Robot Vehicle System Research

This document reflects an effort to research alternative methods by which we will be able to implement our project requirements. Specifically, this document will research the possibility of using a Raspberry pi with a pi camera, and the OpenCV library to control our robotic vehicle system.

We would most likely be using the newest Raspberry Pi 3 Model B+, which would come with built-in wireless LAN, and BLE (<https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/)>. The board runs for about $30. There would not be a SD Card or a power unit, so for each Pi they would have to be purchased separately (SD card price depends on size & $10.99 respectively). There would also need to be a camera system that can connect to the raspberry pi (<https://www.raspberrypi.org/products/camera-module-v2/)>. It can either be the pi camera which is $24.99 or a cheaper webcam. It would ideally be a low power camera with enough resolution so process the images (i.e. it could be a usb webcam).

These components would be used to capture the image, and then once we have the image we can process it on the raspberry pi directly or on an application that controls our whole system. The library that I have familiarity with is OpenCV (<https://opencv.org/)>. The library is written in C++ and provides many functions and tools used for computer vision. There are also Python and Java wrappers that allow you to write code in those languages. There is also a lot more support and documentation for OpenCV as it has become more popular since computer vision is an increasingly popular topic. The current version of OpenCV is 3.4.3 and was released 8/29/18 so it is very current. The library would provide us with almost all the resources we need to perform collision detection and other features, such as calculating the position relative to the lead vehicle.

There are GPIO pins on the Raspberry Pi that we could use to control the robot vehicle, but ideally we will already have the vehicle built and will just need to figure out a way to interface with it. We would need to be able to send information to the unit as well as receive it.

Another alternative option would be to use the computer vision and image processing capabilities that can be included with the MATLAB program (<https://www.mathworks.com/products/matlab.html)>. This provides the users with a lot of built-in features that you would otherwise have to explicitly program in OpenCV. The student cost of the program would be $100 and then there is the $10 add-on for the computer vision module. If we are able to find the correct robotic vehicle, this may be the best system to use because it allows you to perform a lot of high-level actions and not waste time coding menial tasks.

The system using MATLAB would be contingent on the robotic vehicle, however for processing the data it is a good tool.

Research After Meeting with TARDEC

The purpose of this research is to look at the different object tracking algorithms available and compare them for use in our vehicle follower system. After speaking with TARDEC, we decided that we wanted to using open source RaspberryPi car kit that has it software available and github. The language that is used in this case is python and there are python binding available for a library called opencv that has some image processing algorithms.

The other purpose of the following research is to define some of the image processing basics, so that I may learn as well as anyone who is reading this document and is new to the material.

**Image Basics**

An image will have a width and height in pixels and can contain 3 channels (RGB components) so the array of an image would look something like (width, height, channels). Most pixels can be represented in two ways. One way is grayscale where each pixel in the image has a value between 0 to 255. A value of zero means the pixel is black and value of 255 is white. In between these values are different shades of gray, with the shades getting darker moving to zero and lighter moving to 255.

The other way of representing them is like above with a value of 0 to 255 for each of the channels RGB. This would look something like (150,50,210) where each value represents how much of each color channel is in the pixel. This represents a color pixel. There are other color channels we can use such as HSV (hue, saturation, value).

When you load an image off the disk into OpenCV, you are loading it into a NumPy array.

Common RGB Tuples

* Black (0,0,0)
* White (255,255,255)
* Red (255,0,0)
* Green (0,255,0)
* Blue (0,0,255)

OpenCV stores these in (B,G,R) order.

Normal matrix representation: # of rows x # of columns

Numpy matrix representation: # of columns x # of rows

For an image’s coordinate system imagine we have an 8-pixel x 8-pixel image. The top left of the image would be referred to as (0,0) and the bottom right of the image would be referred to as (7,7). The pixel locations are written as (column, row).

To display an image

cv.imshow(“Window title”, image)

**Drawing Functions in OpenCV**

These are important because they will be used to help debug our object tracking algorithms. The three drawing methods we will look at are line, rectangle, and circle.

We can use an image as a canvas or initialize our own numpy array.

canvas = np.zeros((300,300,3), dtype=”uint8”)

Once you have a canvas you can draw on it using these commands

cv.line(canvas, (0,0), (300,300), (0,255,0), 3)

This will draw a green line from the top left to the bottom right of the canvas with a thickness of whatever value is assigned ‘3’.

cv.rectangle(canvas, (10,10), (60,60), (0,0,255), -1)

This will draw a rectangle that has a top left corner starting on (10,10) and a bottom right corner on (60,60). If we pass in a negative value to the thickness argument, we will fill the rectangle with the color red.

cv.circle(canvas, (centerX, centerY), radius, color, thickness)

This will draw a circle that is centered on a point and of a designated radius. We also can pass in a color and thickness argument.

Using these functions, we can draw shapes on top of our image or canvas which can be used to help debug what our camera is tracking.

**Image Transformations**

These are methods and techniques that an important in nearly all areas of computer vision.

