly check and calibrate the sensor and the robot's position.
- If the batteries are low, then the processor will throttle and the power supplied will be insufficient to map the room. To prevent this, use fully charged batteries when possible.
- The robot can operate independently of the server program, but ensure that the programs have both exited before starting again to prevent scrambled data.

Technical Manual

Introduction:

The project uses a Raspberry Pi-powered robot car with an ultrasonic distance sensor to traverse and map a room's outer walls autonomously. The main program is constructed of subclasses that each control some aspect of the robot as shown in Figure 1 below. The subclasses—which are launched as individual threads—communicate through ZMQ sockets. The logic thread manages the communications and decides the robot's path based on the ultrasonic sensor readings. The program also stores its ultrasonic readings as absolute points representing the walls and streams the points over the local network to a secondary program which can plot the room in real time.

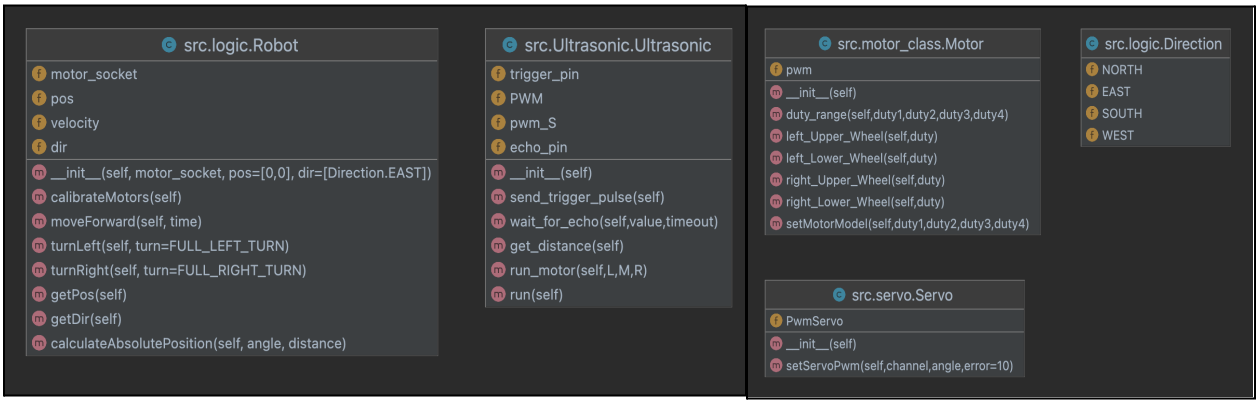

Figure 1. A Class Diagram of the Main Program

Principle of Operations:

The main program launches the logic thread, the primary driver for the mapping and traversal, and provides a simple interface for the user to start the mapping and stop at any time. The other threads involved are the motor thread, sensor thread, and logic thread.

The motor thread is a simple driver that follows the command pattern. It receives the request to move the motors from the robot class, sends the proper signal to the motors over the GPIO pins of the Pi, and sends a response once the request has been completed.

The sensor thread abstracts the robot head which continuously pans the sensor while measuring the distance of objects in front of it. For each angle the head is panned, the sensor reads multiple times, discards outliers to ensure accurate data, calculates the average distance measured, and publishes the reading in polar coordinates.

In addition to the other threads, logic has a singleton instance of the robot class which maintains the robot's position and direction and accepts requests to move the robot forward or turn left/right. Any decision to move the robot must go through the robot class which will send the respective motor request, update its position and direction accordingly, and publish the robot's new position. This ensures that the position of the robot is always accurate as the robot traverses the room. To ensure the robot moves correctly, a simple PID controller is implemented in the robot class to minimize the robot drifting towards or away the wall it is following.

Upon receiving enough readings from the sensor, the robot must decide to move forward to turn. If the robot was approaching a wall straight ahead, then it would turn left as shown in scenario 1 of Figure 2. If the wall that the robot was following on its right was no longer detected, then it would turn right as shown in scenario 2 of Figure 2. Otherwise, the robot would continue moving forward. If no decision could be made confidently, then the robot would move forward slightly.

