VEHICLE SPEED ESTIMATION USING OPTICAL FLOW AND 3D MODELING

by

Indrajeet Datta

A project submitted in partial fulfillment of the requirements for the
degree of Bachelor of Science in Engineering in
Information and Communication Technology

Examination Committee:	Dr. Matthew Dailey (Chairperson)
	Dr. Mongkol Ekpanyapong

Nationality:	Indian

Asian Institute of Technology
School of Engineering and Technology
Thailand
September 2015

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Traffic laws are often broken, and the number of reckless and drunk drivers on the streets is high. With the increasing world population, the number of cars on the road is increasing rapidly, and existing infrastructure is never able to keep up with this increasing traffic. It is increasingly necessary that we implement better traffic law enforcement techniques, so that the safety of people is assured. Speeding on the road is one of the most dangerous forms of traffic law violations, and to regulate speed there is the existing traditional technique of speed estimation via radar systems, which use the Doppler shift phenomenon to detect the speed of moving vehicles. The Doppler shift is the shift in frequency and wavelength from a source as the source moves closer to or away from the observer. This happens because as the source moves closer, each successive wave crest is emitted closer to the observer and every successive waves thus take less time than the previous one, resulting in an increase in the frequency of the sound wave detected. Radar systems use a gun to fire a radar beam at a moving vehicle as it moves towards or away from the observer and calculates the change in frequency, which helps us find the speed of the vehicle. Radar systems, although they have been used for a long time, have many limitations. One of the main limitations of these system is the cosine effect. When the observer is trying to measure the speed of a car moving the road, he/she can only see the component of the motion directly towards or away from the measuring point; the estimated velocity is attenuated by the cosine of the angle between the viewing direction and the direction of motion. For a speed estimate to be accurate, the observer has to stand at an angle close to zero degrees with the vehicle's trajectories, which is not possible, since the observer has to stand in the middle of the road on the line of motion of the vehicle to make that work, which is completely unsafe. Also, the user must have proper training to use a radar device, and often times, the results found are unreliable. In situations where there is more than one vehicle in the field of view, the radar guns often shows erroneous results. Also, one can only detect the speed of a specific vehicle accurately when he or she knows the exact distance of the radar to the moving vehicle. Radars sometimes detect motion of unrelated objects in the surrounding environment like leaves or branches or an airplane overhead. This is called radio interference and is often the reason for inaccurate results. Due to all these limitations of radar systems, it is better to use a system that can measure speed of multiple vehicles moving in arbitrary with more accurately.

Another speed detection technique is by the use of lidar guns that use the lidar technology that helps to measure speed by illuminating the target and analyzing the reflected light. The word lidar comes from the words "light" and "radar". Lidar technology uses the laser's focusing ability and radar's ability of calculating distances by measuring time for a signal to return. Lidar technology uses a wide range of the electromagnetic spectrum to detect objects in the direction of its propagation. To measure the speed of a moving vehicle, lidar speed guns calculate the time of flight of two or more short 905mm wavelengths. The observer needs to aim the gun at the moving vehicle by looking through a telescopic lens and aims at the license plate of the vehicle that reflects the laser pulses back to the lidar gun. The observer aims at the vehicle and pulls the trigger, the gun releases laser pulses, and a timer gets started to store the time required for each pulse to get reflected back to the gun's detector.

Knowing the time of flight, the distance that each of the two pulse traveled can be calculated. The difference of the distance travelled divided by the elapsed time gives the speed of the moving vehicle.

One of the advantages of using lidar guns is that they can detect speed of the vehicle from very far away (around 4000 feet). Another advantage of lidar guns is that it can single out a vehicle out of a group. However, the observer needs to be constantly present at the scene to measure the speed and be sure of the target. Weather conditions can also affect the accuracy of the speed measurements.

Computer vision-based traffic monitoring systems are potentially more efficient and may able to collect more accurate information without requiring a constant human presence. A computer vision system can also monitor more than one vehicle at a time; accuracy is not undermined by the presence of more than one vehicle. Cameras are relatively easy to install and require little maintenance. Computer vision systems may not only help us measure vehicle speed, but they may also provide more information such as vehicle count, license plate numbers, and so on.

1.2 Problem Statement

Although computer vision systems promise better traffic control than simple radar or laser-based speed measurement systems, there are many challenges and a great deal of scope for improved development. Among the challenges in video image processing for vehicle speed detection are shadows and lighting transitions. Lighting transitions from the daytime to evening causes cars to look very different, making them difficult to track with a single strategy. Weather conditions, such as rain or snow that might fall on the road, can also hinder vehicle detection and tracking and may result in inaccurate speed estimation. Another challenge in implementing accurate vision-based vehicle speed measurement systems is that the cameras have to be placed very high above the road in order for simple planar speed models to be accurate, making them expensive or even impossible depending on nearby structure. The main goal of my thesis is to develop an efficient and inexpensive vehicle speed measurement system that can provide results with high accuracy across a variety of imaging conditions from a relatively low height of about 3-4 m.

1.3 Objectives

The main objective is to design, develop and test a method for vehicle speed measurement from a video that incorporates knowledge of the 3D position and orientation of the camera. I will perform the following steps:

1. Implement an optical flow algorithm.
2. Implement clustering of optical flow vectors for elimination of outlier flows.
3. Model observed motion of each cluster as the motion of a cuboid in the world coordinate system.
4. Estimate vehicle motion and speed using the 3D motion of the cuboid.

5. Evaluate the method using ground truth data acquired from a camera mounted three meters above a road.

1.4 Limitations and scope

1. The project does not include detection of vehicle speeds in bad weather conditions such as rain or during night time.
2. The project does not include vehicle classification for other information extraction.

Chapter 2

Literature Review

2.1 Vehicle detection

Iwasaki and Itoyama (2006) propose a method that uses the shadows underneath a vehicle for vehicle detection. This method detects vehicles by estimating the distance between the front and rear tires from the shadows formed underneath each vehicle. In this approach, the camera does not have to be set very high above the ground but rather at a lower position near the sidewalk.

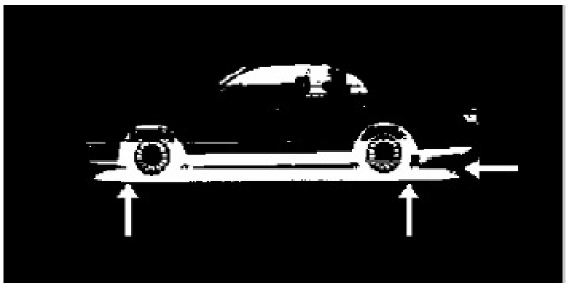


Figure 2.1: Positions of the tires and the boundary between a vehicle and its shadow. *Reprinted from* Iwasaki and Itoyama (2006)

Zhou et al. (2007) propose an algorithm for detecting vehicles that involves approximation from an adaptive background, in which the image is divided into smaller blocks. The method uses principal component analysis (PCA), a general dimensionality reduction technique, followed by a support vector machine (SVM) for vehicle classification.

Yung and Lai (1998) propose an interesting 3D model-based tracking approach in which a 3D solid cuboid having six vertices is fit to vehicles in the images acquired. The vertices of the cuboid are changed until the vehicle image fits the 3D model. This approach efficiently estimates the vehicle location by calculating changes in region proportion in two different images taken at different times.

