Counting Cars OpenCV + Python Issue

Asked 8 years, 9 months ago Modified 3 years, 2 months ago Viewed 39k times

48

I have been *trying* to count cars when crossing the line and it works, but the problem is it counts one car many times which is ridiculous because it should only be counted once.

Here is the code I am using:

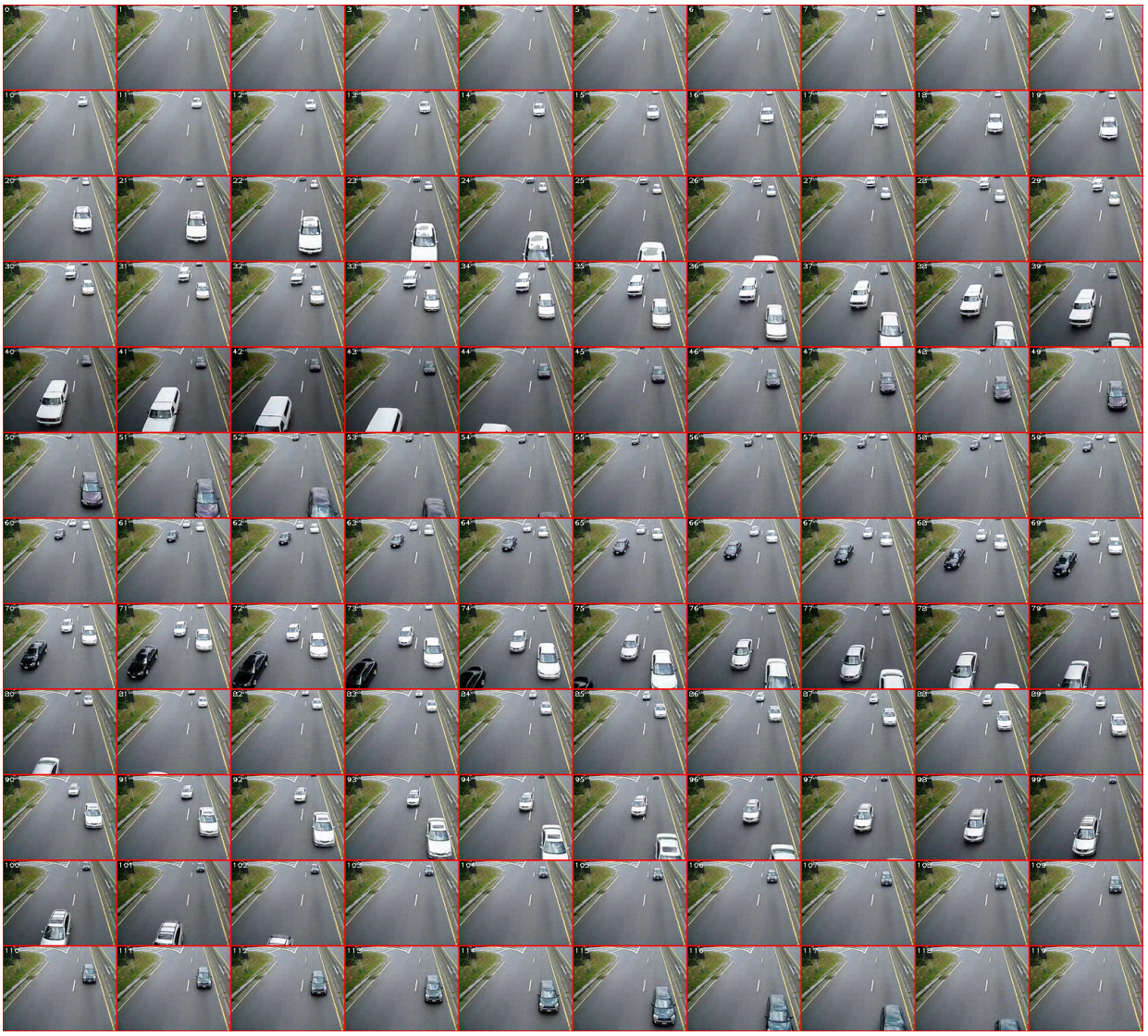
```
import cv2
import numpy as np

bgsMOG = cv2.BackgroundSubtractorMOG()
cap    = cv2.VideoCapture("traffic.avi")
counter = 0

if cap:
    while True:
        ret, frame = cap.read()

        if ret:
            fgmask = bgsMOG.apply(frame, None, 0.01)
            cv2.line(frame, (0,60), (160,60), (255,255,0), 1)
            # To find the countours of the Cars
            contours, hierarchy = cv2.findContours(fgmask,
                                    cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)

            try:
                hierarchy = hierarchy[0]

            except:
                hierarchy = []

            for contour, hier in zip(contours, hierarchy):
                (x, y, w, h) = cv2.boundingRect(contour)

                if w > 20 and h > 20:
                    cv2.rectangle(frame, (x,y), (x+w,y+h), (255, 0, 0), 1)

                    # To find the centroid of the car
                    x1 = w/2
                    y1 = h/2

                    cx = x+x1
                    cy = y+y1
##                    print "cy=", cy
##                    print "cx=", cx
                    centroid = (cx,cy)
##                    print "centoid=", centroid
                    # Draw the circle of Centroid
                    cv2.circle(frame,(int(cx),int(cy)),2,(0,0,255),-1)

                    # To make sure the Car crosses the line
##                    dy = cy-108
```

```
##                     print "dy", dy
                    if centroid > (27, 38) and centroid < (134, 108):
##                         if (cx <= 132)and(cx >= 20):
                        counter +=1
##                             print "counter=", counter
##                     if cy > 10 and cy < 160:
                    cv2.putText(frame, str(counter), (x,y-5),
                                       cv2.FONT_HERSHEY_SIMPLEX,
                                       0.5, (255, 0, 255), 2)
##             cv2.namedWindow('Output',cv2.cv.CV_WINDOW_NORMAL)
            cv2.imshow('Output', frame)
##          cv2.imshow('FGMASK', fgmask)


            key = cv2.waitKey(60)
            if key == 27:
                break

cap.release()
cv2.destroyAllWindows()
```

And the video is on my GitHub page @ https://github.com/Tes3awy/MATLAB-Tutorials/blob/f24b680f2215c1b1bb96c76f5ba81df533552983/traffic.avi (and it's also a built-in video in Matlab library)

How can make it so that each car is only counted once?

The individual frames of the video look as follows:



python numpy opencv image-processing

Share

Improve this question

Follow

edited Oct 11, 2021 at 19:04

Peter Mortensen

31.6k • 22 • 109 • 133

asked Mar 28, 2016 at 0:57

Tes3awy

2,264 • 5 • 30 • 55

I am using OpenCV 2.4.11 and Python 2.7 – Tes3awy Mar 28, 2016 at 1:25

1 I'm well aware of the differences. The point of that example was to give you inspiration on how to refactor and improve your code to make debugging and testing it easier, provide us with insight as to what exactly is happening and why, and in general improve the quality of this question. It's a pity you find that worthless. I was looking forward to dig into this further, had you been willing to help us solve your problem. – Dan Mašek Mar 28, 2016 at 13:14

@Dan Mašek I mean it's worthless right now, I am going to do what you said in the previous when I am finishing my code, I mean in the end all the refactor and improvements will be done. Excuse my English I am an ESL so I don't literally mean worthless, I just mean it's not so important at the moment of speaking, I am sorry I mean no offense at all by saying it's worthless – Tes3awy Mar 28, 2016 at 22:39

1 Answer

Sorted by: Highest score (default)

131

Preparation

In order to understand what is happening, and eventually solve our problem, we first need to improve the script a little.

I've added logging of the important steps of your algorithm, refactored the code a little, and added saving of the mask and processed images, added ability to run the script using the individual frame images, along with some other modifications.

