Assignment 1 - License plate locator

Description

Automatically locate the number plate in the following image. (Available in numberplates2020.zip from Blackboard). You may try a 2D cross-correlation with a template matching the plate border, but you will probably need to 'chamfer' the template by convolving with a Gaussian or some other blurring function. You'll also need a good edge detector such as the Canny Edge detector. Test your method on the other example car images from numberplates2020.zip and show the results.

In your report discuss methods used, problems encountered, performance, and possible solutions. Comment on the problems encountered in plate extraction and the difficulties in designing a general plate extractor.

Marking Scheme

- Coding of a solution to locate one number plate, description and explanation of method, related images and graphs (5 marks)
- Testing method on all other number plates, modification of method as required, showing related images with detection overlay (2 marks)
- Comment on the many challenges to number plate detection including lighting, different shaped plates, different coloured plates, perspective distortion, angle of rotation, and so on and suggest possible solutions. (3 marks)

Introduction to algorithm

Firstly I'll introduce the different methods I've worked with and the algorithm flow step by step. Then I'll run the algorithm to show the results for car 1. I will discuss why some methods were chosen and some abandoned for this case. Next I'll show how the algorithm works for the rest of the images, and discuss why the results are variyng.

Step 1 - Imports

I will import the necessary libraries and need a function to import the image I want to process. I'm using the OpenCV library for Python to do image processing. Numpy is needed for mathematical operations and pyplot to plot results.

The image is imported in colors so I can use the orignal for drawing the countours in the later steps.

In [52]:
```
import cv2
import numpy as np
from matplotlib import pyplot as plt

def import_image(image):
    """Imports the given image in colors"""
    return cv2.imread(image, cv2.IMREAD_COLOR)
```

Step 2 - Preprocessing

Further we introduce different ways of preprocessing the images.

Grayscale

First, you would like to convert your image to grayscale - this is done to reduce information in each pixel, and we keep it to only two dimensions.

Thresholding

Thresholding helps classifying pixels in the images, given the threshold value. There are several types of thresholds available, and I did try out both simple thresholding, adaptive thresholding and Otsu thresholding.

Blur

Bluring is used to remove noise from the image and smooth things out, using a low-pass filter kernel. The kernel size can be chosen. There are many options for different blur methods, but Gaussian is used in this case.

Resizing and cropping

Images may come in varying sizes and therefore it would be nice to change the size of the images I'm processing. The crop methods divides the image into 10 both horizontally and vertically, and crop the image removing top 2/10 and bottom 1/10 of the height, and 1/10 of both left and right side from the width.

Morphology

Methods used for extracting image components that are useful in the representation and description of region shape. I've implemented methods for erosion, dilation, opening and closing.

Remove shadows

A function to remove shadows from the images, which includes using many of the already implemented methods. This method include:

- Dilate the image, in order to get rid of the text.
- Blur the image, this gives a good background image that contains shadows.
- Calculate the difference between the original and the background.
- Normalize the image, so that we use the full dynamic range.
- Threshold the image using Truncated threshold, and then normalize again

In [53]:
```
def grayscaled(image):
    """Turns an image into a grayscaled image"""
    return cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

def threshold(image, thresh=170, maxval=255, threshold_type=cv2.THRESH_TOZERO):
    ret, image_thresh = cv2.threshold(image, thresh, maxval, threshold_type)
    return image_thresh

def adaptive_threshold(image, maxval=255, adaptive_threshold_type=cv2.ADAPTIVE_THRESH_GAUSSIAN_C, threshold_type=cv2.THRESH_TOZE
RO, block_size=11, c=2):
    return cv2.adaptiveThreshold(image, maxval, adaptive_threshold_type, threshold_type, block_size, c)

def blur(image, kernel_tuple=(5,5)):
    return cv2.GaussianBlur(image, kernel_tuple, 0)

def resize(image, size=None, scale_x=2, scale_y=2, interpolation=cv2.INTER_LINEAR):
    return cv2.resize(src=image, dsize=size, fx=scale_x, fy=scale_y, interpolation=interpolation)

def crop(image):
    image_tuple = image.shape
    height, width = image_tuple[0], image_tuple[1]
    return image[height*2//10 : height*9//10, width//10 : width*9//10]

def morph_opening(image, kernel_size=(5,5)):
    kernel = np.ones(kernel_size, np.uint8)
    return cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)

def morph_closing(image, kernel_size=(5,5)):
    kernel = np.ones(kernel_size, np.uint8)
    return cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)

def remove_shadows(image):
    dilated_img = cv2.dilate(image, np.ones((5,5), np.uint8))
    cv2.imshow("dil", dilated_img)

    blurred_img = blur(dilated_img, (3,3))
    cv2.imshow("blur", blurred_img)

    diff_img = 255 - cv2.absdiff(image, blurred_img)
    cv2.imshow("diff", diff_img)

    norm_img = diff_img.copy()
    cv2.normalize(diff_img, norm_img, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8UC1)
    cv2.imshow("norm", norm_img)

    thr_img = threshold(norm_img, 230, 0, cv2.THRESH_TRUNC)
    cv2.imshow("thresh", thr_img)
    cv2.normalize(thr_img, thr_img, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8UC1)
    cv2.imshow("normed_thresh", thr_img)
    cv2.waitKey()
    return thr_img

def dilation(image, kernel_size=(5,5), iterations = 1):
    kernel = np.ones(kernel_size, np.uint8)
    return cv2.dilate(image, kernel, iterations)

def erosion(image, kernel_size=(5,5), iterations = 1):
    kernel = np.ones(kernel_size, np.uint8)
    return cv2.erode(image, kernel, iterations)
```

Step 3 - Edge detector

Using an edge detector would be useful for finding the egdes of the license plates. Here, Canny is used with the option to set min and max values; those who lie between these two thresholds are classified edges or non-edges based on their connectivity. Canny is used because it automatically removes noises first, unlike Sobel and Laplacian. The Canny edge detector in OpenCV is using 5x5 Gaussian filter to reduce noise and non-maximum suppression to remove unwanted pixels.

In [54]:
```
def edge_detector(image, minval=100, maxval=150):
    return cv2.Canny(image, minval, maxval)
```

Step 4 - Finding the license plate

After the preprocessing is done, we can draw countours to identify the license plates.

The find_countours method takes the preprocessed image and the original image and find countours in the image. This is represented as a list. Then I draw all the possible countours in the image (in green).

The mehtod find_license_plate is then used. The list is here sorted on area size, and only the top n entries are included. Every countour calculates an approximation. It approximates a contour shape to another shape with less number of vertices depending upon the precision I specify, which is the epsilon. An epsilon of 0.05 indicates 5% of the arc length.

I iterate through all the top n countours and check one by one if the contour contains four corners, as that would most likely be the number plate.

If there is four corners, it is checked agains several other methods. Some countours marked the whole image as a big box, which gave a perfect square. To avoid this from being the "best" rectangle, countour_is_whole_image is used. Some countours had four corners, but was more vertical than horizontal. Given that all license plates are horisontal rectangles, we could also exclude these countours using is_vertical_countour which calculates of the vertical lines are longer than the horizontal ones. Some circular approximations where also marked with four corners (strange enough), and to avoid this I calculated the difference in height and width to be sure I did not mark a circle as the license plate.

I also included a method for convex hull. The convex hull function checks a curve for convexity defects and corrects it.

