Assignment 1

ELEC4630: Image Processing and Computer Vision

William E. G. Kvaale, University of Queensland 30/03/2020

Automatic Number Plate Recognizer

The first of many challenges I first realized I was facing, was the difference in resolution and image clarity (spatial resolution). The distortion (or skewness) in relation the the number plates was also a factor I though would be challenging to address. The variation in lighting conditions did not vary to much, however, the challenge of having different colored number plates got me thinking I would soon be wandering in to HSV- or LAB space.

Different approaches

Before diving into attempting to solve the problem at hand, I did some research on what technologies and approaches other researchers had done. Mainly I sought out older research papers, since most of the fresher papers that I found on Google Scholar was using some Machine Learning (specifically Deep Learning) approaches.

The key takeaway I was took, was that a Python approach combined with the OpenCV library and some sort of morphological approach might be a way to obtain success.

My first approach to the problem of different resolution, was resizing the images. This was done in the pursuit of transforming the data to a somewhat normalized format. I read several papers advising against, and some stating this could help achieve better result in regards to edge detection tasks. I went found that a suitable minimum width and height was respectively 720 pixels and 1280 pixels. See Listing 1.

The interpolation used in resizing, was OpenCV's mystic cv.INTER_AREA. OpenCV states that it is "resampling using pixel area relation. It may be a preferred method for image decimation, as it gives moire'-free results. But when the image is zoomed, it is similar to the INTER_NEAREST method". In layman's term:

1. **Scale down:** is either a bilinear interpolation with (1,0) as coefficients, or more complicated if both the input and output scales, in regards to width and height, are not integers
2. **Scale up:** is equal to linear interpolation and exact linear interpolation based on the number of channels

I also experimented with the lightness of the images, or more concretely I tried a decoding gamma ($\gamma > 1$) on the images, but empirically I found that an encoding gamma (however so slightly) ($\gamma < 1$) improved performance. I set $\gamma = 0.9$ throughout my different approaches.

```python
def checkSize(image):
    (height, width) = image.shape[:2]
    if height < MIN_HEIGHT:
        return imutils.resize(image, height=MIN_HEIGHT,
        inter=cv.INTER_AREA)
    elif width < MIN_WIDTH:
        return imutils.resize(image, width=MIN_WIDTH,
        inter=cv.INTER_AREA)
    else:
        return image
```

Listing 1: checkSize function in Python

To segment the image, I tested several threshold algorithms; binary threshold, tozero threshold, triangle threshold, adaptive thresholding and Otsu's Binarization. The method proving best in terms of segmenting the images the most sufficient was Otsu's. Otsu's finds the optimal threshold value, instead of having it set manually with plain binary thresholding.

On beforehand of applying the threshold, I would smooth the images using one of OpenCV's built in filters;

`cv.blur(), cv.GaussianBlur(), cv.bilateralFilter()`

The one I had the most success with was the Gaussian Blur.

Method №1

1. Resize image
2. Encode image with γ = 0.9
3. Smooth using Gaussian with 5x5 kernel
4. Otsu's threshold to segment image
5. Bitwise AND threshold with original image
6. Run Canny Edge on the preceding output

My initial approach, which achieved 4 out of 8.

Figure 1: Results using №1

Method №2

1. Resize image
2. Encode image with γ = 0.9
3. Smooth using Gaussian with 5x5 kernel
4. Adaptive Gaussian Threshold
   4.1. Block size = 15
   4.2. C = -3
5. Morphological operations
   5.1. kernel = 2x2 rectangular structuring element
   5.2. 1 iteration with closing
   5.3. 1 iteration using opening

My second approach, which achieved 4 out of 8. However, this time it failed on the Ferrari - but managed to locate the numberplate on the Audi A4.