This is what the script looks like at this point:

```
import logging
import logging.handlers
import os
import time
import sys

import cv2
import numpy as np

from vehicle_counter import VehicleCounter

# ============================================================================

IMAGE_DIR = "images"
IMAGE_FILENAME_FORMAT = IMAGE_DIR + "/frame_%04d.png"

# Support either video file or individual frames
CAPTURE_FROM_VIDEO = False
if CAPTURE_FROM_VIDEO:
    IMAGE_SOURCE = "traffic.avi" # Video file
else:
    IMAGE_SOURCE = IMAGE_FILENAME_FORMAT # Image sequence

# Time to wait between frames, 0=forever
WAIT_TIME = 1 # 250 # ms

LOG_TO_FILE = True

# Colours for drawing on processed frames
DIVIDER_COLOUR = (255, 255, 0)
BOUNDING_BOX_COLOUR = (255, 0, 0)
CENTROID_COLOUR = (0, 0, 255)

# ============================================================================

def init_logging():
    main_logger = logging.getLogger()

    formatter = logging.Formatter(
        fmt='%(asctime)s.%(msecs)03d %(levelname)-8s [%(name)s] %(message)s'
        , datefmt='%Y-%m-%d %H:%M:%S')
```

```
    handler_stream = logging.StreamHandler(sys.stdout)
    handler_stream.setFormatter(formatter)
    main_logger.addHandler(handler_stream)

    if LOG_TO_FILE:
        handler_file = logging.handlers.RotatingFileHandler("debug.log"
            , maxBytes = 2**24
            , backupCount = 10)
        handler_file.setFormatter(formatter)
        main_logger.addHandler(handler_file)

    main_logger.setLevel(logging.DEBUG)

    return main_logger

# ============================================================================

def save_frame(file_name_format, frame_number, frame, label_format):
    file_name = file_name_format % frame_number
    label = label_format % frame_number

    log.debug("Saving %s as '%s'", label, file_name)
    cv2.imwrite(file_name, frame)

# ============================================================================

def get_centroid(x, y, w, h):
    x1 = int(w / 2)
    y1 = int(h / 2)

    cx = x + x1
    cy = y + y1

    return (cx, cy)

# ============================================================================

def detect_vehicles(fg_mask):
    log = logging.getLogger("detect_vehicles")

    MIN_CONTOUR_WIDTH = 21
    MIN_CONTOUR_HEIGHT = 21

    # Find the contours of any vehicles in the image
    contours, hierarchy = cv2.findContours(fg_mask
        , cv2.RETR_EXTERNAL
        , cv2.CHAIN_APPROX_SIMPLE)

    log.debug("Found %d vehicle contours.", len(contours))

    matches = []
    for (i, contour) in enumerate(contours):
        (x, y, w, h) = cv2.boundingRect(contour)
        contour_valid = (w >= MIN_CONTOUR_WIDTH) and (h >= MIN_CONTOUR_HEIGHT)

        log.debug("Contour #%d: pos=(x=%d, y=%d) size=(w=%d, h=%d) valid=%s"
            , i, x, y, w, h, contour_valid)

        if not contour_valid:
            continue

        centroid = get_centroid(x, y, w, h)
```

```
        matches.append(((x, y, w, h), centroid))

    return matches

# ============================================================================

def filter_mask(fg_mask):
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))

    # Fill any small holes
    closing = cv2.morphologyEx(fg_mask, cv2.MORPH_CLOSE, kernel)
    # Remove noise
    opening = cv2.morphologyEx(closing, cv2.MORPH_OPEN, kernel)

    # Dilate to merge adjacent blobs
    dilation = cv2.dilate(opening, kernel, iterations = 2)

    return dilation

# ============================================================================

def process_frame(frame_number, frame, bg_subtractor, car_counter):
    log = logging.getLogger("process_frame")

    # Create a copy of source frame to draw into
    processed = frame.copy()

    # Draw dividing line -- we count cars as they cross this line.
    cv2.line(processed, (0, car_counter.divider), (frame.shape[1],
car_counter.divider), DIVIDER_COLOUR, 1)

    # Remove the background
    fg_mask = bg_subtractor.apply(frame, None, 0.01)
    fg_mask = filter_mask(fg_mask)

    save_frame(IMAGE_DIR + "/mask_%04d.png"
        , frame_number, fg_mask, "foreground mask for frame #%d")

    matches = detect_vehicles(fg_mask)

    log.debug("Found %d valid vehicle contours.", len(matches))
    for (i, match) in enumerate(matches):
        contour, centroid = match

        log.debug("Valid vehicle contour #%d: centroid=%s, bounding_box=%s", i,
centroid, contour)

        x, y, w, h = contour

        # Mark the bounding box and the centroid on the processed frame
        # NB: Fixed the off-by one in the bottom right corner
        cv2.rectangle(processed, (x, y), (x + w - 1, y + h - 1),
BOUNDING_BOX_COLOUR, 1)
        cv2.circle(processed, centroid, 2, CENTROID_COLOUR, -1)

    log.debug("Updating vehicle count...")
    car_counter.update_count(matches, processed)

    return processed

# ============================================================================
```

```
def main():
    log = logging.getLogger("main")

    log.debug("Creating background subtractor...")
    bg_subtractor = cv2.BackgroundSubtractorMOG()

    log.debug("Pre-training the background subtractor...")
    default_bg = cv2.imread(IMAGE_FILENAME_FORMAT % 119)
    bg_subtractor.apply(default_bg, None, 1.0)

    car_counter = None # Will be created after first frame is captured

    # Set up image source
    log.debug("Initializing video capture device #%s...", IMAGE_SOURCE)
    cap = cv2.VideoCapture(IMAGE_SOURCE)

    frame_width = cap.get(cv2.cv.CV_CAP_PROP_FRAME_WIDTH)
    frame_height = cap.get(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT)
    log.debug("Video capture frame size=(w=%d, h=%d)", frame_width,
frame_height)

    log.debug("Starting capture loop...")
    frame_number = -1
    while True:
        frame_number += 1
        log.debug("Capturing frame #%d...", frame_number)
        ret, frame = cap.read()
        if not ret:
            log.error("Frame capture failed, stopping...")
            break

        log.debug("Got frame #%d: shape=%s", frame_number, frame.shape)

        if car_counter is None:
            # We do this here, so that we can initialize with actual frame size
            log.debug("Creating vehicle counter...")
            car_counter = VehicleCounter(frame.shape[:2], frame.shape[0] / 2)

        # Archive raw frames from video to disk for later inspection/testing
        if CAPTURE_FROM_VIDEO:
            save_frame(IMAGE_FILENAME_FORMAT
                , frame_number, frame, "source frame #%d")

        log.debug("Processing frame #%d...", frame_number)
        processed = process_frame(frame_number, frame, bg_subtractor,
car_counter)

        save_frame(IMAGE_DIR + "/processed_%04d.png"
            , frame_number, processed, "processed frame #%d")

        cv2.imshow('Source Image', frame)
        cv2.imshow('Processed Image', processed)

        log.debug("Frame #%d processed.", frame_number)

        c = cv2.waitKey(WAIT_TIME)
        if c == 27:
            log.debug("ESC detected, stopping...")
            break

    log.debug("Closing video capture device...")
```

```
    cap.release()
    cv2.destroyAllWindows()
    log.debug("Done.")

# ======================================================================

if __name__ == "__main__":
    log = init_logging()

    if not os.path.exists(IMAGE_DIR):
        log.debug("Creating image directory `%s`...", IMAGE_DIR)
        os.makedirs(IMAGE_DIR)

    main()
```

This script is responsible for processing of the stream of images, and identifying all the vehicles in each frame -- I refer to them as `matches` in the code.

The task of counting the detected vehicles is delegated to class `VehicleCounter`. The reason why I chose to make this a class will become evident as we progress. I did not implement your vehicle counting algorithm, because it will not work for reasons that will again become evident as we dig into this deeper.

