LAYMO: Autonomous RC Car — Methods

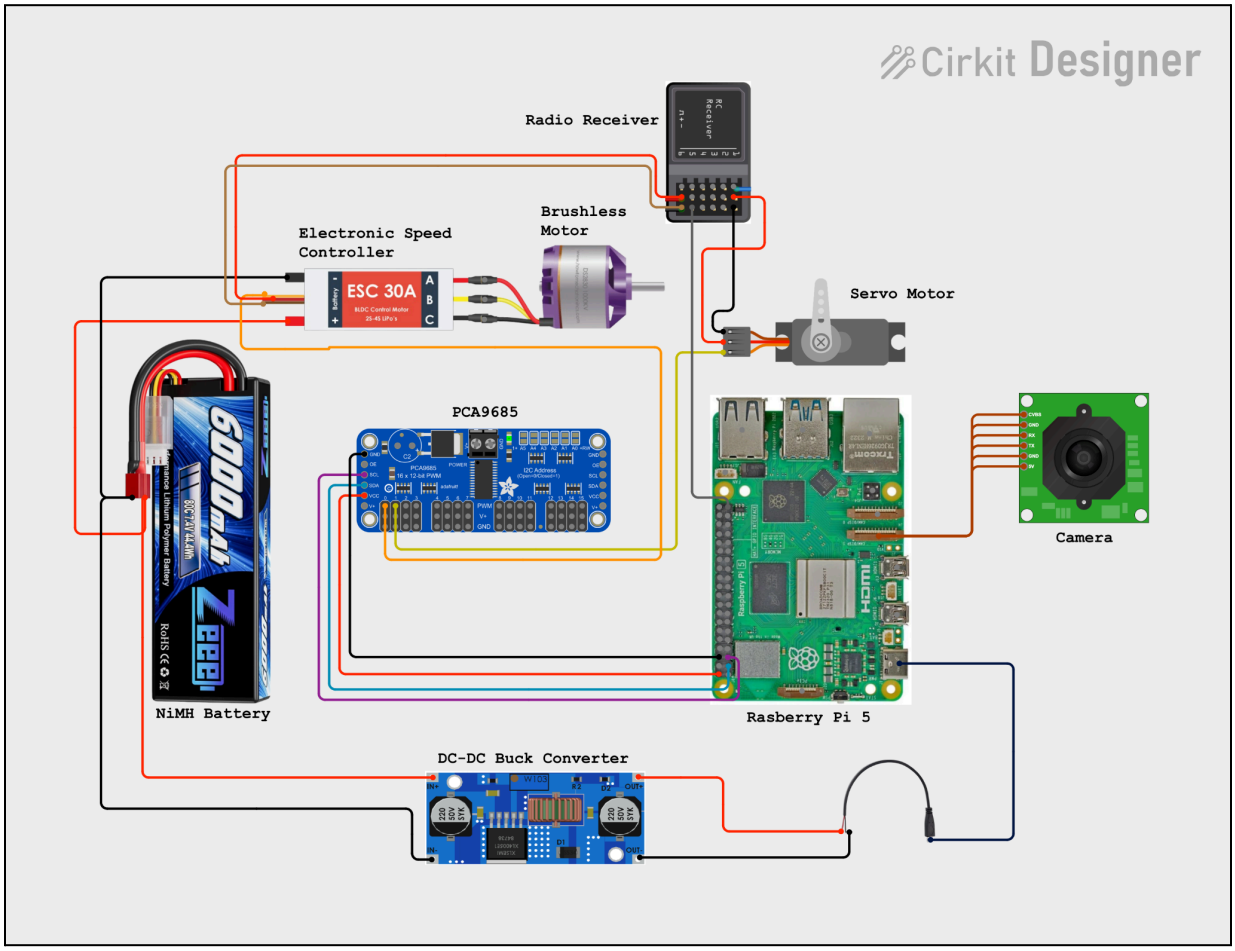
Achieving the goals of this project required solutions in both hardware and software. Rather than solve these in parallel, I focused on sequential development — first creating a hardware platform from which I could iterate and test software solutions.