In [55]:

```
def countour_is_whole_image(countour, image):
    left_upper_corner_width = countour[0][0][0]
    left_upper_corner_height = countour[0][0][1]
    right_lower_corner_width = countour[2][0][0]
    right_lower_corner_height = countour[2][0][1]
    if left_upper_corner_width < 20 \
        and left_upper_corner_height < 20 \
        and image.shape[0] - right_lower_corner_height < 20 \
        and image.shape[1] - right_lower_corner_width < 20:
        return True
    return False

def is_vertical_countour(countour):
    left_upper_corner_width = countour[0][0][0]
    right_upper_corner_width = countour[3][0][0]
    horizontally = abs(right_upper_corner_width - left_upper_corner_width)

    left_upper_corner_height = countour[0][0][1]
    left_lower_corner_height = countour[3][0][1]
    vertically = abs(left_lower_corner_height - left_upper_corner_height)
    if vertically > horizontally:
        return True
    return False

def is_rectangle(countour):

    left_upper_corner_height = countour[0][0][0]
    right_upper_corner_height = countour[1][0][0]
    left_upper_corner_width = countour[0][0][1]
    left_lower_corner_width = countour[3][0][1]

    diff_height = abs(right_upper_corner_height - left_upper_corner_height)
    diff_width = abs(left_lower_corner_width - left_upper_corner_width)
    print(diff_height, diff_width)
    if diff_height < 20 and diff_width < 20:
        return True
    return False

def convex(cnt):
    return cv2.convexHull(cnt)

def find_license_plate(image, countours, image_name, top_n=20):
    countour = sorted(countours, key = cv2.contourArea, reverse = True)[:top_n]
    screen_countour = None #store the number plate contour
    copied_image = image.copy()

    for c in countour:
        copy = copied_image.copy()
        copy2 = copy.copy()
        copy3 = copy.copy()
        cv2.drawContours(copy, [c], -1, (0, 255, 0), 3)

        #con = convex(c)
        #cv2.drawContours(copy2, [con], -1, (0, 255, 0), 3)
        #cv2.waitKey()

        perimeter = cv2.arcLength(c, True)
        approximation = cv2.approxPolyDP(c, 0.05*perimeter, True)
        if len(approximation) == 4: # Contures with 4 corners
            screen_countour = approximation
            cv2.drawContours(copy3, [screen_countour], -1, (0, 255, 0), 3)
            #cv2.imshow("passed", copy3)
            #cv2.waitKey()

            #if not is_rectangle(approximation): continue
            if countour_is_whole_image(screen_countour, copied_image): continue
            #if is_vertical_countour(screen_countour): continue
            break

    cv2.drawContours(copied_image, [screen_countour], -1, (0, 255, 0), 3)
    return copied_image


def find_countour(original_image, preprocessed_image, image_name, color_tuple=(0,255,0)):
    countour, _ = cv2.findContours(preprocessed_image.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    resized = resize(original_image)
    cropped = crop(resized)
    cropped_duplicate = cropped.copy()

    cv2.drawContours(cropped_duplicate, countour, -1, color_tuple, thickness=3)
    cv2.imshow("All countours", cropped_duplicate)
    #cv2.imwrite('./all_countours_'+str(image_name), cropped_duplicate)
    cv2.waitKey()

    return find_license_plate(cropped.copy(), countour, image_name, 10)
```

Step 5 - Run algorithm

I'll use the premade methods from previous steps in preprocessing and locate_license_plate to locate the license plate in image "car1.jpg".

Preprocessing

Doing it this way makes it easy to see and change each step to my need. I could easily add morphing or shadow removal to the preprocessing, and change the order of each step.

Steps in preprocessing for car1

- Convert to grayscale
- Resizing and cropping
  - Working with the images for a long time, I came to the conclusion that on a general basis all images of license plates will have noise close to the image boundary, and that the license plate would almost always be located in the middle of the image, or in the bottom low part.
  - Cropping the image based on this, removes a lot of uninteresting parts from the image that we know for a fact is not the license plate. You could also argue that an image with the license plate close to the image boundaries is a bad sample.
- Blur and Threshold
  - Because this image has quite good lightning and a clear difference in color between car and license plate, the threshold work good for car1. The TOZERO threshold gets the job done here, but BINARY is also an option.

Locate_license_plate

I found that thresholding worked well for some cases, while edge detector was suitable for other images. Therefore the method was divided into either using edge detector or thresholding. Using the two together gave me overall bad results.

Steps in license plate locating for car1

- Blur and Threshold
  - Because this image has quite good lightning and a clear difference in color between car and license plate, the threshold work good for car1. The TOZERO threshold gets the job done here, but BINARY is also an option.
- Finding countours
  - I got really good results with using the basic countour finder. Using the convex hull also gave me basically the same results.

In [61]:
```
def preprocessing(image):
    gray = grayscaled(image)
    resized = resize(gray)
    cropped = crop(resized)
    #no_shadows = remove_shadows(cropped)
    return cropped


def locate_license_plate(image_name, edge=False):
    image = import_image(image_name)
    preprocessed_image = preprocessing(image)

    cv2.imshow("Cropped image", preprocessed_image)
    cv2.imwrite('./preprocessed_'+str(image_name), preprocessed_image)
    cv2.waitKey()

    if edge:
        edges = edge_detector(preprocessed_image)
        cv2.imshow("Edge detector results", edges)
        #cv2.imwrite('./edges_'+str(image_name), edges)
        cv2.waitKey()
        countours = find_countour(image, edges, image_name)
    else:
        blurred = blur(preprocessed_image)
        thresh = threshold(blurred)
        cv2.imshow("Thresholded image", thresh)
        cv2.imwrite('./thresholded_'+str(image_name), thresh)
        cv2.waitKey()
        countours = find_countour(image, thresh, image_name)

    cv2.imshow("Final image with plate detected", countours)
    #cv2.imwrite('./result_'+str(image_name), countours)
    cv2.waitKey()
```

Results

Original image

Preprocessed image

Here we see that the image is cropped and we have removed parts of the picture that is not interesting for our task, like the Maserati logo etc.

Thresholded image

After blurring and using TOZERO threshold, it is significantly easier to detect the license plate

All countours

Here is a representation of all the countours the algorithm found in the image

Final result

The final result is shown with the best fitting approximated countour

Testing on the other cars

The same algorithm is now performed on all the other images. It turns out this detects three cars in total.

In [13]:
```
cars = ['car1.jpg', 'car2.jpg', 'car3.jpg', 'car4.jpg', 'car5.jpg', 'car6.jpg', 'car7.png', 'car8.jpg']

for car in cars:
    locate_license_plate(car)
```

Results

Including car 1, this algorithm detects car2 and car4 as well.

Original image

Final result

Success

Fail

Success

Fail

Fail

Fail

Fail

Testing out other methods

By changing the algorithm, I can obtain better results for some cars. For car 6 and car 7, combining the `remove_shadow` using the **Canny edge detector** is the way to go.

The `remove_shadow` step in now included after the image is cropped, like so:

```
def preprocessing(image):
    gray = grayscaled(image)
    resized = resize(gray)
    cropped = crop(resized)
    no_shadows = remove_shadows(cropped) # The change
    return no_shadows
```

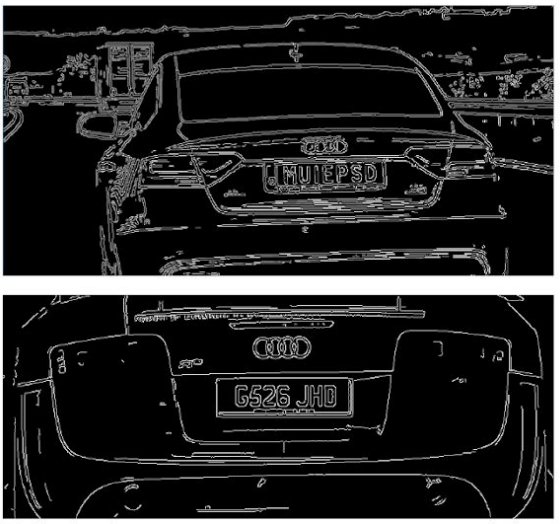
The other cars

Provided the methods I've implemented, I've unsuccessfully managed to detect the license plate for the other cars: car3, car5 and car8. See more in discussion.

In [62]:
```
locate_license_plate('car1.jpg')
#locate_license_plate('car7.png', edge=True)
```

Results car 6 and car 7

Result from the edge detector



Final result

Discussion

Results

This is not a general (enough) method for detecting all license plates out there. I implemented the techniques we learned and discussed in class, but there is room for improvement. You could always keep tweeking some parameters like kernel size or epsilon to gain better results, but I felt applying the different processing methods and see what they did was more educational than finding the perfect sweetspot for some values and watch how that affected the overall result.

There were many trials and errors, and in the end the simple solution with cropping, blurring and threshold was the one giving best results for most pictures at a time.