File `vehicle_counter.py` contains the following code:

```
import logging

# ======================================================================

class VehicleCounter(object):
    def __init__(self, shape, divider):
        self.log = logging.getLogger("vehicle_counter")

        self.height, self.width = shape
        self.divider = divider

        self.vehicle_count = 0


    def update_count(self, matches, output_image = None):
        self.log.debug("Updating count using %d matches...", len(matches))

# ======================================================================
```

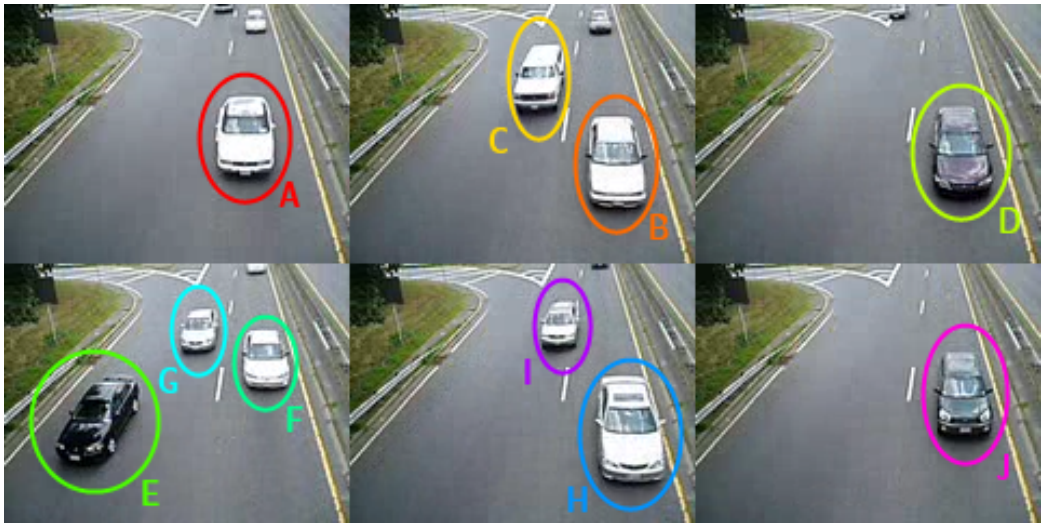
Finally, I wrote a script that will stitch all the generated images together, so it's easier to inspect them:

```
import cv2
import numpy as np

# ======================================================================
```

```
INPUT_WIDTH = 160
INPUT_HEIGHT = 120

OUTPUT_TILE_WIDTH = 10
OUTPUT_TILE_HEIGHT = 12

TILE_COUNT = OUTPUT_TILE_WIDTH * OUTPUT_TILE_HEIGHT

# ================================================================================

def stitch_images(input_format, output_filename):
    output_shape = (INPUT_HEIGHT * OUTPUT_TILE_HEIGHT
        , INPUT_WIDTH * OUTPUT_TILE_WIDTH
        , 3)
    output = np.zeros(output_shape, np.uint8)

    for i in range(TILE_COUNT):
        img = cv2.imread(input_format % i)
        cv2.rectangle(img, (0, 0), (INPUT_WIDTH - 1, INPUT_HEIGHT - 1), (0, 0,
255), 1)
        # Draw the frame number
        cv2.putText(img, str(i), (2, 10)
            , cv2.FONT_HERSHEY_PLAIN, 0.7, (255, 255, 255), 1)
        x = i % OUTPUT_TILE_WIDTH * INPUT_WIDTH
        y = i / OUTPUT_TILE_WIDTH * INPUT_HEIGHT
        output[y:y+INPUT_HEIGHT, x:x+INPUT_WIDTH,:] = img

    cv2.imwrite(output_filename, output)

# ================================================================================

stitch_images("images/frame_%04d.png", "stitched_frames.png")
stitch_images("images/mask_%04d.png", "stitched_masks.png")
stitch_images("images/processed_%04d.png", "stitched_processed.png")
```

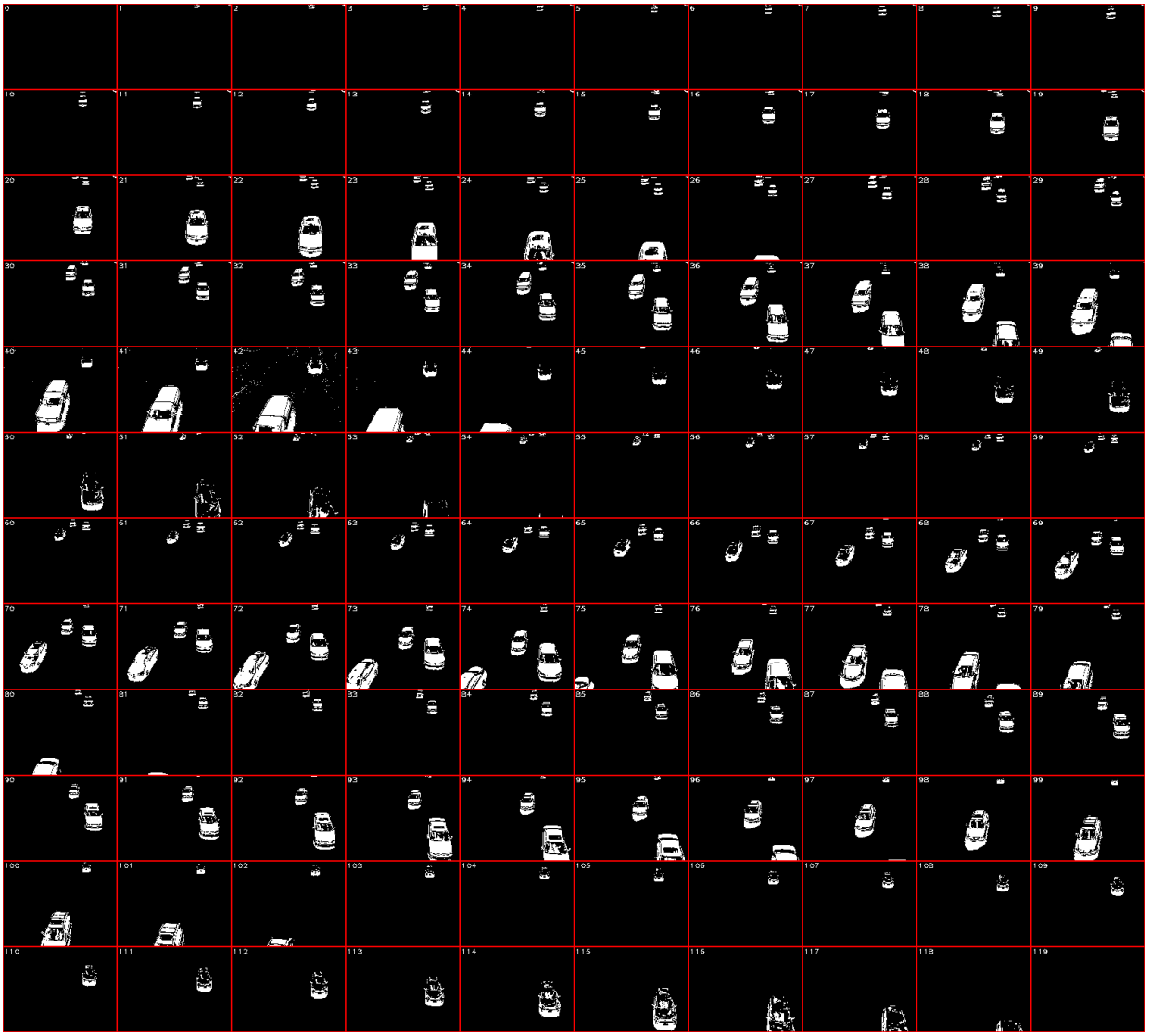
Analysis

In order to solve this problem, we should have some idea about what results we expect to get. We should also label all the distinct cars in the video, so it's easier to talk about them.