The robot is built by modifying a hobby-grade remote control (RC) car. The car is equipped with a 3500kV brushless DC motor to provide power to the wheels, and a servo motor to control steering. The brushless motor is controlled via an electronic speed controller (ESC). Both the servo motor and the ESC receive command inputs in the form of pulse-width-modulation (PWM) signals [1]. The width of each pulse, called a duty cycle, encodes information to set the rotation angle for the servo or motor speed for the ESC. Before modifications, these signals were sent by a hand-held controller communicating wirelessly with an onboard RC receiver; I replace the signal source with a Raspberry Pi 5. The Raspberry Pi is capable of sending software-based PWM signals through its GPIO pins; however, due to context switching in the OS, the duty cycles in these signals are not guaranteed to be precise. This imprecision in the PWM signal can lead to unpredictable "jitter" in the servos, which is a very undesirable trait when attempting to follow a line. To overcome this issue, I use a PCA9685 PWM servo driver. The Raspberry Pi sends commands to the PCA9685 over an I2C bus to set desired PWM signals, and the servo driver handles generating and sending the required signals to the servo and ESC. Conveniently, the `adafruit_servokit` library in python provides higher-level abstractions on top of the PCA9685 that eliminates the need to manually calculate duty cycles [2].

This system allows for precise control of the car's steering and throttle through the Raspberry Pi; however, in order to mobilize the system, a method for powering the Raspberry Pi from on-board the RC car is needed. The Raspberry Pi requires a 5.1V power supply capable of delivering up to 5 Amps [3]. The car already utilizes an 8.4V 3000mAh Nickel Metal Hydride (NiMH) battery to provide power to the ESC. The ESC contains an internal Battery Eliminator Circuit (BEC) to step down this voltage to 6.2V, supplied to each pin on the receiver rail (as measured by a multimeter). This voltage is used to supply power to servo motors; however, it cannot be used for the Raspberry Pi. The voltage is too high, and while this can be accounted for, in testing, I found the BEC is unable to reliably supply sufficient current during spikes of throttle. Because of this, I split power directly from the battery and step down the 8.4V with an external DC-DC Buck Converter. This provides the recommended 5.1V into the Raspberry Pi via a USB-C connector. A wiring diagram of the entire system can be seen in Figure 1.

Figure 1.

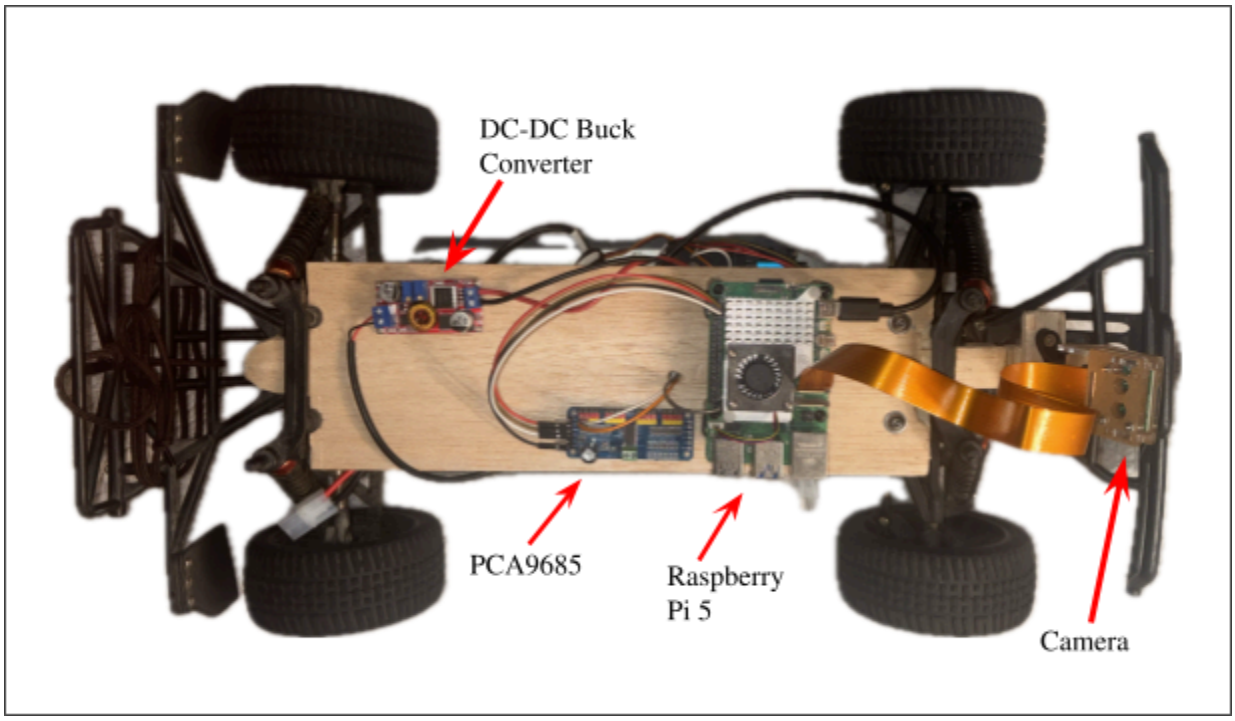
Wiring Diagram of Raspberry Pi Controlled RC Car