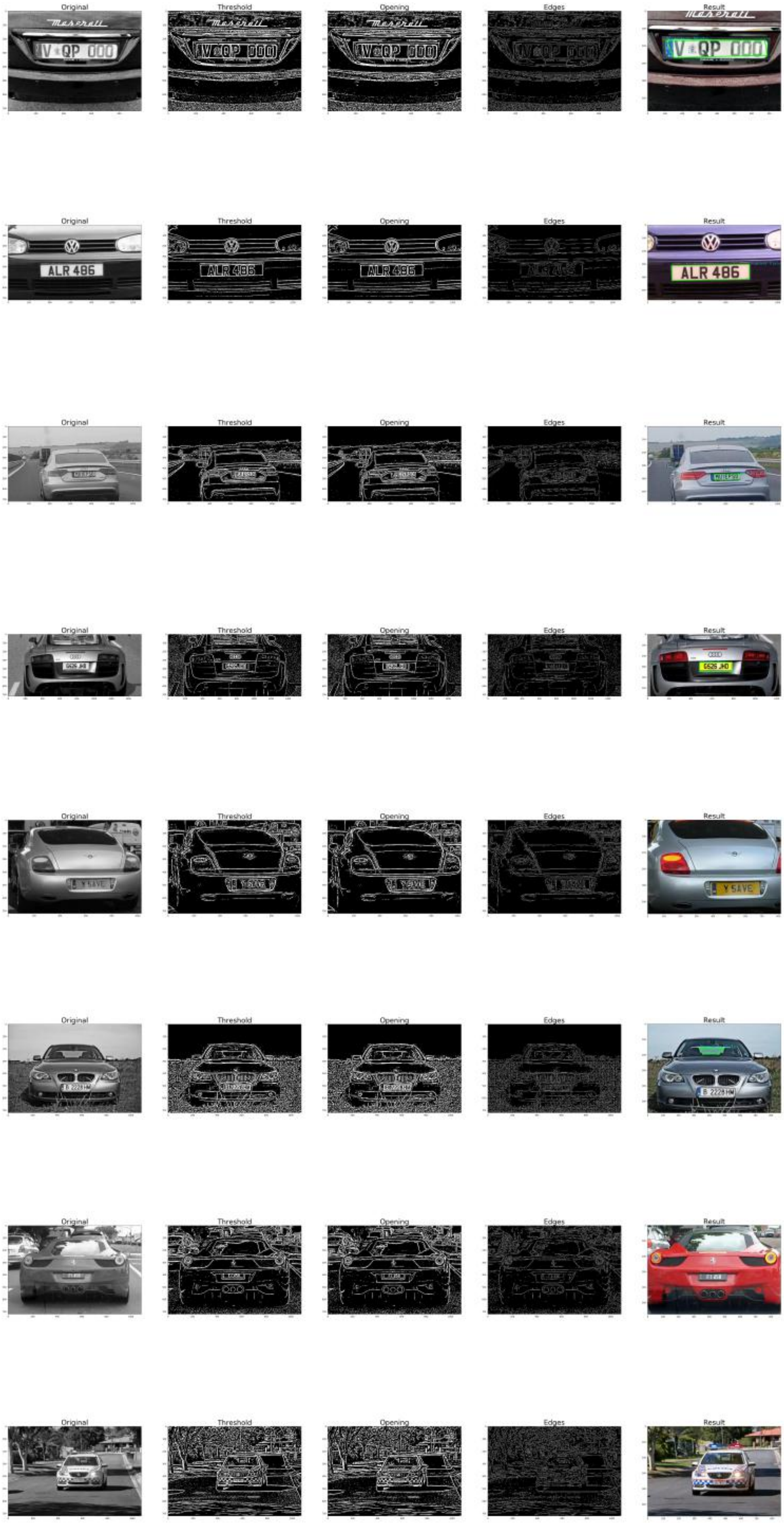

Figure 2: Results using method №2

Method №3

1. Resize image
2. Encode image with γ = 0.9
3. Smooth using Gaussian
    3.1. 7x7 kernel
    3.2. σ_x = 2
4. Adaptive Gaussian Threshold
    4.1. Block size = 15
    4.2. C = -3
5. Bitwise AND threshold with original image

My third and final approach, which achieved 5 out of 8. This also fails on the Ferrari, and the Bentley and Police car.

I never managed to detect the police car accurately, nor the Bentley. The Police car was quite difficult, but could maybe be achieved through template matching (using an exact template). The Bentley I could detect using a different contouring method than all the others, which I found interesting. It was successful by using OpenCV's cv.boundingRect.

