U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Traffic sign detection and classification

Computer Vision - 2019/2020

Carolina Pinto
up201506006@fe.up.pt

Helena Montenegro
up201604184@fe.up.pt

Juliana Marques
up201605568@fe.up.pt

1. Introduction

The main goal of this project is to develop a program for the automatic detection of a subset of all the signs used in Portugal and the classification of the detected sign into a set of classes, according to the combination of sign color and shape. By applying our theoretical knowledge about Image Processing and Analysis acquired in this course we were able to achieve this goal, which resulted in the development of an effective method to detect and classify traffic signs in pre acquired images, as well as images acquired through a computer connected camera.

2. Implementation and Results

We have developed the project in Python, using the **OpenCV** library as a development tool. The code was divided into two separate files:

- **main.py:** acquires an image, either from a computer connected camera or from a file.
- **traffic_sign_detection.py**: detects objects in the image.

In **main.py**, we have implemented the reading of the arguments given in the command line, as well as the respective call to the object detection functions in the scene. You can analyse an image already saved in a file or use the camera instead.

In **traffic_sign_detection.py**, we have implemented the detection of the different classes of objects in different functions. The segmentation of the detection processes in different functions allowed not only the parallelization of work between the team members but also to apply different algorithms according to the shapes we wanted to find.

2.1. Detection of circles

The detection of circles was implemented in the function: **find_circle (img, img_to_show, color)** which receives as arguments:

- **img:** the image to be analysed.
- **img_to_show:** the image to be shown with the detected circles.
- **color:** color of the circle we want to detect.

The detection process has the following steps:

1. Segmentation of the image in the color received as an argument.
2. Thresholding of the image.
3. Smoothing of the segmented image, using a Gaussian function.
4. Detection of edges using the Canny Edge Detector.
5. Detection of the objects' external contours.
6. Application of the Hough transform algorithm to detect circles.

Initially, we had applied the **HoughCircles** algorithm to the smoothed image, without using the **Canny Edge Detector** and the detection of the objects' external contours. However, we have noticed that the parameters for the **HoughCircles** algorithm have to be configured according to each image, which meant that it would work perfectly for some images, but for others, it would either not find circles where it should or find circles where it shouldn't. On top of this, in the case of signs where the outer edge was red and that had a white circle in the middle, the **HoughCircles** would often find the inner circle, instead of the outer one. To solve this, we have decided to first detect external contours and draw them on a blank image, on which we would then apply the **HoughCircles** function.

To see results of each of the implementation steps, check ***Annex 5.1.1.***

For more results, check ***Annex 5.3.1*** and ***Annex 5.3.2.***

2.2. Detection of triangles

For the detection of triangles, we developed a function called **find_triangle (img, img_to_show)** which receives as arguments:

- **img:** the image to be processed.
- **img_to_show:** output image that has the resulting detected triangles.

The detection process has the following steps:

1. Smoothing of the segmented image, using a **Gaussian** function.
2. Segmentation of the image in the color red.
3. Thresholding of the image.
4. Detection of edges using **Canny Edge Detector**.
5. Detection of contours and respective analysis to detect triangles:
   a) First, we check if there are 3 lines that form a triangle.
   b) Confirm if the three lines are convex.
   c) Check that the triangle isn't very small in relation to the image, to avoid identifying small spots as triangles.

To see the results of each of the implementation steps, check ***Annex 5.1.2***.
For more results, check ***Annex 5.3.3***.

2.3. Detection of squares / rectangles

For the detection of squares and rectangles, we developed a function called **find_square(img, img_to_show)** which receives as an argument:

- **img:** image to be processed.
- **img_to_show:** output image that has the resulting detected squares / rectangles.

The detection process for squares/rectangles has the same steps as the detection of triangles, only differing on the analysis of the contours:

1. First, we check if there are 4 lines in the contours.
2. Confirm if the four lines are convex.
3. Check that the square / rectangle isn't very small in relation to the image, to avoid identifying small spots as squares.
4. Check if the angles between the lines are between 70 and 110 degrees, to make sure that the object is a square / rectangle.

To see the results of each of the implementation steps, check ***Annex 5.1.3***.
For more results, check ***Annex 5.3.4.***

2.4. Detection of stop signs

The detection of stop signs was implemented in the function **find_stop (img, img_to_show)**, which receives as arguments the image to be analysed and the image to be shown with the detected octagon. Similarly, to the previous shapes' detection, this process contains its color segmentation and smoothing, followed by the detection of edges with the Canny Edge Detector and the detection of objects' external contours.

