COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

ENABLING SELF-CHARGING SMART CARS WITH DEEP LEARNING

Author: Alejandra Malia Sáez

Director: Hao Wang

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Enabling Self-Charging Smart Cars with Deep Learning

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2024/24 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

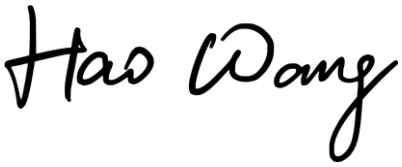
El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Alejandra Malia Sáez Fecha: 20/ 06/ 2025

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO



Fdo.: Hao Wang Fecha: 21/ 06/ 2025

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

ENABLING SELF-CHARGING SMART CARS WITH DEEP LEARNING

Author: Alejandra Malia Sáez

Director: Hao Wang

Madrid

Acknowledgments

I would like to thank Dr. Hao Wang for welcoming me into the IntelliSys Lab, offering me this research project, and guiding me throughout its development. My gratitude also goes to Dr. Kevin Lu, who provided valuable insights and was always willing to help in any way possible. Finally, I am grateful to Stevens Institute of Technology for offering such a wide range of research opportunities that greatly enrich the educational experience and professional development of their students.

DESARROLLO DE AUTOMÓVILES INTELIGENTES CON CARGA AUTOMÁTICA

Autor: Malia Sáez, Alejandra.
Director: Wang, Hao.
Entidad Colaboradora: Stevens Institute of Technology.

RESUMEN DEL PROYECTO

Este proyecto presenta el diseño e implementación de un sistema de vehículo eléctrico autónomo con carga también autónoma utilizando un modelo de navegación basado en aprendizaje profundo y carga inalámbrica estática. Se utilizó un coche robot JetRacer, alimentado por una NVIDIA Jetson Nano, que fue entrenado para seguir de forma autónoma una carretera y navegar hasta una estación de carga cuando los niveles de batería caían por debajo de un umbral. El sistema de navegación emplea una red neuronal ResNet-18 que genera comandos de dirección basados en la entrada de la cámara en tiempo real. El mecanismo de carga utiliza un módulo de transferencia de energía inalámbrica y un sistema de monitoreo de batería integrado mediante sensores INA219. Los resultados demostraron que, tras el entrenamiento adecuado, el sistema completó eficazmente las tareas de navegación y carga autónomas sin intervención humana.

Palabras clave: Vehículos autónomos, Aprendizaje profundo, JetRacer, ResNet-18, Carga inalámbrica, Gestión energética, NVIDIA Jetson Nano, Coche robot con carga autónoma

1. Introducción

Los vehículos eléctricos autónomos (EAV) están transformando el transporte moderno gracias a su promesa de seguridad, eficiencia y sostenibilidad. Sin embargo, persiste una limitación clave: aún se requiere intervención humana para cargar los vehículos eléctricos. Esto interrumpe la autonomía total y limita el potencial de los sistemas sin conductor. Este proyecto aborda este desafío implementando un prototipo de vehículo eléctrico auto recargable utilizando una plataforma de coche robot.

El sistema se construyó con un JetRacer AI Kit, un vehículo autónomo de alto rendimiento y código abierto diseñado para experimentación en inteligencia artificial. En su núcleo se encuentra la NVIDIA Jetson Nano, un microordenador compacto pero potente, diseñado para ejecutar inferencias de redes neuronales y procesar la entrada de la cámara en tiempo real. El JetRacer utiliza una cámara RGB a bordo para recopilar imágenes continuas de la pista. Estas imágenes son procesadas por una red neuronal convolucional profunda—la ResNet-18—que genera comandos de dirección que permiten al coche mantenerse centrado en un recorrido conocido.

Además de la navegación, se integró un sistema de carga inalámbrica estática. El vehículo puede detectar niveles bajos de batería utilizando sensores INA219 integrados en la placa y navegar de forma autónoma hasta una estación de carga. El proceso de carga se realiza mediante un sistema de bobinas transmisor-receptor inalámbrico, permitiendo que el

vehículo se recargue sin necesidad de conexión física. En conjunto, estos sistemas demuestran un ciclo completo de conducción autónoma y recarga autosuficiente.

2. Definición del proyecto

En 2024, las ventas de vehículos eléctricos (VE) superaron los 17 millones a nivel mundial, reduciendo la dependencia del petróleo y promoviendo la movilidad sostenible. Sin embargo, persiste la insatisfacción de los usuarios en torno a la experiencia de carga, en parte por el proceso manual que implica. Abordar este desafío mediante un sistema de recarga automática tiene el potencial de mejorar significativamente la comodidad, la autonomía y la escalabilidad del transporte eléctrico, especialmente en servicios como robotaxis y entregas a domicilio.

El objetivo principal de este proyecto es validar la viabilidad de un VE autónomo auto recargable. Para ello, se construyó y entrenó un coche robot capaz de navegar de forma autónoma mediante aprendizaje profundo y visión por computador, detectar en tiempo real la descarga de la batería, localizar y alinearse con una base de carga inalámbrica, y recargarse de manera inalámbrica mientras está en funcionamiento.

3. Descripción del modelo/sistema/herramienta

El sistema consta de tres componentes principales: un coche robot, un cargador inalámbrico y una pista para la conducción. Tras una amplia experimentación con herramientas y hardware de código abierto, se eligió el JetRacer AI Kit [1] por su fiabilidad, soporte y rendimiento de hardware.

El JetRacer se ensambló siguiendo las directrices del fabricante y se equipó con una NVIDIA Jetson Nano. Se configuró la pila de software para habilitar el acceso a JupyterLab y el control del coche. Tras la calibración inicial, se amplió el algoritmo de seguimiento de la carretera para entrenarlo para la carga autónoma: una vez que el nivel de batería cae por debajo del 30%, el coche activa un segundo modo de navegación para localizar la estación de carga.

Los datos de entrenamiento para el modelo de navegación se recopilaron utilizando una interfaz de conducción manual. Las imágenes de la cámara a bordo se etiquetaron con las posiciones ideales en la pista. Estas imágenes etiquetadas se utilizaron para entrenar un modelo ResNet-18, elegido por su fiabilidad, precisión y tamaño compacto de 43 MB. Aunque se consideraron otras arquitecturas como AlexNet, SqueezeNet y DenseNet, ResNet-18 ofreció un rendimiento superior tanto en velocidad como en precisión.

La última capa de ResNet-18 se personalizó con una capa lineal totalmente conectada de PyTorch para generar las coordenadas (x, y) del punto óptimo en la carretera. El valor x producido por el modelo, que varía entre -1 y 1, representa la desviación lateral respecto al centro de la pista. Se utilizó un controlador Proporcional-Derivativo (PD) para convertir esta desviación en ángulos de dirección:

$$Dirección = K_p * x + (x - last_x) * K_d$$

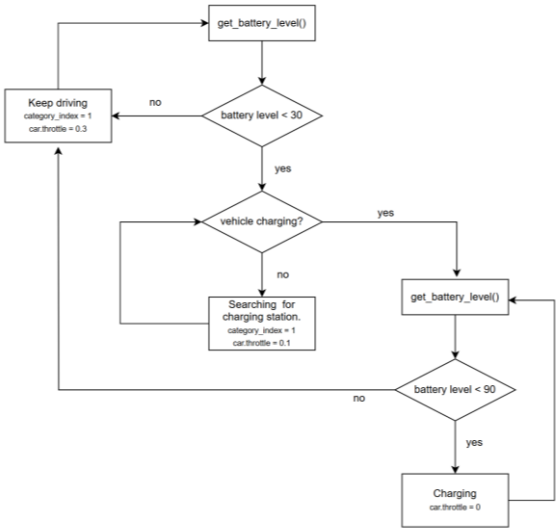
Los valores de ganancia se eligieron mediante pruebas iterativas, siendo $K_p = 1.5$ y $K_d = 8$. Inicialmente, los valores altos de ganancia resultaron extraños, pero posteriormente se comprendió que, dado que la derivada se calcula como una simple diferencia sin dividir por el intervalo de tiempo, un valor alto de K_d compensa el valor pequeño de la derivada debido

a la alta tasa de fotogramas—60 imágenes por segundo. Además, x ya está centrado en torno a cero, por lo que no es necesaria una corrección de desplazamiento (offset). En este caso, se prefiere un controlador PD en lugar de un PID completo (Proporcional-Integral-Derivativo), principalmente para obtener una respuesta más rápida y estable.

Para la carga, se implementó un módulo de transferencia de energía inalámbrica. Se colocó la bobina transmisora del cargador conectado a una fuente de tensión de 24 V en la pista y la bobina receptora se fijó en la parte de abajo del chasis de coche. Cuando están correctamente alineados—a menos de 1 cm de distancia vertical—el sistema suministra 12 V a 3,2 A. Esta configuración permite cargar mientras el coche permanece encendido, extendiendo el tiempo de carga en menos de 2 horas en comparación con la carga por cable.

La monitorización de la batería se realiza mediante el chip INA219, que registra el voltaje y la corriente en tiempo real. La Jetson Nano ejecuta un servidor con una API cuyos endpoint muestran el estado de la batería en tiempo real. Un script de Python independiente consulta este servidor y activa el comportamiento de carga autónoma cuando es necesario.

Para entrenar el vehículo para la carga autónoma, se añadió una nueva categoría a la base de datos denominada “searching_charging_station”. Esto permite que el modelo distinga entre el comportamiento normal de seguimiento de la carretera y la navegación hacia la estación de carga.



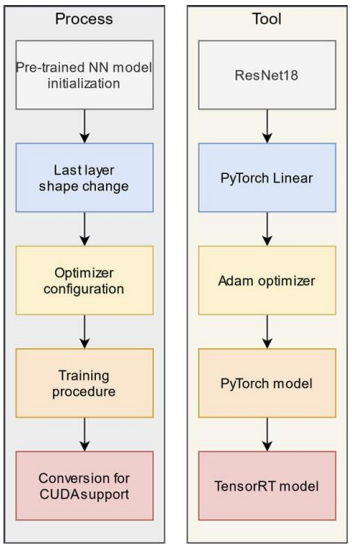


Figura 1: Diagrama de flujo del algoritmo de carga *Figura 2: Diagrama de bloques para la conducción autónoma [3]*

4. Resultados

El modelo ResNet-18 habilitó de manera efectiva el seguimiento de la carretera. Con 20 vueltas de entrenamiento, el coche recorría la pista correctamente, aunque ocasionalmente se salía del carril levemente. Tras aumentar a 50 épocas, el coche se mantuvo dentro de los límites y logró un movimiento suave y centrado.

La integración del comportamiento de carga requirió un entrenamiento adicional. Se recopilaron más imágenes cerca de la base de carga, desde distintos ángulos, para ayudar

al modelo a alinear correctamente el coche con la base. Esto fue fundamental para maximizar la eficiencia de transferencia de energía y evitar pérdidas.

A pesar de estos logros, se planificaron mejoras adicionales que no se implementaron por limitaciones de tiempo. Entre ellas se incluyen la detección de objetos para mejorar la localización del cargador y el desarrollo de una estación de acoplamiento física en la que el coche pueda entrar, para facilitar una alineación constante.

5. Conclusiones

Este proyecto demuestra con éxito un prototipo de vehículo eléctrico autónomo de carga inalámbrica utilizando la plataforma JetRacer. Mediante una combinación de aprendizaje profundo, procesamiento de imágenes en tiempo real y carga inductiva, el vehículo fue capaz de conducirse de manera autónoma y recargarse sin intervención humana.

Los resultados sugieren un gran potencial para desplegar sistemas similares en vehículos autónomos reales, especialmente en contextos como robotaxis, logística e infraestructura urbana inteligente. El trabajo futuro podría centrarse en integrar fusión de sensores, mejorar la localización y aumentar la robustez del hardware para preparar el sistema ante entornos más complejos.

6. Referencias

[1] JetRacer AI Kit: JetRacer AI Kit, AI Racing Robot Powered by Jetson Nano

[2] INA219 data sheet INA219 data sheet, product information and support | TI.com

[3] Kacper Podbucki, Tomasz Marciniak. “Aspects of autonomous drive control using NVIDIA Jetson Nano microcomputer”, IEEE Catalog, 2022 Aspects of autonomous drive control using NVIDIA Jetson Nano microcomputer | IEEE Conference Publication | IEEE Xplore

ENABLING SELF-CHARGING SMART CARS WITH DEEP LEARNING

Author: Malia Sáez, Alejandra.
Supervisor: Wang, Hao.
Collaborating Entity: Stevens Institute of Technology

ABSTRACT

This project presents the design and implementation of a self-charging autonomous electric vehicle system using a deep learning-based navigation model and static wireless charging. A JetRacer robot car, powered by an NVIDIA Jetson Nano, was trained to autonomously follow a road and navigate to a charging station when battery levels dropped below a threshold. The navigation system employs a ResNet-18 neural network that outputs steering commands based on real-time camera input. The charging mechanism uses a wireless power transfer module, and a battery monitoring system integrated via INA219 sensors. Results showed that after appropriate training, the system effectively completed autonomous navigation and charging tasks without human intervention.

Keywords: Autonomous Vehicles, Deep Learning, JetRacer, ResNet-18, Wireless Charging, Energy Management, NVIDIA Jetson Nano, Self-Charging Robot Car

1. Introduction

Autonomous electric vehicles (EAVs) are transforming modern transportation through their promise of safety, efficiency, and sustainability. However, one key limitation persists: human intervention is still required to charge electric vehicles. This interrupts full autonomy and undermines the potential of driverless systems. This project addresses this challenge by implementing a self-charging electric vehicle prototype using a robot car platform.

The system is built using a JetRacer AI Kit, a high-performance open-source autonomous vehicle designed for AI experimentation. At its core is the NVIDIA Jetson Nano, a compact yet powerful microcomputer tailored for running neural network inference and processing real-time camera input. The JetRacer uses an onboard RGB camera to collect continuous images of the track. These images are processed by a deep convolutional neural network—specifically ResNet-18—which outputs steering commands that enable the car to stay centered on a predefined path.

In addition to navigation, a static wireless charging system was integrated. The vehicle can detect low battery levels using onboard INA219 sensors and autonomously navigate to a charging station. The charging process occurs via a wireless transmitter-receiver coil setup, allowing the vehicle to recharge without being plugged in. Together, these systems demonstrate a complete loop of autonomous driving and self-sufficient charging.

2. Project Definition

With EV sales surpassing 17 million globally in 2024, reducing dependence on oil and promoting sustainable mobility, the need for fully autonomous EV solutions is greater than ever. However, user dissatisfaction persists around the charging experience, amongst other factos, the manual process involved. Addressing this with a self-charging system has the

potential to significantly enhance convenience, autonomy, and scalability of electric transportation, especially in robotaxis and delivery services.

The primary goal of this project is to validate the feasibility of self-charging autonomous EV. This was achieved by building and training a robot car to navigate autonomously using deep learning and computer vision, detect battery state in real-time, locate and align with a wireless charging pad, and recharge wirelessly.

3. System and Model Description

The system consists of three main components: a robot car, a wireless charger, and a track. After extensive experimentation with open-source tools and hardware, the JetRacer AI Kit was chosen for its reliability, support, and hardware performance.

The JetRacer was assembled following manufacturer guidelines and equipped with an NVIDIA Jetson Nano. The software stack was set up to enable JupyterLab access and car control. After initial calibration, the road-following algorithm was extended: once the battery level drops below 30%, the car activates a second navigation mode to locate the charging station.

The training data for the navigation model was collected using a manual driving interface. Images from the onboard camera were labeled with ideal track positions. These labeled images were used to train a ResNet-18 model, chosen for its reliability, accuracy, and compact 43 MB size. While other architectures like AlexNet, SqueezeNet, and DenseNet were considered, ResNet-18 offered superior performance in both speed and accuracy.

The last layer of ResNet-18 was customized with a fully connected PyTorch linear layer to output x, y coordinates of the optimal point on the road.