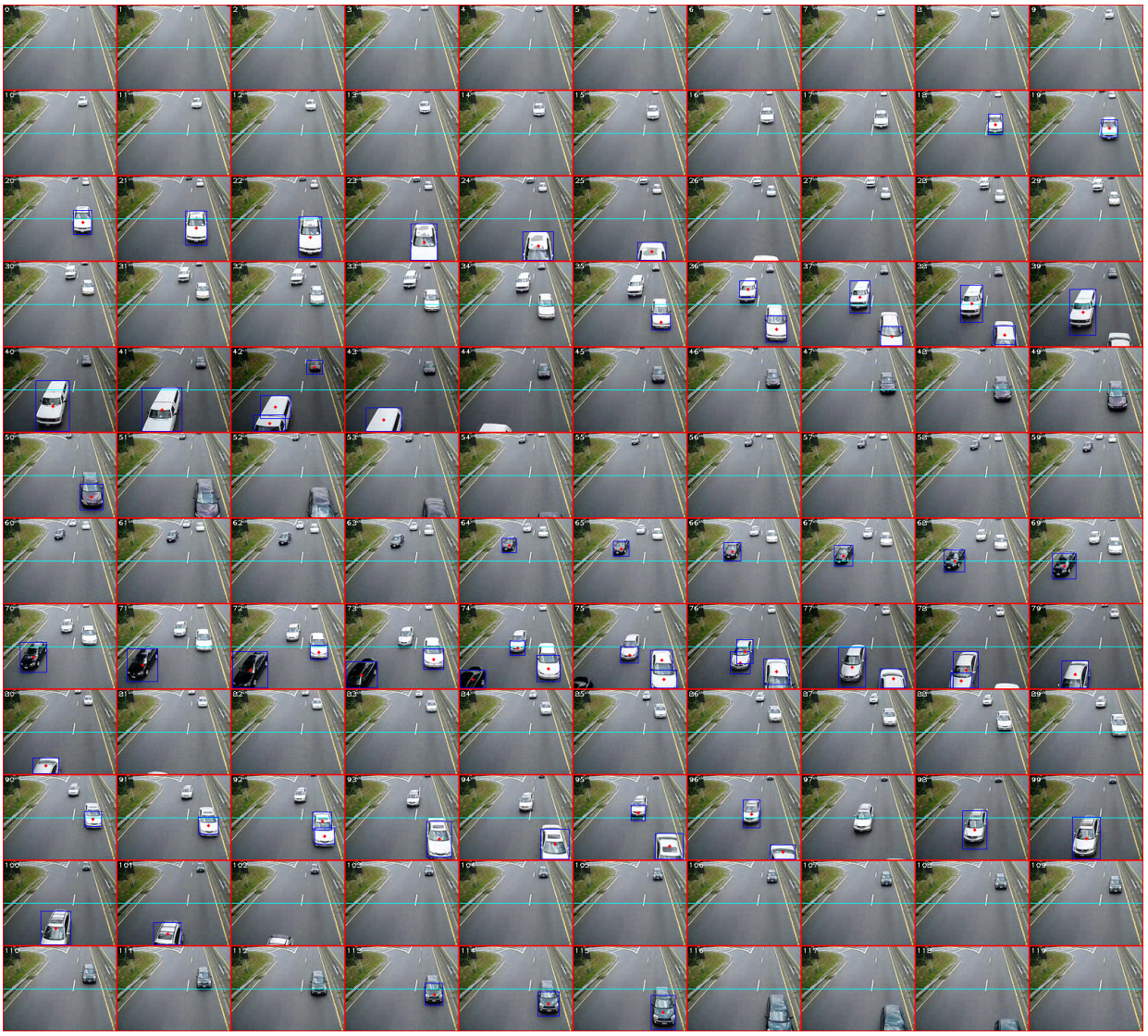


If we run our script, and stitch the images together, we get the a number of useful files to help us analyze the problem:

- Image containing a mosaic of input frames
- Image containing a mosaic of foreground masks:



- Image containing a mosaic of processed frames



- The debug log for the run.

Upon inspecting those, a number of issues become evident:

- The foreground masks tend to be noisy. We should do some filtering (erode/dilate?) to get rid of the noise and narrow gaps.
- Sometimes we miss vehicles (grey ones).
- Some vehicles get detected twice in the single frame.
- Vehicles are rarely detected in the upper regions of the frame.
- The same vehicle is often detected in consecutive frames. We need to figure out a way of tracking the same vehicle in consecutive frames, and counting it only once.

Solution

1. Pre-Seeding the Background Subtractor

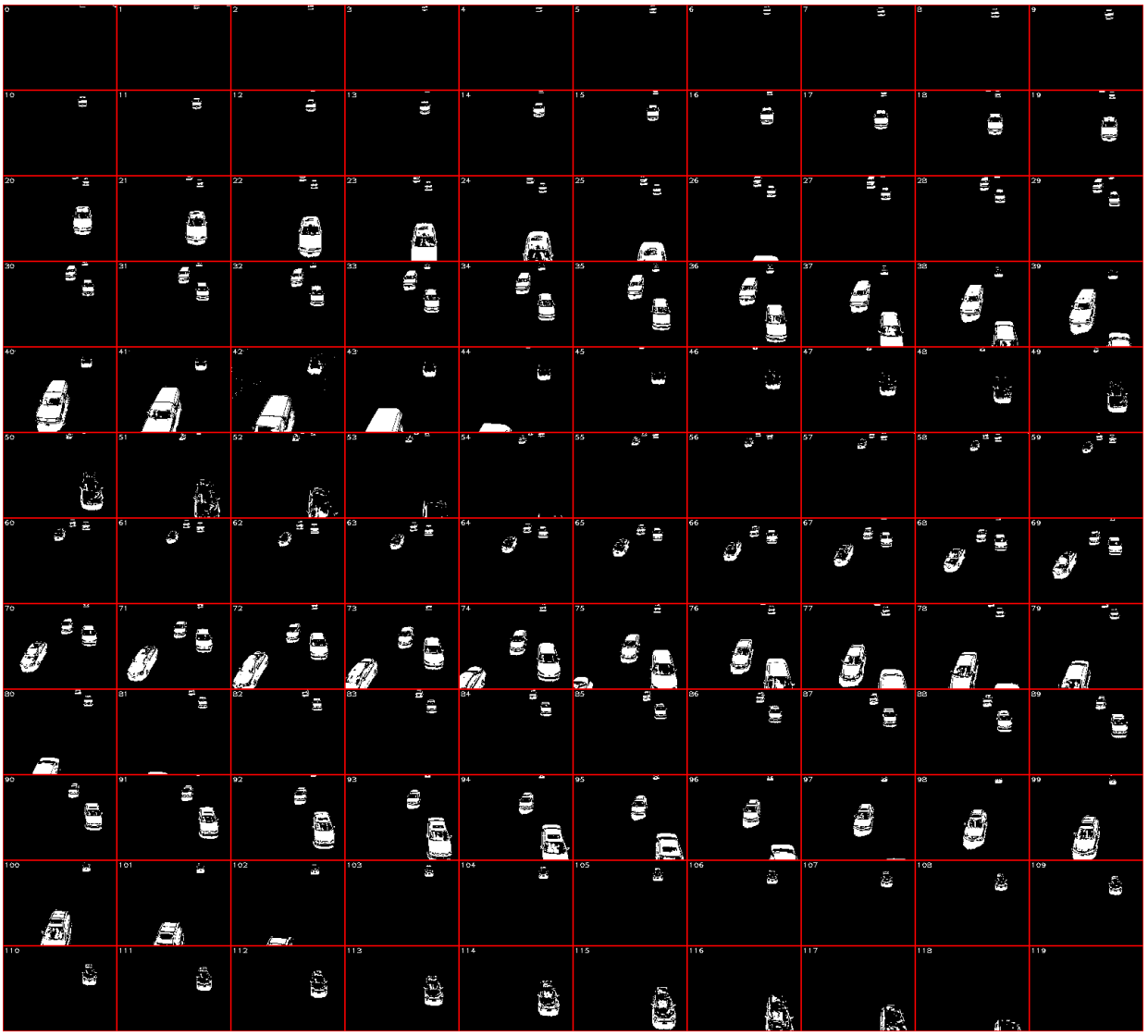
Our video is quite short, only 120 frames. With learning rate of `0.01`, it will take a substantial part of the video for the background detector to stabilize.

Fortunately, the last frame of the video (frame number 119) is completely devoid of vehicles, and therefore we can use it as our initial background image. (Other options of obtaining suitable image are mentioned in notes and comments.)

To use this initial background image, we simply load it, and `apply` it on the background subtractor with learning factor `1.0`:

```
bg_subtractor = cv2.BackgroundSubtractorMOG()
default_bg = cv2.imread(IMAGE_FILENAME_FORMAT % 119)
bg_subtractor.apply(default_bg, None, 1.0)
```

When we look at the new mosaic of masks we can see that we get less noise and the vehicle detection works better in the early frames.

