Autonomous Parking Lot Navigation for Self-Driving Model Car

Team members: Oliver Rzepecki, Isaiah Pajaro, Vraj Panchal, Ryan Elizondo-Fallas

Team SP24-29

Advisor: Hang Liu

Submitted in partial fulfillment of the requirements for senior design project
Electrical and Computer Engineering Department
Rutgers University, Piscataway, NJ 08854

***Abstract*—This research presents a novel approach to achieving autonomous parking capabilities through the fusion of advanced hardware and deep learning techniques. By integrating fisheye cameras and the Nvidia Jetson TX2 computational unit, the project focuses on overcoming challenges associated with panoramic perception and decision-making in parking scenarios. Utilizing image stitching and semantic segmentation algorithms, the system effectively processes environmental data to identify parking slots and surrounding objects. Preliminary results indicate promising accuracy in semantic segmentation and environment processing, particularly with the utilization of TensorRT for GPU acceleration. The proposed framework demonstrates the potential for enhancing autonomous vehicle capabilities, with implications for urban mobility and transportation efficiency. Future endeavors will aim to refine system performance, explore additional sensor modalities, and optimize real-time inference acceleration for seamless integration into practical autonomous parking applications.**

***Keywords*—autonomous driving, computer vision, TensorFlow, model car, Jetson TX2, semantic segmentation, around-view monitor, OpenCV, Image Stitching**

I. Introduction

The advancement of IoT and embedded systems has given rise to the growth and prevalence of self-driving automobiles, which hold the potential to resolve the major problem of car accidents. While completely autonomous cars are not yet accessible to the public, the technology to create them is established. Given that more than 50,000 car accidents occur in parking lots every year in the United States, our project intends to reduce human error from parking lot driving. Drivers in parking lots are very often distracted and not paying full attention when navigating a parking lot [1].

Our project endeavors to construct and put into action a self-driving 1/10th-scale model car that can navigate any parking lot or garage and park itself without any human assistance. Our prototype model car is equipped with four fisheye lens cameras that create a surround view of the car, commonly known as an around view monitor. From there, an onboard Nvidia Jetson TX2 single board computer runs a semantic segmentation deep learning algorithm that classifies each pixel around the car as either a driving area, obstacle, or parking line, which is then used for autonomous decision-making to determine a start and end point of the car. Finally, these points, detected obstacles, and available driving area are used to chart a course for the car. The Jetson communicates this to an Arduino, which interfaces with the car's motor and steering servo to execute the path.

We understand this will be challenging as current autonomous car companies have yet to tackle this effectively. This is twofold, the research is lacking for the particular environment and the data collection is subpar.

Current autonomous vehicles are only allowed on designated highways and streets. This is due to abundant and accurate mapping data of such infrastructure. However, there is a lack of mapped data for parking lots and garages. Furthermore, the environment is more dynamic and demanding, as highways are usually straight areas with long sightlines and predictable traffic patterns, giving ideal conditions for data collection, processing, and reacting. Parking lots are cramped, unusual shapes, and chaotic. This leads to a tough environment where there is not much data and little time to react to changes.

Additionally, current autonomous vehicles rely on the use of LiDAR and complementary sensor array. We are attempting to tackle the issue through optical means, due to the inherent nature of parking lines and cost constraints. For one, LiDAR requires depth to work effectively. However, parking lines and asphalt do not have much depth difference and may not be detected by LiDAR. As such, cameras are the only way to take parking line data from the environment. Using cameras as the only environment data ingress will also keep costs down, however, an implementation that uses other sensors would likely be more effective but cost more and be more complex.

II. Conceptual Design

A. Abbreviations and Acronyms

- CNN – Convolutional Neural Network
- YOLO – You Only Look Once (Algorithm)
- GPU – Graphic Processing Unit
- TPU – Tensor Processing Unit

- FPGA – Field Programmable Gate Array
- AVM – Around View Monitoring
- BEV – Birds Eye View
- ESC – Electronic Speed Controller
- BEC - Battery Elimination Circuit
- GPIO – General Purpose Input/Output
- SSH – Secure Shell Protocol
- PID - Proportional-integral-derivative
- EOL – End of Life
- PWM – Pulse Width Modulation

B. Overview

This project is focused on routing a vehicle through a parking lot and parking fully autonomously. A 1/10 scale model car will be modified with four fisheye cameras stitched together to provide an around-view monitor of the car. An onboard Nvidia Jetson TX2 is used to run a TensorFlow model for image segmentation on the around-view monitor, process decision-making, and initialize movement commands.

C. Image Stitching for Around View Monitor

Vision data is the first step for our vehicle to function and we are implementing it using the feeds of four fisheye lens cameras around the car, thus making up 360 degrees of visual feedback. Image stitching is used to prevent multiples of the same object from appearing in the overlapping fields of view. This is necessary or else the segmentation model will be confused by the multiples of the same object. Wang et al. [13] presents a method to images stitching which uses autoencoder networks to extract feature points from images, allowing for better wide field of view camera images compared to traditional means.

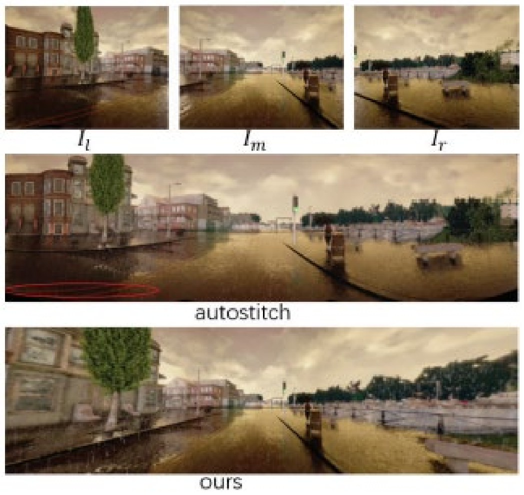

Figure 1 The proposed image stitching method verses a conventional stitcher [13]

Figure 1 shows the advantages of using this proposed method of images stitching as it has better correctively towards the effects of using larger baselines.

