Smart Driving Assistance

— Embedding Systems 2023 Fall ｜ Final Project Report —

Group 5 | B09901116 陳守仁 B08901207 呂俐君

Github repositories URL | https://github.com/lawraa/ESLab_SmartParking

I. Concept & Motivation ｜動機

Driving in Taipei is an integral part of daily life but often comes with its own set of challenges and dangers. To address this, our project aims to design a ***smart driving assistance system***. The goal is to provide real-time alerts about the direction and distance of surrounding objects, offer instructions and help for making safe navigational decisions, and present information in an intuitive visual format to enhance safety on the roads.

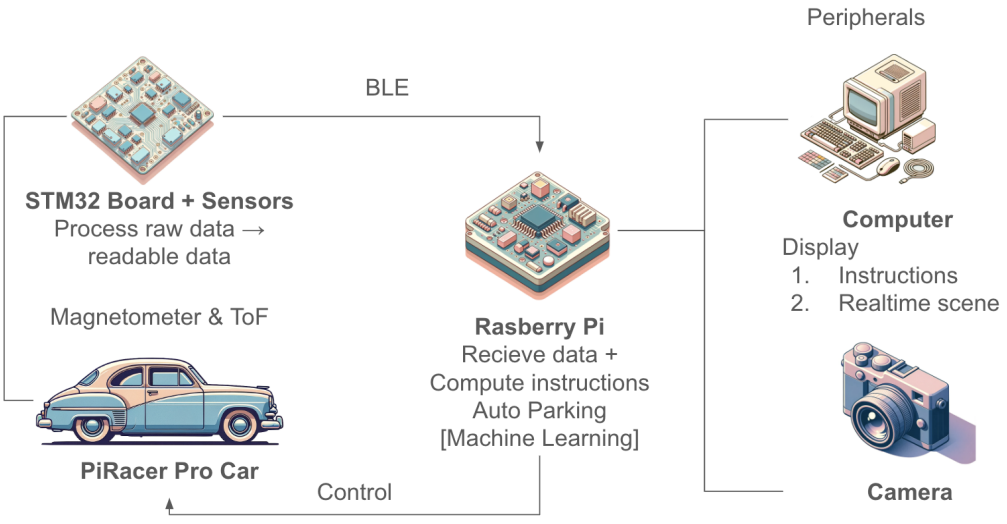
II. Design & Hardware | 功能設計及硬體設備

A. Key Functionality | 主要功能

1. 即時數據 - Realtime Measurement: Distance
    a) Functionality: This feature measures the distance between the car and potential obstacles using the ToF (Time of Flight) sensors on the STM32 B-L4S5I-IOT01A Discovery Kit.
    b) Implementation Detail: The ToF sensors continuously send distance data to the Raspberry Pi via BLE (Bluetooth Low Energy). This real-time data is crucial for making immediate driving decisions.
2. 駕駛模式 - Driving Mode: Normal Driving, Parking Mode
    a) Functionality: The system has two primary driving modes - Normal Driving and Parking Mode. Normal Driving allows for standard vehicle operation, while Parking Mode activates specific algorithms for parking.
    b) Implementation Detail: The mode selection is controlled via the Raspberry Pi, which sends commands to the car's control system. This mode selection changes the behavior of the car in terms of speed, steering, and sensor data interpretation.

