RASPBERRY PI DRIVEN VISION FOR AUTONOMOUS MOBILITY

Andrei Tumbar

Computer Engineering
Rochester Institute of Technology
at1777@rit.edu

ABSTRACT

Classical computer vision has always been a challenge to implement on embedded systems because of the requirement of fast computational power and system memory. This paper presents a possible solution to realtime computer vision aboard a Raspberry Pi 3B for the purpose of driving an autonomous vehicle.

1. INTRODUCTION

Vision processing is a fairly well developed field that looks at processing images to extract certain features. This can include features such as object edges, object classification, or 3D mesh reconstruction with stereo correlation. One of the major challenges of computer vision in robotics and other embedded systems arises from vast computational requirements.

Many robots that operate from the input of a camera will use some form of computer vision to drive their navigation logic. For example, the Perseverance rover uses a combination of its 6 engineering cameras as well as an inertial measurement unit for path planning and pose estimation [1]. One of the main challenges with computer vision for robotics and other embedded system arises from the large computational requirements of vision software. While algorithms are parallelizable making graphics processing units (GPUs) highly attractive, power constraints of these robots usually factors into the choice of control.

There has been some work in the past done to implement vision on the Raspberry Pi. [3] discusses a Raspberry Pi configuration that uses the Raspberry Pi camera module. OpenCV is used to support much of the vision functionality. This project will use a similar hardware configuration, however, will differ in operating system and programming language when comparing to [3]. Instead, a custom compiled, statically linked version of OpenCV is built as well as a leaned down version of Debian Linux is used instead of the one noted in [3].

This paper introduces a vision and navigation system aboard a Raspberry Pi 3B. The official Raspberry Pi Camera Module V2 is used for its impressive streaming features and hardware noise reduction. The vision system is meant to drive a small battery powered car around a track. The Raspberry Pi's low power requirements of $3.7\,\mathrm{W}$ at maximum load of all four internal cores makes it a good option for this application [2]. This paper is divided into three main sections. The background will explain the general design of the car and introduce ideas such control systems used in later sections of the paper. The proposed method section will discuss the specific implementation details of the major components of the car. Finally, the results section will discussed what went wrong, what went right, and future work to continue this project.

2. BACKGROUND

2.1. TI Cup Car Hardware

The purpose of this project is to drive a TI Cup Car. This car includes two direct current (DC) motors used to drive the rear wheels as well as a servo motor to control the steering arms. All three motors are driven with a pulse-width modulation (PWM) signal. The DC motors have a base frequency of $10\,\mathrm{kHz}$ while the steering servo uses a frequency of $50\,\mathrm{Hz}$. Due to the low current limitations of a software drivable general purpose input/output (GPIO) pins, the DC motors cannot be switched directly from the pin headers. A motor driver must be used to switch an external power source controlled by a low drive GPIO. This motor driver is an H-bridge circuit able to switch the high current requirement of the DC motors from the external battery under the control of a $3.3\,\mathrm{V}$ GPIO PWM signal.

The generation of these PWM signals is done by an MSP432P401R microcontroller. This microcontroller is capable of generating multiple simultaneous hardware driven PWM signals and can be used to drive multiple motors. The microcontroller is used instead of the Raspberry Pi as the Pi is only able to generate two hardware driven PWM signals and therefore not able to control the car on its own.

2.2. Camera, Vision & Navigation

The software built to control the vehicle is divided in three main stages:

1. Capture camera frame
2. Detect and threshold track edges
3. Compute path and control car to stay on path

The first stage involves capturing raw images from the camera. The Raspberry Pi Camera Module V2 will capture raw image frames through a fisheye lens. The next step will perform vision processing to raw image frames. Processing is divided into small stages. The collection of processing stages is known as the vision pipeline. To perform certain processing steps, the open-source OpenCV library is used. OpenCV is split into independent modules which are can be built as separate static libraries. This allows a user to compile OpenCV with only a subset of its features to keep the code sizes relatively low. This project uses the image processing module (imgproc) as well as the 3D camera calibration module (calib3d) to support its pipeline.