The x value output by the model, ranging from -1 to 1, represents the lateral deviation from the track center. A Proportional-Derivative (PD) controller was used to convert this deviation into steering angles:

$$Steering\ value = K_p * x + (x - last_x) * K_d$$

The gain values were chosen through iterative testing, being $K_p = 1.5$ and $K_d = 8$. Initially, the high gain values were surprising, but later it was understood that since the derivative is calculated as a simple difference without dividing by the time step, a high K_d compensates for the very small value of the derivative due to the high frame rate—60 frames per second. Also, x is already centered around zero, so no offset correction is needed. In this case, a PD controller is preferred over a full PID (Proportional-Integral-Derivative) controller, mainly to get a faster and more stable response.

For charging, a wireless power transfer module was implemented. A 24V transmitter pad was placed on the track, and the receiver coil was attached under the car. When aligned correctly—within less than 1 cm vertical distance—the system delivers 12V at 3.2A. This setup allows charging while the car remains powered on, extending charging time by less than 2 hours compared to wired charging.

Battery monitoring is achieved via the INA219 chip, which tracks voltage and current in real-time. The Jetson Nano runs a display server that exposes API endpoints showing live

battery statistics. A separate Python script queries this server and triggers the self-charging behavior when necessary.

To train the vehicle for this mode, a new category— "searching_charging_station"—was added to the dataset. This enables the model to distinguish regular road-following behavior from charging navigation.

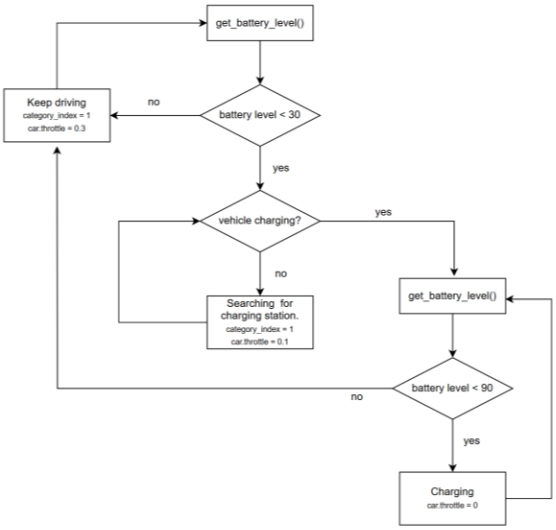


Figure 1: Flow chart self-charging algorithm

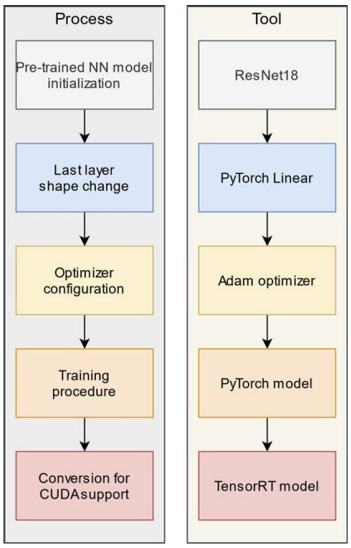


Figure 2: Block diagram for autonomous driving [3]

4. Results

The ResNet-18 model effectively enabled road following. With 20 training epochs, the car could drive along the track but occasionally crossed lane boundaries. After increasing to 50 epochs, the car consistently remained within bounds and maintained smooth, centered motion.

Integrating charging behavior required further training. More images near the charging pad were collected to help the model align the car correctly with the pad. This was critical to maximize power transfer efficiency and prevent misalignment.

Despite these achievements, further enhancements were planned but not implemented due to time constraints. These include object detection for improved charger localization and the development of a physical docking station to aid consistent alignment.

5. Conclusions

This project successfully demonstrates a prototype for a self-charging autonomous electric vehicle using the JetRacer platform. Through a combination of deep learning, real-time image processing, and inductive charging, the vehicle was able to drive autonomously and recharge itself without human input.

The results suggest strong potential for deploying similar systems in real-world autonomous vehicles, especially in contexts like robotaxis, logistics, and smart urban infrastructure. Future work could focus on integrating sensor fusion, improving localization, and enhancing hardware robustness to prepare the system for more complex environments.

6. References

[1] JetRacer AI Kit: JetRacer AI Kit, AI Racing Robot Powered by Jetson Nano

[2] INA219 data sheet INA219 data sheet, product information and support | TI.com

[3] Kacper Podbucki, Tomasz Marciniak. “Aspects of autonomous drive control using NVIDIA Jetson Nano microcomputer”, IEEE Catalog, 2022 Aspects of autonomous drive control using NVIDIA Jetson Nano microcomputer | IEEE Conference Publication | IEEE Xplore

Table of Contents

List of Figures

List of Tables

Chapter 1. INTRODUCTION

The development of electric autonomous vehicles (EAVs) has transformed the transportation industry into a more efficient, safe and sustainable option. Advancements in artificial intelligence and the Internet of Things have enabled significant progress in safe autonomous driving systems. Cameras and sensors get real-time data that is processed to provide decision-making capabilities to the vehicle and control the systems of the car, i.e. the braking or the steering adjustment. To ensure security and reliability, self-driving cars need uninterrupted battery supply. Current approaches to energy management often rely on manual intervention. The vehicle informs the operator that the battery is low, and a person must plug the car into the power supply. So much for an autonomous car.

To address this challenge, the present project aims to design and implement a self-charging system for autonomous vehicles, applying deep learning techniques to achieve smooth navigation. The system will enable the vehicle to autonomously detect low battery levels, analyze real-time road conditions, and identify the nearest charging station. Using a combination of deep learning algorithms, the vehicle will navigate efficiently to the changing station.

This project focuses on building a real-world testbed that consists of battery-powered robot cars equipped with essential hardware components, including a minicomputer such as a Raspberry Pi, an NVIDIA Jetson chip, an RGB camera and the necessary control systems. The system aims to improve the performance of smart cars in terms of navigation and charging process to achieve their autonomy and reliability.

1.1 PROJECT MOTIVATION

The motivation for this project arises from the need to address critical challenges in the field of autonomous driving electric vehicles (EVs). Autonomous driving technology has made remarkable progress in recent years, with advancements in artificial intelligence, sensor suite, and real-time data processing enabling vehicles to navigate urban environments, recognize traffic signals, and interact safely with pedestrians and other vehicles. Various companies have integrated vehicle's autonomy in different ways: Tesla has developed semi-autonomous cars; companies like Waymo, Cruise, Apollo-Go and Argo.ai offer fully autonomous robotaxis for ridesharing; Aurora and Pony.ai produce fully autonomous vehicles for trucking and logistics. These companies have demonstrated the feasibility of fully autonomous driving in controlled environments, and regulatory frameworks in several countries are beginning to accommodate pilot programs and limited deployment of self-driving cars on public roads.

However, despite these technological leaps, the issue of charging remains a significant barrier to achieving full autonomy. Currently, most charging processes require human intervention to plug in the vehicle, which disrupts the driverless experience promised by autonomous EVs. Addressing this challenge is essential to realizing the potential of autonomous electric mobility, as it would allow vehicles to operate continuously without human oversight.

To eliminate the need for human intervention during charging, a promising solution is the integration of wireless charging systems into the pavement. These inductive charging technologies allow vehicles to recharge simply by parking over designated spots, without requiring any physical connection. This approach preserves the autonomous operation of electric vehicles by allowing them to recharge without manual intervention, supporting more efficient operation during scheduled charging stops for robotaxis and autonomous logistics. This type of charging achieves 80% to 90% efficiency compared to traditional charging cable tools. Charging, therefore, takes longer than if the car was plugged in. For example, fully

charging a battery pack should take around one-hour using cables and 1.11 hours using wireless charging (at an efficiency of 90%) [20].

Several pioneering projects have already demonstrated the feasibility of wireless charging for electric vehicles. For instance, IndyGo, the public transit agency in Indianapolis, has implemented wireless charging for its electric buses, enabling them to charge their batteries while waiting at bus stops and at the end of the bus route. In Sweden, Volvo conducted a pilot test embedding wireless charging pads into the road surface, allowing specially equipped electric vehicles to charge while parked. Additionally, Qualcomm Halo has demonstrated dynamic wireless charging, where vehicles can receive power while driving over specially equipped road segments, further enhancing the potential for uninterrupted autonomous operation. They implemented the dynamic wireless charging road technology at a 100-meter test track in Versailles, France.

This bachelor's thesis aims to integrate these two advanced technologies: autonomous driving and static wireless charging, enabling electric vehicles to operate independently while minimizing the need for human intervention during charging stops, enhancing the autonomy of the EVs.

Moreover, the growing demand for sustainable and intelligent transportation systems has highlighted the importance of optimizing computational efficiency too. Deep learning techniques already provide a useful solution for addressing these challenges, as they allow for real-time analysis of sensor data, road conditions and energy usage. By utilizing deep learning algorithms, this project will try to contribute to the development of smart vehicles capable of making autonomous decisions.

Even though the project focuses on the implementation of this technology in smart cars, the potential applications of this self-charging technology extend beyond autonomous vehicles to other forms of transportation such as drones. Drones, which are widely used for delivery services, surveillance or monitoring, face similar challenges related to energy depletion. By adapting the proposed self-charging system, drones could autonomously detect low battery levels, locate nearby charging hubs, and navigate efficiently to recharge. This capability

would significantly enhance the autonomy and reliability of drone operations, particularly in remote or inaccessible areas.

Chapter 2. DESCRIPTION OF THE TECHNOLOGIES

2.1 AUTONOMOUS DRIVING

There are six levels of vehicle driving autonomy defined by SAE J3016 standard, which establishes a six-level classification system for driving automation, ranging from no automation to full autonomy:

- Level 0: No automation
- Level 1: Driver Assistance
- Level 2: Partial Automation
- Level 3: Conditional Automation
- Level 4: High Automation
- Level 5: Full Automation

In level 0 of automation, the human driver performs all aspects of driving, so there is no assistance provided by the system, although the car may issue warning. In the next level, the vehicle assists with either steering or speed management, but not both, and the driver must be ready to take control at any time. Automated features at this stage can include Adaptive Cruise Control (ACC), Automated Parking Assistance, and Lane Keeping Assistance (LKA). In partial automation (level 2) the vehicle controls both steering and acceleration/deceleration, but the driver must remain engaged and always monitor the environment to detect objects and events. Level 3 allows the driver to safely turn attention from driving tasks in certain known, controlled environments, such as freeways, but must be ready to take over if alerted by the system.

Level 4's high automation enables the vehicle to operate autonomously in specific conditions without requiring the driver to monitor the environment; if the system cannot handle a situation, it will safely stop or reroute on its own. However, the driver must still activate the

system when conditions allow. Finally, at Level 5, complete automation is achieved—the vehicle can perform all driving tasks under all conditions—no human driver is required.

2.2 IMPLEMENTATION OF TESTBED

The successful implementation of this project relies on a combination of resources and technologies. In terms of hardware, robot cars were constructed and equipped with wireless chargers. The robot cars used in the testbed are designed for experimental autonomous driving, supporting features such as line following, object detection using its onboard camera, which assigns it to level 3 of automation ("Conditional Automation"), since the car can autonomously control both steering and acceleration in a controlled environment, but human oversight is required for safety and validation.

The robot cars used in this project are manufactured by Waveshare, which is a company specializing in the design and production of robotic kits, among other electronic solutions. Two types of robot cars were used: the PiRacer AI Kit and the JetRacer AI Kit. Both vehicles are designed to support open-source autonomous driving projects, the details of which are elaborated in the following sections.

2.2.1 PIRACER AI KIT

The PiRacer AI Kit consists of all necessary accessories to build an autonomous AI racing robot powered by a Raspberry Pi 4 that supports the opensource DonkeyCar project for deep learning and self-driving.

Figure 3: PiRacer AI Kit [1]

2.2.1.1 PiRacer Kit components

The kit includes all mechanical elements of the chassis, bumpers, camera arm and wheels. The steering system is controlled by the MG996 servo that delivers 9 kg/cm torque at 4.8 V connected to the two front wheels. The dual gearmotor rear wheel drive is provided by two 37-520 motors with 1:10 reduction rate and has a 740 RPM idle—or no load—speed. The motors are designed to run efficiently and safely when supplied with 12-volt DC power. The car also comes with a 0.91inch OLED display with a resolution of 128x32 pixels that can be programmed to show information about the use of the computer and the battery of the car.

Figure 4: PiRacer/JetRacer Display parameters

In this project, the display shows the following parameters: the current IP address, the current power consumption of the system, the voltage being supplied to the car, the current being drawn by the system—in figure 4, the car is not moving and the motors are not engaged, so there is little or no current being drawn—, the battery percentage, and, lastly the GPU, CPU, RAM, and disk usage percentage.

The PiRacer's 12.6 V power supply is provided using three rechargeable Li-Ion 18650 batteries connected in series.

A highly integrated expansion board manages communication between the minicomputer and the car. It ensures that the multiple battery cells are charged evenly, maintaining battery health and maximizing runtime. The board uses sensors (like the INA219 chip) to monitor the battery's voltage and current in real-time. It includes circuits to protect the batteries and the components. Additionally, it provides the circuitry to control the DC motors and the servo for steering and speed control.

Lastly, the car is equipped with an OV5647 camera with a resolution of 5 megapixels—the camera sensor can capture images made up of approximately eight million pixels—and a wide angle of 160 degrees.

2.2.1.2 Minicomputer – Raspberry Pi 4

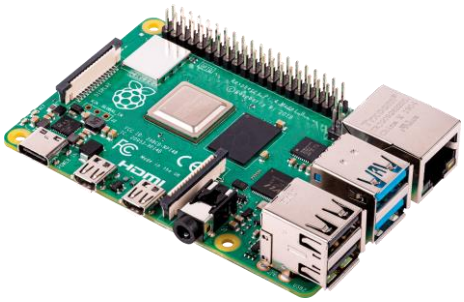


Figure 5: Raspberry Pi 4

The PiRacer AI Kit is powered by a Raspberry Pi 4. This Raspberry Pi model features a BCM2711B0 64-bit 1.5GHz quad-core processor and comes equipped with 4GB of RAM. Its HDMI outputs support 4K resolution, enabling high-resolution dual display setups. Networking capabilities are enhanced with gigabit Ethernet, Bluetooth 5.0, and dual-band WiFi, providing faster and more reliable connectivity. The board includes two USB 3.0 ports and two USB 2.0 ports and is power via a USB-C connector.

The following table summarizes the specifications of the raspberry pi 4:

GPU	Videocore IV 500 MHz
CPU	Quad-core BCM2711B0 64-bit 1.5GHz
RAM	4 GB
Operating system	Raspbian

Table 1: Raspberry Pi 4 Specifiations

2.2.1.3 DonkeyCar Platform

Donkey is an open-source self-driving car platform for remote-controlled cars written in Python, chosen for its simplicity and accessibility [4]. It supports various kinds of autopilots including those based on neural networks, computer vision and GPS. Donkey is most used in Raspberry Pis but is also supported by NVIDIA Jetson PCs.

DonkeyCar employs the following frameworks to enable the car's self-driving behavior:

- TensorFlow [5] is an open-source machine learning and artificial intelligence framework from Google. TensorFlow is used for deep learning, applications such as image recognition and word processing such as natural language processing or speech recognition.
- OpenCV [6] (abbreviation for Open Computer Vision) is a free program library with algorithms for image processing and computer vision. It is written in the programming languages C, C++, Python and Java and is available as free software under the terms of the Apache 2 License.
- TensorRT [7] is a machine learning framework. It was released by NVIDIA to use AI algorithms on their hardware. TensorRT has been optimized to run on NVIDIA GPUs. This is probably the fastest way to execute algorithms.
- Keras [8] enables the rapid implementation of neural networks for deep learning applications. It is an open-source library written and can be used together with frameworks such as TensorFlow.
- PyTorch [9] is currently one of the most popular frameworks for developing and training neural networks. It is characterized above all by its high flexibility and the possibility of using standard Python debuggers. There is no need to compromise on training performance.

2.2.2 JETRACER AI KIT

The second robot type to be utilized is the JetRacer, a high-performance AI-powered car built on the NVIDIA Jetson Nano platform. The self-driving ability can be enabled either by using the DonkeyCar platform or by installing Waveshare's image for JetRacer. See reference [3] for a detailed description of how to setup the software.