First, background estimation is used to get a binary mask of the vehicle by first selecting a stationary image of the background. The image consisting of vehicles and noise are then subtracted from the stationary image.This gives a mask of the image without the background. The image is then turned into a binary mask.
This binary image of the vehicle is then fitted into a 3-D deformable cuboid with six vertices corresponding to the visible outer edges of the projected 3-D cuboid, $M = \{V_k | k = 0, 1, 2, 3, 4, 5\}$. The 2-D region is the projection of the edges of the cuboid forming vertices v_0 to v_5

Then the $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ components of this 2-D model are transformed to 3-D space. This transformation is carried out by considering the following camera model in which the focal length is f, the camera height is h, and the camera is focusing at the point o. Any 3-D point $p = (p_u, p_v, p_w)$ is projected

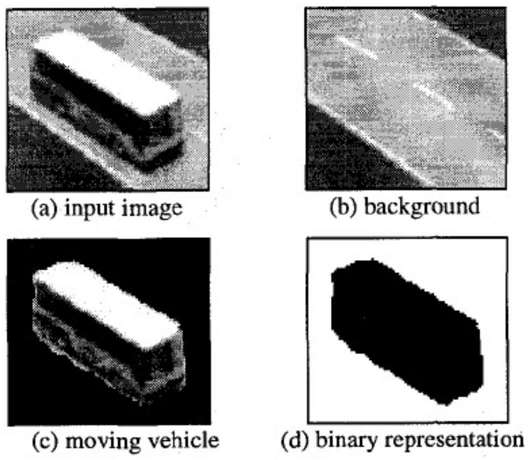


Figure 2.2: Binary representation of a vehicle. *Reprinted from Yung and Lai (1998)*

onto the 2-D plane as $p = (p_x, p_y)$.

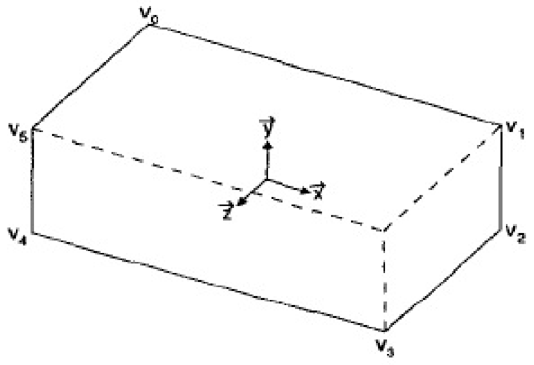


Figure 2.3: Generalized Deformable Model. *Reprinted from Yung and Lai (1998)*

The 3-D to 2-D mapping function (ϕ) is given by

$$p_x = p_u \sqrt{\frac{f^2 + p_y^2}{(d+p_v)^2 + (h-p_w)^2}}$$

$$p_y = f \frac{h.p_v + d.p_w}{d.(d+p_v) + h.(h-p_w)}$$

The 2-D to 3-D mapping function (ϕ^{-1}) is given by

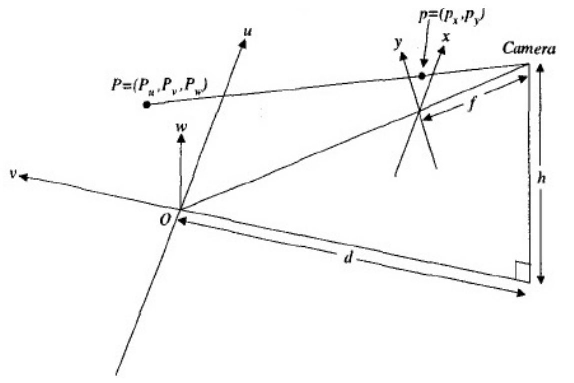


Figure 2.4: Camera Model. *Reprinted from* Yung and Lai (1998)

$$p_u = p_x\sqrt{\frac{(d+p_v)^2+(h-p_w)^2}{f^2+p_y^2}}$$

$$p_v = \frac{p_y(d^2+h.(h-p_w))-f.d.p_w}{f.h-p_y.d}$$

For calculating the height and the width of the 3-D model, we consider the vertices v_2, v_3 and v_4 to be on the surface of the ground and apply the 2-D to 3-D mapping function ϕ^{-1} considering $p_w = 0$.

$$width =\mid \phi^{-1}(v_2, 0) - \phi^{-1}(v_3, 0) \mid$$

$$depth =\mid \phi^{-1}(v_3, 0) - \phi^{-1}(v_4, 0) \mid$$

$$height = v_1 - v_2$$

$$\phi^{-1}(v_1, height) = \phi^{-1}(v_2, 0) + \begin{pmatrix} 0 \\ 0 \\ height \end{pmatrix}$$

Wang and Zhang (2014) propose another approach that exploits shadows under the vehicle for detection. The process of detecting shadows under a vehicle uses an Adaboost cascade with Haar-like features. Adaboost is a machine learning meta-algorithm. The Haar-like feature method detects objects in an image by comparing the pixel intensities of adjacent rectangular regions. It sums up the pixel intensities in each region and then calculates the difference between the sum. A cascade of classifiers is used to progressively eliminate false alarms detected by the Haar-like feature detectors. A vehicle is localized in a region of interest by using histogram of oriented gradient features and support vector machine algorithms.

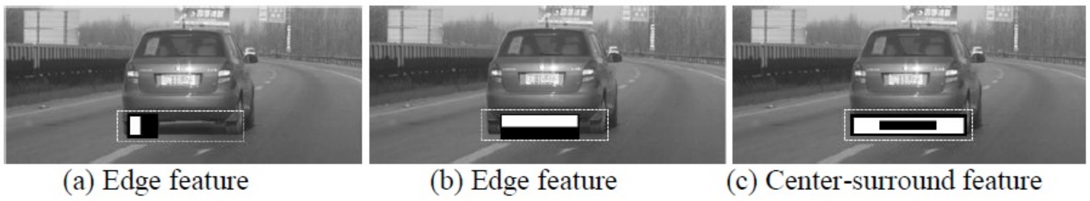


Figure 2.5: Haar-like features for shadow detection. *Reprinted from* Wang and Zhang (1998)

2.2 Vehicle speed estimation

Dailey et al. (2000) propose a method using uncalibrated cameras to measure the speed of vehicles. The algorithm assumes that they have no control over camera movement and cannot get information such as camera focus, tilt, or angle, and it also assumes that they have no authority to set marks on the road. Development of such an algorithm is useful in situations with pan-tilt-zoom cameras that security personnel can control and move, since the algorithm does not require information about camera movements. Vehicles are treated as if they travel along a straight line. Vehicles are tracked to obtain scale factors to estimate the real world distance represented by pixels, then the distance travelled is combined with the known frame rate to estimate speed. First, five or more images are obtained with a sampling rate that is fast enough so that the vehicles do not move more than one vehicle length between images. Then, sets of three sequential images are created, and for each set, a median filter is applied to each image. Then, a difference image contrasting the first and the second image and the second image and the third image are obtained. Sobel edge detection is applied to the two difference images. The edge images are then thresholded to create binary edge images, and the two binary images are thresholded to obtain moving edge images. Dilation and erosion are applied to the moving edge images. A set of points are identified for the j hulls in the moving image, and the centroid is calculated for the jth convex hull. Then, a set of points are calculated for the bounding boxes of the j convex hulls. Then, a set of these collinear points is selected from the sequential images, and a best fit line is estimated passing through these points. For each of the collinear bounding boxes in the sequential images, pixel length is estimated along the v-direction, and the scale factor is estimated for the v location of the centroid using mean vehicle length. Then, a set of scale factor estimates are used to estimate the slope and the scale factor function. Finally the inter-frame distance and the mean inter-frame distance are estimated, and the ratio of the inter-frame mean and the frame rate is calculated to estimate the speed of the vehicle.