The final stage in the vehicle control process will take processed images from the vision pipeline and plan a path for the car. It will classify track edges as splines and attempt to find a path within the edges of the track. This stage will keep the car on course using the Proportional-Integral-Derivative (PID) control paradigm.

PID is a control method that looks at bringing a generic system to a desired state while keeping overshoot oscillation and steady state error at a minimum. The basic idea is that given a set of inputs, a system can determine its current state and provide controls to the system to bring it into a desired state. In the context of the car, the desired state is a center position on the track and PID will control the steering on the car.

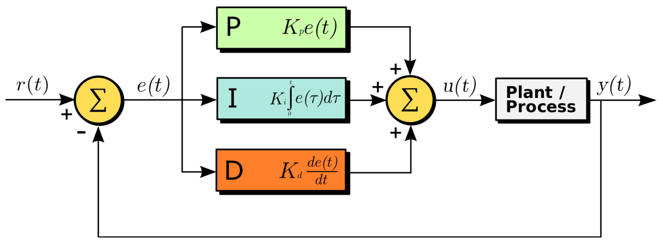

Fig. 1. Proportional-Integral-Derivative control driving mathematics.

Fig. 1 shows the driving mathematics behind the PID control system. $r(t)$ being the current estimated state of the system derived from the various sensors on-board. $e(t)$ is the error, which, as previously mentioned, can be derived from a desired position and a current position. The three terms that control the feed sent back to the robot are weighted by parameterizable coefficients. These coefficient are heavily system dependent and must be manually tuned depending on physical characteristics of the robot and control algorithm. The three parameters, K_p, K_i, and K_d each react on the error correction of the robot in their own unique way. Here is a table of the effects each coefficient have on the error response of the robot.

Coefficient	Too low	Too High
K_p	System will not reach target state before it becomes unstable.	System will oscillate around the target state.
K_i	System will undershoot the target during steady state.	System will overshoot the target during steady state.
K_d	System will reach the target slowly.	System will attempt to reach the target too quickly and cause overshoot.

Using the guidelines outlined in table above, the PID parameters may be tuned to allow the car to stay on the desired course. During this process, it is best to start with K_i and K_d of 0 as the system is still able to operate without these parameters. Adjusting three parameters all at once can become very difficult. For the purposes of a vehicle such as this one, the integral gain in the PID can even be left at zero for the final design. The steady state of the car is not reachable at high speeds in turns. The derivative gain will allow the system to adjust faster to the given desired position as it is able to take the previous state and see how its adjustment effected the new state of the system.

2.3. Software Framework

All of the subsystems of the car cause great complexity in the overall design of the software. For this reason, each component is split up into independently operating subsystems. Messages can be sent synchronously or asynchronously between components to pass data and requests between them. Synchronous message dispatch occur in the thread that the request was made, asynchronous messages conversely will be dispatched in the thread of the component in question.

The component and messaging schema discussed is a very common software design for small and medium sized robotics projects. NASA's Jet Propulsion Laboratory has created a general purpose framework for just this purpose – FPrime [4]. In addition to the ability to model the generic connections and messaging between components, FPrime also provides many tools and features that allow developers to focus on the design of their robot's software. This includes features such as a commanding and sequencing paradigm in which components may define their own commands. FPrime is able to compile a script-like sequence of these component commands [4]. FPrime also allows components to define their own events (EVR - EVent Record) and telemetry (Tlm) to provide the developers and robot operators with status logs and health statistics about the system. These EVRs and Tlm may be viewed in FPrime's in-browser interface called the Ground Data System (GDS).

2.4. Bill of Materials

The car extends the standard TI Cup Car kit by adding a Raspberry Pi and a new camera. Here is a bill of materials for the car including prices for each part.

