


UNIVERSITY of WASHINGTON

*ADAS Hardware Tool Check*
*3/24/16*

# Table of Contents



# List of Figures

# A - Activity 1: Vehicle Identification

## A.1 - Program Description

The vehicle identification code, `car_detect.m` identifies nearby vehicles and tracks the vehicle directly in front. The code can be broken down into five main parts as identified in Figure 1 below. The IO stage simply processes frames in order and reads and writes to the file system. The stereo stage takes the stereo frames and generates a point cloud such that a depth (in meters) is assigned to each pixel. The stereo code is very similar to the provided Matlab example, and the input video is also from the Mathworks examples. Though it would have been ideal to use other video sources, there were initial issues working with other stereo videos. Despite the low FPS and jumps in the Mathworks video, it still provides a good demonstration of our code. Note that the `stereoParams.mat` file is used to calibrate the stereo cameras and the `thresholdPC.m` applies threshold bounds to the point cloud. The tweaks section is comprised of image processing techniques aimed to increase the likelihood of a correct vehicle detection. In the detect blocks, a trained cascade object detector is used to identify possible vehicles by returning the coordinates for one or many bounding boxes. The detection algorithm relies on a pre-trained detector. Although custom detectors were tested, the final submission uses the Mathworks provided `CarDetector.xml`. The detection stage also attempts to score the bounding boxes so that if multiple vehicles are detected in a single frame, the one that is most likely to be the vehicle of interest is colored green (high score) and all others are red. Finally, as a fallback, if detection fails, the Prediction code attempts to predict the location and distance of the vehicle in front using a weighted sum and simple parameter thresholds. The prediction code is represented with a cyan box annotation and is only executed if the detector fails (30% of the frames).