Figure 6: JetRacer AI Kit [2]

2.2.2.1 JetRacer AI Kit components

The kit features the same mechanical, steering, and drive components as the PiRacer, but it is equipped with a more powerful camera, specifically, the Sony 8 megapixels camera IMX219 with 160-degree field of view and a resolution of 3280x2464. The JetRacer also includes a dual mode wireless NIC (AC8265), supporting both 2.4 GHz and 5 GHz WiFi as well as Bluetooth 4.2, providing high-speed WiFi connectivity, a stable Bluetooth connection and low latency.

2.2.2.2 Minicomputer – NVIDIA Jetson Nano B01

The central unit is a NVIDIA Jetson Nano [10] responsible for controlling the steering, drive and camera systems as well as processing data and training the model for enabling the self-driving behavior of the robot car.

This microcomputer was designed mainly for use in tasks related to artificial intelligence, so it allows running many neural networks, ideal for working with images.

The Jetson Nano has a quad-core ARM Cortex-A57 CPU and a 128-core NVIDIA GPU. Other important technical specifications are presented in the following table:

GPU	128-core Maxwell
CPU	Quad-core ARM A57 1.43 GHz
RAM	4 GB 64-bit LPDDR4
Connectivity	M.2 Key e.g. for network card
Operating system	Linux for Tegra – JetCard 4.5.1

Table 2: Jetson Nano B01 Specifications

2.2.3 INA219

Waveshare's robot kits include an onboard INA219 chip specifically for monitoring battery voltage and charging current in real-time. By continuously monitoring battery voltage and current, the INA219 helps prevent over-discharge, over-current, and short circuits. In this project, the INA219 is also used to detect when the battery level is low and signal the car to drive to the charging station.

The INA219 chip functions as a current shunt and power monitor with a maximum accuracy of 0.5%. It measures the voltage drop across a small resistor—a shunt—in series with the battery to determine the current flow. The integrated chip senses across shunt on buses that can vary from 0 V to 26 V. It uses a single +3 V to -5.5 V supply, drawing a maximum of 1mA of supply current. It offers programmable calibration, filtering, and averaging, which can be adjusted via software to optimize performance for specific use cases.

The INA219 communicates with the main controller (here, a Raspberry Pi or a Jetson Nano) via an I2C interface with 16 programmable addresses [11].

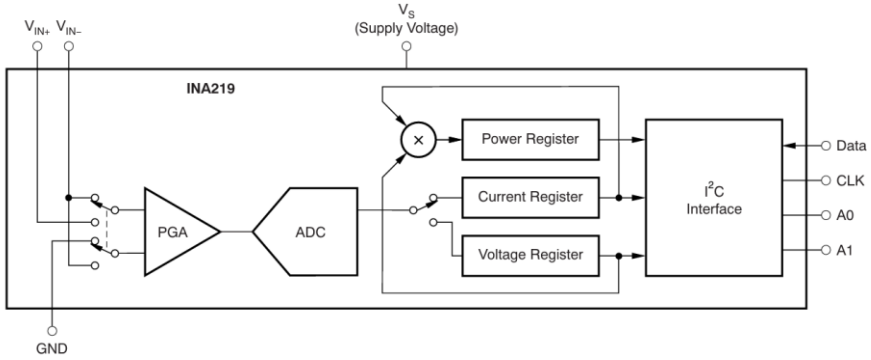


Figure 7: Block Diagramm of the INA219 [11]

The INA219 has a shunt resistor—around 0.1 Ω or 0.01 Ω—connected between the pins V_{IN+} and V_{IN-}. The differential amplifier measures the voltage difference between V_{IN+} and V_{IN-}—the voltage drop across the shunt. The voltage drop is proportional to the current flowing, so the ADC converts the shunt voltage into a 12-bit number. At the same time, the chip also measures the bus voltage—the voltage between V_{IN+} and GND—to calculate the power, which it gives in milliwatts. The current, voltage and power values are stored in internal registers than can be accessed via I2C bus [12].

The car uses 18650 batteries, and the voltage of every battery is 3.7 V. Generally, the voltage per battery is 4.2 V when fully charged. The power of EV is down when the voltage of the whole system is similar to 9V (it is not accurate), Waveshare recommends charging the batteries if the voltage displayed is lower than 10V.

This is how the battery level has been determined in the project:

$$Battery\ Percentage = \frac{Current\ Voltage - Min.Voltage}{Max.Voltage - Min.Voltage} * 100$$

Battery discharge isn't perfectly linear, so this method provides only an approximate state of charge. Although not perfectly accurate, this method is practical for this robotic application, where precise battery monitoring is less critical than in commercial EVs.

2.2.4 THE WIRELESS CHARGING STATION

The system uses wireless power transfer modules, each consisting of two coils—one for transmission and one for reception. The transmitting coil is connected to a 24 V voltage source. The receiving coil is positioned at the bottom of the car, and the wires from the receiving board are connected to the vehicle's charging input. The coils have an outer diameter of 82 mm and an inner diameter of 30 mm. The transmitter module board measures 17 mm by 30 mm, while the receiver module board measures 30 mm by 54 mm.

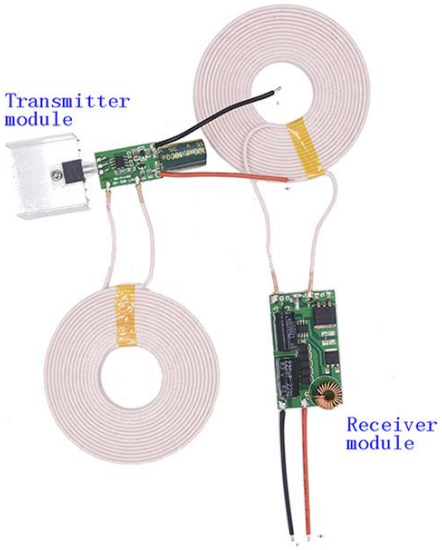


Figure 8: Wireless Power Transfer Module

The input voltage of the module is 24 V, and the output power is 36 W. The output voltage remains constant, but the output current depends on the distance between the transmitter and receiver modules. The table below illustrates the relationship between distance of the transmitting and receiving modules of the charger and the output voltage and current:

Output voltage	Output current	Distance
12 V	3.2 A	at 5 mm
12 V	3.1 A	at 12 mm

12 V	3 A	at 13 mm
12 V	2.6 A	at 14 mm
12 V	2.3 A	at 16 mm
12 V	2 A	at 17 mm
12 V	1.6 A	at 20 mm
12 V	1.4 A	at 23 mm
12 V	1.2 A	at 24 mm
12 V	1.1 A	at 25 mm

Table 3: Relationship between distance of charger modules and output voltage and current

2.3 RESNET-18

The ResNet-18 architecture is a specific version of the Residual Network (ResNet), a deep convolutional neural network architecture developed by Microsoft in 2015 to tackle the vanishing gradient problem and improve the training of very deep networks.

The vanishing gradient problem in neural networks happens when the network has many layers, and the gradients used to update weights become exponentially smaller, especially those in the early layers—as they propagate backward through layers. This means that the first layers learn very slowly or not at all, making it hard for the network to improve its performance. This causes poor convergence and suboptimal solutions, so the algorithm is not able to train deep networks effectively [31].

ResNet solves this problem by using skip connections—or residual connections—which bypass nonlinear layers by adding input of a layer directly to its output. For a residual block,

the ouput is $F(x) + x$, where $F(x)$ is the transformation by weight layers. So, instead of just passing information from one layer to the next, ResNet adds shortcuts that let information (and gradients) skip over one or more layers. This makes it much easier for the network to pass important signals backward through all the layers during training, so even the first layers can keep learning effectively. In ResNet, during backpropagation, the gradient can flow directly through the skip connection without passing through nonlinear transformations. This avoids repeated multiplication of small derivatives [32].

In simple terms, these shortcuts help prevent the important updates from getting lost or becoming too small, allowing very deep networks to be trained successfully.

Figure 9 provides a visual representation of the ResNet-18 architecture.

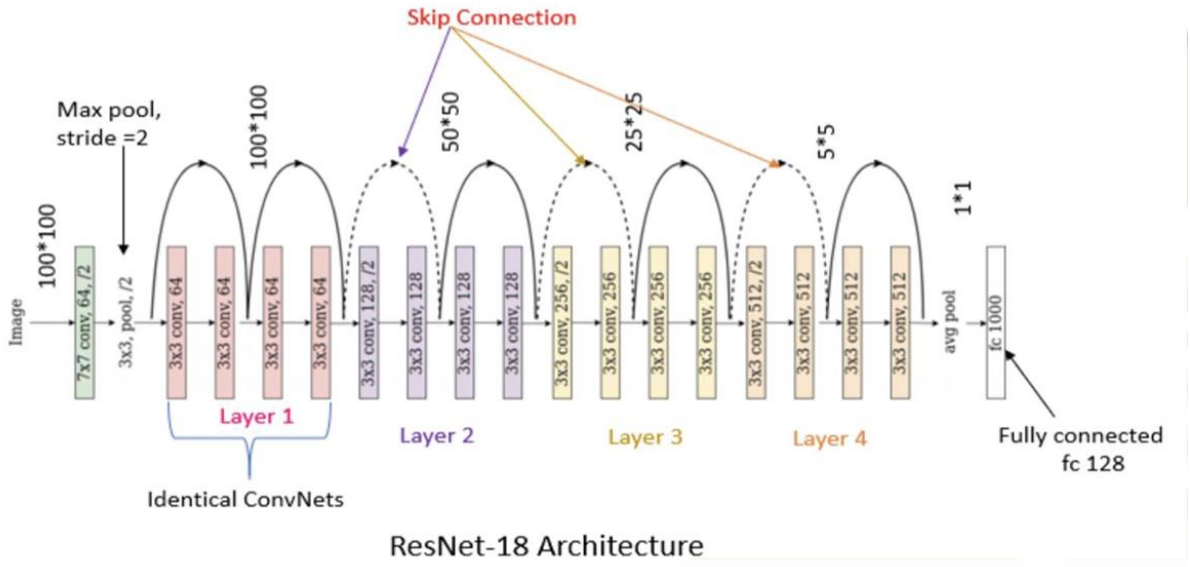


Figure 9: The ResNet-18 architecture [30]

Chapter 3. STATE OF THE ART

This chapter analyses the current technologies developed for autonomous driving and self-charging mechanisms.

3.1 AUTONOMOUS DRIVING TECHNOLOGIES

Autonomous vehicles use multiple sensors like cameras, radar, LiDAR, GPS, and inertial measurement units to understand the environment. Cameras capture images to detect and classify objects and to measure distances, aiding in, for example, lane-keeping and traffic sign or pedestrian recognition. Radar uses radio waves to detect the distance, speed and size of objects, working in all weather and lighting conditions. This technology is crucial for cruise control and collision avoidance. Light Detection and Ranging (LiDAR) emits laser pulses to generate 3D maps of the surroundings, allowing for object detection, and spatial awareness. GPS provides precise global positioning, enabling accurate navigation and route planning. Inertial Measurement Units (IMUs) measure acceleration and angular velocity to track movement and orientation.

The collected data from these sensors is then combined and used as data input for the algorithm. By combining the data from different sensors, the system can compensate for each sensor's weaknesses.

3.1.1 LIDAR (LIGHT DETECTION AND RANGING)

LiDAR is a remote sensing technology that uses laser pulses to measure distances between a vehicle and an object and recognize the object. As shown in figure 10, the system uses a laser transmitter in 905-1550 nm region of near-infrared or short-wave infrared wavelengths. The receiver detects the reflected light to compute the time-of-flight (ToF), which is the time between emission of the ray (t_1)—which gets partially reflected by an object back to the sensor—and the detection of this reflection (t_2). The system derives the distance between

LiDAR and target from the estimated ToF. Advanced LiDARs measure distances and create high-resolution 3D maps of the environment.

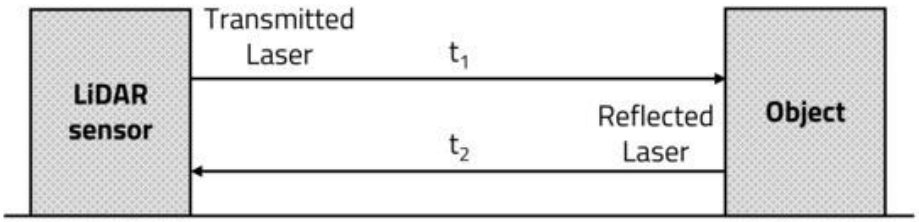


Figure 10: Time-of-flight (ToF) of laser [15].

The laser of the LiDAR sensor can be classified into a pulsed (Time-to-Digital Converter or TDC), amplitude-modulated continuous-wave (AMCW), and frequency-modulated continuous-wave (FMCW) according to the transmission principle. These are the main techniques to estimate ToF and range:

- Time-to-Digital Converter (TDC) LiDAR:
  The TDC uses a short pulse with high peak power and a typical duration on the order of the nanoseconds. The principle of operation of this technique is to measure the time between the emission of a new pulse and the reception of the corresponding reflection, thus performing a direct ToF measurement. This method offers sufficient precision and a simple and easily scalable principle of operation [13].

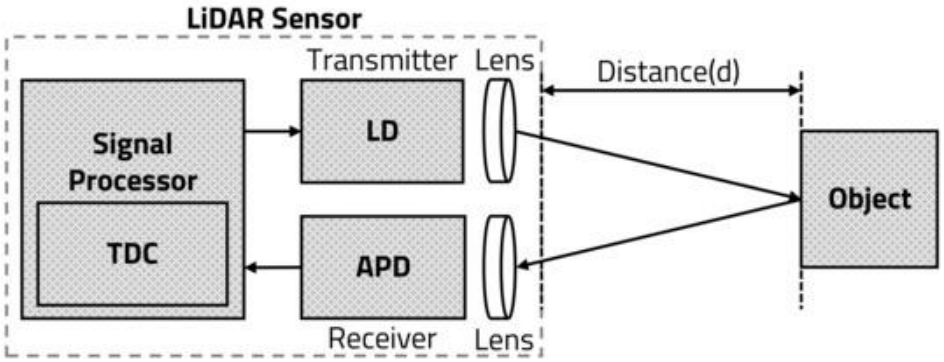
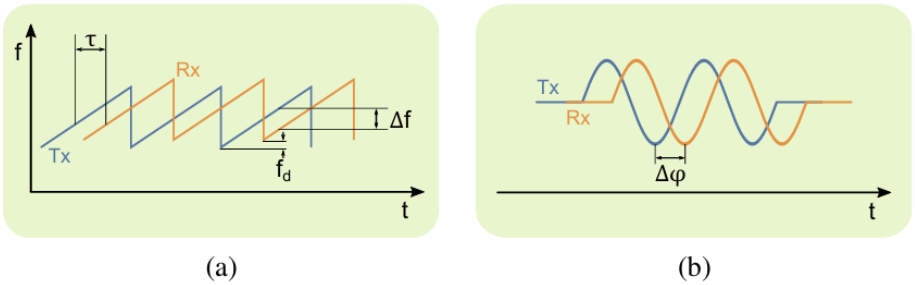


Figure 11: Structure and operation of conventional TDC LiDAR sensor [14].

A TDC LiDAR sensor consists of a laser diode (LD), an avalanche photo diode (APD), a time-to-digital converter (TDC), and a signal processing unit (Figure 11). The LD transmits the laser, which is focused through a light transmitting lens. The ray is reflected from the object and received by the APD through the light-receiving lens. The TDC measures the difference between the time the LD transmits the laser and the time the APD receives it and the TDC calculates the ToF. The signal-processing unit converts the computed ToF to the distance between the LiDAR sensor and the object [14].