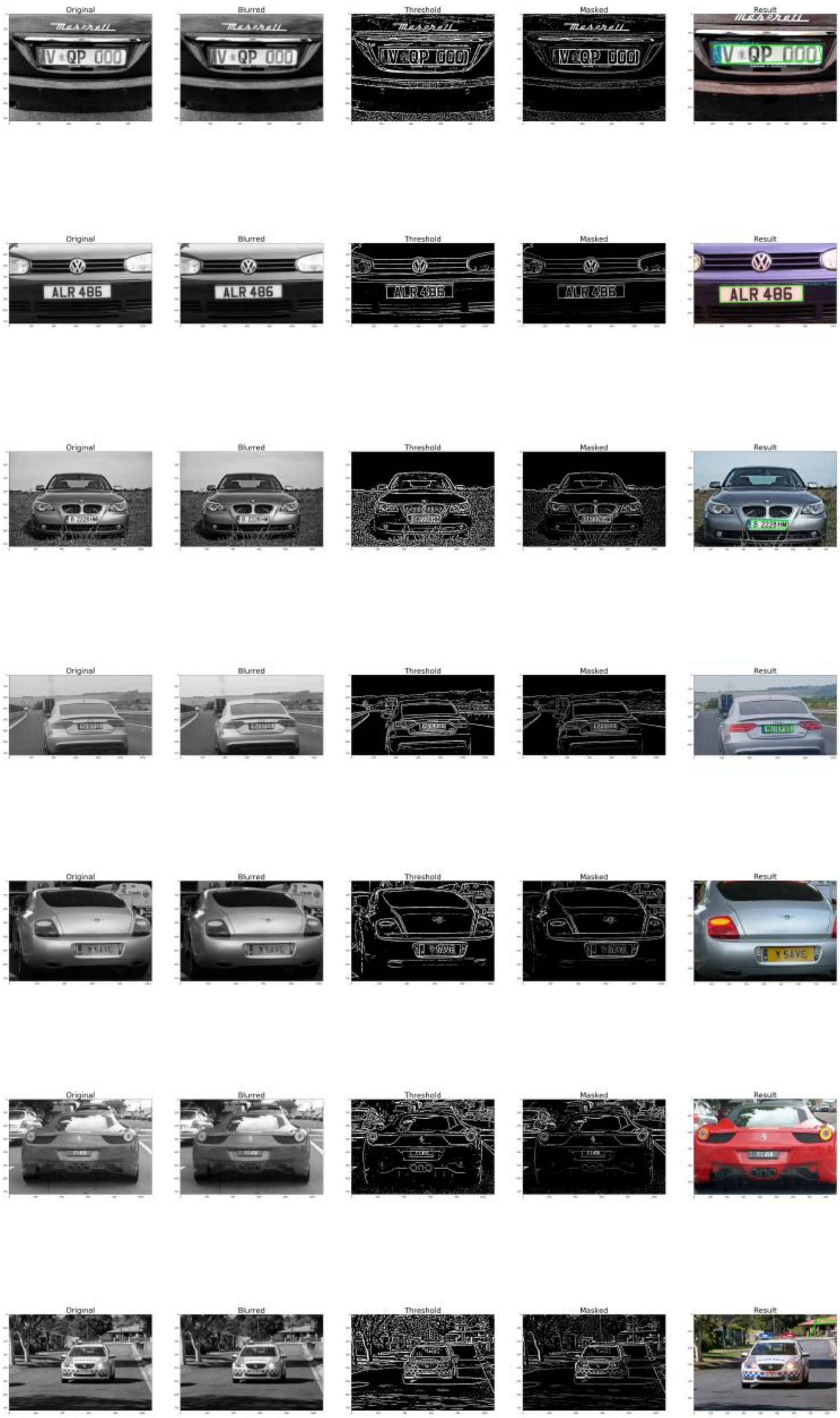

Figure 3: Results using method №3

Pantograph Tracking and Position Graphing

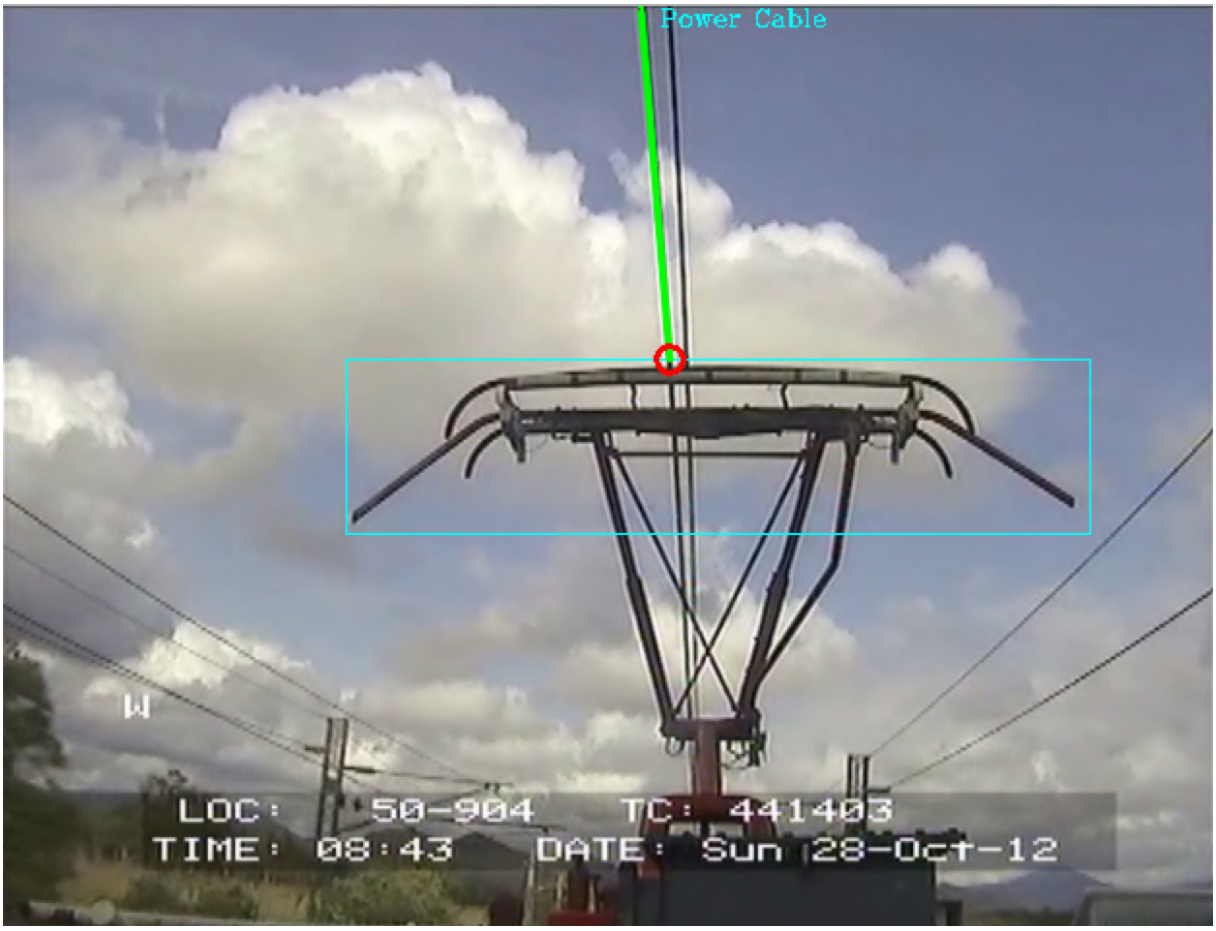

Figure 4: Pantograph tracking using MOSSE

This was my first time ever working with videofiles, and tracking - and I was quite excited upon starting working on this part of the assignment. I quickly realized that working on videofiles is not that different from working with still images, if you handle it frame by frame.

OpenCV delivers several object trackers; BOOSTING, MIL, KCF, TLD, MEDIANFLOW, GOTURN, MOSSE and CSRT. For my case, I found MOSSE being the best fit. MOSSE (Minimum Output Sum of Squared Error) utilizes adaptive correlation and outputs stable correlation filters (as long as it is initialized with one frame - preferably the ROI).

In regards to the differentiate the suspension cable and power cable, I had a rather simple approach. By observation, it would appear thicker, that being having a larger circumference, or being closer to the camera, was not at all time clear. Regardless, I used this as my selection criteria.

Different approaches

By following the lectures, my initial thought was that success could most certainly be achieved by using a standard or probabilistic Hough transformation to find the lines, and the intersections of such lines. Nevertheless, my attempts in implementing this did not bear fruit, and therefore I chose to go down a different path - a morphological approach. I got inspired by OpenCV's tutorial on line detection, and used this as a baseline for my approach to achieve the best results.

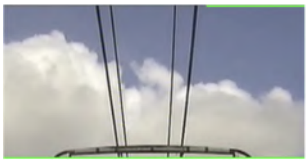

Figure 5: Hough transformation (standard)

Figure 6: Hough transformation (probabilistic)

Even though the probabilistic Hough transformation could yield a good fit, it was highly unstable to noise, such as lines crossing.

Successful method

1. Convert to grayscale
2. Adaptive Gaussian Threshold
   2.1. Block size = 15
   2.2. C = -3
3. Morphological operations
   (a) Vertical structuring element of dimensions 1x16
   (b) Erode 1 iteration
   (c) Dilate 2 iterations
4. Smooth image using bilateral filtering
   (a) diameter = 11
   (b) σ_{color} = 17
   (c) σ_{space} = 17
5. Run Canny Edge on the preceding output