*Translation*

Translation is the shifting of an image along the x and y axis. Using translation, we can shift the image up, down, left, right, or any combination. To do this you define a translation matrix which will tell us how many pixels and in what direction our image will be shift. We then can use the OpenCV method called warpAffine(original, translation, size of output image).

For the translation matrix we define it as such:

M = np.float32([[1, 0, 25], [0, 1, 50]])

This translation matrix, when applied, shifts out image right 25 pixels and down 50 pixels.

*Rotation*

Another method that is useful in computer vision is rotating images. This can be done by specifying around which point the image will be rotate. OpenCV allows you to choose an arbitrary point. We can get the image’s width and height and then get its center from that. Similar to the translation methods we will need to define a rotation matrix. We can do this with a direct call to the OpenCV method

M = cv.getRotationMatrix2D(center, degrees, scale)

The degrees rotated is rotation counter-clockwise, same direction as pi increasing on unit circle.

We then apply this matrix using the warp affine method as we did above.

The result will look something like this

rotated = cv.warpAffine(image, M, (w, h))

We are able to get the center of the image very easily.

(h, w) = image.shape[:2]

center = (w / 2, h / 2)

*Resizing*

In order to resize an image, we can use an OpenCV convenience method, but first we must find the ratio of the new dimensions to the old in order to keep the aspect ratio of the image. Let’s say that we want our new image to have a width of 150 pixels. We divide new width by old width

r = 150.0 / image.shape[1]

From here we can calculate the new dimensions of the image:

dim = (150, int(image.shape[0] \* r))

resized = cv.resize(image, dim, interpolation = cv.INTER\_AREA)

The last argument is the algorithm actually used for resizing and this one has been known to give the best results for it. We could also have used a known height to obtain the new width ratio in the same way to preserve aspect ratio of the image.

*Flipping*

We are also able to flip an image around either the x-axis, y-axis, or both. This is easily done using a built in method

flipped = cv.flip(image, 1)

The second argument is the flip code corresponding to the direction.

* 1 – flip horizontally (around y-axis)
* 0 – flip vertically (around x-axis)
* -1 – flip horizontally and vertically

*Cropping*

Cropping an image removes the outer parts of the image that we are not interested in. We can accomplish this by using NumPy array slicing.

cropped = image[30:120, 240:335]

This will extract a rectangular region with the top-left located at (240,30) and ending at (335, 120). Supply y-axis values before x-axis values.

After we display images in OpenCV we want our script to wait so we use the cv.waitKey(0) function which takes an argument that represents the ASCII value of the key that will need to be pressed to continue. If the argument is 0 then that represents an infinite time and any button will need to be pressed.

The above information was mostly taken from a book called Practical Python and OpenCV by Adrian Rosebrock. While I own the pdf from which this research was done (I can provide a copy if requested), he does have a website which has many tutorials and is a great place to learn about computer vision and python. The website can be found here: <https://www.pyimagesearch.com/>

**Comparison of Object Tracking Methods**

The article discusses a comparison of tracking algorithms available in OpenCV using C++. For their testing purposes they used three feature detectors, three pure trackers and one complex tracking framework.

Feature detectors are not necessarily trackers, but they work by trying to detect a specific object in each frame. In order to do this there are usually two images required. One image is the video or the frame obtained from the live stream and the other is the image of the object that we wish to track. The image of the object we want to track can be obtained from the frame of the live stream. The feature detectors would then try to detect features in the object to be track and then try to find the best mapping of those features against the current frame.

The most important thing about feature detectors is that the object being tracked is large enough and it has detectable features such as edges and texture. The reason that feature detectors cannot really be called trackers is that there is inconsistency between the frames. A tracker will follow trajectory, whereas a detector will find the best match for the two images with leads to instability if something is mistaken to be the tracked object. This is especially apparent when there are objects in the picture that are part of a repeating pattern such as a fence or windows on a house.

In OpenCV there are currently three useful feature detectors: SURF, SIFT, and ORB. The first two detectors are patented and use floating-point numbers, and the last one is less precise because it uses integers, but it also has a friendlier license.

Trackers are designed to only track the object by following its trajectory, and predicting its future locations as well as correcting any errors in the process. The biggest problem with pure trackers such as these are when the camera moves to fast or the object suddenly changes speed or direction. In OpenCv there are currently three useful pure trackers: MIL, Boosting, and MedianFlow.

There are also tracking frameworks that are very complex that attempt to provide a comprehensive solution to the tracking task. The biggest disadvantage of a system such as this is that they use more memory and consume more power while processing. The article mentions one of these tracking frameworks to be known as TLD (Tracking-Learning-Detecting). The acronym stands for the three parts of the system that work independently.

* Tracker –tracks a blob of pixels from frame to frame
* Detector –finds a similar observed object so that the tracker can be corrected if needed
* Learner – estimates the errors in the detectors and updates the detector in order to help avoid the same error in the future.

The basic parts of the TLD framework is that they can run simultaneously, each on a separate processor, or separately as tasks. You can also optimize each part.