D. Image Segmentation for Parking Slots

After the stitched image is created from the surroundings, a CNN will be used for semantic segmentation. In the parking lot, a CNN will be used to identify the drivable area based on parking lines and surrounding parked cars. The model will be trained using either an open-source data set, our own collected data, or a hybrid of both. The You Only Look Once (YOLO) detection algorithm revolutionized real-time object detection and is an option for our object detection model [8]. This algorithm processes an entire image in a single pass through a CNN and immediately identifies objects with a calculated probability. Before using the algorithm, the data set must be prepared. In each image, annotations are required to set the location and class of each object in the image. The model can then be trained and installed onto the Jetson TX2 module.

Jang and Sunwoo's pre-trained parking space detection model is used for its performance and use of AVM in its training data. This model employs semantic segmentation, which is the process of The model has a precision rate of 96.81% and a recall rate of 97.80%. A deep-learning-based encoder and decoder algorithm is implemented to understand the environment surrounding the vehicle [9]. An example of the model's semantic segmentation is below. Parking space lines are highlighted in green while other vehicles are highlighted in red. Given these groupings, an available parking space can be identified from the drivable space between the sets of parking lines.

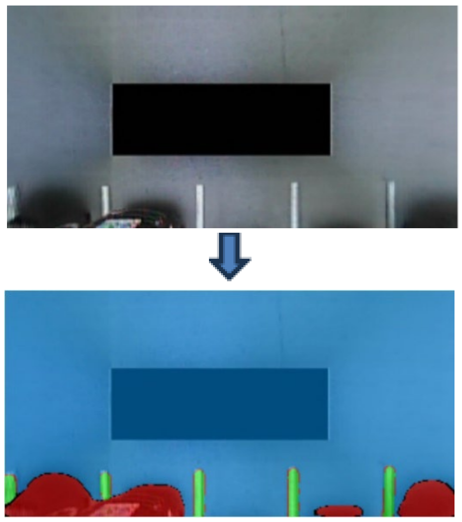
Figure 2 Input and output of CNN for semantic segmentation developed by [9]

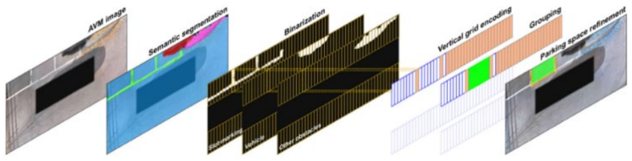

Figure 3 Semantic Segmentation image processing [9]

With semantic segmentation complete, further processing with binarization, encoding, and grouping results in a distinguished output with highlighted groups for each detected class.

The model was trained in TensorFlow 1 but the code has been ported to work with TensorFlow 2. With the model implemented and working, our next effort will be to run the model on GPU to accelerate the inference time. Due to the nature of a moving vehicle in a real-time environment, it is critical that our system processes the environment with as little latency as possible. If the inference performance on the Jetson TX2 is subpar, a separate accelerator will be utilized, such as the Google Coral USB Edge TPU ML Accelerator coprocessor [14].

Depending on our test environment and the AVM system we set up with our hardware, we may yield better results by training the neural network with our own data from the camera setup with the Jetson TX2. By using similar data for training and inference, we can expect more consistent results. Another benefit of self-training the model is that acceptable results can be obtained even when reducing the input image size to the model. The input size has a significant effect on inference time. Given the size and computation power limits of our project, such optimizations are required. We discuss this later in the results section.

E. Hardware

The model car we are using for this project is based on the Tamiya TT-02 chassis, a popular and affordable hobby RC car kit. The model car is 1/10 scale compared to an actual vehicle. The kit does not arrive pre-assembled, allowing our team to fully customize the model car for our needs of this project, adding a single board computer, a set of cameras, and potentially dedicated hardware to accelerate our inference model.

Our host board will be the central processor of the operation, responsible for hosting the operating system, receiving camera streams, running the semantic segmentation model, and controlling vehicle operation. For this, we will use an Nvidia Jetson TX2 vs the original idea of a Raspberry Pi because of its onboard GPU, which will be helpful for our semantic segmentation model. We then plan to transplant the Jetson to a third party carrier board to reduce the physical footprint.

For data input, we will use four wide angle USB fisheye cameras connected to a USB hub, which will be connected to the Jetson. A camera will be mounted on each side of the model car to capture the front, left, right, and back of the car. The wider viewing angles provide overlap, which is useful for locating landmarks in video stitching and guaranteeing we have zero blind spots in the stitched surround view.

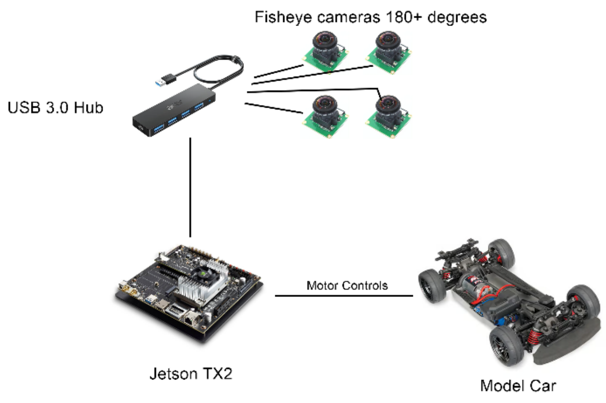

Figure 4 Planned hardware architecture

The amount of data to be streamed from these cameras into the host board and the subsequent image stitching will be a challenge, but it is reasonable to accomplish with the Jetson TX2.

We are considering two designs for mounting the cameras, we call them the wireframe and birdhouse implementations. CAD drawings with a reference image of the chassis are shown below, giving a better visualization of each method.

