ECE 492/592 - Introduction to Autonomous Systems

Project A4: Parking Finder

Team Members:
Rohith Iyer
Andrew Knowles
Kunal Saboo
Gabriella Smith

Table of Contents

1 Abstract

This project focuses on designing the payload and image processing software to enable an Unmanned Aerial Vehicle (UAV) to autonomously fly along one side of a parking lot, find and mark all the empty parking spaces, and return their geographical coordinates to a user on the ground. To perform this task, the project is broken down into several subsystems: the payload and its accompanying hardware and software, the mission control software, the image processing software, and the location finding software. This report details the work referenced in the design of the solution as well as the processes followed and software used for the implementations of the location finding and image processing algorithms. Furthermore, it explains the approaches used to accomplish each sub-challenge as well as the hardware used in the construction of the payload and the testing done throughout the project.

2 Introduction

The objective of this project is to design an autonomous UAV that will be able to travel along the perimeter of a parking lot and, with the help of data processing done on the ground, find a minimum of 80% of the empty parking spaces in a parking lot along with their geographical coordinates. After completion of the mission, the drone will then return to its launch point.

This solution is useful as it can be applied to virtually any large parking lot. In crowded areas such as malls, schools, and stadiums, visitors often spend lots of time and gas driving up and down the aisles looking for a parking spot. Empty spots are generally hard to find without extensive searching and this problem is further compounded when available spaces are obstructed by larger vehicles. The easiest way to search for empty parking spots is by viewing the entire lot from above, and our solution does just that. Using a small UAV equipped with an onboard camera, our drone can quickly scan the parking lot and send images of the lot to a ground computer for processing. The ground computer can then return the coordinates of the empty parking spaces.

This problem can be solved by breaking the primary challenge down into four sub-problems: navigation, image acquisition, image processing, and location finding. The mission navigation and image acquisition tasks are completed with the help of an onboard Raspberry Pi functioning as a companion computer and a USB camera to take the necessary pictures. This data, along with the corresponding altitude, latitude, and longitude of the drone at the time the picture is taken, is then sent to a computer on the ground. The ground computer then uses algorithms implemented in python to process the images and return the locations of the empty parking spots.

Our solution is designed to work effectively when all the parking spaces are oriented in the same direction. In addition, although we have programmed the drone to complete its mission at the Spring Hill Park and Ride lot by NC State University's Centennial Campus, we are confident that adjusting our code to work at other locations would take under an hour to complete.

3 Related Work

To get an idea of where to start working on the project and a foundation of what our final results might look like, we did some research for similar projects that might give us some direction. In our research, we found the article "Find where to park in real time using OpenCV and Tensorflow". [1] This article details a preliminary algorithm for detecting car spots that we based our detection algorithm on. In it, the author describes how they detected the lanes and columns of the lot they were observing before continuing on and detailing how they found the empty and open spots. This algorithm gave us the methodology we used for lane and spot detection, as well as a codebase that allowed us to expand on to make our algorithm for detecting a spot to be open or occupied.

In our work for this project, we also ran into some issues with finding the corners of the image once we had detected the gray in the image. To assist us we reached out to Professor Sichitiu and were directed towards SAS for assistance. In contacting them we were able to reach out to Mark Matthews and Russ Taylor who were able to assist us in our algorithms and guide us towards what would end up being a large part of our final working algorithm. Their primary assistance was with the algorithm for correctly defining the mask to make finding corners easier and more efficient.

4 Approach

4.1 Overview

The overall system architecture consists of the drone airframe, the autopilot and corresponding sensors, the payload, and its associated sensors, and the ground control station (GCS). The companion computer communicates with the autopilot via a MAVLink connection over USB. In turn, the autopilot controls the airframe and its sensors. In addition, the companion computer runs the necessary software to execute the drone's mission, including navigation, control of the camera, and storage of necessary data.