This detection differs from the previous ones in the analysis of the contours. We assume that the angles between the 8 lines that form an octagon must be between 30 and 60 degrees, and that the contours must be convex.

The results of this detection can be found in ***Annex 5.3.5.***

2.5. Improvements

Detection of STOP signs, as previously explained.

Detection of multiple signs in an image: the algorithms implemented allow the detection of multiple signs in an image, which can belong to the same class or different ones. For this, we have developed 4 different functions as described above, all of which detect the respective class of objects in the original image and draw its contour on the image to be shown. Inside each function, the algorithms allow us to find every object of the respective class. However, there is a restriction on circles: the respective centres need to be apart in a range of at least a sixth of the image's height, so that the Hough Transform algorithm doesn't find many misinterpreted circles in the same space. The results can be found in ***Annex 5.3.6.***

Detection of poorly illuminated signs: In order to detect poorly illuminated signs, the range of the colors in the Segmentation step was devised to detect darker colors, which would be found in images with poorly illuminated signs. The results can be found in ***Annex 5.3.7***.

Detection of slanted signs: slanted signs are detected to some extent in the triangle, square and stop detection, since we have left some margin for angles between contours in these classes. For STOP signs, the angles don't have to be around 45 degrees, they can vary between 30 degrees and 60 degrees, which also allows to detect slanted stop signs to some extent. In squares and rectangles, we also defined a margin for the angles, between 70 and 110 degrees. In triangles we don't check the angles, since three convex lines always form a triangle, independently of the angle between them, which facilitates the detection of slanted triangles. The results can be found in ***Annex 5.3.8.***

Detection of partially occluded circular signs: when it comes to circles, the Hough Transform only needs a set of points to recognize an object as a circle. On one hand, that's what makes this algorithm detect circles on objects such as inside squares packed with letters, on the other hand, this allows to recognize partially occluded circular signs. The results can be found in ***Annex 5.3.9.***

3. Difficulties and Limitations

It was difficult to implement a form of detection that would work in every image of a traffic sign. We had to consider the parameters used in the different algorithms for the detection to be as accurate as possible and we tested using plenty of road images.

Specifically, the detection of circles was hard, since **HoughCircles** parameters should be specified according to the image. We would often change these parameters for it to work on a specific image only to find out that it stopped working for another picture. It would mistakenly find circles inside blue square signs packed with words. In order to combat this and to get a more accurate circle detection we decided to first apply the Canny Edge Detector, find the external contours of objects in the image and apply the Hough Circles on the obtained contours, which resulted quite well.

The STOP sign detection was also especially hard, since it would often detect octagons where there were circles. We had to meticulously consider the parameters we used to find contours and to analyse them, so that it gave the best results possible for recognizing octagons without recognizing circles.

4. Conclusion

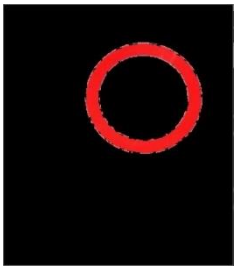
In conclusion, this project allowed us to consolidate and apply our knowledge about Image Processing and Analysis. The results obtained in this project were very promising as we were able to fulfil the main goal of this project. Furthermore, we were also successful in implementing some of the suggested improvements, such as the detection of multiple signs in an image, the detection of stop signs, detection of poorly illuminated signs as well as the detection of slanted signs.

5. Annexes

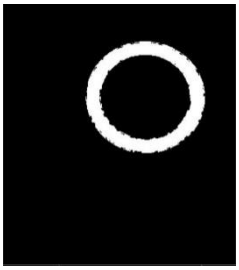
5.1. Results of each of the implementation steps