- Full-waveform LiDAR: not only takes information about some time instants where certain conditions are observed but instead sample the complete received waveform. The main drawback of this approach is that the currently available analog-to-digital converters (ADCs) needed to sample the received waveform are costly for high sampling rates (greater than 1 giga samples per second) and result in greater power consumption than TDCs.
- Amplitude-Modulated Continuous-Wave (AMCW) is a full-waveform LiDAR. The idea behind this technique is to sample the complete received waveform, instead of taking information about time instants like the TDC does. The distance is measured by comparing phases of the transmitted and backscattered detected waves (See Figure 12b). This type of system is designed to operate in very low range and indoor scenarios. AMCW offers low hardware requirements, since low bandwidth and less power demanding waveforms are being used, typically sinusoidal or square [14].
- Frequency-Modulated Continuous-Wave (FMCW) uses a continuously emitted laser beam whose frequency is modulated. The system emits a laser beam whose frequency changes over time. The light reflects off objects and returns to the sensor, where it is mixed with a portion of the original emitted signal. Then, a beat frequency analysis is conducted, where the difference in frequency between the emitted and reflected signals—the beat frequency—is measured. This frequency difference encodes on the one hand, the distance to the object from the time delay, and, on the other hand, its relative velocity from the measurement of the Doppler Shift.



(a) (b)

Figure 12: (a) FMCW, the channel delay can be obtained from the frequency difference between transmitted and received waveforms and the speed of the target by measuring the Doppler Shift fd; (b) AMCW, the cannel delay is calculated by measuring the phase shift between transmitted and received waveforms. [14]

Each of these LiDAR systems is best suited to specific technologies. TDC works best for systems requiring high-precision time measurement, AMCW excels in short-to-medium range, and FMCW is ideal for scenarios where both distance and velocity data are needed. For this reason, TDC is commonly used in autonomous vehicles or in the medical industry, while AMCW is more suitable for indoor robotics, and FMCW is particularly advantageous for autonomous driving and for the aerospace industry.

3.1.2 CAMERA

Cameras play a central role in autonomous driving by providing the vehicle with real-time visual information about its surroundings, seeking to imitate human vision.

With specialized image sensors and advanced AI analysis, camera systems can detect and recognize objects. Unlike the point clouds of lidar, imaging technology offers object detection and precise image analysis. Specifically, camera sensors detect and classify objects such as vehicles, pedestrians, cyclists or obstacles and depending on the identified object, corresponding decisions are taken. Cameras also identify lane boundaries, road markings, and signs helping the vehicle stay within its lane and follow traffic rules.

More advanced cameras, such as stereo or multi-camera setups, can estimate depth, allowing the vehicle to precisely measure distances and navigate safely. Multi-camera setups around

the vehicle allow for a panoramic view, minimizing blind spots. The limitations of camera sensors rely in the fact that the performance can degrade in case of poor lighting or adverse weather conditions.

They are more accessible and less expensive than radar or lidar, and they are particularly effective at classifying and interpreting. The computational power required to process the data is the camera's drawback. With 30 to 60 frames per second, the most recent high-definition cameras can produce millions of pixels every frame to create detailed imagery. As a result, real-time processing of many megabytes of data is required.

All this collected data is then fed into a control system and processed with deep learning algorithms to real-time decisions about driving actions. Leading autonomous vehicles companies rely primarily on convolutional neural networks for object detection, classification, and lane recognition

3.1.3 RADAR

The Radar sensor emits a radio signal, that gets reflected by objects in the car's surroundings, and is then analyzed to determine the location, speed, and direction of the detected objects using the Doppler effect.

Car radars usually operate in the frequency band from 76 GHz to 81 GHz and typically have lower resolution than LiDAR or cameras, which makes it more difficult to identify small objects or distinguish between similar objects but radar can transmit and receive signals over longer distances than LiDAR, making it useful for detecting objects at greater ranges. For example, long-range radars can detect and measure the speed of objects up to 200 meters away. Also, radar can operate in a wider range of weather conditions compared to LiDAR and cameras [18] [19].

3.1.4 AUTONOMOUS DRIVING TASKS AND SENSOR FUSION TECHNIQUES

The sensory perception system is composed of many subsystems responsible for tasks for the car's self-driving behavior. In each subsystem, a sensing or navigation mechanism—

LiDAR, radar, cameras or GPS—outputs data of a specific type, which is processed with the corresponding computer science technology—such as computer vision, artificial intelligence, or communication technology—to then guide the car.

To combine the ideas and devices from these various technologies, the Internet of Things (IoT) framework is used to perform the necessary tasks.

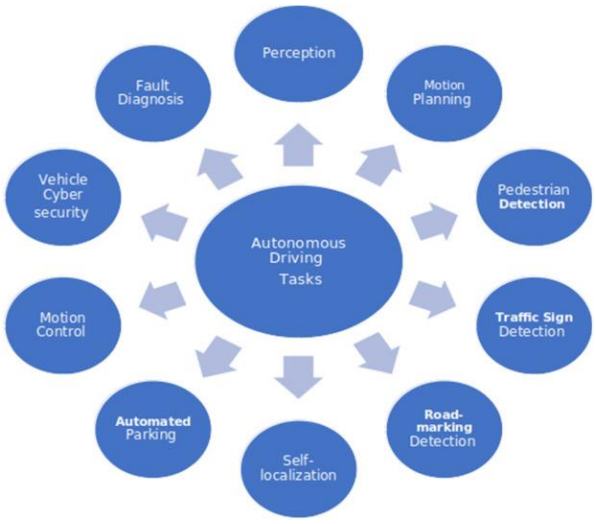


Figure 13: Autonomous driving tasks [16]

Each task is performed by a combination of sensors and specific AI algorithms.

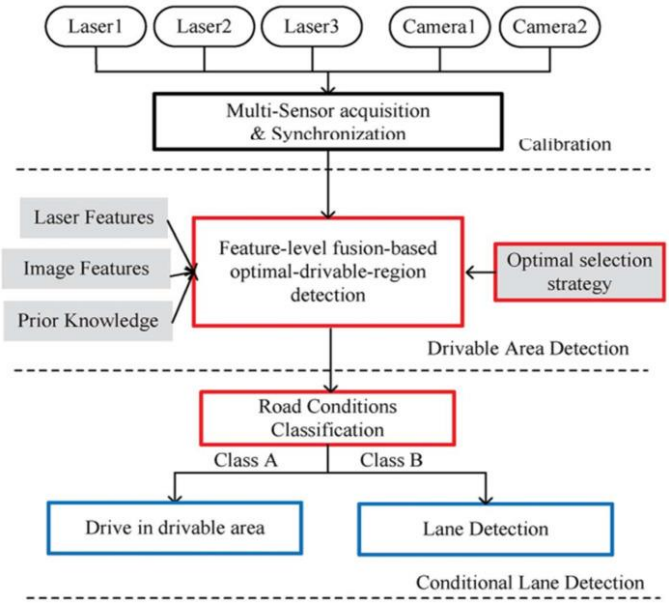


Figure 14: Flowchart for detection algorithm : shows that the input data comprises a 3-Dimensional LiDAR point cloud, camera image, and OpenStreetMap. The key processing stages comprise Region Of Interest, choice in assortment image, lane feature abstraction in bird's eye view of the image, and ego-lane recognition. To do the fitting of lane features to the scientific depiction models, Least-Squares-Method (LSM) and Random-Sample-Consensus (RANSAC) are applied. [17]

The diagram in figure 15 illustrates a sensor fusion framework commonly used in autonomous driving for lane detection. The process begins with data acquisition from multiple sensors—specifically, three LiDAR (laser) sensors and two cameras. To ensure that data from all sensors represent the same moment and coordinate space, a synchronization and calibration stage is implemented.

Following synchronization, the system performs feature-level fusion. In this stage, relevant features are extracted from the laser and image data, along with any prior knowledge available—such as road maps or previously learned models. These features are integrated to detect the optimal drivable region.

Then, the system proceeds to classify the road conditions. The output is typically divided into two broad categories: Class A—Drive in drivable area—and Class B—Lane Detection. Based on this classification, the autonomous system adjusts its behavior accordingly. If the road is classified as Class A, the vehicle continues driving within the identified drivable area.

If classified as Class B, the system engages additional lane detection mechanisms to maintain accurate navigation within narrow or challenging road segments.

3.1.5 AI ALGORITHMS IN AUTONOMOUS VEHICLES BASED ON LEVELS OF AUTOMATION

At lower levels of autonomy (L0-L2), driver assistance systems primarily utilize rule-based and probabilistic methods for tasks like adaptive cruise control or lane detection. Higher levels (L3–L4) rely on machine learning and deep learning—especially convolutional neural networks (CNNs)—for perception tasks like object detection and classification using data gathered by various sensor types. These algorithms are combined with reinforcement learning for complex decision-making and route planning. Full automation (Level 5) requires robust sensor fusion, 3D mapping capabilities, and deep reinforcement learning approaches for adaptive behavior prediction and high-level route planning [16].

3.1.6 TECHNOLOGIES USED IN LEVEL 2 AND 3 OF AUTOMATION

Since the PiRacer and JetRacer are classified as autonomous robot cars of autonomy levels 2-3, the review of current technologies has been focused on these levels.

Level 2 (Partial Automation) vehicles allow for some automated driving features, while still requiring the driver to remain engaged and always have control. 50-70% of the systems use AI algorithms. These AVs rely primarily on camera and radar inputs to perform autonomous tasks, such as navigation assistance, lane keeping, stop-and-go control, and basic environmental adaptation. For detecting objects, lanes, and traffic signs, Convolutional Neural Networks (CNNs), Support Vector Machine (SVM) models, and Decision trees process visual data from cameras. Decision-making tasks like lane change assistance and adaptive cruise control are typically managed by rule-based algorithms. Additionally, Proportional-Integral-Derivative (PID) controllers regulate key self-driving variables—such as speed, acceleration, braking, and steering—through feedback loops to ensure stable behavior. In autonomy level 2, PID controllers are used for basic acceleration and braking adjustments.

For Level 3 autonomy, or Conditional Automation, the vehicle can perform the entire Dynamic Driving Task (DDT) under specific conditions, but the driver must remain engaged and ready to intervene. In this case, 90-95% of the systems use AI algorithms, mostly for perception, self-localization, motion planning and control, and automated parking.

The perception task uses techniques also employed in level 2 of autonomy but incorporates advanced sensors like LiDAR to provide 3D point cloud information.

Self-localization involves approximating the vehicle's location and positioning within a map, which is crucial for control and path planning. Global Navigation Satellite System (GNSS), Inertial Measurement Unit (IMU), and LiDAR supply key data for algorithms like the Kalman filter or the Monte-Carlo-Localization (MCL) model.

Motion planning determines the paths considering other vehicles and impediments. The system takes inputs like the vehicle state and builds an obstacle map. Algorithms include Sampling-based techniques like Rapidly exploring Random Tree (RRT), Artificial Potential Field (APF), Optimal control techniques, and Reinforcement-Learning (RL) models for velocity control, along with hybrid A-star algorithms for impediment prevention.

Motion control involves executing steering and acceleration commands to follow planned paths. It integrates data from GPS, camera, and LiDAR. PID controllers are combined with reinforcement learning for velocity and steering control. Other key algorithms take care of trajectory tracking, energy consumption and navigating complex scenarios like roundabouts. [16]

3.2 SELF-CHARGING MECHANISMS

Major EV manufacturers have developed various solutions to help charge EVs. The solutions are primarily grouped into two categories: static and dynamic charging. In both cases, the transmitting coil is installed under the pavement and the receiver is in the bottom car. The difference between these approaches lies in the charging process. For static charging, the car is parked over the primary coil and remains there until it achieves the desired charging level.

In dynamic charging, the car gets charged while driving, since the primary coil is placed in the pavement at spaced locations. Figure 16 shows an example for dynamic charging.

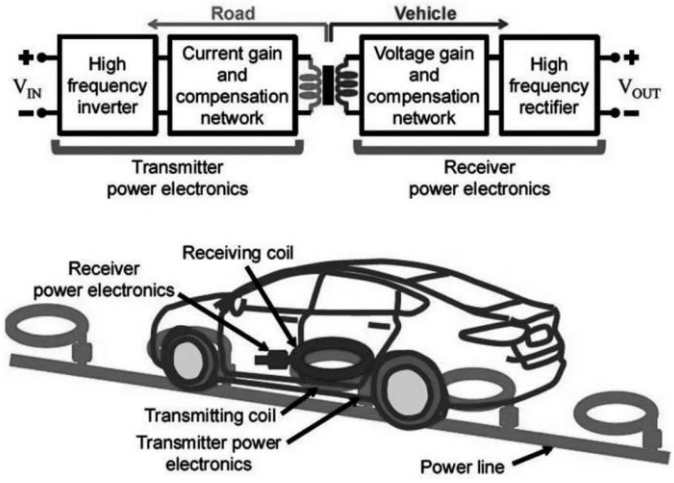


Figure 15: Wireless charging for Automotive [21]

Figure 17 shows a circuit for wireless charging. The square on the left shows the transmitter inverter. A DC source supplies direct current power to the system. Capacitors help filter and stabilize the voltage. Then, a Pulse Width Modulation (PWM) controls the switching of the inverter to produce the desired AC waveform. Lastly, the inverter circuit converts DC to high frequency AC. The three squares in the middle show three types of coupling mechanisms that can be used to transfer energy wirelessly: (a) inductive power transfer (IPT) uses magnetic fields generated by two physically separated coils—this is the most common methos in EV wireless charging; (b) conventional capacitive wireless power transfer (CWPT) uses electric fields between plates to transfer energy across a gap; and (c) magnetic gear wireless power transfer (MGWPT) utilizes coils to transfer energy efficiently across a larger air gap, allowing for more flexibility in alignment. On the right, the receiver circuit consists of a rectifier circuit that converts the received AC power back to DC to charge the battery.

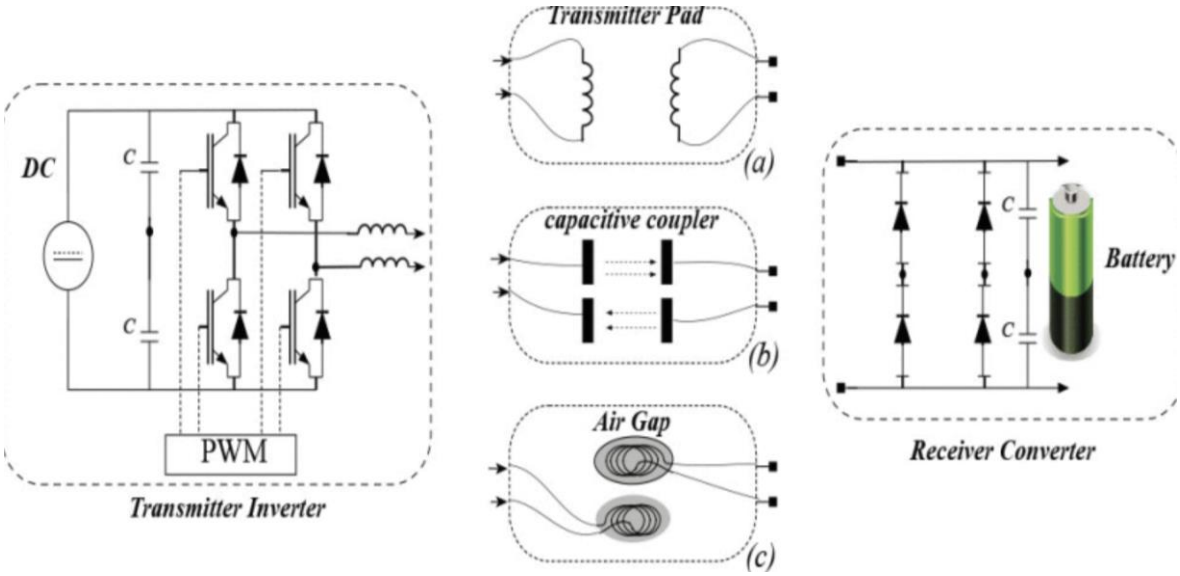


Figure 16: (a) Inductive power transfer (b) Capacitive wireless power transfer, (c) Magnetic wireless power transfer. [20]

Chapter 4. PROJECT DEFINITION

4.1 JUSTIFICATION

According to the International Energy Agency (IEA), global electric car sales exceeded 17 million in 2024, which means that more than 20% of the cars sold worldwide in 2024 were electric. China being leader in the market exceeded the 11 million of electric car sales. [23]

The widespread adoption of electric vehicles has directly reduced global demand for oil by replacing traditional fuel-powered cars with ones that run on electricity. Because there are so many electric cars in use around the world, over 1 million fewer barrels of oil were needed each day for transportation in 2024 than they would have if those cars had used gasoline or diesel instead. [23]