From the ground, the GCS computer accesses stored data via a mapped network drive over Wi-Fi. It also communicates with the companion computer over a wireless MAVLink connection. Finally, all image processing and subsequent location finding algorithms are run on the GCS computer.

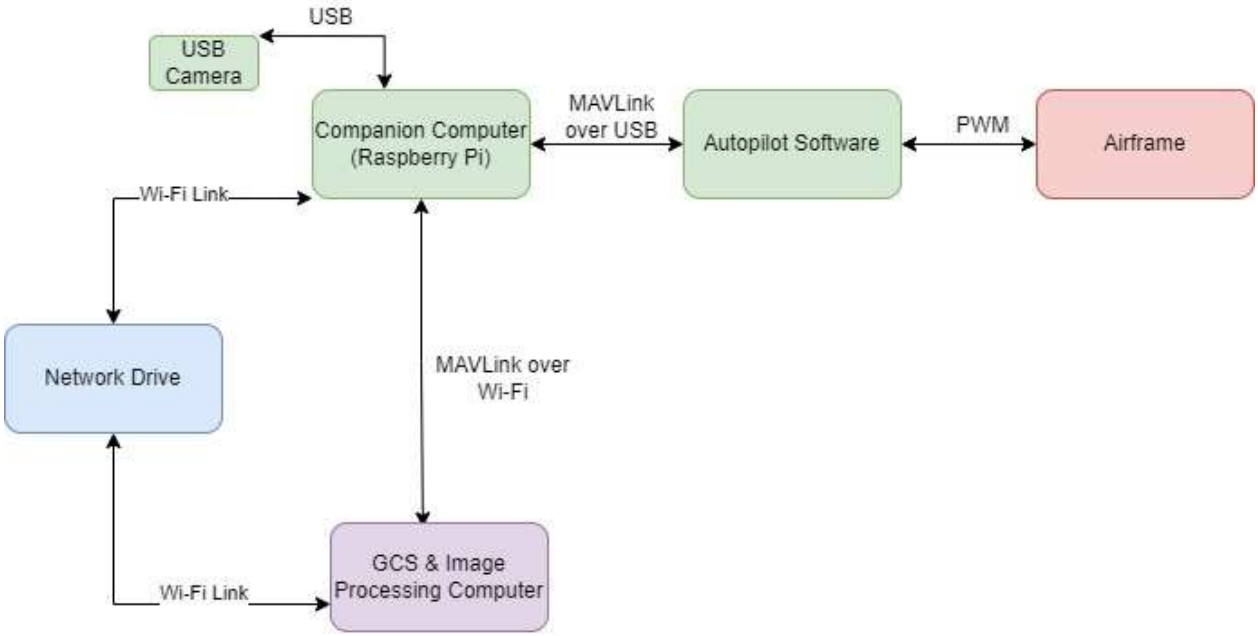

Figure 4.1.1 - System Architecture

4.2 Mission Software

The mission control software was programmed in Python, making use of several different libraries. In particular, we used the DroneKit Python library to create commands for our UAV and communicate with the ArduPilot flight controller using a low-latency link. While many of the DroneKit library commands were used, several custom functions were created for pre-flight checks, traveling to a location, taking a picture, and retrieving coordinates from a JSON file.

Before sending the command to have the drone take off, it is necessary to check if the drone is capable of taking off and carrying out the mission. This includes making sure the drone battery has sufficient voltage, the drone is in guided mode and armed, and communication has been established. If any of these conditions are in the incorrect state, then the drone will not allow take-off and will display an error message. Landing and returning to the home location requires changing the flight mode rather than sending a command. For returning to the home location and landing, the flight mode must be changed to “RTL”. We also implemented another function to check if the UAV has reached the target coordinates. This function essentially measures the error difference between the UAV’s current position and the target position. While the error is greater than a specified value alpha, the function returns false. When the error

becomes less than alpha, then the function returns true, indicating that the UAV reached the target location.

