A Computer Vision Based Approach to Traffic Flow Analysis

Craig Hiller

Abstract

For my project, I developed a system and method to analyze image data from highway traffic cameras to determine the rate at which traffic is moving. My approach consisted of two main components: camera calibration and speed calculations from the retrieved images.

Camera calibration is required to compensate for the image perspective. To do this, I retrieved a set of images from the traffic camera. I then calculated the number of feet that each pixel represents using an optical flow algorithm based on block matching. A calibration table containing the feet-per-pixel calculation as a function of position in the X and Y directions was generated.

After I calibrated the camera, I proceeded to use images in real time to calculate the average speed on the road. Using the same optical flow algorithm, I calculated the distance vector of each block of pixels between successive images. By applying the calibration table adjustments, I was able to determine the speed of the pixel block. Averaging the speeds along the road yielded the average speed of traffic on the road.

Using this system, a real-time web based traffic map similar to that of Google Traffic could be generated. The system could also send messages through email or text to alert users that traffic on their route is slow.

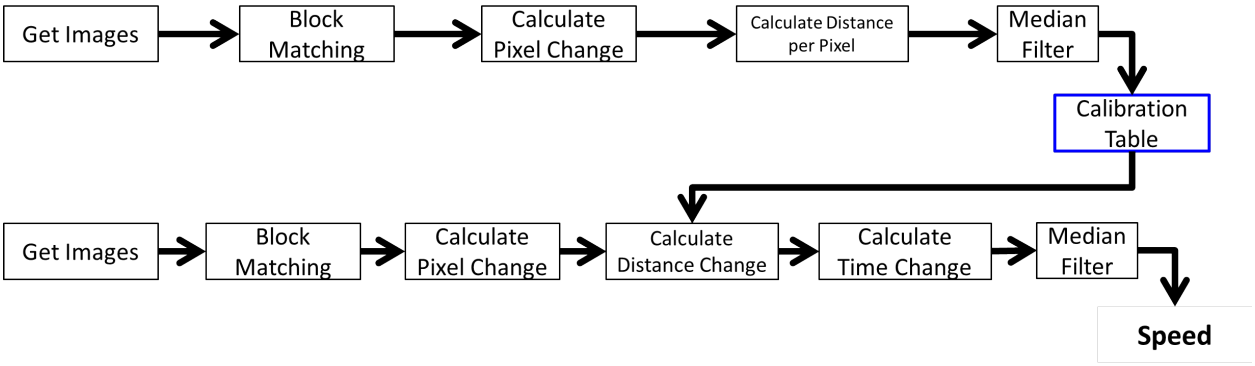
Introduction

In today's speed crazed society, knowing where traffic is and how fast it is going has become crucial information. Probably the most well-known traffic data source is the traffic data provided on Google Maps. However, this data is not always accurate and can at times be the exact opposite of reality. For example, the Google map may show a road as being green indicating fast moving or little traffic when in reality the traffic is essentially stopped. Further, traffic cameras are available in many areas that are currently not covered by Google's traffic data, which allows for increased coverage.

This project was designed to broaden the availability of traffic data and to provide it in real-time to an end user. Researchers in Lithuania and Norway have done similar work that locates the individual cars and tracks them; their work also detects and reads license plate numbers. Their work requires test drives with known speeds in order to calibrate the system. [1]

Experiment

Diagram



Setup

Before I could start to develop software for this project, I needed to setup my development environment. This included installing the Python programming language (version 2.6.7), the Python Imaging Library (PIL) and the Open Computer Vision (OpenCV) Library.

After these components were installed and working, I was able to begin data collection and experimentation.

Data Collection

The main form of data collection in this project was retrieving images from traffic cameras. The cameras that I utilized for experimentation and development of my system had web pages with a JPEG image that updated fairly often, fewer than two seconds between images.

To fetch the images, I wrote a Python program that would retrieve the image URL, and

Figure 1: Image from the Whitestone at 14th Ave traffic camera[2]

if the image had changed since the previous retrieval, it would be saved to the computer with its filename being the time it was retrieved. This program was then run as a cron job on a Linux machine to collect a burst of 50 images every 10 minutes, between 30 seconds and 2 minutes worth of data per burst. I did not want to overload the servers with my requests, so I limited the number of image retrievals. Even so, this yielded a very large set of data ($> 100,000$) images for experimentation.