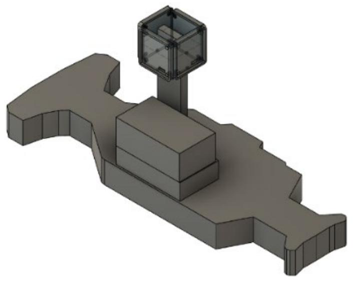

Figure 5 “Birdhouse” camera mounting implantation

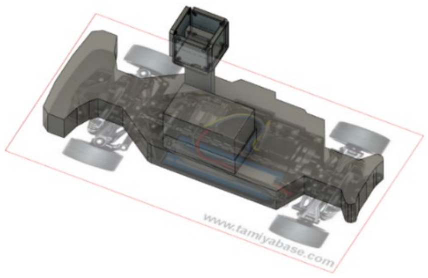

Figure 6 “Birdhouse” camera mounting implantation with car overlay

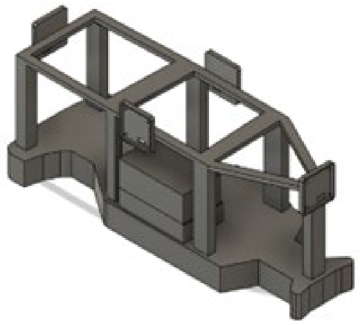

Figure 7 “Wireframe” camera mounting implantation

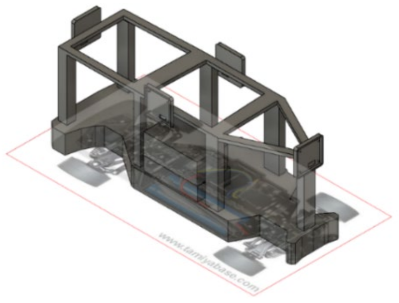

Figure 8 "Wireframe" camera mounting implantation with car overlay

Calculations show that both implementations will be able to fully utilize the 120 degrees vertical and 180 degrees horizontal viewing angles of our cameras. However, we will determine which implementation is best once we test with the cameras in hand.

Vehicle controls are broken down into the steering servo and the motor. Power for the motor and servo will be handled from the ESC supplied from the RC kit. What is missing is control signals for the control wire, which is usually handled by a radio receiver for an RC car. Here we plan to substitute the radio receiver with a direct connection to the Jetson TX2 using GPIO pins configured to be outputs. However, it is unsure that the logic will translate between the two without a level shifter (the Jetson's GPIO is 3.3 volts logic while the ESC and servo logic is at 5 volts). There are also some concerns of delay and waveform resolution from the Jetson as it is also responsible for running the previously mentioned software and the signals may not be clean and/or fast enough. If these issues arise, we will investigate offloading some work to a microcontroller and look to establish communication between the Jetson and microcontroller.

Communication between the operator and Jetson will be done over the Jetson's built-in Wi-Fi capabilities. Here the Jetson will create a hotspot which can be SSH'd into and thus the script can be activated. In other words, the car's autonomous mode can be activated and deactivated over Wi-Fi. We decided against the script turning on automatically to prevent scenarios where the car starts inferencing and moving the car when we are trying to do testing on it.

Finally, if model inference time is slow or has a significant latency from camera input to labelled output, we can implement additional dedicated hardware for acceleration, such as GPUs, FPGAs, or TPUs. If budget and time allows, we can investigate additional sensors to aid in car autonomy, such as LiDAR and sonar distance sensors.

III. DETAILED DESIGN

A. Overview

With an initial conceptual design complete for our project, we've developed a much more detailed and refined system design for our project once we finalized and received our hardware purchases. The workflow of our autonomous model car consists of six phases:

1. Gather input from all four surrounding cameras
2. Construct around view of vehicle with image stitching
3. Run semantic segmentation inference on around view to classify vehicle surroundings
4. Map a start and end point for vehicle based on surroundings
5. Use pathfinding algorithm to map trajectory from start to end point
6. Use pathtracking algorithm to guide car along path

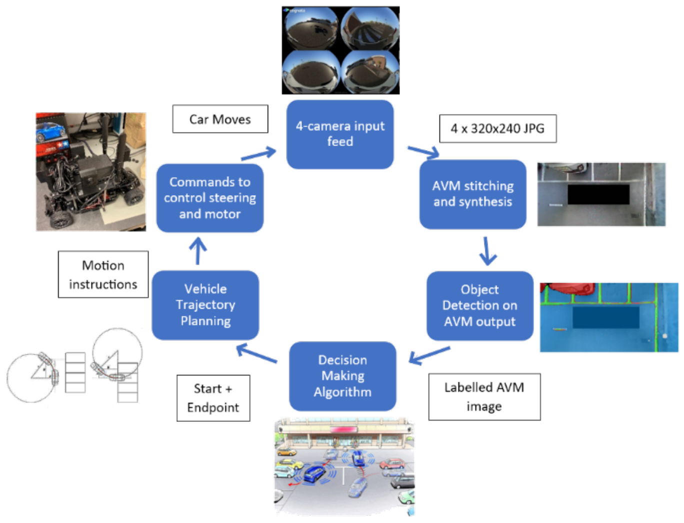
4-camera input feed
4 x 320x240 JPG
AVM stitching and synthesis
Object Detection on AVM output
Labelled AVM image
Decision Making Algorithm
Start + Endpoint
Vehicle Trajectory Planning
Motion instructions
Commands to control steering and motor
Car Moves

Figure 9 Flowchart for Autonomous Parking System

The project's development can be split into two cycles: development of each individual component and the integration of each component. Each of the six phases of the autonomous parking system, illustrated in the flowchart, is considered a component in the project. The following subsections describe, in more detail, the implementation and design of each component.