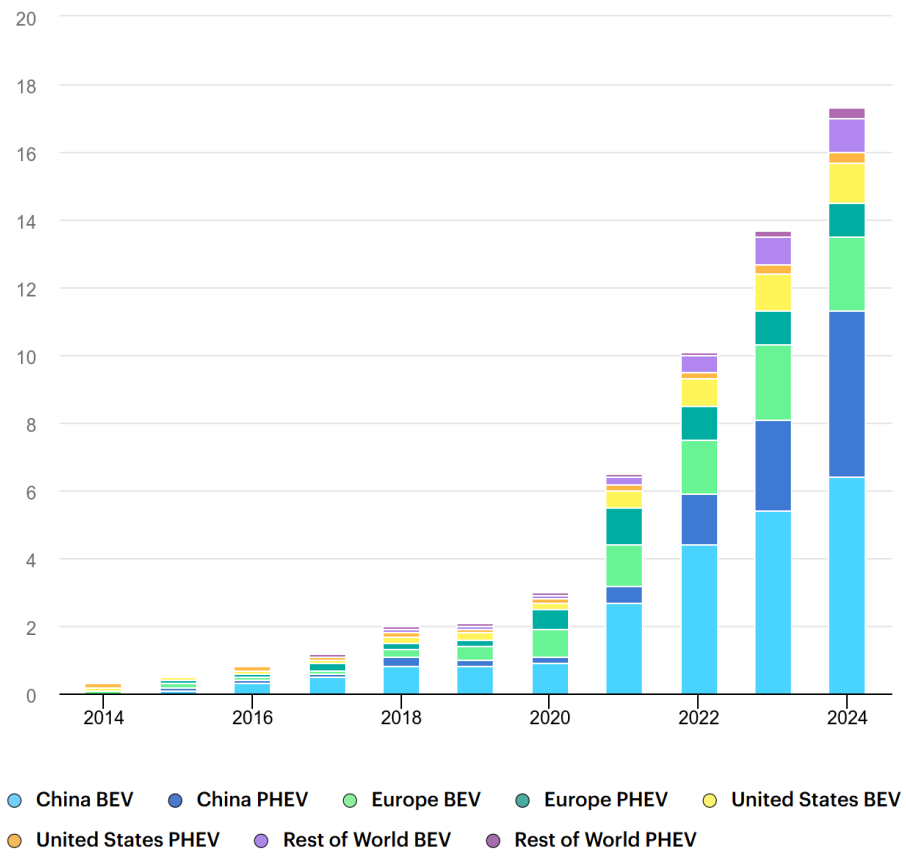


Figure 17: Global electric car sales, 2014-2024, BEV = battery electric vehicle; PHEV = plug-in hybrid vehicle. Includes new passenger cars only. Includes new passenger cars only. IEA (2025), Paris https://www.iea.org/data-and-statistics/charts/global-electric-car-sales-2014-2024, Licence: CC BY 4.0

As figure 18 shows, the stock of electric cars is not spread evenly across the world. In China, around one in ten cars on the road is completely electric, in Europe the ratio is closer to one in twenty, whereas in the US, only one in seventy cars is fully electric. Although this number is low, the sales of conventional (non-electric) cars remained flat, it did not show an increase, while electric car sales increased to 1.6 million in 2024, with the sales share growing to more than 10%. In other words, while electric car sales continued to rise, the demand for traditional gasoline and diesel vehicles stopped increasing and essentially stayed at the same level, showing no meaningful growth. [23]

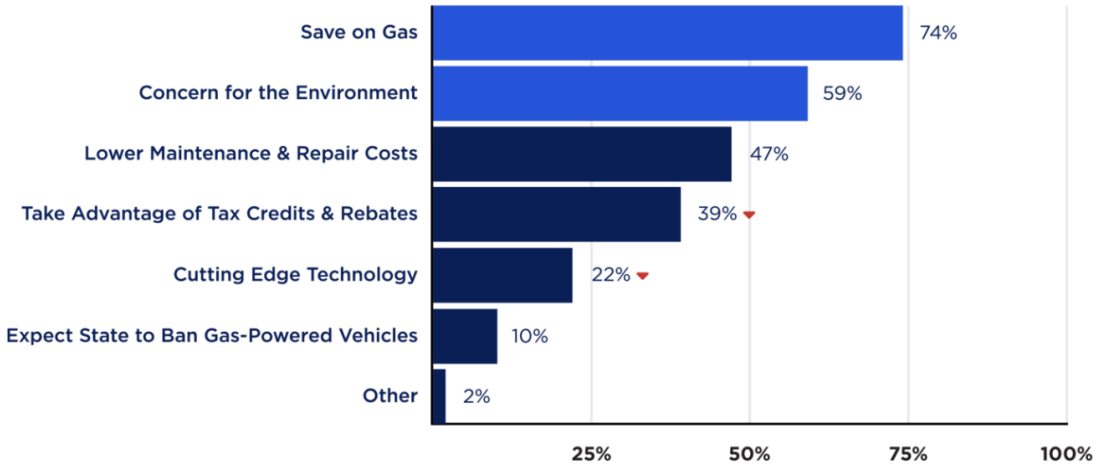


Figure 18: Reasons for Interest in Electric Vehicles in the US in 2025. [25]

Figure 19 shows reasons for interest in EVs in the US in 2025. Respondents cite gas savings, environmental concerns, and an appreciation for the lower maintenance costs associated with EVs. According to AAA Newsroom's 2024 survey on driving cost analysis, EVs had the lowest fuel cost of any vehicle type, based on a national average electricity price of 15.9 cents per kilowatt hour (kWh). EVs also had the lowest maintenance costs among all models [25]. The second most voted reason for changing to EVs is the concern for the environment. Getting drivers to switch from gas-powered to electric vehicles (EVs) is essential for the

U.S. to be carbon-neutral by 2050 [26], and since the price of renewable energy keeps going down, renewable energy is pushing out fossil fuels in the electricity generation market. [24]

The global shift toward electric vehicles is both necessary and inevitable to achieve significant reductions in oil consumption and greenhouse gas emissions. While the pace of EV adoption in the US remains slower than in China or Europe, the steady growth in electric car sales and the stagnation of conventional vehicle demand indicate a clear transition underway. The rapid uptake of EVs in China and Europe demonstrates the viability and benefits of this technology, setting a blueprint for the future of transportation worldwide. As renewable energy becomes increasingly affordable and accessible, electric vehicles will play a crucial role in decarbonizing the transport sector and securing a sustainable future, making their widespread adoption not just desirable, but essential.

A recent study by Transport Focus in the UK found that many electric vehicle users were dissatisfied with the charging process because they had to wait for their cars to charge and were unable to run errands during that time [27]. This project aims to capitalize on the growing EV market and address this issue by introducing autonomous self-charging electric vehicles. The proposed solution enables the car to drop off the user at the mall, supermarket, or office, then drive itself to a charging station while the person completes their errands and finally return—fully charged—to pick up the driver when they are finished.

4.2 OBJETIVES

The objective of the project is to show the effectiveness and practicality of the idea of an autonomous self-charging EV through inductive wireless charging. To do so, a robot car is used, and the vehicle is programmed and trained to drive autonomously, navigate its environment, and locate the charging station when needed. This demonstration highlights how such technology can enable electric vehicles to independently manage their charging process, enhancing user convenience and overall system efficiency.

First, the project involves building a battery-powered robot car equipped with an RGB camera. The vehicle's hardware includes a JetRacer, which was configured and programmed using resources provided by Waveshare's official website and GitHub repository. Careful calibration of the vehicle was conducted to ensure accurate operation and responsiveness to environmental inputs.

The next step was to train the car using neural networks that analyze road images. This training process enabled the car to autonomously navigate its environment through calculated optimal routes and camera inputs. The performance of the neural network was rigorously tested and iteratively optimized.

The integration of energy management constitutes another objective of this project. These algorithms were designed to monitor energy consumption while enabling the vehicle to identify low battery levels and locate nearby charging stations.

4.3 METHODOLOGY AND PLANIFICATION

The plan followed primarily a linear structure with the following objectives: Hardware and Software Setup, Calibration, Model Development for Autonomous Navigation, Model Development for Energy Management, Real-World Testbed Implementation, Performance Analysis and Iteration, and Documentation and Final Report Preparation. Each phase setup, development and implementation phase are followed by rigorous testing and optimization. Tasks are assigned specific time periods with minor overlaps.

Documentation is updated at the end of each optimization phase to include summaries of key insights and outcomes. Once a task is completed and optimized, the plan transitions to the next phase. While the phases are distributed sequentially, the structure allows for iterative improvements to previous phases as needed.

Project Objectives	23/12/2024	30/12/2023	02/01/2025	09/01/2025	16/01/2025	23/01/2025	30/01/2025	06/02/2025	13/02/2025	20/02/2025	27/02/2025	03/03/2025
Hardware and Software Setup, Calibration												
Model Development for Autonomous Navigation												
Testing and Optimization												
Model Development for Energy Management + Testing and Optimization												
Real-World Testbed Implementation												
Performance Analysis and Iteration												
Documentation and Final Report Preparation												

Table 4: Chronogram starting date December 23rd 2024 and end date March 9th 2025

Project Objectives	10/03/2025	17/03/2025	24/03/2025	31/03/2025	07/04/2025	14/04/2025	21/04/2025	28/04/2025	05/05/2025	12/05/2025	19/05/2025	26/05/2025
Hardware and Software Setup, Calibration												
Model Development for Autonomous Navigation												
Testing and Optimization												
Model Development for Energy Management												
Real-World Testbed Implementation												
Performance Analysis and Iteration												
Documentation and Final Report Preparation												

Table 5: Chronogram starting date March 10th 2025 and end date June 1st 2025 (the remaining time was dedicated to documentation)

4.4 ECONOMIC ESTIMATION

The cost and purchase link of the elements used for the project are summarized in table 6. Depending on the number of purchased cars the economic estimation varies. For this

specific testbed, two cars and one charging station were used, so each item on the table was purchased once.

PiRacer AI Kit + Raspberry Pi 4	*$222.99*	*PiRacer AI Kit, Supports DonkeyCar Project, AI Autonomous Racing Robot Powered by Raspberry Pi 4, Deep Learning, Self Driving*
JetRacer AI Kit + Waveshare Jetson Nano developer kit	$362.99	JetRacer AI Kit, AI Racing Robot Powered by Jetson Nano
Wireless Charger	$49.98	Amazon.com: Taidacent 12V 3A High Power Long Distance Wireless Inductive Coil Modules Wireless Charging Coil Wireless Transmitter and Reveiver Coil Wireless Charging Power Supply Module : Electronics
DC 4.0mm x 1.7 mm Male Power Pigtail Cable	$8.99	Amazon.com: COOLM DC 4.0mm x 1.7mm Male Power Pigtail Cable Male Connectors DC Cable 4.0 x 1.7mm Male Power Plug Connector Cable 10pcs : Electronics

Table 6: Price and purchase link of items used for the project

Chapter 5. SYSTEM DESIGN

The system consists of three core components: a robot car, a wireless charger, and a predefined track map. Since the vehicle software used in this project is based on open-source tools and platforms, extensive experimentation was required to identify a functional configuration. Multiple combinations of community-sourced setups and software versions were tested. After numerous iterations and adjustments, the JetRacer AI Kit was selected as the most suitable platform for the implementation.

5.1 VEHICLE CONFIGURATION

First, the car was assembled as indicated in the assembly manual [28]. Then the Jetson Nano was configured with the required software stack as detailed in the sections below. Once the base road-following model was established, the algorithm was extended to include an additional feature: the vehicle would autonomously search for a charging station when the battery level dropped below 30%.

5.1.1 JETRACER SOFTWARE SETUP

The first step involved flashing the JetRacer system image (as referenced in [29]) to a properly formatted microSD card with at least 64 GB of storage. It is recommended to use the Etcher tool for this task. After flashing, the SD card was inserted into the Jetson Nano Developer Kit, which was then powered on.

The next step is to connect the Jetson to the network. The Jetson can either be configured following Step 3 of the Waveshare JetRacer Wiki [3] or, alternatively, connected to a monitor to manually edit the *wpa_supplicant.conf* file (see Appendix II, Section [1]). Once connected, the vehicle's display will indicate the assigned IP address. It is advisable to configure the device for automatic Wi-Fi connection on boot; otherwise, the Wi-Fi password must be re-entered each time the device is restarted (see Appendix II, Section [2]).

Upon startup, the car automatically connects to the Wi-Fi, and launches two *systemd* services:

- The jetcard_display service starts a server on port 8000, to provide device statistics, such as power usage or battery levels. The API endpoint table is simple:

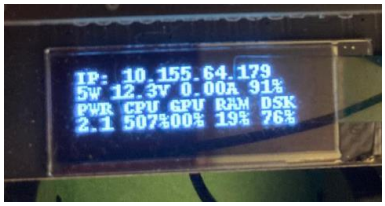
GET	*/stats/on*	Shows the device parameters on the display:  *Figure 19: JetRacer Display parameters*
GET	/stats/off	Turns off the device parameters on the display
GET	/text/<text>	Returns and shows in the display the text parsed in the parameter <text>.

Table 7: API endpoint table for jetcard_display service

- The *jetcard_jupyter* service starts a JupyterLab server on port 8888, which is used to program the car's functionalities.

After the car connects to Wi-Fi, and the *jetcard_display* service initializes, the display screen shows real-time the parameters of the device. As soon as the car connects to the Wi-Fi, the display will also show the assigned IP address. Users can access the JupyterLab interface by entering `http://<jetracer_ip_address>:8888` in the browser address bar. By default, the username and password are both *jetson*.

Once inside the JupyterLab environment, a terminal con be launched to confirm that the JetRacer is running on the master branch of the repository (see Appendix II, Section [3]).

Additionally, the Jetson's power should be configured to 5 Watts, to prevent drawing excessive current from the battery pack (see Appendix II, Section [4]).

All essential scripts related to the car's navigation, training, and control are located in the `/jetcard/notebooks/` directory.

5.1.2 BASIC MOTION OF THE CAR

To understand the fundamental control logic of the JetRacer, the initial testing phase employed the basic_motion.ipynb script provided by Waveshare. This script utilizes the jetracer Python library, which includes the NvidiaRacecar class—a central interface for managing the robot's motion. The *NvidiaRacecar* class has two key attributes: *throttle*—controls the forward or reverse motion of the vehicle—and *steering*—controls the angular direction of the front wheels. Both attributes accept values within the range [-1,1]:

throttle	-1	Car drives backwards at full speed
throttle	0	Car stops
throttle	1	Car drives forward at full speed
steering	-1	Car steers completely to the left
steering	0	Car goes forward
steering	1	Car steers completely to the right

Table 8: Throttle and Steering extreme cases

5.1.2.1 Car steering

The steering system of the JetRacer is operated by a servo motor that physically rotates the front wheels. Ideally, a steering input of 0 should result in the wheels being perfectly aligned in the forward direction. However, mechanical tolerances and assembly variations often

introduce slight misalignments. As a result, the vehicle may unintentionally veer to one side, even if the software believes it is driving straight.

To correct for this, two additional parameters in the *NvidiaRacecar* class—*steering_gain* and *steering_offset*—hat can be used to calibrate the steering. The control signal sent to the steering mechanism is computed using the equation:

$$y = a * x + b$$

Where,

$$y :\equiv value\ written\ to\ the\ motor\ driver$$
$$a :\equiv steering\ gain$$
$$b :\equiv steering\ offset$$
$$x :\equiv steering$$

The steering gain represents the maximum angle the wheel can turn. The steering offset controls the steering displacement, and it can compensate for steering displacement error caused by mechanical assembly.

The calibration of the car is extremely important. This calibration is done by adjusting the `steering_gain` and `steering_offset` of the car. It consists of an iterative process. It begins by setting the steering gain to 1 and the steering offset to zero. Depending on how accurately the car follows a straight path, these parameters are adjusted. In this project, the values that produced optimal steering behavior were `car.steering_gain=0.65` and `car.steering_offset=-0.25`. If modifying the value of the steering gain and offset is not enough for the car to steer properly, adjustments to the steering rod should be made. The steering rod is the mechanical component that connects the servo motor to the steering mechanism of the car's wheels.