Jhumat (2014) proposes a method to estimate speed in which vehicle speed is calculated by measuring the absolute differences between frames. A vehicle mask is calculated by performing thresholding operations, the centroid of the mask is calculated, and the displacement of the centroid over time is used to estimate the speed of the vehicle. First, the video of the moving vehicle is accessed and the number of frames is calculated. Then, from all the frames, only two frames and a background frame are selected. Then, the difference of the two frames with the background frame is calculated, a grayscale image is created, and the grayscale image is converted into a binary image. Then, the binary image is filtered to remove false detection of lane markings and other structures by using morphological processing, and the disconnected components of the vehicle are joined together by performing morphological closing. Then, the centroids of the maximum area of both images are estimated, and the horizontal distance is calculated by subtracting the x-value of both the centroids.

This distance is then converted from pixels to meters by comparing it with the length of the white lane marking, which is assumed to be known. The pixel length of the white lane marking is divided by the known length of the white lane marking in meters, gives the pixel/meter ratio. Multiplying this ratio with the difference of the x-values of the centroid gives the distance traveled by the vehicle. Then the calculated distance travelled by the vehicle is divided by the time-elapsed between the frames to calculate the velocity of the moving vehicle.

2.3 Optical flow algorithms

Optical flow is a technique that is used to predict the motion vector of a moving object that can be used for segmenting an image to track moving objects. It can be defined as the apparent motion of intensity patterns in an image due to the movement of the camera relative to the object. Optical flow vectors help determine the motion of each pixel of a moving object so that we can predict its location in a future time.
For example, for every pixel (x, y) in an image, we want to know where that pixel is relocated to in the next frame. Let the motion vector be (u, v). So, the intensity $I_1(x, y)$ of the pixel (x, y) in the image I_1 must be equal to the intensity $I_2(x + u, y + v)$, i.e.,

$$I_2(x + u, y + v) = I_1(x, y)$$

Deriving the above equation as a function of time we have,

$$I(x + u, y + v, t + 1) = I(x, y, t)$$

The above equation is called the brightness constancy assumption. This is the key assumption we make in optical flow, which is made on the fact that if objects move in the scene the intensity of the pixels that move along with it should not change.

2.3.1 Horn and Schunck method

Horn and Schunck (1981) use a differential technique to calculate the optical flow that is computed by using a gradient constraint known as the brightness constancy assumption and global smoothness. The algorithm, famously known as Horn-Schunck algorithm is implemented in two steps. The first step is to estimate the partial derivatives and the second step is to minimize the sum of the errors.

The partial derivatives of the brightness constancy assumption are estimated through a Taylor series expansion. The equation for the brightness constancy assumption is

$$I(x + u, y + v, t + 1) = I(x, y, t)$$

For the Taylor series expansion, we assume that x and y as functions of t. So,

$$I(x+u, y+v, t+1) = I(x,y,t) + \frac{\partial I}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial I}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial I}{\partial t}\frac{\partial t}{\partial t}$$

But, from the Brightness Constancy Assumption, we know that,

$$I(x+u, y+v, t+1) = I(x,y,t)$$

which implies

$$\frac{\partial I}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial I}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial I}{\partial t}\frac{\partial t}{\partial t} = 0$$

or

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v = -\frac{\partial I}{\partial t}$$

where $\frac{\partial x}{\partial t} = u$ and $\frac{\partial y}{\partial t} = v$
The above equation can also be written as

$$\nabla I(x,y)^T \begin{bmatrix} u \\ v \end{bmatrix} = -\frac{\partial I}{\partial t}$$

which is the gradient derivative of I dot product with the flow vector (u, v)

In the above equation there are two unknowns u and v and one equation to derive them from. So, more constraints are required. So, that means that only the assumption that the intensity of the pixels do not change is not enough to estimate the optical flow vectors because it does not gives us an idea about the path of the pixel motion. The next step is to impose the smoothness assumption which assumes that pixels that are near to each other will have similar flow vectors.

The smoothness assumption is imposed by the following equation:

$$(\frac{\partial u}{\partial x})^2 + (\frac{\partial u}{\partial y})^2 + (\frac{\partial v}{\partial x})^2 + (\frac{\partial v}{\partial y})^2$$

Finally we need to minimize the result of the following equation:

$$\sum_{(x,y)}(\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial v}y + \frac{\partial I}{\partial t})^2 + \lambda((\frac{\partial u}{\partial x})^2 + (\frac{\partial u}{\partial v})^2 + (\frac{\partial v}{\partial x})^2 + (\frac{\partial v}{\partial y})^2)$$

2.3.2 Lucas and Kanade method

Lucas et al. (1981) tries to estimate the displacement of a neighbourhood of pixels by dividing the image into smaller windows and the optical flow equation is assumed to hold for all pixels within the window. This algorithm is famously known as Lucas Kanade method. It is assumed that the displacement of the image contents between two nearby frames is really small and is almost constant within the window.

Let the local image flow vectors be (V_x, V_y) then,

$$I_x(q_1)V_x + I_y(q_1)V_y = -I_t(q_1)$$

$$I_x(q_2)V_x + I_y(q_2)V_y = -I_t(q_2)$$

$$I_x(q_n)V_x + I_y(q_n)V_y = -I_t(q_n)$$

where q_1, q_2, ..., q_n are the pixels inside the window and $I_x(q_i)$, $I_y(q_i)$, $I_t(q_i)$ are the partial derivatives of the image I with respect to x, y and time t evaluated at point q_i at current time.
The matrix form of the above equation can be written in the following way

$$A = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \vdots & \vdots \\ I_x(q_n) & I_y(q_n) \end{bmatrix}, V = \begin{bmatrix} V_x \\ V_y \end{bmatrix} \text{ and } B = \begin{bmatrix} -I_t(q_1) \\ -I_t(q_2) \\ -I_t(q_n) \end{bmatrix}$$

The above system has more number of equations than the number of unknowns and therefore it is over-determined and a compromised solution is obtained by the least square principle.

$$A^T A v = A^T b \text{ or } v = (A^T A)^{-1} A^T b$$

where A_T is the transpose of matrix A.
So, it computes,

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i) I_y(q_i) \\ \sum_i I_y(q_i) I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i) I_t(q_i) \\ -\sum_i I_y(q_i) I_t(q_i) \end{bmatrix}$$

$i = 1$ to n

Chapter 3

Methodology

In this chapter, I propose my methodology for accomplishing the objectives of my thesis.

3.1 System Overview

The model is based on finding the optical flow of the moving vehicle through the Lucas-Kanade algorithm and using a 3-D model to track it and estimate its velocity.