B. Camera Input

The cameras purchased are 2MP 180-degree fisheye lens cameras running on the MJPEG compression format or YUV compression format. An important note with fisheye lens cameras is their ultra-wide angle creates a distorted view with warping around the edges. For our purposes we are using MJPEG as it has the lowest bandwidth of the two formats. The downside is that the output images are less textured as compared to the YUV. But due to having to undistort the images the lower detail helps make our calculations faster. The maximum resolution of the cameras using this format is 1920x1080p at a framerate of 30 frames per second.

To stitch the camera inputs together we must first undistort the input feed to get an image that can be stitched. That is, the input contains feature points clear enough to stitch together. This requires us to calibrate the camera input using a checkerboard pattern to gain a distortion matrix which will be used on the camera image to undistort the feed leaving an image with undistorted feature points. To do this calibration a checkerboard is displayed onto the cameras and by knowing the exact dimensions of the board and how far it is placed we can accurately judge how distorted the matrix is and in turn undistort it. This calibration is called intrinsic calibrations and once one camera is calibrated correctly, we can use the same distortion matrix to undistort the rest of the cameras. This means only one camera needs to have intrinsic calibrations preformed onto it.

C. *Around View Monitor*

To create a BEV, the floor data needs to be obtained. To do this, we can take the bottom portion of the camera feed, obtained via a second calibration to measure depth of the image and overlay that as the corresponding side of the BEV. This is like intrinsic calibration except now we place a checkerboard a far distance away from the camera and take a top-down photo of the camera and checkerboard. With these we can judge how far away the checkerboard is to the camera and give us an accurate depth matrix. We call this calibration extrinsic calibrations and when we make this calibration, we need to set a reference point on where this camera is placed on the car. This means that we need to run this calibration for every camera on the car. This in turn leads to an image where the car can be assumed to be in the center and each side front, back, left, right have their corresponding camera floor information projected to each side leaving us with an illusion of a top-down image.

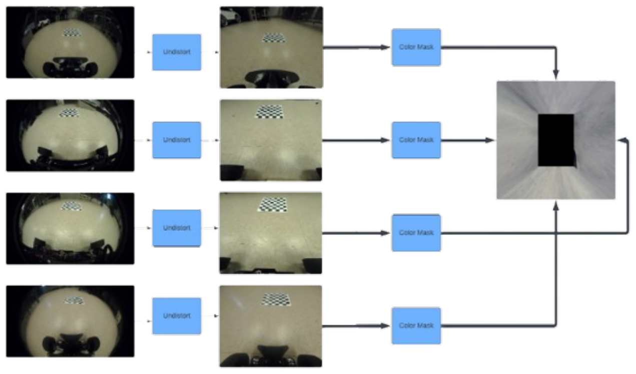

Figure 10 BEV generation workflow

D. *Semantic Segmentation*

A semantic segmentation module was developed to load and run inference on the model by outside programs. In terms of integrating the semantic segmentation model with the other project components in the system design, the model receives a 320x240 pixel JPG image from the around view image processing model. After running inference, the model outputs a segmentation mask of the labeled surroundings including free driving space, obstacles, and parking lines in different color classes.

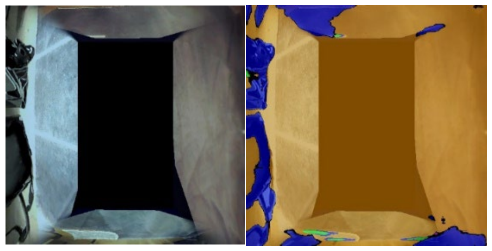

Figure 11 Input (left) and output (right) of semantic segmentation model.

With some complications, we were able to get the model optimized on TensorRT, allowing the model to be run using the onboard GPU. This improved model size, load times, and inference times. This also freed up valuable CPU resources and system memory.

E. *Decision Making*

Due to time constraints, we were unable to implement this part. For now, we used human input to decide an endpoint (as the start point is the current position of the car). This endpoint along with the mask above is then be inputted for the trajectory planning and path tracking.

F. *Trajectory Planning*

For trajectory planning, we used the popular Hybrid A* (pronounced A star). The algorithm takes in the labeled map of the environment produced by the segmentation model as an array. From there, it checks the class of the pixels and if it is seen as parking lines or an obstacle, marks it as undriveable. The algorithm then takes the end point produced from before and maps a path between the two points. We use Hybrid A* because it considers that we do not have omni wheels and sometimes cannot follow a direct path (unlike normal A*).

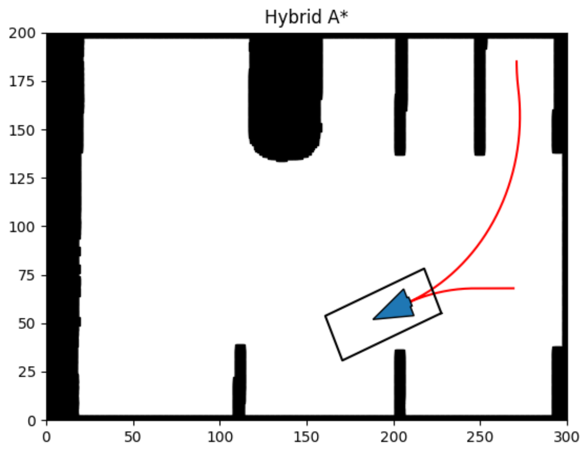

Figure 12 Hybrid A* Visualization

The output is a list of tuples of (x, y, yaw), where x and y are coordinates and yaw is the direction the car is facing (in degrees). For example, a yaw of 90 degrees is facing north, 180 is west, 270 is south, and 0/360 is east. We then take this list and feed it into our path tracking algorithm.

G. *Path Tracking*

For path tracking, we use the Pure Pursuit + PID controller algorithm. Pure Pursuit takes states as inputs and then attempts to follow these states. Pure Pursuit takes in vehicle dimensions to work properly; however, it was designed for a real car. As such, the inputs we gave were 1/10 scale but did not work properly, so a lot of trial and error was used to find input parameters that would work.