All these functions are called in the main function, where we planned out the UAV mission. After collecting the coordinates from the JSON file and passing the pre-flight checks, the UAV would take off from its starting position. For every set of coordinates pulled in from the JSON file, the UAV would point towards the correct heading, travel to that location, pause briefly, and then take a picture as further discussed in section 4.3.2. After all waypoints have been traversed, the UAV will carry out the RTL function and export the mission data to a file on a shared drive between the companion computer and GCS.

4.3 Payload

4.3.1 Hardware

The payload consists of a Raspberry Pi companion computer for running the mission and image acquisition software, a USB camera, and a camera mount as detailed in the table below.

Hardware	Link
SAM4 Drone	Class Lab
Mobius 4K Action Cam	https://mobiusactioncam.com/products/mm4k/?v=e2ae933451f4
Raspberry Pi 4	https://www.amazon.com/Raspberry-Model-2019-Quad-Bluetooth/dp/B07TC2BK1X/
Camera mount	Custom 3D Print
Miscellaneous hardware	Class Lab

Table 4.3.1

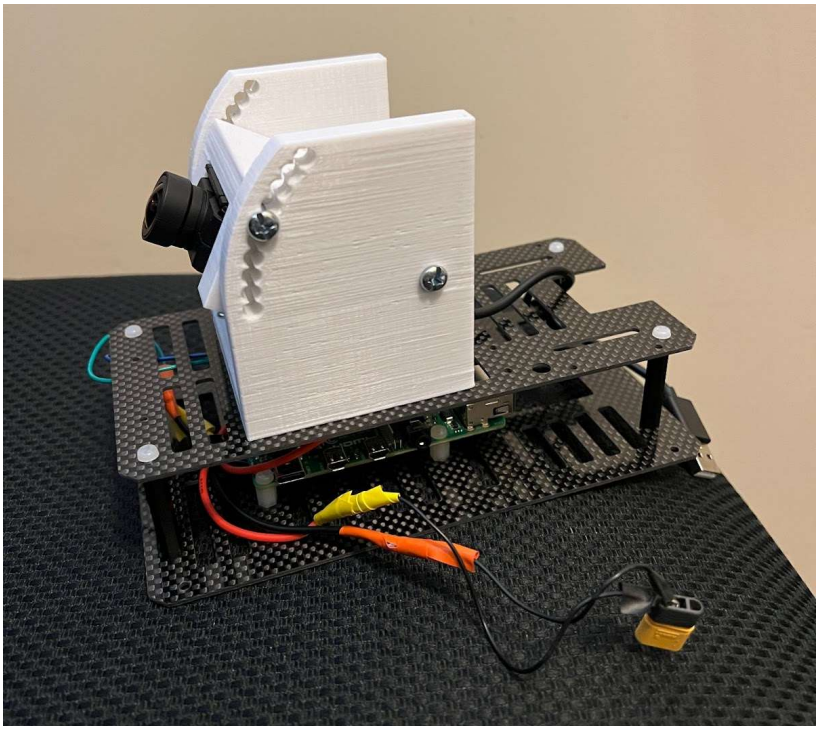

Figure 4.3.1 - Assembled Payload

4.3.2 Software

To take the pictures, the OpenCV library in Python was installed onto the companion computer. One concern we had was ensuring that the pictures taken were of sufficient quality to use in the image processing stage of the project. To address this challenge, an image quality metric called BRISQUE (Blind/Referenceless Image Spatial Quality Evaluator) was used. This metric does not require any reference images and assigns a number from 0-100 to the image based on the distortion/blurring it detects. The lower the number, the higher the quality of the image. To achieve this, we installed the brisque python library. We wrote our code to continue taking pictures until the BRISQUE metric returned a value at or below the desired threshold.

Since the image processing and location finding is not completed on the drone's companion computer, a method of transferring the images and corresponding location data from the drone to the ground was needed. To achieve this, we set up a Samba server on the raspberry pi and wrote the necessary code to store images and other data in this public drive as it is received. Provided the ground computer is connected to the same network as the companion computer, it can easily access any files saved into the mapped network drive.