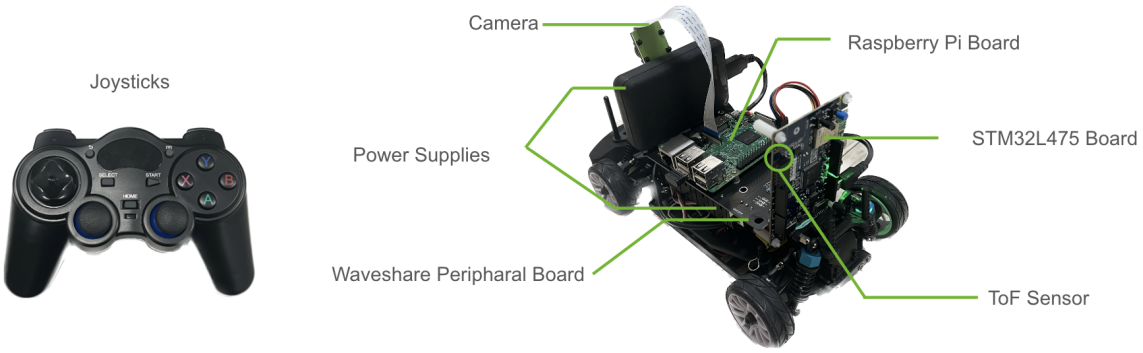
3. 危險偵測 - Danger Detection: Warning Messages, Brakes Control
    a) Functionality: Danger Detection involves identifying potential hazards like imminent collisions and automatically applying brakes or issuing warning messages.
    b) Implementation Detail: This feature uses the distance data from the ToF sensors. If a threshold distance is breached, the system triggers the brakes or displays warning messages on the screen.
4. 螢幕顯示 - Screen Display: FPV Scene, Mode, and Messages
    a) Functionality: The screen display shows the First-Person View (FPV) from the car's camera, current driving mode, and any important messages or alerts.
    b) Implementation Detail: The Raspberry Pi processes the camera feed and overlays information about the driving mode and messages. This is streamed using MJPEG over HTTP.
5. 自動停車 - Auto Parking
    a) Implementation Detail: Auto Parking was intended to use machine learning (CNN Behavioral Cloning) feed in camera view, steer, throttle values, to enable the car to park itself autonomously by learning from human driving patterns.

B. Architecture Diagram | 架構圖

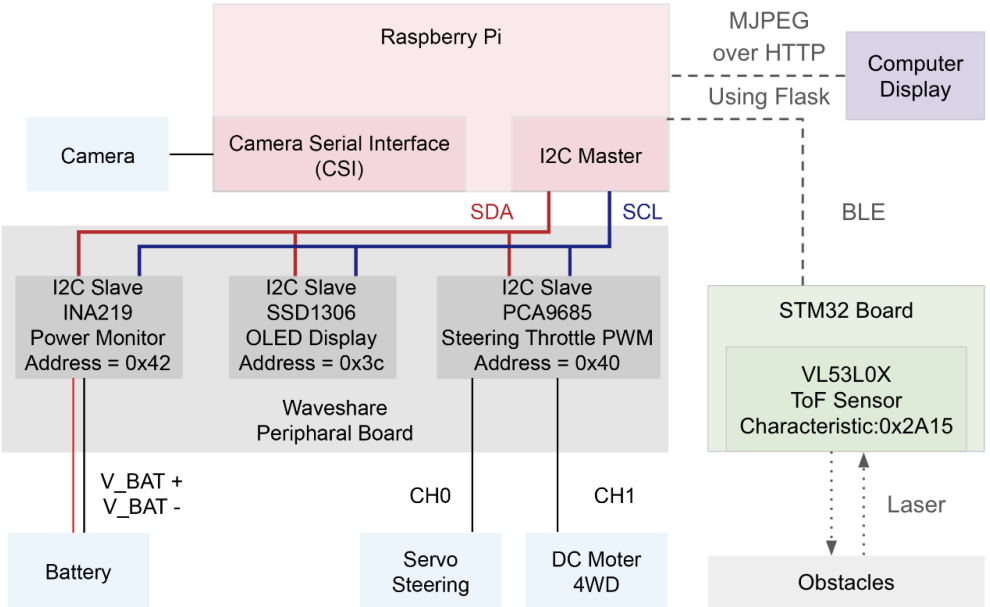


C. Devices | 裝置

1. STM32L475 Board
    a) VL53L0X Time of Flight Sensor
2. Raspberry Pi 3 Board
3. PiRacer Pro
    a) Waveshare Paripharal Board
    b) HD wide angle camera
    c) Joystick



D. Hardware Block Diagram | 硬體架構



III. Methodology - Connection & Communication | 連接與溝通實作

A. Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is used in controlling motors.