Part	Price ($)
Raspberry Pi Camera Module V2	46.00
Pastall Raspberry Pi Camera Cable	7.99
Standard M12 Thread Interface 180° Fisheye Lens, 1.44mm 1/2.5 Wide Angle	16.08
LoveRPi Performance Heatsink	4.31
Servo Steering Arms	17.99
Motor Driver RB-WAV-77	28.99
Chassis Kit ROB0170	98.75
Brushed Motor Kit KIT0167	25.00
MG996R 55g Metal Gear Torque Digital Servo	18.88
Reusable Zip Ties Assorted Sizes	18.99
UCTRONICS 0.96 Inch OLED Module 12864 128x64 Yellow Blue SSD1306	6.99
Tenergy 7.2V Battery Pack for RC Car	39.99
Sourcingpower Universal RC Battery Charger for 3.6V-9.6V	19.99
MSP432P401R	30.00
Raspberry Pi 3B	42.63
Beato PCB adapter for motor driver	10.00
Total	413.59

3. PROPOSED METHOD

3.1. Hardware

The car operates with two compute boards. As previously mentioned, an MSP432P401R microcontroller is used to provide the PWM signals to the motors. There are two throttle motors on the rear wheels and a single servo that will control the servo arms. The Raspberry Pi runs the rest of the software that controls the camera and performs communication with GDS.

Fig. 2 shows the basic wiring scheme of the car. An Inter-Integrated Circuit (I2C) connection is used to send commands from the Raspberry Pi to the microcontroller. An organic light-emitting diode (OLED) display is also connected

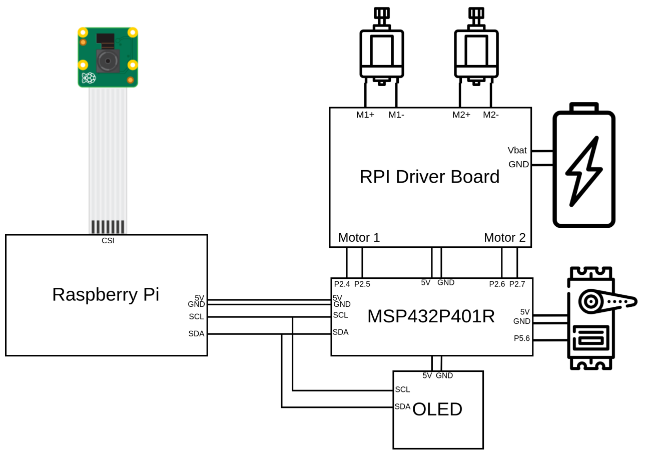

Fig. 2. Hardware wiring of Raspberry Pi, MSP432P401R and other hardware.

to the common I2C bus. The display is used to print the internet protocol (IP) address of the Raspberry Pi on boot-up so that a user will know where to shell into via a secure-shell (SSH) connection. The Raspberry Pi is the master device in the I2C chain. The Pi is able to arbitrate the common I2C bus by providing the device's unique address before each packet. This will allow the Pi to command both the OLED display and microcontroller via the same I2C interface.

The Raspberry Pi Camera Module shown in the upper left corner of Fig. 2, is connected via a ribbon cable to the Pi's Camera Serial Interface (CSI). Due to field-of-view (FOV) limitations of the default camera module's lens, a fish-eye lens with 180° FOV is used.

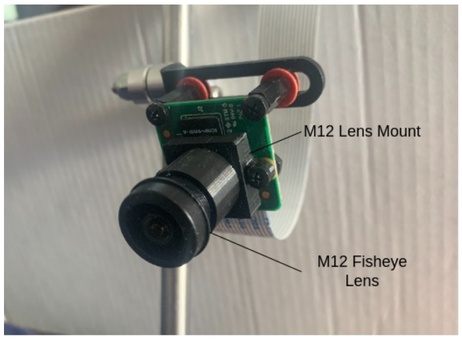

Fig. 3. Camera lens and mount.