4.4 Image Processing

Utilizing python and its OpenCV library, we were able to manipulate the image that was taken from the drone into one that could be used to identify the open parking spots in the lot. This was done by applying a series of functions from the OpenCV library to first remove noise, find the gray in the image, and morph the found edges to the edges we wanted them to fit. A more detailed overview is given as follows. First, a bilateral filter was applied to blur the image and remove as much extraneous noise from cars in the lot and surrounding environment as possible. Then, two bitwise masks were applied: the first to find the majority of the gray and the second to catch any holes that the first might have missed due to parked vehicles there. We next changed the image to black and white to get a block that would cover the whole lot before using a series of closes, erosions, dilations, and opens to smooth the edges and remove any remaining noise. Finally, we identified the corners by creating a convex hull around the mask and using the functions from the library to find the most likely fit for the corners of the lot we are testing. From here, all that was needed was to take those corners and skew the original image to them to make it appear as if the image was taken from above. This concluded the image processing functionality, and the image was passed to the spot identification code to identify each of the parking spots and return them to the user.

4.5 Spot Classification and Location Labeling

After finding the bounding boxes using our image processing pipeline, we needed to classify them as either empty or occupied. To classify the boxes, a convolutional neural network (CNN) was trained on a set of labeled spot images using a stochastic gradient descent optimizer. The model uses transfer learning principles and is based on a commonly used image classification model called VGG16. To classify the images, snippets of the flattened image based on the bounding boxes are separated and passed through the input of the classification model. The model then returns a 1 or a 0 which corresponds to occupied or empty spaces, respectively. The classified boxes are then separated based on this categorization.

Once the classification is complete, each empty spot needs to be labeled with the location in latitude and longitude for the user to navigate to. First, the center of the parking spot is found in pixels. Then, the pixel coordinates are turned into fractions of the image height and image width. Next, these fractions are turned into latitude and longitude by being multiplied by latitude-longitude vectors for the height and width of the parking lot. The latitude-longitude vectors are found based on the corner coordinates of the parking lot provided as input to the program. This results in an accurate position of each empty parking space.

5 Experimental Results

5.1 SITL Testing

Before our physical testing, we ran multiple runs of our software in the loop testing. Our environment consisted of a Docker container representing the autopilot and drone, the payload software running on a Raspberry Pi connected over Wi-Fi, and the ground station software which runs on the same computer as the Docker container. Mavproxy is used to connect the autopilot and the Raspberry Pi. A shared drive located on the Raspberry Pi is made available to the computer running the ground station software. After numerous SITL runs debugging code errors, multiple successful runs were completed with sample images from Google Maps to verify the accuracy of the pipeline.

5.2 Physical Testing

After completing our SITL testing, we did some physical testing with the UAV in the Lake Wheeler Field. Since there was no parking lot here, the testing consisted mainly of taking off, flying to a given location, taking a picture, and returning to the launch position. This sequence was executed successfully, and we were able to verify that the payload was fully functional and that there were no other errors in the code we may have missed in our SITL testing.

Next, we made a few testing runs at the Spring Hill Parking lot. At this location our goal was to simulate the final demo: have the UAV take-off, capture a picture at a specified location, and then return to launch. During our testing, we took multiple pictures in a series of trials to determine the best vantage point for taking pictures as well as the optimal camera angle. Based on various factors we observed in these trials, which are further discussed in section 6.2 of this report, we were able to determine the best location for taking pictures as well as the best angle to position the camera on the payload.