3.2 System Design

The steps for the methodology used in this model are as follows:

- Background subtraction by finding absolute difference of two consecutive frames and then thresholding the result.
- Using morphological operations such as dilation and erosion to eliminate noise.
- Finding convex hulls of the foreground blobs.
- Eliminating blobs that do not satisfy constraints on area, height, width or diagonal size.
- Finding feature points using foreground blobs as a mask
- Finding optical flows for the feature points found in the previous step for each foreground blob in the next frame.
- Matching foreground blobs found over consecutive frames by taking the least distance from the center of the blobs to the the flow heads of the blobs from the previous frames instead of consensus blob method of matching flow head lie on the blobs so that there is lower error due to bad optical flow heads. More information will be provided in the matching algorithm section.
- Keeping track of the convex hulls of each blob over consecutive frames.
- Taking the center of a blob's bounding box and finding the corresponding ground plane position using the inverse homography matrix assuming a height zero.
- Guessing cuboid parameters from ground plane projection of the bounding box's center point and average optical flow direction.

- Modelling the observed motion of the optical flow associated with the blobs as the motion of a cuboid in the world coordinate system on the ground plane.
- Optimizing the dimensions of the cuboid using the Levenberg-Marquardt non-linear least squares estimation algorithm with cost function taking account the observed motion of the optical flow heads.
- Measurement of Euclidean distance between cuboid positions in successive frames.
- Estimation of vehicle speed from the motion of the cuboid over multiple frames.

3.3 Background Subtraction through absolute difference and thesholding

Background subtraction is a preprocessing step that finds the foreground objects of interest, in our case moving vehicles. The background consists of all image pixels that are static over time. I use the absolute difference between every consecutive frame pairs and threshold the result yielding a binary image. The binary image consists of white (pixel value 255) and black (pixel value 0) pixels. The white pixels represent pixels likely to be projections of moving vehicle and the black pixels represent the pixels likely to be on the background.

The OpenCV function `void` **`absdiff`**`(InputArray` **`src1,`** `InputArray` **`src2,`** `OutputArray` **`dst`**`)` calculates the per-element absolute difference between two frames and stores it the destination frame. The output array consists of absolute difference between every corresponding pixels in the two frames. This means the pixels that change the most will have higher pixel values than those that did not change at all.

Thresholding is a method by which image intensities are segmented into groups that are to be emphasized. The threshold operation performs a comparison of each pixel intensity value with a given threshold. The differentiated pixels are assigned to 0 (black) or 255 (white) according to whether they are below or above the threshold.

I perform the thresholding on the difference image obtained from the the absolute difference operation between two consecutive frames. I set the threshold so that pixels with a lower absolute difference than the threshold value are changed to black and the ones with a higher absolute difference value are set to white. The white connected components are called foreground blobs.

The OpenCV function `double` **`threshold`**`(InputArray` **`src,`** `OutputArray` **`dst,`** `double` **`thresh,`** `double` **`maxval,`** `int` **`type`**`)` helps us obtain a thresholded image. This function takes an input image on which the thresholding operation is to be done and returns an output image. The parameters `thresh` specifies the pixel intensity threshold value on which the thresholding operation is to be done. The parameter `maxval` specifies the value to be set to the pixels above the threshold value. The parameter `type` specifies the type of thresholding to be done.

3.4 Elimination of noise from the thresholded image through morphological operations

Morphological operations are image processing operations which are related to the shape or morphology of features in an image. The morphological operations processing on the relative ordering of pixel values of mainly binary images. The operations require a small structuring element which is a binary matrix. The structuring element is used to test the provided binary image by placing the origin of the structuring element at all the the pixels of the provided binary image. The new binary image consists of non-zero values for pixels of the original image that passed the test and zero values otherwise.

Dilation is a morphological operation in which the structuring element is tested on all pixels of the original binary image my placing the anchor point of the structuring element on every pixel of the image and replacing the pixel value with the maximum pixel value from the pixels overlapped by the structuring element. It maximizes the white regions in a binary image.

Erosion is a morphological operation in which the structuring element is tested on all pixels of the original binary image by placing the anchor point of the structuring element on every pixel of the image and replacing the pixel value with the minimum pixel value from the pixels overlapped by the structuring element. It maximizes the black regions in a binary image.

In my case, dilation is performed on the thresolded difference image so that the white parts of the image that belong near to each other and probably part of the same vehicle are connected and then erosion is performed so that white pixels that originated due to noise are removed.

3.5 Contour detection

A contour is a the boundary of an object in an image. It is basically a list of points that represent a curve in an image. It contains a vector of points which consists of all the image points that depict the curve. In OpenCV the contours are found with the help of a function

```
void findContours(InputOutputArray image,
OutputArrayofArrays contours, OutputArray hierarchy, int mode,
int method, Point offset=Point())
```

The parameter image is the input image that we want to find contours of. The contours parameter is a two dimensional vector of points that contains the contour points. Every curve is represented as a vector of points and the parameter contours contains a vector of these vector of points representing all the contours in the image.

In our case, the contours of the the foreground segmented image are obtained through the **findContours** method and the obtained contour points are stored in a vector of vector of `Point`.

3.5.1 Polygon Approximation

Polygon approximation is an approximation of a polygon around a contour with fewer vertices than the contour.

Finding the convex hull of a contour is a type of polygon approximation in which a convex hull of a set of points (say X) is the smallest convex set of points among the set of points X that contains all the points in the set of points X.

In my case, I find the convex hull of each contour that I obtained from the previous step using OpenCV's inbuilt function `void` **`convexHull`**`(InputArray` **`points`**`, OutputArray` **`hull`**`, bool` **`clockwise`**`=false, bool` **`returnPoints`**`=true )`

3.6 `Blob` Class

The convex hulls obtained from the contours are stored in separate class called `Blob`. The constructor of the blob class takes the convex hull points obtained from the contours points. The `Blob` class has the following attributes:

- `vector<Point>` **`contour`** : vector of Point of the convex hull of the blob.
- `Rect` **`Bounding_Rectangle`** : Bounding Rectangle of the convex hull of the blob.
- `double` **`area`** : Area of the bounding rectangle.
- `double` **`width`** : Width of the bounding rectangle.
- `double` **`height`** : Height of the bounding rectangle.
- `double` **`diagonalSize`** : Diagonal size of the bounding rectangle.
- `double` **`aspectRatio`** : Aspect ratio of the bounding rectangle.
- `double` **`flowDistance`** : Length of the optical flows.
- `double` **`angleOfMotion`**: Angle of the average optical flow vector.
- `Point` **`center2D`** : 2D center point of the blob with respect to the image coordinates.
- `Point` **`topRightCorner`** : Top right corner point of the bounding rectangle.
- `Point` **`bottomRightCorner`** : Bottom right corner point of the bounding rectangle.
- `Mat` **`mask`** : Binary mask obtained from the convex hull points.
- `vector<Point2f>` **`featurePoints`** : Vector of points to store the feature points.
- `vector<Point2f>` **`flowPoints`** : Vector of points to store the optical flow head points.
- `Point3f` **`center3D`** : 3D center point of the cuboid with respect to the world coordinates.