Fig. 3 shows the physical configuration of the fish-eye lens with the camera module. The M12 lens mount shown in Fig. 3 is a 3D printed lens mount meant for the M12 thread present on the lens in use. Although this lens will cause considerable distortion, it is vital to use a high horizontal FOV lens to be able to see both edges of the track immediately next to the car. The default lens on the camera module has a horizontal FOV of $53°$ [6]. A small horizontal FOV causes the camera's viewport to cut off the peripheral vision the car needed for this application.

3.2. Component Topology

This section will discuss the general layout and interaction between all the system components. Connections between components indicate some relationship or flow of data such as an image transfer or message dispatch.

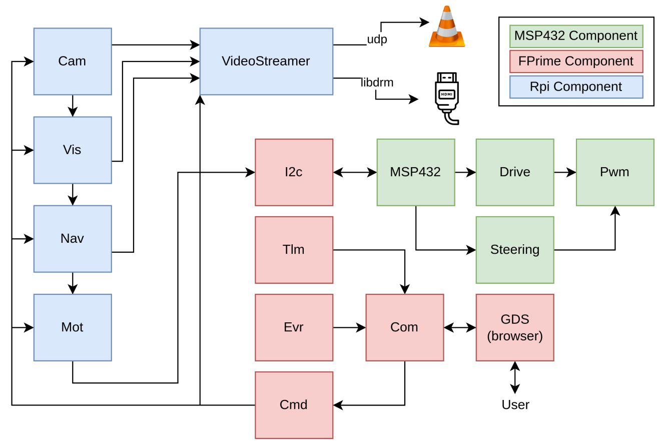

Fig. 4. Full system component topology.

Fig. 4 shows the basic component layout of the project. The camera will acquire image frames and is able to stream to the VideoStreamer or the vision component (Vis). Similarly, Vis is able to stream to the VideoStreamer or to the navigation component (Nav). The video streamer is a component that is able to draw image frames on an HDMI display or to encode a sequence of frames as an H.264 video live stream. This live stream can be viewed externally by a VideoLAN (VLC) media player client. The VideoStreamer is imperative to development and debugging as image frames may be annotated to show algorithms in action.

The MSP432P401R microcontroller is connected to the I2C bus and, as discussed before, will provide the motor control for the DC and servo motors. Both the steering and the dc modules are small wrappers around the pwm module. The pwm module will interface with the TimerA peripheral on-board the MSP432P401R to generate hardware triggered PWM waves.

3.3. Motor (Mot)

The microcontroller's sole purpose in this design is to control the motors. The Raspberry Pi needs a communication protocol for encoding command packets sent to the microcontroller. The 'Mot' component will handle this interaction by encoding a command packet and sending it out over the I2C bus. A command packet will consist of two single byte fields: opcode and value. Opcodes denote a specific command such as 'left_forward' or 'right_backward' and will take an argument in the value field.

Mot will allow control from other components via a more generic synchronous messaging port. The steering port will allow floating point values from -1 (left) to 1 (right), while the throttle port will allow two throttle values from -1 (backwards) to 1 forwards for each of the two DC motors. Mot is therefore responsible for encoding a software friendly floating point number, to a hardware friendly command packet to send to the microcontroller.

3.4. Camera (Cam)

The camera component (Cam) is responsible for interfacing with the camera. This software will use libcamera to interface directly with the kernel driver and hardware. Once a camera stream is initialized, Cam will need to manage the frame buffers sent and received from the camera. Image acquisition requests can be made to the camera by queuing a frame buffer for the camera to work with. When an image is acquired, the camera will send a filled image buffer back to Cam and Cam will then send out the frame to a registered listener.

Looking at Fig. 4, Cam, Vis, Nav, and VideoStreamer are all interconnected. These connections refer to image frames being passed through the components via asynchronous messaging. It is the responsibility of each component to either pass the image frame to another component, or to return the image frame to Cam. When the frame is returned to Cam, it is reused for acquisition by placing on the frame buffer queue.

