THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Team 10 – The JAMALinator

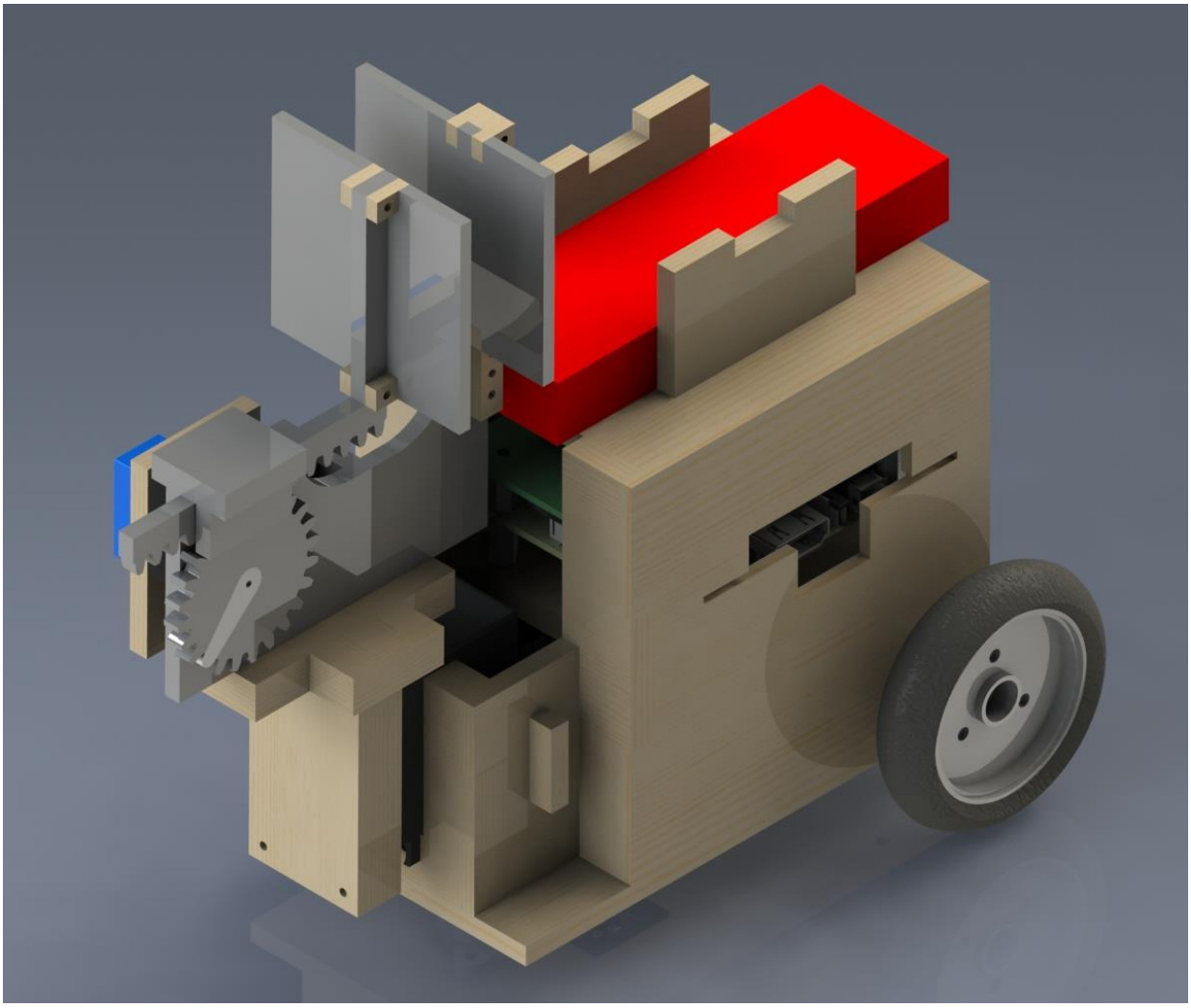

Fig 1. The JAMALinator SOLIDWORKS CAD render

Hammad Mohmand

Luca Cittadini

Matthew Christensen

Taro Bense

Table of Contents:

Introduction – The JAMALinator

The rapid growth of technology has transformed human-computer interaction and infrastructure. Mechatronics, electronics and electrical engineers are at the forefront of these changes, designing smart electromechanical systems to tackle practical, complex problems. Smart Autonomous Vehicles (SAVs) and the relevant “smart city” infrastructure is proposed to improve human quality of life, by reducing accident fatalities and traffic congestion as mentioned in [1]. One of the challenges engineers face is the design, development and safe integration of SAVs, this report describes how Team 10 navigated this challenge on a smaller scale.

In the ENGMT280/ENGEE281 course, we worked on a project to design, build, and program a Smart Autonomous Vehicle (SAV) meant to operate as a taxi in a model "smart city". This scaled-down city includes roads, intersections, and designated pick-up and drop-off spots, each marked with colour labels to simulate real-world challenges. The SAV must navigate the road network, find the correct route, and complete pick-up and drop-off tasks at specific locations.

Autonomous vehicles are a growing field with potential applications in delivery services and ride sharing. This project offers hands-on experience with AV technology, exploring how it might influence future urban transport. The mock city setup includes cameras and sensors, reflecting the infrastructure found in emerging smart cities. Beyond technical skills, the project also highlights the need for a mix of software, hardware, and communication systems, showing how different engineering disciplines work together to solve modern problems.

In our ENGMT280/ENGEE281 project, we integrated Vision Mātauranga to align the design of our Smart Autonomous Vehicle with Māori values. This approach helped us consider how the SAV might support community needs in a way that respects local contexts and promotes sustainability. Throughout the report, we outline the development process, from analysing the project brief to designing the chassis, placing sensors, laying out circuits, and developing control logic. This project not only showcases our technical skills but also highlights the challenges and potential of autonomous systems in real-world applications.

Design Development

SolidWorks Design

Designing the chassis and arm/claw mechanism is an incredibly important component in the overall design of the SAV, the design of every other component depends upon these two. There were four original designs, from the four designs, one design had the best chassis, another had the best arm/claw design, these were selected to go into the final design.