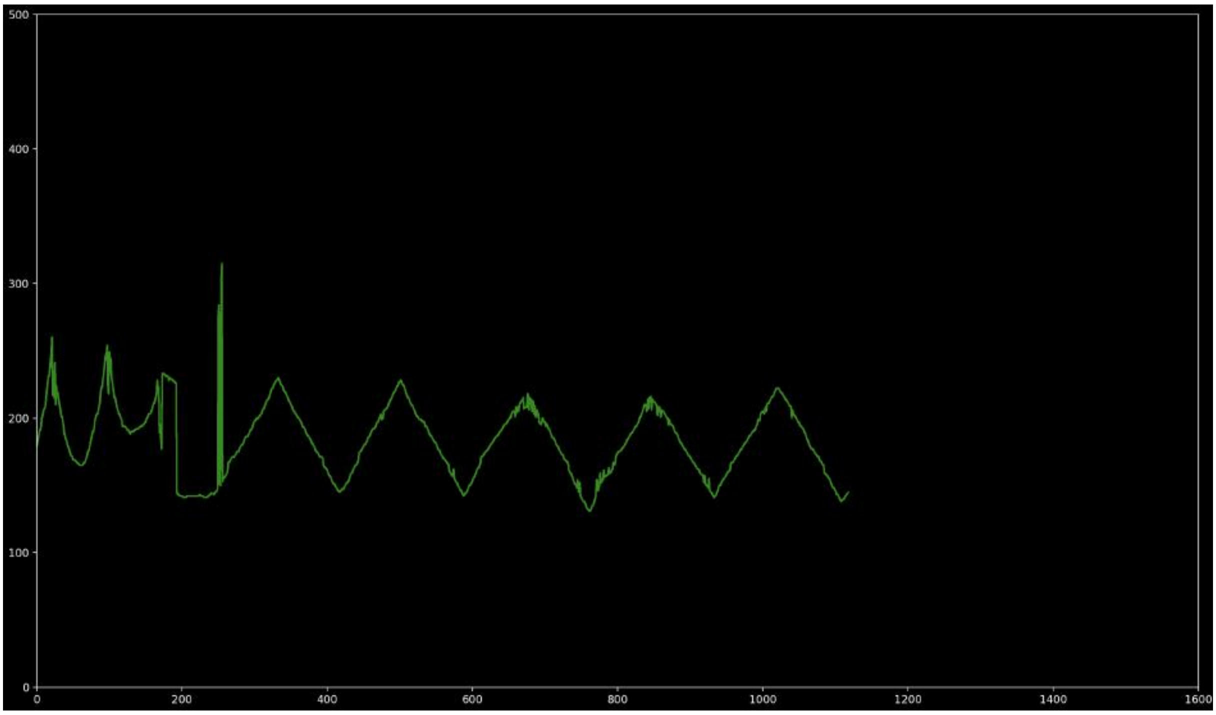

Figure 7: Graphing the position of intersection

Challenges to pantograph detection

One of the main challenges was to get rid of the clouds in the background, and be left with a clean good shot of the contours of the cables. If the sky was always cloud free, this would not have been an issue - but this is not La La Land. During the night I would assume one would either have the cable painted with some self illuminating matter, or have improve lighting conditions using spotlights. Heavy could also be of a problem, since it might be recorded on the camera as a *line*. Varying the structuring element might be an approach here to avoid detecting *lines of rain*.

Figure 8: Detecting left power cable

Figure 9: Detecting right power cable

Figure 10: Flickering due to two simultaneous power cables

As a result of having two visible power cables simultaneously, there was a flicker in the graphing. This could be handled by creating a rule based system, however, I do not think it is easy to generalize this situation and I do not have the domain knowledge regarding which it should *choose*.

Figure 11: The ROI for tracking power cables

This is the view of the processed ROI. As one can see, it is not perfect, but it works very well in my opinion. I was unable to denoise any further without affecting the power- and suspension cable, which is undesirable. I could might improve my solution by applying some morphological operations to remove unwanted lines (noise).

Sources

https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html

https://en.wikipedia.org/wiki/Gamma_correction

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

https://docs.opencv.org/

https://scikit-image.org/docs/dev/

The entire sourcecode for this project can be found in my GitHub repository.