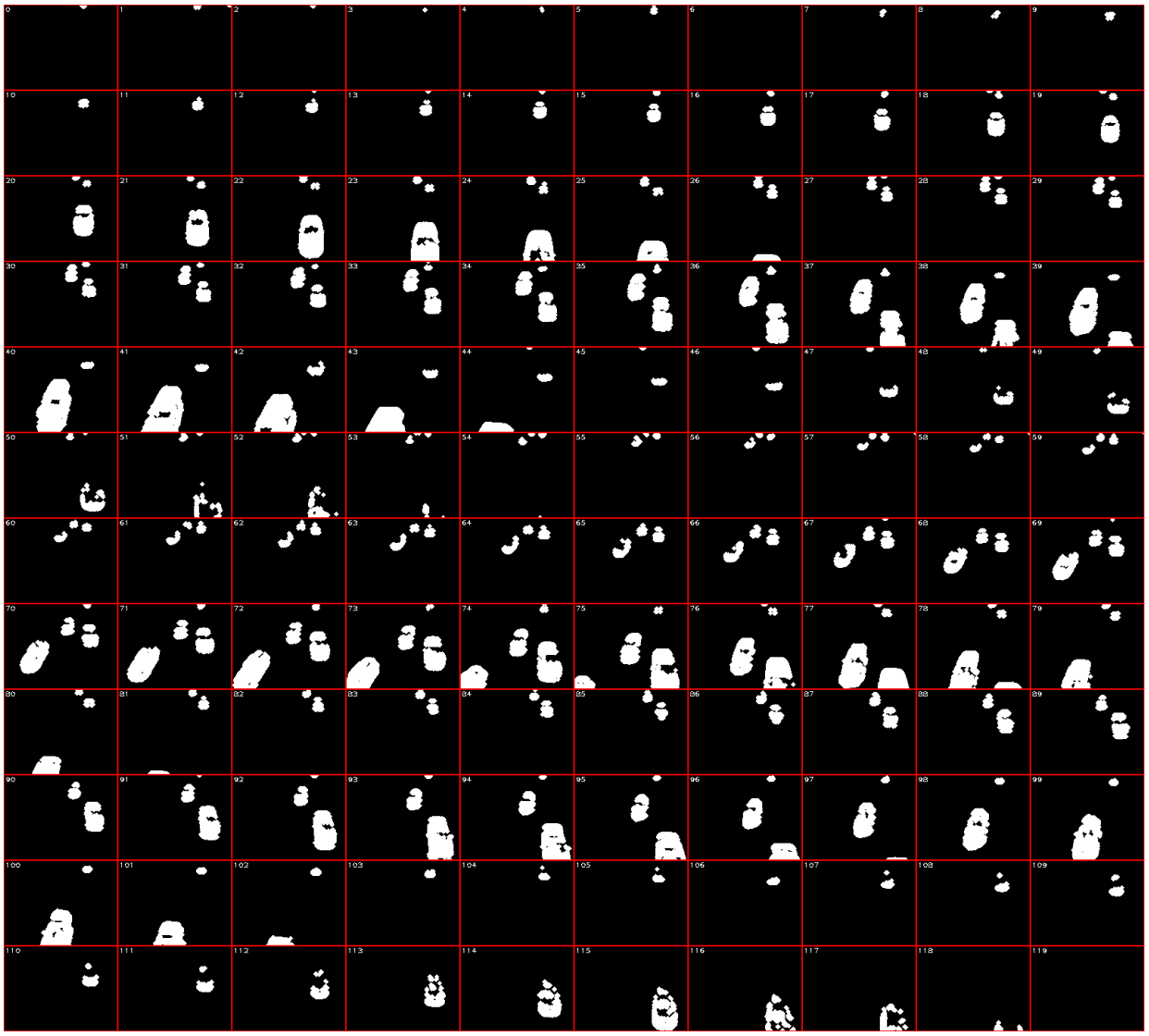


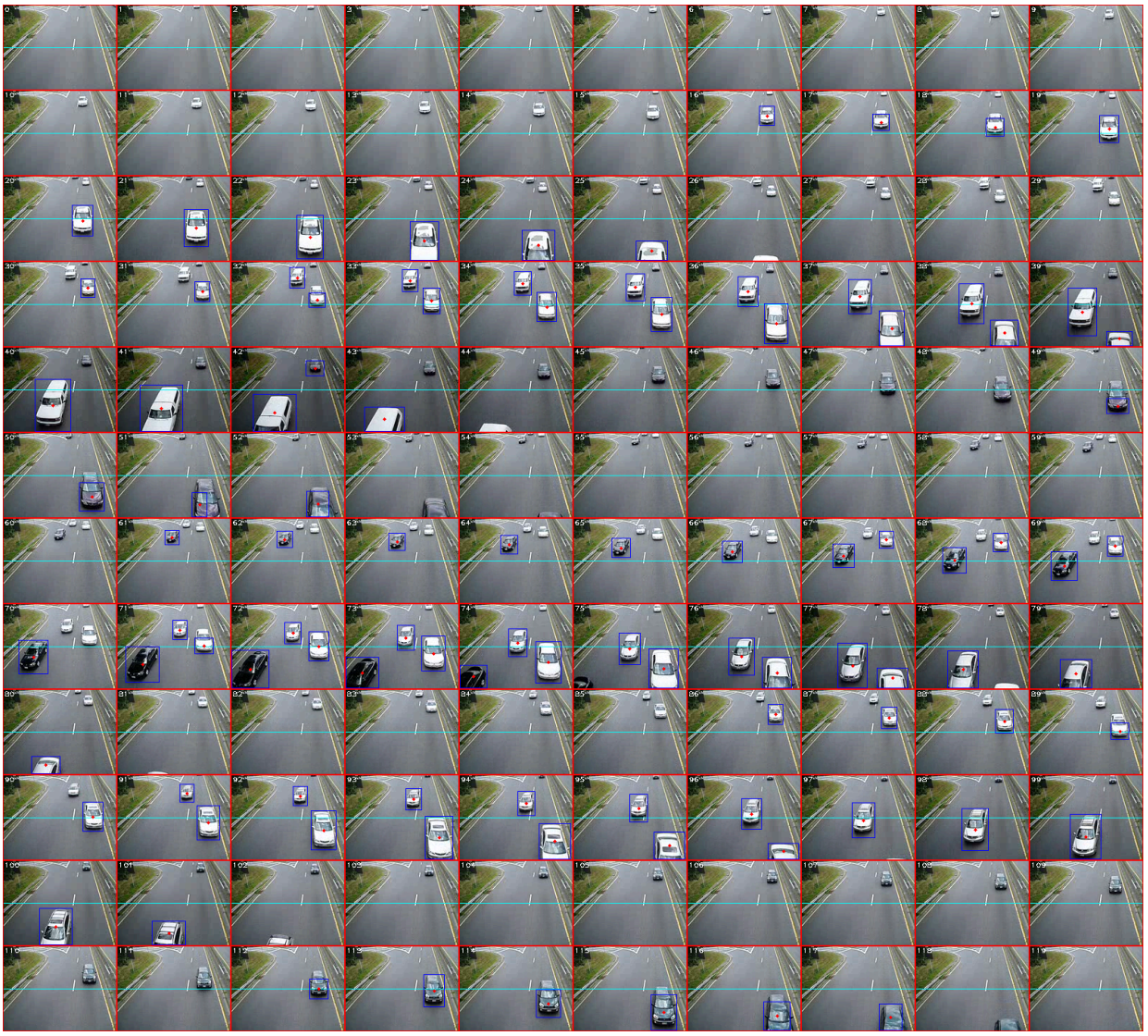
2. Cleaning Up the Foreground Mask

A simple approach to improve our foreground mask is to apply a few morphological transformations.

```
def filter_mask(fg_mask):
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))

    # Fill any small holes
    closing = cv2.morphologyEx(fg_mask, cv2.MORPH_CLOSE, kernel)
    # Remove noise
    opening = cv2.morphologyEx(closing, cv2.MORPH_OPEN, kernel)

    # Dilate to merge adjacent blobs
    dilation = cv2.dilate(opening, kernel, iterations = 2)

    return dilation
```

Inspecting the masks, processed frames and the log file generated with filtering, we can see that we now detect vehicles more reliably, and have mitigated the issue of different parts of one vehicle being detected as separate objects.