The chosen chassis design was the simplest, it was small, compact and simple in shape as it was essentially just a rectangular box. Another advantage of this chassis was the height from the ground, at about 10mm from the ground, this design was closer than the others which was beneficial for the reflectance sensor. The optimal distance for the reflectance sensor to work is between 10 – 40mm from the object it is sensing, the most optimal distance being 10mm. There were a couple of issues with it, however, primarily was the concern about structural integrity of the base as this was only 3mm thick.

The chosen arm/claw design was both simple and functional, this design incorporated a rack and pinion mechanism for the claw. While this was a little harder to design within SolidWorks, it meant that there would be no need to have an extra small servo to act as a spindle. With other claw designs there would have to be considerations concerning navigating the wall that the Lego man stands next to. With the rack and pinion design, all that the arm has to do is lower the claw to either side of the SAV. It also meant there was less wasted space within the enclosing area between the claws.

There were a lot of design iterations and changes due to either a realisation of a better design, or due to the merging of the chassis and claw that came from different designs. The arm was incredibly big compared to the size of the new chassis it was being fitted to. The chassis had to have a thicker base to support the weight of the claw, so this was changed from 3mm to 4.75mm. This increase in thickness of the base then meant that the caster ball was too low which then pushed up the rest of the SAV, this is not ideal, to fix this the top part of caster ball assembly was removed. This is not how it is intended to be used because now there is nothing keeping the caster ball and metal bars inside their enclosure, so three large holes were put into the base of the chassis, this means that the middle piece of the caster ball assembly can slot into these, to properly secure this, an additional 3mm MDF piece was added to slot into the walls with tabs, then allowing the caster assembly to be screwed into something.

As a result of implementing the large arm assembly, the whole chassis length had to be shortened, but because of the size of the claw assembly at the end of the arm, a portion of the front had to be shortened to about half the height of the rest of the chassis. The side walls at this height also had to be brought closer to the centre so that the mechanism can move fully. To add extra support for the large servo, there were cut-outs made in the short sidewalls for extra pieces of MDF to be slotted in, these were designed to assist with holding the large servo in place.

To hold the battery, the chassis was designed with a roof and two small walls to slot into the roof, this just contains the battery enough so that it doesn't slide off but is incredibly accessible. To hold the *Raspberry Pi*/PCB, an additional MDF piece was added to be slotted into the side walls, just above the motor box, this also turned out to be very good for the structural integrity of the entire chassis. A couple of last considerations were small cut-outs for the *Micro-USB* and *USB-A*, this allowed for easy access to these when initially testing hardware and software.

Because the arm was large in all directions, it also meant that after it's addition the SAV was now over the length limit, so the length had to be shortened. Due to the way that the claw swings around and the size of it, the end part of the chassis needed to be totally changed, so there is a portion at the end of the chassis that has shorter walls. To hold the arm/claw assembly, the front wall was made 9mm thick because there wasn't much to secure the large servo (the large servo is the connection between the chassis and the arm assembly).

PCB Design

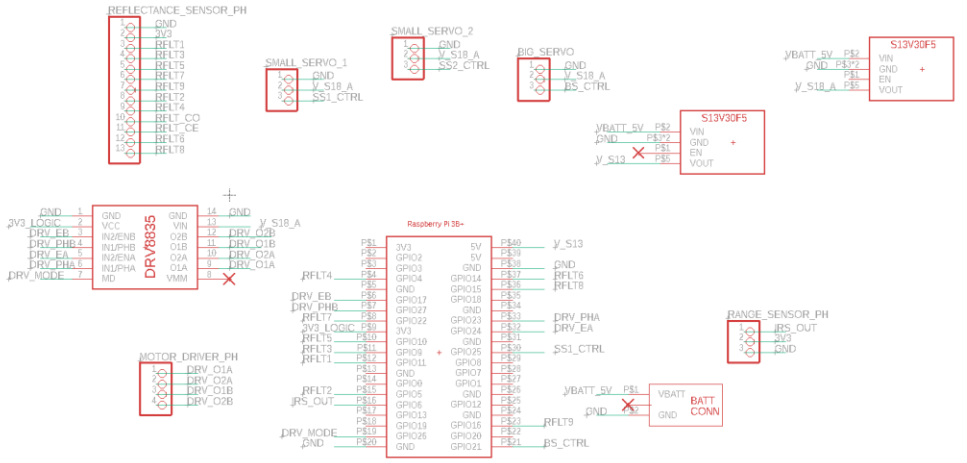

Fig 2. PCB schematic V1

Since both sensors have low voltage and low current needs, we connected the *Pololu Digital Distance Sensor* and the *QTRX-HD-09RC* Reflectance Sensor to the *Raspberry Pi* 3B+'s 3.3V port for the PCB design. The *QTRX-HD-09RC* runs within a range of 2.9V to 5.5V and uses maximum 0.17 mA per LED, making it appropriate for the Pi's 3.3V rail. Similarly, the Pololu Digital Distance Sensor has a current demand of 30 mA and typically runs between 3V and 5.5V. The *Raspberry Pi*'s 3.3V supply is sufficient to power these parts directly, so no further power regulation is needed.

For the servos, we connected them to a *Pololu 5V Step-Up/Step-Down Voltage Regulator S18V20F5* because servos require higher current, especially under load. The *FEETECH FS90 Micro Servo* operates at 4.8V to 6V and typically draws around 200-300 mA under normal operation, reaching up to 700-800mA during stall conditions. The *FEETECH Standard Servo FS5106B* has an operating voltage range of 4.8V to 6V, drawing approximately 800-900 mA under load, with even higher current draws up to 1.8 – 2.2A during stall conditions. The regulator ensures these servos receive stable power without overloading the *Raspberry Pi*'s limited current capacity.

The same 5V voltage regulator is connected to the motor driver to provide the current required for motor operation. To make sure the signal was interpreted correctly, we connected the 3.3V rail of the *Raspberry*

Pi to the logic side of the motor driver. The system makes sure that the motors receive enough current while the Pi controls them without interfering with power by isolating the power required for the motors from the logic control.

Ideally, we wanted to connect the *Raspberry Pi* to its own dedicated voltage regulator to ensure a consistent and stable power supply. This prevents power disruptions caused by the servos and motors, maintaining smooth operation across all components. By using separate power sources for the high-power and logic components, the system achieves reliable and efficient performance. However, when designing the PCB, the mini servo was connected to the same power supply as the *Raspberry Pi*, which is not ideal.