Calibration

Due to the inherent perspective nature of the images (Figure 1) retrieved, the corresponding size of a pixel in the real world is not a constant. In order for speed to be calculated in a useful measurement (miles / hour) instead of pixels/second, the conversion factor between pixel and distance as a function of x and y coordinates in the picture must be calculated through the process of calibration. Not knowing the camera's height, angle, field of view, and other variables increased the difficulty of this process.

Initial Approach

To attempt to solve the problem of calibration, I first tried to use geometry and known distances to derive a multi variable function of pixel position that returned the real-world distance equivalent of that position. This involved measuring the location of a vehicle of known size (a standard semi-trailer of 48 feet in length) in multiple image frames and tracking its change in pixel location. I then constructed a diagram (Figure 2) representing the situation and attempted to solve for the distance that the truck traveled between measurements. This yielded six law of cosine equations with six unknowns. After attempting

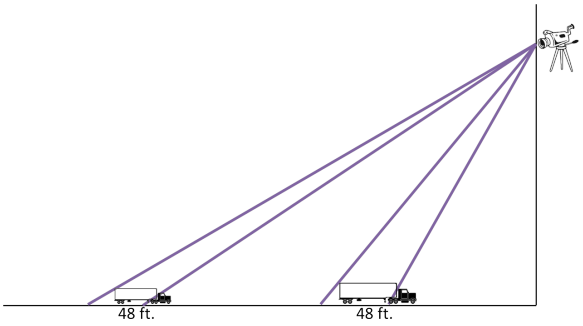


Figure 2: Geometry setup for the initial approach to calibration

to solve this system by hand, I entered my system of equations into Mathematica, which given three hours did not solve it. Even if it were to be solvable, it would have required the operator to measure accurately the pixel distances, which is not an easy task. With frequent camera angle and zoom changes, it was evident that this method was impractical.

Final Approach

After not being able to solve the system of equations, I began to think of other means of calibration. Initially, I attempted to derive a continuous function for the conversion factor between pixels and distance, but I realized that a discrete function, a table, would work just as well, if not better, and be simpler and more accurate to compute.

I began reading about the OpenCV library and its features in the book *Learning OpenCV* [3], and I determined that calibration would best be done through the help of an optical flow calculation algorithm. I chose a block-matching optical flow algorithm because it was the simplest to understand conceptually. For a given set of pixels (block) in a first image, the algorithm searches in a second image for a similar block and returns the change in both the x and y directions.

The underlying assumption is that the speed of the traffic during calibration is either the speed limit (or 100%) when upon visual inspection the traffic camera feed shows normal traffic conditions. This assumption can be circumvented by having a known speed measurement for the interval in which the calibration images are captured. Over time, the calibration table can be dynamically updated when a 100% speed is calculated. This enhancement has the possibility to eliminate all calibration but would be the subject of future work.

Because the speed of the traffic, S, change in time, Δt, from the difference in timestamps of

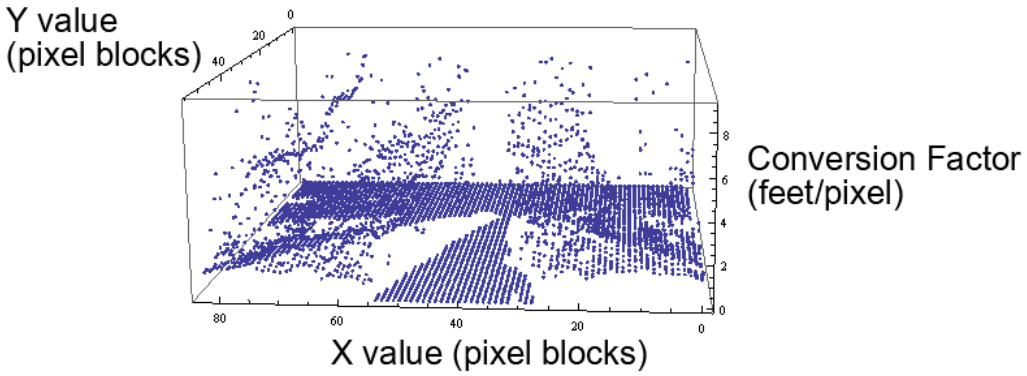


Figure 3: Plot of x and y position and conversion factor from calibration process

the two images, and change in pixels in the x and y directions is known for each block, the conversion factor between pixel and distance can be calculated as follows:

$$\Delta \text{ Distance} = \Delta d = S\Delta t$$

$$\Delta \text{ Pixels} = \Delta p = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

$$\text{Conversion Factor} = \frac{\text{distance}}{\text{pixel}} = \frac{\Delta d}{\Delta p}$$

These conversion factors are then generated for each block over at least 100 images. To eliminate variation caused by wind moving the camera and trees along the roadside, JPEG compression artifacts, and other factors, I take the median of the calculated values for each block across all of the calibration images.

