OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Feasibility Testing of Real-Time Object Segmentation and Recognition Models in Small Embedded Systems

Kaden Suderman

Electronic and Computer Engineering

School of Engineering

University of Galway

Supervisor & Co-Assessor

Dr. Adnan Elahi

Dr. Peter Corcoran

In partial fulfilment of the requirements for the degree of

M.E. Electronic & Computer Engineering

April 2024

Table of contents

Glossary

SAM – Segment Anything Model

YOLO – You Only Look Once

DL - Deep Learning

RPI – Raspberry Pi

AI - Artificial Intelligence

Declaration

I hereby declare that the work presented in this thesis has not been submitted for any other degree or professional qualification, and that it is the result of my own independent work.

Name: ___Kaden Suderman____

Date: _____April 2024_________

Abstract-- With the rapid growth of computer vision and artificial intelligence technologies, there is a growing emphasis on augmenting small embedded systems with deep learning capabilities. The aim of this project is to analyse and demonstrate the feasibility of integrating object segmentation and recognition models with these systems via connection to a remote deep learning capable computation platform. Through the deployment of traditional algorithms and modern inference methods, the study aims to enhance the efficiency and capabilities of small-scale systems in resource-constrained environments. Results indicate that while on-device machine learning offers reliability and no need for a network connection, it entails substantial computational overheads, greatly limiting the performance of viable models. Conversely, network-based inference methods are capable of fast inference speeds, usage of large cutting-edge models, and real-time connection speeds on reliable networks. The findings underscore the importance of tailoring system architectures to specific application requirements and optimizing machine learning models for embedded deployment. Ultimately, this research contributes to the broader embedded system field, offering insights into the practical implementation challenges and potential avenues for future advancements.

Chapter 1: Introduction

1.1 Project Introduction

The market sizes of both machine learning (ML) and embedded systems are steadily increasing, with an estimated market size of $258.6 Billion by 2032, but implementing ML in small systems with real-time speeds continues to be a challenge [1]. Furthermore, ML applications on these systems are often bottlenecked by small form-factors and computational abilities. This project aims to demonstrate the feasibility of using externally processed GPU-accelerated machine learning models on small embedded systems in real-time.

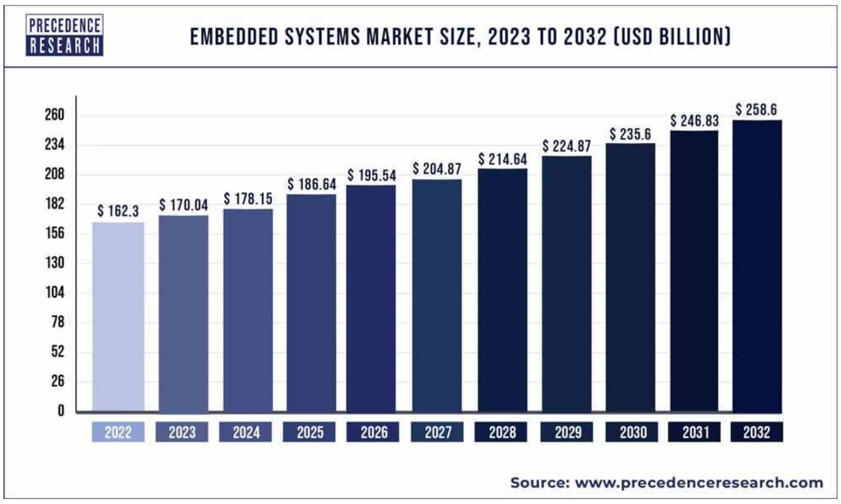


Fig 1.1 Embedded systems market size.

To investigate the feasibility of this task, multiple ML models with various sizes and abilities have been applied to a test embedded system, demonstrating the benefits and challenges of this approach. Furthermore, comparative testing has been performed, assessing the real-time abilities of small ML models performing computation on the embedded device itself and large deep learning models performing computation separately and communicating wirelessly with the embedded system. This project aims to demonstrate the feasibility of using these externally processed GPU-accelerated machine learning models on small embedded systems in real-time.

The chosen embedded system to apply these differing types of ML models for comparative testing is a smart vehicle system capable of automated driving capabilities. This chosen system is a self-driving concept but primarily exists to demonstrate the applicability of this concept to various network-connected applications which use similar embedded systems, i.e., security robots, IoT devices, robotics systems, educational devices, or any edge-computing applications. Vehicle movement, line detection, and tracking algorithms execute on the edge-device and handle the navigation. This software adds computational load to the embedded device, emulating typical systems which must perform computational tasks in addition to ML models. This demonstrates the cooperation between edge and remote computing and provides "safety" for insufficient network situations.

The development of a smart vehicle embedded system supplemented with advanced object detection and self-driving capabilities stems from the escalating advancements in computer vision and artificial intelligence technologies. With a heightened emphasis on automating and enhancing the driving experience, and budding possibilities of effectively combining embedded systems with external real-time computation systems, this project emerges as vital to utilizing current technologies fused with new techniques to examine and critique the results.

The primary goal is to construct a miniature smart-vehicle seamlessly integrated with an embedded computer, leveraging the abilities of deep-learning algorithms to increase system capabilities. In pursuit of this objective, the project amalgamates insights gleaned from studies of related self-driving technology and deep-learning model research. Synthesizing these insights will lead to the establishment of best practices in the field while introducing innovative advancements.

A key highlight of introducing GPU-accelerated ML processing to small systems is the enhanced decision-making capabilities they facilitate, providing the capacity to generalize identification for unknown objects. This critical attribute, also known as zero-shot learning, enables the system to make informed decisions even in the presence of unclassified elements, and is an impressive technique being incorporated into the largest cutting-edge models [2]. This feature, along with various other considerable benefits, provide ample motivation for investigating the feasibility of integrating large remote ML models into embedded systems.

The miniature smart-vehicle will traverse a specially designed model environment, tasked with the identification and tracking of diverse objects, including pedestrians and lanes. When complete, the system will navigate a test environment to verify the capabilities and integration of self-driving vehicle software and hardware with advanced and cutting-edge approaches.

Through this contextual backdrop, the project not only aligns with the evolving landscape of driving automation but also seeks to push the boundaries of self-driving technology, promising a transformative paradigm in autonomous navigation.

1.2 Goals & Objectives

For this project, a comprehensive set of objectives has been outlined which collectively establishes the form and function of the proposed miniature smart-vehicle embedded system enhanced via the use of GPU-accelerated deep learning models. The outlined objectives encompass the following key aspects.

1.2.1 Research and Implementation Strategy

Extensive research into existing small-scale smart vehicle systems and deep-learning models guides the selection of optimal software and hardware components. This strategic approach ensures the utilization of cutting-edge technology tailored for this specific implementation. This objective is completed primarily through a literature review which analyses various available research papers and information from similar and related projects.

1.2.2 Smart Vehicle Embedded System

The next objective is the development of a miniature smart vehicle which leverages the computational power of a Raspberry Pi to compute simple algorithms and simulate a typical embedded system's computational load. This system will serve as the foundational platform for integrating deep learning models and ultimately provide a foundational system for the addition of the enhanced functionality and capabilities of ML models.

1.2.3 Remote Computer-Vision Platform

This piece of the development phase involves the creation of a sophisticated computer-vision platform. Operating separately from the smart vehicle embedded system, this platform communicates wirelessly with the device via a Wi-Fi network connection. Utilizing a webcam integrated into the vehicle, real-time image frames stream to external computer-vision models, are processed, and any relevant information is returned. This acts as a supplement to the on-board processing equipment and provides advanced computation with minimal latency. Multiple deep-learning models are used in this platform to assess their relative abilities and performance metrics.

1.2.4 Construction of a Demonstrative Model

The testing aspect of this project relies primarily on the construction of a small-scale demonstrational model. This model serves as a test area where the vehicle can autonomously navigate basic conditions which utilize both the on-board computation and external processing. It consists of a roadway for the vehicle to follow, along with multiple objects which must be detected by the models at an appropriate range.

1.2.5 Advanced Deep-Learning Capabilities

Beyond only operating the chosen models externally, further advancements are added to the system. A system for receiving feedback from the models is used, likening the overall simulated application as closely as possible to real-world environments. This feedback exists on the embedded system and consists of retrieving the modified video frames to be applied to the vehicle’s environment.

1.2.6 System Testing and Validation

Various testing and validation procedures within the miniature model environment assess the system's functionality. These tests ensure the reliability and capabilities of the entire system’s operations under diverse scenarios with the goal being to maintain real-time functionality. Comparison of the different deep-learning models is performed throughout this objective with standard tests performed between all models and comparative measurements of their capabilities taken.

1.2.7 Demonstration of Completed System

Ultimately, the project culminates in a fully functional self-driving vehicle. Integrated with an embedded computer, this vehicle proficiently navigates the model environment. Real-time video is streamed from an integrated webcam to an external system for computation and the results returned and received on the embedded system. The demonstration showcases the successful fusion of computer vision and object recognition technologies in enabling autonomous navigation.

1.3 Conclusions

This project represents a significant leap forward in driving automation research, leveraging both established best practices and cutting-edge deep learning algorithms. By combining computer vision and artificial intelligence, it aims not only to meet its defined objectives but also to lay the groundwork for broader advancements in incorporating machine learning models remotely with embedded systems. The successful integration of a miniature smart vehicle with an embedded computer not only signifies a breakthrough in autonomous navigation but also demonstrates the practicality of deploying large ML models in compact systems. Drawing from extensive research in self-driving technology, this project establishes and refines best practices while introducing novel approaches to enhance object detection, decision-making adaptability, and software-hardware integration. These achievements not only advance the fields of computer vision and object recognition but also pave the way for future innovations, providing new insights and understanding in the realm of large deep learning model assisted embedded systems.

Chapter 2: Literature Review

2.1 Introduction

The integration of cutting-edge technologies and established methodologies within the research of miniature self-driving vehicles and cutting-edge deep learning models is central to the advancement of automated driving systems and any embedded systems benefitting from higher level computation. Rooted in a comprehensive literature review, this report section explores foundational techniques and technologies important to the related development steps. Emphasizing the critical role of microcontrollers, particularly the Raspberry Pi, this introduction delves into the significance of sensors, notably camera modules and wireless connection equipment, in enabling efficient object detection and navigation functionalities. Furthermore, it highlights the algorithmic techniques primarily used in the software design, with a focus on edge and line detection approaches as used in lane detection systems. Additionally, usage of the external machine learning models is explored with analysis on the potential for integrating cutting-edge object detection and segmentation, albeit with challenges for implementation. Overall, this comprehensive analysis advocates for the strategic combination of emerging technologies and proven methodologies in advancing research on self-driving and similar embedded systems.

2.2 Critical Evaluation of Hardware Design

Analysis begins with investigation into the array of fundamental hardware component choices made throughout related research. These components play a pivotal role in enabling smart vehicle capabilities and must be carefully assessed prior to implementation.

2.2.1 Microcontrollers

As the foundational piece of hardware within this project, microcontrollers are compact, yet powerful, processing units which perform calculations, analysis, and generally act as the brains of the smart vehicle. Through the management of the various integrated hardware components, they provide the capabilities for sensor data processing, necessary algorithms for navigation and object detection, and control of motor functions. They are, in essence, responsible for managing and executing the various instructions which govern vehicle movement and overall behavior, making an educated choice highly important.

When approaching the choice of microcontroller for this application, the key factors for consideration are performance requirements, energy efficiency, processing power, cost, and compatibility. To conduct a review of the choices made in other similar applications, papers and publications were broken down and analysed.

Performance demands specific to smart vehicles, like real-time processing or sensor data handling, are the first critical criteria to be met. Research by Ansari et. al. (2015) showcases a 2-sensor system implemented on a Raspberry Pi Model B. A common choice throughout literature, this model is a grounded choice due to its decent processing capabilities of a quad-core ARM processor, which can strongly handle sensor data processing and algorithms. It is also highly cost effective, has multiple connectivity options (Wi-Fi, Bluetooth, etc.), and is backed by extensive community resources. In similar research conducted by Shahane et. al. (2022), a Raspberry Pi (model unknown) is used for primary computation and data processing but is supplemented via an Arduino Uno. In this implementation, the Uno handles the car movements and communications to the motor systems. This solution reduces the workload on the Raspberry Pi and enables more processing power to be contributed to algorithms and interpreting sensor data. This is a promising option; however, it is held back in some respects as more components means a higher power draw, and no guarantee of the simplicity of integrating another complex processing unit into the device ecosystem.

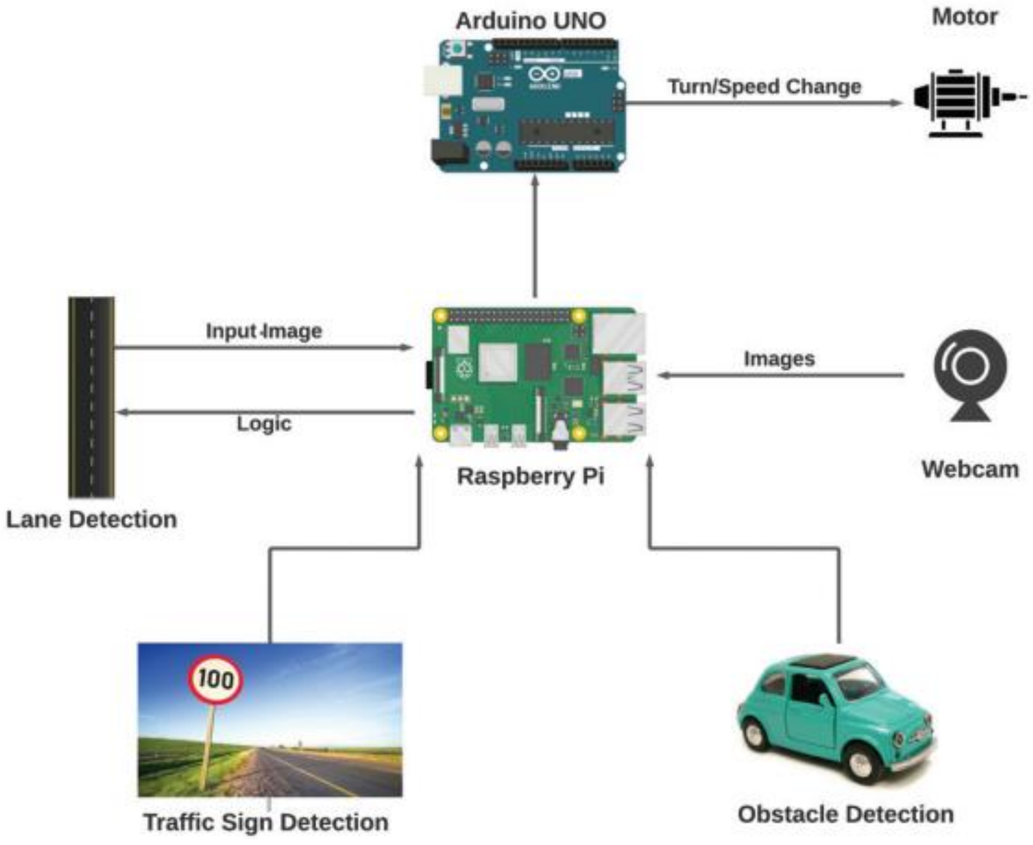


Fig. 2.1 multi-processor system implementation suggested by Shahane et. al. (2022).

Further papers, including another similar research implementation by Mohan et. al. (2019), continue to emphasize the choice of Raspberry Pi 4b for this and similar designs. Due to its combination of processing power, versatility, and affordability, the Raspberry Pi emerges as the ideal choice. Further, considering its ubiquitous adoption in similar related research, it stands out as an obvious choice.

2.2.2 Sensors

By sampling and providing crucial data and information about car surroundings, sensors act as a self-driving vehicle system's sensory organ. These sensory components range from ultrasonic and LiDAR (Light Detection and Ranging) sensors to cameras and many more options, which all collect real-time information and environmental data to be sent to and processed by integrated or external software. Primarily used within this research's scope for object detection and navigation, the synergy between all chosen sensors is paramount to enabling autonomous functionality. Critical analysis of this choice is done to ensure components picked are up to the latest standards and follow trends in technological advancement.

Throughout the analysis of research into this area, a common theme was identified. That is, that the most important sensor component in creating a smart-car system is the presence of a camera module. As can be seen in work by Ansari et. al. (2015), processing of road identification is performed exclusively through the data received from the camera module. This approach is supported by others, such as in similar research conducted by Shahane et. al. (2022), and Dhavalikar et. al. (2018), showcasing a clear necessity for including a camera module in the system. As also denoted by Ansari et. al. (2015), a "Pi Camera", or CSI (Camera Serial Interface) Camera, is a common and recommended choice, due to its ease of integration with Raspberry Pi's. Other choices of camera include webcams or standalone cameras. With these considerations, both a webcam and CSI Camera were picked for the implementation. The webcam would be used as the primary source of transmitted video, and the CSI camera acting as the input for processing required for line detection.

2.2.3 Other Sensors

It is important to consider other sensors to be implemented in coordination with a camera module. One prominent choice is an ultrasonic sensor, able to provide proximity information using high-frequency sound waves, useful for object detection. As highlighted in the study, "Raspberry Pi Based Autonomous Car", by Giripunje et. al. (2019), ultrasonic sensors excel in measuring distance of nearby objects. They are cost effective and highly accurate in close range applications, however, utilizing sensors in conjunction with an ultrasonic can alleviate weak points, such as their short-range sensing and inability to accurately receive reflections from objects which deflect soundwaves away from the sensor.

Assessing other potential sensors, a paper by Gazis et. al. (2021) makes considerable contributions to the information on sensor analysis and functionality. Radio detection sensors, suitable for medium and long-range radar applications, can replace or supplement ultrasonic sensors. However, these are, problematically, not suitable for small applications but suited better for full-sized vehicles. Another supplemental option, LiDAR sensors, are useful for high-speed collection of 3D spatial information. These sensors require high sensitivity detectors but excel in constructing models of the surrounding space. Though, considering the cost of these systems ranges from eight to ten thousand dollars, they aren't a viable option. This ultimately leaves the selection of possible sensor configurations highly limited, with cameras and ultrasonic sensors being the primary candidates. Therefore, it can be concluded that the best possible option is a camera with the potential addition of an ultrasonic sensor.

2.3 Critical Evaluation of Software Design

Next, it is important to critically assess software design within the context of autonomous vehicle systems and machine learning platforms. Drawing from related research performed on miniature self-driving vehicles, as well as real-world software implementations of deep learning platforms, the software architectures and algorithms assessed will be essential in deciding software factors best suited to the final product.

2.3.1 Line Detection

Line detection, the primary means by which the vehicle moves, aligns the vehicle with the correct line on the road it is travelling on and guides its movements. Seemingly simple, many techniques exist for this process. Focusing on research within the context of this project, only systems which utilize data from the CSI camera module, or those with similar specifications, have been included.

Offering a comprehensive and efficient solution in their research on designing an autonomous car using Raspberry Pi, Ansari et. al. (2015) describes a line detection algorithm with a focus on utilization of a fusion technique, where rather than lanes being identified and tracked, the road size and shape is tracked, and lines painted on according to specific parameters. This integrated approach enhances adaptability and offers improved line detection across diverse road conditions.

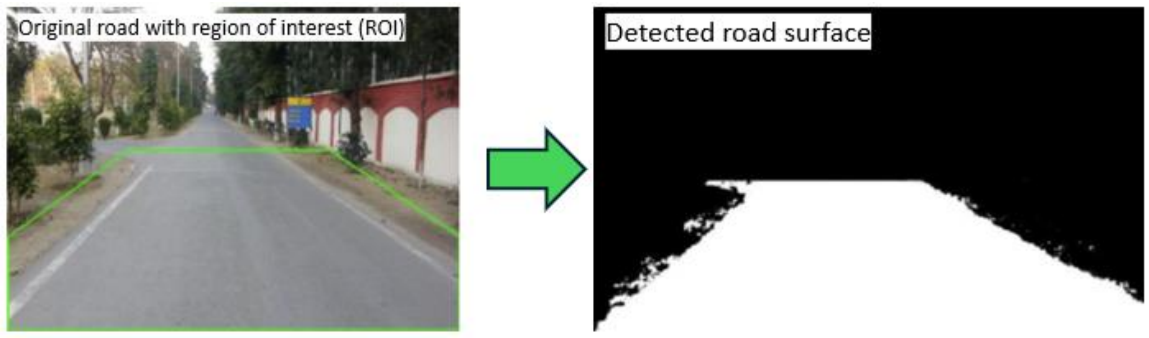


Fig. 2.2 Road size and shape tracking in ROI, Ansari et. al. (2015).

Similarly, other algorithms, such as in research by Shahane et. al. (2022), utilize the same techniques to achieve results, such as the Hough Transform for line generation and techniques such as Canny edge detection for road identification. These techniques are seen throughout the literature and present an approach worthy of serious consideration in the final implementation.

2.3.2 Obstacle Avoidance & Navigation

A key feature of a self-driving vehicle is to identify obstacles present in the movement plane. Accomplishing this task will be completed through one of two main avenues:

A) Utilize hardware techniques such as ultrasonic sensors to detect objects, then on-system algorithms to make the appropriate decisions.

B) Utilize machine learning via an off-device deep-learning platform to detect obstructions and perform the corresponding reactions.

When choosing the methodologies and algorithms to use, critical examination regarding the limitations of the system cannot be left out. These consist of sensor accuracy, computational abilities, and environmental conditions, and are the primary constraints for analysis in the following sections.