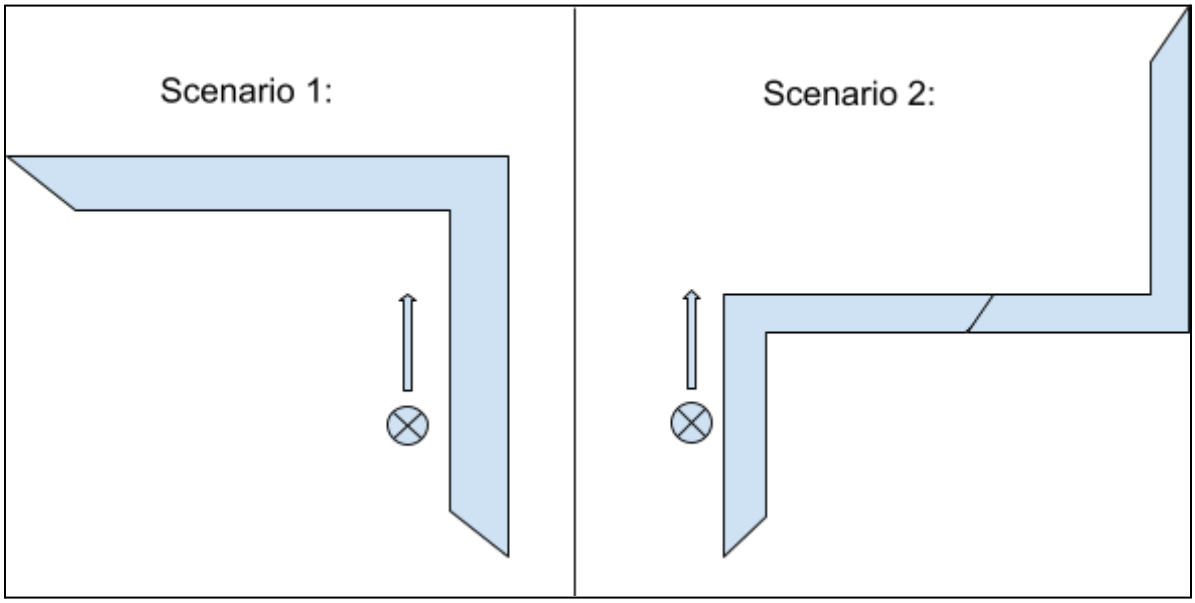

Figure 2. The scenarios where the robot would turn left or right

While traversing, the readings from the ultrasonic sensor are filtered and transformed into points that are published locally to any device running the GUI server which plots the robot as it moves and the walls detected. The robot follows the walls around a room until it has performed a net total of four left turns which marks that the robot has completed a complete loop.