PCB Traces

As the voltage regulators and all components directly powered by them must carry higher current, the PCB's traces were made thicker. As thicker traces have less electrical resistance, they can carry higher currents more effectively without overheating or dropping voltage. The increased width reduces the chances of overheating and ensures that the power delivered to components such as servos, motors, and the motor driver is stable and sufficient to meet their high current demands.

On the other hand, parts that run at lower currents, like the sensors attached to the *Raspberry Pi*'s 3.3V rail, were constructed with smaller traces. These components draw very little current, so thick traces are not necessary. A decreased PCB area is achieved by using thinner traces for low-current paths, allowing a more compact and efficient design without compromising electrical performance.

PCB Board Size

The PCB was designed to be a similar size to the *Raspberry Pi*, making sure it takes up as little space as possible. Female headers were soldered to the PCB, so the GPIO pins of the Pi could connect directly to the PCB, allowing it to sit neatly on top. However, because the *Raspberry Pi* has tall USB ports, the PCB had to be carefully shaped to avoid them. This ensures the board fits perfectly without any interference. The result is a compact and efficient setup that integrates smoothly with the *Raspberry Pi*, with no obstructions.

Component Placement

For our PCB design, we implemented a mix of male and female headers to ensure versatility and ease of maintenance:

- **GPIO Pins**: We soldered a female header to the PCB, allowing for a direct connection with the male GPIO pins of the *Raspberry Pi.*
- **External Components (Servos and Sensors)**: Male headers were used for these connections. This setup allows external components to plug directly into the PCB, simplifying assembly and replacement.

- **Internal Components (Voltage Regulators and Motor Driver)**: We also used male headers here, with these components inserted into female headers. This modular approach allows us to easily replace or swap out parts if they become damaged or faulty, without needing to desolder them from the PCB

Software

Reflectance Sensor Interfacing

The *Pololu QTRX-HD-09RC* reflectance sensor is the sensor of choice for the line following system in the SAV. It works by charging a capacitor in the sensor channel circuit, then discharging the capacitor and measuring the discharge time. Depending on the discharge time, it will 'tell' whether the channel is detecting a dark/light or dull/reflective surface. The algorithm depicted in the flow chart in *Fig. X* explains how this would be achieved using a computer program.

The final program used in the system only implements this algorithm in python, with no extra features. During the testing phase, programs were developed to automatically range the sensing thresholds using the sensor values from that instant. The automatic ranging proved to be too time consuming to develop and pointless as it was difficult to calibrate the sensor autonomously, due to the limited space in the starting block as the SAV needs to move back and forth over the line to calibrate.

The program eventually outputs a 9-bit binary array, with 1 representing the light zones/the line and 0 representing black or the background of the track.

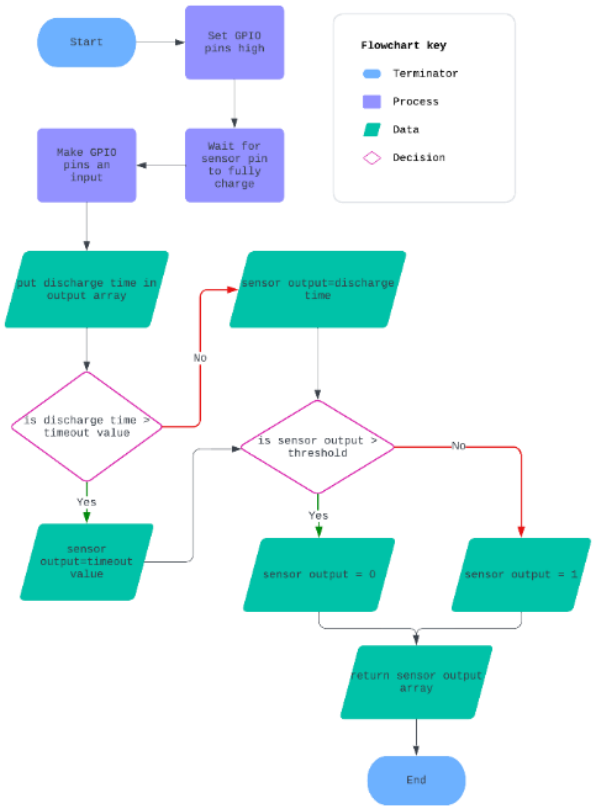

Fig. 3. Reflectance sensor algorithm

Line following logic

Line following logic is handled by the **LineFollower** class which was developed in Python to control various aspects of the SAV's movement including self-correction of deviation from a centre line, detection of forks in the track, and course navigation. The class is designed to interact with the GPIO pins of a *Raspberry Pi 3b+* using the RPi.GPIO module and is passed two GPIO pin lists, corresponding to those connected to the *QTRX-HD-09RC* reflectance sensor and the *DRV8835* motor driver, and a list of directions which should be passed as bool values.

Deviation from the marked centreline is determined by comparison of processed *QTRX-HD-09RC* output lists and lists containing pre-observed turn patterns. Edge cases are included to handle situations where the SAV has totally deviated from the track (reflectance sensor 'seeing' all black) or has reached the end of the track (reflectance sensor 'seeing' all white. The LineFollower class then corrects the position of the SAV on the track by calling the **Drive** class which control the SAV's motors.

The detection of forks is also handled by the LineFollower class. Three cases are considered which correspond to the approximate positions achieved by the SAV while on the fork. These include

- **Past the fork**: When the robot has passed through a fork and is on the other side driving straight, the middle three sensors may detect black, while the edge sensors may detect white, indicating the SAV is at a point where the track is diverging.
- **Boarding the fork**: As the robot nears a fork, the line appears wider. This is detected when more than five sensors detect white while the sensors on the extremes detect black, signalling the fork's approach.
- **Turning at the Fork**: While turning onto or off a fork, the central sensors detect white, while either one of the edge sensors may detect black, indicating the robot is in the process of navigating through the fork.

If any of the pre-determined bit patterns corresponding to the above cases is observed by the reflectance sensor, the SAV will recognise it is at a fork.

