University of Colorado
Boulder

AUTONOMOUS VEHICLE PROTOTYPE SYSTEM

BY

SIDDHANT JAJOO
SARTHAK JAIN

UNDER THE GUIDANCE OF:

PROFESSOR SAM SIEWERT,
UNIVERSITY OF COLORADO BOULDER

8/9/2019

BOARD USED: NVIDIA JETSON NANO.

TABLE OF CONTENTS

INTRODUCTION

A large number of the corporate giants are working on the problem of self-driving cars, without having a marketable prototype at this stage. However, each one of the systems of these companies will only be available to the cream of the market, not being available for the common man during the early stages. The program we are working on will be a low-cost software, having the ability to be integrated with any car having a sufficient set of requirements.

The topic we have selected, is a self-driving car prototype using pedestrian detection, lane following, traffic-signal detection and vehicle detection. Speaking on a smaller scale, the rationale behind selecting this topic is the chance to integrate all the skills acquired previously during the course, along with the opportunity to look into advanced machine vision algorithms and possibly machine learning algorithms as well.

The final prototype will involve the use of image processing transforms for preprocessing in order to clean up each subsequent frame to remove unwanted objects. Noise will be removed using either Gaussian filters or the median filtering techniques. Subsequent processing will involve detection of each particular object on different threads, employing multi-processing capabilities of the Jetson. The final objective is to reach a stage where the program can emulate to some degree a self-driving car, by slowing down or stopping when it detects pedestrians, follow a lane by tracking the curves of the lines marking roads, detect signs and react appropriately, and detect vehicles in front as well, to slow down if the vehicle in front is too close, or accelerate within speed limits given no vehicle is in front.

FUNCTIONAL REQUIREMENTS

1. Accurate detection of objects and targets:
   In the early testing stages, often times the program would detect a reflection or a side-object as the target rather than the intended target itself. Making sure these false negatives are eliminated so that only the positive images are processed and used for detection is one major requirement. The elimination of negative images would lead to speed-up in the program, and accurate detection of objects is a must.

2. Maintaining a feasible frame rate:
   Each processing stage is highly computationally intensive, and when run on a single core, gives an extremely low rate of Frames/second. We have used multithreading and the use of a scheduler to regulate our frame rate, and obtain a feasible rate, but further processing could lead to better rates. If the frame rate falls too low, the program would no longer be real-time, and be of virtually no use. The frame rate can be made to reach higher values, but at the cost of accuracy. A balance would need to be achieved between frame rate and accuracy of detection.

3. Integration of all services:
   All the services must be appropriately integrated to offload processing to the different CPUs. Ensuring that the processing of one service does not affect the other services will be key to ensuring a successful program. Writing an individual program for each service is straightforward enough, but when the services are integrated into one program, all sorts of discrepancies like timing mismatches, insufficient processor and deadlocks when interacting between the shared resources come up. Allocating CPU resources as well to ensure the output is such that the car has enough time to react is also important.

4. Real-time scheduling requirements:
   Initially, the design involved the use of semaphores to co-ordinate the release of services based upon when the main() function would grab a frame. However, this led to scheduling problems, causing deadlocks. To resolve this, we had to come up with the use of scheduler to coordinate timing usage of each service. The analysis of time period and request rate was integral to our analysis and design. During the course of this design, timing mismatches did come up, but were resolved and have been explored further on.

5. CPU, memory and latency requirements:
   Based on how frequently each service is allowed to run, the processor utilization of the hardware varied greatly. At a particular setting in our program, it is possible to achieve over 90% CPU utilization on all 4 processors, however this is not recommended as the user interface does not take too kindly to this setting. It is optimal to maintain CPU usage at around 70%-80%, for each processor, which we have achieved.

MACHINE VISION REQUIREMENTS

Each service had its own set of minimal, target and optimal requirements to be achieved.

1. Pedestrian Detection:
   The minimal requirement was the detection of all pedestrians in a particular frame without any being missed. However, the stage reached for this service is the target stage. Initially, pedestrian detection was quite slow, giving a frame rate of 5-6 FPS. Using a combination of scheduling policy and optimized parameters for the detection API, a higher frame rate of 10-15, depending upon the size of the input video was achieved. This is quite a sufficient frame rate, seeing as the rate of reading input frames is 30. According to our scheduling policy, we assigned a deadline of 67 ms, which is achievable upon slightly finer tuning. The optimum set of requirements seems to be rather far off at this stage, since the input video being used for testing has a low resolution, and detection of pedestrians on the periphery of the camera frame is not feasible given this algorithm and hardware.

2. Lane Following:
   The minimum requirement was to detect the lanes on either side of the vehicle, which has been achieved. Solid lines on either side have been detected, and the algorithm can be made to detect broken lines with minor threshold changes to the code. A lot of processing using different transforms is performed in this service, so passing parameters between function calls was the tricky part. We can aim for the target requirement of detecting the center between the two lanes, and matching it with the center of the entire frame (assumption being the camera is mounted along the central line of the car).

3. Traffic signal detection:
   Initially, the plan was to detect a number of road-signs and integrating it with a traffic signal detection system in order to provide some semblance of a truly smart system. However, even if the different road signs and the traffic signals are detected, segmenting and classifying them would be the next problem stage. At this point, Haar cascade classifiers are being used to detect traffic signals, based upon the color. The catch of this is that if another appropriately round object with the exact shade of red, green or yellow appears in the frame, the program classifies that too as a traffic signal. This can be rectified further in the code, but the minimum requirement of detection of traffic signals even from afar has been achieved.

4. Vehicle detection:
   Vehicles in front as well as to the side are currently being detected. However, when the vehicle gets too close, it is not detected. This is something to be improved in further stages. Further scope could be measurement of the distance between the vehicle in front and the position of the car, so that if the distance reduces too much, a STOP signal can be sent to the braking unit.