The PID controller tries its best to stay on the path. One thing that these two combined account for and not Hybrid A* is momentum. There may be times where you are going too fast/slow, and you may over/undershoot a checkpoint. PID looks to correct this to stay on track.

The result is a list of pairs of (delta acceleration, delta steering angle). The first element is the change in acceleration, where positive meant speed up/press the gas while negative meant slow down/press the brake at that moment in time. The second element is the change in steering angle, where negative meant turn the steering wheel left by that magnitude and positive meant turn the steering while right by that magnitude. This then gets fed into the Jetson which commands the Arduino to do the proper car controls.

H. Hardware Choices

We will start with the car. The TT-02 was chosen as it is a cheap RC car that has already been semi modified by others for robotics projects. In particular, the guide of the JetRacer by Nvidia played a big role in the decision as that team had already figured out mounting. Thus, the mounting holes, hardware, and platform were already planned for us. Using another car platform would require additional time for each of these. The platform from the JetRacer project was meant for the Jetson Nano and a single camera, as such, some modification was needed to fit our use case. In particular, all raised areas were removed as we planned to mount everything in its own enclosure and mate the enclosure to the platform. We also extended the platform to 235 mm (which is the max build size of the Biqu B1, the 3d printer we used).

Our host board decision originally started as a Raspberry Pi, but research revealed that the performance was subpar due to the lack of a GPU/TPU. We then looked for a platform that contained either and stumbled upon the Jetson family from Nvidia. We ended up with the TX2 (an older platform from 2017 - which proved to be problematic later due to software support) because the school had it and it would not remove from our budget. We then transplanted the Jetson to a Quasar carrier board from Connect Tech to reduce the physical footprint. We were limited in selection and Connect Tech is one of the few manufacturers that made a carrier board for the TX2. We lucked out as the school already had this, but we were prepared to purchase this if they did not as there was no other option in our budget. Furthermore, changing from the devkit to the custom carrier required us to use a custom board support package, complicating the flashing process a bit. The result was worth it as we get a massive space savings from 170 mm x 170 mm to 87 mm x 50 mm.

The cameras needed to have a viewing angle of at least 170 degrees, support MJPEG video format, and needed to connect via USB as that was the only connection available on our Jetson. We ended up with the ELP 180-degree USB2.0 fisheye camera.

Figure 13 USB 2.0 camera used

These cameras were then mounted around the car on each side. Our initial testing weeded out our old designs as they were either not strong enough (birdhouse) or too big to fit on a 3d printer (wireframe). We ended up on individual mounts for each camera, resulting in mounting that fit on the 3d printer while not compromising strength. We later superglued angled blocks to angle the cameras toward the ground to capture more usable data (vs the sky).

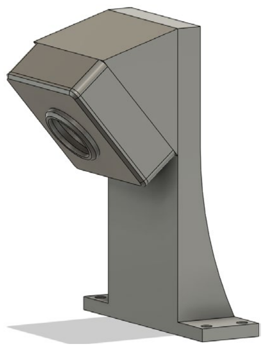

Figure 14 Final Camera Mounting Stands

As a refresher, both power and control for the motor and steering servo are needed. Thankfully, the supplied RC kit has an ESC that will power both. In particular, the ESC supplied with the kit is a Hobbywing 1060. The ESC also has an internal BEC, which steps down the voltage from the battery's 7.2 volts to 6 volts, perfect for our servo's max of 6.8 volts. For control, we need to send pulses ranging from 500 to 2500 microseconds long, 5V "tall" (5V logic). Our initial testing used a direct connection from the Jetson's GPIO to the ESC. Thankfully, our concerns in voltage logic did not seem to effect operation, so a level shifter was not needed. However, there were concerns with delay as the Jetson would be busy with other OS tasks before it would send control signals. Additionally, we had issues with the square waves/pulses not being clean enough. Because of the aforementioned, we switched our hardware structure to include an Arduino microcontroller. Using the microcontroller removed the previous delay and resolution issues but added some complexity. To address the control signals, we set up UART serial communications between the Jetson and Arduino, where the Jetson is the master, and the Arduino is the slave. For UART, we used 8N1 and originally used 9600 baud rate. We attempted to increase this until the hardware did not react as expected and ended up at 115200 baud rate as a reliable but responsive rate.

Finally, the Wi-Fi communication between a user computer and the Jetson did work. We were able to connect via SSH and then activate the necessary programs. However, due to university IT security, we were not able to setup static IPs and SSH in certain buildings, so in practice we used a monitor hooked up to the Jetson to activate the necessary programs.

The final prototype is a 1/10th scale TT-02 RC car with a platform on top, hosting the Jetson, cameras, and Arduino Uno.

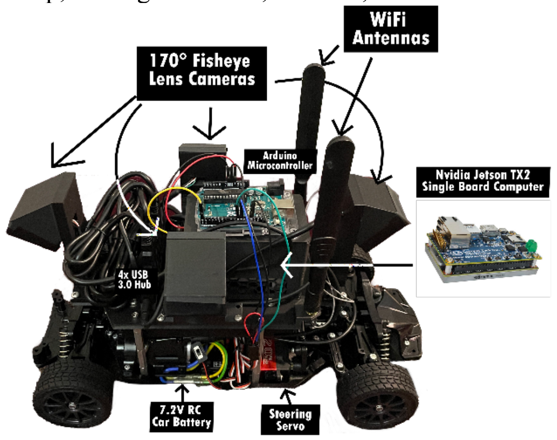

Figure 15 Labeled model car equipped with purchased hardware.

IV. Approach

A. Test Environment

A 1/10th scale parking lot tapestry was purchased to use as our test environment. We can use this tapestry, with the design show in the figure below, to fully test all steps of our autonomous system throughout the development process.