The result of this process is a two-dimensional table which is indexed by x and y values and returns the pixel–distance conversion factor. Plotting the conversion factors with respect to x and y (Figure 3) shows that the conversion factors do not change linearly across the image and the function that would have described this relationship would have been quite complicated.

Speed Calculations

The calculations for speed are done in much the same way as the calculations for calibration, only in reverse. Pairs of images are processed with the optical flow algorithm, which returns the change of position for each block. As the result of calibration, for each block, I have a

Figure 4: Results from New York camera [2]

conversion factor between the change in pixels and physical distance.

$$\frac{\text{Distance}}{\text{Pixel}} \times \text{Pixels} = \text{Distance}$$

However, traffic might not always travel in the same direction across all roads in an image, so different directions must be differentiated from one another. I separated the traffic directions by using the sign of Δy. If it were positive, then traffic would be traveling towards the bottom of the frame, and if negative, then the traffic would be traveling towards the top of the frame. Each speed calculation, done across five or more images, was placed into the corresponding array, Δy positive or negative, and the median speed from each array was calculated, again to avoid any anomalous outlying points.

Results

I ran my program on a variety of images representing different situations as summarized in Figure 4. Images are from the New York camera [2]. However, this was only on one camera. To test that my programs worked on cameras other than just this one in New York, I chose a

Figure 5: Image from Pennsylvania camera [4]

camera near Valley Forge, Pennsylvania. An image [4] with speed calculations on it is shown in Figure 5. Using Python and PIL, I generated a more visually appealing representation of the data using a pseudo traffic light to represent the traffic speeds (Figure 6). Seventy–five percent or higher is represented as green, between 50% and 75% is yellow, and less than 50% is represented by a red light.

Verification

So far, all speeds shown have been calculated but not verified in the real world. Because it was unfeasible to measure the speeds of the traffic, capture a set of images, have the cars slow down and repeat the process I built a scale model of this system.

I needed a vehicle that would travel at a constant but controllable speed. To do this, I built a small "car" out of LEGO Mindstorms that I programmed using a programming language called *Not Quite C*. This allowed me to control the motor functions and, consequently, the speed of the car.

With the help of a remote timing device on a digital camera, I took 150 images at one second intervals as the car traveled across the floor. I ran my calibration program on this set of images to create the table for 100% speed, then captured more images for testing, at both 100% and 50% speed.

At 100%, my speed calculation program calculated a speed of 98.8% and at 50% speed, the calculated value was 50.7% (Figure 7). This led me to conclude that my results from

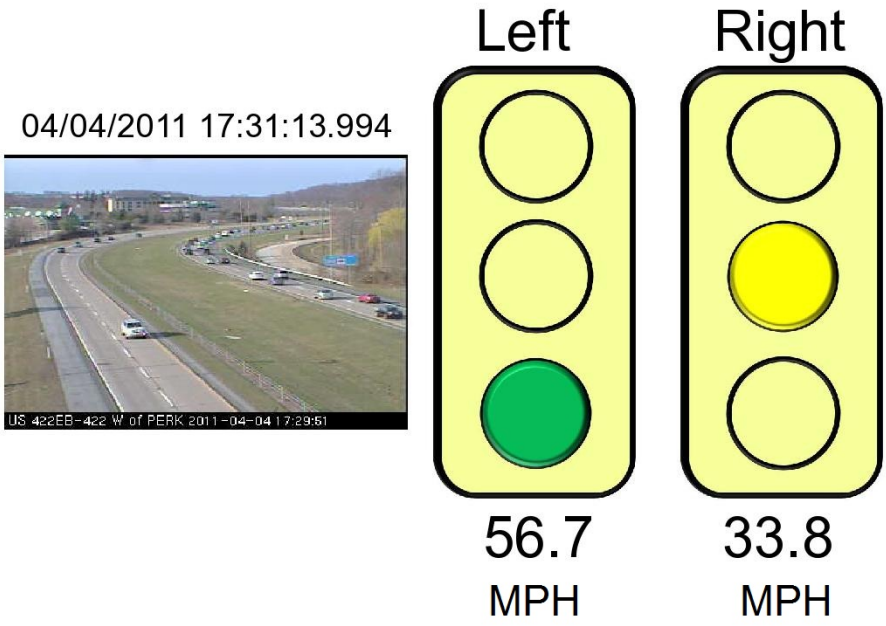


Figure 6: Simple graphical representation of traffic conditions

the real-world traffic were accurate with minimal error.