FUNCTIONAL DESIGN OVERVIEW

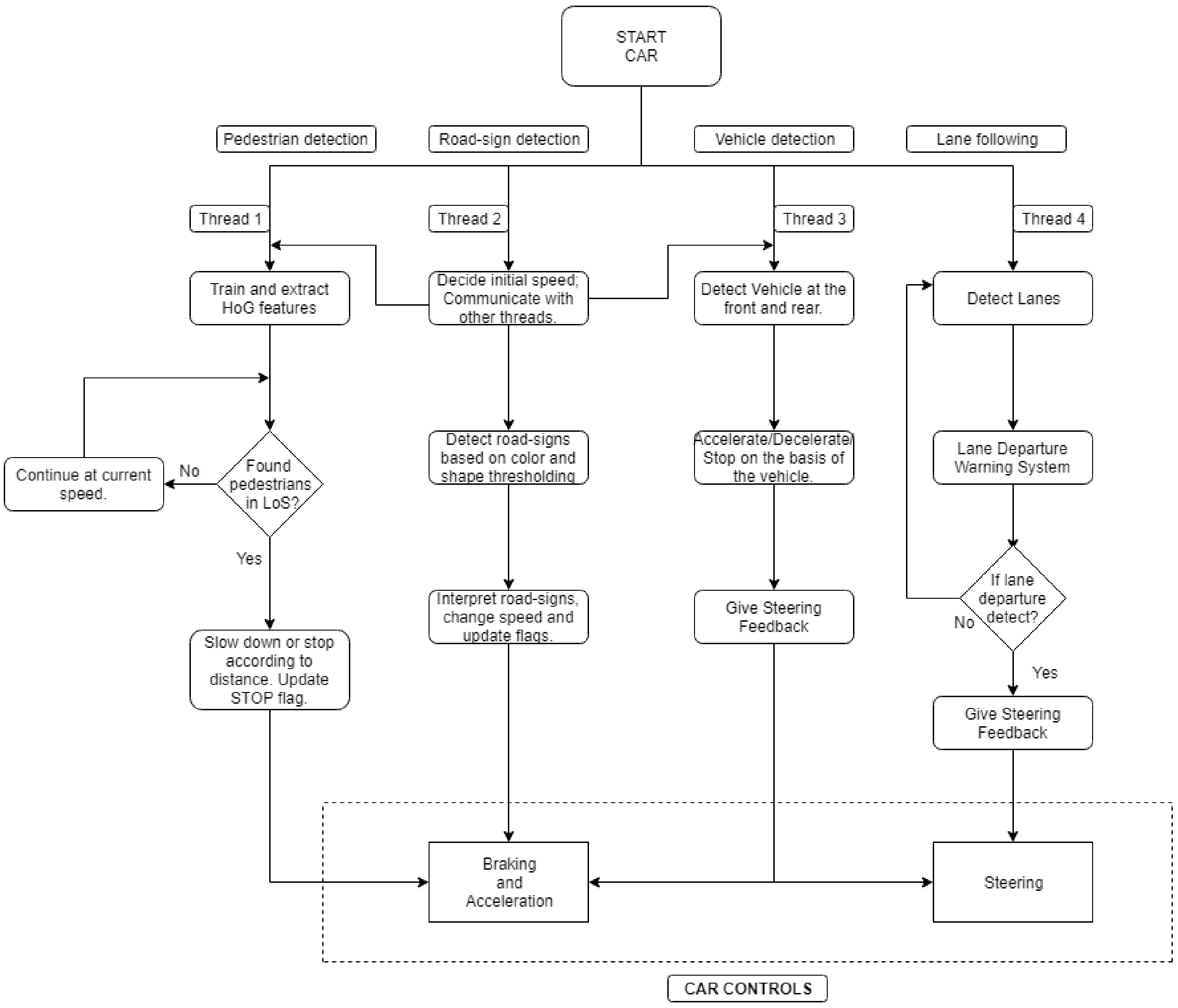
INTRODUCTION

The Project consists of majorly 6 parts:

1. Basic Structure
2. Scheduler
3. Lane Detection
4. Pedestrian Detection
5. Vehicle Detection
6. Traffic light Detection.

The project was kicked off by building Land Detection and Pedestrian Detection functions by Siddhant and Sarthak respectively. After creating base working functions for lane and pedestrian, they were integrated into a single project. In order to make them work smoothly, a lot of factors had to be constantly adjusted and modified. The goals that had to be achieved were an acceptable FPS and correct detection of objects in question. Care had to be taken that the CPUs do not get overloaded and the system had to be robust. In addition to this, the design had to be such that it should be fail-safe and should work in even in worst-case situations. A failure of a single service should not affect the other ongoing services. For example: If in any case lane detection gets blocked at some point, the system should keep on functioning irrespective of the situation of other services. In order to achieve this, different methods were employed through the project development course such as use of semaphores and mutexes, different scheduling policies such as CFS and SCHED_FIFO, use of OpenCV, use of a scheduler and rate monotonic analysis. In order to get optimal results with the constraint of resources at hand, Haar cascade classifiers, HOG, Canny and Hough transforms, down and up conversion of images, resizing of images were used. We will discuss how the different parts of the project were implemented by looking at the functional diagram and the basic flow of the project.

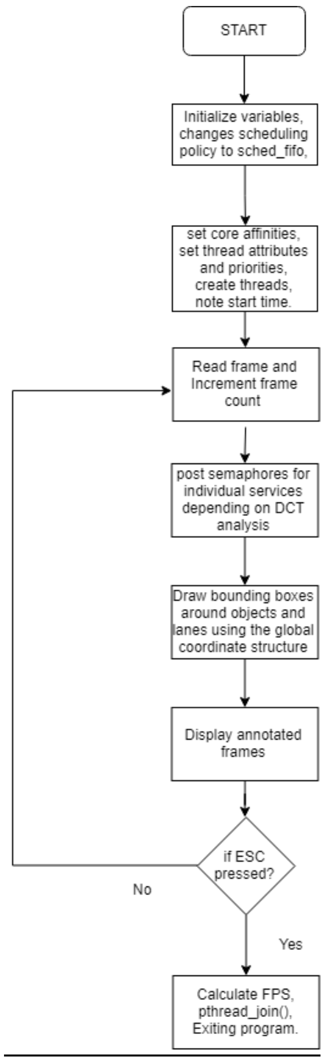
PROJECTED FUNCTIONAL DESIGN


START CAR
Pedestrian detection
Road-sign detection
Vehicle detection
Lane following
Thread 1
Thread 2
Thread 3
Thread 4
Train and extract HoG features
Decide initial speed; Communicate with other threads.
Detect Vehicle at the front and rear.
Detect Lanes
Continue at current speed.
No
Found pedestrians in LoS?
Yes
Detect road-signs based on color and shape thresholding
Accelerate/Decelerate/ Stop on the basis of the vehicle.
Lane Departure Warning System
Slow down or stop according to distance. Update STOP flag.
Interpret road-signs, change speed and update flags.
Give Steering Feedback
If lane departure detect?
No
Yes
Give Steering Feedback
Braking and Acceleration
Steering
CAR CONTROLS

The block diagram above is the projected functional design overview of the project. Some parts of it are up and running perfectly while some still need some refining. The project basically consists of 4 services – Lane Detection, Road Sign Detection which has been updated to detect traffic signals on roads, Vehicle Detection and Pedestrian Detection. The lane detection module would detect lanes and trigger a warning if it crosses a lane thus enabling the lane departure warning system with steering feedback. Vehicle Detection would detect the vehicles at the front in order to determine if it should decelerate or not based on the distance of the vehicle in front. Pedestrian Detection would detect any pedestrians on road and if present any, would trigger the vehicle to stop, slow down or continue at the current speed. The same is applied for road sign detection, if it detects a stop sign, the vehicle must stop. .An additional module was added to detect traffic lights. This would detect red lights on the streets and make decision to whether stop the car or not.

1. Basic Structure:
   The Basic structure of the project consists of multiple threads and a scheduler. Each service has been assigned a particular thread. Pthreads were used in order to achieve this. The main logic behind using multithreading was to improve the speed of the system. The scheduler was designed in the main process along with reading the frames from the video and drawing the appropriate rectangles and lines on the image. The project was executed on Jetson-Nano which has 4 CPUs thus taking advantage of the multithreading concept. A global structure was declared which would store the coordinates of the objects detected in order to draw lines or bounding boxes around them. Since the final output from multiple threads had to be drawn on the same frame, a global frame was declared which would read the frames from the video. The threads would then clone this frame to work locally and detect objects and note the coordinates in the global structure. These coordinates would then be used in order to draw bounding boxes on the global frame. A scheduler was also implemented in order to keep a tab on the individual services and it was used to deploy services at particular intervals of time using semaphores. In addition to this, mutexes were also used in order to protect the global variables from data race conditions.