Challenges with lisence plate detection

Working with this assignment gives a good understanding to what kind of challenges there is in license plate detection.

Colors on signs: Out of the eight images provided, there are many different signs. Firstly they come in different colors; white, yellow and black. Especially the black sign on a red/darker car is hard to work with. When the license plate is the same color as the car/background, thresholding will normally do no good. My method is therefor not very good for the red Ferrari (car5).

Lightning and shadows: The reason I created a `remove_shadows` method was because it became clear that the shadows in the images was creating a lot of trouble. Like discussed in class, thresholding is a very good method if it is done in environment with **controlled illumination**, which is mostly not the case for these kind of pictures. The edge detector was not that good either on the images with a lot of shadows, like car3.

Size and resolution: There is no guarantee that the samples are of high quality. The images provided here are all in different sizes, either closeups of the plate like car1 and car2, but also images with a lot of unimportat infomation. That is why I chose to resize and crop the images, and assume that all license plates will probably be located in the cropped area. While running my algorithm, the `find_countours` method detected the rear or front window as possible license plates, because they were rectangles. By cropping out the top of the car, you reduce the chance of this.

Alternative methods

Template matching: An alternative method is template matching. I was long considering this as an alternative, but chose not to implement it for several reasons. First of all, I would have to choose one sample to match with. If that was a white license plate, it would be hard to detect the black one. Also the ratio of length and width of the license plates are varying in the samples given, and the images are all different sizes. I think template matching would be great if it was given that all plates were in one format.

Classification of plates: Another method is to classify the images based on their characteristics, eg. all white plates are treated in some way, white yellow plates are treated in another way. That would prove for a more complex and probably better algorithm but also way more time consuming to create.

Enhance the countour algorithm: I faced some problems with the countour algorithm. Eg. for car7, it found a countour around the license plate using my standard algorithm, but for some reason it was not considered an approximation with four corners. Understanding and tweeking the contour algorithm could prove better results as well. There were a lot of times when I think the preprocessing did a good job, but the countours were dissapointing. Another way could be to fill the top_n countours and paint that on the image, and then find the plates. Unfortunately, I did not find a good way to achieve this.

Disclaimer

Part 1 of this assignment was written using Jupyter Notebooks for Python. There was a problem exporting the Notebook to PDF, therefore I had to screenshot the results and put them here as pictures. I will send the notebook files and the html file to the tutor after submitting the PDF.

I learnt from my mistake and part 2 is written in Pycharm Editor.

The code in text format

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

def import_image(image):
    """Imports the given image in colors"""
    return cv2.imread(image, cv2.IMREAD_COLOR)

def grayscaled(image):
    """Turns an image into a grayscaled image"""
    return cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

def threshold(image, thresh=170, maxval=255, threshold_type=cv2.THRESH_TOZERO):
    ret, image_thresh = cv2.threshold(image, thresh, maxval, threshold_type)
    return image_thresh

def adaptive_threshold(image, maxval=255,
adaptive_threshold_type=cv2.ADAPTIVE_THRESH_GAUSSIAN_C, threshold_type=cv2.THRESH_TOZERO,
block_size=11, c=2):
    return cv2.adaptiveThreshold(image, maxval, adaptive_threshold_type, threshold_type, block_size, c)

def blur(image, kernel_tuple=(5,5)):
    return cv2.GaussianBlur(image, kernel_tuple, 0)

def resize(image, size=None, scale_x=2, scale_y=2, interpolation=cv2.INTER_LINEAR):
    return cv2.resize(src=image, dsize=size, fx=scale_x, fy=scale_y, interpolation=interpolation)

def crop(image):
    image_tuple = image.shape
    height, width = image_tuple[0], image_tuple[1]
    return image[height*2//10 : height*9//10, width//10 : width*9//10]

def morph_opening(image, kernel_size=(5,5)):
    kernel = np.ones(kernel_size, np.uint8)
    return cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)

def morph_closing(image, kernel_size=(5,5)):
    kernel = np.ones(kernel_size, np.uint8)
    return cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)

def remove_shadows(image):
    dilated_img = cv2.dilate(image, np.ones((5,5), np.uint8))
    cv2.imshow("dil", dilated_img)

    blurred_img = blur(dilated_img, (3,3))
    cv2.imshow("blur", blurred_img)
```

```
    diff_img = 255 - cv2.absdiff(image, blurred_img)
    cv2.imshow("diff", diff_img)

    norm_img = diff_img.copy()
    cv2.normalize(diff_img, norm_img, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX,
dtype=cv2.CV_8UC1)
    cv2.imshow("norm", norm_img)

    thr_img = threshold(norm_img, 230, 0, cv2.THRESH_TRUNC)
    cv2.imshow("thresh", thr_img)
    cv2.normalize(thr_img, thr_img, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX,
dtype=cv2.CV_8UC1)
    cv2.imshow("normed_thresh", thr_img)
    cv2.waitKey()
    return thr_img

def dilation(image, kernel_size=(5,5), iterations = 1):
    kernel = np.ones(kernel_size, np.uint8)
    return cv2.dilate(image, kernel, iterations)

def erosion(image, kernel_size=(5,5), iterations = 1):
    kernel = np.ones(kernel_size, np.uint8)
    return cv2.erode(image, kernel, iterations)

def edge_detector(image, minval=100, maxval=150):
    return cv2.Canny(image, minval, maxval)

def countour_is_whole_image(countour, image):
    left_upper_corner_width = countour[0][0][0]
    left_upper_corner_height = countour[0][0][1]
    right_lower_corner_width = countour[2][0][0]
    right_lower_corner_height = countour[2][0][1]
    if left_upper_corner_width < 20 \
        and left_upper_corner_height < 20 \
        and image.shape[0] - right_lower_corner_height < 20 \
        and image.shape[1] - right_lower_corner_width < 20:
        return True
    return False

def is_vertical_countour(countour):
    left_upper_corner_width = countour[0][0][0]
    right_upper_corner_width = countour[3][0][0]
    horizontally = abs(right_upper_corner_width - left_upper_corner_width)

    left_upper_corner_height = countour[0][0][1]
    left_lower_corner_height = countour[3][0][1]
    vertically = abs(left_lower_corner_height - left_upper_corner_height)
    if vertically > horizontally:
        return True
    return False

def is_rectangle(countour):

    left_upper_corner_height = countour[0][0][0]
```

```
    right_upper_corner_height = countour[1][0][0]
    left_upper_corner_width = countour[0][0][1]
    left_lower_corner_width = countour[3][0][1]

    diff_height = abs(right_upper_corner_height - left_upper_corner_height)
    diff_width = abs(left_lower_corner_width - left_upper_corner_width)
    print(diff_height, diff_width)
    if diff_height < 20 and diff_width < 20:
        return True
    return False

def convex(cnt):
    return cv2.convexHull(cnt)

def find_license_plate(image, countours, image_name, top_n=20):
    countour = sorted(countours, key = cv2.contourArea, reverse = True)[:top_n]
    screen_countour = None #store the number plate contour
    copied_image = image.copy()

    for c in countour:
        copy = copied_image.copy()
        copy2 = copy.copy()
        copy3 = copy.copy()
        cv2.drawContours(copy, [c], -1, (0, 255, 0), 3)

        #con = convex(c)
        #cv2.drawContours(copy2, [con], -1, (0, 255, 0), 3)
        #cv2.waitKey()

        perimeter = cv2.arcLength(c, True)
        approximation = cv2.approxPolyDP(c, 0.05*perimeter, True)
        if len(approximation) == 4: # Contures with 4 corners
            screen_countour = approximation
            cv2.drawContours(copy3, [screen_countour], -1, (0, 255, 0), 3)
            #cv2.imshow("passed", copy3)
            #cv2.waitKey()