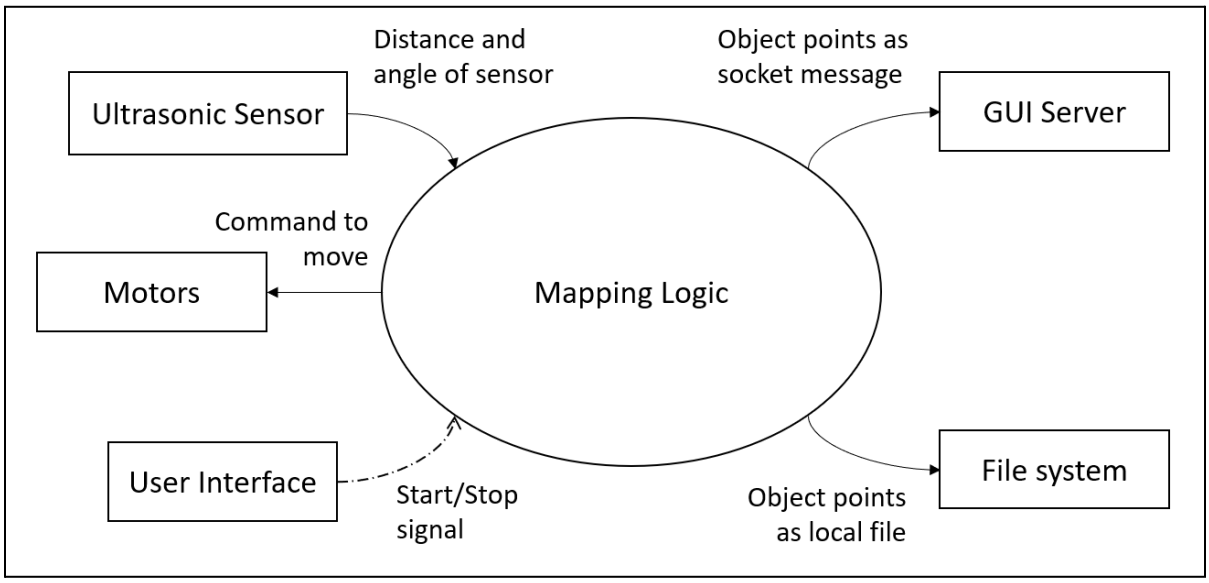

Figure 3. Context Diagram for the Mapping Logic

Explanation of Design:

Main

The main code launches multiple threads for logic, motors, and server. It then waits for the user to input "START" or "STOP" to then signal to the logic thread using a ZMQ REQ/REP socket. The code also logs messages using the logging library. Once the threads are started and the input is received, the code waits for the threads to terminate before closing the sockets and exiting.

Logic

The data flow diagram shown in Figure 4 shows the essential data processing functions of the mapping logic found in the logic.py file.

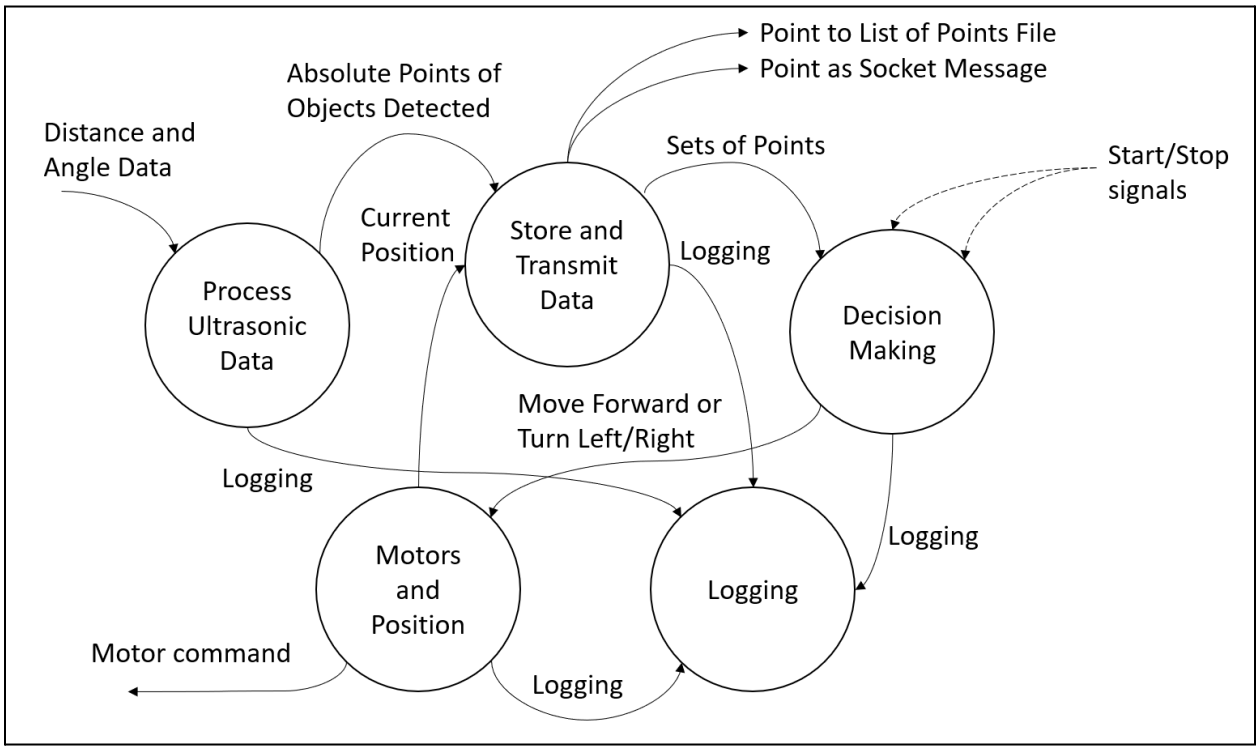

Figure 4: Data Flow Diagram for the Mapping Logic

The program flow is as follows:

1. Import the necessary libraries and class components
2. Define the *Direction* enumeration
3. Create a ZMQ context with four ZMQ sockets along with a ZMQ *Poller* to manage the multiple sockets:
    a. one for receiving START/STOP signals from *main* (start_socket)
    b. one for sending the motor movement requests (motor_socket)
    c. one for subscribing to data from the sensor thread (sensor_socket)
    d. one for publishing of data to the server (server_socket)
4. Create an instance of the Robot class, calibrate its motors, and wait for the START signal
5. Turn the robot to be parallel to the wall, set the robot's starting direction and position

6. Enter a loop until the robot has reached its starting position
    a. Poll all the sockets
        i. If a STOP signal is received from *main*, terminate the program
        ii. If any new points are received from *sensor*, add them to the list of points and calculate the absolute position of each point
        iii. If *motor* responds that it has moved, set the robot as ready to move again
    b. Decide on where to move
        i. If points are detected in front of and to the right of the robot, turn left
        ii. Else if no wall is detected to the right of the robot, turn right
        iii. Else move forward and if the robot is drifting, correct using a simple PID controller to realign with wall
7. Stop the robot and close all the sockets.

Sensor

The function continuously pans the robot's ultrasonic sensor back and forth, measuring the wall distance using the sensor, and publishes the measured distances. More specifically, it must:

1. Import the necessary libraries along with the *Servo* and *Ultrasonic* classes
2. Create a ZMQ context and socket for publishing sensor data to *logic*
3. Create an instance of the *Servo* and *Ultrasonic* classes
4. Enter a loop until an interrupt signal is received
    a. Read multiple times from the ultrasonic sensor to reduce noise
    b. Removes outliers using median, then calculate the average distance measured
    c. Send the average distance and current angle to *logic* through the socket
    d. Update and pan the sensor angle
5. Close the socket

Motors

This function contained in the motor.py script receives motor movement requests in the form of [action, time] from logic.py, executes them, then responds when it has finished.

1. Import the necessary libraries and the *Motor* class from the Freenove code provided
2. Create a ZMQ context and server socket for REQ/REP communication with *logic*
3. Create an instance of the *Motor* class
4. Enter a loop until an interrupt signal is received
    a. Wait for a request on the socket
    b. Parses the request message
    c. Executes the corresponding motor movement for the specified time
        i. “Move forward” moves all four motors forward
        ii. “Turn left” moves the right motors forward and the left motors backwards
        iii. “Turn right” moves the right motors backwards and the left motor forward
    d. Send a reply message indicating the movement was completed
5. Close the socket

Server

This program receives points published by the robot's *logic* script and displays them on a scatter plot in real-time. The *MyWidget* class that forms the GUI window creates a *PyQtGraph* plot with a timer. Upon timing out, the *getNewData* function is called which waits to receive a point from a ZMQ socket, adds the point received to a set of points, and updates the plot. The program also listens for "done" messages that signal the completion of mapping and end of data transmission.

1. Import the necessary libraries including PyQt5, PyQtGraph, NumPy, and ZMQ
2. Define a *MyWidget* class that extends PyQtGraph's *GraphicsLayoutWidget* to create a custom widget for displaying the live scatter plot
    a. Set the window's layout, title, plot, and other properties
    b. Create and bind a timer to *MyWidget*
6. Define the getNewData method that is called on each time out:
    a. Receive new data published to a socket
        i. The data is either the robot's position or a sensor reading
    b. Parse the data (string) and transforms it into a point (pair of values)
    c. Add the data to the existing set of data
    d. Update the plot with the new data
7. Create a ZMQ context and socket that subscribes to the robot using its IP address
    a. Subscribe to all messages
8. Create an instance of the *MyWidget* class which starts the timer and opens the window
9. Update the plot until a "done" message is received. Then stop the timer, close the socket, and prompt the user to exit.