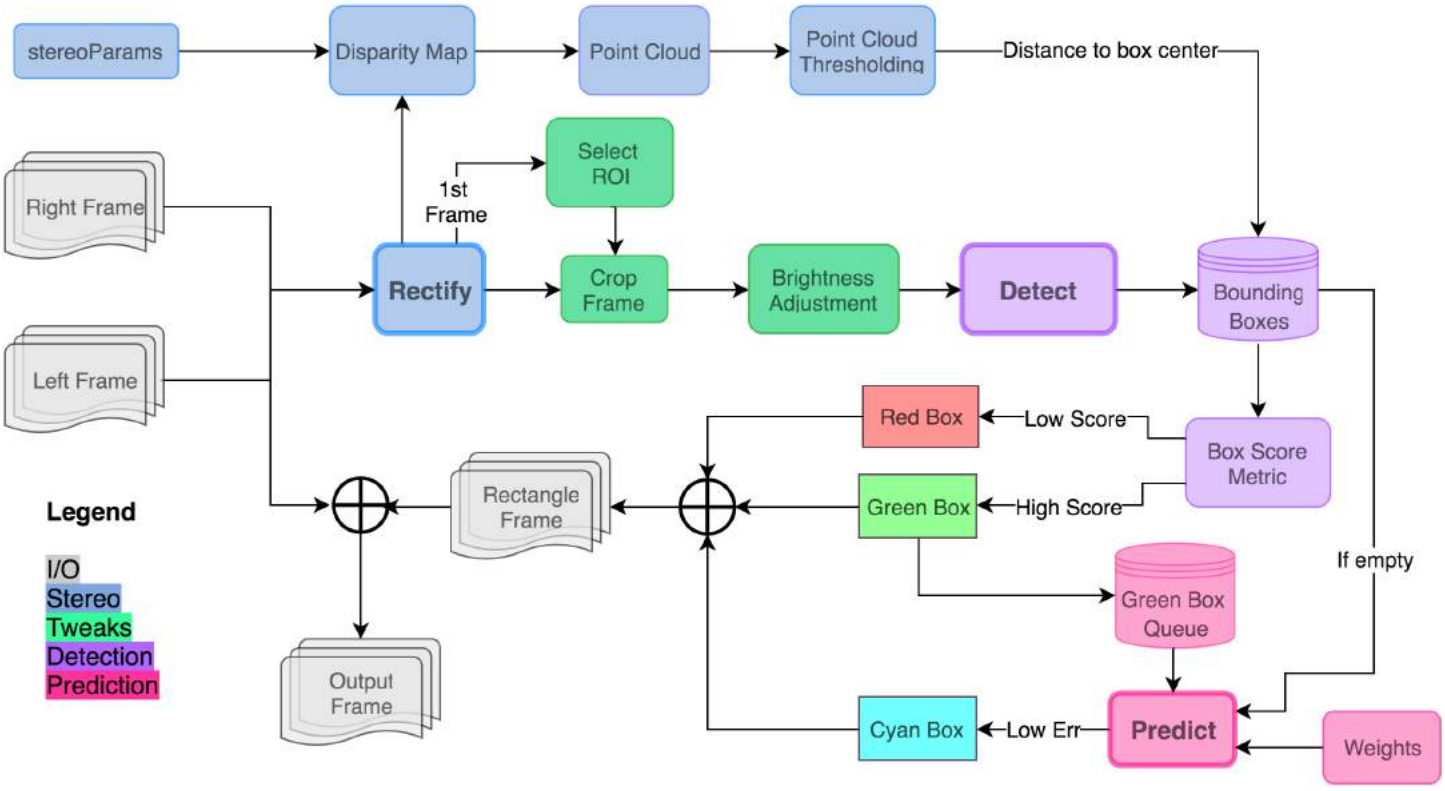


FIGURE 1 BLOCK DIAGRAM OF ACTIVITY 1 CODE

## A.2 - Image Processing Tweaks

In order to improve detection of vehicles located directly ahead of the camera, a region of interest (ROI) is implemented which selects a portion of each frame to analyze, instead of the entire frame. This both increases processing speed, and improves detection accuracy. The user can toggle an option to specify an ROI or use a default ROI. If the user toggles the option to specify an ROI, then the first frame of the video appears and a region is defined using a resizable rectangle. Double-clicking confirms the selection and further processing will use that ROI specified by the user. An additional enhancement to improve detection is increasing the brightness of the ROI. The input video has many shadows and dark cars that are not identified by the detector. After converting to the YCbCr color space, the brightness is increased and converted back to the RGB color space for further processing. Adjusting the brightness allowed for a 15% increase in the vehicle detection stage. In production code, the ROI and brightness tweaks could be dynamic based on the environment and mounted cameras.

## A.3 - Detection

The detection stage is the heart of the processing. It relies on the cascade object detector and its ability to recognize objects after being trained with positive and negative example of the object of interest. Mathworks provided a detector that seems to work well, but is somewhat of a black box, since it is unknown how many images were used to train the detector. Additional custom detectors were made as an attempt to outperform the Mathworks one, but no significant improvements were observed. When the detector is executed on a frame, it returns the coordinates of boxes that bound each vehicle that has been identified. Each box is then analyzed and given a score that represents the likelihood of being the vehicle of interest. This score is based on the height to width ratio, pixel location and overlap with past high scoring boxes. Since it is assumed vehicles don't move that many pixels between frames, this method is effective at detecting and tracking vehicles, however, if the detector doesn't return any boxes (which is common), none of this code will execute and the frame will fail to annotate the frame with the vehicle location. For this reason, the prediction code was added with the hopes of drawing a box on every frame.

## A.4 - Prediction

In order to improve upon the cascade object detector, predictive techniques were implemented to account for the case where the detector fails to detect a car in the current frame. If there has been sufficient data collected on previously tracked vehicles and no vehicle was detected in the current frame, the predictive algorithm is executed. The prediction algorithm uses a weighted sum of user specified length and previously collected data in the form of a queue to compute a predicted bounding box. The weights are set up such that newer frames have a higher influence on predictions. Since the weight vector is user specified, the weights can be optimized.

$$y = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$

Equation 1: Weighted sum where *y* is predicted box, *w* is non-normalized weights, and *x* is previous frame data