            #if not is_rectangle(approximation): continue
            if countour_is_whole_image(screen_countour, copied_image): continue
            #if is_vertical_countour(screen_countour): continue
            break

    cv2.drawContours(copied_image, [screen_countour], -1, (0, 255, 0), 3)
    return copied_image


def find_countour(original_image, preprocessed_image, image_name, color_tuple=(0,255,0)):
    countour, _ = cv2.findContours(preprocessed_image.copy(), cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)
    resized = resize(original_image)
    cropped = crop(resized)
    cropped_duplicate = cropped.copy()

    cv2.drawContours(cropped_duplicate, countour, -1, color_tuple, thickness=3)
```

```
    cv2.imshow("All countours", cropped_duplicate)
    #cv2.imwrite('./all_countours_'+str(image_name), cropped_duplicate)
    cv2.waitKey()

    return find_license_plate(cropped.copy(), countour, image_name, 10)

def preprocessing(image):
    gray = grayscaled(image)
    resized = resize(gray)
    cropped = crop(resized)
    #no_shadows = remove_shadows(cropped)
    return cropped


def locate_license_plate(image_name, edge=False):
    image = import_image(image_name)
    preprocessed_image = preprocessing(image)

    cv2.imshow("Cropped image", preprocessed_image)
    cv2.imwrite('./preprocessed_'+str(image_name), preprocessed_image)
    cv2.waitKey()

    if edge:
        edges = edge_detector(preprocessed_image)
        cv2.imshow("Edge detector results", edges)
        #cv2.imwrite('./edges_'+str(image_name), edges)
        cv2.waitKey()
        countours = find_countour(image, edges, image_name)
    else:
        blurred = blur(preprocessed_image)
        thresh = threshold(blurred)
        cv2.imshow("Thresholded image", thresh)
        cv2.imwrite('./thresholded_'+str(image_name), thresh)
        cv2.waitKey()
        countours = find_countour(image, thresh, image_name)

    cv2.imshow("Final image with plate detected", countours)
    #cv2.imwrite('./result_'+str(image_name), countours)
    cv2.waitKey()

cars = ['car1.jpg', 'car2.jpg', 'car3.jpg', 'car4.jpg', 'car5.jpg', 'car6.jpg', 'car7.png', 'car8.jpg']

for car in cars:
    locate_license_plate(car)

locate_license_plate('car1.jpg')
#locate_license_plate('car7.png', edge=True)
```

Assignment 1 - Pantograph cable detector

Description of task

You are provided with a video of a pantograph which provides the electrical power to a train (Eric2020.mp4). The electrical cable slides across the carbon brush of the pantograph in a zig zag manner. Your task is to graph the position of the power cable on the pantograph over time from the video. Note that both the suspension cable and the power cable are visible and that we only want to track the power cable. Note further that at some times in the video there are two or more power cables visible. We only want to track the lowest cable which is powering the train. See if you can work out a way to separate the cables.

Marking scheme

- Coding of a solution to locate intersection of power cable and pantograph, description and explanation of method, related images and graphs showing processing of subsequent frames (5 marks)
- Describe how the power cable and suspension cable might be differentiated (2 marks)
- Comment on the many challenges to pantograph detection including lighting, day night vision, rain, clouds etc and suggest possible solutions. (3 marks)

Overview of solution

We want to find a way to locate the intersection between the power cable and the pantograph. This could be for industrial purposes where you can detect if there is something wrong with the connection between the two.

My solution will consist of an object detector to find the pantograph, and then I've used image processing to separate the power cable from the suspension cable and marked the intersection. Then I plot the x-position of the intersection for each frame in a graph.

Step 1 - Imports and libraries

I've decided to use OpenCV library for image processing, Matplotlib for creating graphs and Numpy for mathematical and array operations.

Video is imported using the VideoCapture from OpenCV and played frame by frame in the `play_video` method. See comments in code for how to use the video player.

The code:

```python
import sys
import numpy as np
import cv2
import matplotlib.pyplot as plt


def import_video(videofile):
    """Returns the given videofile as a VideoCapture"""
    return cv2.VideoCapture(videofile)


def play_video(video):
    """
    Simple method to play the given video.
    Will stop if no more frames are found or if 'q' is pressed on keyboard
    """
    while video.isOpened():
        ok, frame = video.read()
        if ok:
            cv2.imshow('frame', frame)
            if cv2.waitKey(25) & 0xFF == ord('q'):
                break
        else:
            break

    # Clean up after playing video
    video.release()
    cv2.destroyAllWindows()
```

Step 2 - Object tracking

We are to track the pantograph, which is a moving object. Therefore it would be a good idea to use an object tracker from OpenCV.

An object tracker sets an initial Region of Interest (ROI) by a bounding rectangle. Then for each frame it looks at the difference (in eg. Euclidean distance) from the last frame to the new frame, and adjust accordingly if the object has moved. A good object tracker is fast and able to pick up objects it has “lost” in between frames. There are many different object tracker algorithms offered by OpenCV.

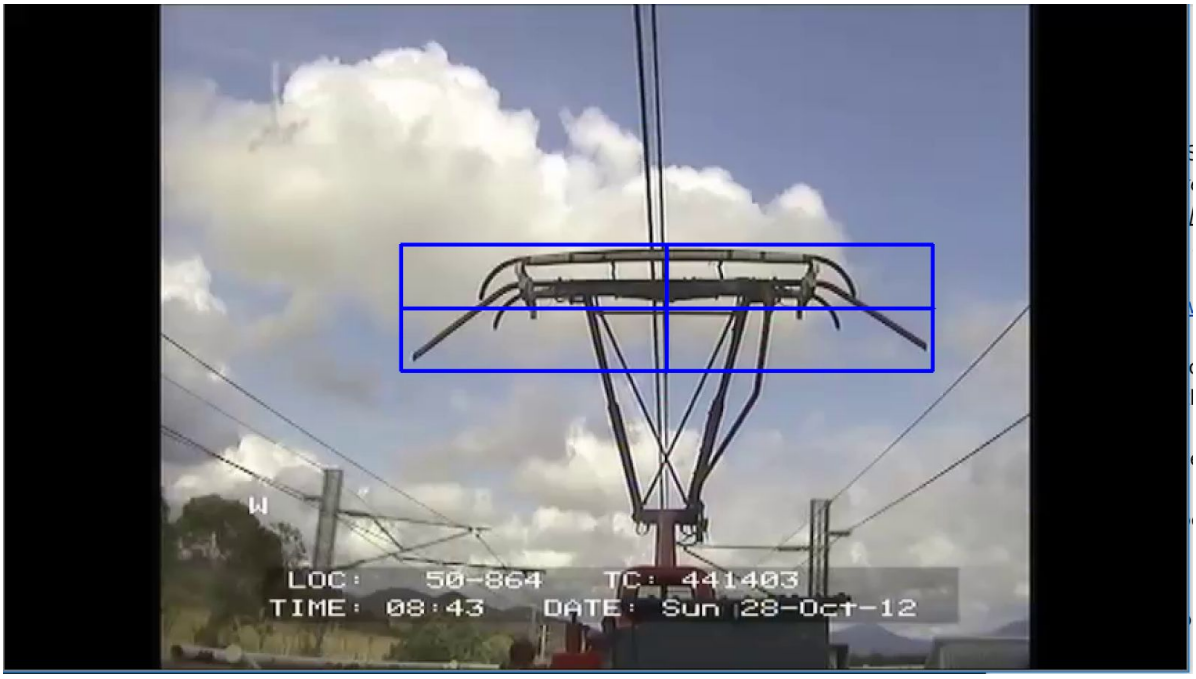
Because I’m doing many operations on the image - as well as plotting the graph - I wanted the video to run smoothly with a high frame rate. Therefore I went for the Minimum Output Sum of Squared Error (MOSSE) object tracker.

About the MOSSE object tracker:
"MOSSE tracker is robust to variations in lighting, scale, pose, and non-rigid deformations. [...] MOSSE tracker also operates at a higher fps (450 fps and even more). To add to the positives, it is also very easy to implement, is as accurate as other complex trackers and much faster"

- https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/

I also tested Boosting, KCF and CSRT trackers, but none if them was as fast as MOSSE, and they gave basically the same results when it came to the tracking.

This is implemented by reading the first frame of the video, and then the user sets the ROI of the image. The ROI is set as covering the top part of the pantograph in the crossline between the pantograph and the cables, like in the picture.



The ROI is selected by dragging the mouse over the frame, and when set you can click 'Enter' or 'Spacebar' tab on keyboard to start the tracking.

The code:

```
31  def tracking(video):
32      """
33      Executes the tracking of the cable
34      MOSSE tracker from OpenCV is used to locate the pantograph
35      """
```

```
43      # Set up the tracker
44      tracker = cv2.TrackerMOSSE_create()
45
46      try:
47          # Stops program if there is any problem with the video
48          if not video.isOpened():
49              sys.exit()
50
51          # Reads the first frame so we can create the bounding box
52          ok, frame = video.read()
53          if not ok:
54              sys.exit()
55
56          # selectROI lets us select an area in the first frame that is the
57          # region of interest
58          bounding_box = cv2.selectROI(
59              "Frame", frame, fromCenter=False, showCrosshair=True)
60
61          # Initialize the MOSSE tracker
62          tracker.init(frame, bounding_box)
63          frame_number = 0
64
```

Step 3 - Processing of frame and finding the cable

I want to detect the power cable, which is the thickest of the two. To do this i do some preprocessing of the image.

There is a lot going on in the video, but we are only interested in the pantograph and the cables. The cables are located in the area above the object tracker frame. Therefore I crop out this part of the video and work with those frames only.
The cropped part is then sent to a `find_contour` function which process the image and finds the largest cable among the two and then creates an approximation to that contour. Because the contour is actually found on the cropped frame, it will not match when I draw the contour on the original frame. Therefore I implemented a transpose method as well placing the contours correctly.

The intersection point is then found as the bottom pixel of the cable line, where it collides with the pantograph object tracking. It is drawn as a circle in the video.

The code:

```python
def crop(frame, bounding_box):
    """
    Crops the frame above the bounding box of the object tracking
    """
    copied = frame.copy()
    cropped = copied[2: int(bounding_box[1]), int(
        bounding_box[0]): int(bounding_box[0]) + int(bounding_box[2])]
    return cropped
```

```python
        # Run loop as long as there are more frames in the video
        while True:
            ok, frame = video.read()
            if not ok:
                break

            # Update the tracker and the bounding box for each frame
            ok, bounding_box = tracker.update(frame)
            if ok:
                # Crop the frame above the bounding box, find the contour
                cropped = crop(frame, bounding_box)
                contour = find_contour(cropped)
                transposed_contour = transpose(contour, bounding_box)

                # Find the intersection point and draw it on the frame
                intersection_point = (contour[1][0][0], contour[1][0][1])

                cv2.drawContours(
                    frame, [transposed_contour], -1, (0, 255, 0), thickness=3)
                cv2.circle(
                    frame, intersection_point, 8, (128, 0, 255), -1)
                cv2.imshow('frame', frame)
```

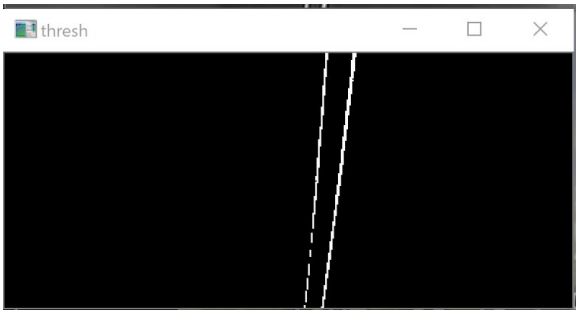
```python
def find_contour(frame):
    """
    Process each frame and draws a contour over the cable
    """
    processed = processing(frame)
    contours, _ = cv2.findContours(processed, cv2.RETR_TREE,
                                   cv2.CHAIN_APPROX_SIMPLE)

    # The cable is chosen as the top-1 contour sorted on biggest area
    contour = sorted(contours, key=cv2.contourArea, reverse=True)[:1]

    # Creates an approximation of the cable line with fewer points
    epsilon = 0.1 * cv2.arcLength(contour[0], True)
    approx = cv2.approxPolyDP(contour[0], epsilon, True)

    return approx
```

```
107     def transpose(contour, bounding_box):
108         """
109         Because the contour of the pantograph is found in a cropped frame,
110         we have to transpose the contour values for height and width to fit
111         the original frame
112         """
113         c = contour
114         c[0][0][0] += bounding_box[0]
115         c[1][0][0] += bounding_box[0]
116         return c
```

Results from the preprocessing

First I make a grayscale of the image, and then creates a mask.

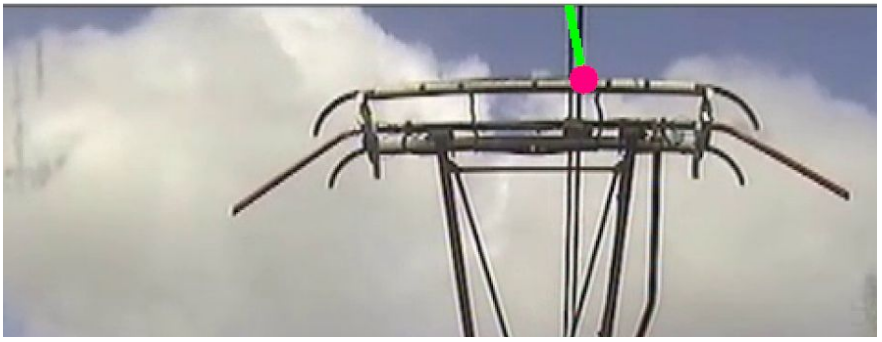
A simple Binary Threshold is then applied to the mask with a relatively high threshold.



Further I create two different kernels for dilation and erosion, size 7x7 and 5x5 respectively. I did this instead of using closing directly because I wanted the dilation to be more aggressive than the erosion. I also experimented a bit with number of iterations as well. So doing it this way gives me more freedom to adjust the parameters myself. The dilation helps making the cables thicker and fill gaps in the line.

Because the cables are often directly above each other, the dilation makes them seem merged into one cable when they are close.

Here is how the line is detected when the lines are to close. Without the erosion the green line crosses both of the cables and give small errors.



The erosion helps separate the two cables as it makes the cables thinner, but at this point the lines are complete and do not lack any pixels as they did after the thresholding.

Step 4 - Plot the graph

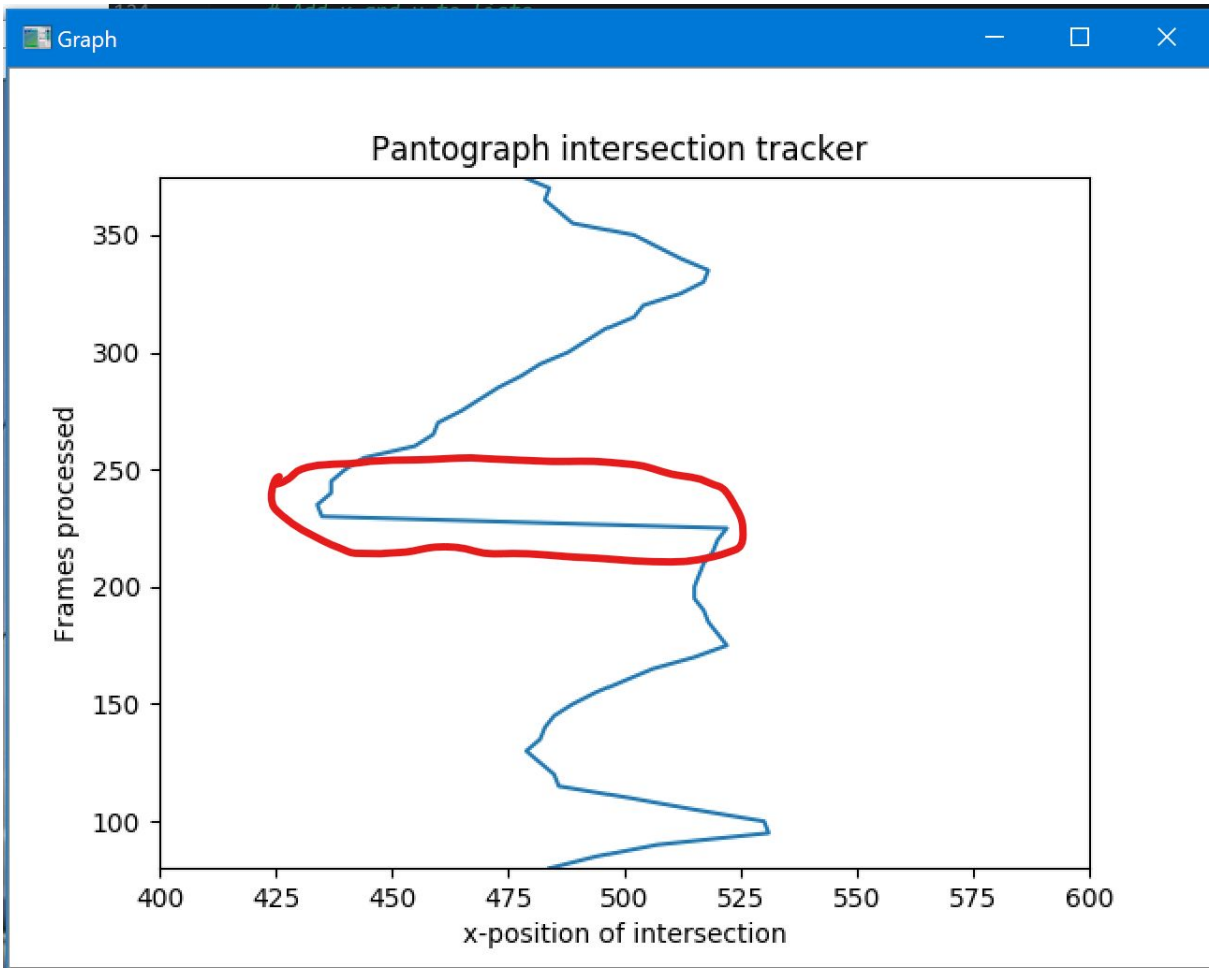
To plot the position of the intersection, I look at the x-position of the intersection and plot it against the number of frames. Only every fifth frame is used to plot the graph, to secure a smoother curve, remove sudden spikes and also for the video to run smoother.

The code:

```
                # Set up plot to call animate() function for every fifth point
                frame_number += 1
                if frame_number % 5 == 0:
                    point = contour[1][0][0]
                    animate(frame_number, xs, ys, point, fig, ax)

                # Quit when no more frames or on pressed 'q'
                if cv2.waitKey(25) & 0xFF == ord('q'):
                    break

        # Clean up
        video.release()
        cv2.destroyAllWindows()

    # Exit if there is an error with leading the video
    except BaseException:
        sys.exit()
```

```
def animate(frame_count, xs, ys, point, fig, ax):
    """
    Creates the graph tracking the position of the cable
    Updates the graph with one point each time it is executed
    """
    # Add x and y to lists
    xs.append(point)
    ys.append(frame_count)

    # Limit x and y lists to 300 items, divide by 5 because we only read every
    # fifth point
    xs = xs[-(300 // 5):]
    ys = ys[-(300 // 5):]

    # Draw x and y lists
    ax.clear()
    ax.set_ylabel("Frames processed")
    ax.set_xlabel("x-position of intersection")
    ax.set_title("Pantograph intersection tracker ")
    if frame_count > 300:
        ax.axis([400, 600, ys[0], ys[-1]])
    else:
        ax.axis([400, 600, 0, 300])
    ax.plot(xs, ys)