- *Efficient control* of the power delivered to a device
- *Precise Motor Speed* and *Position Control*

Procedure

1. Parameters

   PWM frequency = 50 Hz

   PWM Resolution

   Max Raw Value = 2^{16} - 1
2. Steps:
   a) User Input: User input a range of -1 to 1 for motor speed.
   b) Get Duty Cycle from Percent:

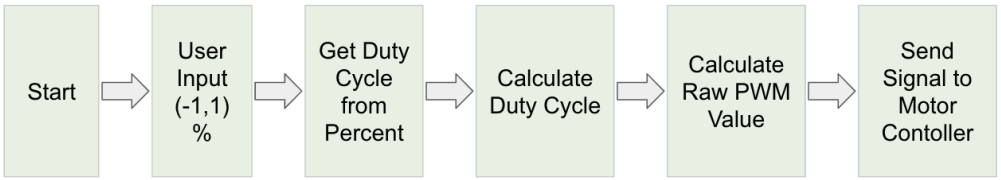
   $$0.0015 + (user\_input\_value\{-1, 1\} \times 0.0005)$$

   c) Calculate Duty Cycle/Raw PWM Value

   $$Raw\ Duty\ Cycle = Max\ Raw\ Value * (\frac{Time}{PWM\ Wavelength\ of\ 50Hz})$$

   d) Send Signal to Motor Controller

```python
pwm_controller.channels[channel].duty_cycle = raw_value
```



PWM Flow Chart

B. Inter-Integrated Circuit (I^2C)

I^2C Connection for Peripheral hardwares for reading and controlling signals.

Peripherals	**Address**
Servo Steering (Channel 0) and Motor (Channel 1)	0x40
OLED Display	0x3c
Battery	0x42

C. Camera Serial Interface (CSI) Connection

Use a CSI connection for our Raspi Camera for the following reasons:

1. Consumes **low power**.
2. It has **lower pin count**, which reduces EMI and PCB space and complexity issues.
3. **Simple** interface protocol
4. **High bandwidth** to support higher resolution

D. Hypertext Transfer Protocol (HTTP) Communication

1. Local Server:
    a) Real-time data processing and immediate decision-making
    b) Local server setup also reduces latency compared to cloud-based or web server.

Setting Up the Local Server on Raspberry Pi:

(1) First configured the Raspberry Pi with a suitable **network** to ensure it can handle incoming and outgoing data requests.

(2) We then installed and imported libraries to support our server applications.

```python
from flask import Flask, Response
import cv2
from picamera2 import Picamera2
from libcamera import Transform
```

(3) To access the server, we established a specific IP address and port number. This way, by entering the IP address in a web browser on the local computer, we could access the server interface.

```python
def gen_frames():
    # Configuration for camera, including flipping the image
    .........

@app.route('/video_feed')
def video_feed():
    return Response(gen_frames(), mimetype='multipart/x-mixed-replace;
boundary=frame')
```

```
if __name__ == '__main__':
    # Start the Flask application
    # Change to your own IP to use
    app.run(host='172.20.10.2', port=8082)
```

2. Data format:
   MJPEG transmits a sequence of JPEG images, making it easier to handle on less powerful hardware like the Raspberry Pi. This format is widely supported and can be easily integrated into web interfaces for real-time video streaming.
   Pros:
   a) **Simplicity**: Easy to implement and requires less computational power.
   b) **Real-Time Capability**: Suitable for live video streaming without significant processing delays.
   c) **Compatibility**: Supported by most web browsers and devices without needing additional plugins or software.
   Cons:
   a) **Lower Video Quality**: MJPEG might not deliver the same video quality at the same bitrate as more advanced codecs.

```python
# Encode the frame for streaming
    ret, buffer = cv2.imencode('.jpg', frame)
    frame = buffer.tobytes()
    yield (b'--frame\r\n'
       b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n')
```

3. Multi-Threads & Lock:
   a) Multi-threading is an efficient way for the CPU to manage multiple tasks simultaneously. In this case, it allows the CPU to collect data while updating realtime scenes at the same time.

```python
# mycar/car_controller.py
import threading
```

```
# Starting Flask web server and BLE scanning in separate threads
  flask_thread = threading.Thread(target=run_flask_app)
  flask_thread.start()
  ble_thread = threading.Thread(target=continuous_scan)
  ble_thread.start()
```

b) The `threading.Lock` serves as a mutual data manager that ensures the data is only acquired by one thread once a time to avoid inconsistency.

```python
# mycar/global_state.py
import threading

class GlobalState:
    def __init__(self):
        # Initialize the default state
        ......
        self.lock = threading.Lock()  # Lock for thread-safe operations
```

E. Bluetooth Low Energy (BLE) Communication

1. Concept

   We use BLE for communication between STM32 board and a Raspberry Pi, for sending sensor data for these strengths:

   a) **Low Power Consumption**: BLE is designed for low power usage, making it ideal for battery-operated devices like sensors on an STM32 board.

   b) **Compatibility and Accessibility**: BLE is widely supported, and Raspberry Pi has built-in support or can easily be equipped with BLE capability. This makes the integration of devices simpler.