After predicting the bounding box of the vehicle, an overlap error is calculated to decide whether the predicted box is sufficiently accurate. The region of overlap between the predicted and the most recent detected bounding box is computed to give an overlap error. This value is then normalized to give an overlap error ranging from 0 to 1; where 0 implies a good prediction, and 1 implies a bad prediction. If the overlap error is within a specified threshold and the

cascade object detector fails to detect a vehicle for the current frame, then the predicted bounding box is drawn on the current frame in the color blue. In addition to the predicted blue bounding box being displayed, an overall prediction error is also reported in white as a percentage. The prediction error is calculated as a sum of the overlap error and the distance error, where distance error is the difference in distance between the predicted and most recent detected bounding box. The average overlap prediction error is 0.17, meaning an 83% overlap exists between the predicted box and previous detections. The distance error comes out to be 0.0271 m which equates to a less than 3 cm deviation in the following distance prediction. Though the complexity of this algorithm is simple compared to something like a Kalman filter, or neural network, it demonstrates how a fallback to detection can be used to increase the confidence and better achieve the goal of tracking the vehicle in front.

## A.5 - Output Video Results

The code is executed by running the `car_detect.m` code. The video player will open and play the frames (slower than real-time). On each frame, annotations are added depending on if the code detects or predicts a vehicle. For each successful detection/prediction, a box is drawn with a label of the syntax: 'D: X m, C: AxB px' where 'D' refers to the distance to the vehicle and 'C' refers to the center point of the bounding box relative to the center, where x center increases from right to left and y increases from top to bottom. Although the rub If the text 'Tracking…' is displayed in the lower center, the detection code has identified a high scoring vehicle which is identified with a green box. Red boxes are low scoring boxes that are generally a vehicle, but not the one of interest. If detection fails and prediction succeeds, 'Predicting…' will be displayed in the lower center and a cyan bounding box is present.

In analyzing the results, 116 green boxes and 50 predicted boxes are displayed out of 181 frames; a total of 166 out of 181 well exceeds the 50% detection requirement. A sample frame is shown below in Figure 2.


FIGURE 2 VEHICLE SUCCESSFULLY DETECTED (GREEN) AND ANOTHER (RED) WHICH IS NOT OF INTEREST

The following distance to the green/cyan boxes is plotted below in Figure 3. Upon visual inspection and comparison to the output video, this seems correct. The sharp jumps and discontinuities are a result of the choppy frame rate of the input video, and false positive detections.

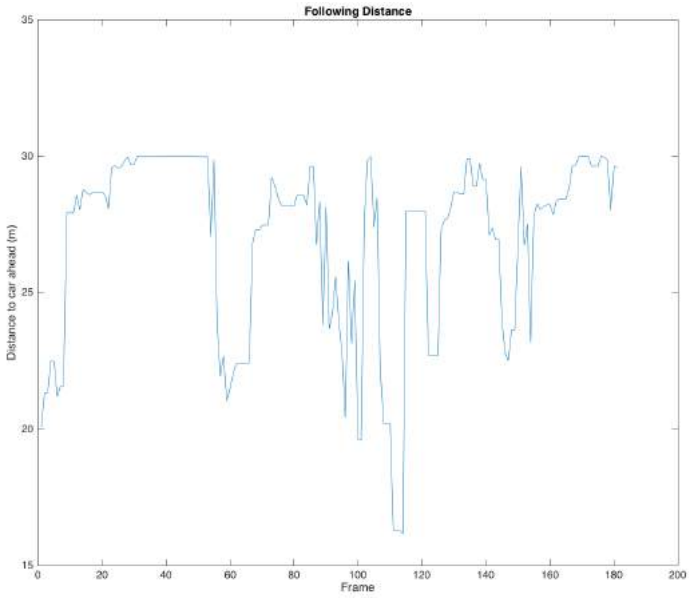


FIGURE 3 FOLLOWING DISTANCE PER FRAME

Validation code compiled from C was also used to help validate the real-time predication Matlab code. The validation code is much more accurate at identifying and tracking vehicles and can be used as a benchmark model. As our realtime code improves, it can continuously be compared and validated against this benchmark. Unfortunately, this code is anything but real-time as it takes about 5 seconds per frame to find all vehicles (no ROI or other processing is used to speed up). This method uses Kalman tracking and a much more advanced vehicle detection model to both identify and track vehicles. Each vehicle is given a uniquely colored rectangle box. Also, note that this code does not use stereo information. It is purely for detecting and tracking. The code can be given upon request as it is not a direct requirement for this activity. A screenshot is shown below in Figure 4.