Conclusion:

The main thread is responsible for setting up and starting the program, while the logic thread handles the communication between threads, processes data received to calculate the positions of the room walls, and decides how to proceed around the room using a simple voting system and PID controller. The sensor thread reads the ultrasonic sensor and sends out relative polar coordinates. The motor thread moves the robot. The second program runs a server thread that receives data from the robot and displays a working image of the room. All of the threads use ZMQ sockets for communication.

In conclusion, the autonomous mapping robot employs many features of embedded systems including networking to communicate with other processes or devices, threading to split up the program into concurrent tasks, and I/O to interact with the physical world. The simple yet efficient design allows the robot to traverse a room and map its walls using a single ultrasonic distance sensor. The algorithm uses a combination of sensor readings, motor control, and decision-making logic to guide the robot around the room, recording both its movements and surroundings. While the system is not highly advanced or sophisticated, it provides an example of the basic concepts, design processes, and implementation of embedded systems.

Project Personnel

Overview:

The roles and responsibilities of the group members varied depending on their individual skills and expertise. In general, all members of the group were responsible for contributing to the project design, implementation, and testing by providing input and ideas during group meetings, working on project-specific tasks, and testing the robot regularly. Given the scope and duration of this project, it was important for all members to communicate effectively and work together in order to ensure the success of the project.

Additionally, all members were expected to attend weekly meetings and actively participate in discussions about the project. These meetings provided an opportunity for members to share their progress, discuss any challenges they were facing, and come up with solutions to any problems that may have arisen. It was important for every member to attend these meetings and contribute to the discussions to ensure that the project would be completed successfully.

Responsibilities:

The group members who worked on this project were Zakariyya Al-Quran, Edwin Campbell, and Michael Delaney. Each group member was expected to construct, setup, and test their Raspberry Pi 4B and Freenove 4WD robot when the project began. Then, tasks were delegated to each member. Edwin worked with the team on the overall goals of the project, calibration, and logic design. Michael worked on the motor thread, concurrency, and logic design. Zakariyya worked on the sensor thread, graphical user interface, networking, and logic as well. All group members were expected to test all the code. The program was mostly tested and then demoed on Zakariyya's robot, although cross- robot functionality was verified using Michael's robot as well. Members also had to contribute to the project documentation including multiple presentations and a final report.

Appendix

Parts List:

- Freenove Smart Car Robot with 4WD and ultrasonic sensor
  - https://www.amazon.com/dp/B07YD2LT9D
- Raspberry Pi 4B running Raspberry Pi OS
  - https://www.okdo.com/p/okdo-raspberry-pi-4-4gb-model-b-starter-kit/
- Any laptop capable of SSH and running Python files

Libraries Used:

- PyZMQ for communication between threads and devices
  - https://github.com/zeromq/pyzmq
- Freenove 4WD Smart Car Code for the robot's motor and sensor drivers
  - https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for_Raspberry_Pi
- PyQt5 & PyQtGraph for GUI framework and real-time plotting
  - https://github.com/pyqt
- Threading for concurrency between logic, motors, and sensor
  - https://docs.python.org/3/library/threading.html

Program Files:

File Name	Description
main.py	The main program that executes on the robot. This file imports all the necessary files, controls the execution, and launches all the relevant threads.
logic.py	This file contains the actual logic for the room mapping and decision making. Within logic, it connects to the sockets to communicate with the main thread (waiting for a "START" or "STOP"), sensor thread (waiting until the robot has panned entirely), and motor thread (sending commands to move the motor and receiving acknowledgement). It also saves the data processed into points and publishes the data over a socket for any listeners. The decision on whether to go straight, turn left, or turn right uses a voting system. The traversal also contains a simple PID controller to correct any drift in the robot's movement.
sensor.py	File containing the thread which pans the ultrasonic sensor, collects readings, filters outliers, and sends average distance measured at each angle.
motors.py	File containing the thread which accepts requests and moves motors.
server.py	The second program that runs on another device. It connects (subscribes) to the robot, creates a GUI, and plots points as they are received.
sensor_test.py	Tests the sensors.py and server.py functionality.
motors_test.py	Tests the motors.py functionality by accepting user-inputted requests.