5.1.2.2 Car throttle

In addition to steering, the JetRacer's motion is also governed by a throttle control system. Similar to the steering configuration, the throttle is associated with a gain parameter that can

be tuned to adjust the vehicle's speed response. The throttle control signal is computed using the following linear relationship:

$$y = a * x$$

Where,

$$a :\equiv throttle\ gain$$
$$x :\equiv car.throttle$$
$$y :\equiv value\ written\ to\ the\ speed\ controller$$

The throttle input is directly mapped to the motor driver. A positive throttle value propels the car forward, while a negative value causes the car to move in reverse. If the vehicle is already in motion and a negative throttle is applied, it will decelerate or brake depending on the magnitude of the input.

The throttle gain serves as a scaling factor to control the vehicle's top speed. A gain value of 0 results in no movement, whereas a value of 1 allows the car to reach its maximum programmed speed. This parameter is particularly useful for limiting speed during testing or in constrained environments where precision is more critical than velocity.

5.1.3 CONTROLLING THE CAR REMOTELY

To control the car remotely, an Xbox controller is used. This functionality is implemented through the `teleoperation.ipynb` file.

To start the remote control, the Xbox gamepad receiver is connected to the Jetson Nano via USB. The controller's proper detection can be verified by visiting the website https://html5gamepad.com/ . Write the INDEX value of the corresponding device in the first cell of the Jupyter Notebook file (e.g. `controller = widget.controller(index=1)`, if the indicated index is 1)—the index number is shown in the webpage once the gamepad receiver is plugged in, and any bottom is pressed on the controller.

The next step is to lower the maximum speed value to ensure proper control of the car. For example, a throttle gain of 0.2 is typically sufficient for this purpose. The vehicle's steering gain and offset values, previously determined during the calibration process, should also be set.

To map joystick inputs to the vehicle's steering and throttle controls, it is necessary to identify the corresponding axes on the gamepad. This can be done by running the following code in the notebook:

```
import ipywidgets.widgets as widgets
controller = widgets.controller(index=1)
display(controller)
```

The following interactive widget will appear:

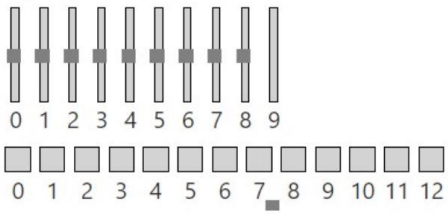


Figure 20: Controller widget

When a joystick is moved or a button is pressed, the widget will either darken a button in the second row or animate a bar in the first row. This interaction allows users to determine the axis index associated with each joystick movement. For example, this code is used for the right joystick to control the throttle of the car—moving the joystick up, increases speed and moving it down decrease it—and for horizontal axis of the left joystick to indicate the steering direction:

```
left_link = traitlets.dlink((controller.axes[0], 'value'), (car, 'steering'),
transform=lambda x: -x)
right_link = traitlets.dlink((controller.axes[3], 'value'), (car, 'throttle'),
transform=lambda x: x)
```

5.1.4 ROAD-FOLLOWING CAPABILITIES

5.1.4.1 Data Collection for Neural Network Training

The script *interactive_regression.py* has been programmed for data collection and for the training of the model. This script is used to gather the data for the model, to train it and to generate the *.pth* file, which includes the learned parameters (weights and biases) contained in the state dictionary. For convenience of use, it is worth connecting the wireless controller to the computer to be able to remotely control the vehicle. In the next stage, the camera is initialized, and the car is positioned on the road. After running the script, the widgets shown in figure 22 help with the data collection process. The desired dataset, where the images are going to be saved, is selected, then the category—at this point there is only one category and it is called "apex", another category was created in the next steps for the charging process:

```
CATEGORIES = ['apex']
```

The user drives the car using the controller, advancing a few centimeters and stopping the car often to save the images. Each time the car moves forward, the user clicks on the left image to indicate the ideal path for the vehicle to stay centered between the lanes. The selected images are automatically saved into the specified dataset and category (see widget for details). Each image filename encodes the x and y coordinates of the user's click, representing the target point on the ideal path. The "Count" option displays the total number of images saved in the dataset along with their corresponding coordinates. After the car completes a full loop around the track, the "epochs" counter is manually incremented. It is recommended to complete at least ten loops to gather a sufficient amount of data.

To build a robust model, it is essential that the dataset contains a diverse set of images. This means collecting not only images where the car is following the optimal path, but also images representing boundary conditions—such as when the car is near or at the edge of the track. Including these edge cases ensures the model learns how to recover and return to the correct trajectory, improving its ability to handle real-world scenarios and unexpected situations.

Additionally, images should be captured from various angles and lighting conditions to further enhance the dataset's variety and the model's generalization capabilities.

After enough data has been collected, the model is trained by clicking the "train" bottom, and the learned parameters (weights and biases) contained in the state dictionary are saved into a ".pth" file.

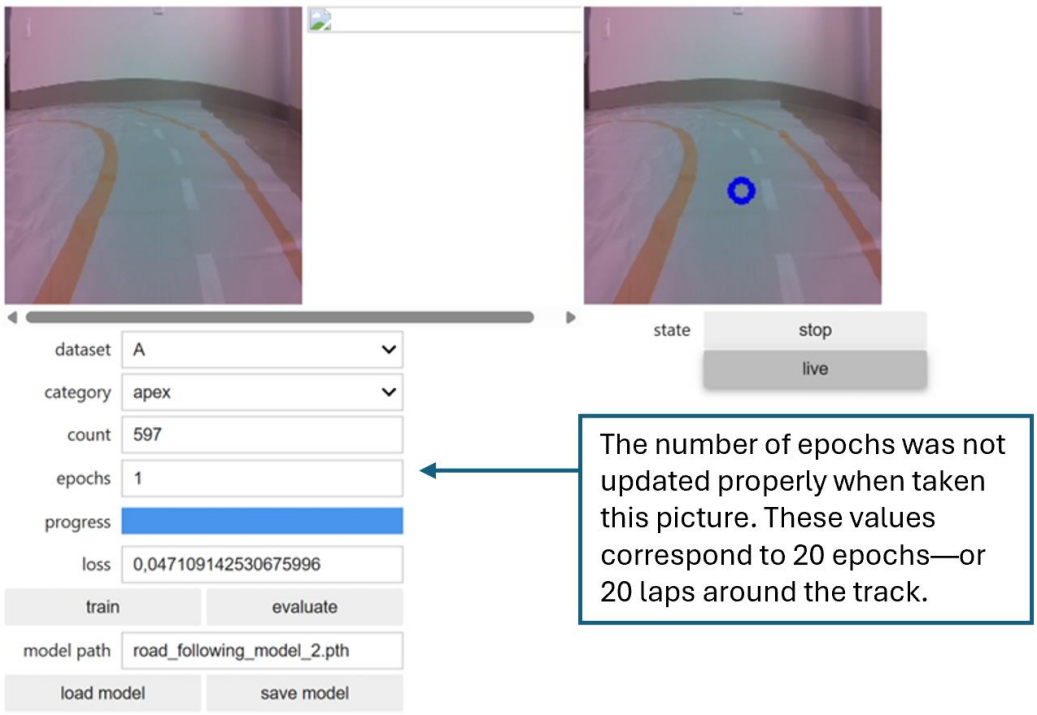


Figure 21: JupyterLab widget for data collection. 20 epochs, 597 images saved, 0.047 loss in training

For the storage of the data, a directory structure was chosen for the files rather than a specialized dataset to keep the data collection simple, transparent, and compatible with standard machine learning workflows. This approach allows for easy access, organization, and backup of the data, and is well-suited for image-based datasets where advanced querying or metadata management is not required.

5.1.4.2 The ResNet-18 model

The algorithm for autonomous driving of the Waveshare JetRacer AI car was based on the ResNet-18 neural network. The architecture of this neural network is explained in section 2.3. Figure 23 illustrates the architecture and data flow of the neural network-based autonomous driving system. The column on the left shows the step of the data flow process and the right column specifies the tool used for each process. The pre-trained ResNet-18 NN model is used. The last layer of the algorithm is shaped for self-driving purpose. The linear layer in PyTorch (`torch.nn.Linear`) is a fully connected layer that applies the formula:

$$y = Wx + b$$

where:

$$x \text{ is the input } (features \text{ extracted by earlier layers}),$$
$$W \text{ are the weights learned dureing training},$$
$$b \text{ is the bias}$$
$$\text{and } y \text{ is the output } (\text{the predicted steering direction})$$

After going through convolutional layers, the network has extracted features from the image—such as patterns—but the goal of this model is not just to extract features, but to predict something—the steering angle, so the linear layer takes all the features extracted from the image, applies weights to them and outputs a final prediction for the steering angle. Specifically, in this case, it is programmed in the following way: `model.fc = torch.nn.Linear(512, output_dim)`

Where:

```
output_dim = 2 * len(dataset.categories)
```

Therefore, the full connected layer gets 512 values and outputs in this case, two values—the x and the y coordinate of the ideal path—because the dataset currently has just one category called “apex”.

The next step is the optimizer configuration, done by the Adam optimizer, which is used during training to reduce the error between predicted and true steering angles.

The training is done using the PyTorch model, which is just a code representation of the neural network built using PyTorch's tools and it helps define the architecture of the network and how it is trained.

Lastly, the TensorRT model is a high-performance deep learning inference library. After training the model in PyTorch, it is converted to a TensorRT model to make it faster—in this case, three times faster, from 29.4 to 90.2 frames per second [33]. The TensorRT model is specifically designed to optimize real-time inference on NVIDIA devices like the Jetson Nano.

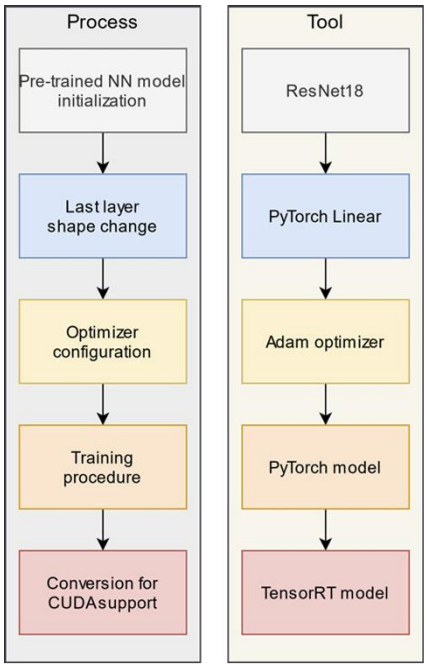


Figure 22: Block diagram of data flow for autonomous driving [33]

Other architectures were considered, such as AlexNet, SqueezeNet or DenseNet. In practice, the car followed the road most accurately when trained with ResNet architecture. ResNet-18 has superior accuracy than AlexNet or SqueezeNet. The ability to train deep networks reliably, combined with strong community support, confirmed that it was the correct choice.

While DenseNet is also a strong contender, ResNet's balance of simplicity and performance makes it a preferred choice for this specific task. Another advantage of the ResNet-18 is the small size of the model at the level of 43 MG.

Figure 22 shows that the user interface for data collection indicates the loss function of the trained model for that specific number of epochs—in the picture, the training data showed a loss function of 0.047 for 20 epochs. In reference 33, an experiment was carried out to check the impact of the number of epochs on the loss function and the behavior of the vehicle in autonomous driving mode (see table 9).

Epochs number	**Training data**	**Validation data**	**Training time [s]**
30	0.042	0.035	754.7
50	0.024	0.018	1229.6
100	0.014	0.008	2459.1

Table 9: Mean Loss function Value for Different Epochs Number [33]

The values show the efficiency of the neural network model training process. They support the affirmation that the value of the loss function decreases with the number of epochs recorded.

5.1.4.3 Road Following Process

After training and optimizing the model, the system outputs two values: the x and the y coordinate of the ideal path of the car to stay in the road. The steering of the racecar is based on the x value, which represents the deviation from the center ranges from -1 to 1. To compute the steering angle of the car a Proportional-Derivative (PD) controller is used with the following derivative:

$$Steering = x * K_p + (x - last_x) * K_d$$

An iterative process was undertaken to find the ideal gain values, being:

$$K_p = 1.5$$

$$K_d = 8$$

Initially, the high gain values were surprising, but later it was understood that since the derivative is calculated as a simple difference without dividing by the time step, a high K_d compensates for the very small value of the derivative due to the high frame rate—60 frames per second. Also, x is already centered around zero, so no offset correction is needed.

In this case, a PD controller is preferred over a full PID (Proportional-Integral-Derivative) controller, mainly to get a faster and more stable response. Adding the integral term in a fast, real-time system can cause instability, especially when there is no long-term correction error possibility. Furthermore, the integral term is designed to eliminate small residual errors over time, but in a robot's car road following system, these small errors are not critical, and the system is reset every new frame. In other words, since the controller corrects lateral errors based on the camera input and there is no need for ultra-precise long-term correction, the PD controller is sufficient to keep the car centered in the path.

5.1.5 SELF-CHARGING SYSTEM

For the charging process, the transmitter module of the wireless charger was placed directly on the track to simulate a stationary charging station. It was proposed to use several wireless chargers to simulate dynamic charging. However, due to the time required and the associated costs, it was decided to carry out static charging instead. The following sections explain the design of the static charging station.

The vehicle requires four hours to get fully charged when powered off and connected via a traditional wired charger. In contrast, the implemented wireless system enables charging while the vehicle remains powered on, increasing the total charging time by less than two additional hours. This efficiency is largely due to the minimal separation—less than one centimeter—between the transmitter and receiver modules, which allows the system to operate at its maximum rated output of 12 V and 3.2 A.

5.1.5.1 Electronic design

The transmitter module is connected to a 24 V voltage source and placed on the track as shown in figure 24.

Figure 23: Transmitter module placed on the road for static charging

The receiver coil of the wireless charging system is attached to the underside of the vehicle. The coil's module board is connected to a male pigtail barrel plug cable, maintaining correct wire polarity. The cable is then plugged into the car's female power connector, allowing the received current to flow directly into the vehicle's power system.

With this design, the car gets charged when placed over the transmitter module, as shown in figure 25.

Figure 24: Car charging at station

5.1.5.2 Battery monitoring

For the monitoring of the battery state of the vehicle, the functionality of the display server was extended to include new endpoints that provide real-time information regarding the battery state. This modification enables external scripts and services to access up-to-date battery and charging status data.

The script *display_server.py* implements the logic required to calculate and display various parameters related to the vehicle's battery status. Specifically, the logic for displaying statistics obtained from the INA219 sensor is as follows:

```
elif(self.ina219 != None):
    bus_voltage = self.ina219.getBusVoltage_V()        # Voltage on V- (load
side)
    current = self.ina219.getCurrent_mA()              # Current in mA
    p = (bus_voltage - 9)/3.6*100
    if(p > 100): p = 100
    if(p < 0): p = 0
    if(current < 0): current = 0
    if(current > 30):
        self.charge = not self.charge
    else:
        self.charge = False

    if(p < 30):
        Low_Battery = not Low_Battery
    else:
        Low_Battery = False

    if(Low_Battery == False):
        self.draw.text((110, -2), ' ', font=self.font, fill=255)
    else:
        self.draw.text((110, -2), '!', font=self.font, fill=255)

    if(self.charge == False):
        self.draw.text((600, -2), ' ', font=self.font, fill=255)
    else:
        self.draw.text((120, -2), '*', font=self.font, fill=255)
    self.draw.text((4, top), power_mode_str + (" %.1fV")%bus_voltage + ("
%.2fA")%(current/1000) + (" %2.0f%%")%p, font=self.font, fill=255)
    self.ina219.battery_level = p
```

This logic ensures that the battery level and charging state are accurately monitored and displayed. When the battery level falls below a certain threshold, a warning symbol is shown on the display. Similarly, the charging state is indicated by an asterisk when active.