Problems

Besides the initial problem with calibration, other problems included image 'jitter' and computational efficiency.

Image Jitter

When I expanded my data set to include a camera in Pennsylvania, I came across a problem that I call jitter. As was done in the initial image collection, I downloaded the images. However, this time, I noticed that many frames were repeated similar to those shown in Figure 8. The ABABCD image sequence shown is an example of what is retrieved and below is the UNIX time stamp for when I retrieved the images. When these images were used in the speed calculations, speeds about 10 times faster than expected were calculated. In order to use this traffic camera (and others hosted by the TrafficLand service), I needed to remove the images affected by jitter. Fortunately, these images were retrieved very close

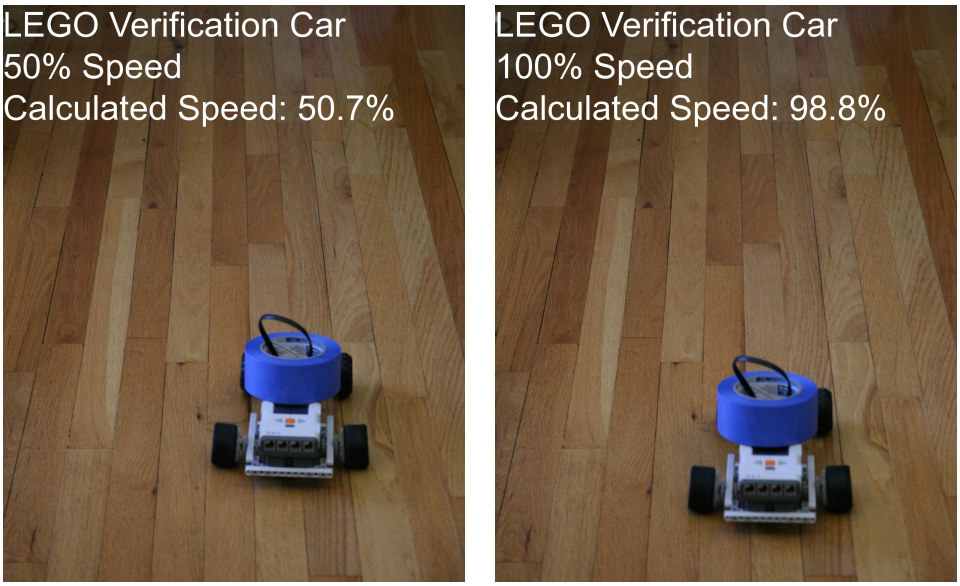


Figure 7: LEGO Verification Car with actual and calculated speeds as percentages of normal