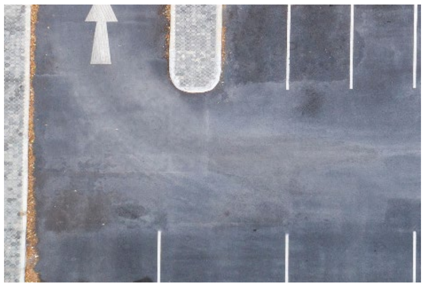

Figure 16 Test parking lot printed onto test environment tapestry.

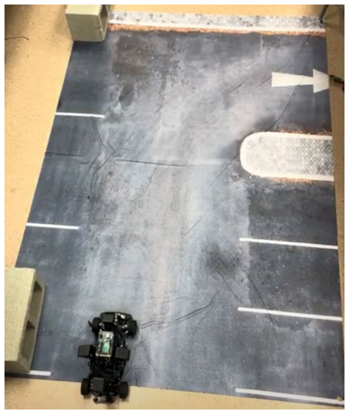

Figure 17 Test environment in lab with model car.

With a physical setup in the lab, the model car's autonomous parking can be tested. Engaging autonomous mode for the model car is signaled over a Bluetooth keyboard wirelessly connected to the Jetson. A main Python script was developed to import the software modules of each project component and to integrate their functionalities.

B. Implementation Challenges

1. Camera Challanges

Four different camera feeds displayed at the same time led to bandwidth issues. Since video stream information is complex it was impossible for us to stream all four cameras at the same time since it was too much for the only USB2.0 data bus on our TX2. We attempted to mitigate the issue by using MJPEG compression format, which is the most bandwidth efficient format and lower resolutions. Ultimately, the solution was to only have 2 cameras on at a time and alternate the video streams. This bypassed the issue entirely, however, was not ideal as the overhead of turning on/off the cameras bottlenecked the system. If budget and time allowed, investing in more expensive CSI cameras and supporting carrier board would buy us more bandwidth.

2. Software Challenges

Most issues during the implementation of the semantic segmentation model on the Nvidia Jetson TX2 arose because of the platform's age as well as its lack of support. The Jetson's age made it EOL, and the latest versions of libraries were not supported on the Jetson. For example, we were stuck with Python 3.6.9 while the newest TensorFlow and ONNX (which is what we wanted to use) needed Python 3.8+. Additionally, finding guides was not helpful as many of the guides only worked on the newer Jetson Nano. We also had to install and build several packages from source to get them working on the Jetson TX2, primarily anything that relied on CUDA.

The reason being is the unusual format of the GPU bus. Normal installation sees the GPU through the PCIE bus, but the Jetson does not have its GPU accessed through the PCIE bus but instead directly from the kernel. The result is that normal installations fail since they do not see the GPU where they expect it to be. Nvidia has created some tutorials to build TensorFlow, TensorRT, and OpenCV from source, but we found that these guides were outdated and had to do troubleshooting on our own. However, once we got it installed, we used CUDA and the GPU to accelerate our workflow.

Additionally, one issue we ran into was running the TensorFlow model on our Jetson's GPU. To do this, we needed to run the model using CUDA or optimize it using TensorRT. However, using CUDA or TensorRT effectively required the model to be in the Keras format. The model was originally trained in TensorFlow 1, but our Jetson used TensorFlow 2. This was an issue because our model was of Frozen Graph format, but TensorFlow 2 switched to Keras format. TensorFlow's documentation to migrate the model to Keras and the scripts to TensorFlow 2 relied on functions that no longer existed. Because of this, we scraped running with

CUDA and the TensorRT preprocessing, instead running it as is through TensorRT to convert the model. The result was not perfect as we were unable to create a full TensorRT engine but settled with a TensorFlow-TensorRT model (tf-trt). The difference is that some (not all) of the tensor operations utilize GPU, but it is better than no GPU utilization. We suspect that converting to a Keras model beforehand and then running it through TensorRT optimizations will produce a full TensorRT engine, thus the whole model will use GPU only.

3. *Time Constraints*

We began working on this project October 2023, finalizing research, goals, and a workflow by December. We then began implementation January 2024. However, due to us needing to wait for specific hardware it was difficult for us to make progress during this time. In the end we were able to implement every module to work independently and came very close to a complete product and if we were given another semester, we are very confident that the model car would make great strides in full autonomy through rigorous testing.

C. *Required Knowledge Base*

With an ambitious project targeting multiple disciplines within engineering, a wide range of expertise from each teammate was the driving force for the progress the team made. In terms of hardware, an understanding of embedded devices and low-level hardware communication via signals and PWM was essential in interfacing the model car's speed and servo controller with the onboard computer. Hands-on experience with handheld tools was also very important in constructing the model car. Computer-animated drafting (CAD) experience was essential in designing the camera and onboard computer mounts on the model car.

In terms of software, a familiarity and knowledge of programming with software dependencies in Python, particularly OpenCV, was essential for implementing both around view monitor and semantic segmentation. Knowledge of camera specifications including field of view, resolution, framerate, compression method, and interface were all essential in selecting an adequate camera for an around view monitor.

D. *Existing Standards Impacting System Design*

Development of our project was most largely affected by the USB standard. Even though USB 3.0 is backwards compatible with USB 2.0 devices, a USB 2.0 device will only operate at USB 2.0 speeds. In the context of our project, the four surrounding cameras are USB 2.0 devices while the USB hub that every camera is connected to is a USB 3.0 hub. Unfortunately, the cameras were limited to the transfer rate of USB 2.0, despite being connected to a USB 3.0 hub, and we consequently had bandwidth issues.

Another set of standards used in our project is the software and operating system of our Nvidia Jetson TX2. OpenCV is a standardized package for image processing in Python and is capable of directly interfacing with camera input devices [11]. Ubuntu is the Linux distribution that was used as the operating system for the single board computer. This largely affected system design because we could only implement supported packages that are compatible with Ubuntu 18.04 for all of our software development.