FIGURE 4 DETECTION USING THE C BASED CODE

# B - Activity 2: Sign Identification

## B.1 - Program Description

The goal of this activity is to identify stop signs in a given frame. The main code file is `stopsign_detect.m` and the helper functions, `train_detector.m`, `annotate_box.m`, and `extract_red_sign.m` are also referenced. Two methods are presented to achieve this detection. First, the cascade object detector which utilizes a detector, trained by our team, to identify stop signs in various lighting and angle. If stop signs are identified, the bounding box(s) is returned and annotated as a cyan box on the frame. If no sign is detected, the experimental color based detection tries to find a stop sign using color thresholding and blob detection. Similarly, any identified stop signs are represented by a bounding box and annotated in green. Just as in Activity 1, an ROI is used to improve detection and performance. The block diagram process is shown below in Figure 5.

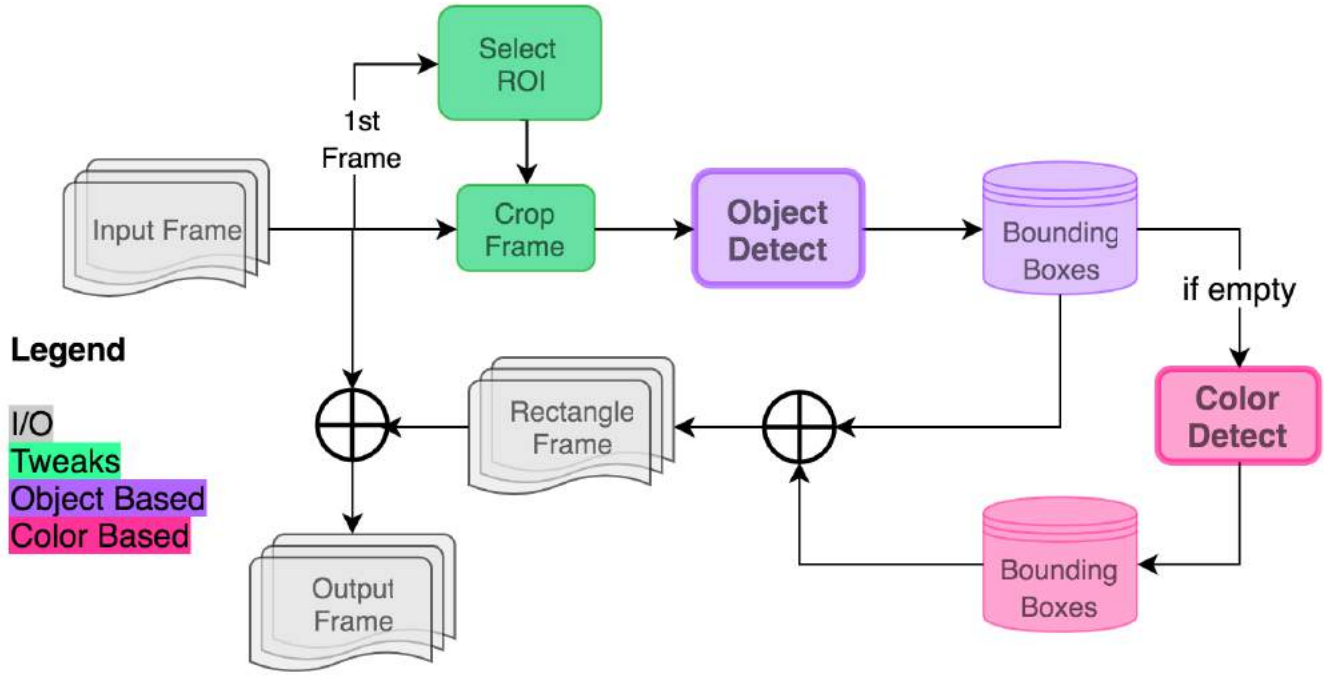


FIGURE 5 BLOCK DIAGRAM OF ACTIVITY 2 CODE

## B.2 - Object Detection

The code for object detection is very similar to Activity 1. The main difference is that it relies on a detector generated by the UW EcoCAR team. This was the biggest challenge of the activity since the goal was to detect the sign in > 90% of the frames. To train the detector, a large database of positive and negative samples needed to be obtained. To make this process easier, a simple Java program was developed to display the first 100 Google image search results for a certain query like 'stop sign photo'. The GUI, shown in **Error! Reference source not found.**, allows one to easily choose whether or not to put that result into the database as either a positive or negative example. After collecting roughly 300 positive stop sign images and about 1400 negative images of roads that did not contain stop signs, it was believed there was enough training data to create an accurate detector. After experimentation and tuning in the `train_detector.m`, it was determined that using 10 stages with a FAR of 0.4 yielded the optimum combination.