0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119



3. Tracking Vehicles Between Frames

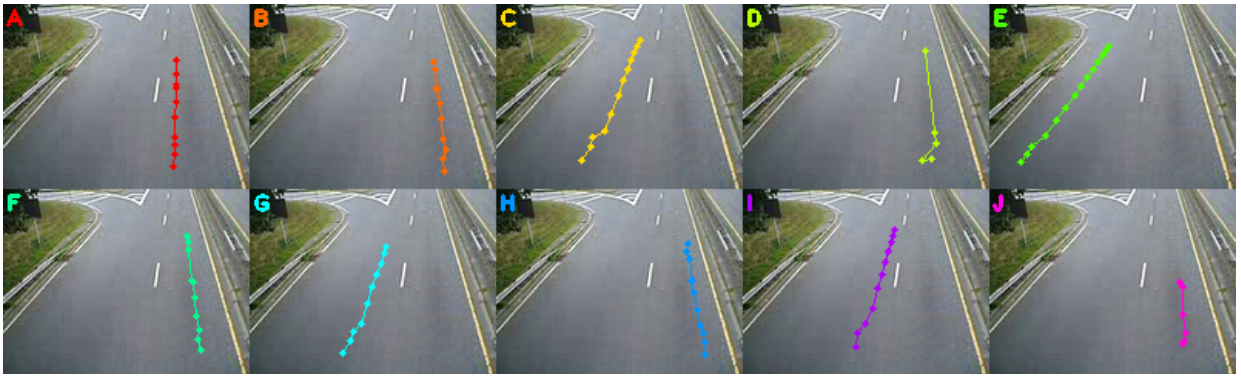
At this point, we need to go through our log file, and collect all the centroid coordinates for each vehicle. This will allow us to plot and inspect the path each vehicle traces across the image, and develop an algorithm to do this automatically. To make this process easier, we can create a reduced log by grepping out the relevant entries.

The lists of centroid coordinates:

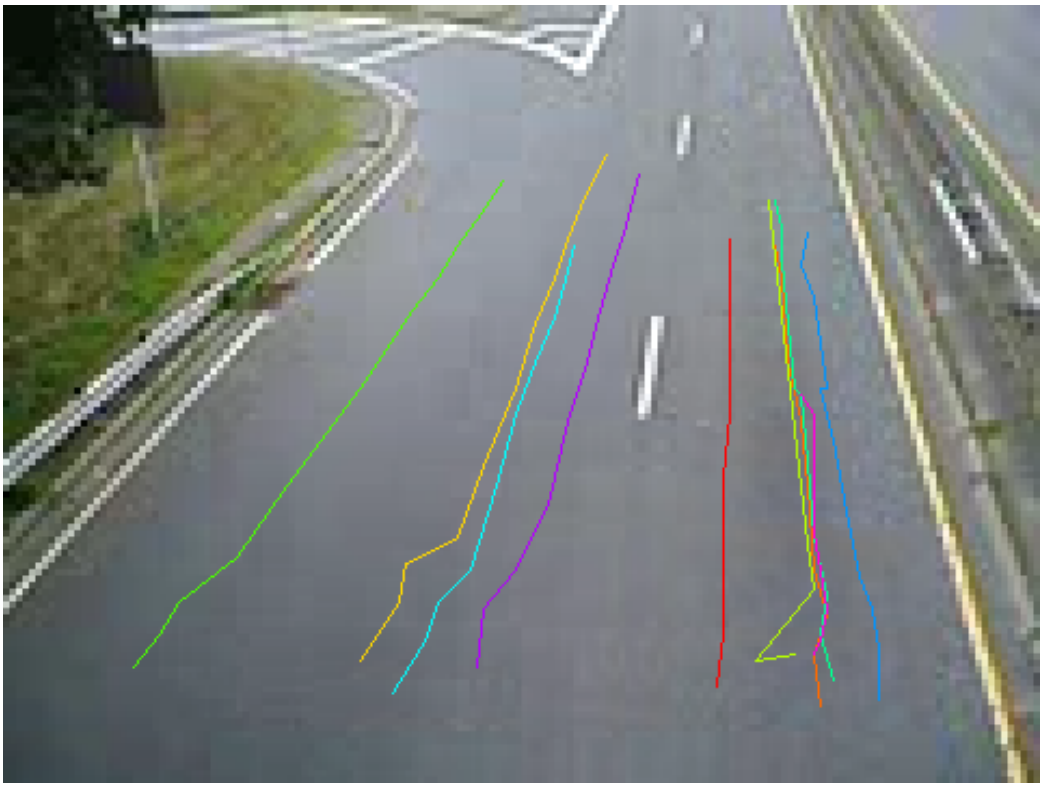
```
traces = {
    'A': [(112, 36), (112, 45), (112, 52), (112, 54), (112, 63), (111, 73),
(111, 86), (111, 91), (111, 97), (110, 105)]
    , 'B': [(119, 37), (120, 42), (121, 54), (121, 55), (123, 64), (124, 74),
(125, 87), (127, 94), (125, 100), (126, 108)]
    , 'C': [(93, 23), (91, 27), (89, 31), (87, 36), (85, 42), (82, 49), (79,
59), (74, 71), (70, 82), (62, 86), (61, 92), (55, 101)]
    , 'D': [(118, 30), (124, 83), (125, 90), (116, 101), (122, 100)]
    , 'E': [(77, 27), (75, 30), (73, 33), (70, 37), (67, 42), (63, 47), (59,
53), (55, 59), (49, 67), (43, 75), (36, 85), (27, 92), (24, 97), (20, 102)]
    , 'F': [(119, 30), (120, 34), (120, 39), (122, 59), (123, 60), (124, 70),
(125, 82), (127, 91), (126, 97), (128, 104)]
```

```
    , 'G': [(88, 37), (87, 41), (85, 48), (82, 55), (79, 63), (76, 74), (72,
87), (67, 92), (65, 98), (60, 106)]
    , 'H': [(124, 35), (123, 40), (125, 45), (127, 59), (126, 59), (128, 67),
(130, 78), (132, 88), (134, 93), (135, 99), (135, 107)]
    , 'I': [(98, 26), (97, 30), (96, 34), (94, 40), (92, 47), (90, 55), (87,
64), (84, 77), (79, 87), (74, 93), (73, 102)]
    , 'J': [(123, 60), (125, 63), (125, 81), (127, 93), (126, 98), (125, 100)]
}
```

Individual vehicle traces plotted on the background:

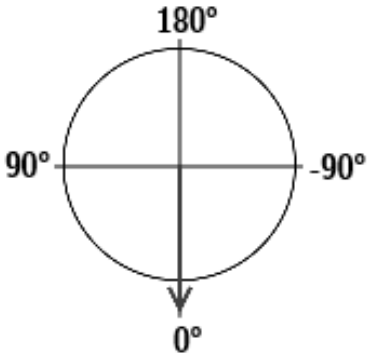


Combined enlarged image of all the vehicle traces:



Vectors

In order to analyze the movement, we need to work with vectors (i.e. the distance and direction moved). The following diagram shows how the angles correspond to movement of vehicles in the image.