The final hardware component utilized is an Arducam 5MP OV5647 camera to capture the sensory input for the system. This particular module was chosen due to its affordability and ability to record relatively high frame rates of 90 frames per second [4]. It is attached straight at the front of the car and angled such that it captures the ground slightly ahead. Figure 2 shows the mounting location of the camera and an example of the camera's perspective can be seen in Figure 5.

Figure 2.

Top Down View of Modified Car



To mount these additional components (Raspberry Pi, PCA9685, DC-DC Buck Converter, Camera), I designed and built a modular platform from balsa wood that can easily be attached and detached from the RC car's frame (Figure 2).

With the hardware complete, I began to focus on the logic required to make our car autonomously detect and follow a blue line of tape on the ground. To simplify our problem, I keep the speed constant and focus on adjusting only the steering angle to keep the line centered in the car's camera frame. The high level logic of our main control loop is described in Figure 3. Our goal in the software architecture was to encapsulate the implementation details of each step such that our main control loop code is kept simple for readability and adaptability. Figure 4 shows the UML diagram for the project.

Figure 3.

Line Following Algorithm

Algorithm: Line Following Control Loop

```
(1)  set speed constant
(2)  while not at end of the line do
(3)      capture a frame from the camera
(4)      line center = detect average x-coordinate value of the line in the frame
(5)      error = line center - center of frame
(6)      set the steering to a value proportional to error
(7)  end while
(8)  set speed = 0
```

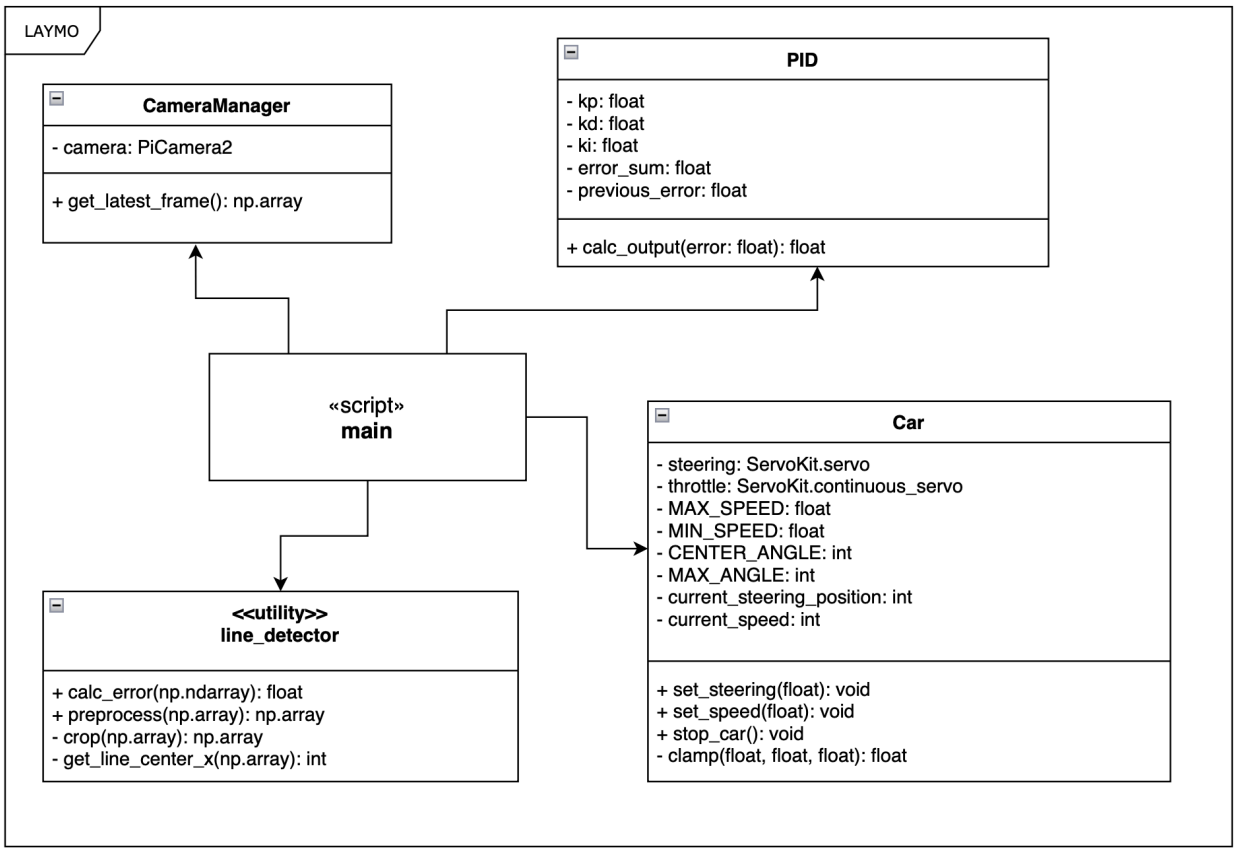
The `Car` class is a wrapper over the `adafruit_servokit` library which exposes an API such that values between -1 and 1 are passed to set steering and speed (invalid values are clamped to valid range). This allows users to control the car without having to set servo positions that would require knowledge about the physical limits of the car. This is important when dealing with software that interacts with hardware systems because, for example, servo motors can be easily burned by attempting to set their position beyond the car's physical steering limits.