Once a sign is identified, the `annotate_box.m` function is called to draw a cyan box indicating the object detector found the sign. Despite our efforts to create an accurate detector, only 20% detection was achieved in the competition

provided video. This could be a failure in our training method, limitation in the Mathworks detector or a problem with our code. Rather than continuing the tedious work of building a training database, a color based detection algorithm was added as an experimental fallback.

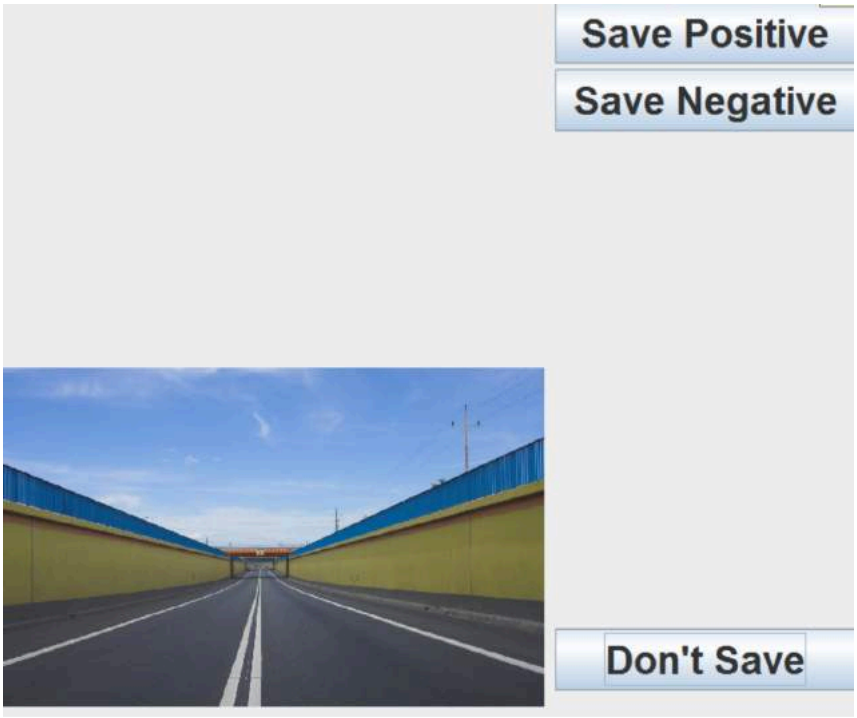


FIGURE 6 JAVA IMAGE SCRAPER TO POPULATE TRAINING DATABASE

## B.3 - Color Based Detection

To achieve > 90% detection, the stop sign needed to be identified even when the sign is represented as just a few pixels. To do so, we took advantage of the stop sign's unique red color and applied color threshholding in the HSV color space to create a binary mask such that red pixels are 1 and everything else is 0. This essentially creates white blobs in regions with red color. This type of thresholding often yields small specs of white pixels scattered across the mask. Since we are interested in a group of pixels, the function `bwareaopen(BW , 70)` is applied to the binary mask, effectively removing white clusters smaller than 70 px, and leaving the larger blobs. Next, the morphological operation `imclose(BW, strel('disk', 15))` is applied to solidify the blobs by filling any holes smaller than 15 pixels with white. All of this processing takes place in the `extract_red_sign.m` function. Following this, `regionprops` command is used, which will create a set of bounding boxes for each blob. Then, each box in the set is analyzed, and only boxes exhibiting a ratio close to a square are kept since a stopsign's height and width are equal. Finally, the `annotate_box.m` function is called to draw a green box indicating that the color detector found the sign.