For use with the car, the camera will capture 1640x1232 pixel image frames at 30 frames per second (FPS). The camera's saturation is zeroed as a gray-scale image is used for image processing. The white balance is set to auto which will help calibrate the color balance in different lighting environments.

3.5. Vision (Vis)

The Vis component is possibly the most important component in the robot. The purpose of this component is to process the image frame in such a way that the navigation component can use it for controlling the car. Vis will receive raw image frames from the camera and apply a series of transformations to the image. Each transformation is known as a pipeline stage where the culmination of all the stages is known as the pipeline.

The entire pipeline in Vis is completely modular. This means that the pipeline stages can be added or the entire pipeline cleared via a set of sequence commands. Pipeline stages may have parameters associated with them which can similarly be set via command. This modular design allows for easy changes to be made to the vision pipeline as well as allowing multi-purpose use of the vision component. For example, a race vision pipeline can be initialized during the race boot-up sequence to perform edge detection. Similarly, a calibration pipeline can be set up to compute the camera pose and be used for perspective correction. Another advantage of this method is simplicity in implementation. Each pipeline stage is an extremely simple processing step and therefore reduces the chance of programming errors.

3.5.1. Edge Detection

When designing the race pipeline, the input into the navigation must be considered. The purpose in the Vis component is to apply transformations to the image that will provide Nav with information to place itself on the track. For this project, the most important features to extract from the image frame are the edges of the track. We would therefore expect Vis to transform the raw image to only include track lines. This can be done with an edge detector.

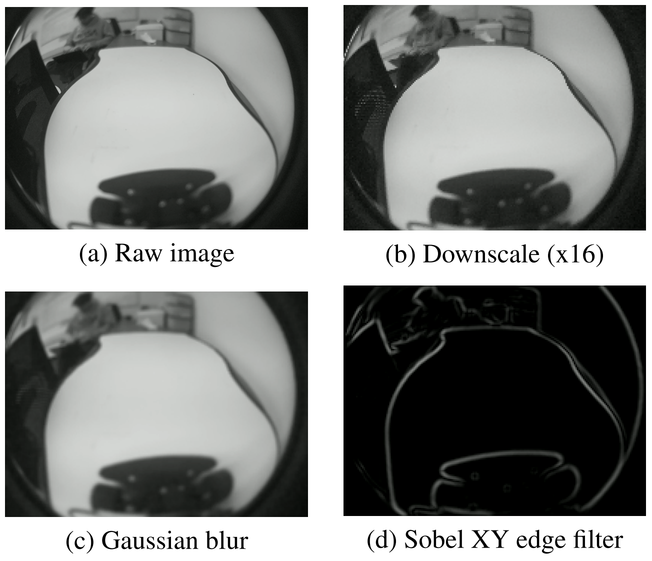

Fig. 5. First four stages of the race pipeline applied to the same image.

Fig. 5 shows an image of the car on a section of the track. This test is especially interesting because the track is next to a white wall. With a 16x downscale applied to the image shown in Fig. 5 (a), it is difficult to make out the edges of the track even for the human eye. The Sobel filter however is able to appropriately find these edges using its convolution kernel by applying filter similar to a 2 dimensional gradient. The purpose in applying a Gaussian blur filter to the image before the Sobel filter is to essentially apply a low pass digital filter. This will smooth out noisy regions as well as make track edges thicker to get rid of small gaps in the line edges. Like the Sobel filter, the Gaussian blur is a convolution kernel. However, instead of applying a gradient functor, a weighted average of surrounding pixels is used. This racing pipeline will use a 3x3 Gaussian blur kernel as it has reasonable computational performance and yields an acceptable output.