In research conducted by Day, et. al. (2019), a Raspberry Pi equipped with ultrasonic sensors detected pedestrians and obstacles through Histogram of Oriented Gradients algorithm and Haar-classifier algorithms. Results were good, implying the techniques used are a viable option. One discussed setback is the outperformance of both algorithms by deep learning methods, and points to the potential avenue of utilizing off-microcontroller processing to complete heavier tasks. These methods will be further examined before implementation but may provide a highly accurate approach.

2.4 Deep Learning Platform

This report section delves into the integration of a deep learning platform to enhance the capabilities of an embedded system from a remote standpoint, presenting insights garnered from existing literature. Leveraging the power of deep learning algorithms, this approach aims to bolster the functionality and performance of the self-driving car or any embedded system, paving the way for novel applications and advancements. The existing body of literature has been explored to provide a comprehensive overview of the methodologies, challenges, and potential benefits associated with employing remotely enhanced embedded systems.

2.4.1 Limitations and Requirements of Chosen Application

Looking first at research related to utilizing remote DNNs (deep neural networks) to improve the computational abilities of embedded systems, the specific challenges and requirements for the chosen application will be considered. Batzolis, et. al. (2023) in research on the limitations and solutions to implementing machine learning in embedded systems, states that:

> "To effectively integrate machine learning into these systems, a number of issues must be resolved, including resource limitations, data availability and connectivity, real-time demands, robustness and dependability, safety, and security concerns."

Taking these points into mind, the chosen methodology and techniques must consider the available resources on the Raspberry Pi, its connectivity abilities, and the real-time demands of the chosen application.

2.4.2 Meta Segment Anything Model (SAM)

Finally, an array of machine learning models was analysed for application on the remote deep learning platform. Looking first at the Meta Segment Anything Model (SAM), this deep learning model presents an innovative approach to segmenting diverse visual elements within an image. With this model, object detection and segmentation could be significantly improved from what is possible on-device, as all objects in a scene are segmented simultaneously, providing the system an elevated comprehension of scene dynamics. The Meta SAM model is also the largest segmentation dataset to date, with "... over 1 billion masks on privacy respecting images," Kirillov, et. al. (2023) and could enable finer and more precise recognition of pedestrians, and other objects, refining its decision-making abilities. This dataset, titled SA-1B (Segment Anything 1 billion), has 11 million high quality images with 1.1 billion segmentation masks and is a new dataset introduced in Meta's SAM research paper [16]. It does not contain class labels but solely consists of mask annotations of every object in the image scene. This results in processed frames which contain high levels of object detail and provide exemplary data for use in computer vision systems.

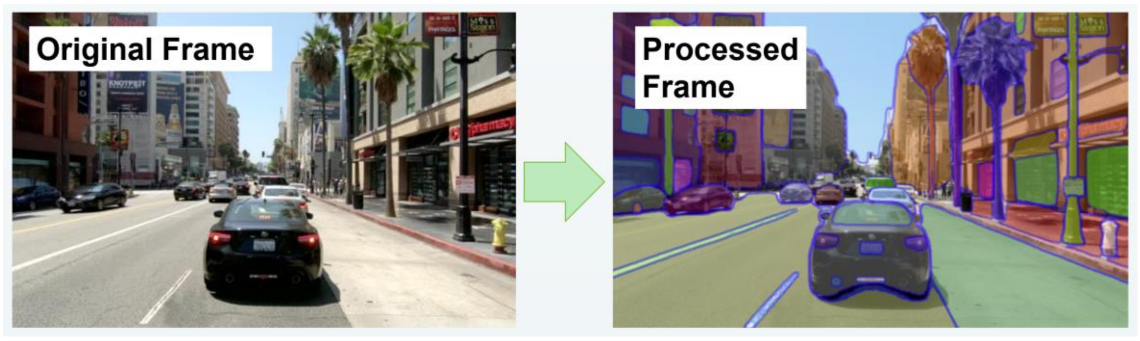


Fig. 2.3 Showcase of the abilities of Meta's segment anything model.

Further, the model's ability to generalize for objects in a scene, i.e., determine a plausible class for an object while only having been trained on similar images, is impressive. This presents an opportunity for a model to increase its understanding of input sensor data and have a more comprehensive understanding of objects. This technique, also referred to as zero-shot learning, is a capability of the most cutting-edge computer vision systems in place and pushes the limits of what has been done in this specific field. Another version of this model, "FastSAM", has been documented by Zhao, et. al. (2023) and modifies the model from using a Transformer architecture to using a Convolutional Neural Network architecture, providing up to 50 times higher run-time speeds. This is promising for applications requiring fast speeds, as this can be applied effectively to real-time applications.

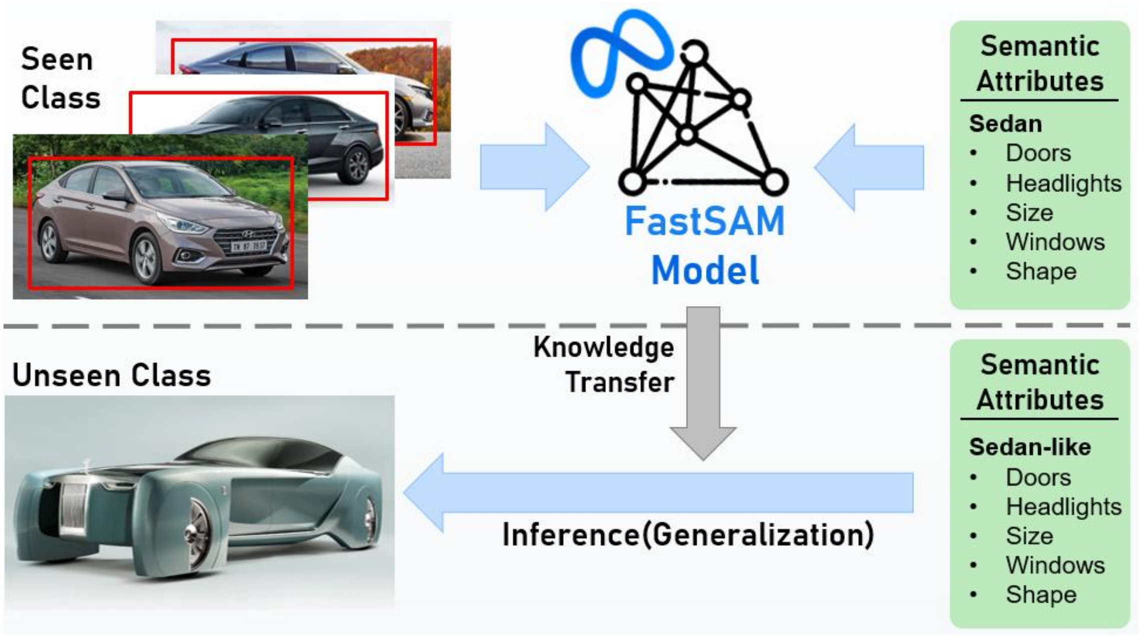
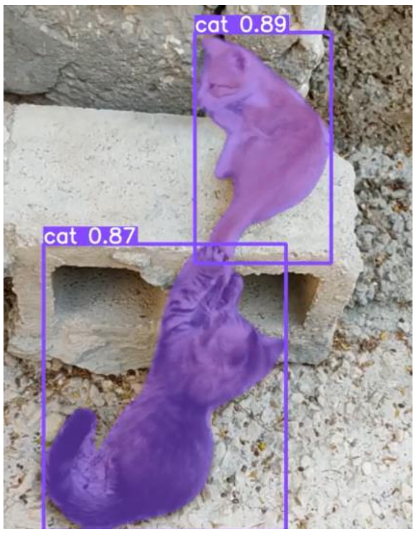


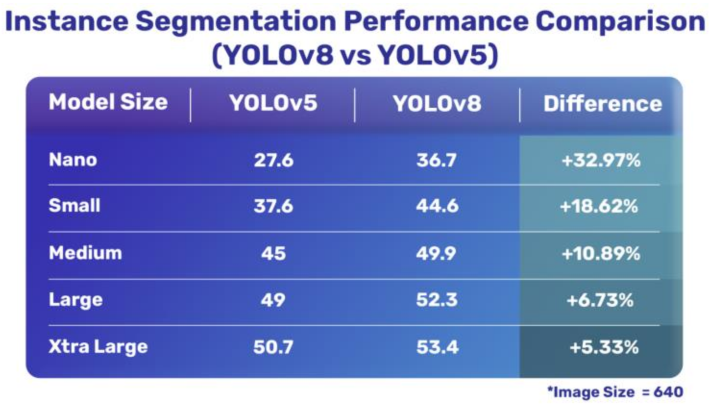
Fig. 2.4 Diagram of how a machine learning model capable of zero-shot learning operates.

2.4.3 YOLOv8 (You Only Look Once, Version 8)

YOLOv8, introduced in 2023 by deep learning company Ultralytics, is the latest cutting-edge, state-of-the-art (SOTA) model, building on previous versions to enhance performance, flexibility, and efficiency [12]. It utilizes a neural network to predict bounding boxes and classes and has object segmentation functionality which is similar to Meta's SAM. Differing from FastSAM, this model is trained on the COCO (Common Objects in Context) dataset. This set features 330k images, with 200k annotations for object detection, segmentation, and captioning, and 80 classes used in this model's training.

With cutting edge features, demand is high to integrate this software into any device capable of benefitting from its capabilities. It is also capable of running in real-time, and, unlike FastSAM, is capable of identifying and labelling detecting classes in an image. This model provides a similarly effective and modern option for remote attachment with embedded systems but varies in its abilities enough to offer some meaningful comparative differences.





Instance Segmentation Performance Comparison (YOLOv8 vs YOLOv5)

Model Size	YOLOv5	YOLOv8	Difference
Nano	27.6	36.7	+32.97%
Small	37.6	44.6	+18.62%
Medium	45	49.9	+10.89%
Large	49	52.3	+6.73%
Xtra Large	50.7	53.4	+5.33%

*Image Size = 640

Fig. 2.5 YOLOv8 segmentation results and performance comparison. [13]

As seen in the above figure, object segmentation and classification are possible, with the above frame being selected from a real-time video feed. With the object segmentation head of YOLOv8 stemming from the earlier YOLOv5, comparison of their performance provides some insight into the relative improvements gained over the older version. Rath, S. in their comprehensive guide to SOTA object detection, outlines some of these metrics in the table above [13]. These are scored according to their mAP (Mean Average Precision), which is a metric used to evaluate algorithm accuracy by calculating the average precision across multiple classes or categories. The available YOLOv8 model sizes range from Nano, for the model containing the lightest weight parameters, to Xtra large, for the model with highest parameters and most input data. Lower-level versions have the highest improvement in performance, but all versions improve by a minimum of %5 mAP score. This is promising, as the 'Xtra large' model can be deployed remotely and compared to a 'nano' model deployed locally on an embedded system.

2.4.4 Conclusions

In conclusion, this literature review aims to provide a comprehensive examination of foundational technologies and methodologies used in related literature, which are crucial to the development of the chosen embedded system and deep learning platform. Synthesizing insights from existing related research, the significance of microcontroller and sensor choice have been underscored, and the important of performance, energy efficiency, and real-time processing capabilities in the chosen hardware platform has been covered. Moreover, software design methodologies have been analysed with key algorithms and techniques for line detection and navigation reviewed, to collect insights as relevant. Further, exploring possibilities for the integrated deep learning platform reveals the potential to enhance computational capabilities remotely. Examining the Meta Segment Anything Model, its subsidiary version FastSAM, and YOLOv8 has demonstrated their promise for improving object segmentation and detection tasks from a remote standpoint. These offer positive implications for enhanced decision making and real-time performance capabilities. Moving forward, these insights will be leveraged to strategically implement the following project steps and lays the foundational pieces for the fulfilment of project objectives. Ultimately, this analysis will guide the way for the research and development of a system to further the capabilities and functionalities of autonomous embedded vehicle systems, or any similar embedded systems capable of benefitting from deep learning technologies.

Chapter 3: Foundational Techniques

3.1 Introduction

This section walks through the foundational equipment used and the techniques and knowledge required to proficiently utilize the tools and software selected. Each section breaks down the parts involved in the overall system and explains the application and necessary information for a quality implementation.

3.2 Microcontrollers (Raspberry Pi)

A firm grasp on the usage and application of microcontrollers, particularly the Raspberry Pi microcontroller, is pivotal due to their necessary processing power and versatility in managing sensor data and algorithms in the system. Furthermore, their performance and nearly ubiquitous use in miniature smart-vehicle applications across various research studies highlights them as an obvious choice, but successful application is benefitted by a moderate level of background knowledge.

3.3 Sensors (CSI Camera and Webcam)

As gathered during review of related literature, CSI cameras and webcams are regarded as crucial components for object detection and navigation. They are validated by their widespread use and effectiveness in diverse research implementations, emphasizing their significance in providing real-time data in a lightweight form-factor. Minimal previous experience is required, but an understanding of various camera output modifications, such as saturation and contrast, alongside basic comprehension of web streaming techniques benefits overall project success.

3.4 Software Design Evaluation

Experience and understanding of common edge detection techniques, such as Canny edge detection and Hough lines, and knowledge of approaches used in the fusion of various streams of sensor data is essential to a robust detection system. It is important to gauge the varying viability of different approaches used in related studies and pick the approach best suited to adaptability and diverse road conditions.

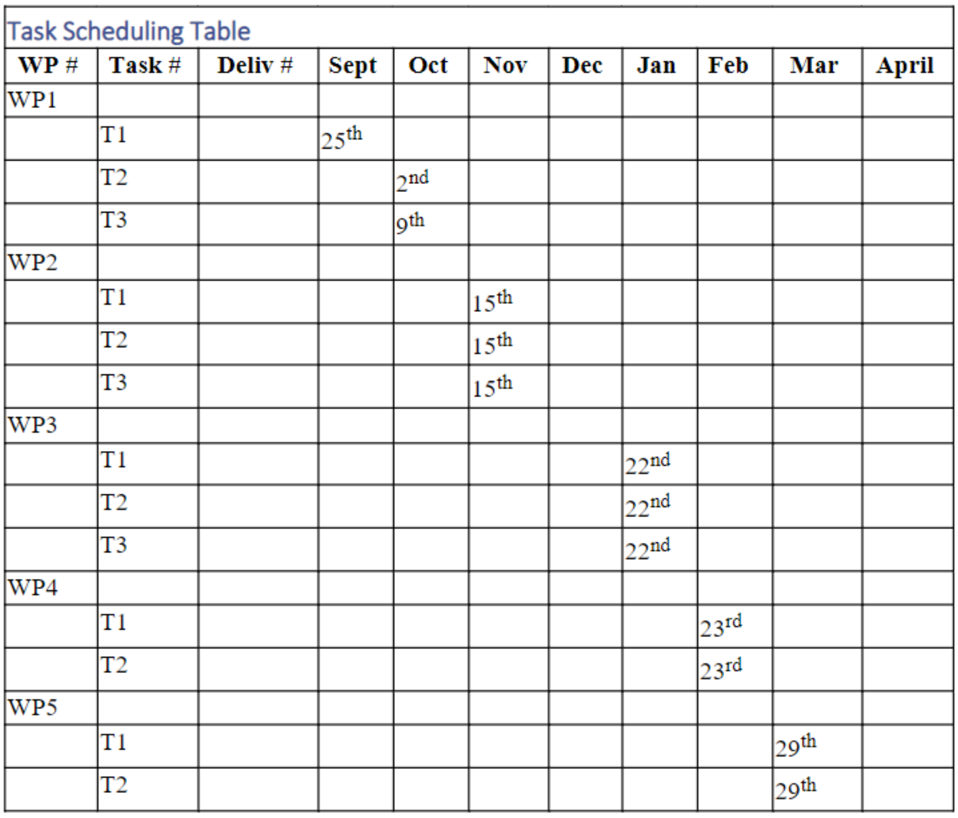
3.5 Deep Learning Models and Platform

Developing the platform for communicating and processing the information sent via the embedded system requires forethought into the entire design methodology. Knowledge of remote systems and computer systems which can support the various deep learning models is a benefit, as well as some basic foundational knowledge of communication protocols.

The Meta Segment Anything Model presents groundbreaking possibilities for object segmentation through its advanced learning capabilities. However, its implementation poses challenges due to its innovative yet complex nature and should be approached with some background comprehension of deep learning and implementation in small form-factors. Further, the YOLOv8 model previously discussed offers similar benefits and opportunities, but should be approached in a similar manner, with thorough background research performed into how it operates and how to deploy it on the proposed platform.

3.6 Comprehensive Roadmap

Serving as a guide for the integration of all project components and outlining the steps involved in multiple levels of detail, experience in developing realistic roadmaps for project completion is vital to project success. Working with the project's supervisor, a table estimating completion dates for project work packages (WP) was created.



Task Scheduling Table

WP #	Task #	Deliv #	Sept	Oct	Nov	Dec	Jan	Feb	Mar	April
WP1										
	T1		25th							
	T2			2nd						
	T3			9th						
WP2										
	T1				15th					
	T2				15th					
	T3				15th					
WP3										
	T1						22nd			
	T2						22nd			
	T3						22nd			
WP4										
	T1							23rd		
	T2							23rd		
WP5										
	T1								29th	
	T2								29th	

Fig. 3.1 Roadmap detailing estimated completion of work packages.

3.7 Conclusions

In summary, this section highlights the essential equipment and techniques needed for developing the autonomous smart vehicles embedded system and remote deep learning platform. Understanding microcontrollers, such as the Raspberry Pi, and the role of CSI cameras and webcams in object detection and navigation is crucial. Knowledge of common edge detection techniques and sensor data fusion is necessary for software design, while familiarity with deep learning models like the Meta Segment Anything Model and YOLOv8 requires background comprehension. Finally, developing a comprehensive roadmap is vital for successful project integration. The next chapter on System Design will begin to outline the integration and application of these foundational elements within the context of creating a miniature self-driving vehicle system, thus giving the reader further insight into the specifics of the project.

Chapter 4: Proposed System Design

4.1 Introduction

This section introduces the overall system design, including a block diagram which illustrates the integration of crucial components vital for completion of the embedded system and deep learning platform. This visual representation outlines the framework of the vehicle's autonomous system, highlighting the interconnectivity among these components and the broader DL platform. Effective integration stands as a key factor in achieving the defined work packages, necessitating an evaluation of component selections and their potential integration. The exploration of each system block, namely the Power Supply, Sensors, Microcontroller, GPU PC, and Chassis & Motor System, offers detailed insights into their specific functionalities and how they interact. Finally, a system component diagram visualizes the physical combination of these components, providing further detail on their role within the vehicle system.

4.2 System Block Diagram

Below is a system block diagram which illustrates how fundamental project components such as microcontrollers, sensors, and software modules work together for this project's autonomous miniature self-driving vehicle and deep learning platform. It visually maps the integration of components, forming the basic framework for the complete autonomous navigation vehicle and system.

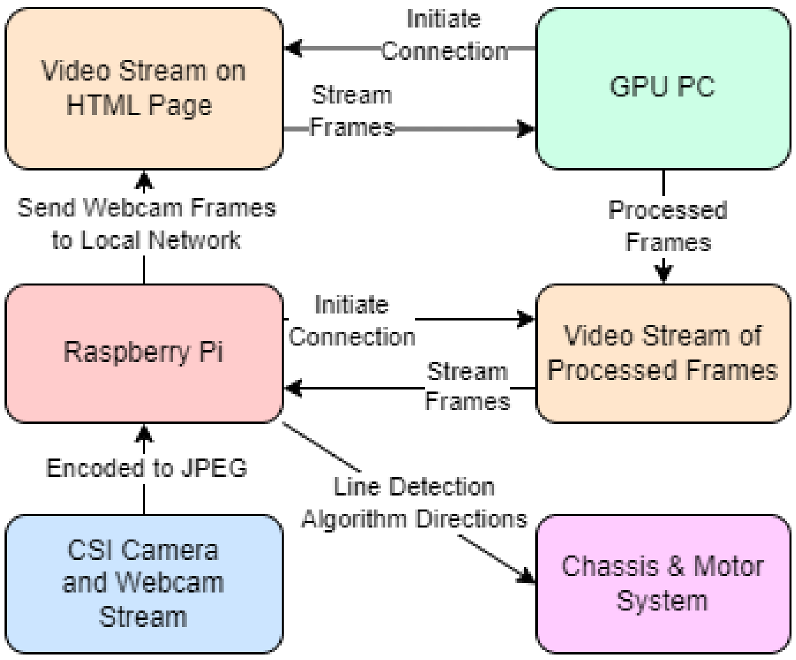

Video Stream on HTML Page
Initiate Connection
Stream Frames
GPU PC
Send Webcam Frames to Local Network
Processed Frames
Raspberry Pi
Initiate Connection
Stream Frames
Video Stream of Processed Frames
Encoded to JPEG
Line Detection Algorithm Directions
CSI Camera and Webcam Stream
Chassis & Motor System

Fig. 4.1 System block diagram illustrating vehicle software and hardware integration.

First, a brief description of each component and an overview of the entire system will be given, before diving further into each specific block and the proposed specifications of each. The Raspberry Pi microcontroller will be the main hub for information exchange, and ultimately control and direct the entire ecosystem. It receives both camera streams, the webcam and CSI camera, and either inputs it to the line detection algorithms, for the CSI camera, or a hosted HTML page of the video stream, for the webcam. The line detection algorithm processes this input and provides the appropriate directions to the chassis and motor system, which control vehicle movement. The frames sent to the HTML webpage are hosted there, waiting for the deep learning platform, i.e., the graphics processing platform (GPU) computer, to initiate a connection and retrieve them for processing. After processing, the GPU PC hosts its own HTML page of results, which the Raspberry Pi initiates a connection with and receives them as a stream of frames. In essence, these components combined compromise the basic system as a whole and its operation, and the following section will further elaborate on each component's specifics and details.