## B.4 - Competition Provided Video Results

Running the `stopsign_detect.m` code is similar to Activity 1. As stated, there are two types of annotations, indicated by the text in the bottom center of the frame. Each box is labeled 'Dim AxB px Center: CxD px' as defined by the rules, where 'Dim' is the box dimension and 'Center' is the coordinates of the box center, where x center increases from right to left and y increases from top to bottom. With the color detection enabled, the code identifies 229 boxes in the 368 frames which is fair given that the stop sign is the only present until about frame 240. This 95% (229/240) detection rate is a huge improvement to our 20% detection only using the cascade object detector. An example of the cascade detector finding the sign is shown Figure 7 below.


FIGURE 7 COMPETITION PROVIDED VIDEO OUTPUT FRAME

Unfortunately, the experimental color detection only works when there is not a lot of red pixels in the frames. If for example, a red car drove by, it would likely detect the car as a stop sign given the width to height ratio fits into the specified bounds. For this reason, it was disabled for the team video submission. With further tuning, this color based method could prove to be a formidable fallback, but as of now it should be considered experimental

## B.5 - Team Provided Video Results

The team supplied video came from YouTube (link) and was downloaded with standard quality. Aside from disabling the color based detector and tweaking the ROI, the code is equivalent to the competition provided submission. The ROI could eventually be self-tuned and dynamic; however, it is fixed for now. This video has red pixels other than stop signs, so it is easy to see why the color based detection is not as reliable. Common false positives include taillights, red cars, buildings and other red signs. An example frame is shown below in Figure 8.

Improvements to color based detection mainly fall under the tuning category. A similar prediction algorithm as shown in Activity 1 could be used to better identify stop signs based on previous frames. This historical data could also be used to tune the height to width ratio and help reduce false positives. Additionally, signs are static and many of the false positives are attempts to track moving vehicles. Filtering could be added to prevent detection of moving objects. Finally, optical character recognition (OCR) could be used to attempt to identify the characters 'S T O P' in a detection. If none of the letters are seen, the detection could be ignored. This, however, would greatly impact the real-time performance. A validation model, like the one demonstrated in Activity 1, could be used to provide a benchmark for detection. Such a model could meticulously scan each frame for stop signs and output box coordinates. Then when the real-time model is tested, the outputted boxes could be spatially compared to the benchmark results and converted to an error parameter that can be used for iteratively tuning the model.


FIGURE 8 SHOWS FALSE POSITIVES OF COLOR DETECTOR

The cascade object detector did manage to find all of the stop signs aside from one that was almost parallel with the road. An example frame is shown in Figure 9. The main issue is that it usually doesn't detect the sign until the car is within 10 meters away. To improve this, the detector could be trained with smaller images of stops signs. Transformations and rotations could also be applied to the training set to help identify signs at different angles. A more advanced classifier model could also be used to dramatically improve performance. A recurrent neural network, for example, could autonomously train and tune a complex network for classifying stop signs as well as other signs. In summary, more training data and a more complex detection scheme could greatly improve results.