- `vector<Point3f>` **`vertices3D`**: Vector of Point3f containing the eight vertices of the cuboid.
- `double` **`initialCuboidLength`** : Initial guess of the cuboid's length.
- `double` **`initialCuboidWidth`** : Initial guess of the cuboid's width.
- `double` **`initialCuboidHeight`** : Initial guess of the cuboid's height.

The `Blob` class has the following methods:

- `void` **`findFeatures`**`(Mat` **`currentFrame)`**
  This method takes in a Mat object which is the input gray scale image, creates a mask on that image from the contour points of the blob and the finds the good features of the image with the applied mask. This is achieved through the OpenCV's inbuilt method `void` **`goodFeaturesToTrack`**`(InputArray` **`image`**`, OutputArray` **`corners`**`, int` **`maxCorners`**`, double` **`qualityLevel`**`, double` **`minDistance`**`, InputArray` **`mask`**`=noArray(), int` **`blockSize`**`=3, bool` **`useHarrisDetector`**`=false, double` **`k`**`=0.04 )`. The method **`goodFeaturesToTrack`** takes the image that we need to find features of as input image and outputs array of corner points that is a vector of `Point2f` which stores all the floating point image points of the features it found. After that it uses OpenCV's built in function `void` **`cornerSubPix`**`(InputArray` **`image`**`, InputOutputArray` **`corners`**`, Size` **`winSize`**`, Size` **`zeroZone`**`, TermCriteria` **`criteria`**`)` to refine the corner points found to the subpixel level. The feature points returned from the two methods are stored.
- `void` **`findFlow`**`(Mat` **`currentFrame`**`, Mat` **`nextFrame`**`)`
  This method takes two gray scale images which are supposed to be two consecutive frames of a video. It uses `void` **`calcOpticalFlowPyrLK`** `(InputArray` **`prevImg`**`, InputArray` **`nextImg`**`, InputArray` **`prevPts`**`, InputOutputArray` **`nextPts`**`, OutputArray` **`status`**`, OutputArray` **`err`**`, Size` **`winSize`**`=Size(21,21), int` **`maxLevel`**`=3, TermCriteria` **`criteria`** `int` **`flags`**`=0, double` **`minEigThreshold`**`=1e-4 )` method of OpencCV to find the optical flow points of the feature points obtained from the findFeatures method.
- `void` **`getCenter3D`**`(Mat` **`inverseHomography`**`)`
  This method finds the ground plane 3D point of the center of the bounding box using the inverse homography matrix.
- `void` **`getFlowheads3D`**`(Mat` **`inverseHomography`**`)` This method finds the ground plane 3D points of the of the flow heads using the inverse homography matrix.
- `void` **`setInitialCuboidParams`**`(double` **`initialCuboidLength`**`, double` **`initialCuboidWidth`**`, double` **`initialCuboidHeight`**`)`
  This method initializes the **`initialCuboidLength`**, **`initialCuboidWidth`** and **`initialCuboidHeight`** variables.
- `void` **`getInitialCuboidVertices3D`**`(Mat` **`inverseHomography`**`)`
  This method finds the eight vertices from the cuboid using the inverse homography matrix and the initial cuboid parameters.

3.7 Track class

The blobs obtained from every consecutive frame are stored as individual Blob objects. Blobs in each frame are matched with the blobs in the previous frame using a matching algorithm. When a blob is matched with another blob from an earlier frame, the collection of these matched blobs can be stored in a vector of blobs. This vector of blobs can be stored as track object so that further operations on these matched blobs get easier. The track object belongs to the track class which has the following attributes:

- `vector<Blob>` **`trackBlobs`** : A vector of blobs that are part of the track.
- `Scalar` **`trackColor`** : A Scalar variable which takes arbitrary R, G, B values to give each track a unique color.
- `bool` **`trackUpdated`** : A Boolean variable that stores `true` if the track is updated.
- `bool` **`beingTracked`** : A Boolean variable that stores `true` if the track is being tracked.
- `int` **`matchCount`** : An integer variable which stores the number of consecutive frames in which a matching blobs was found.
- `int` **`noMatchCounter`** : An integer variable which stores the number of consecutive frame passed without finding a matching blob.
- `int` **`trackNumber`** : An integer variable which stores the track number.

The constructor of the class does not take any parameter. I initialize the `tracknumber` to 0, the integer variable to store the number of consecutive frames without match to 0, the boolean value if the track is being tracked or not to true, the boolean value if the track is updated or not to false.

The track class has the following methods:

- `void` **`addToTrack`**`(Blob` **`blob`**`)`
  This method takes a blob object as a parameter and pushes it back to the vector of blobs of the track class.
- `void` **`drawRects`**`(Mat` **`outputFrame`**`)`
  This method takes a mat image as a parameter and draws the bounding rectangle of the last blob in the vector of blobs stored in the track class. The color of the bounding rectangles is the random color generated when track object is declared.
- `void` **`drawTrackFeatures`**`(Mat` **`outputFrame`**`)`
  This method draws all the feature points of the last blob in the vector of blobs.
- `void` **`drawTrack`**`(Mat` **`outputFrame`**`)`
  This method draws the center of the last blob in the vector of blobs in the track and also draws a line joining the center of the last 10 blobs to see the motion of the track.
- `void` **`drawTrackInfo`**`(Mat` **`outputFrame`**`)`
  This method draws the track number on top of the bounding rectangle.

- `void` **`drawInitialCuboid`**`(Mat` **`outputFrame`**`, Mat` **`CameraMatrix`**`, Mat` **`distCoeffs`**`, Mat` **`rvec`**`, Mat` **`tvec`**`)`
  This method draws the inital cuboid with the initial guess for length, width and height.

3.8 Matching Algorithm

The convex hulls obtained from the foreground blobs are used to initialize the blobs. The blobs obtained from the convex hull area checked for minimum area of the bounding rectangle, the minimum width of the bounding rectangle, the minimum height of the bounding rectangle, the minimum diagonal size of the bounding rectangle and the minimum ratio of the contour convexHull area with the bounding rectangle area. If the blobs are above the minimum requirements then the blob is pushed back to a vector of blobs (say, $currentFrameBlobs$). Now, if it is the first frame of the video then for all the blobs, the `findFeatures` method is called which find features of the blobs and stores it in the `featurePoints` attribute of the blobs class then a track object is created and the blob is add to the vector of blobs in each track object using `addToTrack` method. All the tracks are then pushed back to a vector of tracks (say, $Tracks$). Now after the first frame the vector of tracks contain as many track objects as there were blobs found from the first frame. After the first frame, the blobs obtained from every frame thereafter are push into the method `matchBlobs(vector<Blob>, Mat, Mat)`. The method `matchBlobs(vector<Blob>` **`&frameBlobs`**`, Mat` **`currentFrame`**`, Mat` **`nextFrame`**`)` takes the vector of blobs found from each frame and the current frame and the next frame. The method loops through all the tracks store in the vector of tracks and calls the `findFlow` function in the last blob of every track. The function finds the flow points of every blob from the feature points of the blobs and stores it in the `flowPoints` attribute of the blobs class. Also the boolean attribute that stores if the track is updated or not is set to false for all the tracks. Then the loop is ended.