To facilitate remote access to the battery level and charging state, additional endpoints were implemented using Flask, as shown below:

```
@app.route('/battery_level')
def battery_level():
    global server
    if server.ina219 is not None:
        return str(server.ina219.battery_level)

@app.route('/charging')
def charging():
    global server
    if server.charge == False:
        return str(0)
    if server.charge == True:
        return str(1) # Car is charging

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=3000, debug=False)
```

Upon startup, the Jetson device automatically initiates a service called *jetcard.display*. It was observed that this service utilizes a cached version of *display_server.py*, which prevents recent changes from taking effect upon restarting the vehicle or the service. Therefore, before executing the road-following script, the automatic display server should be disabled and a new service run. For this purpose, port 3000 was used. See Appendix II, section 5 for guidance on how to kill a service and start a new one.

With this setup, the road-following script can access real-time battery and charging information through the following endpoints:

GET	http://localhost:3000/battery_level	Returns the battery level in percentage (between 0 and 100)
GET	http://localhost:3000/charging	Returns an integer indicating the charging state: {0: not charging, 1: charging}

Table 10: Extended jetcard_display API endpoint for battery monitoring

5.1.5.3 Software

In terms of redefining the already existing ResNet-18 model, a new category was defined for the dataset, named "searching_charging_station." The list of categories is now as follows:

```
CATEGORIES = ['apex','searching_charging_station']
```

As a result, the model output is now four-dimensional:

```
[x_apex, y_apex, x_searching_charging_station, y_searching_charging_station]
```

It was determined that the dataset class does not require modification. However, the dataset now includes an additional directory, "searching_charging_station," which contains images collected while the vehicle searches for the charging station. During data collection, it is important to ensure that images are saved for both of the following scenarios:

- Scenario 1: The vehicle has sufficient battery and follows the lane markings.
- Scenario 2: The vehicle has low battery, follows the lane markings, and transitions to searching for the charging station upon identification.

The subsequent section provides details regarding the battery monitoring logic. The decision-making process is further explained in the road-following section.

Lastly, for the vehicle to search for the charging station, some changes were made to the infinite loop for road following. Two functions were defined: get_battery_level() and get_charging_state(). The first function extracts from the endpoint "localhost:3000/battery_level" the car's current battery level and the second function returns whether the car is charging or not.

The conditional statement within the control loop is designed to manage the vehicle's behavior based on the current battery level and charging status, as follows:

- If the battery level falls below 30% and the vehicle is not charging, the system initiates a search for the charging station. In this case, the category index is set to 1, thereby selecting the pair of coordinates in the model output corresponding to the "searching_charging_station" category, and the vehicle's speed is reduced.

- If the battery level is low and the vehicle is charging, this indicates that the car in in the charging station. Consequently, the vehicle is brought to a stop.

- If the battery level exceeds 90% while the vehicle is charging, it is determined that charging is complete, and the vehicle should leave the station. The vehicle is then started, and the category index is reset to "apex."

- In all other cases, the battery level is considered sufficient, and the vehicle continues to operate under normal driving conditions.

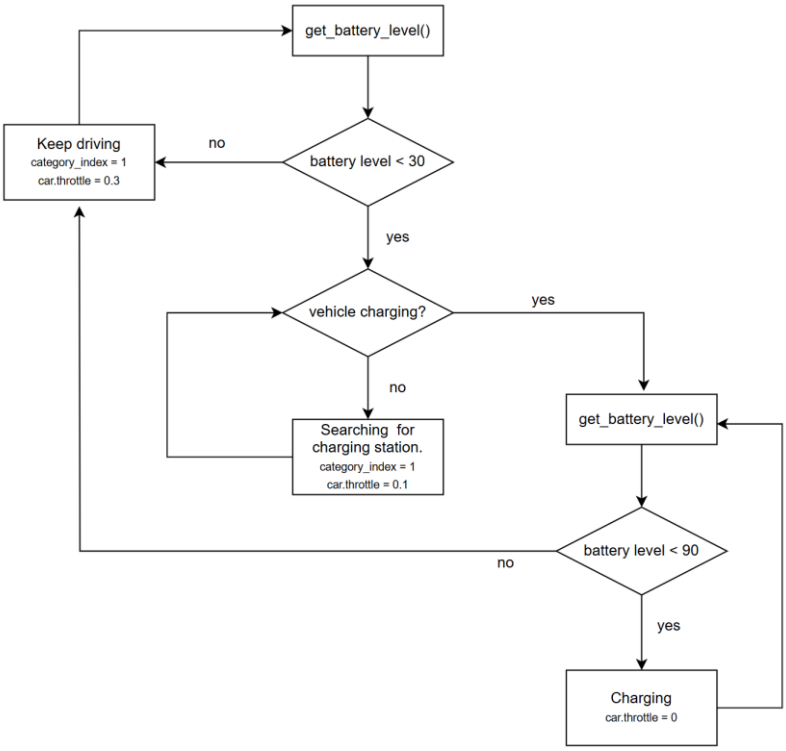


Figure 25: Flow chart self-charging algorithm

See Appendix II section 6 to see the code changed in the driving loop of the road following script.

5.1.6 GENERAL ISSUES ENCOUNTERED

5.1.6.1 Programming the PiRacer AI Kit

Several approaches were explored to initiate the PiRacer and achieve full functionality. Despite these efforts, various technical difficulties prevented successful operations. The following section outlines the steps taken, the encountered issues, and the rationale behind each attempt, with the intention of providing a useful reference for future troubleshooting and development.

The first approach followed the official Waveshare Wiki for DonkeyCar on Raspberry Pi [1], which provides a step-by-step installation guide for configuring the PiRacer with the DonkeyCar software. The process began with flashing the recommended operating system image—preferably on a SanDisk SD card, as advised by technical support—and proceeded through the installation of necessary libraries, setup of a Python virtual environment, and installation of DonkeyCar and its dependencies. Several Ubuntu images were initially tested but resulted in kernel issues; subsequently, Debian Bullseye was installed, which resolved some problems but introduced new ones (see Appendix II section 7 for detailed installation).

Detailed steps included updating and upgrading system packages, installing Python and OpenCV dependencies, configuring the virtual environment, and cloning the DonkeyCar repository [34]. During the installation, compatibility issues arose, particularly with Python version requirements and the transition from *setup.py* to *pyproject.toml* for package management (see Appendix II section 8). These challenges required upgrading Python to version 3.11, which involved compiling from source and updating environment paths to ensure accessibility of installed scripts.

Despite following the official documentation and resolving multiple dependency issues, the installation process encountered a persistent problem: while steering calibration was successful, throttle calibration failed, and the vehicle could not be driven via the web

interface. Adjustments to the configuration file did not resolve this issue. Waveshare was contacted for recommendations to solve this issue, and they sent a pre-configured image. Section 9 of appendix II contains the pre-configured image and a detailed explanation for the installation. But unfortunately, the throttle issue was not solved. To determine whether the issue originated from the circuit board, the connections and voltage levels were carefully inspected; however, no faults were identified. Further troubleshooting included reaching out to the DonkeyCar community via Discord (#software channel), where it was clarified that the current version of DonkeyCar does not support the stock PiRacer. Given that both the Waveshare Wiki and technical support had already been consulted without resolving the problem, it was ultimately decided to discontinue work with the PiRacer and transition to the JetRacer AI Kit for further development.

5.1.6.2 Battery Power not working on PiRacer and JetRacer

While troubleshooting the throttle issue, it was initially considered that the vehicle might have a safety feature preventing it from accelerating while connected to external power. However, after disconnecting the power supply and attempting to operate the vehicle on battery power alone, the system powered off and did not function as expected. Subsequent investigation revealed that, for the first use, the batteries must be initialized by connecting the charger, turning on the vehicle, and then unplugging the charger. This process activates the battery protection circuit, enabling the batteries to supply power to the vehicle properly.

5.1.6.3 JetRacer – Blurred camera content

During the process of verifying camera functionality, it was observed that, although the camera was recognized and operational, the captured images were extremely blurry, making object identification impossible. After confirming that all necessary modules were installed and updated, and that the camera was properly detected by the Raspberry Pi, it was determined that the issue was not software-related. Further investigation revealed that the focus of the camera can be adjusted manually by carefully rotating the lens assembly. This adjustment is sometimes required, as certain camera modules are shipped with a default

focus setting that may not be suitable for all applications, resulting in blurry images until properly focused.

Chapter 6. ANALYSIS OF RESULTS

The neural network model developed for this project processes real-time images captured by the vehicle's onboard camera and outputs corresponding steering values to maintain the car's position on the road. The model's performance was evaluated based on the number of training epochs. With 20 epochs, the vehicle was able to follow the road; however, it frequently crossed the orange boundary lines, particularly when navigating curves, indicating limited generalization to more complex driving scenarios. Increasing the training to 50 epochs resulted in improved performance: the vehicle remained within the lane boundaries and exhibited only minor deviations from the center of the road. This configuration was determined to offer a suitable compromise between driving precision and model complexity.

When images related to the charging process were incorporated into the training dataset, it became necessary to further increase the number of epochs, especially for scenarios near the charging station. This adjustment was essential to prevent the vehicle from overshooting the station and to achieve optimal alignment between the wireless charging modules. Precise alignment is critical to ensure efficient energy transfer and to avoid mechanical interference with the charging infrastructure.

Despite these advancements, certain enhancements remain unimplemented due to time constraints. Integrating an object detection system would allow the vehicle to more reliably identify and approach the charging station, while the construction of a dedicated docking station would facilitate consistent and accurate alignment with the charger. These improvements would significantly increase the system's robustness and operational reliability, particularly in more dynamic or unstructured environments.

Chapter 7. CONCLUSION AND FUTURE RESEARCH

This project addresses a critical challenge in the development of autonomous electric vehicles: enabling a truly autonomous charging process. As described in the project definition, while significant progress has been made in autonomous navigation and real-time decision-making using artificial intelligence and sensor technology, the charging process typically still requires human intervention. By implementing a system where the vehicle can autonomously detect low battery levels, navigate to a charging station, and initiate charging without human input, the autonomy and operational efficiency of electric vehicles are significantly enhanced. This advancement has the potential to transform the user experience, allowing users to leave their vehicles running errands while the car independently manages its charging needs and returns to pick them up once recharged. Such a system not only facilitates everyday use but also supports continuous operation for services like robotaxis and autonomous logistics, where minimizing downtime is crucial.

The solution developed serves as a testbed for the concept of autonomous vehicles capable of managing their own charging cycles. Additionally, this work provides an in-depth review and integration of advancements contributed by the open-source and academic community in the field of the PiRacer, JetRacer and the NVIDIA projects.

The self-driving process implemented in this project is based on the use of embedded systems and neural networks for real-time perception and control. The system developed in this project utilizes a camera module for data acquisition and neural network models trained to interpret visual input and generate steering commands. The central processing unit, an embedded microcomputer with GPU-based CUDA architecture, enables the execution of these models in real time.

The training process involved collecting a dataset under controlled conditions, manually labeling direction vectors, and empirically tuning the hardware configuration. The neural network architecture was selected for its balance of accuracy and efficiency, with additional

optimization performed to improve inference. This neural network model inputs an image taken from the camera and returns a steering value for the car to stay on the road.

For the self-charging process, the system integrates battery monitoring and decision-making logic that enables the vehicle to identify when charging is necessary, locate the charging station, and autonomously initiate the charging procedure.

However, the current solution is limited to operation within a known and controlled environment. The system's ability to generalize to new or more complex scenarios remains untested and represents a significant area for future work. Despite these limitations, the project highlights the promise of autonomous self-charging vehicles and establishes a practical foundation for further research and development in this area.

In summary, this work demonstrates a first practical approach to autonomous self-driving and self-charging vehicles, combining state-of-the-art methods and community-driven innovations. While constrained by the simplicity of the test environment, the results affirm the viability and future potential of such systems for more advanced and autonomous mobility applications.

7.1 MODEL LIMITATIONS AND OPPORTUNITIES TO IMPROVE LEVEL OF AUTONOMY

The current implementation relies solely on images captured by the vehicle's onboard camera for both road following and driving to the charging station. While this approach demonstrates the feasibility of vision-based navigation, it limits the vehicle's awareness of its environment. As reviewed in the state-of-the-art section, integrating additional technologies such as LiDAR for three-dimensional environmental mapping or GPS for global path planning could substantially increase the vehicle's level of autonomy. These enhancements would enable obstacle detection, dynamic path adjustment, and operation in unfamiliar or complex environments, moving the system toward higher levels of autonomous driving capability.

7.2 DATABASE CONSIDERATIONS

A file-based dataset was employed in this project due to its simplicity and suitability for the scale of data generated by a single camera in a controlled environment. This approach allows for straightforward data management. However, as further technologies for autonomous driving are considered, the scope of data expands a more scalable and efficient database solution would be necessary. Advanced database systems can support higher data throughput, facilitate real-time data processing, and ensure the integrity and accessibility of large datasets required for more sophisticated autonomous driving systems.

7.3 CHARGING TECHNOLOGY

Inductive wireless charging, while enabling fully autonomous operation, is less efficient and slower than traditional plug-in charging. According to recent studies, wireless charging systems typically achieve 80–90% efficiency, resulting in longer charging times compared to plug-in solutions—charging a battery pack may take approximately 10–20% longer wirelessly than via plug-in methodshttps://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/52358752/53cac593-8f66-4115-9996-a9b04f975db6/TFG__NOT-FINISHED.pdf [20]. To mitigate this limitation, the integration of solar panels has been considered as a supplementary energy source. This hybrid approach could provide continuous charging during daylight hours, further extending the vehicle's operational range and reducing reliance on static charging infrastructure.

Combined with the solar energy approach, using multi-cell architecture for the batteries could further enhance charging efficiency, maintaining the vehicle's fully electric nature while minimizing environmental impact. Modern electric vehicles often utilize multi-cell battery architectures to distribute charging loads and enable faster charging rates. By employing multiple charging cells, the system can simultaneously charge different sections of the battery pack, reducing total charging time.

7.4 RACING ROBOT CARS

In addition to serving as a testbed for real-world autonomous vehicles, the developed platform is highly applicable to the field of racing robot cars. In competitive environments, the ability for a vehicle to autonomously manage its energy requirements and optimize pit-stop strategies can provide a significant advantage. Furthermore, this system could be directly implemented and adapted for robot car platforms, enabling rapid deployment and experimentation in robotics competitions and educational settings.

Chapter 8. REFERENCES

[1] PiRacer AI Kit Wiki: http://www.waveshare.com/wiki/PiRacer_AI_Kit

[2] JetRacer AI Kit: JetRacer AI Kit, AI Racing Robot Powered by Jetson Nano

[3] JetRacer AI Kit Wiki: JetRacer AI Kit - Waveshare Wiki

[4] DonkeyCar platform documentation: Donkey Car

[5] TensorFlow documentation: https://www.tensorflow.org/

[6] OpenCV documentation: https://opencv.org/

[7] TensorRT documentation: https://developer.nvidia.com/tensorrt

[8] Keras documentation: https://keras.io/

[9] PyTorch documentation: https://pytorch.org/

[10] Jetson Nano Brings the Power of Modern AI to Edge Devices | NVIDIA

[11] INA219 data sheet, product information and support | TI.com

[12] INA219 Current Sensor Circuit Diagram and Datasheet - Homemade Circuit Projects

[13] Amazon.com: Taidacent 12V 3A High Power Long Distance Wireless Inductive Coil Modules Wireless Charging Coil Wireless Transmitter and Reveiver Coil Wireless Charging Power Supply Module : Electronics

[14] IEEE Xplore Full-Text PDF: An Overview of LiDAR Requirements and Techniques for Autonomous Driving

[15] Accuracy–Power Controllable LiDAR Sensor System with 3D Object Recognition for Autonomous Vehicle

[16] Autonomous Vehicles: Evolution of Artificial Intelligence and the Current Industry Landscape

[17] Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms - ScienceDirect

[18] An Overview of Autonomous Sensors - LIDAR, RADAR, and Cameras

[19] (PDF) SURVEY ON SENSOR FUSION FOR AUTONOMOUS DRIVING: TECHNIQUES AND CHALLENGES

[20] A Comprehensive Analysis of Wireless Charging Systems for Electric Vehicles | IEEE Journals & Magazine | IEEE Xplore

[21] Wireless Power Transfer - Overview and Applications

[22] Inductive Power Transfer - an overview | ScienceDirect Topics
[23] Trends in electric car markets – Global EV Outlook 2025 – Analysis - IEA
[24] Why Are Electric Cars Today So Huge? - CleanTechnica
[25] AAA: Americans Slow to Adopt Electric Vehicles | AAA Newsroom
[26] The long road to electric cars in the U.S.
[27] Transport User Voice February 2024 - Electric Vehicle Charging Survey - Transport Focus
[28] JetRacer Assembly Manual - Waveshare Wiki
[29] https://drive.google.com/file/d/1ZBdqrwhW2n1uN8rughF7Puw98o76kUcH/view?usp=sharing
[30] IJRTI
[31] Vanishing Gradient Problem: Causes, Consequences, and Solutions - KDnuggets
[32] deep learning - Why do ResNets avoid the vanishing gradient problem? - Artificial Intelligence Stack Exchange
[33] Aspects of autonomous drive control using NVIDIA Jetson Nano microcomputer | IEEE Conference Publication | IEEE Xplore
[34] Get Your Raspberry Pi Working - Donkey Car

APPENDIX I: ALIGNMENT WITH THE SUSTAINABLE DEVELOPMENT GOALS (SDGS)

This project aligns closely with several United Nations Sustainable Development Goals (SDGs), contributing to global efforts toward sustainable innovation. The key SDGs addressed are outlined in this section.