2. Scheduler:
   The initial design for this project included just use of 5 semaphores in order to control the 4 services. The idea was to post all the 4 services using a single semaphore from the main process as soon as a frame is read from the video. In turn when the respective threads complete their execution, a semaphore was posted from the respective thread which the main process used to wait on. This would ensure that the next frame would only be read if all the threads complete their execution. The scheduling policy used here was CFS i.e. completely fair scheduler. This method was very time consuming and a lot of processor time was spent idle. The FPS obtained was very low. This design had a lot of timing problems which used to result in deadlocks. CFS scheduling policy along with semaphores was the cause for the deadlocks. The program used to stop at any point of time.
   In order to improve this, 4 different semaphores were used in addition to the existing 4 semaphores by replacing the 5th semaphore. This solved the problem of deadlock, but the method was still inefficient fiving us FPS of 2 to 5. We then tried setting the affinity of different CPU cores to particular threads in order to improve performance. This did not have any significant performance improvement over the previous one.
   Since the FPS obtained was not more than 5, we thought of changing the scheduling policy to SCHED_FIFO to obtain a more deterministic behavior. Having a deterministic behavior helped us to find the root cause of the problems and improve them.


START
Initialize variables, changes scheduling policy to sched_fifo,
set core affinities, set thread attributes and priorities, create threads, note start time.
Read frame and Increment frame count
post semaphores for individual services depending on DCT analysis
Draw bounding boxes around objects and lanes using the global coordinate structure
Display annotated frames
if ESC pressed?
No
Yes
Calculate FPS, pthread_join(), Exiting program.

Scheduler Design

By implementing SCHED_FIFO we saw an increase in the FPS. Furthermore, implementing a scheduler drastically improved the FPS to 15 with all the 4 services being implemented. Now all that needed to be done was to have the correct DCT parameters and create a feasible schedule in order to get a much higher FPS. In order to take full advantage of the SCHED_FIFO scheduling policy, a scheduler was necessary in order to deploy the services at particular points of time. One point to note here was that elimination of mutex gave a much higher FPS i.e. at least 6-8 more as compared to the code with mutexes incorporated in it. After carefully reviewing the code, it was observed that the mutexes were placed around functions that took the maximum time in the entire code. One way to evade this is to use local variables in those functions as opposed to the global variables. The values then stored in these local variables would be then copied to the global variable with mutexes placed around this copy function. As copy function take a lot less amount of time, the downfall of FPS due to mutexes would be eliminated.
Thus, after several iterations, additions and modifications, a proper scheduler with SCHED_FIFO scheduling policy and rate monotonic analysis was implemented for this real time system.

3. Lane Detection:
   The image obtained by cloning the global frame is preprocessed at first. The preprocessing includes down converting the image to reduce the resolution by 2 and the chopping of the top half of the image. The top half consists of sky thus it has no requirement in lane detection. The obtained images' contrast is then increased in order to differentiate between the roads and the lanes easily. Now in order to detect lanes, a mask is created which accurately detects yellow and white lanes with the help of appropriate threshold values to detect yellow and white lanes. The mask is created by converting the image to hls and hsv color space to detect the white and yellow lanes. This yellow and white mask is then ORed in order to obtain a single mask. This mask obtained undergoes an AND operation with the contrast image in order to obtain the lanes. The resulted frame obtained from the previous step is applied with a Gaussian Blur. This step ensures it smoothens the image without affecting the contours of the frame. The next step is to employ edge detection using Canny Transform with the appropriate threshold values to detect the dashed lines as well as the bold ones. Once the canny transform is applied, we get an edge detected image. This frame will also consist of certain background objects as well which fall in the range of yellow and white color. In order to eliminate these background objects and noise, a ROI mask is created and applied to the frame. This will eliminate all the noise and background objects that do not belong to the ROI. The obtained canny image with the ROI is applied Hough Line transform in order to obtain the coordinates that represent straight lines. These coordinates are then joined using line() function in order to mark the detected lanes.

4. Pedestrian Detection:
   Pedestrian detection is a simple algorithm as compared to that of lane detection. The global frame is cloned, and the obtained cloned frame is converted to grayscale. It is further resized in order to reduce the computational load on the CPU i.e. to make the system much faster. Then a simple multiscale function is called with the appropriate parameters using the Hog Descriptor. The parameters are fixed by a trail and hit method depending on which set of parameters yields the best results.

5. Vehicle Detection:
   In this algorithm, the same preprocessing is done as in lane detection to downsize the frame resolution and chop off the top part in order to reduce the computational work. Haar Cascade Classifier is used in order to detect the vehicles. The multiscale function is used in order to detect vehicles with appropriate parameters. Vehicle Detection algorithm requires some improvements in order to remove the false positives, yet it can detect cars in front and at the sides of the vehicle. The FPS obtained is reasonable, but the accuracy needs to be improved which we would work on in the upcoming days.

6. Traffic Sign Detection:
   This too uses Haar Cascade Classifiers. The process is the same as vehicle Detection which includes converting the image to grayscale, increasing contrast and then applying the multiscale

function. Traffic Sign algorithm also requires some improvements in terms of accuracy and eliminating false positives. The FPS would be taken care of eventually.

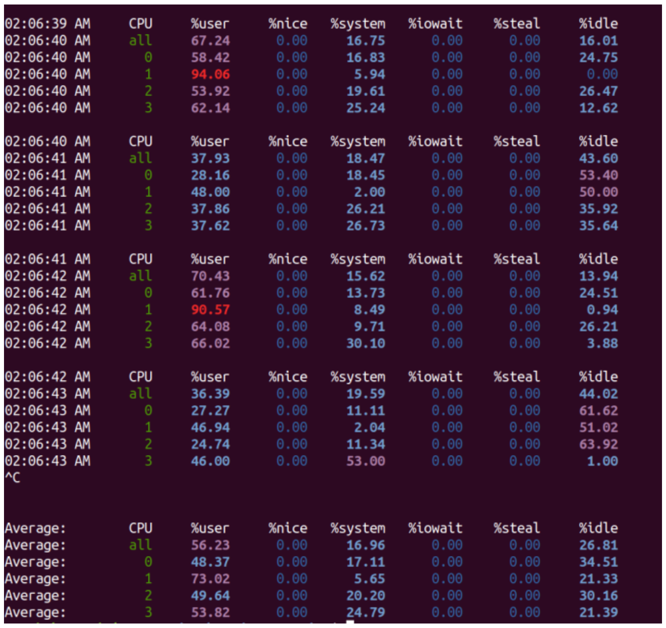
So, all in all, the FPS that we are getting is fluctuating in 20s somewhere around 24- 27 for an ideal 30fps frame rate.

ANALYSIS - PERFORMANCE

1. Robust Program design:
   The code has been written in such a way that the entire program is quite robust, in keeping with the basic concepts of real-time multithreaded systems. 4 threads have been created, which we can assume the OS will distribute evenly between each of the available 4 processors, whenever free using SCHED_FIFO as the scheduling policy. Each thread runs one of the 4 services. Even if one service does fail, it will do so gracefully, by allowing the other services to continue processing. The output will lack the results of the failed service, but the program will still be operational.