3.5.2. Pose Calibration

When working with camera's on robots, it is very common to calibrate the pose of the camera [1] [3]. This refers to computing the rotation and translation matrix relative to a known point on the ground plane. Calibrating the pose would allow the image frame to be transformed into "birds-eye" view orientation. This orientation is obviously advantageous to the navigation algorithm as its easier to form a navigational path when looking directly above. Depth estimation on a planar surface is trivial when looking from directly above.

OpenCV supports pose calibration via its calib3d library. This project uses a chessboard calibration target as it is the most commonly used target for pose calibration. This is simply a set of black squares on white paper tapped to a piece of cardboard and layed out in-front of the car. OpenCV is able to find the corners between the squares and provide the user with a set of image coordinates. Because these corners are at known positions on the calibration target, OpenCV is able to derive pose information from the location of the these squares.

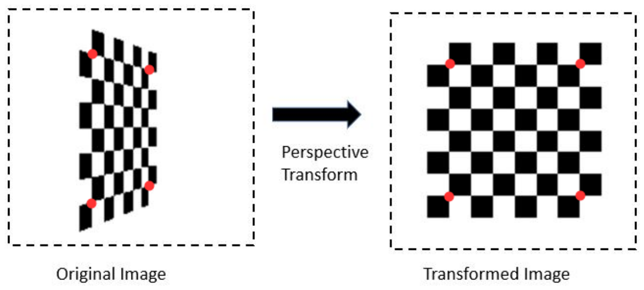

Fig. 6. Perspective transform applied to chessboard image.

Fig. 6 shows an example of a chessboard with a perspective transform applied to it. To build the transformation matrix, two sets of four points must be provided. The first set of points lies on the raw image and maps out the input plane of the transform. The second set of points will lie on the transformed frame which, in this case, should be evenly spaced to warp the image to birds-eye view.

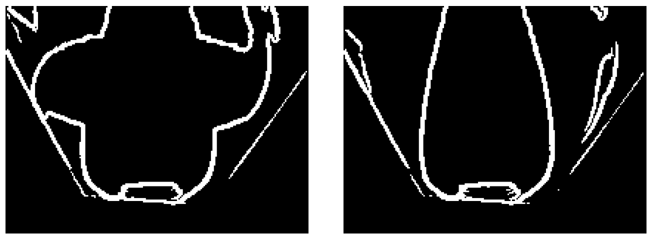

Fig. 7. Vis processed images of intersection and straightaway.

The full vision pipeline's results shown in Fig. 7 show a combination of processing done in Fig. 5, a warp transform, and finally a binary threshold to create hard edges. Fig. 7 shows very good results near the center of the image with relatively straight lines on the straightaway (right). The distortion around the edges of the frame are caused by lens distortion from the fish-eye lens. Fish-eye lens distortion is a common problem in robot that require a large FOV such as [1]. The distortion can be corrected with the calibrated camera model discussed above. This transform however is too computationally intensive for the Raspberry Pi and cannot maintain a stable frame stream for the navigation step.

3.6. Navigation (Nav)

The final step to control the car is to ingest Vis processed image frames and to derive a desired path around the track. The idea here is to find points on the track edge and build a spline that draws out the edges of the track.

3.6.1. *Line & Center classification*

The classification of track lines is a relatively in-depth process. A method that utilizes OpenCV's contour finding HoughLinesP algorithm was initially investigated. This path was immediately dropped as the algorithm was far too slow to be used for realtime contour classification and can only be used on straight lines. This project uses an iterative algorithm to trace lines and find track edges.

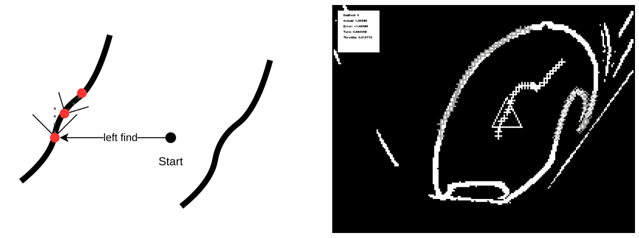

Fig. 8. Vis processed images of intersection and straightaway.