4.2.1 Microcontroller – Raspberry Pi 4b

The Raspberry Pi microcontroller serves as the central processing unit and operational core within the entire system. Its primary function encompasses the control and coordination of the hardware components and software modules used in navigation. The microcontroller will perform all computation and process incoming sensory data from integrated sensors to facilitate real-time environmental perception. This processing involves data interpretation, analysis, and decision-making algorithms, crucial for navigation, object detection, and avoidance.

Furthermore, the Raspberry Pi microcontroller oversees the execution of control commands, managing motor functions and steering mechanisms in response to processed data. This enables physical movement and manoeuvring of the vehicle within its environment. Its role extends to facilitating communication between software components which ensures seamless interaction and synchronization between the sensor, processing, and control systems. This integration and communication capability are pivotal for the system's cohesive functioning, allowing accurate responses to dynamically changing environmental conditions.

Choice of microcontroller was again largely based on related research, but also due to the versatility of the Raspberry Pi. Its open-source architecture and diverse compatibility with many software libraries offers flexibility in implementing and refining algorithms for navigation, object detection, and decision-making processes. Consequently, the microcontroller serves as the foundational element to the system and is essential for realizing the project outcomes.

4.2.2 Power Supply

Beginning with the power supply system, the vehicle will be operating off rechargeable Lithium-Ion batteries. These batteries will be attached underneath the vehicle chassis, and will power the microcontroller, sensors, and chassis/motor system. The type and size of battery was predetermined due to the chassis constraints, and will consist of two Samsung, ICR-18650, 3.7V, Lithium-Ion Rechargeable Batteries. These batteries have a 2.15Ah capacity, recharge in approximately 3 hours and, based on current testing, can power the vehicle for approximately 1-2 hours per charge cycle. These batteries will sufficiently power the vehicle, and, due to their connection with the microcontroller via the chassis circuitry, don't require any modifications or additional componentry.

4.2.3 Sensors

Based on the previously performed literature review, choice of sensors focuses on which cameras are best suited to the desired project outcomes. For its ease of integration, low computational load, and availability of documentation, the CSI camera was chosen to implement the classical technique line following algorithms. It provides a low power and easy to implement option, good for simple uses. The webcam was integrated into the system design as it became clear that a higher resolution camera which could be adjusted separately from the line following camera was necessary. For this reason, a C920 Logitech webcam was chosen, which offers high framerates (60FPS), resolution (1080p), and automatic focus. This serves to enhance the output of the deep learning models and improves the overall abilities of the deep learning platform.

4.2.4 Object Recognition and Segmentation Models

An object segmentation model is a deep learning algorithm capable of analysing images, recognising objects, and separating them from the other objects in a scene. Applying these capabilities to the current self-driving system gives a greatly increased level of comprehension which aids in decision-making and vehicle safety. This system will operate on the microcontroller, with sensor data from the camera being input into the algorithm and returned wirelessly. Algorithm output can then be processed by the microcontroller for any unexpected object, unseen conditions, or any data useful to the embedded system. In future systems, this model could be used exclusively for self-driving decision-making, negating the need for multiple cameras and visual processing techniques.

4.2.5 Chassis & Motor System

Finally, operation of the chassis and motor system will proceed according to the data received from the microcontroller. These components come from a preconfigured kit which provides the chassis, which is a PCB (printed circuit board), with attached servo motors controlling the wheels. Built for use with a Raspberry Pi, it is installed onto the kit PCB and communication is performed through the pin headers on the RPI, which are connected to the chassis circuit board. This board then connects to the 4 motors and communicates using software which comes with the kit and can be modified to improve or alter its functionality. No processing is done by the chassis circuit board or motor system, and they only respond reactively to the microcontroller commands. This is the final step in the navigation process, through which the system will navigate autonomously through its environment.

4.3 System Component Diagram

The system component diagram serves as a visual representation of the entire system, depicting the interconnectedness and integration of physical components within the autonomous vehicle system. The illustrative diagram below outlines the fundamental hardware pieces, including sensors, microcontrollers, and software modules, showing their functional relationships and dependencies. Ultimately, this illustrates the chosen embedded system in its entirety, and demonstrates the feasibility of integrating a remote, GPU accelerated deep learning platform into an embedded system.

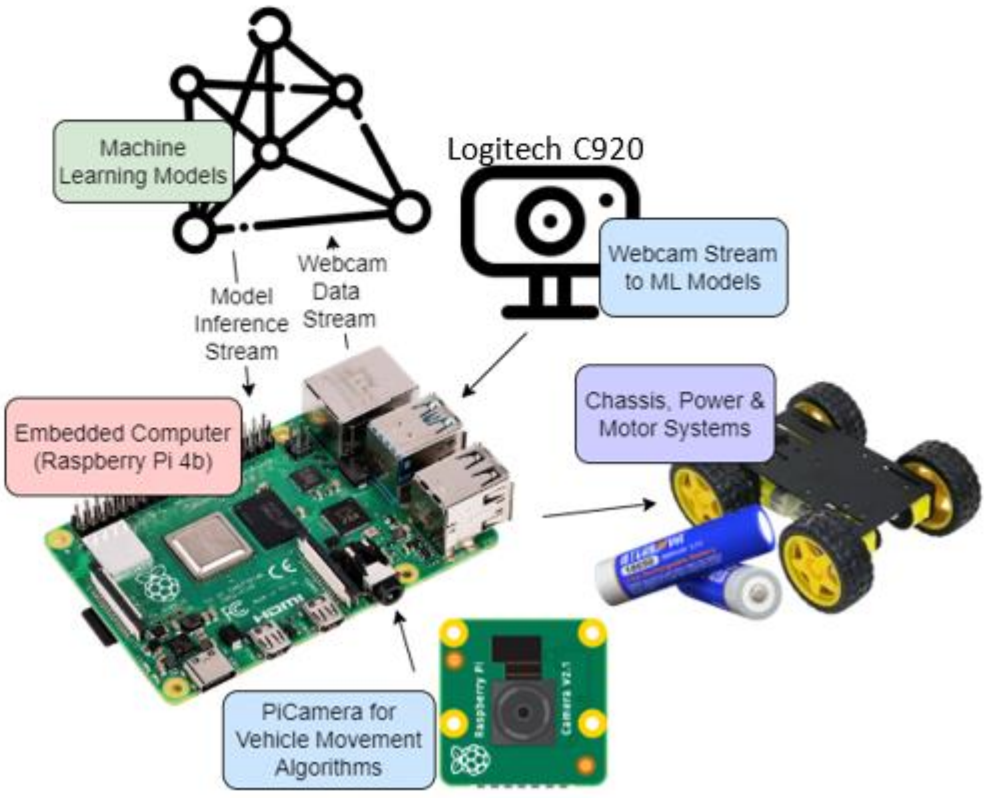

Machine Learning Models
Logitech C920
Webcam Data Stream
Model Inference Stream
Webcam Stream to ML Models
Embedded Computer (Raspberry Pi 4b)
Chassis, Power & Motor Systems
PiCamera for Vehicle Movement Algorithms

Fig. 4.2 System component diagram illustrating integration of physical components.

4.4 Conclusions

In summary, this section introduces a system block diagram that illustrates the integration of pivotal components crucial for enabling autonomous navigation in the proposed self-driving vehicle. The visual representations presented give the framework of the vehicle's autonomous system, emphasizing the interconnections among these components. Successful integration is imperative for achieving the work packages, warranting a thorough evaluation of component selections and their integration potential. The detailed exploration of each system block—Power Supply, Sensors, Microcontroller, Object Segmentation Model, and Chassis & Motor System—provides comprehensive insights into their specific functionalities and mutual interactions. Additionally, a system component diagram further explains the physical amalgamation of these components, offering a detailed perspective on their roles within the vehicle system.

Chapter 5: Feasibility Testing & Proof-of-Concept

5.1 Introduction

An initial project implementation was first pursued, aiming to demonstrate project feasibility and develop a proof-of-concept system containing preliminary versions of the desired product. This stage consists of foundational steps, such as assembling the vehicle hardware and completing its setup with the attached Raspberry Pi. The subsequent sections document these steps, outlining all the work done in achieving a stable initial state for the vehicle which proves the proposed design a viable pursuit and is critical for integrating and advancing the project's more sophisticated work packages. This phase represents the foundation which the project's future advancements and functionalities are built upon and lays the groundwork for these subsequent project phases.

5.2 Hardware Setup & Testing

The system hardware consists of the sensors used for data capture, car chassis which consists of header pins and a circuit board connected to the microcontroller, LEDs which can be controlled, and the 4 motors which control vehicle movement. The following section details the setup of these components.

5.2.1 Hardware Assembly

The base chassis kit consists of a printed circuit board (the chassis), 4 motors, 2 servo motors for controlling the angle of the CSI camera and various components for securing the pieces together. Not all these components are used in the final system, but initial setup includes assembling it all into the predefined form before modifications could be made. The assembly of these parts involved following a guide provided with the kit and proved relatively simple. The chassis PCB is preassembled with preselected chips such as the PCA9685, an I2C bus controller used as a 16-channel PWM (pulse-width modulation) servo driver for movement. This chip controls all the servos on the device, and the block diagram and pinout for it are below. Examining this chip and the other on the PCB gives a deeper understanding of how the board itself works and is an important step in gaining complete comprehension of the entire system.

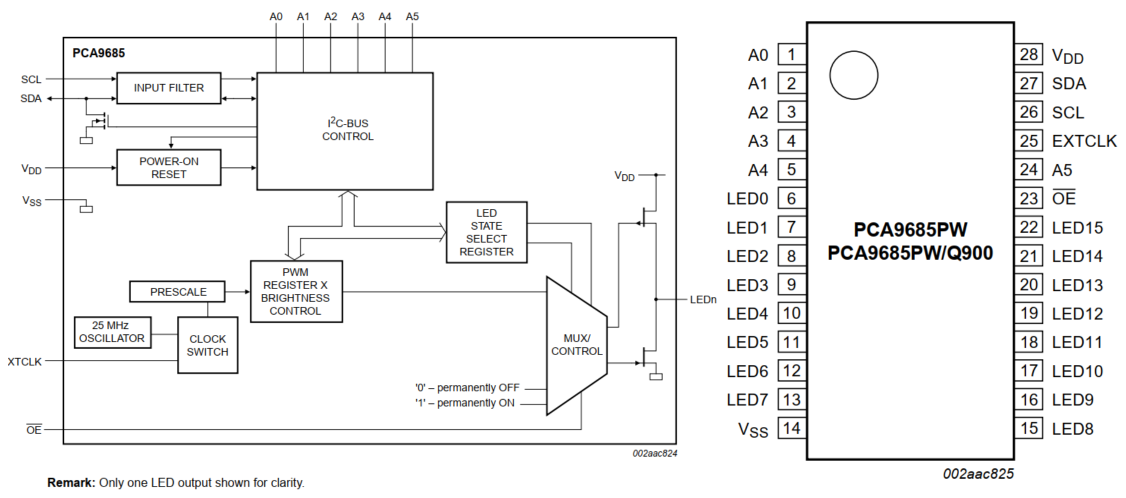


Fig. 5.1 Block Diagram and Pinout for PCA9685

However, problematically, there were issues with the included servo motors, which ultimately needed to be replaced and lengthened the assembly process. When the kit was assembled, the Raspberry Pi could be connected and provide computational abilities. Connection of the microcontroller to the chassis and sensors was simple, with connecting pins and headers between the chassis and board being the only requirements. Finally, with the hardware assembly complete, the power supply was connected, and the full system tested. These tests were successful, leaving the vehicle in a ready state for the advancements proposed in the project work packages.

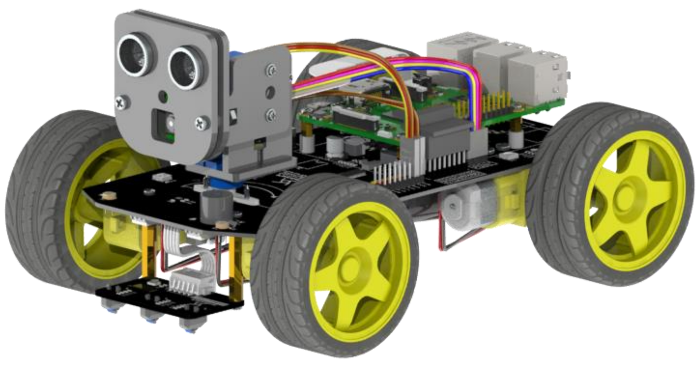


Fig. 5.2 Model of initial car kit configuration after assembly

After the kit was assembled, it was deemed necessary to make some modifications which would accommodate the additional webcam and remove any unnecessary components. So, the components which housed the CSI camera and other unnecessary components were removed, leaving a more efficient system better suited to capture high quality video and communicate wirelessly. Seen below, the result was achieved by placing the Logitech C920 webcam onto the top of the chassis and orienting the CSI camera at a 45-degree downward angle, facing the forward direction of the vehicle.

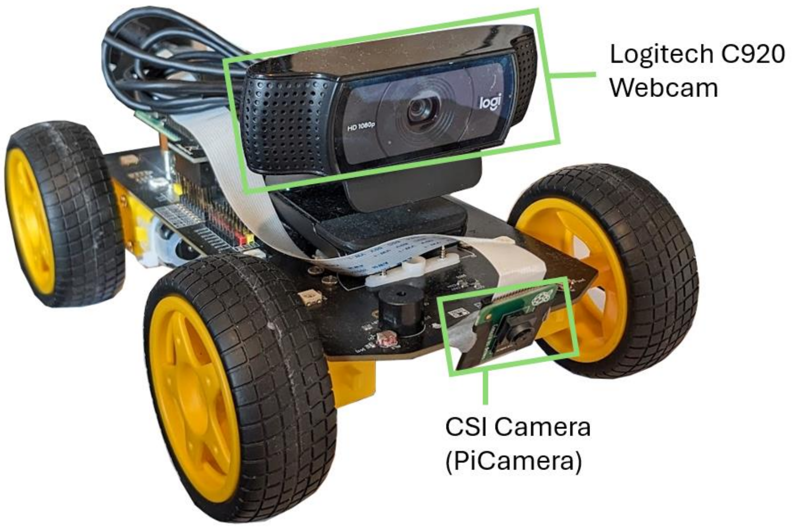


Fig. 5.3 Completed vehicle hardware setup.

With these modifications complete, the system hardware reached the completed assembly stage. To verify the system's capabilities, 4 tests of the basic functionalities were performed. Seen in the table below, these consist of a system boot, where the operating system and all basic system software are run, a motor test which involves a simple driving maneuver, a camera test to capture photos with both cameras, an LED test to blink the on-board LEDs included in the kit, and a test of how long the chosen batteries can power the system.

System Boot	**Motor Test**	**Camera Test**	**LED Test**	**Battery Test**
Operating system booted successfully with no issues.	All 4-wheel servos are working correctly.	Photos taken with no issues, manual focus on CSI camera needs adjustment.	LEDs operate as intended.	Batteries work as intended and have an average battery life of 47 minutes.

Table 5.1 Results of the 4 system hardware tests performed.

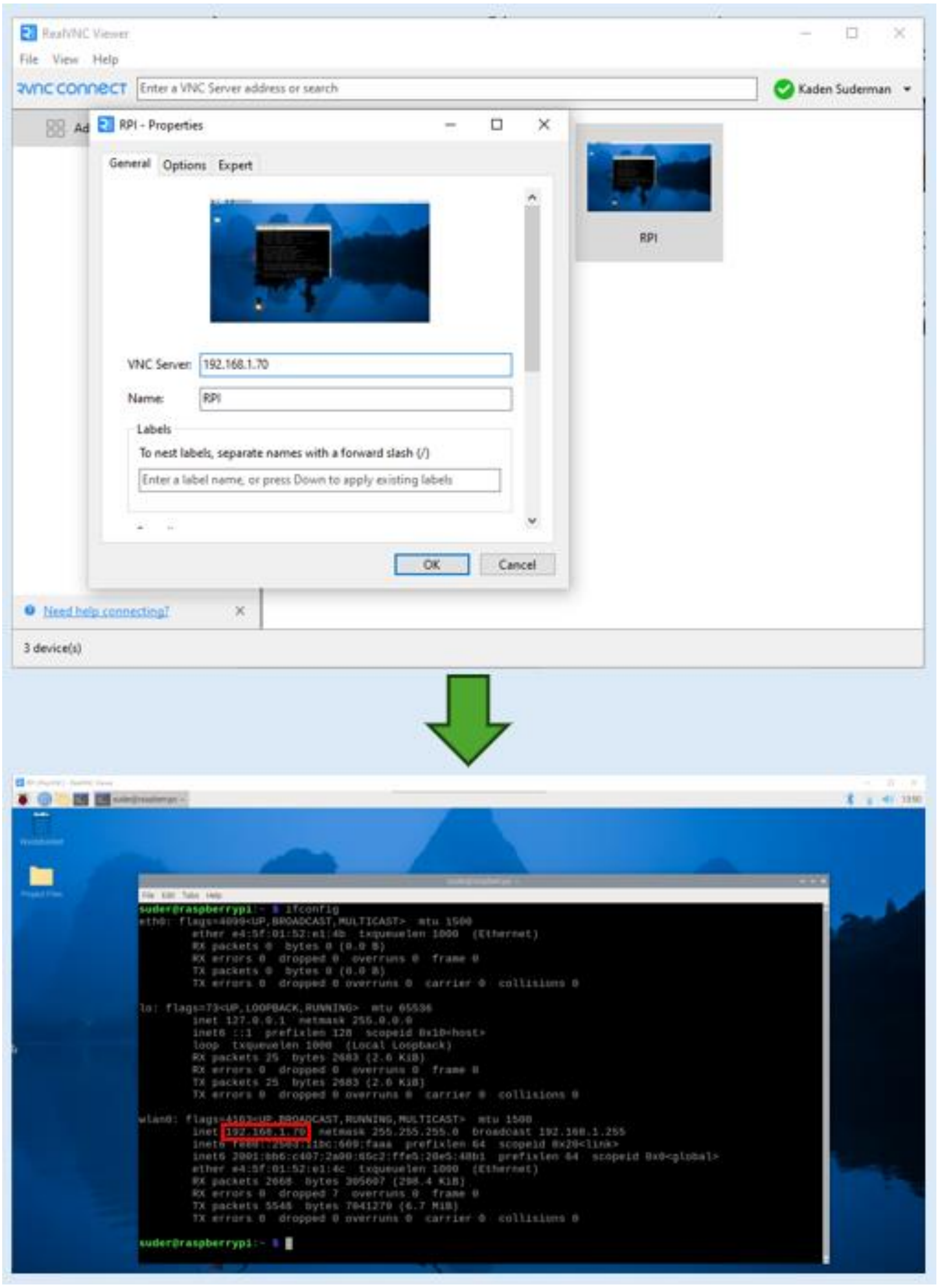
In conclusion, all the hardware tests were passed successfully, with the only necessary change being a slight focal change for the CSI camera. Since these cameras are simple, they don't have automatic focus, but manual adjustments can be made by twisting the lens. This adjustment gives a clearer view of the road in front of the vehicle and thus provides clearer input data for the line following algorithms. With the hardware setup and testing complete, development of the software components and development of the relevant algorithms begins.

5.3 Software Setup & Testing

The software setup and testing for this phase focuses on installing and configuring the operating system and necessary packages to perform some preliminary tests on the system. This includes running demo code provided by the company supplying the chassis kit, reviewing this code for future changes, and establishing the foundation for later development.

5.3.1 Operating System & Headless Connection

The software setup beings by installing a stable operating system on the Raspberry Pi. To do this, the latest version of the Raspberry Pi operating system was installed and configured. Specifically, the 64-bit version of the Raspberry Pi OS was selected, which is a more computationally powerfully version, providing better compute speeds than the 32-bit version. To connect to the microcontroller wirelessly, headless setup was configured. This involves enabling it in the configuration settings of the microcontroller and utilizing the RealVNC program on an external computer for connection. This requires enabling wireless support on the Raspberry Pi, configuring the IP address, and establishing a connection between the two devices.



VNC Server: 192.168.1.70

Name: RPI

Labels

To nest labels, separate names with a forward slash (/)

Enter a label name, or press Down to apply existing labels

```
suder@raspberrypi:~ $ ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether e4:5f:01:52:e1:4b  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 25  bytes 2683 (2.6 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 25  bytes 2683 (2.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.70  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::2503:11bc:609:faaa  prefixlen 64  scopeid 0x20<link>
        inet6 2001:bb6:c407:2a00:65c2:ffe5:20e5:48b1  prefixlen 64  scopeid 0x0<global>
        ether e4:5f:01:52:e1:4c  txqueuelen 1000  (Ethernet)
        RX packets 2668  bytes 305607 (298.4 KiB)
        RX errors 0  dropped 7  overruns 0  frame 0
        TX packets 5548  bytes 7041279 (6.7 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

suder@raspberrypi:~ $ 
```

Fig. 5.4 RealVNC Remote Connection to Raspberry Pi

Seen above, the RPI is added as a client connection in RealVNC using it's IP address. This address is found by executing the 'ifconfig' command in a terminal and returns various pieces of information on the RPI's network connection. With this setup complete, the system could be operated remotely, and the operating system was working properly.