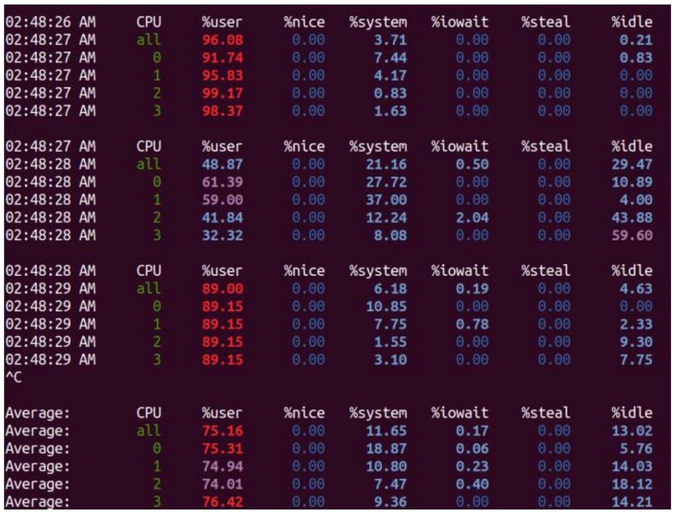
2. Worst-case execution(CI – Critical Instance):
   The presence of a scheduler ensures that even if all the services are released at the same time (t=0), their subsequent execution will be controlled and deterministic. The frequency at which we release each service is such that each service is able to complete its execution within the deadline, and the presence of 4 processors reduces the planning complexity. Even if we consider the main thread, which would bring the design to 5 threads running on 4 processors, the execution time of main has been accounted for and the release of threads has been done keeping that in mind.

3. Processor utilization:
   Although the hardware does have 4 processors, during the development phase, we still found ourselves running short of cores. This was mainly due to the strict design of our program, which has since been relaxed to keep in mind practical conditions. We were focused on making sure that every service is able to process every frame, without any frames being skipped. Due to this, the processors were almost completely utilized, having no leeway. If we consider the especially intensive service of pedestrian detection, it uses Histogram of Oriented Gradient features and an SVM classifier to detect and populate a vector of rectangles with the detected people. If we examine processor utilization, running just this one service with affinity set to one core, we see the utilization for all processors shoot up to 70%. This led to the conclusion that the service itself is so heavy, that OpenCV offloads execution to the other processors. As this was the scenario with only one service, adding 3 more services would undoubtedly lead to a higher CPU utilization, possibly causing a lag in the program. The program was then modified using the scheduler to release services only at specific intervals of time. Currently, the main thread completes one execution in about 33 ms. This is not an accurate value, but an approximate one, as we have arrived at this value using the delay caused by WaitKey and the time taken in reading a frame. Using time differencing, we determined this value, and are releasing the services such that CPU utilization for all 4 CPUs varies in the range of 50%-70%. However, by changing the interval at which each service is released to one execution cycle of main (33 ms), we can cause overload of the CPUs, a seen in the two screenshots of htop below:



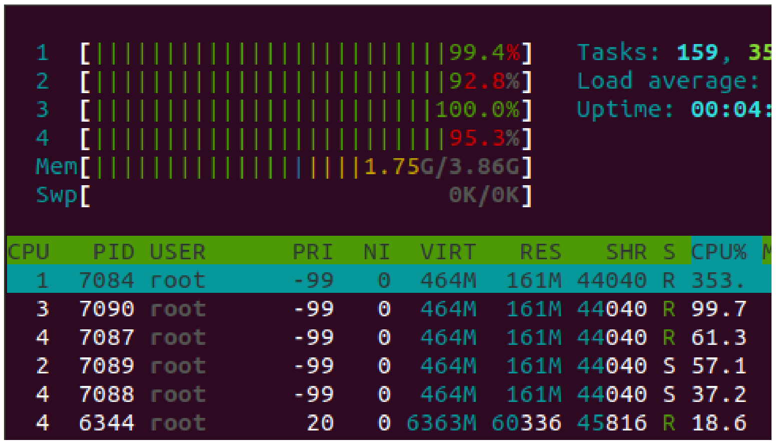
```
02:06:39 AM     CPU    %user    %nice  %system  %iowait   %steal    %idle
02:06:40 AM     all    67.24     0.00    16.75     0.00     0.00    16.01
02:06:40 AM       0    58.42     0.00    16.83     0.00     0.00    24.75
02:06:40 AM       1    94.06     0.00     5.94     0.00     0.00     0.00
02:06:40 AM       2    53.92     0.00    19.61     0.00     0.00    26.47
02:06:40 AM       3    62.14     0.00    25.24     0.00     0.00    12.62

02:06:40 AM     CPU    %user    %nice  %system  %iowait   %steal    %idle
02:06:41 AM     all    37.93     0.00    18.47     0.00     0.00    43.60
02:06:41 AM       0    28.16     0.00    18.45     0.00     0.00    53.40
02:06:41 AM       1    48.00     0.00     2.00     0.00     0.00    50.00
02:06:41 AM       2    37.86     0.00    26.21     0.00     0.00    35.92
02:06:41 AM       3    37.62     0.00    26.73     0.00     0.00    35.64

02:06:41 AM     CPU    %user    %nice  %system  %iowait   %steal    %idle
02:06:42 AM     all    70.43     0.00    15.62     0.00     0.00    13.94
02:06:42 AM       0    61.76     0.00    13.73     0.00     0.00    24.51
02:06:42 AM       1    90.57     0.00     8.49     0.00     0.00     0.94
02:06:42 AM       2    64.08     0.00     9.71     0.00     0.00    26.21
02:06:42 AM       3    66.02     0.00    30.10     0.00     0.00     3.88

02:06:42 AM     CPU    %user    %nice  %system  %iowait   %steal    %idle
02:06:43 AM     all    36.39     0.00    19.59     0.00     0.00    44.02
02:06:43 AM       0    27.27     0.00    11.11     0.00     0.00    61.62
02:06:43 AM       1    46.94     0.00     2.04     0.00     0.00    51.02
02:06:43 AM       2    24.74     0.00    11.34     0.00     0.00    63.92
02:06:43 AM       3    46.00     0.00    53.00     0.00     0.00     1.00
^C

Average:        CPU    %user    %nice  %system  %iowait   %steal    %idle
Average:        all    56.23     0.00    16.96     0.00     0.00    26.81
Average:          0    48.37     0.00    17.11     0.00     0.00    34.51
Average:          1    73.02     0.00     5.65     0.00     0.00    21.33
Average:          2    49.64     0.00    20.20     0.00     0.00    30.16
Average:          3    53.82     0.00    24.79     0.00     0.00    21.39
```

Screenshot of moderate CPU utilization over 4 seconds.