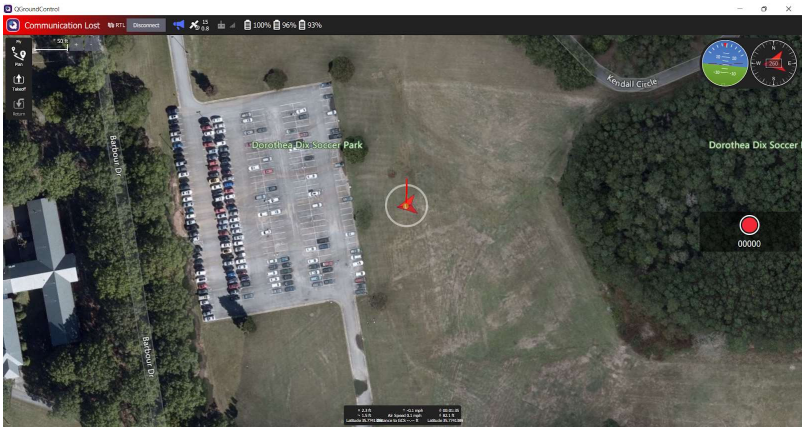

Figure 5.2.1: QGroundControl after mission completion

Figure 5.2.2: Images from Final Testing

Finally, we conducted our project demo at the Spring Hill Parking lot. Using the adjustments we made in our previous trials, we had our UAV take off with the previously determined configurations. As planned, the drone flew to its pre-specified location, took a picture, and then returned to launch. After obtaining the captured image from the shared network drive with our drone's companion computer, the GCS was able to run it through our image processing algorithm and identify the empty parking spots in an overlay of the image. A list of the coordinates of the empty parking spots was also returned to the GCS.

6 Analysis

Throughout this project, we encountered several challenges with both the hardware and software components of our UAV. Although most of these challenges were unanticipated until we did testing at the parking lot, we were able to adapt our solution to overcome them.

Initial tests of our image processing algorithm were performed on a picture of the testing parking lot obtained via Google Maps. However, when we did our first testing run at the actual parking lot we realized that, because of the altitude we were flying at, the picture taken from the drone captured parts of the surrounding blue sky in addition to the parking lot. Since the Google Maps image we were using as a reference did not contain the sky, we needed to make adjustments to our code to account for this additional variable.

We were also concerned about needing to make adjustments to the camera code due to the presence of wind at the testing site leading to a possible blurring of the picture. Anticipating that this might be an issue, we implemented the BRISQUE algorithm detailed in section 4.3.2 of this report, and did not run into any issues with picture quality during our physical testing.

The image processing algorithm was also vulnerable to certain image features, particularly darker colors, and distinguishing corners. This was particularly a problem in instances where a car was parked in a corner spot or, as in the case of our testing area, where a large tree would sometimes cast a shadow or cover a corner itself. The algorithm would sometimes have a hard time distinguishing what should be included and what should be removed. This issue ended up being revolved using two separate functions in our code. The first was the mask identifier which used two masks to identify a majority of the gray areas, and a second to fill larger holes left by cars in it. A series of morphs on the masks were performed later to remove the noise caused by these issues and make it to where the contours could be more easily identified. If issues persist, they would not affect the ability of the code to later on detect corners (examples shown in appendix B).

The next issue that came up related to this is that when identifying the corners was finding the correct corners once the mask had been made. The goodFeaturesToTrack function that is provided and recommended for doing this exact thing would sometimes struggle to identify the actual corners first. This was due to rough edges, the lanes to enter and exit the parking lot, or any other object that it might distinguish as a possible corner throwing off or scaling that we would perform later. To get around this we polled for more than the anticipated amount of corners at a much lower threshold than normal then distinguished which ones were the real corners mathematically by assuming that the corners of the lot would be the closest distinguishable ones to the edges of the actual image. This ended up being our final solution due to us working this assumption into our locations for where we would take the image from by making it a given and favoring the results towards the horizontal lines found as they were much cleaner than the vertical ones. This was due to the perspective of the images taken (examples shown in appendix B).