Fig. 8 shows the basic idea behind the iterative algorithm used for line classification. A configurable start point on the image is used to search for the left and right track lines. A linear search for the edge is performed for both left and right edges. When they are found, a step process is used to trace along the path of the line. The next point is found by searching along a circle of configurable radius out from the current point. Fig. 8 left shows three lines protruding from the detected point. The center dotted line is the start search angle which will be equal to the angle that the last point is found at. The solid lines represent the angle restrictions placed on the angle sweep search. By iterating along this search path, a set of points can trace the track lines. with relatively high accuracy.

Although the track lines can be detected well, the center line is more difficult to classify. Fig. 8 right attempts to draw a center through the detected track edges. The left and right track edges were properly detected with a fairly high degree of accuracy. Center points are derived by simply averaging the points from the left and right sides of the track. This evidently cannot work as the outer edge of the track will have a longer arc than the inner edge of the track. A similar method attempts to find the center line by approximating the track edges as 2D parametric splines. Theoretically, by tracing a normal line from one of the edges, the line should intersect with the opposite point on the track. In practice however, the distortion due to the fish-eye lens causes the normal lines to lack this mathematical property. The result of a poorly mapped center line makes it difficult to control the car through turns.

3.6.2. *PID Control*

The final control of the car is done by providing a PID process with a current error estimate. Using the perspective transform, the center of the image corresponds to the center line of the car. By looking at the track's center line derived in the previous step, we can derive the distance the car is from the center of the track. Depending on the current speed of the car, points on the center line may be used to derive an error estimate.

The PID controller can be tuned to adjust the response of the car given error over time. PID parameters can be set via command and saved to disk using FPrime's parameter database. Because the center line classification is so unreliable in the corners, a good set of PID parameters was never found.

4. RESULTS

Although the car could travel very slowly (about 25 DC duty cycle) around a simple track, the navigation of the car suffered from a host of problems that caused the final product to not be ready for race day. This section of the paper looks to discuss what went wrong, what went right, as well as provide possible future work that could improve the performance of the car.

4.1. What went wrong

One of the issues seen throughout many portions of this project was performance. Although many steps were taken to minimize the effect of the low compute power of the Raspberry Pi, all four Pi cores were running at maximum load when the race pipeline was processing frames. Performance issues attempted to be solved by downscaling the raw image frame. While this worked for many of the linear and convolutional operators, an undistort operation for correcting fish-eye lens distortion could never be added to the pipeline.

Although performance was an issue, the main problem in this project was the overall complexity of the navigation. While the vision pipeline provided surprisingly good images with little more than track edges showing, the chosen navigation algorithm was not reliable for every track configuration and car orientation. The car could handle well on straight portions and single turns however suffered from misclassification of track lines when multiple turns were chained together.

4.2. What went right

Although ultimately the car never started the race, many portions of this project were successful. The motor control across the I2C bus was built to be robust and could handle packet sending errors if a physical connection became unreliable. The camera and vision system were designed well and worked with little flaws. There was a lot of work that went into getting a reliable raw stream from the camera as well as encoding the video stream and transmitting it over the network. Much of the time in this project was spent getting the basic features working. This took time away from developing a reliable navigation algorithm.

Finally, the downlink capabilities for EVRs and telemetry as well as uplink of command sequences and dispatch were essential in providing a high quality development environment. This capability made it possible to quickly build and test new vision pipeline features as well as set parameters for certain stages. Without the modular design of the Vis component and the capabilities of the FPrime framework, this project would never have gotten as far as it did.

4.3. Future work

Much of the framework and features on this project are not likely to be changed significantly in the future. The Cam, Vis, Mot, and VideoStreamer are all fairly well thought out and do not require any significant redesign. To make this car operate at its full potential, the method for navigation needs to see further work. Possible routes for improvement include applying an adaptive threshold instead of an edge detector to classify raw image pixels as ‘track’ and ‘no-track’. This may eliminate the performance issues as well as possibly providing navigation algorithm with a simpler processed frame to work with.