5.1.1. Detection of circles

1. Initial image

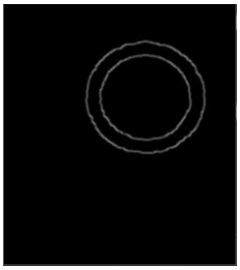


2. Segmentation

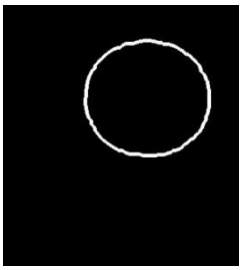


3. Thresholding

4. Smoothing



5. Canny Edge Detection



6. External Contours

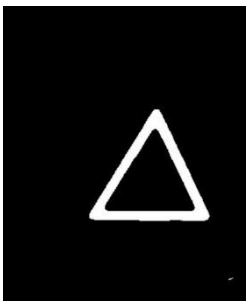
7. Final Result after Hough transform

5.1.2. Detection of triangles

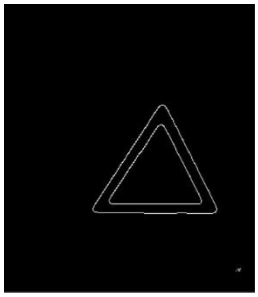
1. Initial image

2. Smoothing

3. Segmentation



4. Thresholding



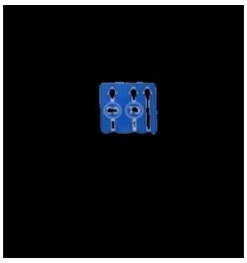
5. Canny Edge Detection

6. Final Result after detection and analysis of contours

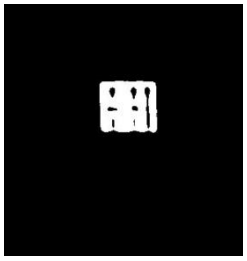
5.1.3. Detection of squares/rectangles

1. Initial image

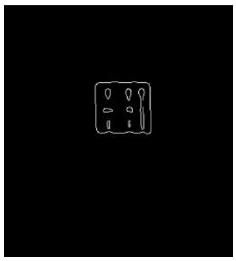
2. Smoothing



3. Segmentation



4. Thresholding



5. Canny Edge Detection

6. Final Result after detection and analysis of contours

5.2. Code

5.2.1. main.py

src > main.py

```python
import sys
import cv2
import argparse
import numpy as np
import traffic_sign_detection as traffic

parser = argparse.ArgumentParser(description="Traffic sign detection")
parser.add_argument('method' ,help='\'camera\' or \'file\'', type=str)
parser.add_argument('-i', '--image', dest='path', default='img.jpg',type=str)
img = None

# parse all the arguments received
args = parser.parse_args()
arg = args.method

# get images from camera
if arg == 'camera':
    cap = cv2.VideoCapture(0)
    img = None
    while(True):
        ret, frame = cap.read()
        frame_to_show = frame.copy()

        # Find shapes
        traffic.find_shapes(frame, frame)

        cv2.imshow('Image', frame)
        key = cv2.waitKey(1)
        esc_key = 27
        if key == esc_key:
            break

    cap.release()
    cv2.destroyAllWindows() # to destroy all open windows

# open preacquired image
elif arg == 'file':
    print('Opening image ' + args.path)
    img = cv2.imread(args.path)
    if img is None:
        print('Image not found')
        quit()
    img_to_show = img.copy()

    # Find shapes
    traffic.find_shapes(img, img_to_show)

    # Show result
    cv2.imshow('Image', img_to_show)
    cv2.waitKey(0) # so that the window doesn't close right away
    cv2.destroyAllWindows() # to destroy all open windows
```

5.2.2. traffic_sign_detection.py

src > traffic_sign_detection.py

```python
import argparse
import cv2
import math
import numpy as np
import traffic_sign_detection as traffic

"""
Description: function to call each of the functions that detect the wanted classes.
Attributes:
    img: the image to be analysed.
    img_to_show: the image to be shown with the detected signs.
"""
def find_shapes(img, img_to_show):
    find_circle(img, img_to_show, "red")
    find_circle(img, img_to_show, "blue")
    find_triangle(img, img_to_show)
    find_square(img, img_to_show)
    find_stop(img, img_to_show)


"""
Description: function to detect circles.
Attributes:
    img: the image to be analysed.
    img_to_show: the image to be shown with the detected circles.
    color: color of the circle we want to detect.
"""
def find_circle(img, img_to_show, color):
    # Segment image according to the color received as argument
    img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    if color == "red":
        img_red1 = cv2.inRange(img_hsv, (0, 190, 70), (10, 255, 255))
        img_red2 = cv2.inRange(img_hsv, (170, 90, 50), (180, 255, 255))
        img_color = img_red1 + img_red2
    else:
        img_color = cv2.inRange(img_hsv, (105, 150, 70), (130, 255, 255))

    result_color = cv2.bitwise_and(img, img, mask=img_color)
    img_gray = cv2.cvtColor(result_color, cv2.COLOR_BGR2GRAY)

    thresh = cv2.threshold(img_gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)[1]

    # Smoothing of Image
    thresh = cv2.GaussianBlur(thresh, (9, 9), 0)

    # Get Outer Contours of Objects
    canny = cv2.Canny(thresh, 100, 200)
    canny = cv2.GaussianBlur(canny, (5, 5), 0)
    cnts, _ = cv2.findContours(canny, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
    blank = np.zeros((img.shape[0], img.shape[1], 3), np.uint8)
    for c in cnts:
        approx = cv2.approxPolyDP(c, 0.01 * cv2.arcLength(c, True), True)
        if len(approx) > 4:
            cv2.drawContours(blank, [c], -1, (255, 255, 255), 4)

    # Detection of circles
    circles = cv2.HoughCircles(cv2.cvtColor(blank, cv2.COLOR_BGR2GRAY), cv2.HOUGH_GRADIENT, 2, img.shape[0] / 6,
        param1=200, param2=105, minRadius=0, maxRadius=0)

    # Draw circles on the image
    if circles is not None:
        circles = np.round(circles[0, :]).astype("int")
        for (x, y, r) in circles:
```