A third function of the LineFollower class is to handle the SAV's navigation of the course and index which turns remain in each trial run. A simple state machine which made use of the fork detection method was implemented to accomplish this task. Every time the SAV continues on from a fork, the LineFollower moves increments to the next turn direction in its internal turns list. After the second turn, the SAV will pause and wait to receive further turn instructions from the camera tower. Pictured below is a UML class diagram depicting the relationship between LineFollower and other classes implemented by the SAV

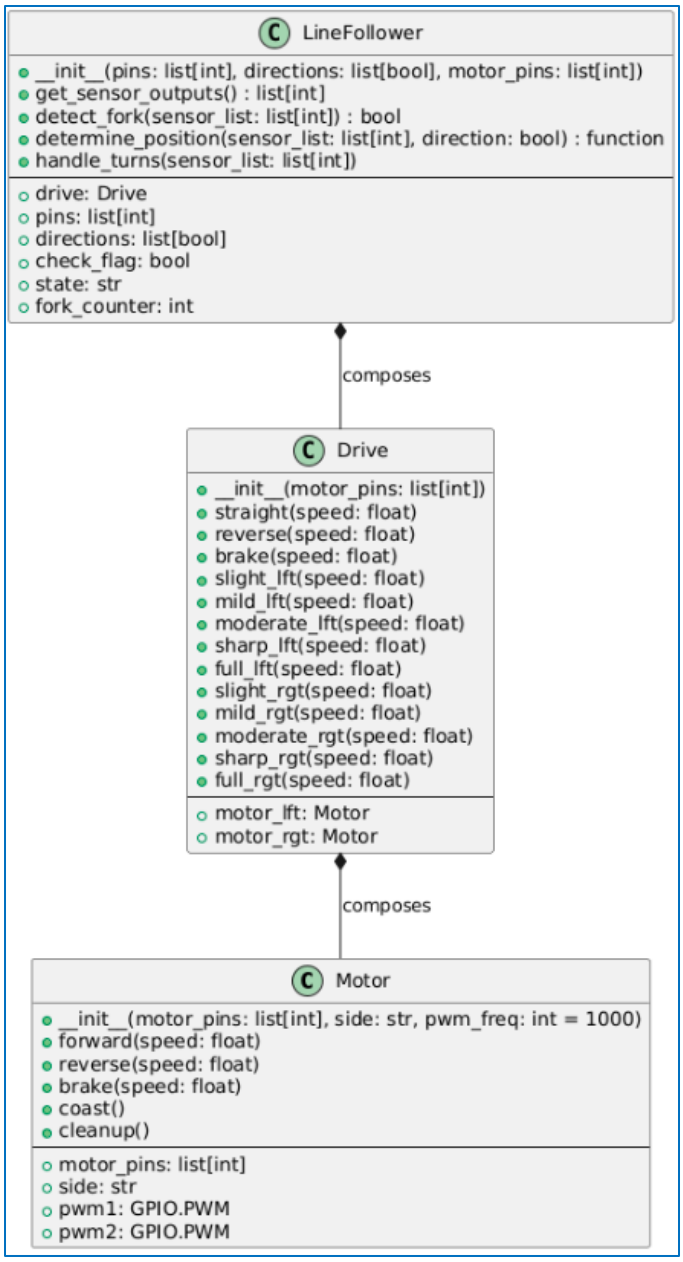

Fig. 4. UML class diagram of line follower system

Motor Control

The *Pololu DRV8835 Dual Motor Driver Carrier* was used to control the motors, it is a carrier board for the *DRV8835 dual H-bridge IC*. This carrier board makes it easy to control two different brushed DC motors with an embedded system. A simple digital-logic-like control scheme is used to control motor outputs, which can be seen in *Table X*.

The drive control python program was simple, functions were created for each operation and the corresponding pins were given the signals described in *Table X*. Controlling the motor itself was a bit

more involved with generating and stopping the PWM signals. This can be seen in the code snippet from *motor.py* provided in the Appendix.

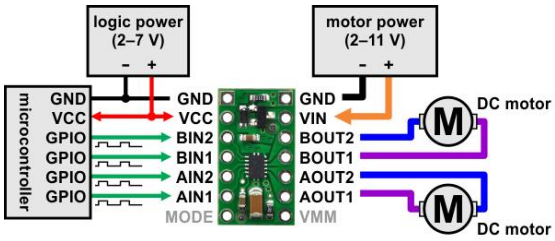

Fig. 5. Wiring diagram for carrier board to motors

TABLE I

Drive operations for Pololu DRV8835 Dual Motor Driver Carrier

xIN1	xIN2	xOUT1	xOUT2	Operation
0	0	Z	Z	Coast
PWM	0	PWM(H/Z)	PWM(L/Z)	Forward/coast at speed *PWM %*
0	PWM	PWM(L/Z)	PWM(H/Z)	Reverse/coast at speed *PWM %*
PWM	1	L	PWM(L/H)	Reverse/brake at speed *100% − PWM %*
1	PWM	PWM(L/H)	L	Forward/brake at speed *100% − PWM %*
1	1	L	L	Brake low

Motor Interfacing and the Drive Class

The control of the motors and gearbox is managed by the **Motor** and **Drive** classes. These classes were developed in Python to handle the control of DC motors connected to the *Raspberry Pi* through the *DRV8835* driver.

The **Motor** class is responsible for controlling individual motors on the SAV's *Tamiya 70168 Double Gearbox*. Motor direction and speed is managed by interfacing with a *DRV8835* driver via GPIO pins configured for digital PWM. Three parameters must be passed upon instantiation. These include a list of two GPIO pin numbers connected to the motor driver, a string ('left' or 'right') indicating which motor in the gearbox to be controlled, and PWM signal frequency (default value of 1kHz). It should be noted that **Motor** make's use of the **IN/IN** control interface mode on the *DRV8835*.

Individual motor behaviour is controlled by four functions within the Motor class: *forward(speed)*, *reverse(speed)*, *brake(speed)*, and *coast(speed)*. Note all the drive behaviour functions except *coast()* must passed be a parameter 'speed' to set the motor speed: A 0 passed would indicate 0% duty cycle while a 100 would indicate 100% duty cycle. Note this value only by a float between 0 and 100 (inclusive)

- The *forward*() function initiates the motor's forward motion by setting the GPIO pins corresponding to *IN1 and IN2* high and low respectively and adjusts the PWM duty cycle to regulate speed by sending a digital PWM through the *IN1* pin.