5.3.2 Software Testing

The car chassis kit is supplied with software to communicate with and control the sensors and motors. These files are a starting point for developing the algorithms but are first used to connect the RPI with the chassis components. Establishing these connections involves running various Python scripts included in a GitHub repository supplied by the kit provider company [14]. Once this repository is downloaded, scripts corresponding to different pieces of hardware, i.e., the sensors, lights, and motors, are run to test the components and ensure they work properly.

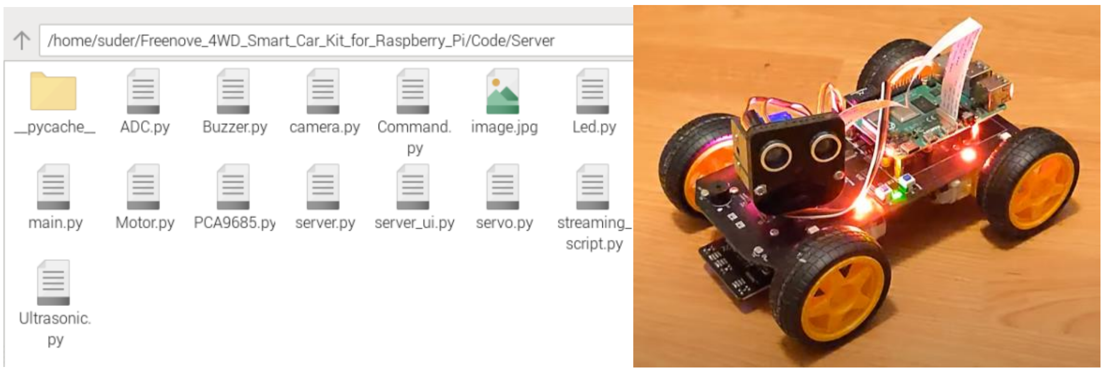


Fig. 5.5 Repository of code files provided in kit and LED test.

All the provided test files are tested and run successfully, meaning the vehicle portion of proof-of-concept setup is finished. To gain a deeper understanding of how the provided system software works, some time is spent parsing the provided software files. To understand how the motor control works, the PCA9685 python script is examined (see Appendix B). The program defines a class called PCA9685, which controls the 16-channel PWM servo driver. This chip is a popular PWM controller and is commonly used to drive servos and LEDs. The program uses the 'smbus' library to communicate with the chip via the I2C protocol. It defines methods for writing and reading data from registers on the chip, setting the PWM frequency, and setting the duty cycle, the time a circuit is on and off, for individual channels. This is the most important code to comprehend, as it establishes connection to the motors, necessary for the later addition of line tracking functionality. With this stage finished, software setup and testing are complete.

5.3.3 Deep Learning Platform Testing

Lastly, the deep learning platform is tested to verify the model functionality. This platform consists of a computer with a GPU operating separately from the device and communicating via a network connection. To test that they would function, the two models, FastSAM and YOLOv8, were downloaded to the computer. The appropriate libraries were installed, and each was run with out-of-the-box settings on a few images. FastSAM was tested first on the two input images below. The left image segments everything and shows positive results as it accurately segments all objects in the image. The right image is prompted with the text, ‘The brown dog’, and segments the left dog quite accurately.

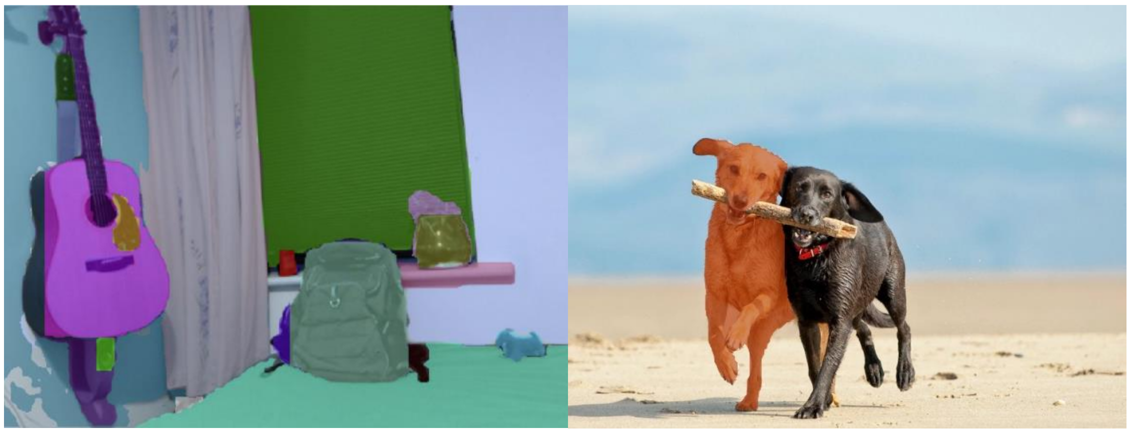


Fig. 5.6 Test images for FastSAM.

YOLOv8 is also tested, with two versions of it being run. Both are the ‘Xtra’ large versions, requiring more processing power but giving better results. The first output in the figure below is the version with the additional capability of segmentation, and the second is the slightly lighter version without. Both were processed successfully and produced good initial results. With these tests complete, the proof-of-concept phase is successfully concluded.

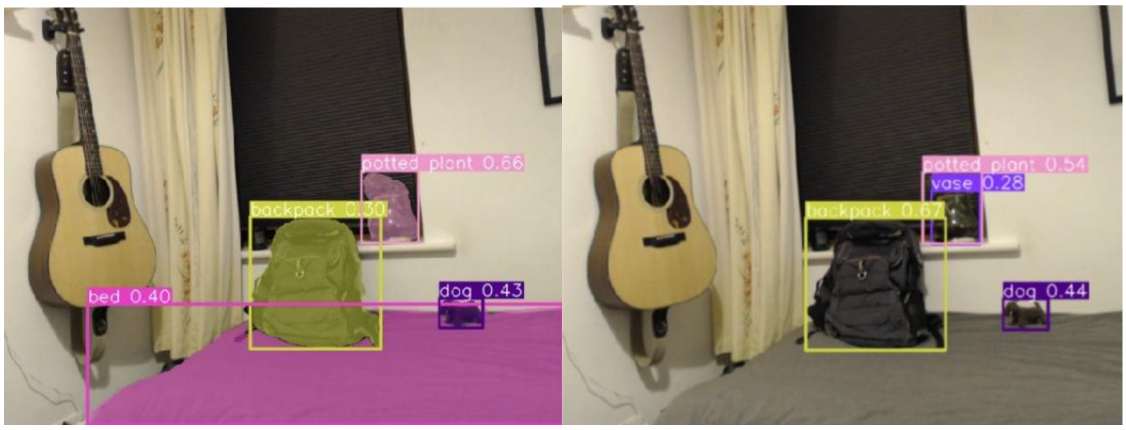


Fig. 5.7 Test images for YOLOv8 and YOLOv8-seg.

5.4 Conclusions

As explored, this project's initial stages centre on system setup and software testing, ensuring the establishment of the operating system on the Raspberry Pi and validating software functionalities controlling sensors and motors. Hardware assembly involves the construction steps and troubleshooting, overcoming challenges with servo motors to finalize the chassis construction. Successful testing validates the entire system's functionality, marking a milestone and preparing the vehicle for the upcoming implementation of more complex project components and functionalities. These foundational phases form the cornerstone for the project's evolution and expansion into more advanced work packages, with the next section detailing the specifics of project implementation.

Chapter 6: Core Project Implementation

6.1 Introduction

With the desired project work packages defined and the overarching design developed in the previous sections, the core blocks of the implementation are begun. This section focuses on the methodology used in developing the line following algorithms, the deep learning platform, and the demonstrational model where all can be tested. These preliminary implementations will be later refined but are brought to a core development stage upon which later analysis, enhancements, and refinements can be made. The first stage in achieving the core aims and objectives outline for the project is the line following system and its methodology.

6.2 Line Following Algorithm Design

The design for the line following algorithm is relatively simple and utilizes classical and simple algorithmic techniques to maintain a reasonable load on the embedded system. A block diagram of the proposed design is displayed below. Frames are first received by the program from the CSI camera, which is installed at a 45-degree downward angle at the front of the car. This keeps the region of interest (ROI) for the system within approximately 0.3 meters in front of the vehicle. This serves to reduce false positives on the line detection.

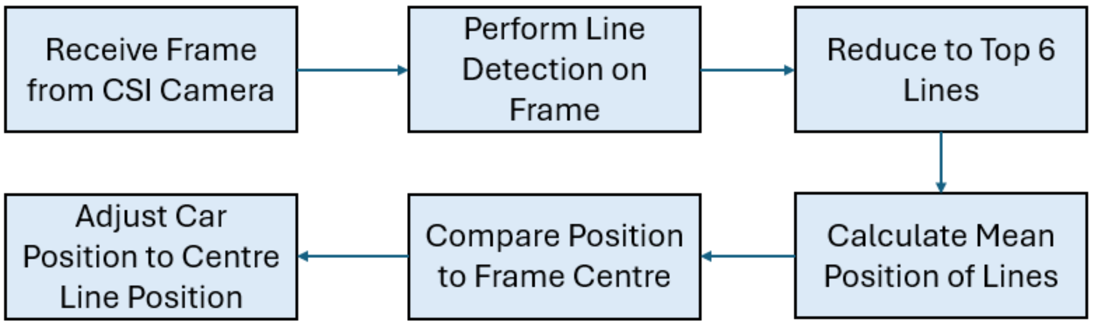

Receive Frame from CSI Camera
Perform Line Detection on Frame
Reduce to Top 6 Lines
Calculate Mean Position of Lines
Compare Position to Frame Centre
Adjust Car Position to Centre Line Position

Fig. 6.1 Block Diagram of line following design.

Each video frame is then processed through the line detection algorithm to detect all lines, then reduced to the top 6 lines by length and confidence. The average position of the lines is then calculated to find the midpoint between them. This tells the system if the camera, and thus the car, are off-centre from the desired line by comparing its position to the centre of the frame. Finally, the distance between the average and the true frame centre are calculated and the car's position is adjusted to centre it on the line. With the block diagram assembled, developing the software implementation is the next step.

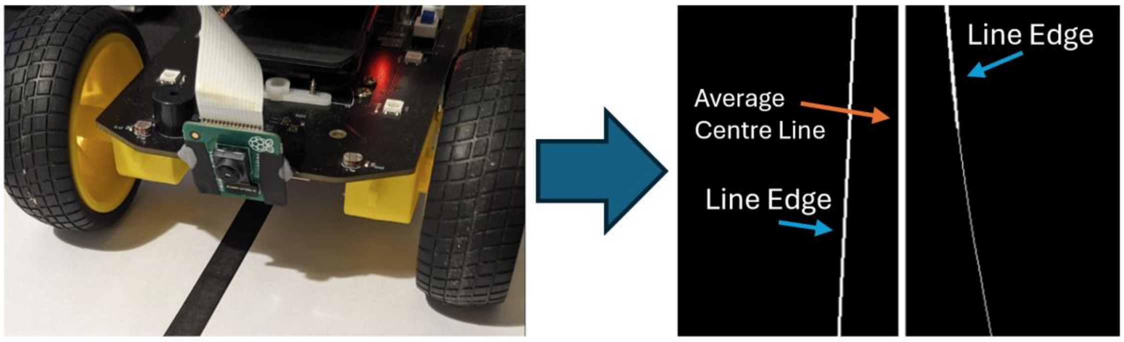
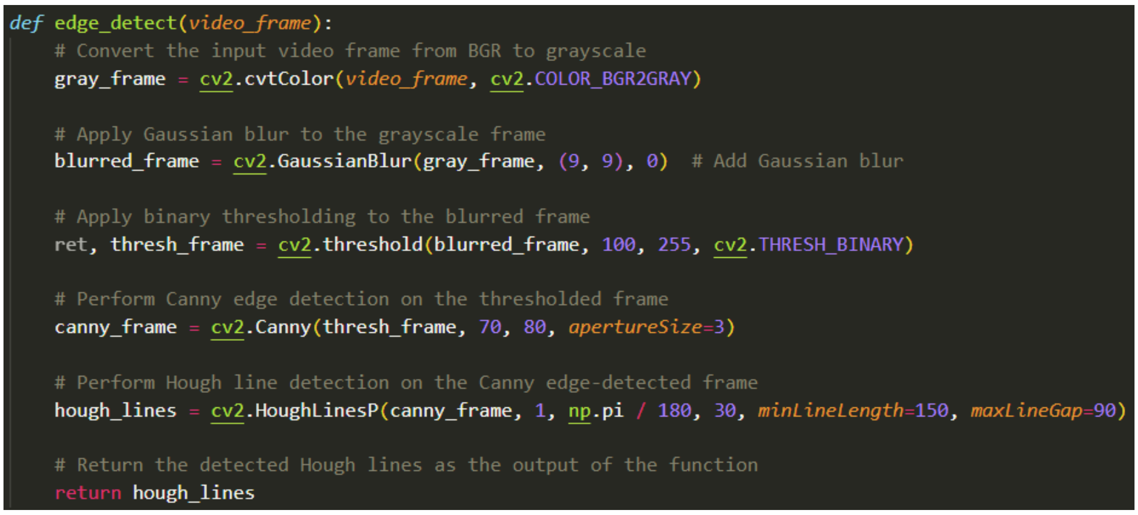


Fig. 6.2 Showcase of image conversion from CSI camera to line position.

6.3 Line Following Software

Developing the software for the line following mechanism utilizes suggestions from related research and relies on classical techniques. The techniques chosen are like those suggested in previously reviewed research by Shahane et. al. (2022) and are considered classical computer vision techniques because they are traditional methods and algorithms used for analysing and processing visual data in computer vision tasks. The designed algorithm is a function, displayed below, which steps through each necessary step described in the following sub-sections.



```python
def edge_detect(video_frame):
    # Convert the input video frame from BGR to grayscale
    gray_frame = cv2.cvtColor(video_frame, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur to the grayscale frame
    blurred_frame = cv2.GaussianBlur(gray_frame, (9, 9), 0)  # Add Gaussian blur

    # Apply binary thresholding to the blurred frame
    ret, thresh_frame = cv2.threshold(blurred_frame, 100, 255, cv2.THRESH_BINARY)

    # Perform Canny edge detection on the thresholded frame
    canny_frame = cv2.Canny(thresh_frame, 70, 80, apertureSize=3)

    # Perform Hough line detection on the Canny edge-detected frame
    hough_lines = cv2.HoughLinesP(canny_frame, 1, np.pi / 180, 30, minLineLength=150, maxLineGap=90)

    # Return the detected Hough lines as the output of the function
    return hough_lines
```

Fig. 6.3 Edge detection function code.

6.3.1 Colour Conversion

These steps are all constituent parts of the larger function which encapsulates the entire line generation process. The initial step in the 'edge_detect' function involves converting the input video frame from the original colour-space to grayscale and is done using the 'cv2.cvtColor' function. This function is provided by OpenCV, a popular computer vision library for Python which is used extensively throughout this project. This conversion step simplifies the subsequent processing steps as it reduces the image's computational complexity and changes the focus to the intensity of variations in the image.

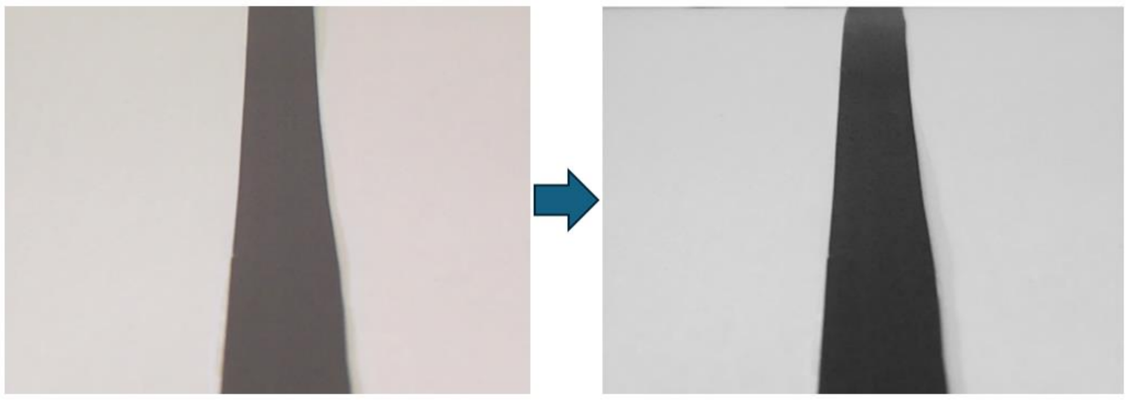


Fig. 6.4 Video frame before and after color conversion.

6.3.2 Gaussian Blur

After greyscale conversion, a Gaussian blur filter is applied via the 'cv2.GaussianBlur' function, which smooths out the noise and fine details in the image. A kernel size of nine by nine is used, which determines the extent of blurring, with larger sizes resulting in stronger smoothing effects and smaller kernels providing more fine details in the image. The blurring operation serves to improve the accuracy of edge detection by reducing high-frequency components in the image and their influence.

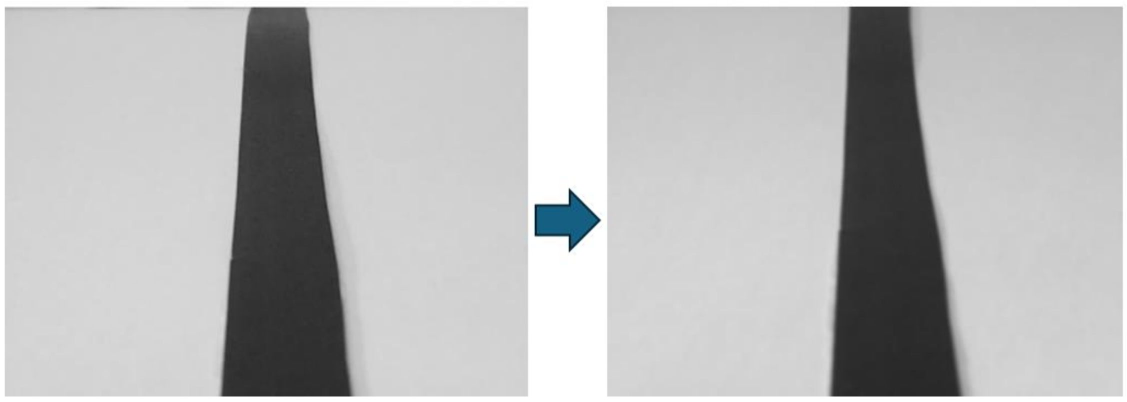


Fig. 6.5 Video frame before and after Gaussian blur.

6.3.3 Binary Thresholding

Once the Gaussian blur filter is applied, the next stage is applying a binary thresholding operation on the blurred greyscale frame. This is done using the 'cv2.threshold' function, which is also supplied by the OpenCV library. This function works by taking threshold values, set to 100 and 255 in this case, and segmenting the image into foreground and background regions based on these values. Effectively, this modifies all pixels in the image to be either a fully white value or fully black value. The arguments define that any pixel values below 100 will be set to 0 (black), and anything above will be set to 255 (white). This enhances the contrast between edges and non-edges, which prepares the image for the subsequent stage of edge detection.

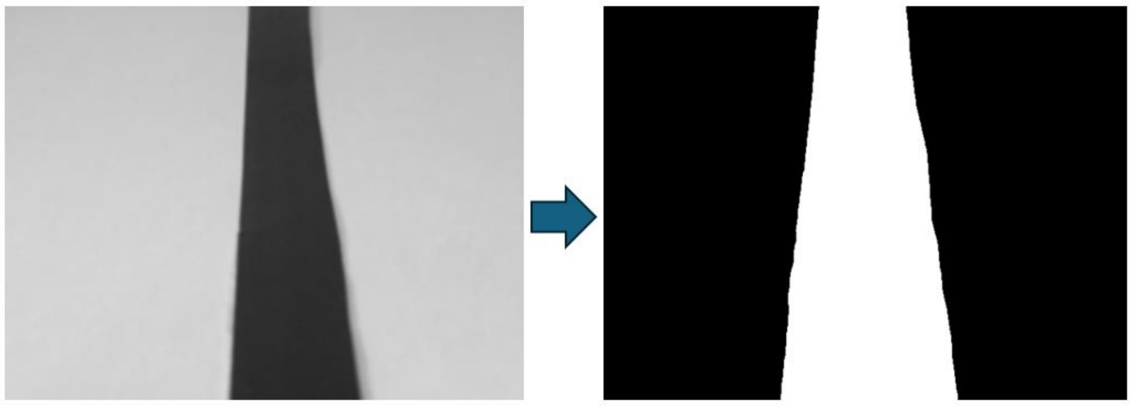


Fig. 6.6 Video frame before and after binary thresholding.

6.3.4 Canny Edge Detection

Next, Canny edge detection is performed on the frame after thresholding using the ‘cv2.threshold’ function. This is a classical technique for edge detection and works accordingly:

1) First, the magnitude and direction of intensity changes as each pixel is examined.
2) Next, suppression of edge size is performed to thin edges and improve accuracy.
3) Then, thresholding is done to remove any lines below or within a range of values.
4) Finally, edge pixels are connected and refined using hysteresis to give the result.

The result after this stage is a binary image where edges are represented as white lines on a black background. The parameters (70, 80) specify the lower and upper thresholds and control how sensitive the filter is to edge detection.