FIGURE 9 STOP SIGN DETECTION USING ONLY THE CASCADE DETECTOR

# C - Activity 3: S32V Interface Demonstration

## C.1 - Workspace

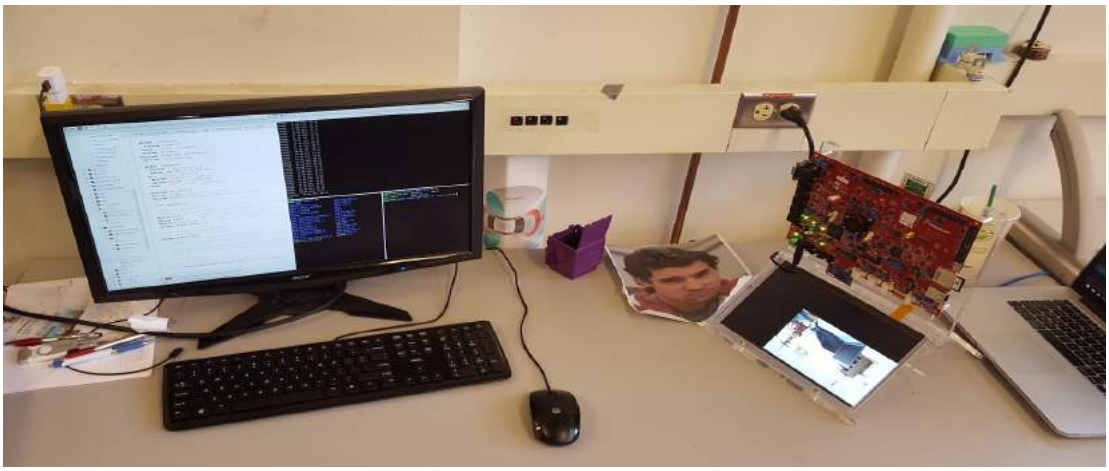

FIGURE 10 WORKSPACE INCLUDES A LINUX WORKSTATION, NXP S32V BOARD, AND MACBOOK PRO

The workspace, shown in Figure 10 above, includes the NXP S32V vision board equipped with OpenCV and the Vision SDK 9.1. The included Sony camera was mounted to slot B, a Linux workstation is utilized for building code and a MacBook was used for playing the YouTube input video (link). In order to communicate between the remote board and host Linux machine, an SSH server was set up on the board. To set up SSH, the `/etc/networks/interfaces` file was edited to include the static IP address, netmask, and gateway. With this workspace setup, files on the board could be edited remotely using `ssh`, and code could be compiled and transferred to the board from the workstation using `scp`. The MacBook was used to display real-time videos in order emulate what a car may be exposed to on the road.

## C.2 - Research

To develop this submission, the Vision SDK environment and toolchain were studied. The demo `isp_csi_dcu.cpp` source code was useful in figuring out how the camera collects frames and processes information. The `face_detection.cpp` code was helpful for understanding how to integrate OpenCV libraries into the codebase. After becoming comfortable with the supplied demos, the submission code was started. The main code is called `isp_framerate.cpp` and is heavily based off of the `isp_csi_dcu.cpp` demo. There were two main objectives for the source code; first, calculate frame rate and then, print text to each frame.

## C.3 - Implementation

To calculate the frames per second (FPS) elapsed time, $\Delta t$ for a single frame was stored. The FPS was then calculated as.

$$FPS = \frac{1}{\Delta t}$$

Equation 2: FPS Calculation

For debugging purposes, the FPS calculation was printed to the console. The next step was to print this information on each frame before outputting to the display. To alter the frames, the optimized OpenCV libraries were employed. It was confirmed that the necessary OpenCV library paths were linked correctly, then the next step was to display text on the screen.