Figure 26: Goal 7 - Affordable and clean anergy And Goal 12 – Responsible Consumption and Production

The system's ability to identify battery levels on electric vehicles and to help them navigate through the most efficient route determined by deep learning algorithms, directly minimizes energy waste and promotes the efficient use of renewable energy. Moreover, by enabling self-charging the project enhances the attractiveness of EVs, helping transition to clean energy systems. The emphasis on energy-efficient navigation and renewable energy integration helps reduce the environmental footprints of EVs, supporting the goal of responsible consumption and production.

Figure 27: Goal 9 - Industry, Innovation and Infrastructure

The system also represents a groundbreaking advancement in the automotive industry, enhancing innovation. This project contributes to the creation of resilient and intelligent infrastructure for EVs.

Figure 28: Goal 11 - Sustainable cities and communities And Goal 13 – Climate Action

Enabling vehicles to autonomously manage their energy consumption and charging needs, reduces traffic congestion, allows for smooth traffic and enhances mobility. It also reduces significantly the environmental impact of urban transportation, since it generates zero greenhouse gas emissions.

By addressing these SDGs, the project demonstrates its commitment to advancing sustainable transportation solutions while contributing to a cleaner, smarter future.

APPENDIX II

[1] Editing the wpa_supplicant.conf file:

From the Jetson terminal, use sudo to open an editor of choice (e.g. nano, micro, vi) and edit the file located in:

```
/etc/wpa_supplicant/wpa_supplicant.conf
```

If the file does not exist, create and open the file with root privileges and set the proper permissions, for security:

```
sudo nano /etc/wpa_supplicant/wpa_supplicant.conf
sudo chmod 600 /etc/wpa_supplicant/wpa_supplicant.conf
sudo chown root:root /etc/wpa_supplicant/wpa_supplicant.conf
```

Add a network to the network block by defining the SSID and password (PSK) for the Wi-Fi network:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=wheel
update_config=1

network={
    ssid="MyWiFiNetwork"
    psk="your_wifi_password"
}
```

See wpa_supplicant - Gentoo Wiki to understand what parameters wpa_supplicant.conf supports beyond ssid and psk. For example, for a WPA2 network:

```
network={
    ssid="NetworkSSID"
    psk="NetworkPassword"
    key_mgmt=WPA-PSK
}
```

For an open network:

```
network={
    ssid="OpenNetworkSSID"
    key_mgmt=NONE
}
```

After editing the file, restart or reload wpa_supplicant to apply changes:

```
sudo systemctl restart wpa_supplicant
```

[2] Configure NetworkManager to connect automatically to Network

First, install nano:

```
sudo apt update
sudo apt install nano
```

Second, enable automatic login:

- Open a terminal and edit the LightDM configuration file:

```
sudo nano /etc/lightdm/lightdm.conf
```

- Add the following lines under the [Seat:*] section:

```
autologin-user=your_username
autologin-user-timeout=0
```

Replace "your_username" with the actual username.

Third, configure NetworkManager to connect automatically:

- Open the NetworkManager connection editor:

```
sudo nm-connection-editor
```

Select the Wi-Fi network, click "Edit", and check the box for "Connect automatically"

Fourth, disable the sudo password prompt:

- Open the sudoers file:

```
sudo visudo
```

- Add the following line at the end of the file:

```
your_username ALL=(ALL:ALL) NOPASSWD: ALL
```

Replace "your_username" with the actual username.

Lastly, ensure NetworkManager starts on boot:

```
sudo systemctl enable NetworkManager
```

And reboot to apply changes:

```
sudo reboot
```

[3] Switching the JetRacer to the Master batch

```
cd Jetracer
sudo git checkout master
sudo python3 setup.py install
sudo reboot
```

[4] Configuring power mode to 5W: launch a new terminal and enter the following commands to select the 5W power mode.

```
sudo nvpmodel -m1
sudo nvpmodel -q
```

At this time, the OLED should display: `MODE: MAXN`

[5] Killing jetcar.display service in port 8000 and starting a new one in port 3000

1. View all active services filtering by those that start by "jet"

```
systemctl list-units --type=service | grep jet
```

2. Disable the jetcard_display service

```
sudo systemctl stop jetcard_display
```

3. Manually run the display_server.py script to enable posting of status information on port 3000:

```
sudo python3 display_server.py
```

[6] Battery management functions and changes in loop for road following and autonomous charging

```
# %%
```

```
from utils import preprocess
import numpy as np
import requests

def get_battery_level():
    try:
        resp = requests.get("http://localhost:3000/battery_level")
        return float(resp.text)
    except Exception as e:
        print("Could not get battery level:", e)
        return None

def get_charging_state():
    try:
        resp = requests.get("http://localhost:3000/charging")
        return int(resp.text) # 0-not charging ; 1-charging
    except Exception as e:
        print("Could not get charging state:", e)
        return None

CATEGORIES = ['apex','searching_charging_station']

Kp = 1.5
Kd = 8
last_x = 0

car.steering_offset = -0.25
car.steering = 0
car.throttle_gain = 0.3
car.steering_gain = 0.65
car.throttle = -0.8

category_idx = CATEGORIES.index("apex")

while True:
    image = camera.read()
    image = preprocess(image).half()
    output = model_trt(image).detach().cpu().numpy().flatten()

    battery_level = get_battery_level()
    charging = get_charging_state()


    if battery_level<30 and charging == 0:
        category_idx = CATEGORIES.index("searching_charging_station.\n")
        car.throttle_gain = 0.1
        print("Searching for charging station.\n")
    elif battery_level<30 and charging == 1:
        car.throttle_gain = 0
        print("Charging station found")
    elif battery_level>90 and charging == 1:
        category_idx = CATEGORIES.index("apex")
        car.throttle_gain = 0.3
```

```
        print("Charged! Returning to road.\n")
    else:
        category_idx = CATEGORIES.index("apex")
        car.throttle_gain = 0.3
        print("all good")
    print(battery_level,category_idx)

    x = float(output[2 * category_idx])
    y = float(output[2 * category_idx + 1])

    car.steering = (x * Kp + (x - last_x) * Kd)
    last_x = x
```

[7] PiRacer Software setup following Waveshare's Wiki

1. Install libraries

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential python3 python3-dev python3-pip python3-
virtualenv python3-numpy python3-picamera
sudo apt-get install python3-pandas python3-rpi.gpio i2c-tools avahi-utils
joystick libopenjp2-7-dev libtiff5-dev gfortran
sudo apt-get install libatlas-base-dev libopenblas-dev libhdf5-serial-dev git ntp
```

2. Install libraries for OpenCV

```
sudo apt-get install libilmbase-dev libopenexr-dev libgstreamer1.0-dev libjasper-
dev libwebp-dev
sudo apt-get install libatlas-base-dev libavcodec-dev libavformat-dev libswscale-
dev libqtgui4 libqt4-test
```

3. Setup virtual environment

```
python3 -m virtualenv -p python3 env --system-site-packages
echo "source env/bin/activate" >> ~/.bashrc
source ~/.bashrc
```

4. Install DonkeyCar Python code

    i. Create a project directory

```
mkdir projects
cd ~/projects
```

    ii. Clone donkeycar codes from Github

```
git clone https://github.com/waveshare/donkeycar
cd donkeycar
```

```
git checkout master
pip install -e .[pi]
pip install tensorflow==1.13.1
pip install numpy -upgrade
```

iii. Verify TernsorFlow installation

```
python -c "import tensorflow"
```

If there are no errors, then the installation is normal and the following warning is normal.

```
r_util' does not match runtime version 3.5
  return f(*args, **kwds)
/home/pi/env/lib/python3.5/importlib/_bootstrap.py:222: RuntimeWarning:
builtins.type size changed, may indicate binary incompatibility. Expected 432,
got 412
  return f(*args, **kwds)
```

5. Install OpenCV

```
sudo apt install python3-opencv
```

If you fail at the last command, use this command to install OpenCV again.

```
pip install opencv-python
```

Check if the OpenCV is installed successfully

```
python -c "import cv2"
```

If there are no errors, then you have installed OpenCV!

Note: If prompted with `__atomic_fetch_add_8` undefined error, first, exit the virtual environment with the deactivate command, run the following command, and then retest.

```
deactivate
echo "export LD_PRELOAD=/usr/lib/arm-linux-gnueabihf/libatomic.so.1 " >>
~/.bashrc
source ~/.bashrc
```

6. Install Service OLED Display

```
cd ~
git clone https://github.com/waveshare/pi-display
cd pi-display
sudo ./install.sh
```

7. Create DonkeyCar

```
cd ~/projects/donkeycar
donkey createcar --path ~/mycar
```

After runing, files will be generated and saved in the directly ~/mycar

[8] Preparing the system to use *pyproject.toml* instead of *setup.py* file for installation

This problem arouses when following the steps in the DonkeyCar Documentation. Everything worked properly until the `pip install -e .` step, when the following error appeared, which means that DonkeyCar has moved to a modern build system using pyproject.toml, but pip is still expecting *setup.py*.

To fix this:

1. Since the Python version is about to be changed, the created environment with python 3.7 has to be erased

```
rm -rf /home/jetracer01/env
```

i. Ensuring that the correct version of python is being used

```
python --version
pip --version
```

ii. Ungrading pip, setuptools, and wheel

```
pip install --upgrade pip setuptools wheel
```

iii. Intalling DonkeyCar with Modern Build System

```
pip install .
```

When running `pip install .` , the following error message might appear:

```
"Package 'donkeycar' requires different python: Python: 3.7.3 not in
'<3.12,>=3.11.0'.
```

Installing python 3.9 does not solve the problem, so the python 3.11 is installed. `pip install python3.11` might not install it, so python 3.11 is compiled from source:

i. Installing dependencies

```
sudo apt update
sudo apt install -y build-essential zlib1g-dev
  libncurses5-dev libgdbm-dev libnss3-dev
  libssl-dev libreadline-dev libffi-dev curl
```

If the following error message does not show, skip this explanation and continue with step 2:

```
Error: E: Unable to locate package libncurses5-dev
E: Unable to locate package libssl-dev
```

This indicates that the package lists may be outdated, or that the version of Raspberry Pi OS in use does not include these packages by default. The following steps outline how to resolve this issue:

a. Updating Package lists

```
sudo apt update
sudo apt upgrade -y
```

b. Enabling deb-src repositories

```
sudo nano /etc/apt/sources.list
```

It is important to ensure that the following lines are present; if they are missing, they should be added accordingly:

```
deb http://deb.debian.org/debian bookworm main contrib non-free
deb-src http://deb.debian.org/debian bookworm main contrib non-free
```

If an older version of Raspberry Pi OS is being used, replace "bookworm" with "bullseye" or "buster" as appropriate.

After making the necessary changes, save and exit the file. Then, update the package lists again using the following command:

```
sudo apt update
```

c. Installing missing dependencias

```
sudo apt install -y build-essential zlib1g-dev libncurses-dev \
  libgdbm-dev libnss3-dev libssl-dev libreadline-dev libffi-dev curl
```

If the error message

```
Unable to locate package
```

still appears, consider trying the following alternative package names:

```
sudo apt install -y libncurses-dev libssl1.1 libssl3
```

d. Verify dependencies

```
dpkg -l | grep libncurses
dpkg -l | grep libssl
```

2. Downloading and Extracting Python 3.11

```
cd /usr/src
sudo curl -O https://www.python.org/ftp/python/3.11.6/Python-3.11.6.tgz
sudo tar -xvf Python-3.11.6.tgz
cd Python-3.11.6
```

3. Building and Installing Python

```
sudo ./configure --enable-optimizations
sudo make -j$(nproc)
sudo make altinstall
```

Steps 2 and 3 require the use of **sudo**. If it is preferable to avoid using root privileges, consider the following alternatives:

4. Verify installation

```
wget https://www.python.org/ftp/python/3.11.6/Python-3.11.6.tgz
tar xf Python-3.11.6.tgz
cd Python-3.11.6
./configure --enable-optimizations
make
sudo make altinstall
```

5. Set Python 3.11 as default

```
sudo update-alternatives --install /usr/bin/python3 python3 /usr/bin/python3.11 1
sudo update-alternatives --config python3
```

In this case, the wheel script was not installed in `/usr/bin/python3.11`, but rather in `/home/piracer01/.local/bin`. To resolve this, the shell configuration file was edited to include this path:

```
nano ~/.bashrc
```

The following line was added at the end of the file:

```
export PATH=$PATH:/home/piracer01/.local/bin
```

And the shell configuration file was updated:

```
source ~/.bashrc
```

After making this change, the scripts installed in `/home/piracer01/.local/bin` should be accessible from the command line.

Next, the environment was created again and activates executing the following commands:

```
python3 -m venv env --system-site-packages
echo "source ~/env/bin/activate" >> ~/.bashrc
source ~/.bashrc
```

6. Then, the DonkeyCar guide for getting the RaspberryPi ready was followed to install the required libraries (Step 4 of reference 34)

[9] Setting up software for PiRacer following Waveshare's support recommendations

The following link contains the pre-configured image provided by Waveshare:

https://emea01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fdrive.google.com%2Ffile%2Fd%2F1h92LSykGYtcuSrPS1YoNxfHG_CdT_6Gs%2Fview%3Fusp%3Ddrive_link&data=05%7C02%7C%7Ca188729f5a014318ba2f08dd4cfdfe22%7C84df9e7fe9f640afb435aaaaaaaaaaaa%7C1%7C0%7C638751375946602663%7CUnknown%7CTWFpbGZsb3d8eyJFbXB0eU1hcGkiOnRydWUsIlYiOiIwLjAuMDAwMCIsIlAiOiJXaW4zMiIsIkFOIjoiTWFpbCIsIldUIjoyfQ%3D%3D%7C0%7C%7C%7C&sdata=5vQHJm%2B0FGyXD8iEuald1eId8Sgpr9zs6oEIgFDPGXs%3D&reserved=0

The image was flashed into the SD card, and `pip install -e.[pi]` was run to install the driver and to configure the Pi board.

It is also recommended to verify that all commands, including installations and upgrades, have been executed successfully. To do this, refer to the steps outlined after the `pip install` command in Step 4 of the "DonkeyCar for Pi-Setup Raspberry Pi" guide from the Waveshare Wiki [34]. Check whether these subsequent steps were automatically performed during your installation process. If not, execute them manually to ensure that all required components are properly installed and configured.

When setting up the board, the following error message appeared:

This can be solved either with a system installation or withing the virtual environment.

i. System installation

```
sudo python3 -m pip install -U pip
sudo python3 -m pip install -U setuptools
```

ii. Whitin the venv:

```
pip3 install -U pip
pip3 install -U setuptools
```

After solving this error, the numpy library had to be installed:

```
sudo apt install python3-numpy
```