- The *reverse*() function reverses the motor's direction by inverting the *IN1* and *IN2* signals and modulating the PWM duty cycle to control the reverse speed by sending a digital PWM through the *IN2* pin.
- The *brake()* method initiates braking by setting both *IN1* and *IN2* pins high, then sends a digital PWM through *IN1* and *IN2* which determines the deceleration. A *speed* of 100 corresponds to a full brake, while a *speed* of 50 would affect a gentler stop
- The *coast*() function allows the motor to spin freely without applying power or braking (i.e. coast) by setting both GPIO pins low, effectively disconnecting the motor from the power supply.

An additional *cleanup()* function was included in the class to terminate PWM signals to the driver and call *GPIO.cleanup*(), which sets all used pins to inputs.

The **Drive** class builds on the Motor class by managing two independent motor instances which represent the left and right sides of a gearbox. Upon initialization, the Drive class instantiates two Motor objects, each configured with their respective GPIO pins and designated as "LEFT" or "RIGHT." Utilising the methods provided by the Motor class—such as *forward(), reverse(), brake()*, and c*oast()*—the Drive class synchronises the operation of both motors, enabling the robot to execute turns of varying degrees as well as the standard forward, brake, and reverse movements from Motor. This abstraction allows for high-level control commands to be issued without needing to manage individual motor operations directly.

Example Case: Executing a Slight Left Turn

When the SAV to performs a slight left turn, the Drive class implements functions from the Motor class to adjust the speed of each motor; reducing the speed of the left motor by 25% while maintaining the right motor at full speed. The speed differential between motors causes the SAV to turn gently towards the left, allowing for a directional change.

```
def slight_lft(self, speed):
    self.motor_lft.forward(0.75*speed)
    self.motor_rgt.forward(speed)
```

Fig. 6. `slight_lft()` method

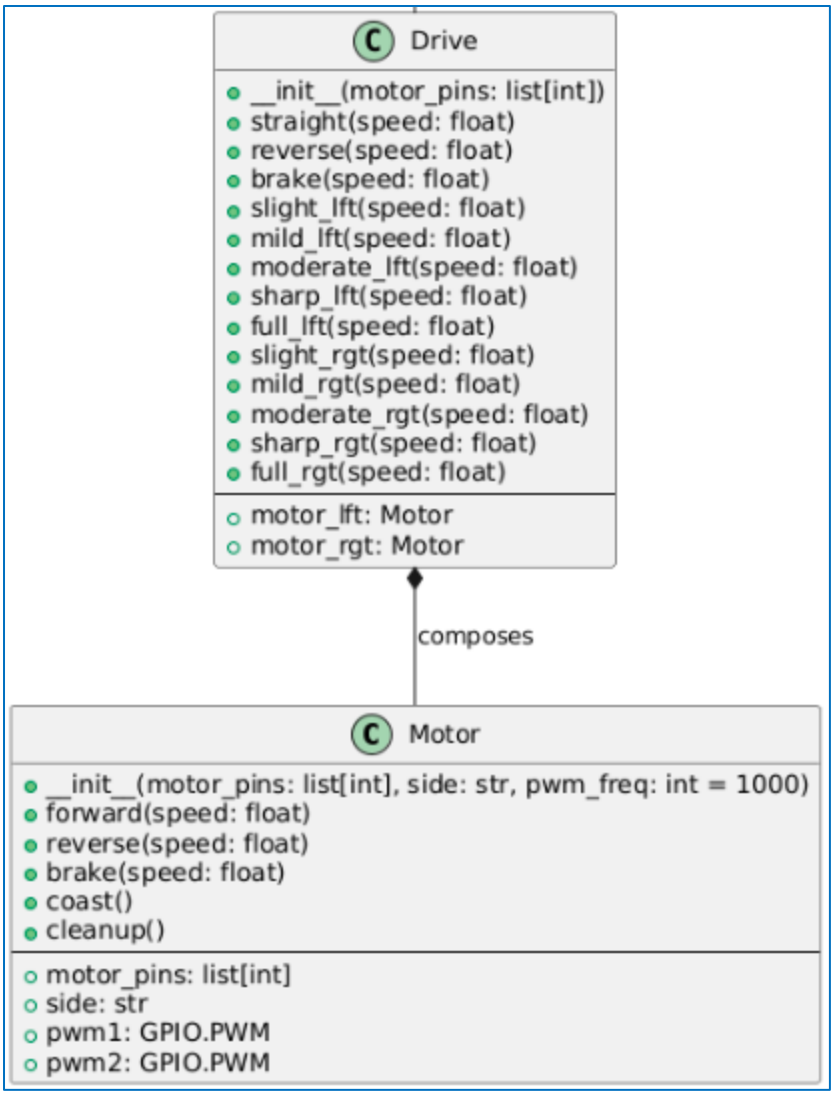

Fig. 7. UML diagram of Drive and Motor class relationship

Range Sensor

The *Pololu Digital Distance Sensor* was used for the object detection system. Interfacing with the sensor through code is simply checking whether the output from the sensor has gone low. When implemented into the system logic, it was discovered that the sensor was detecting the panel on the beginning of the fork branch. To fix this issue, it was decided that the resistor bridge across the configuration jumpers was to be removed to make the maximum sensed distance 5cm instead of the default 15cm. This made it so that the panel is sensed as late as possible

Servo Control

Two different servos were used for the pickup system for the SAV. The mini servo, used for the claw, is the *FEETECH FS90 Micro Servo* and the larger servo, used for the arm movement, is the *FEETECH Standard Servo FS5106B*. Both servos have a similar operating voltage range and pulse width ranges, this allowed for them to be controlled – along with the motors, using the same voltage regulator. Since only one actuator is controlled at a time, current limitations would not be an issue.