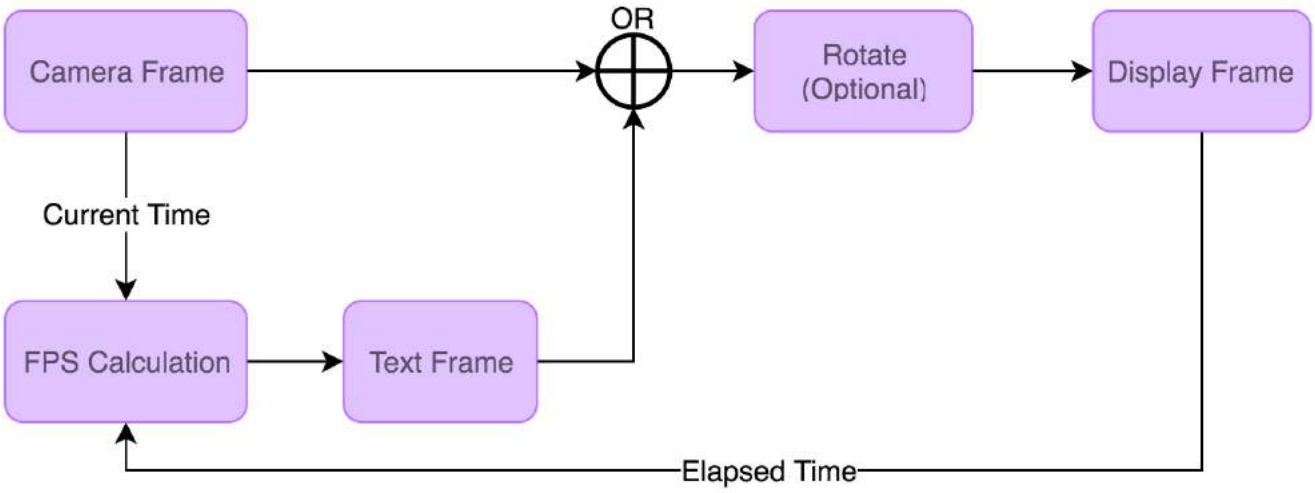


FIGURE 11 BLOCK DIAGRAM OF ACTIVITY 3

The line: `Mat frame = Mat(720, 1280, CV_8UC3, lpFrame)` allows the conversion to OpenCV's class `cv::Mat` from `a void * lpFrame`, (a pointer returned from OS Abstraction Library (OAL) memory allocation function). The `cv::Mat` frame is used to draw text on the frame using `cv::putText` function, which provides parameters such as font face, font scale, color, thickness, etc. Once the text was successfully displayed on the bottom left of the screen, it needed to be rotated to correct orientation since the text was shown upside down relative to the camera frames. This issue stems from the fact that the built-in display was installed upside down. Though it is not a requirement of this activity, the issue was investigated to make the display easier to read. There are two stages where the frame rotation can be adjusted; pre and post rotation.

Pre-rotation is directly handled by the Sony camera by configuring camera geometry. The code below flips the display such that the frame is shown upside down without any additional post-rotation.

```
SONY_Geometry_t lGeo;
SONY_GeometryGet(CSI_IDX_0, &lGeo);
lGeo.mVerFlip = 1;  // flip about y-axis
lGeo.mHorFlip = 1;  // flip about x-axis
SONY_GeometrySet(CSI_IDX_0,&lGeo);
```

Post-rotation is handled from the frame after it's been represented as a `cv::Mat` data structure, which supports the `cv::flip` function to rotate. Note that a flip operation about a single axis is not equivalent to rotation. The flip operation should be about both the x and y-axis to be equivalent to rotating 180 degrees. The following code performs 180 degree rotation if `ORIENTATION` is set to 1.

```
if (ORIENTATION) {
  flip(frame, frame, -1);
}
```

The third argument to the flip operation is the orientation code. -1 yields a flip about x and y-axis.

The results of this rotation are not trivial. The FPS is about 3x faster when `ORIENTATION == 0` and the `cv::flip` function is omitted, therefore final submission initializes `ORIENTATION == 0`. The video submission of the display output rotated using editing software so that it is clearer to the viewer.

## C.4 - Issues

Workspace issues mostly came down to misunderstanding the Vision SDK capabilities and workflow. There were initial problems getting networking on the board and additional issues building the demos. After improved understanding of the toolchain and directory structure, the necessary file system tweaks were applied and the demos were finally functional. It was also discovered that the make clean operation sometimes deleted headers that are needed to build the code. This was fixed by editing the make file.

In the implementation, there was trouble using the OpenCV namespace, but the solutions came down to properly linking the libraries when building the program. Additionally, understanding how to convert lpFrames to a `cv::Mat` was a struggle, however, looking at other demo code helped resolve this. Lastly, the rotation issue as discussed above.