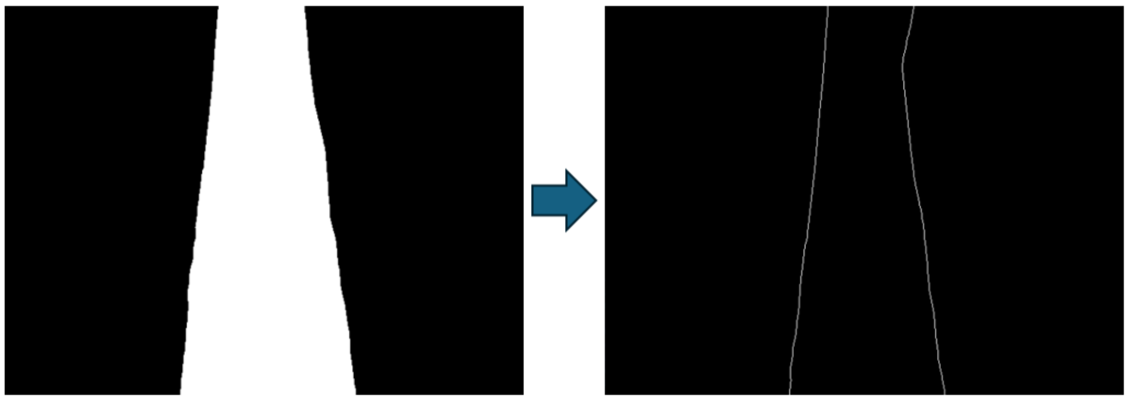


Fig. 6.7 Video frame before and after Canny edge detection.

6.3.5 Hough Line Detection & Line Filtering

The final filter is applied to the Canny edge-detected frame and utilizes the ‘cv2.HoughLinesP’ function. This is an important filter to use after edge detection, as this algorithm analyses patterns of edge pixels to identify lines in the image. It returns the detected lines as line segments which are defined by their endpoints. It is adjusted using various parameters, with the most important being 'minLineLength', which sets the minimum required pixel length for a line to be drawn (set to 150), and 'maxLineGap', which is the maximum value of pixels between two edge pixels to still be considered a line (set to 90). This is the last step in the edge detection pipeline and provides a selection of plausible lines to be used in decision-making for the line following algorithm.

All the edge detection filter steps are complete, but the lines still require some parsing to improve their usability. Done in the main section of the program, these lines must be reduced to only those important for tracking the line in front of the vehicle. To do this, the first step is reducing the number of lines on the screen to only include the four most prominent. Testing shows this as the ideal number of lines for this scenario as the detected road, when it curves, needs more than one line section to capture the whole area.

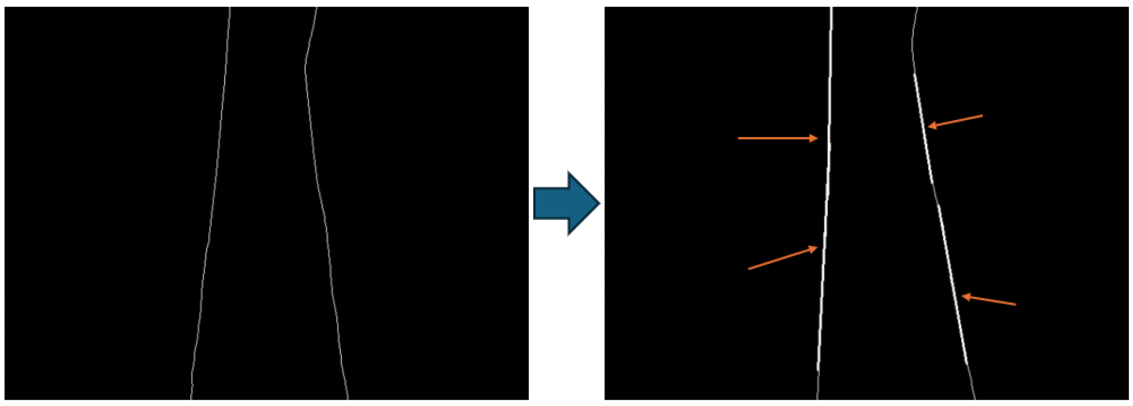


Fig. 6.8 Video frame before and after Hough line detection and filtering.

Further, horizontal lines are also filtered as to negate false positives. To do this, a trigonometric function is used, calculating whether a given line segment is approximately vertical based on its angle with respect to the horizontal axis and a specified angle threshold. The arctan function below calculates the angle of the line segment with respect to the horizontal axis in degrees. If the absolute difference between this angle and 90 degrees is less than a specified angle threshold variable, the line segment is treated as vertical.

$$\text{angle} = \text{abs}\left(\frac{\text{arctan2}(y_2 - y_1, x_2 - x_1) \times 180}{\pi}\right)$$

Fig. 6.9 Function to determine angle of line segment with respect to horizontal axis.

6.3.6 Integration with Chassis

With the line tracking algorithm complete, integration with the chassis system to control the vehicle is the next step. Before starting development, designing a decision tree to outline the steps and decisions the vehicle makes must be completed.

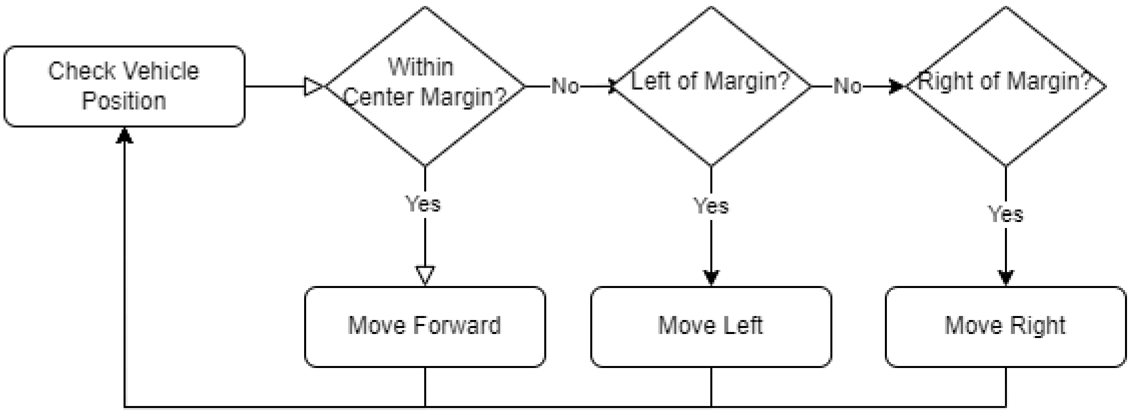
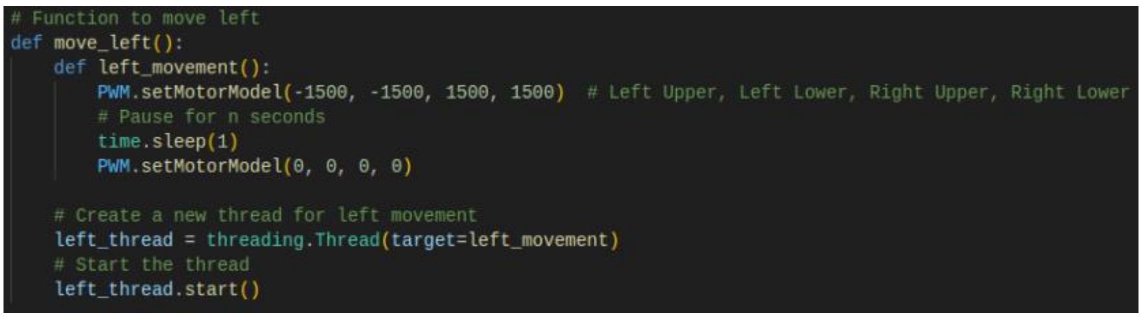


Fig. 6.10 Decision tree of car movement design.

A new function, "check_vehicle_position', is instantiated to check if the average line's midpoint is within a defined center margin of the image center. If within the margin, the car moves forward. If the line is left/right of center the car moves left/right to center the frame. These movements are done by calling other functions which control vehicle movement using modified versions of the predefined software supplied with the chassis kit. These functions make a call to a PWM class which interprets and sends the appropriate signals to the motors. The figure below demonstrates setting the wheel's movement in the 'move_left" function, which sets the two right wheels to a duty cycle of 1500, and the left wheels to 0. The right wheels then move at this speed for 2 seconds, pivoting the car left as the left wheels are not moving.



```
# Function to move left
def move_left():
    def left_movement():
        PWM.setMotorModel(-1500, -1500, 1500, 1500)  # Left Upper, Left Lower, Right Upper, Right Lower
        # Pause for n seconds
        time.sleep(1)
        PWM.setMotorModel(0, 0, 0, 0)

    # Create a new thread for left movement
    left_thread = threading.Thread(target=left_movement)
    # Start the thread
    left_thread.start()
```

Fig. 6.11 Function defined to turn the vehicle left.

Threading is also necessary and allows each movement function to operate independently and asynchronously. This means that if, while changing direction, the center moves back within the margin, the function to move forward can begin, stopping the state of the move left function and keeping it from over-adjusting the vehicle angle. This completes the logic for the self-driving portion of the project, but still requires testing to verify its performance and identify potential issues.

6.4 Deep Learning Platform Design

With line tracking complete, the next project stage focuses on designing and developing the deep learning platform: a separate computational system integrated and communicating wirelessly with the embedded vehicle system. This platform's aim is to enhance the embedded system's abilities and, more importantly, showcase the feasibility and process involved in integrating two such systems. The main objectives of this work package are to receive data wirelessly from the vehicle, process this data using large cutting-edge deep learning models, and perform testing and comparison against the standalone capabilities of the edge device. This section documents that process, beginning with a closer examination of the chosen models.

6.4.1 Chosen Models and Purposes

As discussed in the literature review previously conducted, the choices of models have been filtered down to the final choices of the FastSAM and YOLOv8 models. The reasoning behind picking these models is twofold. Firstly, they are both new and cutting-edge models, both being released in 2023 and already having widespread adoption in the computer vision community. Secondly, they both offer large model sizes with two complex but separate data sets (SA-1B and COCO-128) and different applicable use-cases.

6.4.1.1 FastSAM

Each of these models offers different abilities and benefits. FastSAM can segment all the objects in an image simultaneously, which is otherwise not possible with other models. It can also generalize class attributes of unknown objects presented by comparing them to known classes. It can also segment objects via text prompt, making it useful for custom scenarios without any necessary custom training. This combination opens the door for unique applications, such as:

Automated inventory management to segment items on shelves, identify specific products, and parse stock using text prompts.

Use in manufacturing robots to track surroundings and avoid collisions in unprecedented scenarios.

Assistive technology for the visually impaired, providing real-time auditory descriptions of the surrounding environment.

Endless further examples exist, but this selection demonstrates the uniqueness and applicability of such a model and why it warrants application to deep learning platforms. The one drawback of this model is, since it is a lighter version of the base SAM model, it cannot label objects it identifies but, rather, counts them and identifies them by count number. To supplement this lack, and provide further comparative analysis, the YOLOv8 model will also be examined, tried, and tested.

6.4.1.2 YOLOv8

The YOLOv8 model is also able to run in real-time and offers varying sizes of the model which have differing levels of computational load and processing ability. A version with the addition of object segmentation also exists and is identified by the '-seg' suffix placed after the size variant. The table below shows a comparison of the metrics of these models.

Model	**mAP Val**	**Speed A100 TensorRT (ms)**	**Params (M)**	**FLOPs (B)**
YOLOv8n, YOLOv8n-seg	37.3, 36.7	0.99, 1.21	3.2, 3.4	8.7, 12.6
YOLOv8s, YOLOv8s-seg	44.9, 44.6	1.20, 1.47	11.2, 11.8	28.6, 42.6
YOLOv8m, YOLOv8m-seg	50.2, 49.9	1.83, 2.18	25.9, 27.3	78.9, 110.2
YOLOv8l, YOLOv8l-seg	52.9, 52.3	2.39, 2.79	43.7, 46.0	165.2, 220.5
YOLOv8x, YOLOv8x-seg	53.9, 53.4	3.53, 4.02	68.2, 71.8	257.8, 344.1

Table 6.1 Metric comparison of different available YOLOv8 models. [15]

Across the board, it is apparent that, while similar, the non-segmentation version performs better in all metrics. This is to be expected as increased computational load is applied by this additional capability. However, the computational cost increase is generally reasonable in most cases. The Mean Average Precision (mAP) value evaluates the precision and recall over multiple classes and gives a single scalar value which summarizes performance across all classes. An increase of less than 1 percent is seen, which is impressive as this is primarily the most important metric, defining how well object classes will be recognized. Next, inference speed when the model is deployed on an NVIDIA A100 GPU using the TensorRT optimization framework is compared, measuring time taken to process input data and generate the output predictions. The segmentation values fall within a small margin of the base version, with a slight increase seen with increases in model complexity.

The Params column refers to the number of parameters for the model, measured in millions, with each segmentation version again falling slightly behind the base version. Finally, the FLOPs column, measured in billions, gives the number of floating-point operations per second, and quantifies the computational workload of the model, indicating its computational efficiency. With an increase in model size, the difference in FLOPs lessens, with a ~30% difference for the 'n' model and a ~25% difference for the 'x' model. Ultimately, this means that a larger segmentation model will require less power respective to the smaller models which is an important metric to consider for power savings in small edge devices.

Looking briefly at some proposed uses of this model, numerous options are available. In the realm of self-driving vehicles, one utilization is the improvement of pedestrian, road, and street sign detection. Object detection has long been used in these systems, but the addition of segmentation adds another layer of comprehensive data to be processed. This data can be used to understand to a greater extent the dimensions and characteristics of the road, more accurately measure object depth, and improve system comprehension.

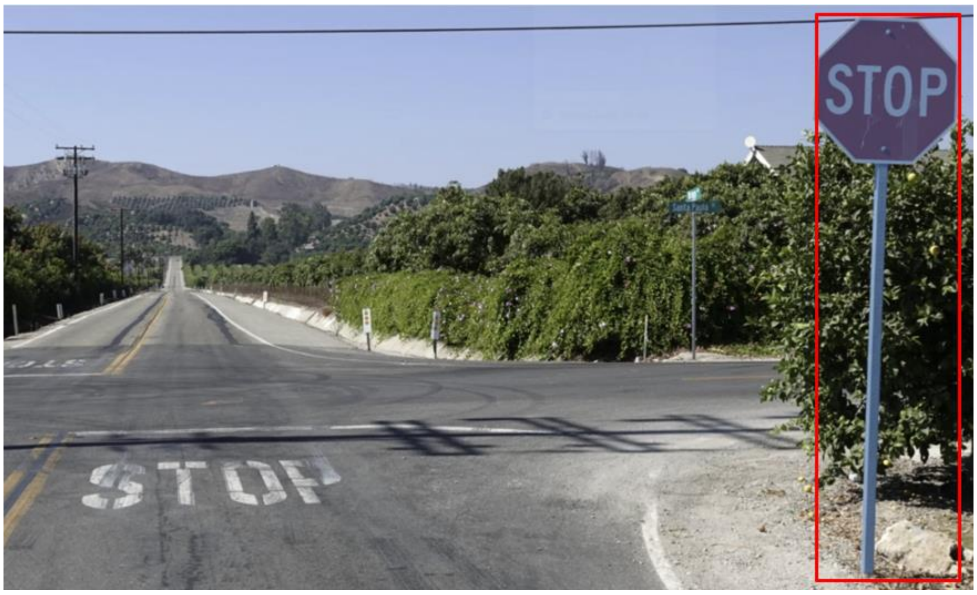


Fig. 6.12 Proposed use of YOLOv8 segmentation in a self-driving application.

6.4.2 Technical Details

Development of the deep learning platform starts with the initial setup of a computer on which to run the models. An available computer was chosen with a GPU and CPU capable of small-scale machine learning tasks and has the follow specifications:

- NVIDIA GeForce GTX 1080 Ti 11GB GPU
- AMD Ryzen 7 1700x 3.4GHz 8-Core Processor

This setup is considered a mid-range deep learning system, suitable for tethered wireless application with many systems, and is ideal for remote acceleration of embedded systems. The first step in setting up the models to operate on this system is downloading and configuring the models, their required packages, and pretrained model checkpoints to use for inference. These model checkpoints are pretrained weights provided by Ultralytics which use the previously described datasets and contain all the required information to support inference “out-of-the-box”, i.e., without needing to train them using custom data. For the extent of the feasibility testing in this project, these checkpoints are what are used for testing and their default capabilities what is assessed. For the design of the software using these models, the 4 following objectives are the focused outcomes:

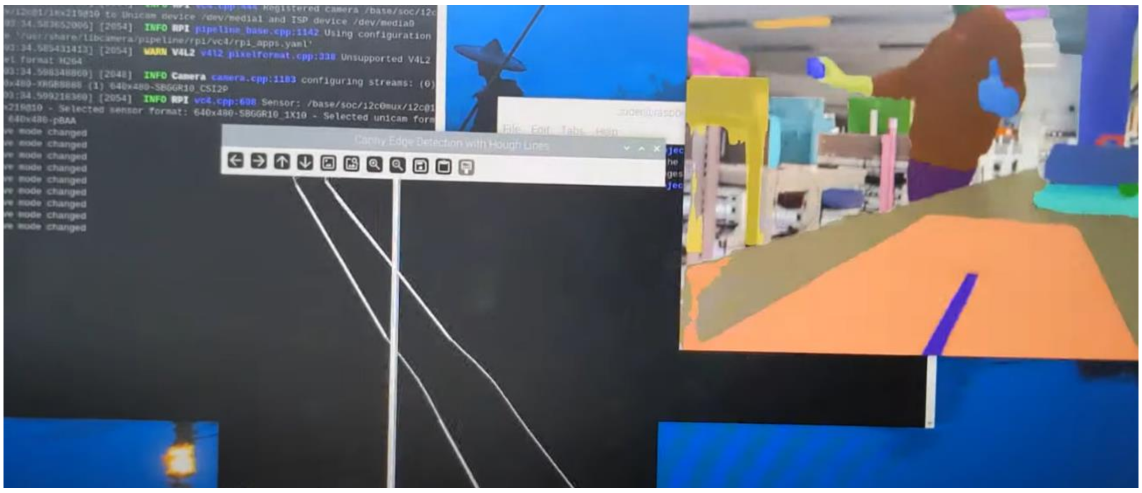
1) Communicate with the embedded system wirelessly.
2) Communicate with the embedded system in real-time.
3) Receive and return data to the embedded system.
4) Improve the functionality of the embedded system.

Communicating wirelessly with the embedded system is a cornerstone of this research, with potential successful results demonstrating the feasibility and applicability of the idea. This communication must also be performed in real-time (i.e. within a specified time constraint of less than 1000ms) to ensure that this system can react appropriately to environmental changes. Return of data from the platform to the embedded system enables this decision-making and must ultimately improve the functionality of the embedded system. With these key points defined, software development begins.

6.4.3 Software Development

Based on the system block diagram covered in section four, the platform collects image frames from a locally hosted HTML page of the video stream, processes them, and then hosts the processed frames on a separate HTML video stream webpage. The RPI is then able to collect these frames and utilize them in various ways on the embedded system. The option of returning commands or intelligent decisions to the RPI rather than video was considered, but for the core implementation this was decided against to keep the scope manageable.

Referring to multiple code repositories of example code and documentation supplied by the developers of FastSAM and YOLOv8, the base program to run inference on the video stream was written [18][19]. A separate Python script for each model captures the video stream from the specified localhost URL of the RPI’s IP address on port 8000. Each frame is processed using the pre-trained FastSAM and YOLOv8 models to segment all objects in the frame and detect objects respectively. The FastSAM script overlays detected object masks onto the original frame, which highlights the detected objects in the video stream, while the YOLOv8 model overlays masks and bounding boxes for each detected class. The processed frames, with overlaid masks and bounding boxes, are then sent to a local host HTML page of the PC's IP address on the same port. These are uploaded in real-time and can be accessed by the RPI so long as it is on the same network. This video stream processing continues until the user interrupts the stream by pressing the 'q' key, which terminates the program and releases the model inference program. With the core features of the deep learning platform complete, it can be integrated with the vehicle design to ensure its functionality. This proved successful, with all the scripts successfully running simultaneously.



```
                                  Registered camera /base/soc/i2c
/i2c@1/imx219@10 to Unicam device /dev/media1 and ISP device /dev/media0
03:34.583652006] [2054]  INFO RPI pipeline_base.cpp:1142 Using configuration
e '/usr/share/libcamera/pipeline/rpi/vc4/rpi_apps.yaml'
03:34.585431413] [2054]  WARN V4L2 v4l2_pixelformat.cpp:338 Unsupported V4L2
el format H264
03:34.598348860] [2048]  INFO Camera camera.cpp:1183 configuring streams: (0)
0x480-XRGB8888 (1) 640x480-SBGGR10_CSI2P
03:34.599216360] [2054]  INFO RPI vc4.cpp:608 Sensor: /base/soc/i2c0mux/i2c@1
x219@10 - Selected sensor format: 640x480-SBGGR10_1X10 - Selected unicam form
 640x480-pBAA
ve mode changed
ve mode changed
ve mode changed
ve mode changed
ve mode changed
ve mode changed
ve mode changed
ve mode changed
ve mode changed
```

Fig. 6.13 Demo of all software scripts running simultaneously on the embedded system.

In essence, the software setup consists of 4 primary scripts which control operations on the embedded vehicle system and on the deep learning platform. These are:

1) Line following algorithms on vehicle system.
2) Webcam stream to local host web server from webcam on vehicle.
3) Frame capture and inference on deep learning platform.
4) Stream of model output to local lost web server.

These four parts make up the greater integrated ecosystem between the embedded device and remote computation platform. This integration provides a seamless mechanism by which the processing power of small edge devices can be magnified greatly to improve and enhance a myriad of preexisting systems. Next, the demo model design showcases the testing environment and the chosen criteria for evaluation.