The article evaluates each feature detector, tracker, and tracking framework by looking at the success, precision, time demands, and performance. They compared the bounding rectangle of the method to a dataset that already has annotations.

What they found was that in terms of success, the TLD framework was not the best, but the pure trackers MIL and BOOST were. They had some reasons one mainly being that the TLD is so heavily correcting itself, that it doesn’t follow the ground-truth bounding rectangle (obtained from the annotations). Therefore, it means that the TLD doesn’t lose the object, but instead it fails to center the objet within its own bounding rectangle.

For performance, they measured the scale ratio of each algorithm. If the object was located precisely then the bounding box dimensions would be more similar to the ground truth than not. What they found was that ORB was the feature detectors were overall more precise, ORB being the most.

As for time demands, they measured the time each algorithm needed to process a single frame. Where SIFT and SURF are very slow because they work with floating-point numbers, ORB is much faster. All the algorithms are faster than SIFT and SURF, except for TLD which is relative to SURF in time due to its robustness and lack of optimization in its OpenCV implementation.

Their conclusion was that because there are limitations in algorithm implementation, there needs to be more detailed analysis created.

We must decide on what we want in our algorithm before we can choose the right implementation. I would suggest looking at tracking algorithms such as MIL, Boost, or MedianFlow.

The article about the comparison of object tracking methods found here: <https://www.matec-conferences.org/articles/matecconf/pdf/2016/39/matecconf_cscc2016_04031.pdf>

**Camshift Algorithm**

Another algorithm that is known as a tracking algorithm is called the camshift algorithm. The purpose of this section is to provide some background information on the algorithm and at the end decide whether or not it will work.

What a camshift algorithm does is record the results from a meanshift performed on every frame.

There are three parts to a camshift algorithm

1. Back Projection
2. Meanshift
3. Track

A back projection is when you take a histogram if an image and it will show the probabilities of colors that might appear in each pixel. First we transform the image into the HSV color space and extract out the hue channel as a single grayscale image. We then use the cv-supplied function “calcBackProject()” to get the back projection of the image. What this function does is calculate the weight of each color in the whole image using a histogram and then its changes the value of each pixel to the weight of its color in the whole image. By doing this method to each pixel we will obtain the back projection image.

In the MeanShift algorithm, it uses an underlying probability density function to find modes in a set of data samples. What it essentially does is find the center of mass of a set of points by making a sphere of radius r that is centered on any of the points. We treat the distance to each point within the circle as a vector and the sum of these vectors is what we call mean shift. By this mean shift we can get the current mass center. The order of steps being

1. Initialize sphere with center and radius
2. calculate current mass center
3. move the spheres center to mass center

We repeat these steps until the current mass center is the same point with the center of the sphere.

In order to track the object, we take each frame captured by our camera and use the meanshift algorithm on every single one, and the initial window of each frame is just the output of the previous one.

<http://eric-yuan.me/continuously-adaptive-shift/>

After further discussion with the professor, we have come up with an implementation that we believe will work for our follower vehicle’s code. This research serves the purpose of outlining the methods and mathematics we will use to implement the follower code.

We are already able to detect QR codes that are attached to the back of the car. The libraries used are called Aruco tags. Using these libraries allows for us to use prebuilt methods that detect the QR codes very effieciently, but also return the corners of the bounding box on the detection.

Once we decide how far away from the car is standard distance (i.e. 8inches, 1ft, 1.5ft, or 2ft) we then take a picture and consider it to be our baseline. This baseline image when run with the Aruco tag detection will give us the corners of the bounding box that we want our vehicle to ideally have in its camera view when in follow mode.

The way we can do this is by creating a matrix with the coordinates of the corners for the standard bounding box and a matrix for the coordinates of the corners for the bounding box of the AR tag that we currently see in the frame of the camera. Our goal would be to create a transformation matrix that calculates the differences between these matrices and this will tell us whether the image we are currently detecting is to the left or the right of the standard image that we are striving for.

Besides detecting left or right of the image, we also want to detect how far or how close we are away from the lead vehicle. This calculation will allow us to maintain a certain distance as well as avoid any collisions with the lead vehicle. This would require us to compare the size of the bounding box of the frame that the camera currently sees and the standard size of the bounding box that we want to vehicle to strive for

To calculate this matrix, we will use some basic linear algebra. We would need to create matrices with four (x,y,z) points. These points would represent the corners of the bounding box of the Aruco tag that we see and we would also have a matrix for the bounding box of the standard matrix.

We could do this by following a similar example with triangles that is found on this stack exchange page:

<https://math.stackexchange.com/questions/557507/how-to-find-a-transformation-matrix-given-coordinates-of-two-triangles-in-r2>

The main difference would be that for the calculation of the difference in z we could instead calculate the differences in the size of the box. If the box size is too small in our current frame we would want the car to move forward closer to the lead vehicle which it would cause the box to get bigger and once it reaches the same size as our standard bounding box we would stop the car.