Another future design change could implement a neural net control system. A neural net could be relatively easy to train using the framework already in place. A game controller could be used to manually drive the car around a track and provide the net with training data. The net could be further refined by providing elapsed time feedback to the learning algorithm that could further adjust pathing. A neural net could work in tandem with the vision system already in place and would simply work with a smaller region of interest to improve performance.

Finally, the choice of a Raspberry Pi was mostly due to the hardware availability of the board. Ideally, the NVIDIA Jetson Nano computer should be used.The Nano includes an enormously powerful 472 giga-floating-point-operations-per-second (GFLOPS) GPU [8]. By comparison, the Raspberry Pi’s four cores can produce about 3.62 GFLOPS [2]. The Nano also does not suffer from virtually non-existent documentation for its GPU which makes it far more supported in open source tools such as OpenCV.

5. CONCLUSION

This project implemented a vision pipeline on a Raspberry Pi to drive a TI Cup car using a Raspberry Pi Camera Module. An edge detector with a perspective transform are used to extract track features from a raw image stream. Although the chosen navigation algorithm did not provide a reliable control system, future work can extend this project to properly control the car. This design has the potential to produce a very fast car if more work is done in the final control step. Many engineering concepts were applied to this project for both the software and hardware design of the car including, network communication, computer vision, I2C communication, frame buffer management, PID control, video encoding, and PWM for motor control.

6. REFERENCES

[1] J. N. Maki, D. Gruel, “The Mars 2020 engineering cameras and microphone on the Perseverance Rover: A next-generation imaging system for mars exploration - space science reviews,” SpringerLink, 24-Nov-2020. [Online]. Available: https://link.springer.com/article/10.1007/s11214-020-00765-9. [Accessed: 20-Apr-2022].

[2] “Power consumption benchmarks,” Power Consumption Benchmarks — Raspberry Pi Dramble. [Online]. Available: https://www.pidramble.com/wiki/benchmarks/power-consumption. [Accessed: 18-Apr-2022].

[3] A. Rosebrock, “Raspberry Pi For Computer Vision,” PyImageSearch, 08-May-2021. [Online]. Available: https://pyimagesearch.com/2019/04/05/table-of-contents-raspberry-pi-for-computer-vision/. [Accessed: 18-Apr-2022].

[4] R. L. Bacchino, T. K. Canham, G. J. Watney, L. J. Reder, and J. W. Levison, “F Prime: An Open-Source Framework for Small-Scale Flight Software Systems,” NASA JPL Beacon, 04-Aug-2018. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/48425. [Accessed: 20-Apr-2022].

[5] "A brief guide to the F´ ground data system," F´. [Online]. Available: https://nasa.github.io/fprime/UsersGuide/gds/gds-introduction.html. [Accessed: 20-Apr-2022].

[6] "Raspberry pi documentation," Camera. [Online]. Available: https://www.raspberrypi.com/documentation /accessories/camera.html. [Accessed: 20-Apr-2022].

[7] "Reference counting," Reference Counting - Python 3.10.4 documentation. [Online]. Available: https://docs.python.org/3/c-api/refcounting.html. [Accessed: 23-Apr-2022].

[8] "Jetson modules," NVIDIA Developer, 22-Mar-2022. [Online]. Available: https://developer.nvidia.com/embedded/jetson-modules. [Accessed: 25-Apr-2022].

Andrei Tumbar is a third year Computer Engineering student in Kate Gleason College of Engineering at Rochester Institute of Technology. He has been working for NASA Jet Propulsion Laboratory for two years in the Mobility and Robotics section. Andrei has been building operations tools for Perseverance, Curiosity, ColdArm, Ingenuity, as well as contributed to the FPrime Flight-Software framework. He is interested in robotics, embedded systems, and has a newfound interest in computer vision.