6.5 Demo Model Design

To demonstrate the abilities of the integrated system and its performance, a demo model must be designed. The core implementation concept for the model involves a basic "road" for the vehicle to follow which consists of white A4 paper with ~2cm black lines printed vertically on each page. The vehicle follows an approximately 2-meter track, stopping at the end, i.e. where no lines are present. The line tracking and streaming software both work simultaneously, and real-time video feed is transmitted from the embedded device and processed output returned to the embedded device.

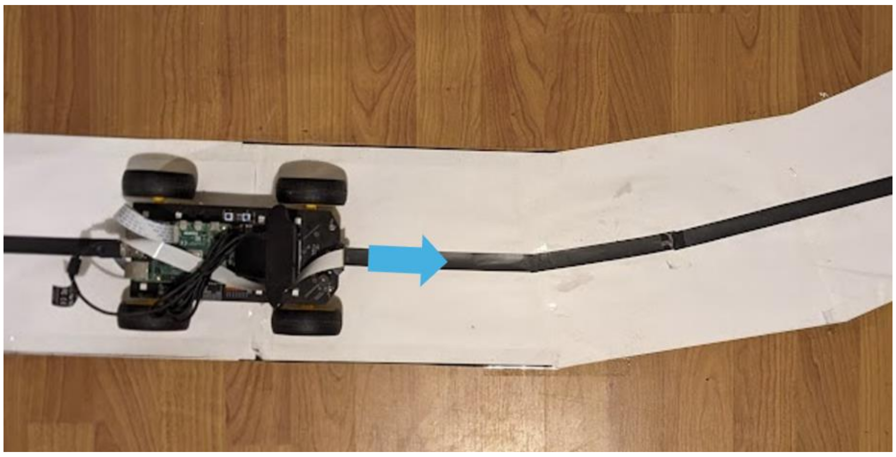


Fig. 6.14 Vehicle line following demo on the test track.

The core model for the demo only has curved lines at less than 45 degrees to the previous line segment. Further improvements may later be made regarding the system's capabilities, test criteria, and refinements. Ultimately, the core implementation stands primarily as a demonstration of the abilities of the entire system and shows the feasibility of adding additional software for deep learning platforms while maintaining high edge device performance.

To pick ideal criteria for analysis, research has been conducted regarding performance testing of embedded systems and machine learning on edge devices [21][22]. The selected criteria are:

- Model Response Latency and Inference Speed.
- Video Feed Resolution and Frames per Second.
- Ability to identify objects presented, such as stop signs and pedestrians.

These criteria encapsulate the critical performance needs of embedded systems and are commonly used in similar comparisons found in research.

In designing the demo model, the focus is on creating a simplified yet representative environment showcasing all capabilities of the integrated system effectively. The model aims to demonstrate line tracking functionality over a manageable distance through the proposed track, but the scope for the model leaves room for further refinement and enhancement. The following sections will evaluate the performance of the system via the specific criteria derived from research on similar embedded systems. This sets the stage for the subsequent section which focuses on analysis of results and refinements of the project.

6.6 Conclusions

In conclusion, the development of the entire system consists of line following algorithms, a deep learning platform, and a demonstrational model. It represents advancements in the realm of embedded systems and the feasibility of their integration with external computational accelerators. The outlined methodology has provided a framework for the design and implementation of core project components and sets the stage for subsequent analysis and refinement. The evaluation of the system against chosen criteria will offer valuable insights for further optimization and innovation in this research field and ultimately aims to drive advancements in real-world applications.

Chapter 7: Results & Refinements

The constituent parts of the video streaming process have been monitored and analyzed to collect data on the model inference speed, ideal video feed resolution, frames per second, and total latency for frames to be sent and returned. The following sections explore these results and provide analysis, potential refinements, and conclusions.

7.1 Deep Learning Platform Performance

The table below lists the metrics sampled for the analysis of the deep learning platform. Looking first at connection latency, this metric measures the time between the RPI capturing a frame and receiving the processed frame back on the RPI. As this is a local host connection, most latency comes from the time it takes to process the image and capture/return the image in the software. These values fall in the range of 500ms to 1000ms, which keeps within the previously specified real-time requirements of 1000ms for the system. The frames streamed were at an FPS (frames per second) of 30 and were returned around 27FPS and 28FPS for FastSAM and YOLOv8x-seg respectively. These are ideal speeds, with very little data loss being seen. The webcam stream is, however, capped at 30 FPS, with the reduction in FPS being due to buffering issues and lag in frame transmission.

Model	Connection Latency	FPS	Inference Speed	Detection (Stop-sign, person)
FastSAM	500-1000 ms	27 FPS	45-50 objects at 8-11ms, Resolution: 480x640	Segmenting at 0.55m, 1.7m
YOLOv8x-seg	500-1000 ms	28 FPS	31ms, Resolution: 480x640	32% at 0.91m 28% at 0.76m

Table 7.1 Deep Learning Platform performance metrics.

The inference speed, how long it takes the model to process and overlay masks on the original image, is very quick for FastSAM which segments 45-50 simultaneous objects in the frame at speeds of 8ms to 11ms. YOLOv8 processes slower at an average speed of 31ms but is still within the ideal speeds of less than 100ms. All processing and inference tasks were performed on the frames at a resolution of 640x480 pixels. The maximum framerate can be estimated with these inference results, with FastSAM able to reach a potential ~90FPS, and YOLOv8 potentially able to reach ~32FPS. This is calculated by dividing 1000ms by the inference speed of each model to give the total inference count per second.

Finally, the detection abilities for the models were tested, with the furthest distance an object could be detected at, and the confidence value (for YOLOv8) being recorded. For this test, the images seen in the figure below were used by printing them each onto an A4 sheet of paper and slowly increasing their distance from the camera until they were no longer detected. These distances were then measured and an average of five tests taken. The images were printed to-scale with the car, the stop sign measuring 15cm top to bottom and the pedestrian scene measuring 14x7cm. As the model car is at a scale of approximately 1:15 (for an average car with length of 4.5m [23]), the size of stop sign and pedestrian scene were scaled accordingly.

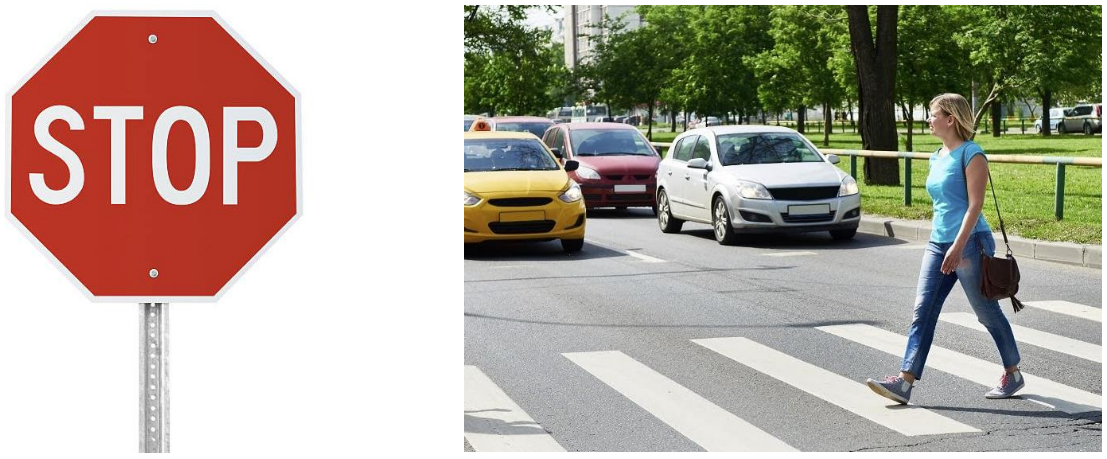


Fig. 7.1 Stop sign and pedestrian photos used in testing.

Results for the models show that both can identify the objects presented at an average distance of 0.55 and 1.7 meters. To obtain these averages, 5 measurements for each object were taken, which can be seen in the plot below. These measurements were taken under the same conditions and demonstrate each model's variability in detection range which, while small, should still be considered for applications.

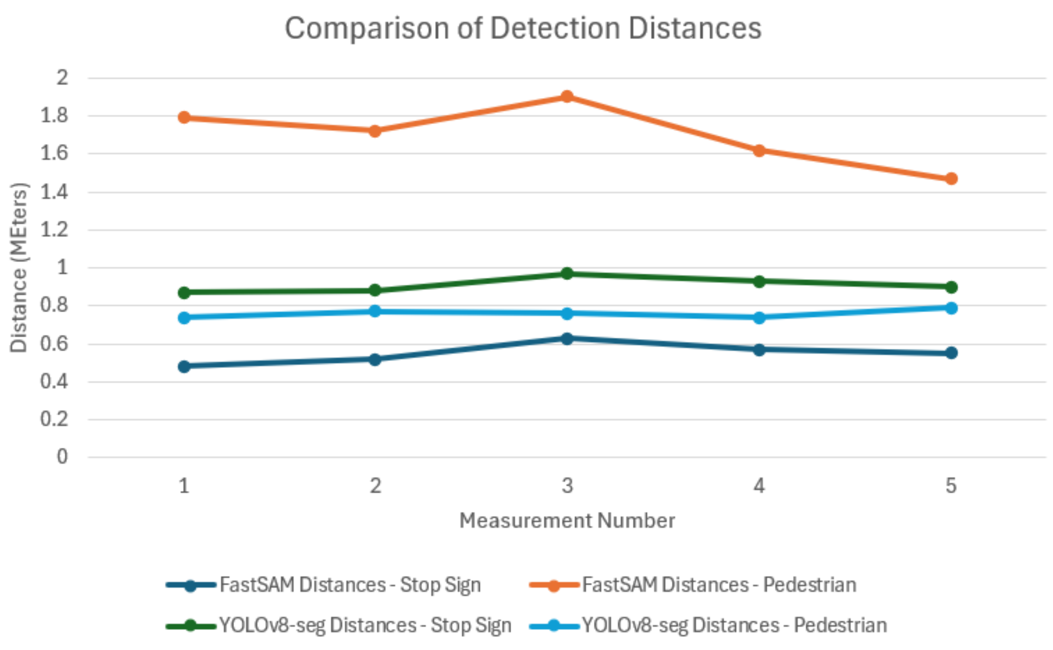


Figure 7.2 Comparison of the detection distances of the pedestrian and stop sign images.

FastSAM was able to segment the person image at a maximum distance of 1.9m, which is significantly further than the other measurements and shows great promise for use in tasks which involve segmenting distant or small objects. The other measurements were relatively similar, with YOLO detecting stop signs the furthest, followed by pedestrians, and finally FastSAM detecting stop signs.

FastSAM does not have confidence values, as it only segments and doesn't classify, but YOLOv8 had average confidence percentages of 32% for the stop sign and 28% for the pedestrian measurements. These confidence values indicate the model's degree of certainty in its classifications at different distances. Despite low confidence values at further distances when compared to FastSAM's segmentation, YOLOv8 still demonstrates the ability to detect and classify objects with reasonable accuracy across varying distances, showcasing its potential for applications requiring real-time object detection and classification.

7.1.1 Conclusions and Improvements

With implementation of the core project functionality for the deep learning platform complete, a few key takeaways should be noted. Both models functioned well, demonstrating efficient performance metrics and overall latency speeds which fall within the desired real-time range of 1000ms. Video stream FPS is ideal, reaching the near 30FPS maximum which, for applications outside of the project scope, is well above what is often necessary and thus provides room for future reduction of computational load and complexity. Inference speed and resolution also performed quite well, with the computer being able to perform inference on a mid-range resolution image at fast speeds. Finally, the segmentation and classification distances and confidences show the model's abilities in a task suited for the automated vehicle system design.

While the models exhibit commendable performance across the performance metrics, there are still opportunities to optimize and refine the design further. Connection latency stands as a serious consideration for improvement as reducing this variable enables a greater array of systems it can be integrated with. Resolving buffering issues and reducing lag in frame transmission could also enhance overall system efficiency and responsiveness. Alterations could also be made to reduce computation costs in areas such as FPS to increase other metrics such as resolution. Modifying system abilities in this way can increase suitability for other applications and broaden the range of use-cases. Ultimately, while model performance was good in testing, there are still opportunities to optimize and refine the design further, enabling broader integration, increased efficiency, and expanded applicability across various use cases.

7.2 Line Following Algorithms and Demo Model Performance

The demonstrative model's performance is based on the abilities of the software and algorithms to navigate the test scenario with speed and accuracy. This section examines these and other capabilities of the model, software, and hardware setups, and provides analysis of the results.

7.2.1 Line Tracking Results

The line tracking algorithms on the embedded system were tested and analyzed with a focus on three primary objectives, those being:

1) Speed – How fast can the vehicle track the lines and move across the track?
2) Accuracy – Are there any situations which diminish or break the algorithm functionality?
3) Capabilities – What amount of processing can be performed on-device simultaneously?

Speed was assessed by placing the car on the track and recording the time taken to reach the end point of the track and stop. This track consisted of six A4 sheets of paper, arranged in different configurations for each test, and totaling an approximate driving distance of 2.4 meters. The car was tested on three different track configurations, seen below, each with increasing complexity.

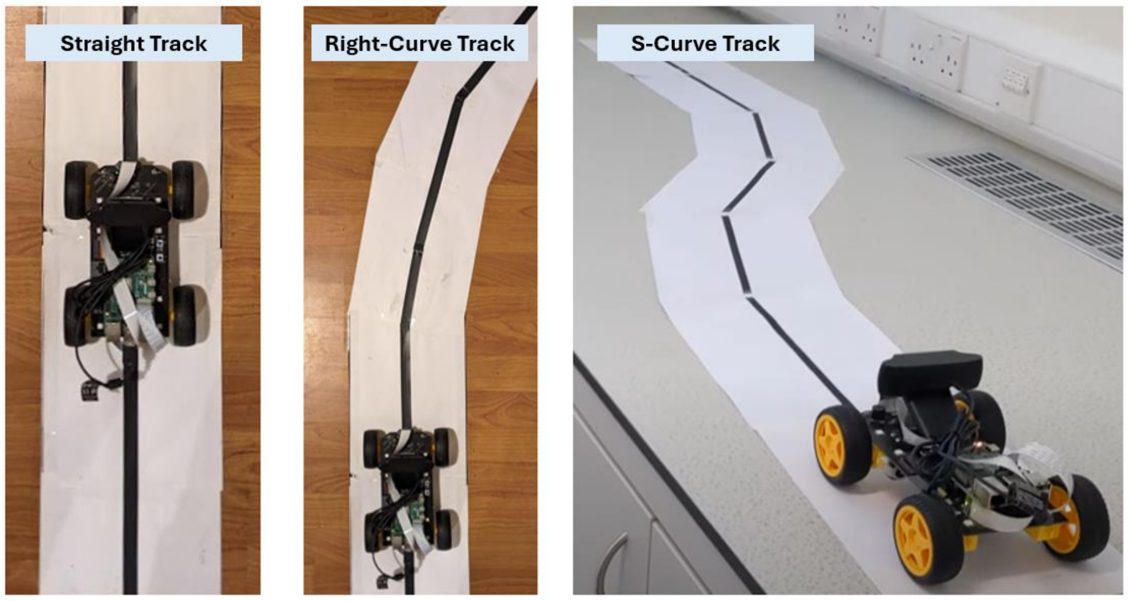


Figure 7.3 The 3 track types used to test the line following abilities.

Each track consisted of the same amount of A4 sheets, was the same length, and was either completely straight, a continual right curve, or an S-curve shape. The car was placed at the start of each track and released, with timing finishing when the car could no longer find the line to track and would thus stop. These timings were recorded five times for each track and the average completion times taken.

Looking at the data for each track, the straight track completed significantly quicker than the others, finishing at an average of 5.5 seconds. The S-Curve track finished second, with an average of 19.8 seconds, and the right-curve track came in last at 31.5 seconds' average. Such a variation in completion times was not expected but is likely attributable to the curvature in each track. Curvature in the track causes a significant slowdown in movement speed as, instead of being able to pivot the wheels a certain degree and continue forward movement, vehicle rotation is accomplished by rotating the right and left wheels in opposite directions to spin the car. This is the only way to turn the vehicle as the chassis doesn't possess a wheel pivoting system and means that any time turning is required, forward motion must stop. While impacting the movement speed of the vehicle, this is only a minor issue which could be resolved with a different vehicle model.

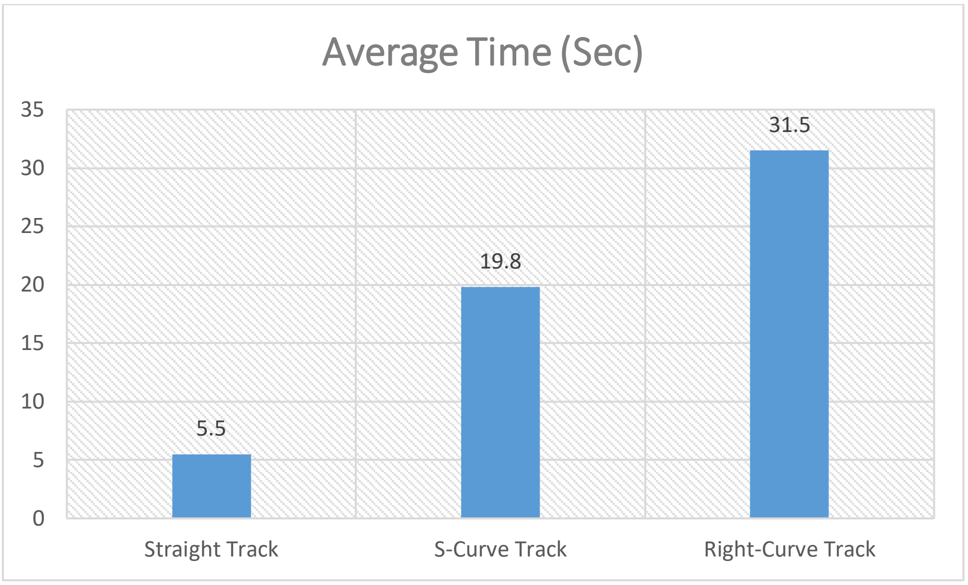


Figure 7.4 Average completion times for each track.

The algorithm accuracy was analyzed simultaneously and demonstrated positive results with a few minor issues. The primary issue noticed was that, if moving at its full speed on a straight section of the track, the time taken to notice a curve would be too slow. This kept the vehicle from turning at the appropriate moment and would push it off the track. This issue was quite simple to fix, and this was done by lowering the vehicle's maximum speed. After this change, tracking would no longer be lost on sharp turns and the vehicle would turn appropriately. Another minor issue slowed down the turning speeds of the vehicle due to the lack of ability to turn the wheels. A slight jitter would occur when the vehicle made directional adjustments which would cause the algorithm to lose track of the line for a very short period which would slow the turning adjustments. Fixing this problem would aid in the time taken to complete routes with curvature and improve model performance and could be done with the addition of a pivoting wheel system.

Different lighting conditions also influenced the ability of the line-tracking algorithm, with harsh lighting and dim lighting being the primary issues. Harsh lighting would cause an over-detection of edges, which would reduce the distance the system could accurately track. Dim lighting would conversely cause an under-detection of edges, leading to gaps in the lines and lessened tracking distance. Solving these problems could be done in two ways, either through adding fixed lighting to the front of the car to illuminate the path forward consistently or using infrared cameras and infrared lighting to give consistent lighting to the approaching track. Both are viable approaches, with the infrared suggestion being more complex but suitable for more conditions, and the simple LED approach being straightforward but adding constant lighting which may not be ideal for some applications.

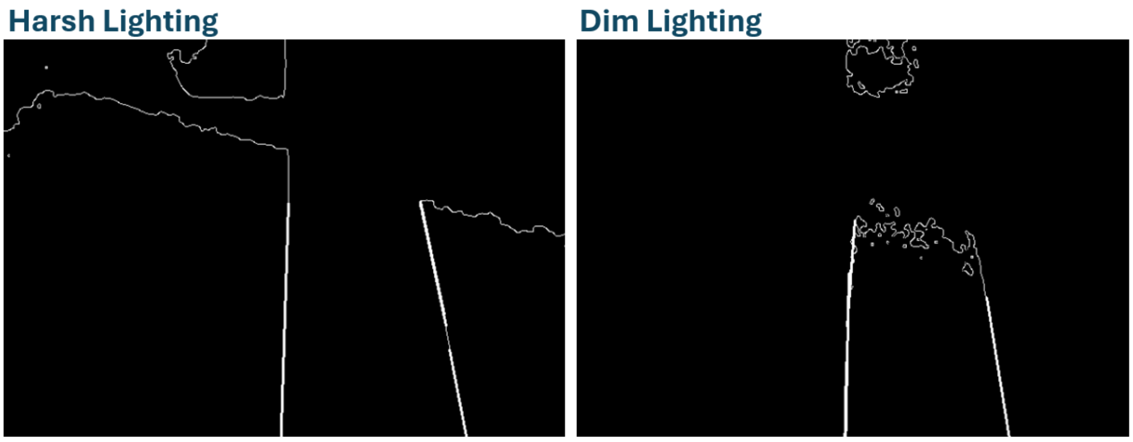
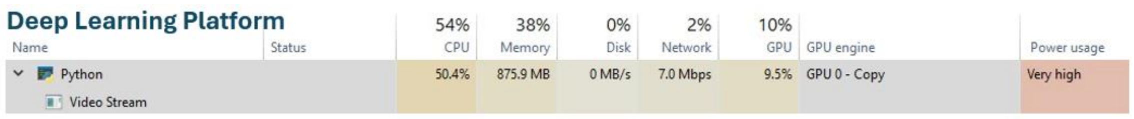


Figure 7.5 Effects of lighting conditions on the line-tracking algorithm.