E. *Regulatory Issues Affecting Design*

All autonomous in the United States are regulated by the NHTSA and USDOT. Companies must work with Federal Motor Vehicle Safety Standards and certify that the company's autonomous vehicle is free of safety risks [12]. Although our model car is merely a prototype, development toward a working product must consider all possible environments, conditions, and situations. On the other hand, since all computation is onboard, there would be no regulatory issues regarding wireless communication. All regulation is concerned with vehicular safety and the reliability of the software guiding the vehicle.

F. *Existing Technology Limitations*

Current autonomous vehicles rely heavily on mapped areas by humans. As such, they are often limited to certain highways or streets that a human has mapped before. This does not work in our situation as there is a lack of human mapped parking lots and parking garages. As such, our vehicle is tasked with mapping the data as it goes, which results in the need for effective AI models and algorithms for on-the-fly navigation. This adds computation overhead to the system.

Data ingress for parking lots is also a challenge that current autonomous vehicles have yet to tackle effectively. Most autonomous vehicles rely on LiDAR for environment data [15]. However, LiDAR relies on depth to be detected. Parking lines do not have much depth difference compared to the asphalt it is painted on, as a result, most LiDAR systems have trouble detecting lines, leaving optical images as the only source of parking line detection.

G. *Results*

With the change from a TensorFlow 1 model to a TensorRT-optimized model, we found interesting results in the model performance and model size before and after the transformation. Our team observed about an 8% difference in inference performance on the model between the two versions. The benchmark ran inference on 500 sample around view images stored on the device of different frames of a car driving in an indoor parking lot. While we did not collect results of inference running on the live webcam feed of the onboard cameras, the observed results of the benchmark indicate a noticeable performance difference on the TensorRT model. Again, we suspect that this performance gap would be greater if our platform supported updated libraries.

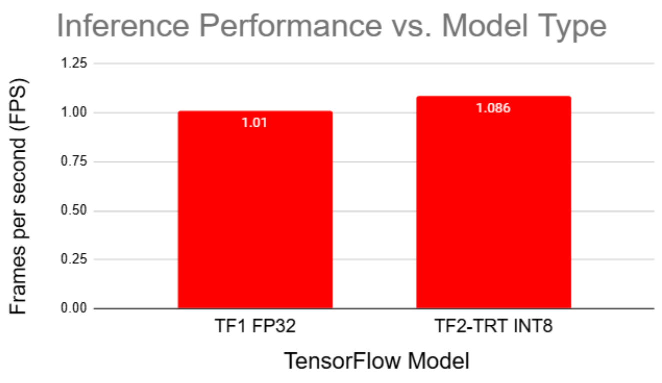

Figure 18 Comparison of performance between original TensorFlow model and ported model with different precision modes.

The TensorRT-optimized model has a significant difference in storage size compared to the original model. It is about 10% the size of the original model, highlighting TensorRT's powerful optimization in terms of model size. This also led to less memory usage during the model loading, leaving more vital memory for the rest of the OS and programs. Future work in further optimizing the model would be to retrain the model from the ground up using TensorFlow 2. The newer version of TensorFlow is better suited for the optimization achieved using TensorRT.

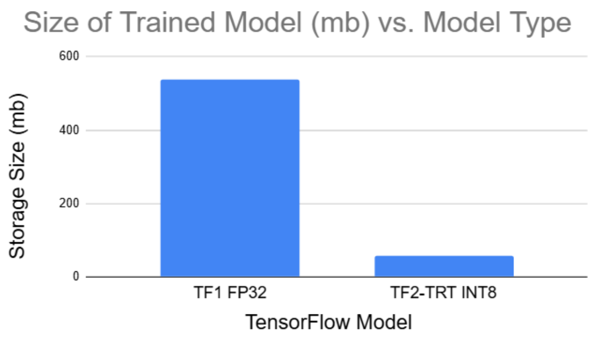

Figure 19 Comparison of storage space of trained model.

V. Cost and Sustainability Analysis

The following will cover the economic, environmental, and social impacts of our project.

A. Economic

The cost of our prototype is $1,080. Using a Jetson Nano instead of a TX2 could have reduced costs, but the opportunity to reduce costs is limited as there are not many cheaper alternatives that provide the same performance. All costs for our prototype model car are outlined in figure 20.

A version scaled to a real car could bring cost savings for drivers due to the autonomous capability. Our design would reduce annual motor vehicle accidents, reducing the amount spent on repairs. The removal of human error in parking lot navigation and lower accident rates could also prompt insurance companies to lower rates as cars become safer.

	Cost Analysis	
	Part Name	***Price (USD)***
1.	Jetson TX2	$399
2.	Jetson Carrier Board	$255
3.	USB 2.0 Cameras	$179.96
4.	TT-02	$109.20
5.	12V Battery Pack (6Ah)	$51.99
6.	RC Battery	$31.49
7.	USB 3.0 Hub	$14.99
8.	Servo	$13
9.	SD Card	$11
10	Mounting Hardware	$11
11	3D Printing Filament	$3.46
12	Total	$1,080

Figure 20 Cost Analysis Table for Prototype

The prototype can be mass produced at a much cheaper rate given changes to the parts used and accepting some performance loss as a result. The chart below shows the mass-produced prototype cost analysis:

	Mass Produced Cost Analysis	
	Part Name	***Price (USD)***
1.	Jetson Nano	$149
2.	USB 3.0 Cameras	$100
3.	TT-02	$109.20
4.	12V Battery Pack (3Ah)	$40
5.	RC Battery	$25
6.	USB 3.0 Hub	$10
7.	Servo	$8
8.	SD Card	$4
9.	Mounting Hardware	$8
10.	3D Printing Filament	$2
11.	Total	$426

Figure 21 Cost Analysis Table for theoretical mass-produced version

Using a Jetson Nano would remove the need for a custom carrier board as the Nano is already small. The change would also allow a smaller battery pack at the cost of reduced runtime.