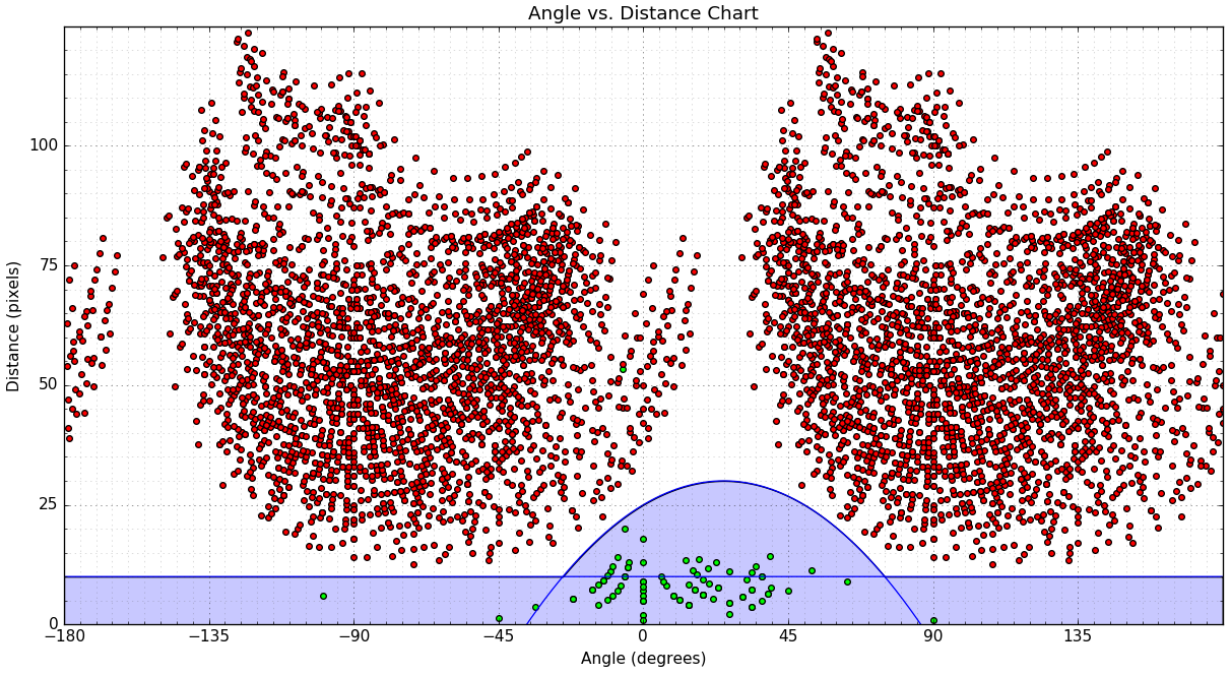


We can use the following function to calculate the vector between two points:

```
def get_vector(a, b):
    """Calculate vector (distance, angle in degrees) from point a to point b.

    Angle ranges from -180 to 180 degrees.
    Vector with angle 0 points straight down on the image.
    Values increase in clockwise direction.
    """
    dx = float(b[0] - a[0])
    dy = float(b[1] - a[1])

    distance = math.sqrt(dx**2 + dy**2)

    if dy > 0:
        angle = math.degrees(math.atan(-dx/dy))
    elif dy == 0:
        if dx < 0:
            angle = 90.0
        elif dx > 0:
            angle = -90.0
        else:
            angle = 0.0
    else:
        if dx < 0:
            angle = 180 - math.degrees(math.atan(dx/dy))
        elif dx > 0:
            angle = -180 - math.degrees(math.atan(dx/dy))
        else:
            angle = 180.0

    return distance, angle
```

Categorization

One way we can look for patterns that could be used to categorize the movements as valid/invalid is to make a scatter plot (angle vs. distance):



- Green points represent valid movement, that we determined using the lists of points for each vehicle.
- Red points represent invalid movement - vectors between points in adjacent traffic lanes.
- I plotted two blue curves, which we can use to separate the two types of movements. Any point that lies below either curve can be considered as valid. The curves are:
  - `distance = -0.008 * angle**2 + 0.4 * angle + 25.0`
  - `distance = 10.0`

We can use the following function to categorize the movement vectors:

```
def is_valid_vector(a):
    distance, angle = a
    threshold_distance = max(10.0, -0.008 * angle**2 + 0.4 * angle + 25.0)
    return (distance <= threshold_distance)
```

NB: There is one outlier, which is occurs due to our loosing track of vehicle **D** in frames 43..48.

Algorithm

We will use class `Vehicle` to store information about each tracked vehicle:

- Some kind of identifier
- List of positions, most recent at front
- Last-seen counter -- number of frames since we've last seen this vehicle

- Flag to mark whether the vehicle was counted or not

Class `VehicleCounter` will store a list of currently tracked vehicles and keep track of the total count. On each frame, we will use the list of bounding boxes and positions of identified vehicles (candidate list) to update the state of `VehicleCounter`:

1. Update currently tracked `Vehicle`s:
    - For each vehicle
        - If there is any valid match for given vehicle, update vehicle position and reset its last-seen counter. Remove the match from the candidate list.
        - Otherwise, increase the last-seen counter for that vehicle.
2. Create new `Vehicle`s for any remaining matches
3. Update vehicle count
    - For each vehicle
        - If the vehicle is past divider and has not been counted yet, update the total count and mark the vehicle as counted
4. Remove vehicles that are no longer visible
    - For each vehicle
        - If the last-seen counter exceeds threshold, remove the vehicle

4. Solution

We can reuse the main script with the final version of `vehicle_counter.py`, containing the implementation of our counting algorithm:

```
import logging
import math

import cv2
import numpy as np

# ============================================================================

CAR_COLOURS = [ (0,0,255), (0,106,255), (0,216,255), (0,255,182), (0,255,76)
    , (144,255,0), (255,255,0), (255,148,0), (255,0,178), (220,0,255) ]

# ============================================================================

class Vehicle(object):
    def __init__(self, id, position):
        self.id = id
        self.positions = [position]
        self.frames_since_seen = 0
        self.counted = False
```

```
    @property
    def last_position(self):
        return self.positions[-1]

    def add_position(self, new_position):
        self.positions.append(new_position)
        self.frames_since_seen = 0

    def draw(self, output_image):
        car_colour = CAR_COLOURS[self.id % len(CAR_COLOURS)]
        for point in self.positions:
            cv2.circle(output_image, point, 2, car_colour, -1)
            cv2.polylines(output_image, [np.int32(self.positions)]
                , False, car_colour, 1)


# ====================================================================

class VehicleCounter(object):
    def __init__(self, shape, divider):
        self.log = logging.getLogger("vehicle_counter")

        self.height, self.width = shape
        self.divider = divider

        self.vehicles = []
        self.next_vehicle_id = 0
        self.vehicle_count = 0
        self.max_unseen_frames = 7


    @staticmethod
    def get_vector(a, b):
        """Calculate vector (distance, angle in degrees) from point a to point
b.

        Angle ranges from -180 to 180 degrees.
        Vector with angle 0 points straight down on the image.
        Values increase in clockwise direction.
        """
        dx = float(b[0] - a[0])
        dy = float(b[1] - a[1])

        distance = math.sqrt(dx**2 + dy**2)

        if dy > 0:
            angle = math.degrees(math.atan(-dx/dy))
        elif dy == 0:
            if dx < 0:
                angle = 90.0
            elif dx > 0:
                angle = -90.0
            else:
                angle = 0.0
        else:
            if dx < 0:
                angle = 180 - math.degrees(math.atan(dx/dy))
            elif dx > 0:
                angle = -180 - math.degrees(math.atan(dx/dy))
            else:
                angle = 180.0
```

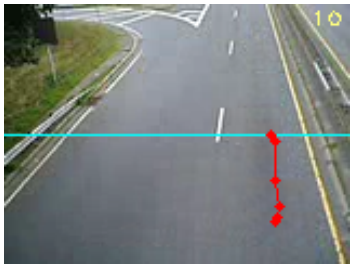
```
        return distance, angle


    @staticmethod
    def is_valid_vector(a):
        distance, angle = a
        threshold_distance = max(10.0, -0.008 * angle**2 + 0.4 * angle + 25.0)
        return (distance <= threshold_distance)


    def update_vehicle(self, vehicle, matches):
        # Find if any of the matches fits this vehicle
        for i, match in enumerate(matches):
            contour, centroid = match

            vector = self.get_vector(vehicle.last_position, centroid)
            if self.is_valid_vector(vector):
                vehicle.add_position(centroid)
                self.log.debug("Added match (%d, %d) to vehicle #%d. vector=
(%0.2f,%0.2f)"
                    , centroid[0], centroid[1], vehicle.id, vector[0],
vector[1])
                return i

        # No matches fit...
        vehicle.frames_since_seen += 1
        self.log.debug("No match for vehicle #%d. frames_since_seen=%d"
            , vehicle.id, vehicle.frames_since_seen)

        return None


    def update_count(self, matches, output_image = None):
        self.log.debug("Updating count using %d matches...", len(matches))

        # First update all the existing vehicles
        for vehicle in self.vehicles:
            i = self.update_vehicle(vehicle, matches)
            if i is not None:
                del matches[i]

        # Add new vehicles based on the remaining matches
        for match in matches:
            contour, centroid = match
            new_vehicle = Vehicle(self.next_vehicle_id, centroid)
            self.next_vehicle_id += 1
            self.vehicles.append(new_vehicle)
            self.log.debug("Created new vehicle #%d from match (%d, %d)."
                , new_vehicle.id, centroid[0], centroid[1])

        # Count any uncounted vehicles that are past the divider
        for vehicle in self.vehicles:
            if not vehicle.counted and (vehicle.last_position[1] >
self.divider):
                self.vehicle_count += 1
                vehicle.counted = True
                self.log.debug("Counted vehicle #%d (total count=%d)."
                    , vehicle.id, self.vehicle_count)

        # Optionally draw the vehicles on an image
        if output_image is not None:
            for vehicle in self.vehicles:
```

```
                vehicle.draw(output_image)

            cv2.putText(output_image, ("%02d" % self.vehicle_count), (142, 10)
                , cv2.FONT_HERSHEY_PLAIN, 0.7, (127, 255, 255), 1)

        # Remove vehicles that have not been seen long enough
        removed = [ v.id for v in self.vehicles
            if v.frames_since_seen >= self.max_unseen_frames ]
        self.vehicles[:] = [ v for v in self.vehicles
            if not v.frames_since_seen >= self.max_unseen_frames ]
        for id in removed:
            self.log.debug("Removed vehicle #%d.", id)

        self.log.debug("Count updated, tracking %d vehicles.",
len(self.vehicles))

# ============================================================================
```