7.2.2 Analysis of Computational Load

Measurements of the computational load on both the Raspberry Pi and the deep learning platform were measured to monitor their processing capabilities and total computational usage. These measurements were performed using the task manager applications on each device, which display details regarding CPU, GPU, and memory usage.



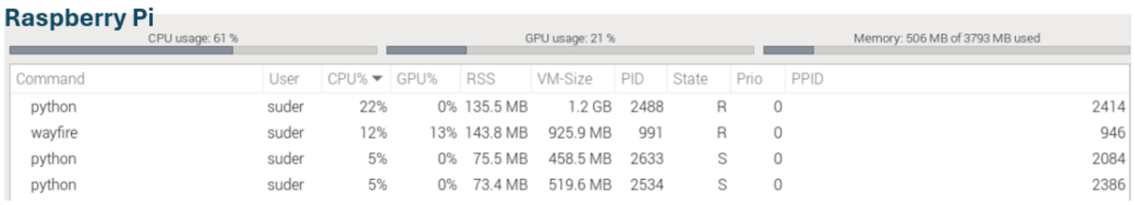
Deep Learning Platform

Name	Status	54% CPU	38% Memory	0% Disk	2% Network	10% GPU	GPU engine	Power usage
Python		50.4%	875.9 MB	0 MB/s	7.0 Mbps	9.5%	GPU 0 - Copy	Very high
Video Stream								

Figure 7.6 Performance metrics of deep learning platform.

The results of the deep learning platform computational usage were lower than expected, showing that only 50.4% of CPU resources and 9.5% of GPU resources were occupied by the models during inference. This is positive, indicating further computation could be performed off the embedded system and larger, more accurate models could be used. Network consumption was 7.0 Megabytes per second, which is a reasonable amount and demonstrates the suitability of the system for smaller network connections. Power consumption is also indicated at very high usage levels, which can be estimated between 100-200 watts power draw.

Performance on the Raspberry Pi was similarly adequate, with lower-than-expected readings for all usage statistics. CPU usage varied, staying below 70% usage, indicating room for further additions to the software algorithms. GPU and memory usage stayed well below their maximums, which is ideal but less impactful than CPU usage as it is less vital for software performance.



Raspberry Pi

CPU usage: 61 % GPU usage: 21 % Memory: 506 MB of 3793 MB used

Command	User	CPU%	GPU%	RSS	VM-Size	PID	State	Prio	PPID
python	suder	22%	0%	135.5 MB	1.2 GB	2488	R	0	2414
wayfire	suder	12%	13%	143.8 MB	925.9 MB	991	R	0	946
python	suder	5%	0%	75.5 MB	458.5 MB	2633	S	0	2084
python	suder	5%	0%	73.4 MB	519.6 MB	2534	S	0	2386

Figure 7.7 Performance metrics of Raspberry Pi.

7.3 Conclusions

To conclude analysis of the system and the refinements made, a brief overview of the full system details is necessary. The figure below demonstrates the core project stage at completion. The full system capabilities include simultaneously executing the line-tracking algorithm, hosting the webcam feed for the deep learning platform, performing deep-learning computation on the webcam stream, and ultimately returning the processed stream to be viewed on the Raspberry Pi in real-time. Further project enhancements will be additional to the key aims of the project, but the current core state represents achievement the key goals defined in the project scope.

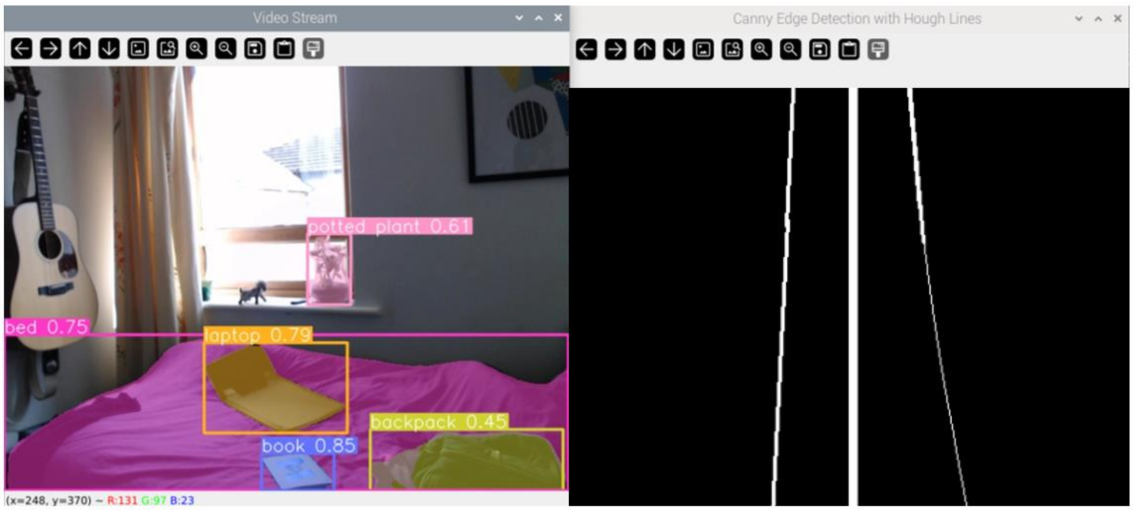


Figure 7.8 Model (YOLOv8x-seg) and line tracking functioning simultaneously.

Findings from the evaluation of the deep learning platforms and line following algorithm performance were closely analyzed. Takeaways for the DL platform include measurements of key metrics such as connection latency, frames per second, inference speed, and object detection abilities, which all fell within reasonable ranges. Both FastSAM and YOLOv8 demonstrated efficient inference speeds and accurate object detection capabilities and fell within the desired real-time range. Buffering issues, however, led to a reduction in FPS during frame transmission. The line following algorithms exhibited commendable speed and accuracy, but suffered minor issues related to vehicle turning speed and sensitivity to lighting conditions. Analysis of computational load on the DL platform and Raspberry Pi revealed ample room for additional processing tasks and enhancements, underscoring the effectiveness of the systems while highlighting areas for refinement. Some opportunities for improvement include reducing buffering issues, modifying chassis characteristics to improve vehicle turning, and mitigating lighting issues. The following section will target some of these problems, enhancing the core project scope and adding additional functionality. Through these optimizations, the overall system can be further enhanced to meet the demands of real-world applications, ultimately expanding usability across various applications.

Chapter 8: Enhancements & Additions

Building on the basic core implementation previously discussed, a few further enhancements and additions were made within the remaining project development timeframe. This section introduces several key improvements to the system which aim to elevate the existing systems. First, additional models were integrated, those being MobilenetSSD and YOLOv8n, onto the edge device to provide greater comparative analysis of the function of the DL platform. These models are introduced, and their functionalities explained, followed by a comparison of their performances to provide insights into their effectiveness for on-device processing tasks. Graphs are used to visualize the results of these comparisons and give a clear understanding of their respective performances. Additionally, modifications to the self-driving algorithm are explored, including a driving mode toggle and the introduction of a threshold slider for lighting adjustments. Finally, modifications to the FastSAM implementation to utilize the text prompt feature are made, and its capabilities and drawbacks highlighted. These enhancements and additions aim to further optimize the systems developed and improve their efficiency and versatility for real-world applications while further demonstrating the feasibility of the proposed embedded system deep learning platform design.

8.1 Additional Models on Edge Device

While comparison of the DL platform models was performed, additional analysis and conclusions could be drawn from comparison with models running on the edge device. The basic premise of this objective was to further elaborate on the feasibility, and superiority, of utilizing the DL platform methodology previously described for a range of applications rather than implementing machine learning models on the device itself. To do this, two models aimed at lightweight applications on small systems were chosen, namely MobilenetSSD and YOLOv8-nano (YOLOv8n). Similarly to the other two models, these models were used with the pretrained weights provided by their respective developers and tests performed to analyse computational load and performance.

The YOLO model was first implemented on the Raspberry Pi, running inference directly on the webcam frames in real-time. This model is twofold, providing a version which detects and categorizes classes in a frame, and a version which also segments the detected classes. This model was able to be successfully deployed, and sample results taken on a test environment, with examples seen in the figure below.

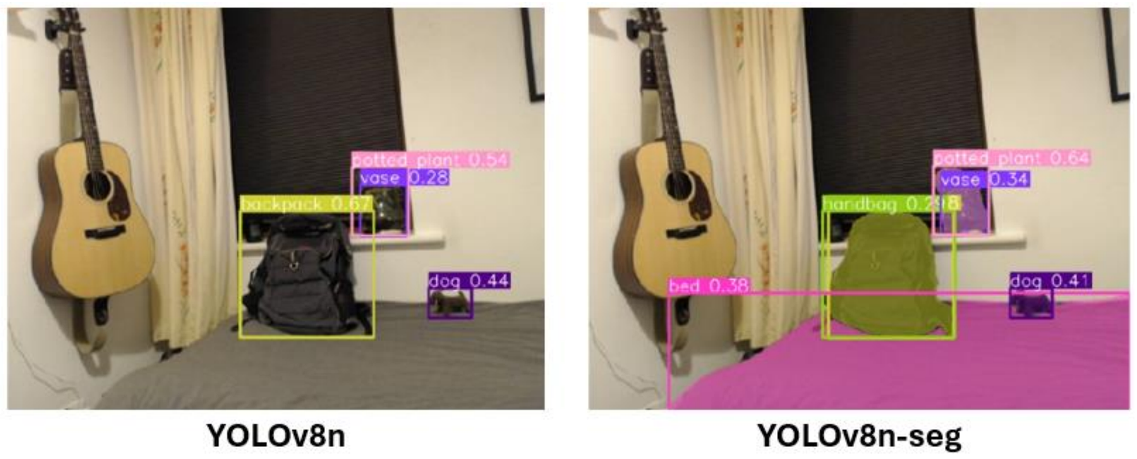
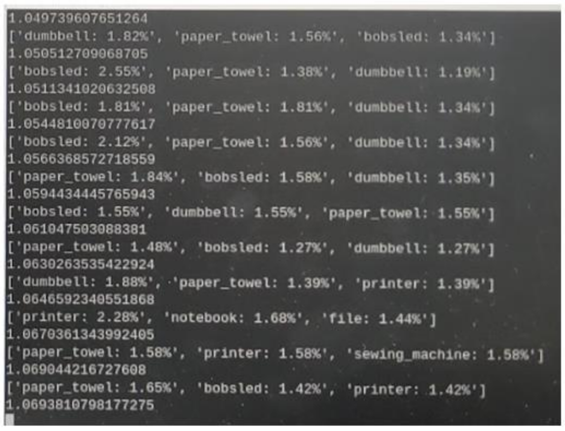


Figure 8.1 Inference results for YOLOv8n and YOLOv8n-seg models.

The rationale for choosing this model was based on its similarity to the deep learning platform as it is a smaller version of the same YOLO version used and thus provides a close comparison in terms of capabilities. It is also a cutting-edge model, typical for what would be commonly used in real-world applications.

A second model was chosen to add another layer of comparison, MobilenetSSD. It is a lightweight convolutional neural network architecture optimized for mobile and embedded devices and designed for object detection tasks. It works by combining the MobileNet architecture with a Single Shot Multibox Detector (SSD) head and offers real-time inference and high performance on resource constrained platforms. Seen in the figure below, the output of the inference is a stream of text in the command line, rather than a display of the processed frames overlaid with bounding boxes. This aims to increase the processing speed of the inference by removing the overhead required for displaying the outputs.

Input Frame

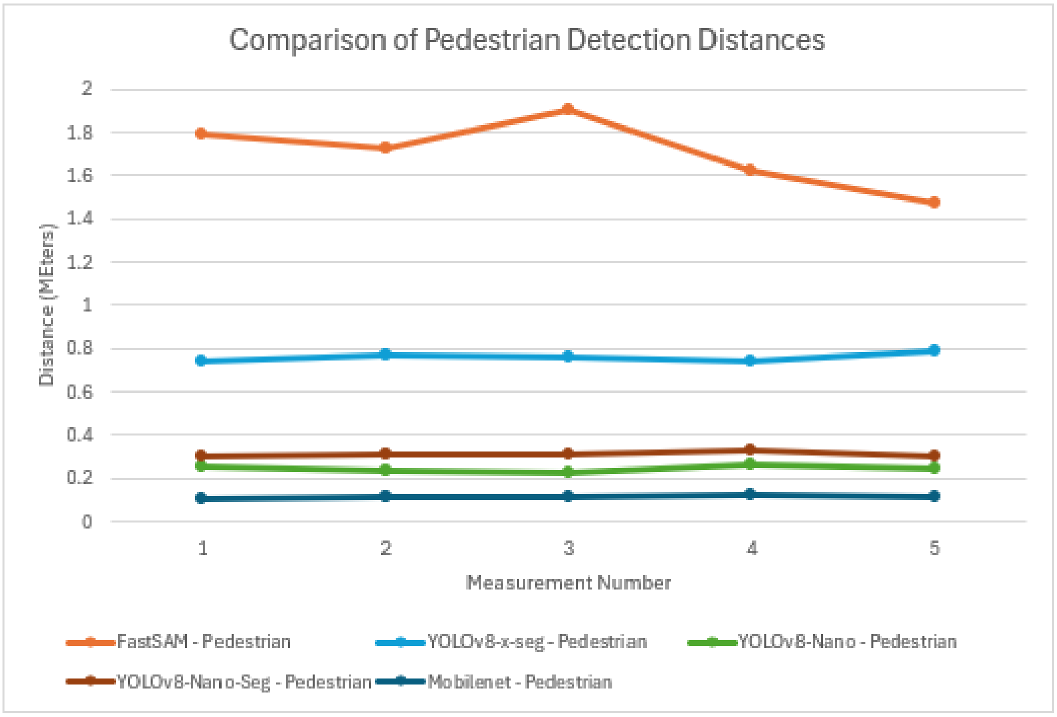


```
1.049739607651264
['dumbbell: 1.82%', 'paper_towel: 1.56%', 'bobsled: 1.34%']
1.050512709068705
['bobsled: 2.55%', 'paper_towel: 1.38%', 'dumbbell: 1.19%']
1.0511341020632508
['bobsled: 1.81%', 'paper_towel: 1.81%', 'dumbbell: 1.34%']
1.0544810070777617
['bobsled: 2.12%', 'paper_towel: 1.56%', 'dumbbell: 1.34%']
1.0566368572718559
['paper_towel: 1.84%', 'bobsled: 1.58%', 'dumbbell: 1.35%']
1.0594434445765943
['bobsled: 1.55%', 'dumbbell: 1.55%', 'paper_towel: 1.55%']
1.061047503088381
['paper_towel: 1.48%', 'bobsled: 1.27%', 'dumbbell: 1.27%']
1.0630263535422924
['dumbbell: 1.88%', 'paper_towel: 1.39%', 'printer: 1.39%']
1.0646592340551868
['printer: 2.28%', 'notebook: 1.68%', 'file: 1.44%']
1.0670361343992405
['paper_towel: 1.58%', 'printer: 1.58%', 'sewing_machine: 1.58%']
1.069044216727608
['paper_towel: 1.65%', 'bobsled: 1.42%', 'printer: 1.42%']
1.0693810798177275
```

Detection Results

Figure 8.2 MobilenetSSD input frame and detection results test.

With edge model setup completed, tests were performed for cross comparison and to contrast abilities versus the DL platform's models. Looking at the distances of pedestrian class recognition, the first graph below shows the performance comparison of the edge models with the DL models. As expected, edge performance is lower, with the YOLO and Mobilenet models recognizing the pedestrian class between 0.1 and 0.35 meters. MobileNet performs the worst of the three edge models, which is to be expected as it is the oldest and least updated model. Comparison of stop sign class detection distances is displayed in the lower graph, with a closer comparison being seen. YOLOv8x-seg performs the strongest, but the YOLO nano version and FastSAM model have similar recognition distances. The Mobilenet model performs the weakest, which is again unsurprising and shows its lack of credibility in distance performance.



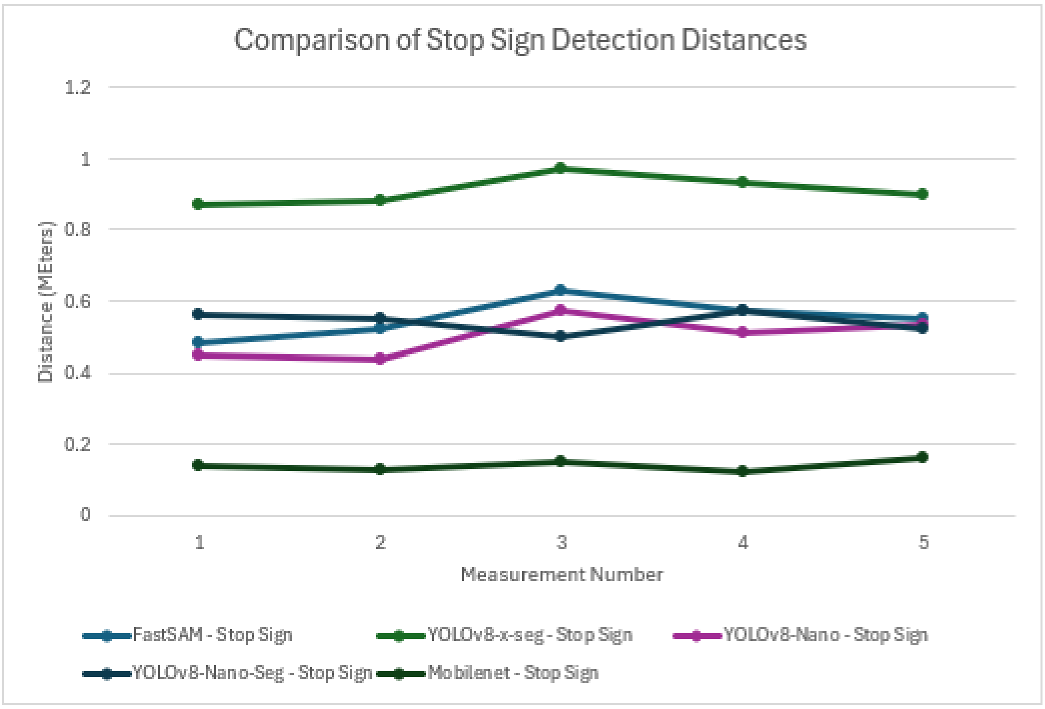


Figure 8.3 Graphs comparing detection distances for pedestrian and stop sign classes.

For further comparison, a table of various features of the models was created to illustrate the benefits and drawbacks of each. Looking first at the connection latency, the edge models have an advantage with no latency due to their local usage. The GPU accelerated models, however, still offer real time speeds by operating under 1000ms. As for FPS, the edge models have a severe disadvantage, running nowhere near real-time speeds with Mobilenet being the only model capable of processing more than one frame per second. This is further illustrated by the inference speeds, taking between two and four seconds for the YOLO edge models to compute a frame and at most one second for Mobilenet.

Model Type	Model	Connection Latency	FPS	Inference Speed	Detection (Stop-sign, person)	Memory Footprint	Energy Requirements & Cost
GPU Accelerated	FastSAM	500-1000 ms	30 FPS	45-50 objects at 8-11ms	Segmenting at 0.55m, 1.7m	Low on-device, medium to large external	High/High – GPU and Server
	YOLOv8-x	500-1000 ms	25-30 FPS	212 ms	32% at 0.91m 28% at 0.76m	Low on-device, medium to large external	High/High – GPU and Server
Edge	Mobilenet	None-Local	2-5 FPS	200-1000 ms	20% at 0.14m 65% at 0.11m	Medium – On device storage	Low/Low – System cost
	YOLOv8-nano	None-Local	<1 FPS	2000-3000 ms	35% at 0.5m 43% at 0.24m	Medium – On device storage	Med/Low - dependent on model size
	YOLOv8-nano-seg	None-Local	<1 FPS	3600-4200 ms	27% at 0.54m 39% at 0.31m	Medium – On device storage	Med/Low - dependent on model size

Table 8.1 Comparison of all features of the DL platform and edge device models.

The detection column reiterates the previous findings on detection distances for the stop-sign and person classes but adds information regarding the confidence levels for the YOLO and Mobilenet models. These results are as expected, showing stronger confidences at further distances for the larger models, with the edge models offering decent confidence levels at slightly lower distances. The memory footprint for each model is also commented on, with the primary takeaways being that models operating on the DL platform offer low on-device memory usage, but medium or large external usage. The edge devices, in contrast, require a moderate level of memory usage on the embedded system but no external usage whatsoever. Finally, the relative system energy requirements and costs are included. These are as expected, with the GPU accelerated models needing high levels of energy to operate and costing significantly more to suit their hardware and networking requirements. Edge model energy usage and cost are generally low but may reach a medium range depending on the desired size of the implemented models.

In essence, the addition of these models on the edge device serves to give a broader understanding of the proposed usage of the external deep learning platform and insight into the prevalence of such a solution. The analysis concludes the superiority of the GPU accelerated models in most inference tasks but highlights the necessity of tailoring the choice of model to the desired system capabilities.

8.2 Modifications to Self-Driving Software

Further enhancements to the core project scope were conducted, including modifications to the self-driving software and algorithms used. While minor improvements, the aim was

to solidify the workings of the vehicle system and create a stronger foundation for the analysis of the other constituent project parts.

First, a toggle to enable or disable the vehicle movements remotely was added. This was a simple adjustment, and its execution required only to adjust a few lines of code to include a Boolean check for the driving state. The state of the driving would be displayed on screen and can be seen in the figure below. This change made further testing easier as the car could be remotely placed on the track before beginning the line tracking.