Our design, by itself with the technology it uses would not provide any tax incentives at this moment, but paired with EVs (Electric Vehicles), which do carry tax incentives in some states and countries, could serve as an incentive for customers to purchase EVs. Owning an EV could also motivate customers to invest in renewable energy and reduce their overall carbon footprint even further, all of which currently have tax incentives set in place already.

Price fluctuations because of resource availability would not be a concern for most of our parts except for the Jetson TX2/Nano which can encounter supply chain issues with their semiconductors. This vulnerability with the main component of our design could impact the overall cost of our prototype and impact its ability to be mass-produced.

B. Environmental

The environmental impact of our product would be no different to modern vehicles regardless of autonomous capability or not. Its carbon footprint would mostly depend on whether the technology was implemented on a fossil-fuel-powered motor vehicle or an EV. As a result, there would not necessarily be a change in consumption or use patterns.

The most significant impact our product could have stemmed from its dependence on semiconductors capable of running the models necessary for object detection, AVM, path planning, and trajectory planning all quickly and efficiently. The potential increase in these powerful semiconductors would increase the demand for the materials needed to manufacture them such as rare earths, and silicon, as well as increase the overall demand for semiconductors, which have already experienced shortages in the past due to increasing demand.

C. Social

Our product was designed to reduce the number of annual motor-vehicle accidents occurring in parking lots/garages, positively impacting people's lives by reducing the annual cost of damages due to accidents and saving lives. Our design meets the community's need for safer forms of private transport in a world of increasing distraction and shortening attention spans.

The design has the potential to change consumption patterns in favor of vehicles with autonomous capabilities. Most hesitance towards autonomous vehicles come in the form of uncertainty in the safety of such vehicles but our design would only make the vehicles safer, increasing trust in autonomous vehicles.

The full-scale model of our prototype would require a team of software and embedded engineers with experience in ML/AI to complete this transition, directly increasing the demand for their skills. It would also increase the demand for safety inspectors with experience working with and grading autonomous vehicles to ensure that the autonomous capabilities are functional, consistent, and reliable.

VI. Conclusion/Summary

We were able to design and implement the foundation for an autonomous vehicle that could self-navigate and park in a parking lot. We recognize that the environment and problem is challenging, and current companies are just beginning to tackle it. In our project we encountered such challenges that forced certain design elements (like using optical data ingress or Hybrid A*). Although the implementation is not fully complete, we've implemented a working around view monitor and semantic segmentation model and integrated both with a model car. At the very least we have provided a good foundation for others to take our workflow and expand on it.

The result should be something that can be scaled up to a full-sized car with the addition of more powerful computer systems. This vehicle could add to car autonomy and contribute toward an overall safer world.

VII. Acknowledgment

We would like to thank Dr. Hang Liu and the ECE Department for their support.

VIII. References

[1] Distraction Can Often Cause Parking Lot Injuries. https://www.nsc.org/road/safety-topics/distracted-driving/parking-lot-safety.

[2] Parking Lot Safety - National Safety Council. (n.d.). Www.nsc.org. https://www.nsc.org/road/safety-topics/distracted-driving/parking-lot-safety

[3] Parekh, D., Poddar, N., Rajpurkar, A., Chahal, M., Kumar, N., Joshi, G. P., &amp; Cho, W. (2022). A review on Autonomous Vehicles: Progress, methods and challenges. Electronics, 11(14), 2162. https://doi.org/10.3390/electronics11142162

[4] Admin. (2022, July 15). What is an autonomous car? - i-ECU. i. https://i-ecu.com/what-is-an-autonomous-car/

[5] Do, T., Duong, M., Dang, Q., & Le, M. (2018). Real-time self-driving car navigation using deep neural network. Paper presented at the *2018 4th International Conference on Green Technology and Sustainable Development (GTSD),* 7-12. 10.1109/GTSD.2018.8595590

[6] Gupta, A., Anpalagan, A., Guan, L., & Khwaja, A. S. (2021). Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues. *Array, 10*, 100057. 10.1016/j.array.2021.100057

[7] Liu, Q., Li, X., Yuan, S., & Li, Z. (2021). Decision-making technology for autonomous vehicles: Learning-based methods, applications and future outlook. Paper presented at the *2021 IEEE International Intelligent Transportation Systems Conference (ITSC),* 30-37. 10.1109/ITSC48978.2021.9564580

[8] Afdhal, A., Saddami, K., Sugiarto, S., Fuadi, Z., & Nasaruddin, N. (2023). Real-time object detection performance of YOLOv8 models for self-driving cars in a mixed traffic environment. Paper presented at the *2023 2nd International Conference on Computer System, Information Technology, and Electrical Engineering (COSITE),* 260-265. 10.1109/COSITE60233.2023.10249521

[9] Jang, C., Sunwoo, M. Semantic segmentation-based parking space detection with standalone around view monitoring system. *Machine Vision and Applications* **30**, 309–319 (2019). https://doi.org/10.1007/s00138-018-0986-z

[10] K. Seshadri, B. Akin, J. Laudon, R. Narayanaswami and A. Yazdanbakhsh, "An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks," *2022 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, 2022, pp. 79-91, doi: 10.1109/IISWC55918.2022.00017.

[11] Open Source Computer Vision. https://docs.opencv.org/4.x/index.html.

[12] Automated Vehicles for Safety. https://www.nhtsa.gov/vehicle-safety/automated-vehicles-safety

[13] Wang, L., Yu, W., & Li, B. (2020). Multi-scenes image stitching based on autonomous driving. 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). https://doi.org/10.1109/itnec48623.2020.9084886

[14] USB Accelerator, Google Coral. https://coral.ai/products/accelerator/

[15] Y. Li and J. Ibanez-Guzman, "Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems," in *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50-61, July 2020, doi: 10.1109/MSP.2020.2973615.

[16] Accelerating Inference in TensorFlow with TensorRT User Guide (2024). https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.