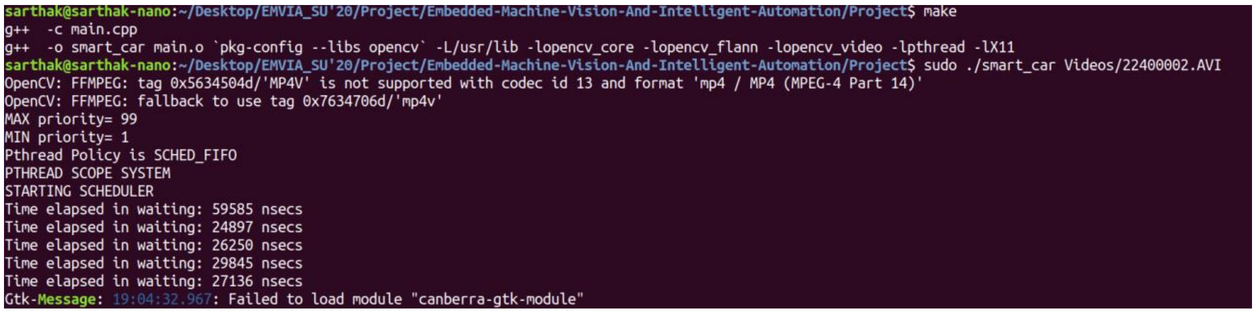
```
02:48:26 AM     CPU    %user    %nice  %system  %iowait   %steal    %idle
02:48:27 AM     all    96.08     0.00     3.71     0.00     0.00     0.21
02:48:27 AM       0    91.74     0.00     7.44     0.00     0.00     0.83
02:48:27 AM       1    95.83     0.00     4.17     0.00     0.00     0.00
02:48:27 AM       2    99.17     0.00     0.83     0.00     0.00     0.00
02:48:27 AM       3    98.37     0.00     1.63     0.00     0.00     0.00

02:48:27 AM     CPU    %user    %nice  %system  %iowait   %steal    %idle
02:48:28 AM     all    48.87     0.00    21.16     0.50     0.00    29.47
02:48:28 AM       0    61.39     0.00    27.72     0.00     0.00    10.89
02:48:28 AM       1    59.00     0.00    37.00     0.00     0.00     4.00
02:48:28 AM       2    41.84     0.00    12.24     2.04     0.00    43.88
02:48:28 AM       3    32.32     0.00     8.08     0.00     0.00    59.60

02:48:28 AM     CPU    %user    %nice  %system  %iowait   %steal    %idle
02:48:29 AM     all    89.00     0.00     6.18     0.19     0.00     4.63
02:48:29 AM       0    89.15     0.00    10.85     0.00     0.00     0.00
02:48:29 AM       1    89.15     0.00     7.75     0.78     0.00     2.33
02:48:29 AM       2    89.15     0.00     1.55     0.00     0.00     9.30
02:48:29 AM       3    89.15     0.00     3.10     0.00     0.00     7.75
^C

Average:        CPU    %user    %nice  %system  %iowait   %steal    %idle
Average:        all    75.16     0.00    11.65     0.17     0.00    13.02
Average:          0    75.31     0.00    18.87     0.06     0.00     5.76
Average:          1    74.94     0.00    10.80     0.23     0.00    14.03
Average:          2    74.01     0.00     7.47     0.40     0.00    18.12
Average:          3    76.42     0.00     9.36     0.00     0.00    14.21
```

Screenshot of very high CPU utilization over 4 seconds.

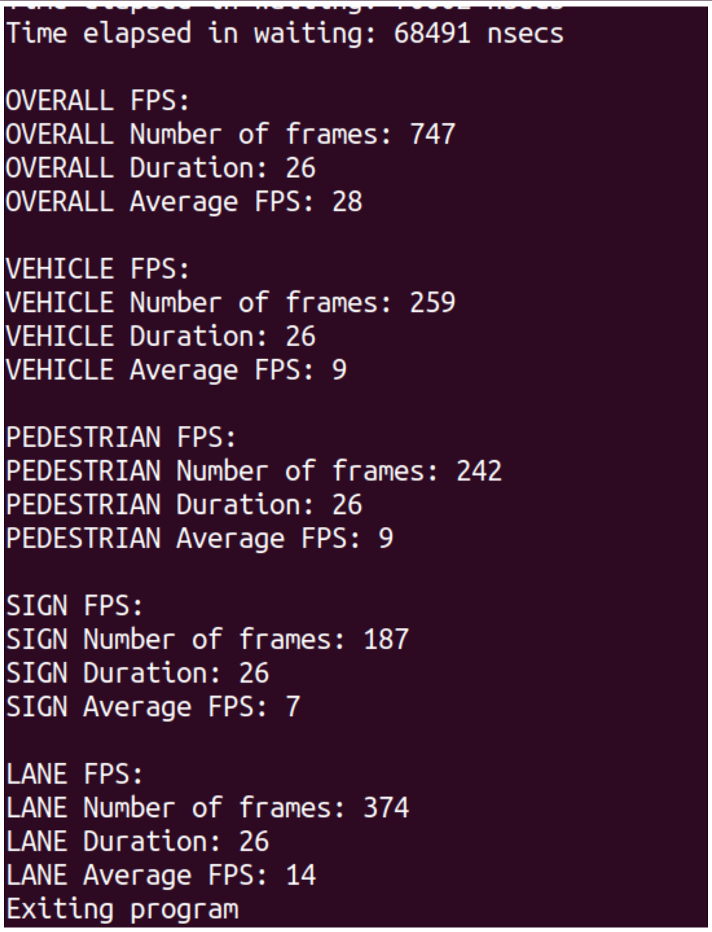


```
  1  [||||||||||||||||||||||||| 99.4%]   Tasks: 159, 35
  2  [|||||||||||||||||||||||||| 92.8%]   Load average:
  3  [|||||||||||||||||||||||||100.0%]   Uptime: 00:04:
  4  [|||||||||||||||||||||||||| 95.3%]
  Mem[||||||||||||||||||||1.75G/3.86G]
  Swp[                        0K/0K]

CPU   PID USER      PRI  NI  VIRT   RES   SHR S CPU%
  1  7084 root      -99   0  464M  161M 44040 R 353.
  3  7090 root      -99   0  464M  161M 44040 R 99.7
  4  7087 root      -99   0  464M  161M 44040 R 61.3
  2  7089 root      -99   0  464M  161M 44040 S 57.1
  4  7088 root      -99   0  464M  161M 44040 S 37.2
  4  6344 root       20   0 6363M 60336 45816 R 18.6
```

4. Frame per second analysis:
   Since our main thread cycles at every 33.33 ms, and the main thread is the one grabbing a new frame in every iteration, naturally its FPS is restricted to 30. We maintain 5 frame count variables, one for each service, and one for the main thread, to determine whether 30 frames are actually being grabbed or not. Considering the latency in grabbing a frame, and the latency introduced by WaitKey, we obtained a frame count of about 25-27 FPS, which is quite good. If we consider a frame count of 30 FPS, with each service being released at its respective rate (10 FPS for Pedestrian detection, 15 FPS for lane following, 15 FPS for vehicle detection, 7.5 FPS for sign recognition), we can see in the screenshot below that we obtain an FPS value quite close to the calculated values.



```
sarthak@sarthak-nano:~/Desktop/EMVIA_SU'20/Project/Embedded-Machine-Vision-And-Intelligent-Automation/Project$ make
g++  -c main.cpp
g++  -o smart_car main.o `pkg-config --libs opencv` -L/usr/lib -lopencv_core -lopencv_flann -lopencv_video -lpthread -lX11
sarthak@sarthak-nano:~/Desktop/EMVIA_SU'20/Project/Embedded-Machine-Vision-And-Intelligent-Automation/Project$ sudo ./smart_car Videos/22400002.AVI
OpenCV: FFMPEG: tag 0x5634504d/'MP4V' is not supported with codec id 13 and format 'mp4 / MP4 (MPEG-4 Part 14)'
OpenCV: FFMPEG: fallback to use tag 0x7634706d/'mp4v'
MAX priority= 99
MIN priority= 1
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
STARTING SCHEDULER
Time elapsed in waiting: 59585 nsecs
Time elapsed in waiting: 24897 nsecs
Time elapsed in waiting: 26250 nsecs
Time elapsed in waiting: 29845 nsecs
Time elapsed in waiting: 27136 nsecs
Gtk-Message: 19:04:32.967: Failed to load module "canberra-gtk-module"
```



```
Time elapsed in waiting: 68491 nsecs

OVERALL FPS:
OVERALL Number of frames: 747
OVERALL Duration: 26
OVERALL Average FPS: 28

VEHICLE FPS:
VEHICLE Number of frames: 259
VEHICLE Duration: 26
VEHICLE Average FPS: 9

PEDESTRIAN FPS:
PEDESTRIAN Number of frames: 242
PEDESTRIAN Duration: 26
PEDESTRIAN Average FPS: 9

SIGN FPS:
SIGN Number of frames: 187
SIGN Duration: 26
SIGN Average FPS: 7

LANE FPS:
LANE Number of frames: 374
LANE Duration: 26
LANE Average FPS: 14
Exiting program
```

This stands to reason that the calculation of our schedule is quite accurate. Of course, this FPS can be changed by changing the request rate for each service, but that would lead to CPU overutilization. The current frame rate values allow a comfortable utilization margin.