    fig.canvas.draw()
    graph = np.fromstring(fig.canvas.tostring_rgb(), dtype=np.uint8, sep='')
    graph = graph.reshape(fig.canvas.get_width_height()[::-1] + (3,))
    graph = cv2.cvtColor(graph, cv2.COLOR_RGB2BGR)
    cv2.imshow("Graph", graph)
```

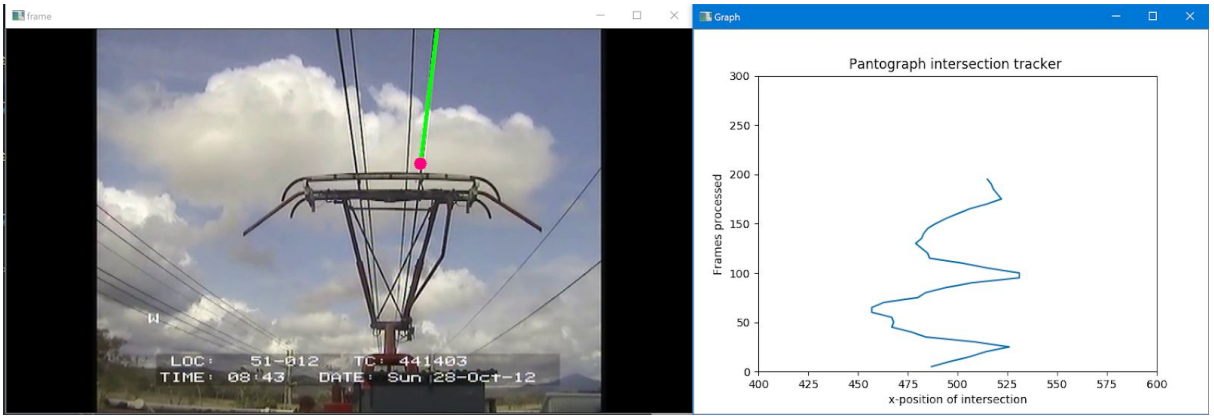
Here is the result of the graph. The marked red region is where there is two power cables and it suddenly changes from one to the other.

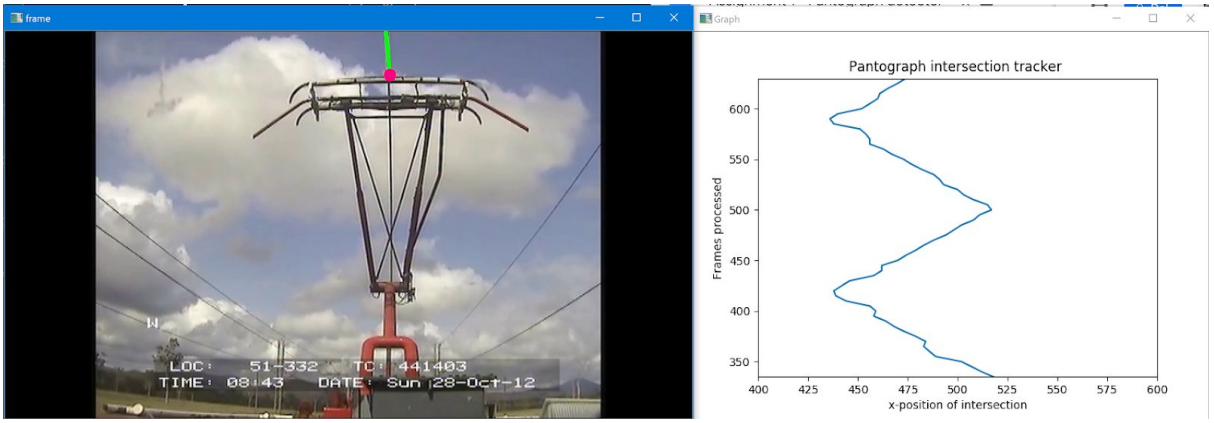


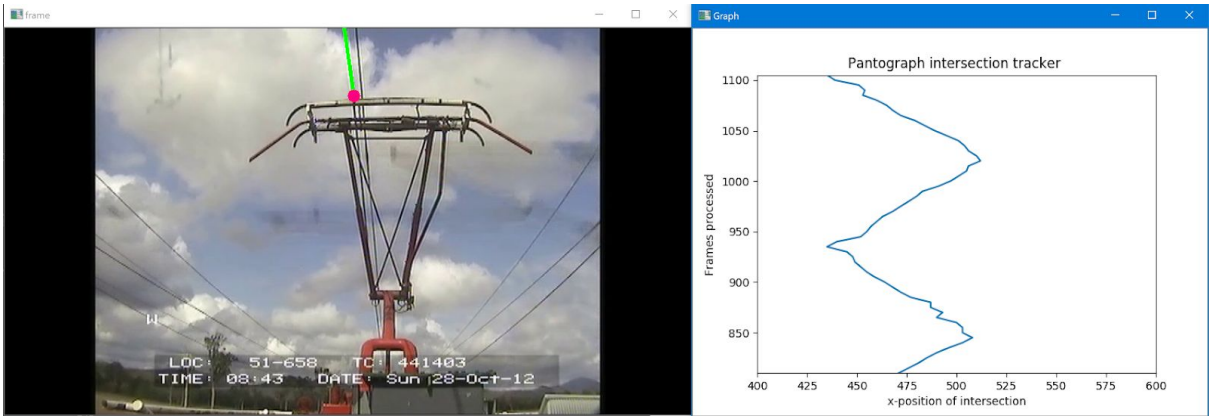
Results

Here are some of the results with both video and graph side by side. The green line follows the power cable and the red dot shows the intersection between the power cable and the pantograph.

The graph is plotted simultaneously and shows the x-position of the intersection per fifth frame.


frame
LOC: 51-012 TC: 441403
TIME: 08:43 DATE: Sun 28-Oct-12
Graph
Pantograph intersection tracker
Frames processed
x-position of intersection


frame
LOC: 51-332 TC: 441403
TIME: 08:43 DATE: Sun 28-Oct-12
Graph
Pantograph intersection tracker
Frames processed
x-position of intersection


frame
LOC: 51-658 TC: 441403
TIME: 08:43 DATE: Sun 28-Oct-12
Graph
Pantograph intersection tracker
Frames processed
x-position of intersection


frame
LOC: 50-904 TC: 441403
TIME: 08:43 DATE: Sun 28-Oct-12
Graph
Pantograph intersection tracker
Frames processed
x-position of intersection

Discussion

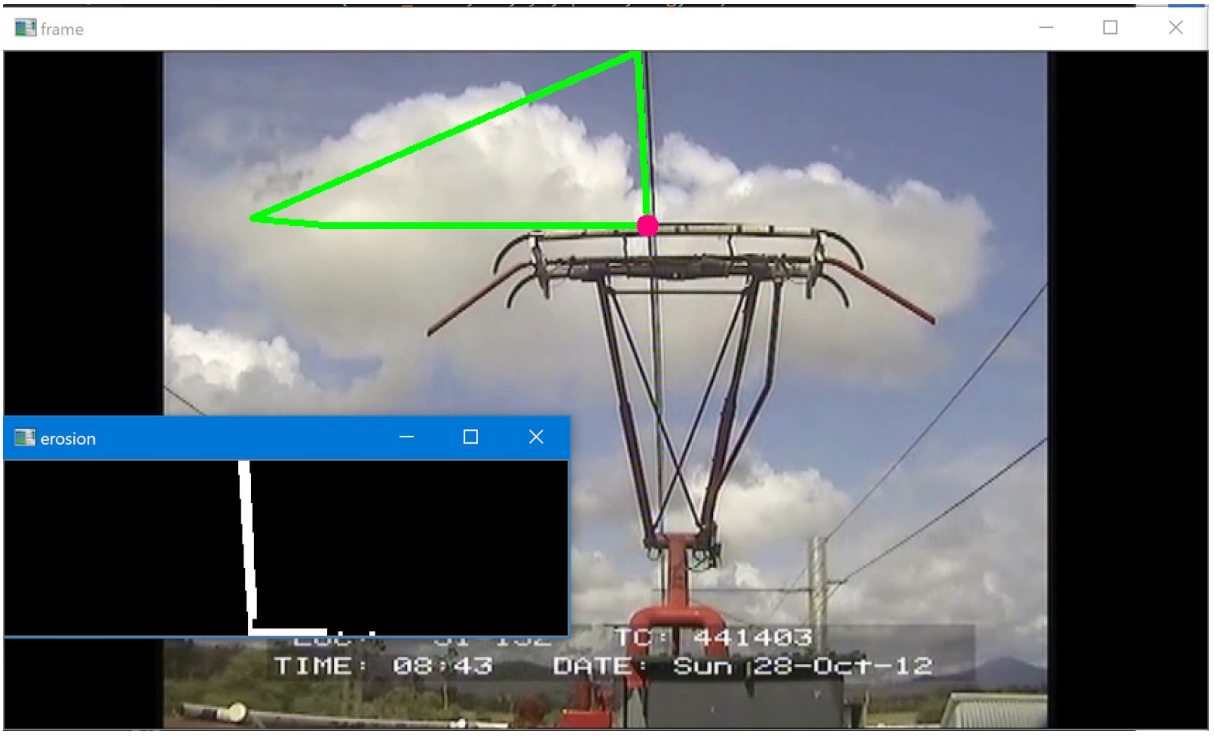
Problems

Some of the errors in this algorithm comes from not differentiating the power cable and the suspension cable correctly. Here I used some image processing techniques and focus on the area size of the cable, to reasonable results. The line detector may be a bit off when the cables are crossing each other. Another approach could be to process the image so that the suspension cable “vanished” after the processing and you are only left with the power cable because it is bigger and more dominant. I believe this would require more tweaking of parameters and may not give a better result.

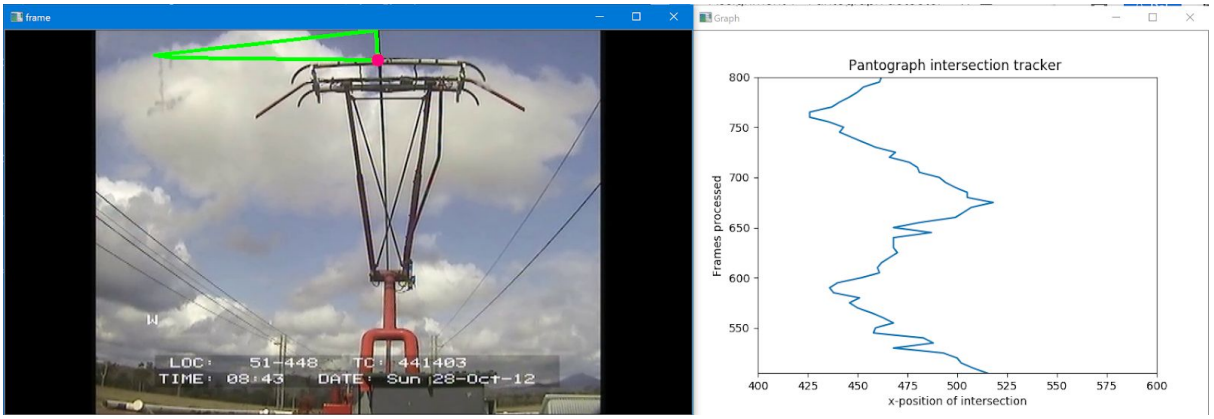
From an image processing view, the background of the cable have some importance. In the video it is cloudy, and the clouds could be detected as lines as well if the preprocessing is not perfect. By using the mask and subtract that from the image combined with a threshold, I successfully removed the background noise from the frame.

Another pain point is when the pantograph stretches out and the processed frame becomes quite small. The object tracker is at this point having some difficulties and that makes the top of the pantograph appear in the processed frame. This leads to the line going crazy.

There we see that the top of the pantograph appears in the erosion frame, which gives the contour finder a problem.



And here we can see how this affects the graph as well, we see much more inconsistency in the x-position.



Another possible pain point is when there is more than two cables. At one point there is two power cables attached and two suspension cables. When they switch out, the program finds the new power cable quite good, and this became less of a problem than I thought. I do not know at which point exactly the train switches from using the first power cable to the next, but I've assumed that when both power cables are attached to the pantograph, they both give power to the train.

Challenges with pantograph detection

Working with this assignment has given me some insights in the challenges of detecting cables and pantographs.

Multiple cables: Like discussed under the "Problems"-section, there could be several cables and hard to know which ones to track.

Lightning and weather: Because trains operate outside, you can encounter all types of lightning due to the weather conditions. Rain could affect the camera lens and give bad video samples. The clouds could be a problem, as well as strong sun altering the lighting. Fog could also be an issue leading to reduced vision for the camera.

Day vs night: At night time, the image processing algorithm would probably be way different. Using lights from the train itself or a night vision camera will give different illuminations.

Alternative methods

I also tried out another method before ending up with my current solution. This algorithm tracked the pantograph and the cable together (not the frame above the tracker with the cable only) using the same object tracker. Then I used probabilistic Hough Transform to find the lines after some processing. My hope was that it would find the top of the pantograph as one horizontal line and the cables as vertical lines, and then find the crossing point of the lines. Unfortunately I was not able to do this,

at it only found the horizontal lines. Therefore this method was dropped, but I kept the code for the Hough Transform.

The probabilistic Hough Transform is good for finding lines and shapes. The Hough transform estimates the probability for a subset of points to belong to the same line. The difference between the normal and the probabilistic version is that you can feed a min_length to set a threshold for the length of each line, removing irrelevant lines. You can also set a min_gap, deciding how big gaps a line can have before it is discarded as a line.