A single PWM signal is used to control a single servo, this can be observed in the code snippet in the Appendix. The duty cycle calculation is as follows, where PWR_{max} is the maximum value of the Pulse Width Range of the servo, PWR_{min} is the minimum value of the Pulse Width Range of the servo, α is the maximum operating angle of the servo, α_o is the desired angle and T is the period calculated from the inverse of the frequency of the PWM control signal:

$$\text{duty cycle } (\%) = \frac{100 \cdot PWR_{min}}{T} + \frac{10\alpha_o \cdot (PWR_{max} - PWR_{min})}{\alpha \cdot T} \quad \text{.....(1)}$$

Equation (1) was implemented as two functions for the two different servos, with simple input angles for the pickup motion.

Machine Vision

The machine vision program runs parallel to the SAV system code on a separate Raspberry Pi 3b+ that is connected to a camera in view of the entire track. The code uses the libraries time, picamera2, cv2, mqtt. The picamera2 library is used to access the camera to produce a live video stream of the model Smart City. The cv2 library is used for processing and interpreting the video stream: Each frame applies select filters and colour space conversions (BGR to HSV) to enable recognition of the red, green, blue and yellow pickup and drop-off points. Then, it converts the colour space to greyscale for effective edge detection. Boxes are appended to the frame where red, green, blue or yellow is detected in a minimum desired area. The program includes functionality to select and locate a desired colour within a frame as well as HSV threshold sliders which enable fine tuning of colour detection in the Smart City's varying light environment. Once the desired colours are selected and the boxes are drawn, the program finds the centroids to locate the pickup and drop-off points. It will then return either a two or four length long list of binary values encoding navigational data. This list is converted to a string and sent to the local network through a publish function in the mqtt library which will then be subscribed to by the SAV's Raspberry PI. Upon receipt of the string, the data is decoded by the SAV's system code and implemented in the navigation of the course.

System Logic

In our setup, we use an array of nine reflectance sensors to monitor the robot's position relative to the line. These sensors provide detailed input on the robot's alignment, allowing it to make precise adjustments. Each sensor measures reflectance values, indicating whether it is over the line or on the

surrounding surface. Based on the combination of inputs from all nine sensors, we determine whether the robot should continue straight, turn left, or turn right.

By using PWM signals to control the motors, we can adjust the robot's speed and direction with precision. This ensures that it remains on track and properly aligned with the line, even when quick directional changes are needed. The feedback from all nine sensors provides real-time data, which our system processes to make continuous adjustments, keeping the robot moving smoothly along its path.

Additionally, our system receives ongoing updates about pickup and drop-off locations. When a specific flag signal is detected, we have the robot execute actions like stopping, reversing, or activating the servos to pick up or drop off items. The servos can position items on either side, enabling the robot to interact flexibly with its environment. By integrating the data from the nine reflectance sensors, motor control via PWM, and live updates on tasks, the robot is capable of autonomous navigation and efficient task completion.

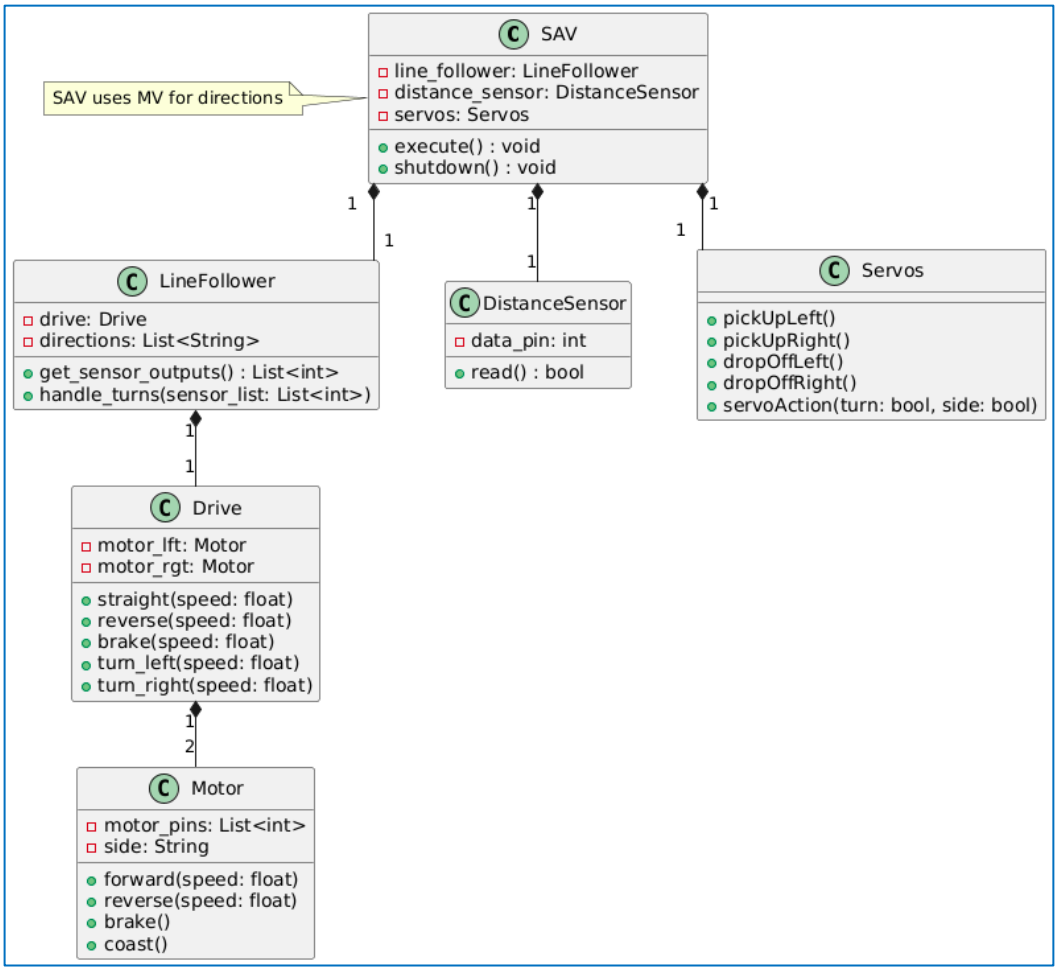

Fig. 8. UML diagram of System Logic Implementation (not all function included)

Results and Discussion

LineFollower Pseudocode Implementation:

- Get outputs from the *QTRX-HD-09RC* reflectance sensor using the *get_sensor_outputs()* method
- Update the internal state of the LineFollower object using the *handle_turns()* method
- Check if the SAV is at a fork in the track using the *detect_fork()* method
- Move the SAV in the required direction using *determine_position()* method

PCB Corrections:

Fig. 9. DRV8835 board layout

One problem we encountered when assembling the SAV involved the V_{IN} and V_{CC} pins on the DRV8835 motor driver. The blue trace pictured right delivered the same 5V to both pins, while the logic signals from the Pi only supplied a 3.3V logic high. We suspect that the discrepancy in voltages supplied to the logic side of the driver led to confusion in the system; where the driver did not know what value to interpret as a logic high. To correct this, the trace connecting the pins was removed, and a wire was soldered to a 3.3V output on the *Raspberry Pi* directly to the V_{CC} logic power supply connection pin.