Code

```python
import os
import glob
import imutils
import cv2 as cv
import numpy as np
from skimage import feature
from skimage import img_as_ubyte
from matplotlib import pyplot as plt
from datetime import datetime

FILEPATH = os.path.abspath("images")
OUTPUT = os.path.abspath(("output"))

MIN_HEIGHT = 720
MIN_WIDTH = 1280
MIN_THRESH = 192
MAX_THRESH = 255


def saveImages(destination, images, label=""):
    for i in range(len(images)):
        img = images[i]
        try:
            cv.imwrite(f"{destination}/out_car_{i}_{label}.jpg", cv.cvtColor(img, cv.COLOR_RGB2BGR))
            print(f"Saving number plate {i+1} \U0001F4BE")
        except IOError :
            return f"Error while saving file number: {i}"
    print(f"Succesfully saved {len(images)} images to {destination}")


def loadImages(filepath, carNumber=' '):
    images = []
    for file in sorted(glob.glob(f"{filepath}/*.jpg")):
        if file.lower().endswith('.jpg'):
            img = cv.imread(file, 1)
            img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
            images.append(img)
    if carNumber !=' ' and isinstance(carNumber, int):
        try:
            return images[carNumber]
        except:
```

```
            return f"Can't find that image of carNumber: {carNumber}"
    return images


def checkSize(image):
    (height, width) = image.shape[:2]
    if height < MIN_HEIGHT:
        return imutils.resize(image, height=MIN_HEIGHT)
    elif width < MIN_WIDTH:
        return imutils.resize(image, width=MIN_WIDTH)
    else:
        return image


def getContourpApproximate(image, edges):
    contours, _ = cv.findContours(edges.copy(), cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
    updated_contours = sorted(contours, key=cv.contourArea, reverse=True)[:10]

    for cnt in updated_contours:
        perimeter = cv.arcLength(cnt, True)
        approx = cv.approxPolyDP(cnt, 0.04 * perimeter, True)
        x = approx.ravel()[0]
        y = approx.ravel()[1]
        padding = 1
        if len(approx) == 4:
            cv.putText(image, "Number Plate", (x + padding, y + padding), cv.FONT_HERSHEY_COMPLEX, 1, (0, 255, 255))
            cv.drawContours(image, [approx], -1, (0,255,0), 3, lineType=cv.LINE_AA)
            print(f"Added a candidate! \U0001F919 ")
            break


def getContoursBoundingRectangle(image, edges):
    contours, _ = cv.findContours(edges.copy(), cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
    updated_contours = sorted(contours, key=cv.contourArea, reverse=True)[:3]

    cv.drawContours(image, contours, -1, (255, 0, 0), 2)
    for cnt in updated_contours:
        x, y, w, h = cv.boundingRect(cnt)

        ratio = float(h)/w
        if ratio < 0.5 and ratio > 0.18:
```

```python
            cv.rectangle(image,(x,y),(x+w,y+h),(0,255,0), 3)


def getCrop(image, factor=1):
    img = image.copy()
    h, w = img.shape[:2]
    return img[factor*(h//10):(10-factor)*(h//10), factor*(w//10):(10-factor)*(w//10)]

def geCannyEdges(image, opencv=True):
    img = image.copy()
    if opencv:
        return cv.Canny(img, 70, 200)
    return feature.canny(img, sigma=2)


def adjustGamma(image, gamma=1.0):
    # source: https://www.pyimagesearch.com/2015/10/05/opencv-gamma-correction/
    img = image.copy()
    inverted_gamma = 1.0 / gamma
    look_up_table = np.array([((i / 255.0) ** inverted_gamma) * 255
                  for i in np.arange(0, 256)]).astype("uint8")
    return cv.LUT(img , look_up_table)


def saveFigures(images, titles=[], rows=1):
    now = datetime.now()
    columns = len(images)
    _ = plt.figure(figsize=(64, 64/columns))
    for i in range(1, columns * rows + 1):
        plt.subplot(1, columns, i)
        if not titles[i-1]:
            plt.gca().set_title(f"Subplot_{i-1}", fontsize=32)
        plt.gca().set_title(titles[i-1], fontsize=32)
        plt.imshow(images[i-1], cmap='gray')
    plt.savefig(f"figures/figure_{i}-{now}.png")
    print(f"Saved a figure \U0001F4BE")


def showFigures(images, titles=[], rows=1):
    columns = len(images)
    fig = plt.figure(figsize=(32, 32/columns))
    for i in range(1, columns*rows+1):
        fig.add_subplot(rows, columns, i)
```

```
        if not titles[i-1]:
            plt.gca().set_title(f"Subplot_{i-1}")
        plt.gca().set_title(titles[i-1])
        plt.imshow(images[i-1], cmap='gray')
    plt.show()


def attemptOne(image):
    orig_img = checkSize(image)
    img = cv.cvtColor(orig_img.copy(), cv.COLOR_RGB2GRAY)

    gammad = adjustGamma(img, gamma=0.9)
    blurred = cv.GaussianBlur(gammad, (5,5), 0)
    _, threshold = cv.threshold(blurred, MIN_THRESH, MAX_THRESH, cv.THRESH_OTSU)

    mask = cv.bitwise_and(img, threshold)

    edges = img_as_ubyte(geCannyEdges(mask))

    getContourpApproximate(orig_img, edges)
    return orig_img


def attemptTwo(image):
    orig_img = checkSize(image)
    img = cv.cvtColor(orig_img.copy(), cv.COLOR_RGB2GRAY)
    adjusted = adjustGamma(img, gamma=0.9)
    blurred = cv.GaussianBlur(adjusted, (5, 5), 0)

    _, threshold = cv.threshold(blurred, MIN_THRESH, MAX_THRESH,
    ↪ cv.THRESH_BINARY + cv.THRESH_OTSU)
    mask = cv.bitwise_and(threshold, threshold)

    kernel = cv.getStructuringElement(cv.MORPH_RECT, (3,3))
    erode = cv.erode(mask, kernel)
    closing = cv.morphologyEx(erode, cv.MORPH_CLOSE, kernel)
    opening = cv.morphologyEx(closing, cv.MORPH_OPEN, kernel)

    getContoursBoundingRectangle(orig_img, opening)
    return orig_img


def attemptThree(image):
    orig_img = checkSize(image)
    img = cv.cvtColor(orig_img.copy(), cv.COLOR_RGB2GRAY)
```

```python
    adjusted = adjustGamma(img, gamma=0.9)
    blurred = cv.medianBlur(adjusted, 5)

    threshold = cv.adaptiveThreshold(blurred ,255,cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY,11,2)
    mask = cv.bitwise_and(img, threshold)
    denoise = cv.fastNlMeansDenoising(mask, h=10, templateWindowSize=7, searchWindowSize=21)

    edges = img_as_ubyte(geCannyEdges(denoise, opencv=False))

    getContourpApproximate(orig_img, denoise)
    return orig_img


def attemptFour(image):
    orig_img = checkSize(image)
    orig_img = getCrop(orig_img)
    img = cv.cvtColor(orig_img.copy(), cv.COLOR_RGB2GRAY)

    gammad = adjustGamma(img, gamma=0.9)
    blurred = cv.GaussianBlur(gammad, (7,7), -3)
    #_, threshold = cv.threshold(blurred, MIN_THRESH, MAX_THRESH, cv.THRESH_OTSU)
    threshold = cv.adaptiveThreshold(blurred, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY, 15, -3)

    mask = cv.bitwise_and(img, threshold)

    edges = img_as_ubyte(geCannyEdges(mask))
    #save_figures([img, threshold, mask, edges], titles=["Original", "Thresholded", "Masked", "Canny Edged"])
    #show_figures([img, threshold, mask, edges], titles=["Original", "Threshold", "Masked", "Edged"])

    getContourpApproximate(orig_img, edges)
    return orig_img


def main():
    images = loadImages(FILEPATH)
    out_images = []

    for image in images:
        output = attemptOne(image)
```

```
        out_images.append(output)

    saveImages(OUTPUT, out_images, label="290320")

main()
```