Frames per second requirements for all threads	Minimum (fps)	Target (fps)	Achieved (fps)
Main thread	10	15	27
Pedestrian detection service	2	5	9
Lane following service	6	10	13
Vehicle detection service	6	10	12
Traffic sign service	2	4	6

DCT Analysis for Services

Service S_i	Requested Frequency ($1/T_i$) (in Hz)	Deadline Frequency ($1/D_i$) (in Hz)	WC frequency ($1/C_i$) (in Hz)
Main thread	30	30	30
Pedestrian detection service	10	10	10
Lane following service	15	15	15
Vehicle detection service	15	15	15
Traffic sign service	7.5	7.5	8

Our logic behind selecting the request rate for each service was quite simple. Having used Rate Monotonic Analysis, the deadline and request rate for all services are the same. On a standard highway, speed limits tend to be around 60 mph. If we consider the human reaction time, which is 250 ms to visual stimulus, the vehicle would still travel about 67 m, before the driver is able to respond. (**distance travelled = speed/360*1609*time**). The request rate we have kept for our slowest service (traffic sign) is 7.5 Hz, corresponding to 35m. In one reaction time of a person, two traffic signal periods can pass. Hence the request rate of each service can be justified as being more than twice as fast as the reaction time of a person.

As for the relative comparison of request rates between each service, lane following, and vehicle detection must have the highest request frequencies. A lane can have curves along any part of the read, and missing even 35m of it can be fatal. We have considered 16m as the maximum distance to be travelled between subsequent processing in lane following, which we think is sufficient. Similarly, for vehicles, a vehicle may cut lanes any time travelling at high speed, and requires a high request rate as well of 15 Hz.

Pedestrians are a high-risk detection service, and therefore we have given that service a comparable request frequency of 10 Hz. The chances of a pedestrian turning up in the middle of a highway are low, and in city lanes, the maximum vehicle speed would anyway by 20 – 30 mph. A request rate of 10 Hz at 30 mph gives a maximum untracked distance of 13m, which seems sufficient.

We have assigned the lowest request rate to traffic signals, of 7.5 Hz. At a speed of 30 mph, it gives a maximum untracked distance of 17m.

5. Latency analysis:

   There are three main reasons for the latency in the program:

   a. WaitKey – WaitKey is actually an API that waits until at least the number of milliseconds specified as argument. It can wait for more time if there are other processes the OS deems to be of higher priority. We found that the WaitKey value is one of the prime factors in determining the program's frame count. Keeping a WaitKey value of 33 means an effective block of 33 ms. Added to the delays caused by the rest of the APIs and processing in the main thread, this given an effective time period of 60 ms. We found that a WaitKey of even 1 ms is sufficient, and are allowing the rest of the processes to dictate the time period of the main program. This can be remedied by creating a separate thread performing only scheduling, and using that along with an effective and deterministic delay function to release other threads. This would allow for a much more reliable program.
   b. Frame read delay – Although a simple API for reading frames from an input source one after the other, it has quite a significant delay, of almost 30 ms
   c. Mutex locks – In order to protect access to the global variables we are using in our program, we are using mutex locks for the variables to prevent race conditions from arising. However, since the main thread has such little processing to do otherwise, it spends a lot of its time actually waiting blocked at one the four mutex locks. To be precise, it spends roughly 30-40 ms in the blocked state. It is also worth noting that this delay causes the frame rate to fall to 16 FPS. Additionally, since much of the processing time is spent blocked, the processor utilization is also quite less, at about 30% - 40% on average.

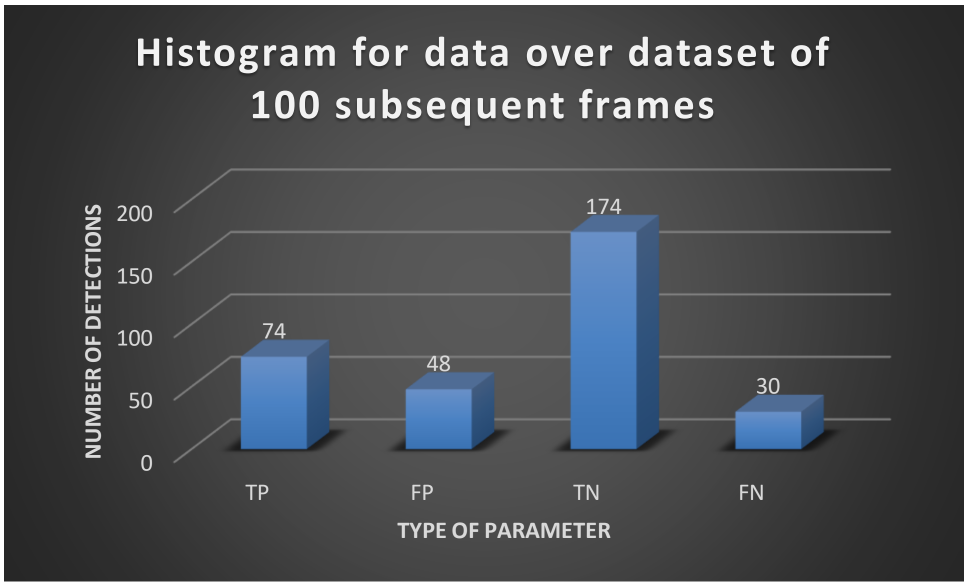
      On the other hand, if we remove the mutex locks, the frame rate rises to 24-25, but the processor utilization rockets to 80%-90% on each processor as well.

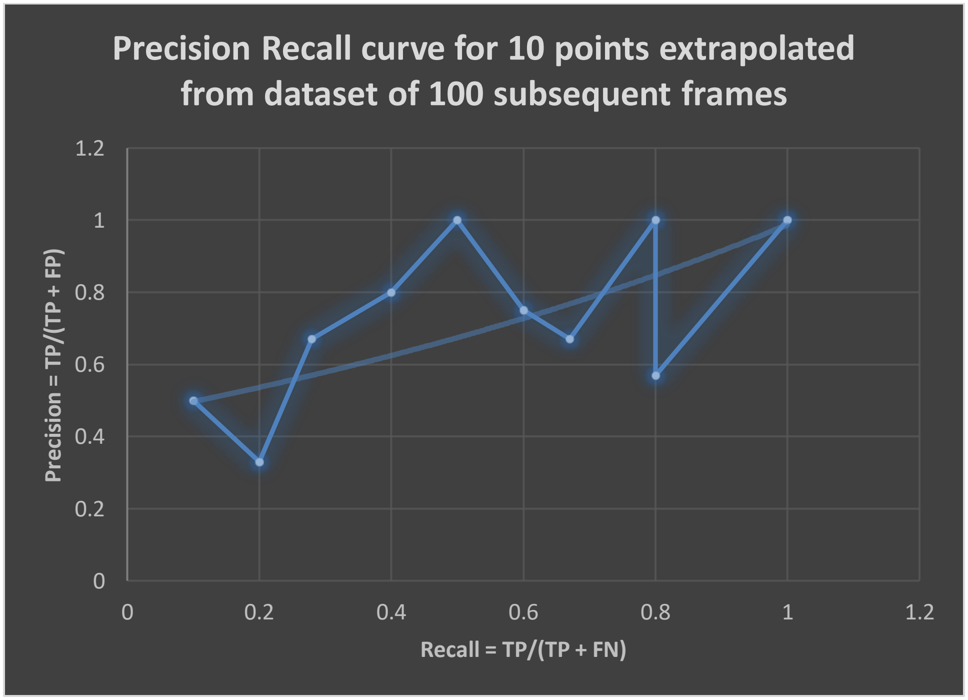
      The reason the mutex locks are so time-consuming is that, they protect the global variables while they are being updated in the different service threads. Each processor-intensive function (detectMultiScale) is protected by a mutex, and it is this function which takes the most time to execute.