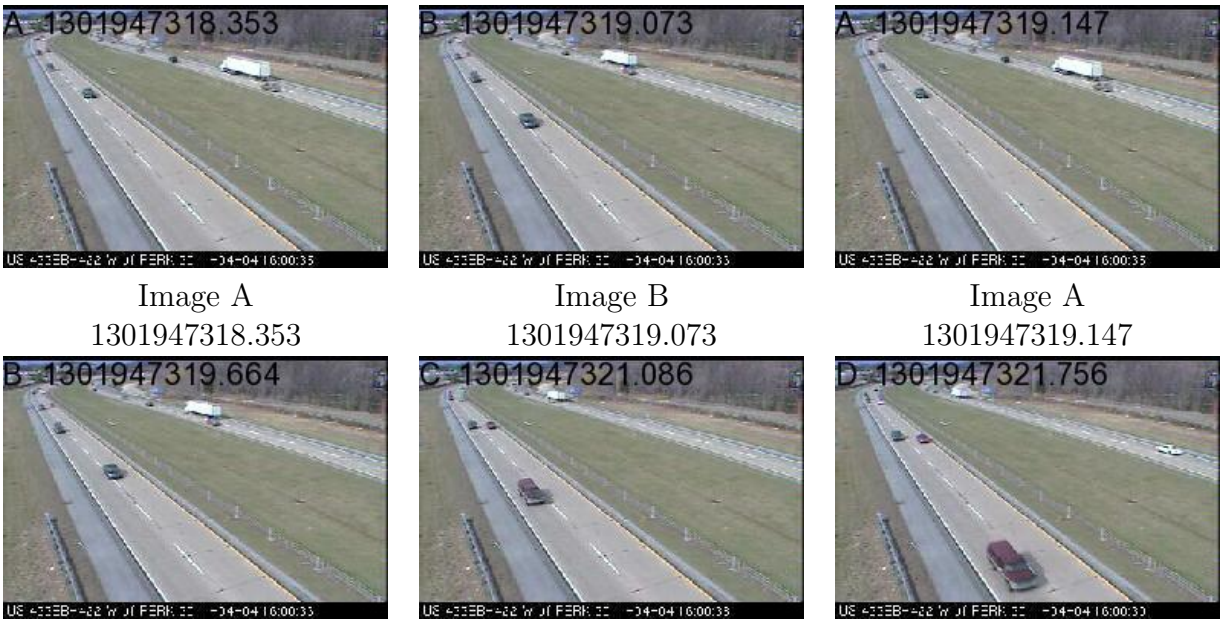


Image A
1301947318.353

Image B
1301947319.073

Image A
1301947319.147

Image B
1301947319.664

Image C
1301947321.086

Image D
1301947321.756

Figure 8: Sequential images experiencing jitter. Timestamps are below the images.[4]

to each other (around a tenth of a second between images). By removing images that were captured within half a second or less from each other, jitter was removed and speed calculations worked correctly again.

Efficiency

The goal of this project was to provide speed data in real time. With all of the code written in Python, an interpreted language, it took approximately 90 seconds to calculate the speed across five images using a 2.4 GHz Core 2 Duo processor. Although this updates more frequently than other traffic products, it is not real time. To improve this, the code for this project could be rewritten in the C++ programming language. Further improvements upon this include using multi threading and parallel processing. The optical flow algorithm could also be rewritten for parallel processing or for a graphics processing unit (GPU). This could increase the algorithms efficiency by two or more orders of magnitude as some GPUs have over 1000 parallel processing cores.

Conclusion

I successfully built a system for automatically calculating the average speed of cars on a road using traffic camera images. After initially attempting to use geometry to convert between distance and pixels, I moved to computer vision. This allowed me not only to solve the calibration problem but also to calculate speed. Using an optical flow algorithm to calculate the change in pixels and using a known percentage or normal speed or the speed limit, I was able to calculate a distance per pixel value, which I then averaged across 150 images.

Using the same optical flow algorithm and the value calculated from calibration, I was then able to determine the distance traveled between frames and convert that to speed because I knew the elapsed time between the images. Two directions of traffic can be differentiated by observing the change in the y direction, and the speeds can then be displayed graphically.

By building a LEGO robotic car, I was able to that verify my calculations and methodology were sound and concluded that the system I created works.

Future Development

In terms of the program itself, it could be further developed by adding in auto-calibration and increasing the speed with which results are calculated. Auto-calibration would be useful to automate fully the system and to introduce new cameras into it without any human interaction. Performance could be increased by optimizing the optical flow algorithm for different hardware, by running the programs on a faster machine, or by optimizing the code to utilize multiple threads and cores.

From a user interface point of view, a website or mobile phone application could be designed that showed a map and a visual representation of the speed calculation with a colored bar or line and the image feed from the traffic camera.

Bibliography

[1] Atkociunas, E., Blake, R., Juozapavicius, A., and Kazimianec, M. (2005). Image Processing in Road Traffic Analysis. *Nonlinear Analysis: Modelling and Control, 10*(4), 315-332.

[2] *Whitestone @ 14 Ave Traffic Camera.* (December 3, 2010 to April 5, 2011). Retrieved from New York City Department of Transportation website: http://207.251.86.248/cctv197.jpg

[3] Bradski, G., and Kaehler, A. (2008). *Learning OpenCV: Computer Vision with the OpenCV Library.* Sebastopol, CA: O'Reilly Media, Inc.

[4] *Traffic Map.* (December 3, 2010 to April 5, 2011). Retrieved from Pennsylvania Department of Transportation website: http://www.511pa.com/Traffic.aspx

[5] *Overview – Python v2.6.7 documentation.* (n.d.). Retrieved Winter, 2010-2011, from http://docs.python.org/release/2.6.7/

[6] *OpenCV 2.1 Python Reference* (n.d.). Retrieved Winter, 2010-2011, from http://opencv.willowgarage.com/documentation/python/index.html

[7] *Python Imaging Library (PIL)* (n.d.). Retrieved Winter, 2010-2011, from http://www.pythonware.com/products/pil/