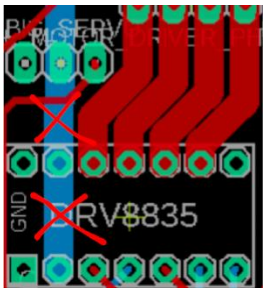

Fig. 10. Adjusted trace depiction

While attempting to resolve this issue, we erroneously scratched a second trace from the board. We intended to use the *V* pin to provide motor power, however as V_{IN} was not causing the issue we restored connection to V_{IN} shortly after. This removed all electrical signals to the motor power supply input. We re-established the connection by soldering a wire directly from the voltage regulator to the motor driver.

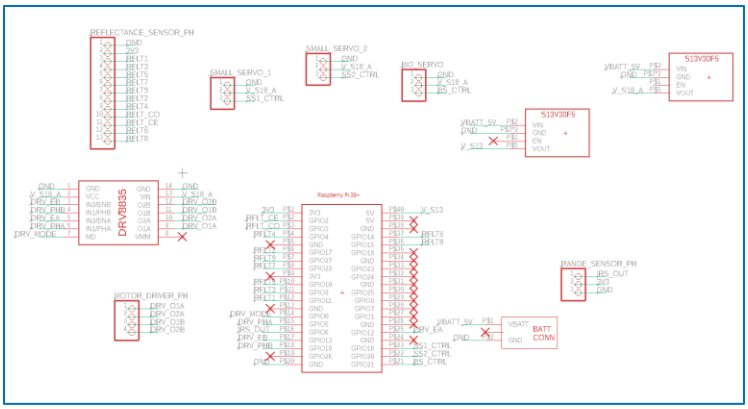

Fig. 11. New PCB schematic

Due to GPIO13 not being functional on our *Raspberry Pi*, the motor driver's phase and enable pins had to be moved around, which then resulted in movement for other GPIO pins, including most of the reflectance sensor pins and the small servo 1 control pin. Changes to the pin usage can be observed in Table I.

TABLE II

Raspberry Pi 3B+ GPIO Pin Usage

Name:	Initial BCM:	Revised BCM:
RFLT2	GPIO17	GPIO5
RFLT9	GPIO27	GPIO16
DRV_MODE	GPIO0	GPIO26
DRV_PHA	GPIO5	GPIO23
DRV_EB	GPIO13	GPIO17
DRV_PHB	GPIO19	GPIO27
DRV_EA	GPIO12	GPIO24
SS1_CTRL	GPIO16	GPIO25
SS2_CTRL	GPIO20	NOT USED

GPIO pin usage alterations

Software Troubleshooting

Throughout the development of the SAV, we faced numerous software challenges that required extensive troubleshooting. One significant issue arose due to our misunderstanding of how functions and dictionaries behave in Python. Initially, we attempted to correspond sensor patterns to relevant *Drive* functions using dictionaries, expecting that the functions would only be executed when their corresponding sensor patterns matched. Here is an example:

```
movement_patterns = {
    (0, 0, 0, 1, 1, 1, 0, 0, 0): self.drive.straight(20),     # 'Continue straight'
}
```

The issue was that we assumed the drive functions (e.g., *self.drive.straight_rgt()*) were only called when a specific sensor pattern was matched. However, in Python, when a function is included in a dictionary with parentheses included, the function is called immediately during the dictionary's creation, not when the condition is matched.

This led to unintended behaviour where all functions were executed as the dictionary was created, regardless of the current reflectance sensor output. To resolve this issue, we refactored the movement pattern dictionary into a series of if-else statements.

```
# Refactored into if-else statements
if fork_list == [0, 0, 0, 1, 1, 1, 0, 0, 0]:
    return self.drive.straight(10)  # Continue straight
```

By switching to if-else statements, we were able to individually check for a match between sensor outputs and predetermined sensor patterns and call the correct driving function, solving the problem.

Race Day Adjustments

- With the additional weight of the servo arm, the speed parameters passed to the *Drive* functions was increased, to compensate for reductions in speed due to additional mass on the SAV.
- We also realised our mass distribution was too far forward so the battery was secured further back relative to the SAV's chassis with double sided tape to achieve a more uniform mass distribution.
- Navigating forks proved difficult, especially where sharp turning was required. The *sleep()* function allowed the SAV time to process the turns, adjust back into position without running off the track, and hence make turns correctly
- Servo speed was also reduced to prevent failure at pick-up points. This was achieved by breaking up the servo movement with for loops to allow for a smoother pickup and drop off
- A second issue involving servos arose on race day. We found that that the servo arms over rotated during the pickup process. The solution to this problem involves backing off the servo arm in the other direction for a short duration to achieve the correct angle for pick up and drop off
- The reflectance sensor threshold was adjusted to indicate black surfaces at a discharge time of 850µs, decreased from an initial value of 1000µs which yielded erroneous results
- The range sensor

Race Day Results

- Navigation of the model Smart City was completed in a time of 35s
- Our SAV was successful in picking up and dropping of the model Smart City citizen

Conclusion

The JAMALinator project required the integration of mechatronics and electrical engineering principles to create a functional Smart Autonomous Vehicle (SAV) capable of navigating a model "smart city." Our team effectively designed, assembled, and programmed the SAV to perform course navigation, identification of pick-up and drop-off points, and transport a model ‘Smart City’ citizen. By combining various components, including the *QTRX-HD-09RC Reflectance Sensor, DRV8835 Dual Motor Driver Carrier, Pololu 5V Step-Up/Step-Down Voltage Regulator S18V20F5, Tamiya 70168 Double Gearbox Kit, FEETECH Standard Servo FS5106B, FEETECH FS90 Micro Servo,* and the RPi.GPIO Module, we were able to design an SAV to achieve our goals. Despite encountering many design challenges such as faulty components, unexpected design errors, and software bugs, we overcame them with diligent troubleshooting and perseverance. The integration of hardware and software components ensured the SAV's success and demonstrated our ability to combine a range of components to achieve an engineering design goal.