      An idea we have to work around this so as not to waste processor cycles in a blocked state, is to use local variables in each thread in the processor-intensive APIs. We can then simply copy these local values to the global variables, protecting this copying action with a mutex. This should result in mutex protection, optimal CPU use, as well as a high frame count.

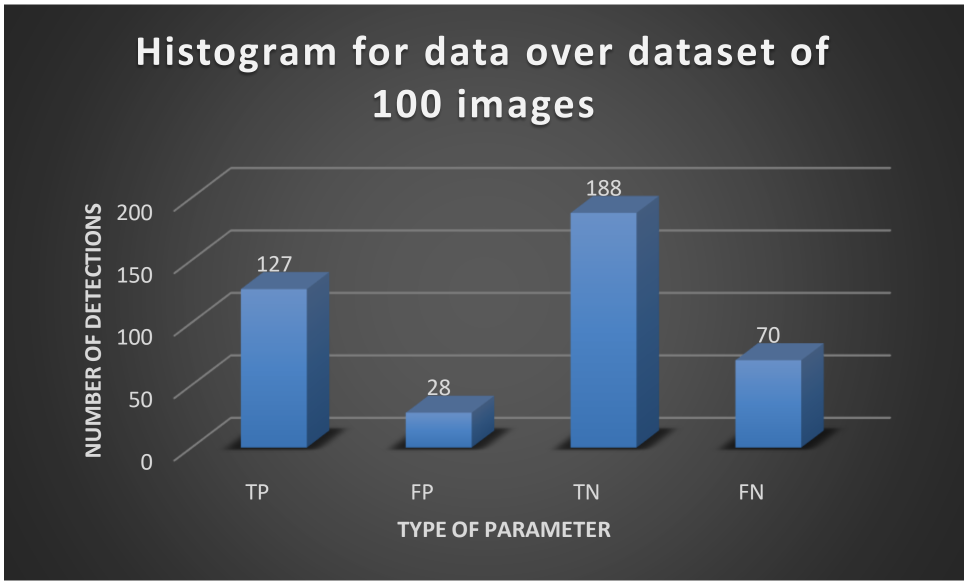
ANALYSIS – MACHINE VISION ALGORITHMS

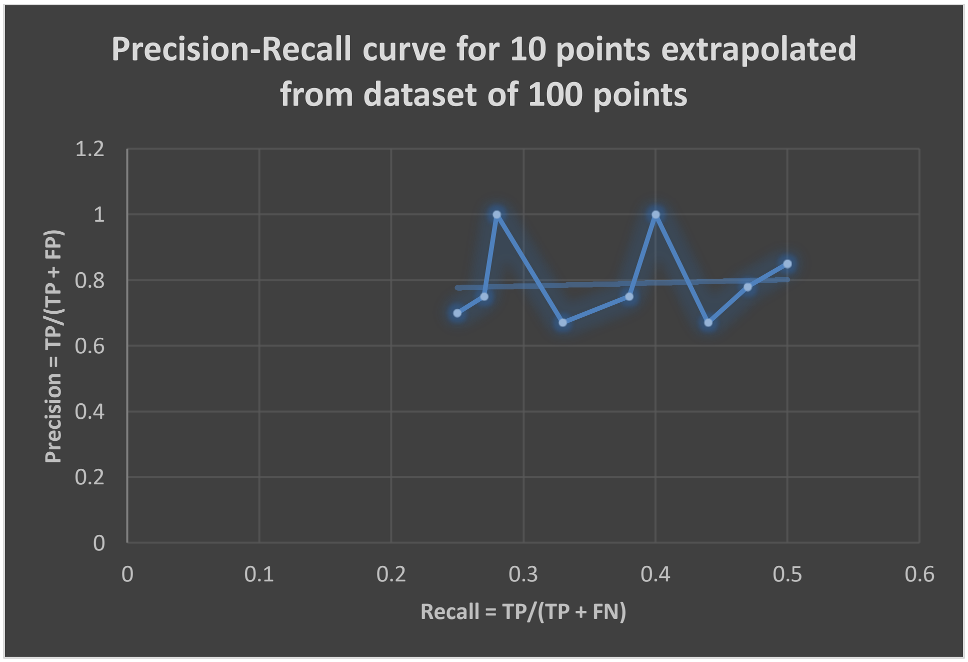
1. Vehicle Detection





2. Pedestrian Detection

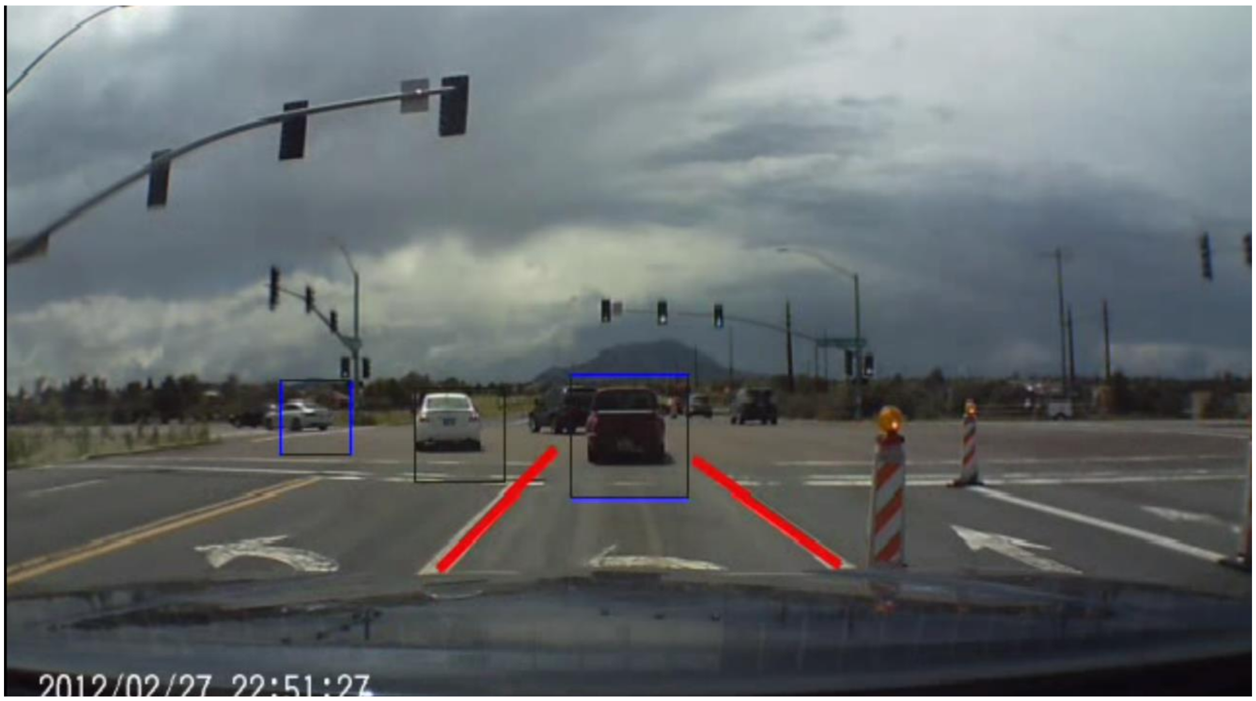




Performance analysis has been performed only for vehicle detection and pedestrian detection, as we could not assume any tangible accuracy measurement parameters for lane following. As we could observe, the lane was detected in almost every frame, but the accuracy could not be measured, as the lane detected sometimes deviates slightly from the actual lane.
For the purpose of traffic-sign recognition, we were unable to obtain a sufficiently large dataset to analyze.