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


def getCrop(frame, point1, point2):
    img = frame.copy()
    padding = 1 # 1 pixel for padding
    return img[0:point1[1] + padding, point1[0]:point2[0] - padding]


def houghlineProbabilisticProcessFrame(frame):
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    gray_copy = np.copy(gray)
    edges = cv.Canny(gray_copy, 50, 150, apertureSize=3)
    lines = cv.HoughLinesP(edges, 1, np.pi/180, 100, 100, 100)
    try:
        for x1, y1, x2, y2 in lines[0]:
            cv.line(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
    except:
        pass


def houghlineStandardProcessFrame(frame):
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    lines = np.copy(gray)
    edges = cv.Canny(lines, 70, 200, apertureSize=3)
    lines = cv.HoughLines(edges, 1, np.pi/180, 192, None, 0, 0)

    if lines is not None:
        for i in range(0, len(lines)):
            rho = lines[i][0][0]
            theta = lines[i][0][1]
            a = np.cos(theta)
            b = np.sin(theta)
```

```
            x0 = a * rho
            y0 = b * rho
            pt1 = (int(x0 + 1000 * (-b)), int(y0 + 1000 * (a)))
            pt2 = (int(x0 - 1000 * (-b)), int(y0 - 1000 * (a)))
            cv.line(lines, pt1, pt2, (0, 0, 255), 3, cv.LINE_AA)
    return lines


def morphProcessFrame(frame):
    # source: https://docs.opencv.org/3.4/dd/dd7/tutorial_morph_lines_detection.html
    verticalSize = 16
    verticalStructure = cv.getStructuringElement(cv.MORPH_RECT, (1, verticalSize))