Then for all the blobs stored in the vector of blobs of the current frame, we loop through all the tracks stored in the vector of tracks and find the summation of distances between the flow heads of the last blobs of the track from the center of the current frame blob and get the average by dividing the sum by the number of flow heads. As we loop through all the tracks and find the average distance from the center of the current frame blob to the flow heads of the last blob of the track, we store the index of the track when the average distance is the least. If the least distance is less than a threshold, we assume that we have found a matching track for the current frame blob. If we do not find a matching track for the current frame blob, we make a new track object and add the current frame blob to the track.

Upon finding a matching track for the current frame blob, we add the blob to the track. We update the Boolean that stores whether the track is being updated to be true. We increment the variable that stores the number of consecutive frames that a match has been found by 1. We update the the variable that store the number of consecutive frame without a match to 0. If the number of consecutive frames that a match has been found is greater than 10. We increment the track count variable by 1 and update the track number of the track to the track count.

Algorithm 1 Matching Algorithm

```
for all tracks do
  findFlow() to find the flow heads from the feature points of each blob in track.
  trackUpdated $\leftarrow$ true
end for
for all current frame blobs do
  double leastDistance $\leftarrow$ some arbitrary large number
  Point center $\leftarrow$ center of the bounding box of the blob
  for all tracks do
    double match $\leftarrow$ true
    double sumDistances $\leftarrow$ 0
    for all flow heads of last blob in track do
      distance $\leftarrow$ distance between flow head and center of current frame blob
      sumDistances $\leftarrow$ sumdistances + distance
    end for
    double averageDistance $\leftarrow$ sumDistances / number of flow heads
    if averageDistance $<$ leastDistance then
      leastDistance $\leftarrow$ averageDistance
      index of least distance $\leftarrow$ number of iteration through tracks
    end if
  end for
  if leastDistance $<$ diagonal length of current frame blob $\times C$ (where $C$ is a constant)
  then
    findFeatures to find feature points of the current frame blob
    add current frame blob to the vector of blobs of the matched track
    trackUpdated $\leftarrow$ true (for the matched track)
    matchCount $\leftarrow$ matchCount + 1 (for the matched track)
    noMatchCounter $\leftarrow$ 0 (for the matched track)
    if matchCount == 10 (for the matched track) then
      trackCount $\leftarrow$ trackCount + 1
    end if
  else
    findFeatures (to find features of current frame blob)
    create new Track
    add blob to the new Track
  end if
end for
```

3.9 Image to ground plane transformation

Camera calibration is used to define a relationship between homogenous 3D points $X = (X, Y, Z, 1)^T$ and homogenous 2D points $x = (x, y, 1)$ as

$$x \propto PX$$

The above equation describes the relationship between a point in the image plane and a point in the real world. P is a $3X4$ projection matrix that can be described as $P = K[R|T]$ where

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

is the matrix of intrinsic camera parameters, R is the rotation of the camera with respect to the ground plane, and $T = -RC$ is a translation vector, with C being the optical center of the camera in the world coordinate system. If we assume the ground to be flat, resulting in $z = 0$.

The projection equation becomes

$$\begin{bmatrix} Wx \\ Wy \\ W \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} p_{11} & p_{12} & p_{14} \\ p_{21} & p_{22} & p_{24} \\ p_{31} & p_{32} & p_{34} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = H \cdot \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

To calculate the ground point corresponding to an image point, we take H's inverse and multiply with the image point.

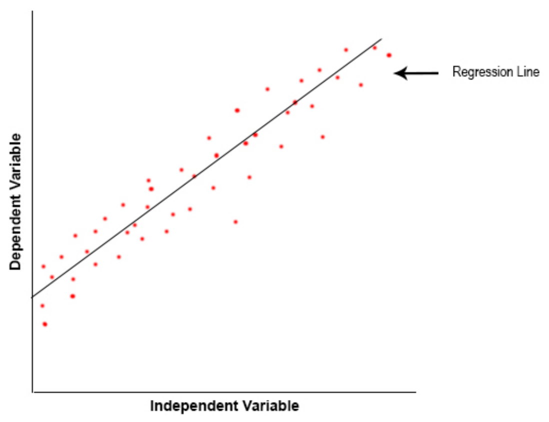
$$H^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} W \cdot X \\ W \cdot Y \\ W \end{bmatrix} \propto \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

3.10 Regression Analysis

Regression analysis is a statistical process of estimating relationships among variables. Regression analysis helps to understand how the dependent variable changes when the independent variable is varied while other independent variable are fixed. The estimation target in regression analysis is the regression function of the independent variables. Regression analysis is used to understand which among the independent variables are related to the dependent variable.

3.10.1 Linear Regression Analysis

Suppose we want to find out if there is any correlation between a variable x and variable y. Here, x is the independent variable while y is the dependent variable. We can get a random sample from the population sample of independent variable x and corresponding y values. This data accumulated can be represented on a scatter graph. The independent variable on the x-axis of the graph and the dependent variable on the y-axis of the graph. We try to formulate a relationship (in this case, a straight line) and analyse as the independent variable is changing, what happens to the dependent variable. If the independent variable and the dependent variable increases as well, there is a positive relationship between the two. On the other hand if the independent variable increases and dependent variable decreases, there is a negative relationship. Linear regression analysis involves finding a straight line that best fits all the different values of dependent variable with respect to different values of the independent variable. This line is called the regression line which is based on the least squares method.



The sample regression line is represented as

$$\hat{Y} = \hat{b_0} + \hat{b_1}X + \hat{\mu}$$

where $\hat{b_0}$ is the Y intercept, $\hat{b_1}$ is the slope of the regression line and $\hat{\mu}$ is the residual or the error term.

The slope $\hat{b_1}$ describes the relationship between the dependent and the independent term. If the slope is positive, it means the dependent variable increases with the increase in the value of the independent variable. If the slope is negative, it means the dependent variable decreases with the increase in the value of the independent variable. The objective is to minimize sum of these errors and make them as small as possible.

If there is statistically significant relationship, there is a true casual relationship between X and Y that is not due to sampling variation or error. Since the data collected is usually a sample of the population, the regression line is a sample regression line. The best fit for a sample regression line will have the value of the slope and the intercept close to that of the population regression line.

3.10.1.1 Linear model estimation through least-square minimization

Let's say we have some values for the independent variable and values of their independent variable.

Lets assume the mean of the X values is $\bar{X}$ and the mean of the Y values is $\bar{Y}$. Then the regression line has to go through the point $(\bar{X}, \bar{Y})$. The distances between the mean $\bar{X}$ of the X values and the X values are $X_1 - \bar{X}$, $X_2 - \bar{X}$, $X_3 - \bar{X}$, $X_3 - \bar{X}$, $X_4 - \bar{X}$ and $X_5 - \bar{X}$ and similarly the distances between the mean $\bar{Y}$ of the Y values and the Y values are $Y_1 - \bar{Y}$, $Y_2 - \bar{Y}$, $Y_3 - \bar{Y}$, $Y_3 - \bar{Y}$, $Y_4 - \bar{Y}$ and $Y_5 - \bar{Y}$. Then the slope of sample regression line is given by

$$\frac{\sum_{i=0}^{N}(X_i - \bar{X})^2}{\sum_{i=0}^{N}(X_i - \bar{X})(Y_i - \bar{Y})}$$

3.10.2 Non-linear Regression Analysis

A dependent variable can depend on an independent variable in which observational data are modeled by a function which is nonlinear combination of the parameters. The regression line in this case is a curve instead of a straight line. A regression model is non-linear if the derivative of the model depends on one or more parameters.