Another issue we ran into when flying the UAV at the parking lot was our misunderstanding of the “point to heading” command. When we send this command for the UAV to point to a certain heading, it only rotates in the positive direction. If the degree amount specified was negative, the UAV would rotate in the positive direction until it reached the target heading. Since we had not planned for this initially, on our first tests at the Spring Hill location when we were running a mission we several waypoints we ended up taking pictures while the drone was still turning to reach its desired location.We ended up not resolving this particular issue; however, it did not become a major issue as we were only using a single waypoint for our primary mission.

7 Conclusion

In conclusion, the problem our UAV project set out to solve was to find all of the empty spots in a parking lot and return their corresponding geographical coordinates. At our final demo, we were able to successfully survey the Spring Hill Parking lot with a UAV, take a picture of the lot from the field beside the lot, and return an ordered list of the coordinates of all the empty parking spots. Despite encountering various challenges throughout our testing, we were able to resolve them reasonably quickly and progress forward with our solution.

While we are satisfied with what we were able to achieve during the semester, we do have a few things we would add, were we to continue this project into another semester. For instance, our algorithm is currently set up to work well on a single image of the entire parking lot. While this works well on smaller lots, to expand our solution’s functionality to larger areas, an image stitching process would need to be implemented that would allow the combination of pictures from a series of waypoints along one side of the lot. This image stitching would also enable us to take higher quality images instead of working with a more zoomed-out, less clear picture.

Another thing we would have liked to add would be a gimbal to our drone for camera stabilization and active targeting of the parking lot. Although we were able to utilize a python image quality library to avoid taking low-quality images when we needed to adjust the angle of our camera that had to be done manually once the drone was on the ground. A gimbal would have ensured that our camera was always stabilized and would have also decreased the testing time needed to adjust the camera. Finally, our solution was designed to work very effectively on mostly rectangular parking lots that have grassy areas along the edges. While we are reasonably confident that, with some minor adjustments, our code would work in lots that were not surrounded by grass, we would like to expand our algorithm to accurately identify spaces in parking lots of a variety of different shapes and sizes.

While we designed our project to work well at the Spring Hill Parking Lot and conducted all drone testing at that location, we are confident that our code will work on parking lots of similar shape and size with only very slight modifications. Furthermore, even though our algorithm was unable to detect the parking spots with 100% accuracy, we were still able to accurately identify the vast majority of the spaces and surpass our target goal of 80% accuracy. Overall, we were very pleased with the results of our project and with how much we were able to accomplish in a very short period.

8 Appendices

Appendix A: References

1. https://towardsdatascience.com/find-where-to-park-in-real-time-using-opencv-and-tensorflow-4307a4c3da03

Appendix B: Photo Examples

All examples are from actual code working on the example image shown below to get a final result.