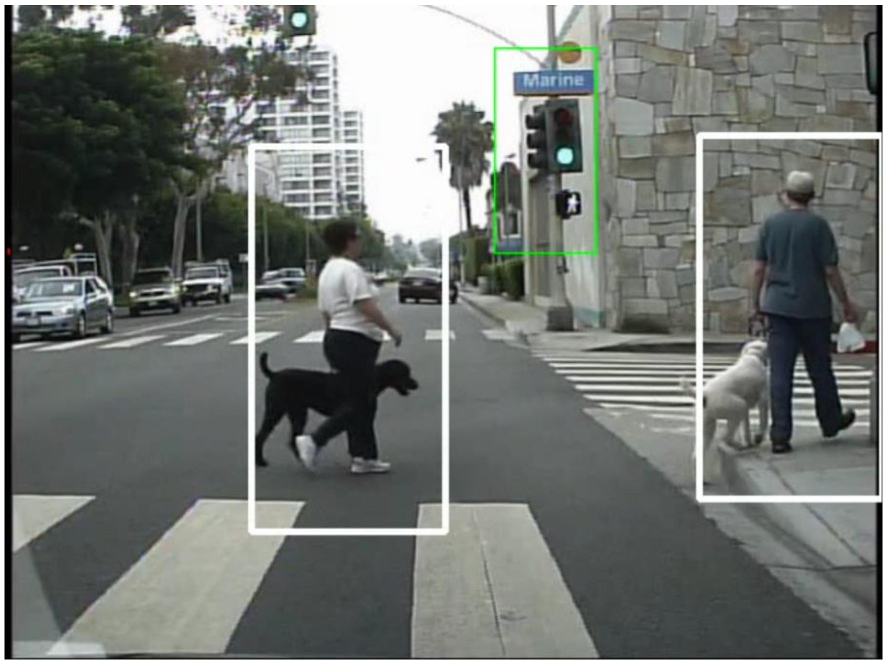
PROOF OF CONCEPT

Testing was done on two sets of videos, one set used for lane detection, and the other as a compiled set of videos from the Caltech Pedestrian dataset. Since no one video contains all 4 features, we have attached two video output along with the submission, the first video is 22400002.AVI, front-facing camera. The video allows us to detect lanes (in red) and vehicles (in blue boxes).

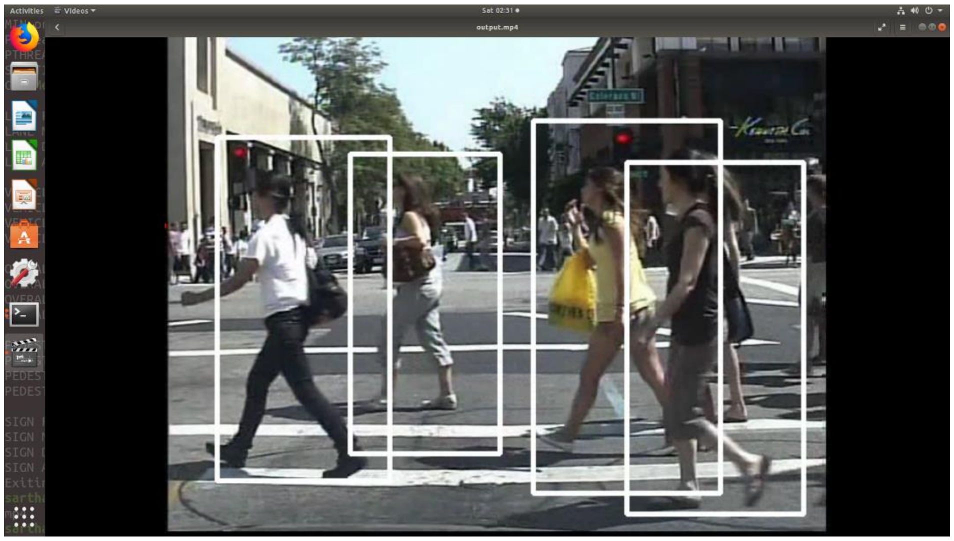


Example Screenshot showing lanes and vehicle detection

Pedestrian detection was performed from a video compiled from the Caltech pedestrian dataset. Two such input videos have been attached, and the program was run on both. In the first image below, we can see pedestrians being detected (white boxes) along with the green traffic light (green box).



Example Screenshot showing pedestrian and traffic signals detection



Example Screenshot showing detection of multiple pedestrians

CHALLENGES FACED

1. Some of the problems that were faced during lane detection were:
   a. One problem that was faced was that multiple lines were being detected for a single lane. This was solved by averaging all the coordinates obtained for that particular line and displaying the averaged line for the lane.
   b. Another one was several horizontal lines were being detected since the mask was not accurate enough due to background noises and other objects and arrow signs on the roads. These were eliminated in the final result by keeping a filter of detecting only those lines that had a slope between 0.5 and -0.5. The slope also helped to bifurcate left and right lines in the frame.
2. Numerous problems were faced while implementing the scheduler. All the problems faced while implementing a scheduler have been described clearly in the scheduler section on page 7 in functional design overview section.
3. Some problems were faced while detecting Vehicles and Traffic-signals. These are work in progress and are expected to be completed in the upcoming days.
4. The only challenge in Pedestrian Detection was to select the correct set of parameters in order to achieve accuracy as well higher FPS.

CONCLUSION AND FUTURE SCOPE

The services lane following, and vehicle detection were implemented by Siddhant Jajoo, while Sarthak Jain worked on the pedestrian and traffic light design. Both worked on designing the scheduler.

To conclude, we can state the results of our analysis. Implementing all four services with their assigned request rates gives an overall frame rate between 24 - 27. This is, provided the main thread executes with a period of roughly 33 ms. Each service delivers an FPS as expected.

Using a scheduler and changing the scheduling policy to FIFO drastically increased the frame rate. Pre-processing of the input image and using the right set of parameters in each individual service was paramount as well.

For future submissions, a lane departure warning system can be implemented by building upon the current lane following system. Traffic sign recognition can be improved and made faster by removing false negatives, adding more road-signs and pre-processing the image respectively. Making the system smarter by detecting vehicles in adjacent lanes and accelerating/decelerating based on the distance between the vehicle in front and the car can be implemented.

REFERENCES

- https://opencv.org/

- Histogram of Oriented Gradients and Object Detection: https://www.pyimagesearch.com/2014/11/10/histogram-oriented-gradients-object-detection/
- Histogram of Oriented Gradients for Human Detection: https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf
- https://www.hackster.io/kemfic/simple-lane-detection-c3db2f
- https://medium.com/@gongf05/p1-finding-lane-lines-on-the-road-7b87fc171da7
- https://towardsdatascience.com/finding-lane-lines-simple-pipeline-for-lane-detection-d02b62e7572b

APPENDIX

The zip folder consists of the following:

1. Code
2. Output Video