References

[1] J. M. Greenwald and A. Kornhauser, “It’s up to us: Policies to improve climate outcomes from automated vehicles,” *Energy Policy*, vol. 127, pp. 445–451, Apr. 2019, doi: https://doi.org/10.1016/j.enpol.2018.12.017.

Appendix

Parts list:

- Raspberry Pi 3B+
- QTRX-HD-09RC Reflectance Sensor
- Pololu Digital Distance Sensor
- DRV8835 Dual Motor Driver Carrier
- Pololu 5V Step-Up/Step-Down Voltage Regulator S18V20F5
- Tamiya 70168 Double Gearbox Kit
- FEETECH Standard Servo FS5106B
- FEETECH FS90 Micro Servo
- 7.4V 11400mAh LiPo Battery

Design resources:

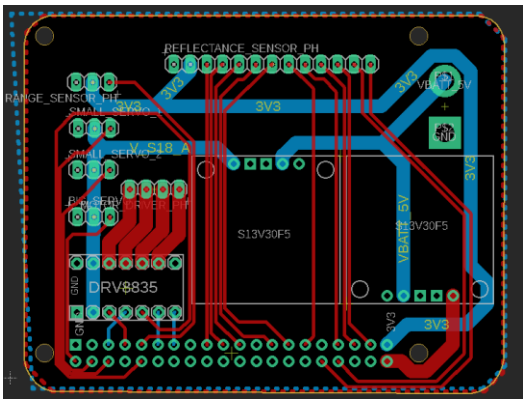

Fig. 12. PCB layout

Code snippets:

motor.py

```
class Motor:
    def __init__(self, motor_pins: list[int], mode = 1):
        GPIO.setmode(GPIO.BCM)

        # set pins as output
        GPIO.setup(motor_pins, GPIO.OUT)

        # setup pwm on the pins
        self.in_1_pwm = GPIO.PWM(motor_pins[0], 50) # starting freq @ 50 Hz
        self.in_2_pwm = GPIO.PWM(motor_pins[1], 50) # starting freq @ 50 Hz

        # start pwm signal with 0% duty cycle
        self.in_1_pwm.start(0.0)
        self.in_2_pwm.start(0.0)

    def forward(self, speed: float):
            self.in_1_pwm.stop()
            self.in_2_pwm.stop()
            # in_1 -> PWM = SPEED
            # in_2 -> 1

            # initial value at 0
            self.in_1_pwm.start(0.0)
            self.in_2_pwm.start(0.0)

            self.in_2_pwm.stop()

            GPIO.output(self.motor_pins[1], GPIO.HIGH)

            self.in_1_pwm.ChangeDutyCycle(speed)
```

line_follower.py

```python
class LineFollower:
    def __init__(self, pins, directions, motor_pins):
        # Ensure BCM mode is set for GPIO
        if GPIO.getmode() != GPIO.BCM:
            GPIO.setmode(GPIO.BCM)

        self.drive = Drive(motor_pins)  # Creating motor object
        self.pins = pins                # List of GPIO pins corresponding to reflectance sensor output lines
        self.directions = directions    # List of directions for the car to turn at forks
        self.check_flag = False         # Initialise flag bool to 0
        self.state = 'on line'          # Initialise LineFollower state to 'on line'
        self.fork_counter = 0           # Initialise fork counter iterative variable to 0

    def detect_fork(self, sensor_list):
        left_fork_sum = sum(sensor_list[:3])
        right_fork_sum = sum(sensor_list[-3:])
        centre_sum = sum(sensor_list[3:6])
        line_sum = sum(sensor_list)

        # Fork detection logic
        if (sensor_list[3] == 0 or sensor_list[4] == 0 or sensor_list[5] == 0) and left_fork_sum > 0 and right_fork_sum > 0:
            return True
        elif line_sum >= 6 and (sensor_list[0] == 0 or sensor_list[8] == 0):    # boarding fork, seeing line increase in width
            return True
        elif centre_sum == 3 and (sensor_list[0] == 1 ^ sensor_list[8] == 1):  # turning off fork, catching far edge of other path
            return True
        return False
```

servo.py

```python
def setSmallServo(angle) -> None:
    dutyCycle = 4.5 + (angle / 120) * 6
    pwmSmall.ChangeDutyCycle(dutyCycle)

def setLargeServo(angle) -> None:
    dutyCycle = 2.5 + (angle / 180) * 10
    pwmLarge.ChangeDutyCycle(dutyCycle)


def pickUpRight():
    setSmallServo(180)
    sleep(1)
    #setLargeServo(180)
    i = 90
    while (i != 180):
        i = i+2
        setLargeServo(i)
        sleep(0.025)
    setSmallServo(45)
    sleep(1)
    #setLargeServo(90)
    j = 180
    while (j != 90):
        j = j-2
        setLargeServo(j)
        sleep(0.025)
    sleep(1)
```

MachineVision.py

```
numBoxes = 0
boxes = []
for contour in contours: # Finds location of
    area = cv.contourArea(contour)
    if (area > 500) & (area < 3000):
        numBoxes = numBoxes + 1
        x, y, w, h = cv.boundingRect(contour)
        boxes.append(int(x + w/2))
        cv.rectangle(FinalImage, (x, y), (x + w, y + h), (0, 255, 0), 2)
        centroid_x = x + w/2
        centroid_y = y + h/2
        cv.putText(FinalImage, "x= " + str(centroid_x) + " y= " + str(centroid_y),
                   (x, y - 10), cv.FONT_HERSHEY_SIMPLEX, 0.25, (0, 255, 0), 1)
```