```
185  def hough_transformP(input_frame, min_length=50, min_gap=10):
186      """
187      Probabilistic Hough transform using OpenCV
188      Input frame should be binary, that's why Canny edge detector is used
189      The probabilistic version analyzes lines as subset of points and
190      estimates the probability of these points to belong to the same line
191      """
192      copy = input_frame.copy()
193      edges = cv2.Canny(copy, 70, 150, apertureSize=3)
194
195      lines = cv2.HoughLinesP(edges, 1, np.pi / 180, 100, min_length, min_gap)
196      for x1, y1, x2, y2 in lines[0]:
197          cv2.line(copy, (x1, y1), (x2, y2), (255, 128, 0), 2)
198
199      return copy
```

Another possible method is to implement some way to distinguish the horizontal lines from the vertical ones. Eg. using image gradients like Sobel in the x and y direction to find the pantograph and cables.

All the code in text format

```
import sys
import numpy as np
import cv2
import matplotlib.pyplot as plt


def import_video(videofile):
    """Returns the given videofile as a VideoCapture"""
    return cv2.VideoCapture(videofile)


def play_video(video):
    """
    Simple method to play the given video.
    Will stop if no more frames are found or if 'q' is pressed on keyboard
    """
    while video.isOpened():
        ok, frame = video.read()
        if ok:
            cv2.imshow('frame', frame)
            if cv2.waitKey(25) & 0xFF == ord('q'):
                break
        else:
            break

    # Clean up after playing video
    video.release()
    cv2.destroyAllWindows()


def tracking(video):
    """
    Executes the tracking of the cable
    MOSSE tracker from OpenCV is used to locate the pantograph
    """

    # Initialize the figure for graph plotting
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
```

```
  xs = []
  ys = []

  # Set up the tracker
  tracker = cv2.TrackerMOSSE_create()

  try:
      # Stops program if there is any problem with the video
      if not video.isOpened():
          sys.exit()

      # Reads the first frame so we can create the bounding box
      ok, frame = video.read()
      if not ok:
          sys.exit()

      # selectROI lets us select an area in the first frame that is the
      # region of interest
      bounding_box = cv2.selectROI(
          "Frame", frame, fromCenter=False, showCrosshair=True)

      # Initialize the MOSSE tracker
      tracker.init(frame, bounding_box)
      frame_number = 0

      # Run loop as long as there are more frames in the video
      while True:
          ok, frame = video.read()
          if not ok:
              break

          # Update the tracker and the bounding box for each frame
          ok, bounding_box = tracker.update(frame)
          if ok:
              # Crop the frame above the bounding box, find the contour
              cropped = crop(frame, bounding_box)
              contour = find_contour(cropped)
              transposed_contour = transpose(contour, bounding_box)

              # Find the intersection point and draw it on the frame
              intersection_point = (contour[1][0][0], contour[1][0][1])
```

```
            cv2.drawContours(
                frame, [transposed_contour], -1, (0, 255, 0), thickness=3)
            cv2.circle(
                frame, intersection_point, 8, (128, 0, 255), -1)
            cv2.imshow('frame', frame)

            # Set up plot to call animate() function for every fifth point
            frame_number += 1
            if frame_number % 5 == 0:
                point = contour[1][0][0]
                animate(frame_number, xs, ys, point, fig, ax)

            # Quit when no more frames or on pressed 'q'
            if cv2.waitKey(25) & 0xFF == ord('q'):
                break

    # Clean up
    video.release()
    cv2.destroyAllWindows()

  # Exit if there is an error with leading the video
  except BaseException:
    sys.exit()


def transpose(contour, bounding_box):
  """
  Because the contour of the pantograph is found in a cropped frame,
  we have to transpose the contour values for height and width to fit
  the original frame
  """
  c = contour
  c[0][0][0] += bounding_box[0]
  c[1][0][0] += bounding_box[0]
  return c


def animate(frame_count, xs, ys, point, fig, ax):
  """
  Creates the graph tracking the position of the cable
  Updates the graph with one point each time it is executed
  """
```

```
    # Add x and y to lists
    xs.append(point)
    ys.append(frame_count)

    # Limit x and y lists to 300 items, divide by 5 because we only read every
    # fifth point
    xs = xs[-(300 // 5):]
    ys = ys[-(300 // 5):]

    # Draw x and y lists
    ax.clear()
    ax.set_ylabel("Frames processed")
    ax.set_xlabel("x-position of intersection")
    ax.set_title("Pantograph intersection tracker ")
    if frame_count > 300:
        ax.axis([400, 600, ys[0], ys[-1]])
    else:
        ax.axis([400, 600, 0, 300])
    ax.plot(xs, ys)

    fig.canvas.draw()
    graph = np.fromstring(fig.canvas.tostring_rgb(), dtype=np.uint8, sep='')
    graph = graph.reshape(fig.canvas.get_width_height()[::-1] + (3,))
    graph = cv2.cvtColor(graph, cv2.COLOR_RGB2BGR)
    cv2.imshow("Graph", graph)


def find_contour(frame):
    """
    Process each frame and draws a contour over the cable
    """
    processed = processing(frame)
    contours, _ = cv2.findContours(processed, cv2.RETR_TREE,
                                   cv2.CHAIN_APPROX_SIMPLE)

    # The cable is chosen as the top-1 contour sorted on biggest area
    contour = sorted(contours, key=cv2.contourArea, reverse=True)[:1]

    # Creates an approximation of the cable line with fewer points
    epsilon = 0.1 * cv2.arcLength(contour[0], True)
    approx = cv2.approxPolyDP(contour[0], epsilon, True)
```

```
    return approx


def processing(frame):
    """
    Process one frame of the video using gray scale, masking, threshold,
    dilation and erosion
    """
    image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    image = 255 - image
    _, thresh = cv2.threshold(image, 200, 255, cv2.THRESH_BINARY)
    kernel_dil = np.ones((7, 7), np.uint8)
    kernel_ero = np.ones((5, 5), np.uint8)
    dilation = cv2.dilate(thresh, kernel_dil, iterations=1)
    erosion = cv2.erode(dilation, kernel_ero, iterations=1)

    return erosion


def hough_transformP(input_frame, min_length=50, min_gap=10):
    """
    Probabilistic Hough transform using OpenCV
    Input frame should be binary, that's why Canny edge detector is used
    The probabilistic version analyzes lines as subset of points and
    estimates the probability of these points to belong to the same line
    """
    copy = input_frame.copy()
    edges = cv2.Canny(copy, 70, 150, apertureSize=3)

    lines = cv2.HoughLinesP(edges, 1, np.pi / 180, 100, min_length, min_gap)
    for x1, y1, x2, y2 in lines[0]:
        cv2.line(copy, (x1, y1), (x2, y2), (255, 128, 0), 2)

    return copy


def crop(frame, bounding_box):
    """
    Crops the frame above the bounding box of the object tracking
    """
    copied = frame.copy()
    cropped = copied[2: int(bounding_box[1]), int(
```

```
        bounding_box[0]): int(bounding_box[0]) + int(bounding_box[2])]
    return cropped


def run_tracker(videofile):
    """
    Main method for running the program
    """
    video = import_video(videofile)
    tracking(video)


run_tracker('Eric2020.mp4')
```