   Compared to other wireless communication methods, BLE exhibits a slightly longer delay. However, in this case, such latency is not a significant concern. Therefore, opting for BLE represents a favorable tradeoff.

2. Implementation

a) ***Setting Up BLE on STM32 as a server***

First We need to import the `BLE` library and our customized Service header files in which we deliver the updated sensor value via BLE.

(1) *Universally Unique Identifier (UUID) & Characteristics*

We assign UUID to our device and characteristics for specific sensors to ensure the correct communication of data.

```
C/C++
/** UUID of the ToF service. lichun added*/
 UUID_DISTANCE_MEASUREMENT_CHAR = 0x2A15,
/** UUID of the ToF service. lichun added*/
 UUID_DISTANCE_SERVICE = 0x180B,

    _tof_uuid(GattService::UUID_DISTANCE_SERVICE),
    _adv_data_builder.setLocalServiceList({&_tof_uuid, 1});
```

(2) *Advertising & Connection*

```
C/C++
 void start_advertising(){
        /* Create advertising parameters and payload */
            ble::AdvertisingParameters adv_parameters(
            ble::advertising_type_t::CONNECTABLE_UNDIRECTED,
            ble::adv_interval_t(ble::millisecond_t(100)));
    }

print_mac_address(); //get the address of this device = 'e3:9f:46:c7:39:e1'
```

(3) *EventQueue & Update*

Event Queue properly allocates CPU for different tasks. In this case, the `_event_queue` manages to update the value of sensors every 300 ms.

In function `update_sensor_value`, we call back to `tof_service` for synchronizing via BLE.

```
C/C++
static events::EventQueue event_queue(/* event count */ 16 * EVENTS_EVENT_SIZE);
       _event_queue.dispatch_forever();
```

```
void on_init_complete(BLE::InitializationCompleteCallbackContext *params)
    {
        ......
        // value updated every 300ms*/
        _event_queue.call_every(300ms,[this] {update_sensor_value();});
        start_advertising();
    }
  void update_sensor_value(){
    ......
    _tof_service.updateDistance(_distance); }
```

b) *Setting Up BLE on Raspberry Pi as a client*

After our STM32 server started advertising, we would start R-Pi as a client to connect via BLE and receive data.

Python

```
# mycar/ble_client.py
def continuous_scan():
       # Connect to a specific device with Mac address = 'e3:9f:46:c7:39:e1'
       dev = Peripheral('e3:9f:46:c7:39:e1', 'random')
       # Set the delegate for handling notifications
       dev.setDelegate(MyDelegate())
       try:
       # Setup and handling of specific BLE services and characteristics
              hrc = dev.getCharacteristics(uuid=UUID(0x2A15))[0]
              print(f"ToF cHandle: {hrc.getHandle()}")
```

IV. Functionality Implementation | 功能實作

A. Real Time Distance Measurement

To measure real time distance of the car, we apply a STM32 board with a built-in VL53L0X module that can sense time-of-flight to get distance in millimeter format. First, We need to import the VL53L0X library for Mbed-OS projects. Here, we periodically get the data by `continuous_polling` mode.

```C/C++
#include "VL53L0X.h"
// Setup ToF Sensor
static DevI2C devI2c(PB_11,PB_10);
static DigitalOut shutdown_pin(PC_6);
static class VL53L0X VL53L0X_Sensor(&devI2c, &shutdown_pin, PC_7);

  void update_sensor_value()
  {
    //update value
    status = VL53L0X_Sensor.get_measurement(range_continuous_polling,
&measurement_data);
    _distance = measurement_data.RangeMilliMeter;
    ......
  }
```

B. Driving Mode

The Pi-racer Pro has a wide range of speed from 100% to 0%. We designed different driving modes for different purposes. Users can simply press button `B` on the joystick to switch modes.