3.10.2.1 Non-linear model estimation

Non-linear model estimation is an iterative process. The following are the steps for non-linear model estimation:

- The starting values are provided by the users.
- The algorithm then adjusts the values of the parameter successively in order to improve the model.
- If the algorithm achieves maximum improvement and further improvement is not possible, then fit is considered converged.
- The first step is to define the error associated with each point i. Lets assume there are four parameters A, B, C, and D. Then the error associated is given by

$$\mathscr{E}_i = y_i - f(x_i, A, B, C, D)$$

- The next step is to define the function as a sum of the squares of errors

$$\sum_{i=1}^{N} \mathscr{E}_i^2 = \sum_{i=1}^{N}[y_i - f(x_i, A, B, C, D)]^2$$

- The next step is to take the partial derivative of the function with respect to each of the parameters

$$\frac{\partial F}{\partial A} = 2\sum_{i=1}^{N} \mathscr{E}_i \frac{\partial \mathscr{E}}{\partial A}$$

$$\frac{\partial F}{\partial B} = 2\sum_{i=1}^{N} \mathscr{E}_i \frac{\partial \mathscr{E}}{\partial B}$$

$$\frac{\partial F}{\partial C} = 2\sum_{i=1}^{N} \mathscr{E}_i \frac{\partial \mathscr{E}}{\partial C}$$

$$\frac{\partial F}{\partial D} = 2\sum_{i=1}^{N} \mathscr{E}_i \frac{\partial \mathscr{E}}{\partial D}$$

- If the parameters are A, B, C, D, then there are four equations with four unknowns. The four equations are

$$F(y) = \sum_{i=1}^{N} \begin{bmatrix} \mathscr{E}_i \frac{\partial \mathscr{E}_i}{\partial A} \\ \mathscr{E}_i \frac{\partial \mathscr{E}_i}{\partial B} \\ \mathscr{E}_i \frac{\partial \mathscr{E}_i}{\partial C} \\ \mathscr{E}_i \frac{\partial \mathscr{E}_i}{\partial D} \end{bmatrix}$$

- To optimize or minimize f, a vector Y is needed to be found such that $F(Y) = 0$. This is just like solving a set of four non-linear equations with four unknowns.

3.10.3 Levenberg-Marquardt Algorithm

The **Levenberg-Marquardt Algorithm (LMA)** which is also known as the damped least squares method is used to solve nonlinear least squares problems which involves fitting a parameterized function to a set of measured data points by minimizing the sum of the squares of the errors between the data points and the function. It iteratively reduces the sum of the squares of the errors between the function and the measured data points by sequentially updating the parameter values. It is a combination of both the gradient descent method and the Gauss Newton method. The Levenberg-Marquardt algorithm acts like the Gauss Newton method when the parameters are close to the optimal values and acts like the gradient descent method when the parameters are far from the optimal value.

Given a set of m empirical datum pairs (x_i, y_i) of independent and dependent variables, Levenberg-Marquardt finds the parameters β of the curve $f(x, \beta)$ such that the sum of squares of the deviations is minimum.

$$\hat{\beta} = argmin_\beta\ S(\beta) = argmin_\beta \sum_{i=1}^{m}[y_i - f(x_i, \beta)]^2$$

The Levenberg-Marquardt algorithm is an iterative process where an initial guess for the parameter β is provided. If there are multiple minimas. the algorithm will converge to a global minimum only if the initial guess is already close to the final solution

3.11 3D modeling of a cuboid around the vehicle

In this section, I describe how the cuboid around the vehicles to be tracked is modeled.

3.11.1 Initial guess of the cuboid

For the initial guess of the cuboid, we take the center of the bounding box of the background blobs obtained to be one of the vertex of the ground plane of the cuboid. We estimate an initial height of the cuboid to be 2 meters, width to be 2 meters and length to be 3 meters. We take the direction of the cuboid from the average direction of the optical flow associated with the respective blob.

3.11.2 Refinement using Levenberg Marquardt algorithm for least squares minimization

We refine the dimensions of the cuboid with each frame using the error function by trying to minimize the error as much as possible using the Levenberg-Marquardt algorithm.

Given cuboid 1 where the optical flow points are $X_1, X_2, ...X_N$, we move the cuboid in the direction of the average optical flows to a distance equal to the average optical flow lengths. Let cuboid 2 be the observed cuboid in the next frame with optical flow points $X'_1, X'_2, ...X'_N$ Then the error is given by,

$$error = \sum_{i=1}^{N} ||X_i - f(X_i, cuboid_1, cuboid_2)||^2$$

where $f(X_i, cuboid_1, cuboid_2)$ is the observed motion of point X_i from $cuboid_1$ to $cuboid_2$

To do this, I find the point of intersection of the line joining the ground plane coordinates of the optical flow points to the projection of the optical flow points to the camera coordinates with $z = 1$ with every plane of the cuboid and note its positions with respect to the planes. Then I move the cuboid in the direction of optical flow motion taking distance traveled as the average optical flow length associated with the cuboid. Then I compare the position of the optical flow points with respect to the observed optical flow points of the cuboid in the next frame. I take the sum of the absolute difference of the Euclidean distances between the observed flow points and the predicted flow points. Then I use, OpenCV's inbuilt class `CvLevMarq(int nparams, int nerrs, criteria TermCriteria, bool completeSymmFlag = false)`

3.11.3 Real world speed estimation

Points of the cuboids closest to the camera are measured and then the real world Euclidean distance between the points in one frame and the corresponding centroids of the cuboids of the next frame is measured by the formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$ This distance multiplied by the number of frames n per second that gives the vehicle speed v.

3.12 Evaluation Plan

....

Chapter 4

Preliminary Experiments

In this chapter, I demonstrate my preliminary experiments ...

4.1 Experimental Design

...