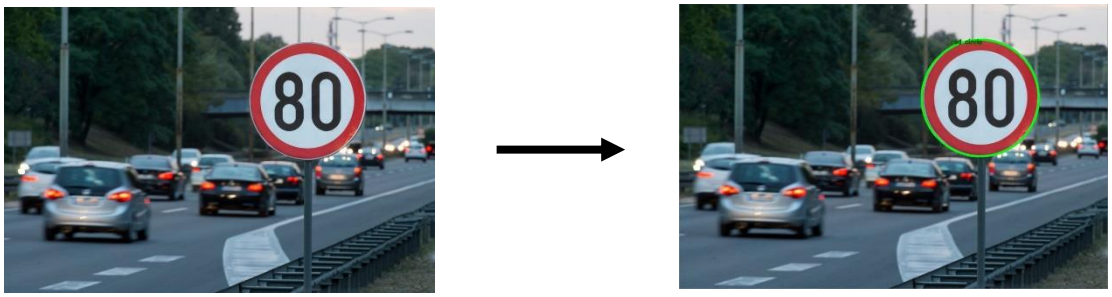
```
            cv2.circle(img_to_show, (x, y), r, (0, 255, 0), 4)
            cv2.putText(img_to_show, color + " circle", ((int)(x-r/2), y-r+10),
                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 105, 0), 2)
            print(color + " circle")



"""
Description: function to detect red triangles.
Attributes:
    img: the image to be analysed.
    img_to_show: the image to be shown with the detected red triangles.
"""
def find_triangle(img, img_to_show):

    # Smoothing of the image
    blurred = cv2.GaussianBlur(img, (15, 15), 0)

    # Converting to HSV color space in order to segment the image according to colors
    hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)

    # Range for lower red
    lower_red = np.array([0, 120, 70])
    upper_red = np.array([10, 255, 255])

    mask1 = cv2.inRange(hsv, lower_red, upper_red)

    # Range for upper range of red
    lower_red = np.array([170, 120, 70])
    upper_red = np.array([180, 255, 255])
    mask2 = cv2.inRange(hsv,lower_red,upper_red)

    mask = mask1 + mask2

    result_red = cv2.bitwise_and(img, img, mask = mask)

    gray = cv2.cvtColor(result_red, cv2.COLOR_BGR2GRAY)
    thresh = cv2.threshold(gray, 20, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)[1]

    # Detect contours using Canny Edge Detector
    canny = cv2.Canny(thresh, 100, 200)
    cnts, _ = cv2.findContours(canny, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Analyse contours to detect triangles
    var =  img.shape[0] * img.shape[1] /math.fabs(cv2.contourArea(3))
    for c in cnts:
            peri = cv2.arcLength(c, True)
            approx = cv2.approxPolyDP(c, 0.04 * peri, True)
            if len(approx) == 3 and cv2.isContourConvex(approx) and var < 10000:
                x = approx.ravel()[0]
                y = approx.ravel()[1] + 2
                cv2.drawContours(img_to_show, [c], -1, (0, 255, 0), 2)
                cv2.putText(img_to_show, "red triangle", (x, y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 105, 0), 2)
                print("red triangle")


"""
Description: function to detect blue squares
Attributes:
    img: the image to be analysed.
    img_to_show: the image to be shown with the detected blue squares/retangles.
"""
def find_square(image, img_to_show):

    # Smoothing of the image
    blurred = cv2.GaussianBlur(image, (15, 15), 0)

    # Color segmentation of the image
    hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
    img_blue = cv2.inRange(hsv, (100, 120, 70), (140, 255, 255))
    result_blue = cv2.bitwise_and(image, image, mask=img_blue)

    gray = cv2.cvtColor(result_blue, cv2.COLOR_BGR2GRAY)
    thresh = cv2.threshold(gray, 20, 255, cv2.THRESH_BINARY)[1]
```

```
    # Detect contours using Canny Edge Detector
    canny = cv2.Canny(thresh, 100, 200)
    cnts, _ = cv2.findContours(canny, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Analyse contours to detect squares / rectangles
    for c in cnts:
        peri = cv2.arcLength(c, True)
        approx = cv2.approxPolyDP(c, 0.02 * peri, True)
        # A square is convex, has four contours and it can't be too small compared to the size of the image