Another addition to the road tracking GUI and functionality was a slider at the bottom of the output video frame which, when adjusted, would modify the lighting thresholds. These thresholds are used by the edge detection algorithm to determine what levels of light are within the threshold of either black or white. To combat varied lighting conditions in different environments, this slider can be shifted to reach the ideal edge detection settings.

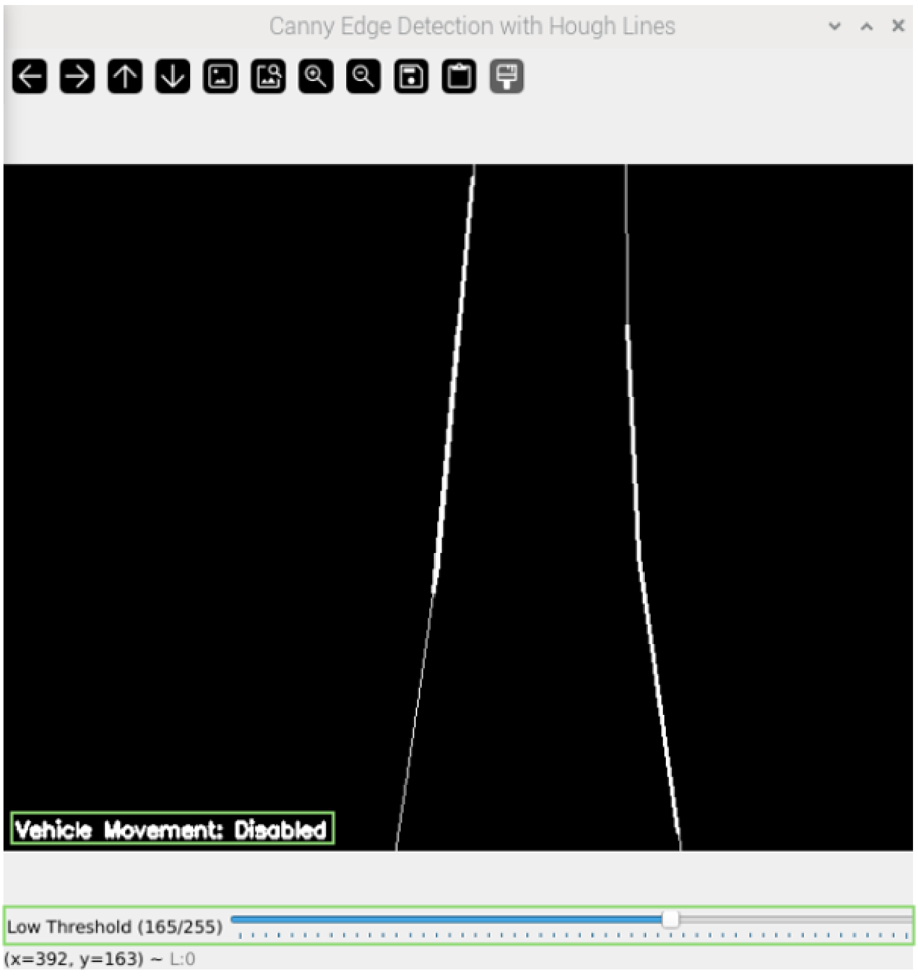

Canny Edge Detection with Hough Lines
Vehicle Movement: Disabled
Low Threshold (165/255)
(x=392, y=163) ~ L:0

Figure 8.4 Photo showing the updated line tracking user interface.

8.3 FastSAM Text Prompt Functionality

Further work was done on the FastSAM model to enable text prompt functionality during real-time inference. Uniquely supported by FastSAM, text prompts can be used during inference to define what objects should be segmented by the model. This is a feature unique to FastSAM and opens the door for cutting edge approaches such as segmenting driving surfaces of autonomous vehicles, segmenting specifical anatomical structures in medical imaging, and numerous other applications. This ability is enabled by CLIP (Contrastive Language-Image Pre-training), a model developed by OpenAI [24], which learns to understand images by associating them with corresponding textual descriptions, thus enabling it to complete a wide range of vision-language tasks with high accuracy.

The original goal was to achieve text prompt use on real-time frames processed by the model, but that proved difficult. Due to a lack of documentation on the FastSAM model, and its relatively short lifespan, there wasn't clear instruction on implementation of this feature. A working version was completed where the text prompts were working on the video feed, but could only be processed at incredibly slow speeds, often taking 10 to 30 seconds for each frame. Near real-time inference speeds should be possible with this method, but only the limited implementation was accomplished during the allotted time frame. Ultimately, this feature could prove highly beneficial for improving applications and with more time would have been possible to implement properly.

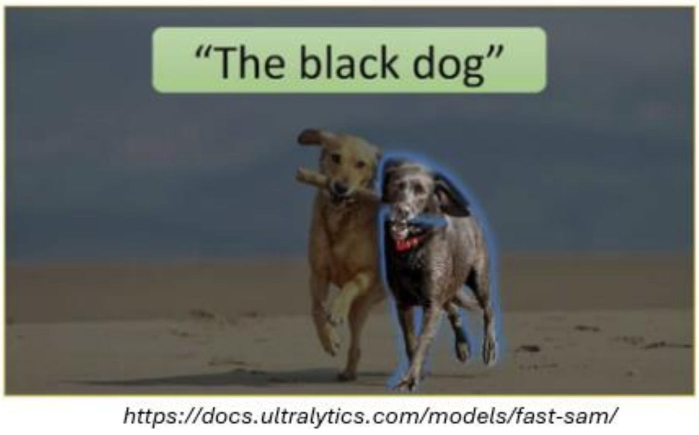


Figure 8.5 Example results of real-time text prompt input implementation for FastSAM.

8.4 Conclusions

In summary, the enhancements and additions made to the core project scope aim to refine the existing systems and broaden the project's scope of analysis. The integration of additional models, namely YOLOv8-n, YOLOv8-n-seg, and MobilenetSSD, onto the edge device provides further comparative analysis with the deep learning platform models, shedding light on their effectiveness for on-device processing tasks, but also highlighting the multitude of benefits gained via use of a DL platform. The chosen graphical comparisons give clear insights into the performance disparities between the edge and DL models, emphasizing the trade-offs in computational load, processing speed, and detection accuracy. Modifications were made to the self-driving algorithm to improve vehicle control and adjust line tracking variables, improving the system's functionality and ease of testing. Additionally, efforts to enable text prompt functionality in the FastSAM model via the CLIP model demonstrate potential for further improvements and applications, although problems exist regarding implementation. These enhancements aim to contribute to a deeper understanding of system capabilities, highlighting the need for tailored model selection based on the specific application, its requirements, and system constraints. Ultimately, further optimization and exploration of these enhancements could yield valuable insights and advancements in embedded systems coupled with DL platforms.

Chapter 9: Conclusions & Future Work

With the completion of the core project objectives, additional work, and refinements completed, project conclusions can be assessed. This section aims to recap the primary discoveries and analysis of the project work as well as suggest future improvements and further work which could be carried out to extend or apply these ideas further.

9.1 Project Conclusions

The findings of this study on feasibility testing of real-time object segmentation and recognition models in small embedded systems shed light on the diverse landscape of inference methodologies and their suitability for various applications. Insights into the process of creating a small autonomous driving embedded system were also gained, with takeaways regarding the use of classical computer vision techniques and the ideal hardware setup being covered.

By leveraging classical computer vision algorithms alongside machine learning-based inference, hybrid autonomous systems can be created to navigate various environments with limited computational resources. Takeaways on designing and implementing the embedded system emphasize the importance of integrating classical computer vision techniques and optimized hardware configurations in the development of small-scale embedded systems. Hardware choice also holds a strong role, with selection of components requiring thorough consideration. Overall, the chosen system proved a success in being an example system for implementing the deep learning platform and was able to complete all desired aims and objectives.

The evaluation of using a remote GPU accelerated system to perform deep learning model inference and computation reveals its commendable performance. This computation over a local network returns pertinent data within 800 milliseconds, lower than the chosen ideal maximum real-time latency of 1000ms. This indicates its aptness for applications necessitating moderate to fast response times. This includes systems such as real-time object detection in surveillance systems or autonomous vehicles, where timely decision-making is crucial. The benefit of integrating an embedded system with this suggested deep learning platform is obvious, but it is imperative to acknowledge the system's inherent dependency on network connectivity. This makes the system susceptible to latency issues and network interference and means any implementations must take this critical factor into consideration and ensure steady and reliable network connections.

Testing of on-device machine learning models was also performed to provide comparison with the DL platform and analyse the current state of machine learning on edge devices. On-device processing emerges as a possible alternative, albeit with certain caveats and drawbacks. It offers robustness and independence from external network dependencies; however, it poses challenges in terms of computational resources and power consumption, particularly on small embedded systems. While there is no network feedback due to their local nature, edge processing computational speeds and abilities are much less capable, with inference speeds under one frame per second for standard models. This underscores the importance of carefully weighing the trade-offs between computational efficiency and system complexity when considering on-device processing solutions. Moreover, on-device processing is expensive in terms of the available computational abilities and power consumption limitations on small systems, which further impact the possibility of implementation on the edge device itself.

Furthermore, the comparative analysis of external GPU-accelerated models and edge models highlighted their respective strengths and weaknesses in different contexts. Best suited for applications with consistent network performance and low interference, a DL platform approach could be used in applications like industrial automation or systems which will remain within a predetermined area with the necessary networks. Conversely, edge models will excel in environments where network connectivity cannot be relied upon and where reliability and robustness are critical. Critical response times for specific systems need to be determined prior to implementing off-device processing, to identify which models and system choices are best suited for the application. With these points considered, selection of the most appropriate inference methodology relies on a thorough understanding of the application requirements and any environmental constraints involved.

9.2 Future Work

Moving forward, future work in this domain should aim to choose a specific application domain, closely analyse its hardware constraints, and design a system tailor made for a real-world application. This entails conducting detailed assessments of task-specific requirements and environmental factors to ascertain the optimal configuration of hardware components and inference methodologies. Applications such as defect identification and removal on production lines could be chosen for deployment of the edge system coupled with DL platform, further demonstrating its feasibility.

Additionally, leveraging domain-specific datasets and fine-tuning techniques could greatly enhance the performance of machine learning models used with the embedded system. Customizing the models to suit the intricacies of the target application could optimize inference accuracy and efficiency, thereby maximizing the utility of these embedded systems in real-world scenarios.

In summary, this study provides insights into the realm of inference methodologies on embedded systems, but there remains much scope for further refinement and experimentation. Through the addressing of the identified challenges and leveraging emerging technologies, it is possible to develop a much more sophisticated embedded system capable of meeting the evolving demands of diverse applications across various industries and domains. The pursuit of future endeavours in this field holds great potential for advancing the frontiers of embedded system design and machine learning integration.

Bibliography

[1] K. Malhotra and Y. Kumar, "Challenges to implement Machine Learning in Embedded Systems," 2020 2nd International Conference on Advances in Computing, Communication Control and Networking

[2] Kirillov, M. (2023). Segment Anything. Meta AI Research.

[3] Ansari, M. (2015). Design and Implementation of Autonomous Car using Raspberry Pi. International Journal of Computer Applications.

[4] Shahane, V. (IEEE). A Self-Driving Car Platform Using Raspberry Pi and Arduino.

[5] Mohan, A. (2019). Autonomous Self-Driving Car using Raspberry Pi Model. IJERT.

[6] Dhavalikar, R. (2018). Traffic Light Detection and Recognition for Self Driving Cars Using Deep Learning. IEEE.

[7] Giripunje, P. (RASPBERRY PI BASED AUTONOMOUS CAR). IJRAR.

[8] Gazis, T. (2021). Examining the Sensors That Enable Self-Driving Vehicles. IEEE.

[9] Day, C. (2019). Pedestrian Recognition and Obstacle Avoidance for Autonomous Vehicles Using Raspberry Pi. Cham Springer.

[10] E. Batzolis, E. Vrochidou and G. A. Papakostas, "Machine Learning in Embedded Systems: Limitations, Solutions and Future Challenges," 2023 IEEE

[11] Zhao, Xu & Ding, Wenchao & An, Yongqi & Du, Yinglong & Yu, Tao & Li, Min & Tang, Ming & Wang, Jinqiao. (2023). Fast Segment Anything.

[12] Ultralytics (2024) YOLO by Ultralytics, Ultralytics YOLOv8 Docs. Available at: https://docs.ultralytics.com/#yolo-a-brief-history (Accessed: February 2024).

[13] Rath, S. (2024) Yolov8: Comprehensive guide to state of the art object detection, LearnOpenCV. Available at: https://learnopencv.com/ultralytics-yolov8/ (Accessed: February 2024).

[14] Freenove, Freenove/freenove_4wd_smart_car_kit_for_raspberry_pi: Apply to FNK0043, GitHub. Available at: https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for_Raspberry_Pi (Accessed: February 2024).

[15] Ultralytics (2024b) Yolov8, Ultralytics YOLOv8 Docs. Available at: https://docs.ultralytics.com/models/yolov8/#supported-tasks-and-modes (Accessed: February 2024).

[16] Meta AI SA-1B Dataset, AI at Meta. Available at: https://ai.meta.com/datasets/segment-anything/ (Accessed: February 2024).

[17] Ultralytics (2024a) COCO Dataset, COCO - Ultralytics YOLOv8 Docs. Available at: https://docs.ultralytics.com/datasets/detect/coco/#key-features (Accessed: February 2024).

[18] noorkhokhar99, Segment anything with webcam in real-time with fastsam, GitHub. Available at: https://github.com/noorkhokhar99/Segment-Anything-with-Webcam-in-Real-Time-with-FastSAM (Accessed: February 2024).

[19] CASIA-IVA-Lab, Casia-Iva-Lab/FASTSAM: Fast Segment Anything, GitHub. Available at: https://github.com/CASIA-IVA-Lab/FastSAM (Accessed: February 2024).

[20] K. Malhotra and Y. Kumar, "Challenges to implement Machine Learning in Embedded Systems,"
2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN), Greater Noida, India, 2020, pp. 477-481, doi: 10.1109/ICACCCN51052.2020.9362893.

[21] Zhang, Xingzhou & Wang, Yifan & Shi, Weisong. (2018). “pCAMP: Performance Comparison of Machine Learning Packages on the Edges.”

[22] Beyond accuracy: Measures for assessing machine learning models, pitfalls and guidelines, Richard Dinga, Brenda W.J.H. Penninx, Dick J. Veltman, Lianne Schmaal, Andre F. Marquand, bioRxiv 743138; doi: https://doi.org/10.1101/743138

[23] Study: Average car size is increasing — will roads still be safe for small cars and pedestrians? Available at: https://www.thezebra.com/resources/driving/average-car-size/ (Accessed: March 2024).

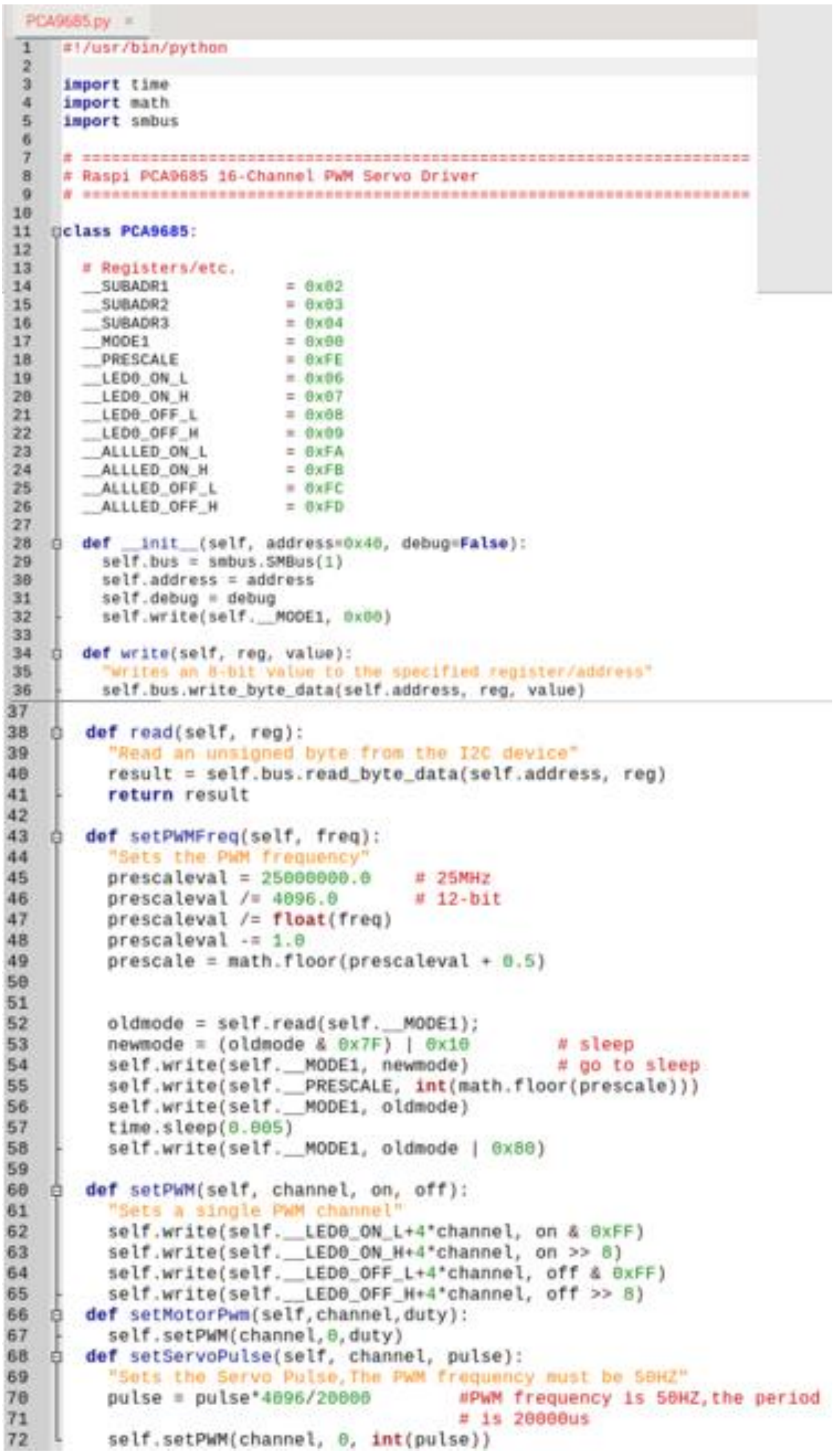
[24] OpenAI (January 2021) Clip: Connecting text and images. Available at: https://openai.com/index/clip (Accessed: March 2024).

Appendix A: GitHub Repository

See the link below for the code repository containing relevant project files.

https://github.com/superKabe/Autonomous-Driving-with-Object-Segmentation

Appendix B: RPI PCA9685 PWM Servo Driver


PCA9685.py

```python
#!/usr/bin/python

import time
import math
import smbus

# ============================================================================
# Raspi PCA9685 16-Channel PWM Servo Driver
# ============================================================================

class PCA9685:

  # Registers/etc.
  __SUBADR1            = 0x02
  __SUBADR2            = 0x03
  __SUBADR3            = 0x04
  __MODE1              = 0x00
  __PRESCALE           = 0xFE
  __LED0_ON_L          = 0x06
  __LED0_ON_H          = 0x07
  __LED0_OFF_L         = 0x08
  __LED0_OFF_H         = 0x09
  __ALLLED_ON_L        = 0xFA
  __ALLLED_ON_H        = 0xFB
  __ALLLED_OFF_L       = 0xFC
  __ALLLED_OFF_H       = 0xFD

  def __init__(self, address=0x40, debug=False):
    self.bus = smbus.SMBus(1)
    self.address = address
    self.debug = debug
    self.write(self.__MODE1, 0x00)

  def write(self, reg, value):
    "Writes an 8-bit value to the specified register/address"
    self.bus.write_byte_data(self.address, reg, value)

  def read(self, reg):
    "Read an unsigned byte from the I2C device"
    result = self.bus.read_byte_data(self.address, reg)
    return result

  def setPWMFreq(self, freq):
    "Sets the PWM frequency"
    prescaleval = 25000000.0    # 25MHz
    prescaleval /= 4096.0       # 12-bit
    prescaleval /= float(freq)
    prescaleval -= 1.0
    prescale = math.floor(prescaleval + 0.5)


    oldmode = self.read(self.__MODE1);
    newmode = (oldmode & 0x7F) | 0x10        # sleep
    self.write(self.__MODE1, newmode)        # go to sleep
    self.write(self.__PRESCALE, int(math.floor(prescale)))
    self.write(self.__MODE1, oldmode)
    time.sleep(0.005)
    self.write(self.__MODE1, oldmode | 0x80)

  def setPWM(self, channel, on, off):
    "Sets a single PWM channel"
    self.write(self.__LED0_ON_L+4*channel, on & 0xFF)
    self.write(self.__LED0_ON_H+4*channel, on >> 8)
    self.write(self.__LED0_OFF_L+4*channel, off & 0xFF)
    self.write(self.__LED0_OFF_H+4*channel, off >> 8)
  def setMotorPwm(self,channel,duty):
    self.setPWM(channel,0,duty)
  def setServoPulse(self, channel, pulse):
    "Sets the Servo Pulse,The PWM frequency must be 50HZ"
    pulse = pulse*4096/20000        #PWM frequency is 50HZ,the period
                                    # is 20000us
    self.setPWM(channel, 0, int(pulse))
```