The `CameraManager` class wraps the setup logic for the PiCamera2 library used to capture frames from the on-board camera [5]. Frames are captured as numpy arrays, which can easily be manipulated for computer vision. The program is able to capture frames at a rate of around 32 per second. I experimented with using a separate thread to continuously capture frames and store them in a buffer; however, no noticeable performance increase was found

during benchmarking tests. Future work will focus on achieving closer to the 90fps stated in the camera specifications [4].

Figure 4.

UML Diagram of Core Software System

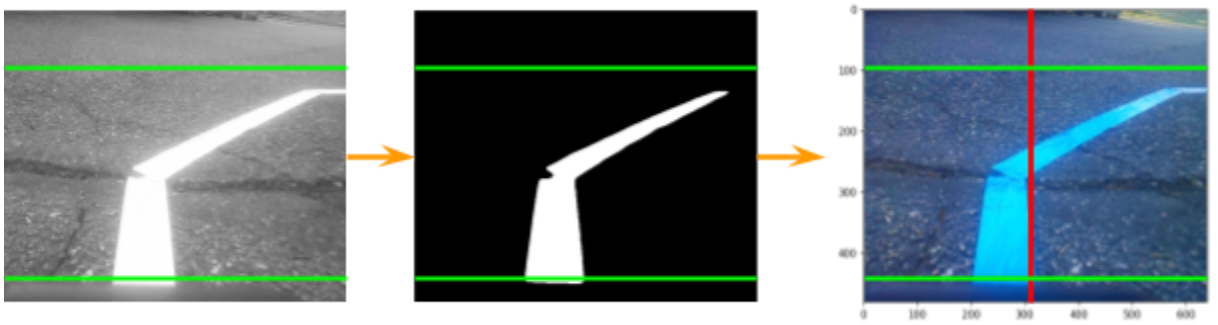


The `line_detector.py` utility script handles the logic of processing and detecting the center of a blue line in a frame. Blue was chosen due to its high contrast with the concrete surface used for the on-campus demo. In addition, because each pixel in an image is represented by red, green, and blue values (each in the range 0-255), by examining just the blue channel of the image, the blue line consistently has the highest values in the image. This allows for the computer vision pipeline to remain simple and fast for real time processing in a

compute-constrained environment. First, the image is cropped, keeping a horizontal band across the frame. The height and location of this band within the frame define the region of interest, which can be adjusted to change how far ahead the car is "looking". I then perform binarization on the blue channel of the cropped frame, setting pixel values above a specified intensity threshold to 255 (white) and those below to 0 (black). The center of the line is estimated as the average column index of white pixels in the binary frame. If the percentage of white pixels in the binary frame is below a predefined minimum (set as a hyperparameter), it is assumed that the line is not in the frame and `None` is returned. Figure 5 shows an example of the computer vision pipeline.