Mode	Speed	Purpose
Driving Mode	50%	Normal navigation, it's fast and suitable for straight and wide tracks.
Parking Mode	22%	Used when users need fine movements that can be better implemented under slower speed. Suitable for rough and narrow tracks or parking.

C. Danger Detection & Controls

The danger detection is similar to the parking sensor widely used in modern automobiles. It utilizes the distance obtained from the ToF sensor, and sets up 2 thresholds values for 3 stages.

1. Safe: When the distance is ≥ 400 mm, it is far from danger.
2. Display Safety Distance: When the distance gets closer but still in a safe range, we *inform users with the distance* without intervention.
3. Emergency Stop: When the distance ≤ 100 mm, the system *takes over the driving controls* so that users can no longer move toward the directions.

D. User Interface

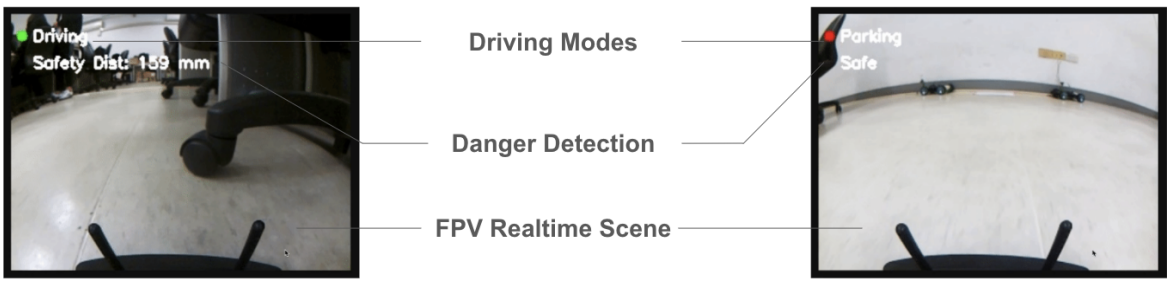
1. Purpose: Allow User to see live their front view and also some info and warnings if the car is in danger.

```python
  transform = Transform(hflip=True, vflip=True)
  picam2 = Picamera2()
  video_config = picam2.create_video_configuration(main={"size": (320, 240),
"format": "RGB888"}, transform=transform)
  picam2.configure(video_config)
  picam2.start()
  # Loop to capture and annotate frames
  while True:
    frame = picam2.capture_array()
```

2. We use CV2 to create updated distance and modes

```python
tof_distance_text = ...
mode_text = f"{global_state.global_state.mode}"
cv2.putText(frame, mode_text, (20, 25), font, 0.5, font_color, line_type)
cv2.putText(frame, tof_distance_text, (20, 50), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
(255, 255, 255), 2)
```

3. Real Time Display



E. Auto Parking

1. Purpose and Concept

   We want to enable smart cars to autonomously park. With the help of AI and machine learning, people will not struggle to park anymore.

2. Methodology

   Follow Training on Steps on

   https://docs.donkeycar.com/guide/deep_learning/train_autopilot/

   Require installation of https://github.com/autorope/donkeycar.git

   a) Training Method

      (1) **CNN Role**: The CNN is responsible for processing the visual input from the car's camera and interpreting it to understand the surrounding environment, particularly identifying suitable parking spots and obstacles.

      (2) **Behavioral Cloning**: Recording human driving behavior (steering angles, acceleration, etc.) in parking. The model then learns to replicate these behaviors, allowing the car to execute similar maneuvers during auto parking.