The program now draws the historical paths of all currently tracked vehicles into the output image, along with the vehicle count. Each vehicle is assigned 1 of 10 colours.

Notice that vehicle D ends up being tracked twice, however it is counted only once, since we lose track of it before crossing the divider. Ideas on how to resolve this are mentioned in the appendix.

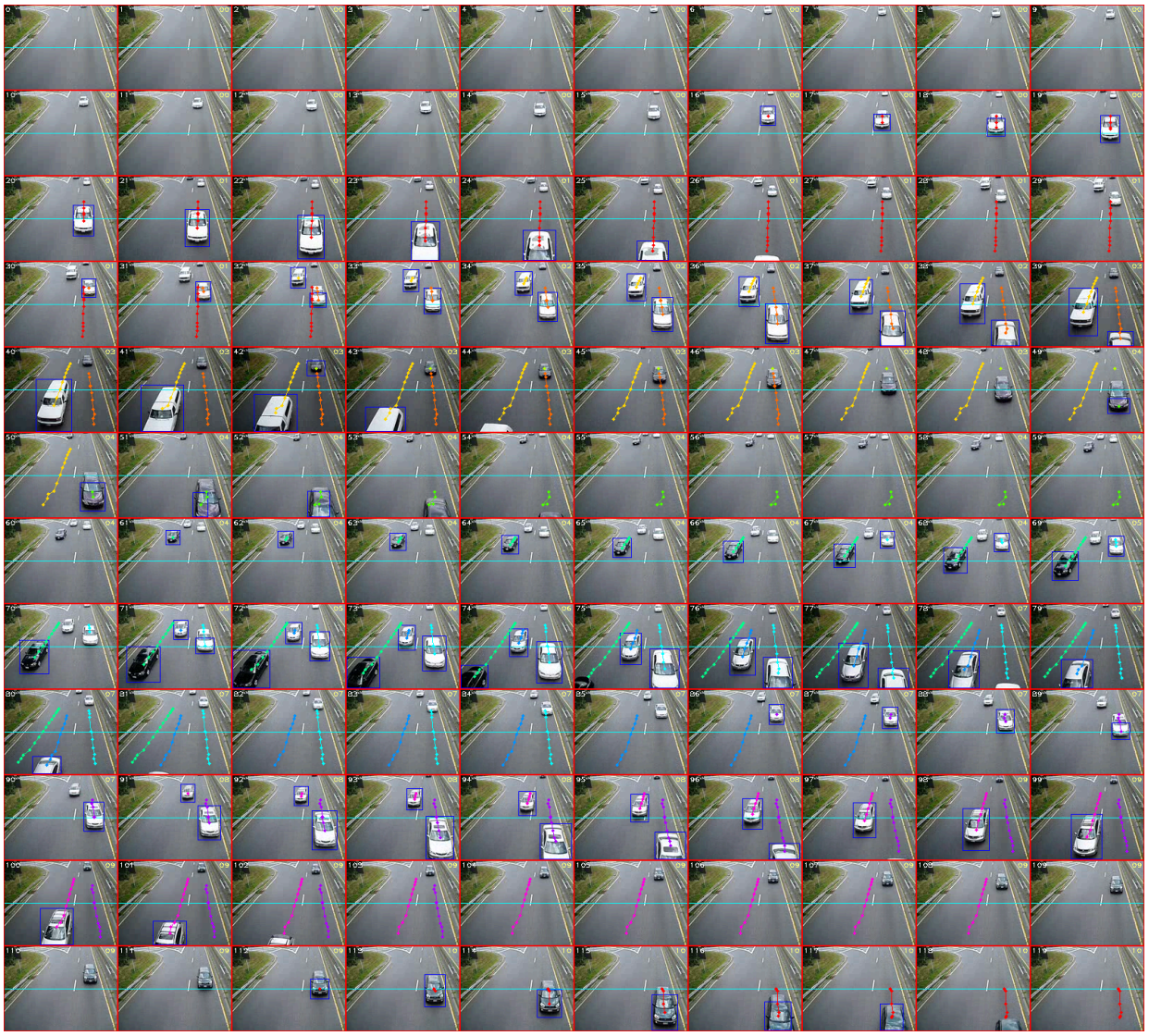
Based on the last processed frame generated by the script



the total vehicle count is **10**. This is a correct result.

More details can be found in the output the script generated:

- Full debug log
- Filtered out vehicle counter log
- A mosaic of the processed frames:



A. Potential Improvements

- Refactor, add unit tests.
- Improve filtering/preprocessing of the foreground mask
  - Multiple iterations of filtering, fill holes using `cv2.drawContours` with `CV_FILLED`?
  - Watershed Algorithm?
- Improve categorization of movement vectors
  - Create a predictor to estimate initial movement angle when vehicles are created (and only one position is known)... in order to be able to
  - Use *change in direction* rather than *direction* alone (I think this would cluster the angles of valid motion vectors close to zero).
- Improve vehicle tracking
  - Predict position for frames where vehicle is not seen.

B. Notes

- It seems it's not possible to directly extract the current background image from `BackgroundSubtractorMOG` in Python (at least in OpenCV 2.4.x), but there is a way to do it with a little work.
- As suggested by Henrik, we can obtain a good estimate of the background using median blending.

Share

Improve this answer

Follow

edited May 23, 2017 at 12:26

Community Bot

1 • 1

answered Mar 29, 2016 at 2:50

Dan Mašek

19k • 6 • 61 • 90

1 historic median of the images, is time consuming, but will (In my humble experience) often provide a decent background estimate, even if you did not have frame 119 - good work btw. :) here is a link: petapixel.com/2013/05/29/… – Henrik Mar 29, 2016 at 12:19

No problem, it was a fun little exercise. See if you can make any improvements, there's a still a lot that can be done. You can put your solution on your github account. You can also try to make it work with a different dataset. What I wrote is tailored to the small amount of data we had. In real-world scenario, we would need a significantly larger sample set to develop a reliable system. – Dan Mašek Mar 30, 2016 at 21:27

@Dan Mašek I have been reading the book Digital Image Processing Gonzalez, and I finished it, and almost implemented everything on Matlab without the Matlab version, I have been using Matlab for more than 5 years now, but I am new to Python and OpenCV that is why I needed some help, and really you helped me more than enough. I will try the potential improvements you wrote and will keep you updated. Thanks again so much – Tes3awy Mar 30, 2016 at 21:35

@Dan Mašek Hi Dan, I have changed to OpenCV 3.1.0-dev, and I am modifying the provided code in this post. I have used createBackgroundSubtractionKNN to remove shadows as an improvement, also I preformed some erosion to delete unwanted pixels if the camera is moving. The problem I am facing now is how to make the drawn line to be vertical instead of being horizontal. BTW this code works pretty well on many projects. – Tes3awy Jul 24, 2016 at 21:31

1 I used different approach of building car pathes. I used minimum euclidean distance to find the most near point and link them into the path. Here is my class for that: gist.github.com/creotiv/febf149ae2211f70fd45c93b4b0218b1 – Andrey Nikishaev Jun 16, 2017 at 15:21