Figure 5.

Example of Image Processing Pipeline



Note. The horizontal green lines represent cropped regions of the image.

The example in Figure 5 also demonstrates an interesting implication of our averaging method — segments of the line further from the camera inherently carry less weight in the average due to linear perspective. In the context of our car, this means the position of the line far ahead can be taken into account, but it has a decreased impact on steering updates compared to the position of

the line close to the car. Scenarios can be imagined where this is advantageous; however, it also limits the ability to pre-emptively react to upcoming corners. A potential solution is to use contour detection algorithms on the binary image to place a bounding box around the entire line, and to use the center of the resulting bounding box as our estimate.

The `PID` class provides an implementation of proportional integral derivative controllers (PID). A PID controller is a type of closed loop system, where "the output [of the system] is compared to the desired output and any differences are used to modify the system's behavior or operation" [6]. In our case, the system is both the RC car and its state (position relative to the line, velocity, etc). The output of our system is the measured distance of the line from the center of the camera frame, where the desired output is 0. Therefore, our steering PID controller uses the difference between the center of the frame and the detected current position of the line (known as the error of the system) to produce an output. This output is used to set the steering angle of the car. The controller is nothing but a function, which takes the form

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)\,dt + K_d \cdot \frac{de}{dt} e(t),$$

where $e(t)$ is the error at time t, $u(t)$ is the output at time t, and the gains K_p, K_i, and K_d are hyperparameters that control the weights of the proportional, integral, and derivative terms, respectively. Our PID class allows for the creation of controller objects by initializing them with values for K_p, K_i, and K_d. In practice, the proportional-derivative controller (integral gains set to 0) was sufficient for most line paths. The output of the controller is found by passing in an error value to the controller object's `calc_output` function. This software design will make it

very easy to add dynamic speed control of the car by initializing a new controller object with its own weight parameter values.

The `controller.py` script utilizes abstractions provided by the helper classes to implement the line following control loop described in Figure 3 with several small additions. Because the slowest speed on the RC car is still often too fast, I pulse the throttle — alternating between the slowest speed and a speed of 0, based on the loop iteration. The script also implements several safety checks to stop the car in unexpected events. Each loop iteration checks the system logs for undervoltage warnings, and stops the car if one is encountered. Additionally, signal handlers stop the car gracefully if the program is terminated due to a fatal exception, a keyboard interrupt (^C), or SSH disconnect. These safety checks are critical because if the program stops running for any reason, the PCA9685 servo driver will continue sending the last PWM signals it was commanded to when the program stopped.

References

[1] Fernandez, A., & Dang, D. (2013). *1s and 0s revisited: The digital stream* (Chapter 9). In A. Fernandez & D. Dang (Eds.), *Getting started with the MSP430 Launchpad* (pp. 127–142). Newnes. https://doi.org/10.1016/B978-0-12-411588-0.00009-1

[2] Adafruit. (n.d.). *Adafruit CircuitPython ServoKit library*. Adafruit Learning System. https://learn.adafruit.com/adafruit-servokit-library-usage

[3] Raspberry Pi Ltd. (n.d.). *Raspberry Pi documentation: Power supply*. https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#power-supply

[4] Arducam. (n.d.). *Arducam OV5647 standard Raspberry Pi camera* (B0033). https://www.arducam.com/arducam-ov5647-standard-raspberry-pi-camera-b0033.html

[5] Raspberry Pi Ltd. (2025, March 20). *Picamera2 library manual* [PDF]. https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf

[6] Dwyer Omega. (n.d.). *What is a PID controller?* https://www.dwyeromega.com/en-us/resources/pid-controllers#:~:text=a%20PID%20Controller%3F,-What%20is%20a%20PID%20Controller%3F,variables%20in%20industrial%20control%20systems.