Figure 8.1: Camera Image

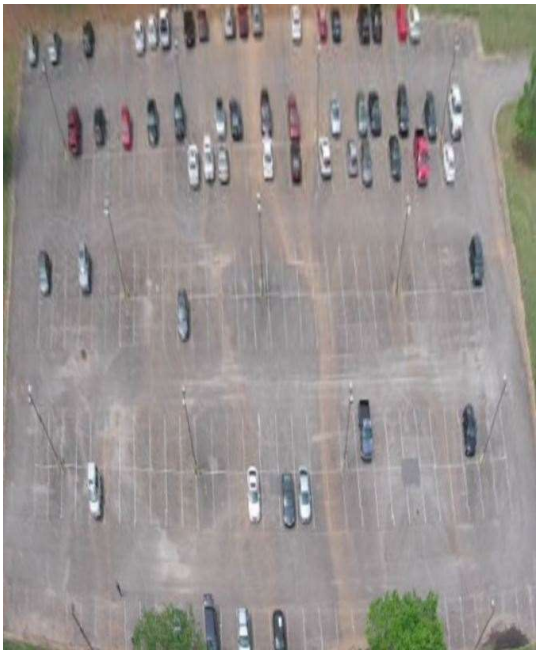

Figure 8.2: Adjusted Image

Mask Examples

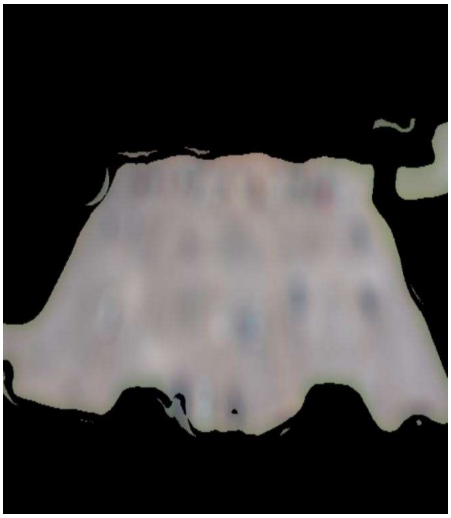

Figure 8.3: Base mask with spot detection

Figure 8.4: No spot detector mask

Morphing Examples

Examples below show the rounds of operations that were performed to show why they were needed to help clear up the mask for future operations.

Figure 8.5: Base Mask with No Manipulation

Figure 8.6: Mask with a round of closing to remove holes

Figure 8.7: Mask after rounds of erosion to remove noise

Figure 8.8: Mask after rounds of opening to further remove noise

Figure 8.9: Mask after rounds of dilation to add back any of the main blocks that were eroded when removing noise

Figure 8.10: Mask after a final round of dilation to smooth edges and ensure corners are accurate to lot image

Figure 8.11: Contours generated that show outline of mask on original image

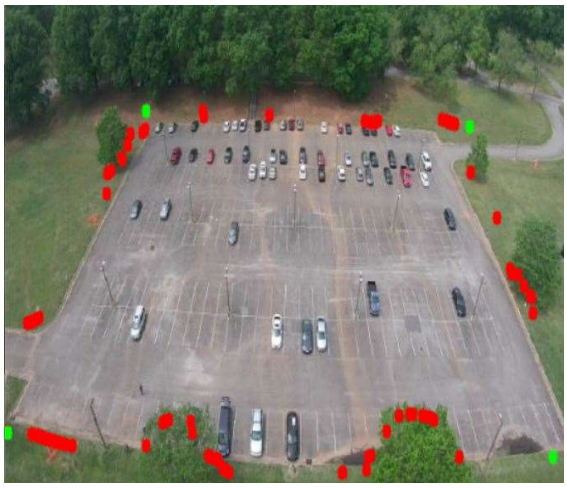

Figure 8.12: List of possible corners found shown in red with best corners highlighted in green

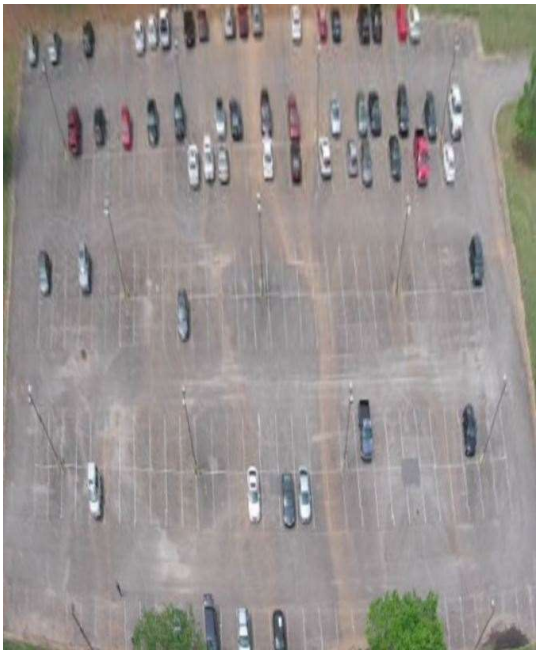

Figure 8.13: Final product after all rounds of cleaning image