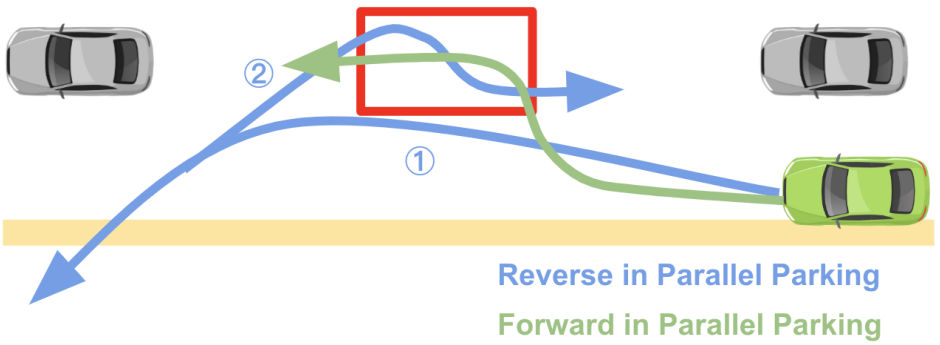
   b) Image Capture Mechanism

      $AUTO\_RECORD\_ON\_THROTTLE = True$ - *recording*

      (1) Turn on condition: when you apply throttle

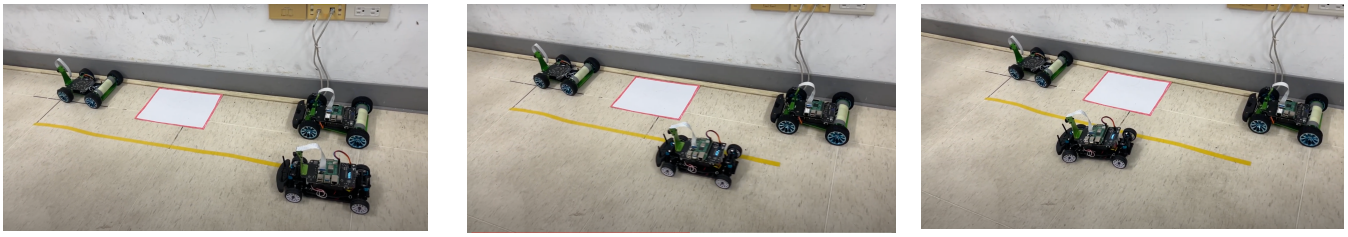
      (2) Turn off condition: when you stop applying throttle

   c) Parking Path



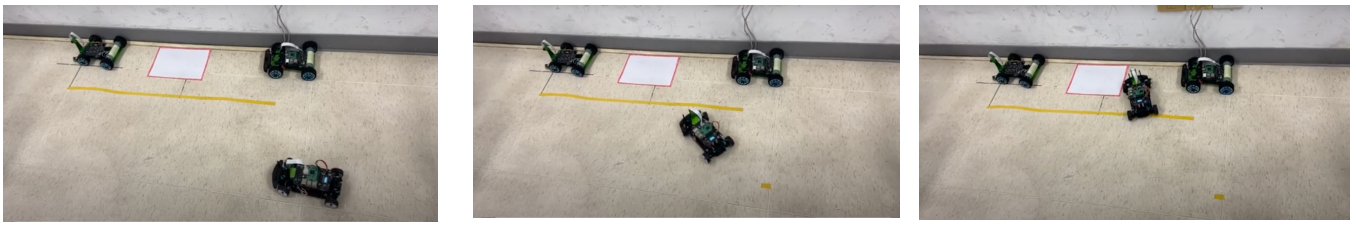
3. Results ｜訓練成果

a) Reverse in Parallel Parking - 10,000 dataset



Result - Auto Reverse in Parallel Parking [video link: https://youtu.be/RTGbjEBoq8I]

b) Forward in Parallel Parking - 31,000 dataset



Result - Auto Forward in Parallel Parking Result [video link: https://youtu.be/lZxFbpjcMp8]

4. Discussion

We personally train our auto parking by parking the car numerous times. At the end, the car was able to make initial movements, but unable to finish through, we think that the problem might be…

a) **Insufficient Training Data**

Analysis: Convolutional neural networks (CNN require substantial amounts of diverse training data to learn effectively. The lack of sufficient data could have limited the model's ability to generalize and adapt to different parking scenarios.

b) **Limitations in Recording Diverse Driving Data**

Issue with Reverse Parking: The training data only recorded when throttle is not zero. This approach neglects scenarios where the car needs to reverse, which is a critical aspect of parking. In the reverse mechanism in RC cars, the ESC must receive a reverse pulse, zero pulse, reverse pulse to start to go backwards.

c) **Training Approach: Detection of Parking Blocks vs. RL**

Parking Block Detection: Training the model to specifically detect 'parking blocks' and navigate towards them might have

been a more effective approach. This method could have allowed for more targeted learning and better performance in real-world parking scenarios.