    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    lines = np.copy(gray)
    lines = cv.bitwise_not(lines)
    lines = cv.adaptiveThreshold(lines, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY, 15, -3)
    lines = cv.erode(lines, verticalStructure, iterations=1)
    lines = cv.dilate(lines, verticalStructure, iterations=2)

    smooth = np.copy(lines)
    smooth = cv.bilateralFilter(smooth, 11, 17, 17)
    rows, columns = np.where(smooth != 0)
    lines[rows, columns] = smooth[rows, columns]

    return lines


def getContour(orig_frame, edges, dx, dy):
    contours, _ = cv.findContours(edges, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
    power_cable = sorted(contours, key=cv.contourArea, reverse=True)[0]
    epsilon = cv.arcLength(power_cable, True)
    approx = cv.approxPolyDP(power_cable, epsilon*0.069, True)
    x, y = approx.ravel()[0], approx.ravel()[1]
    fontScale = (orig_frame.shape[0] * orig_frame.shape[1])/(np.power(10, 6))

    cv.putText(orig_frame, "Power Cable", (x + dx + 10, y + dy + 10), cv.FONT_HERSHEY_COMPLEX, fontScale, (255, 255, 0))
    cv.drawContours(orig_frame, [approx], -1, (0, 255, 0), thickness=2, lineType=cv.LINE_AA, offset=(dx, dy))
```

```
    return approx


def setCrosshair(orig_img, contour, dx, dy):
    x,y = contour[-1].ravel()[0], contour[-1].ravel()[1]
    cv.circle(orig_img, (x + dx, y + dy), 7, (0, 0, 255), 2)


def getPosition(power_cable):
    return power_cable[-1].ravel()[0]


def initGraph(style='dark_background'):
    y_pos = 0
    xs, ys = [], []
    plt.style.use(style)
    fig = plt.figure(figsize=(8, 10))
    ax = fig.add_subplot(111)
    ax.set_xlim(left=0, right=854) # width of videofile
    fig.show()

    return xs, ys, fig, ax, y_pos


def main():
    # Initialize reading video file
    videoCap = cv.VideoCapture("videos/Eric2020.mp4")
    init_frame_read_correctly, init_frame = videoCap.read()
    if not init_frame_read_correctly:
        print("No frames to read...")

    # Set Bounding Box for Region of Interest
    bounding_box = (293, 173, 376, 87) # alternatively: use cv.selectROI('tracking', init_frame)

    # Set tracker and initialize "Toggle switch" for tracking
    tracker = cv.TrackerMOSSE_create()
    initial_kickstart = True

    # Initialize graph, for position of intersection between pantograph and power cable
    xs, ys, fig, ax, y_pos = initGraph()

    while videoCap.isOpened():
        # Start reading video file after ROI is set
```

```python
        frame_read_correctly, frame = videoCap.read()
        if not frame_read_correctly:
            print("No frames to read...")
            break

        # Initialize the tracker
        if initial_kickstart:
            tracker.init(frame, bounding_box)
            initial_kickstart = False

        # Update bounding box for rectangle
        frame_read_correctly, new_bounding_box = tracker.update(frame)
        #print(f"x1: {new_bounding_box[0]}\t y1: {new_bounding_box[1]}\t x2: {new_bounding_box[2]}\t y2: {new_bounding_box[3]}")
        if frame_read_correctly:
            point1 = (int(new_bounding_box[0]), int(new_bounding_box[1]))
            point2 = (int(new_bounding_box[0] + new_bounding_box[2]), int(new_bounding_box[1] + new_bounding_box[3]))
            cv.rectangle(frame, point1, point2, (255, 255, 0), 1)

        # Crop out new ROI and process it
        new_ROI = getCrop(frame, point1, point2)
        processed_crop = morphProcessFrame(new_ROI)

        # Draw contour on the power cable, and set crosshair (red dot) at intersection with pantograph
        power_cable = getContour(frame, processed_crop, point1[0], 0)
        setCrosshair(frame, power_cable, point1[0], 0)

        # Update Graph
        fig.canvas.draw()
        ax.set_ylim(bottom=max(0, y_pos - 50), top=y_pos + 50)
        ax.plot(xs, ys, color='g')
        xs.append(getPosition(power_cable))
        ys.append(y_pos)
        y_pos += 1

        # Show processed- and video frame
        cv.imshow("Cropped", processed_crop)
        cv.imshow("Pantograph", frame)

        # Press 'q' to quit the windows
        if cv.waitKey(1) & 0xFF == ord('q'):
            break

```

```
    # Release memory when job is finished
    videoCap.release()
    cv.destroyAllWindows()


main()
```