4.2 Result

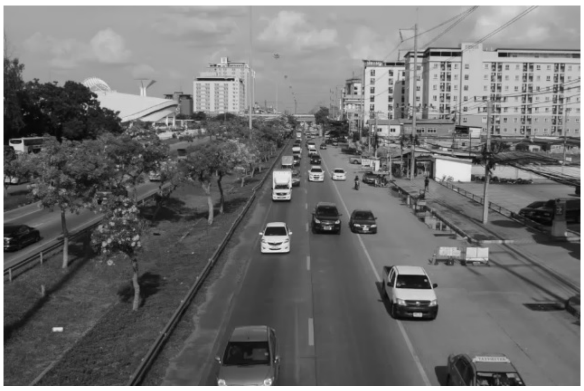


Figure 4.1: Grayscale frame

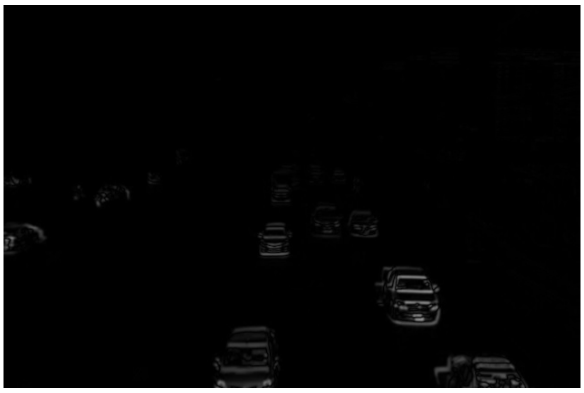


Figure 4.2: Frame after absolute differencing

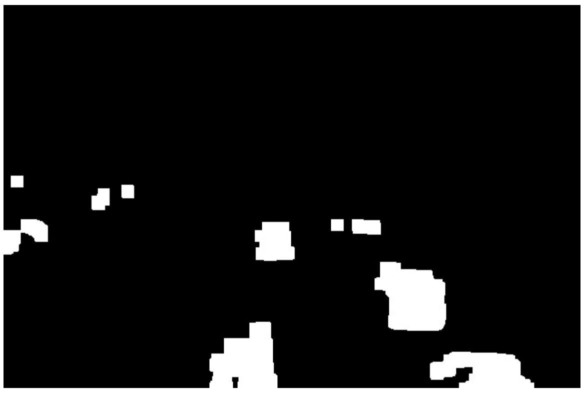


Figure 4.3: Video frame after dilation and erosion

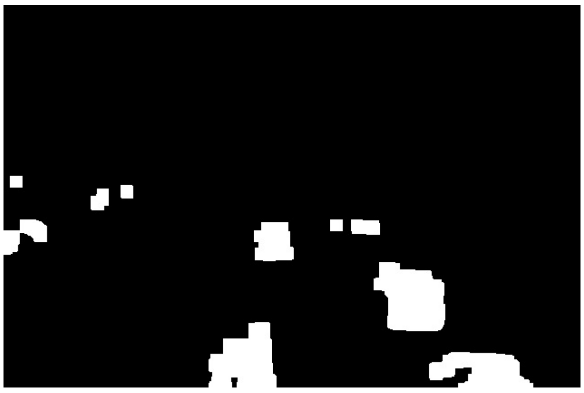


Figure 4.4: Filled contours of the frame

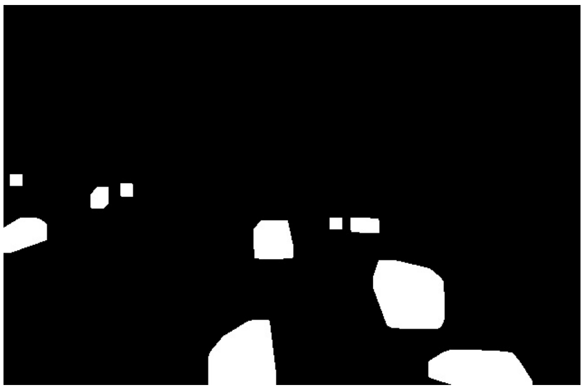


Figure 4.5: Convex hulls of the contours

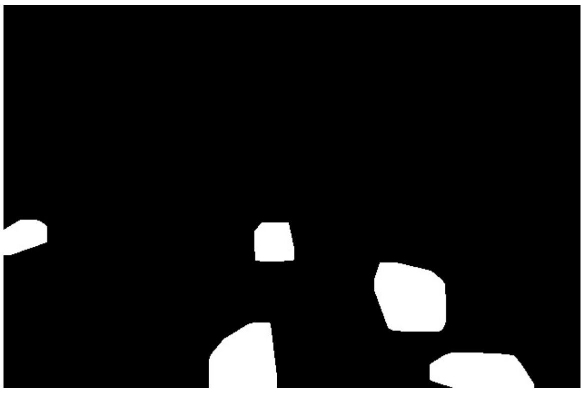


Figure 4.6: Binary image of current frame blobs after filtering non-required blobs

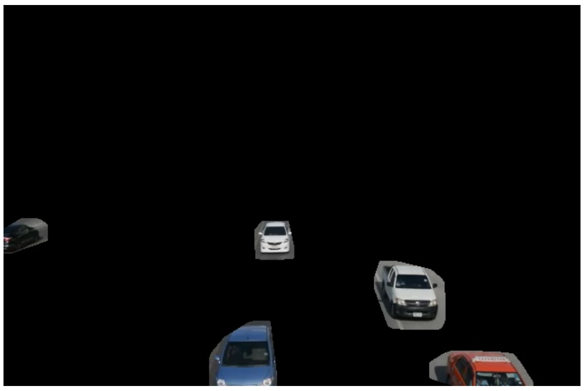


Figure 4.7: Blobs masked on the original frame

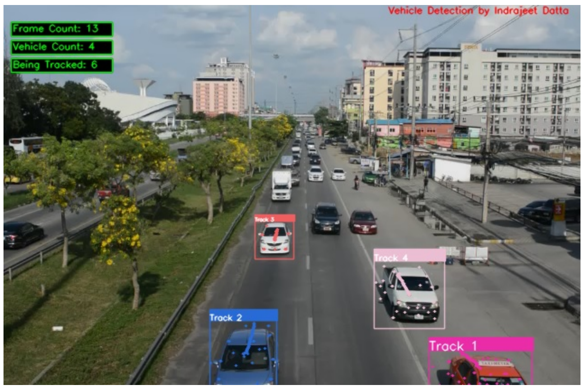


Figure 4.8: Tracked vehicles after implementation of matching algorithm

4.3 Evaluation and Discussion

...

Chapter 5

Work Plan

This chapter describes my proposed planning in order to implement and improve my methodology ...

5.1 Planning

...

5.2 Time Line

Figure **??** shows my proposed time line.

References

Dailey, D., Cathey, F., & Pumrin, S. (2000). An algorithm to estimate mean traffic speed using uncalibrated cameras. *IEEE Transactions on Intelligent Transportation Systems,*, *1*(2), 98-107.

Horn, B. K., & Schunck, B. G. (1981). Determining optical flow. In *1981 technical symposium east* (pp. 319–331).

Iwasaki, Y., & Itoyama, H. (2006). Real-time vehicle detection using information of shadows underneath vehicles. In K. Elleithy, T. Sobh, A. Mahmood, M. Iskander, & M. Karim (Eds.), *Advances in computer, information, and systems sciences, and engineering* (p. 94-98). Springer Netherlands.

Jhumat, S. (2014). Vehicle speed estimation in accident prone areas using image processing. *International Journal of Advanced Research and Communication Engineering*, *3*(5).

Lucas, B. D., Kanade, T., et al. (1981). An iterative image registration technique with an application to stereo vision. In *International joint conference on artificial intelligence* (Vol. 81, pp. 674–679).

Wang, H., & Zhang, H. (2014). A hybrid method of vehicle detection based on computer vision for intelligent transportation system. *International Journal of Multimedia and Ubiqitous Engineering*, *9*(6), 105-118.

Yung, N., & Lai, A. (1998, May). Detection of vehicle occlusion using a generalized deformable model. In *Proceedings of the 1998 ieee international symposium on circuits and systems. (iscas)* (Vol. 4, p. 154-157 vol.4).

Zhou, J., Gao, D., & Zhang, D. (2007, Jan). Moving vehicle detection for automatic traffic monitoring. *IEEE Transactions on Vehicular Technology*, *56*(1), 51-59.