Possible solution - Reinforcement Learning (RL):

RL could be a better training method for this application. By receiving feedback based on actions (parking successfully or failing), the system could learn more effectively through trial and error in a simulated environment.

V. Challenges & Difficulties｜困難與挑戰

A. BLE Connection Issue - <BluePy> Library conflicts with Raspberry Pi 4B

1. Issue:
   When we first set up the BLE connection over STM32 and Raspberry Pi 4B boards, it always displayed errors. By monitoring through `bluetoothctl` and `devices`, we found the device was actually connected, but disconnected right after.
2. Causes:
   After doing some online research, we found many people had similar problems. The reason seems to be that the Btle_helper.py of <BluePy> Library is too old, which would conflict with the new version of Raspberry Pi boards (4B).
3. Possible Solutions:
   Based on the solutions we found, people usually adopted 2 following solutions.
   a) *Library Substitution*:
      Since the `Btle_helper` is not available for update, we also tried using different Libraries such as <pygatt>. However, the dependency of the code and the library is highly correlated, so we ended up adopting the second solution.
   b) *Raspberry Pi version*:
      Since the code works correctly on an older version of Raspberry Pi — Raspberry Pi 3, and it is also compatible with all other devices, we switched to the R-Pi 3 board with original code instead.

B. Display & Connection Delay

1. Issue & Causes:
   Without careful address, sending huge amount of data such as real time images via BLE would create significant latency.
2. Solutions:
   Reducing framing Rate, image resolution, or displaying size are all effective ways to lower the latency.
   Here, we reduced the mage Resolution from 100% to 30% while setting the displaying size from fullscreen to 320*240.

C. Failed to train Auto Parking

1. Issue & Causes:
   The results of auto parking in both ***"reverse in parallel parking"*** and ***"forward in parallel parking"*** failed when a rapid turn was required, especially during backward movements. This can be attributed to the limitations of our sampling method. Since we only capture image data when the Pi-racer Pro is moving, the recorded images only show speed properties of either v+ or v-. However, in real scenarios, backing up requires speed changes from v+ to ***0***, then to v-. Due to the lack of a ***Speed = 0*** stage, our system fails to manage backing up movements, which are crucial in the process of auto parking.
2. Possible Improvement:
   Instead of relying on action imitation, using a “Red Parking Spot” as a key recognition feature or implementing reinforcement learning might enhance the performance in auto parking.

VI. Conclusion｜總結

Overall, we used a multitude of components and technologies to realize various functionalities such as ***real-time distance measurement, adaptive driving modes, danger detection, and visual display integration.*** These elements collectively formed a system for smart driving and parking capabilities.
The following summary table illustrates how we applied the concepts learned in our course “Embedding System” to this project.

Concepts	Methods	Application
Wireless communication	BLE	STM32 Board ↔ Raspberry Pi
	UUID	
	HTTP	Raspberry Pi Server ↔ Display Website
Hardware Controls	I2C	Steering and Throttling
	PWM Processing	Waveshare ↔ Motor
Web Socket	Flask	Local Web Socket
I/O Devices	Inputs	ToF Sensor, Magnenometer, joysticks
	Outputs	Screen Display, PiRacer Pro, OLED Display
Multi-tasking	multi-threading	Receive Real Time data, Process Real Time Scene
	Resource management	ThreadLocks
Real-time OS API	Polling	ToF Sensors
	Interrupting	Emergency Detected
	Event Handling	Event Queue

VII. References｜參考資料

1. VL53L0X library for Mbed: https://os.mbed.com/users/byq77/code/vl53l0x-mbed/
2. mbed-os-example-ble: https://github.com/ARMmbed/mbed-os-example-ble
3. Donkeycar: https://github.com/autorope/donkeycar/releases
4. PiRacer-Py: https://pypi.org/project/piracer-py/
5. Training: https://docs.donkeycar.com/guide/deep_learning/train_autopilot/
6. Special Shoutout to: Professor Wang, TA, Homework 4, Course materials