        if len(approx) == 4 and cv2.isContourConvex(approx) and image.shape[0] * image.shape[1] / math.fabs(cv2.contourArea(approx)) < 10000:
            maxCosine = 0
            i = 2
            while i < 5:
                cosine = math.fabs(angle(approx[i % 4], approx[i - 2], approx[i - 1]))
                maxCosine = max(cosine, maxCosine)
                i += 1
            # We assume that the angles between the contours can vary between ~70 and ~110 degrees.
            if maxCosine < 0.3:
                x = approx.ravel()[0]
                y = approx.ravel()[1] - 5
                cv2.drawContours(img_to_show, [c], -1, (0, 255, 0), 2)
                cv2.putText(img_to_show, "blue square", (x, y), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 105, 0), 2)
                print("blue square")


"""
Description: function to detect stop signs.
Attributes:
    img: the image to be analysed.
    img_to_show: the image to be shown with the detected STOP sign.
"""
def find_stop(img, img_to_show):

    # Color Segmentation of Image
    img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    img_red1 = cv2.inRange(img_hsv, (0, 120, 70), (10, 255, 255))
    img_red2 = cv2.inRange(img_hsv, (170, 90, 70), (180, 255, 255))
    img_color = img_red1 + img_red2
    result_color = cv2.bitwise_and(img, img, mask=img_color)
    img_gray = cv2.cvtColor(result_color, cv2.COLOR_BGR2GRAY)
    thresh = cv2.threshold(img_gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)[1]

    # Smoothing of Image
    blurred = cv2.GaussianBlur(thresh, (3, 3), 0)

    # Detect contours of objects in image
    canny = cv2.Canny(blurred, 100, 200)
    contours, _ = cv2.findContours(canny, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Analyse contours to find octogonal shapes
    for cnt in contours:
        approx = cv2.approxPolyDP(cnt, 0.01 * cv2.arcLength(cnt, True), True)
        if len(approx) == 8 and cv2.isContourConvex(approx) and img.shape[0] * img.shape[1] / math.fabs(cv2.contourArea(approx)) < 10000:
            maxCosine = 0
            i = 2
            while i < 9:
                cosine = math.fabs(angle(approx[i % 8], approx[i - 2], approx[i - 1]))
                maxCosine = max(cosine, maxCosine)
                i += 1
            # We assume that the angles between the contours can vary between ~30 and ~60 degrees.
            if maxCosine > 0.5 and maxCosine < 0.9:
                x = approx.ravel()[0]
                y = approx.ravel()[1] - 5
                cv2.drawContours(img_to_show, [cnt], 0, (0, 255, 0), 6)
                cv2.putText(img_to_show, "STOP", (x, y+10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 105, 0), 2)
                print("STOP")


"""
Description: auxiliary function to determine angles between lines

Attributes:
    pt1: point to determine the angle.
    pt2: point to determine the angle.
    pt0: point to determine the angle.
"""
def angle(pt1, pt2, pt0):
    dx1 = pt1[0][0] - pt0[0][0]
    dy1 = pt1[0][1] - pt0[0][1]
    dx2 = pt2[0][0] - pt0[0][0]
    dy2 = pt2[0][1] - pt0[0][1]

    if (dx1*dx1 + dy1*dy1)*(dx2*dx2 + dy2*dy2) + 1e-10 <= 0:
        return math.sqrt(2) / 2
    return (dx1*dx2 + dy1*dy2)/math.sqrt((pow(dx1, 2) + pow(dy1, 2))*(pow(dx2, 2) + pow(dy2, 2)) + 1e-10)
```

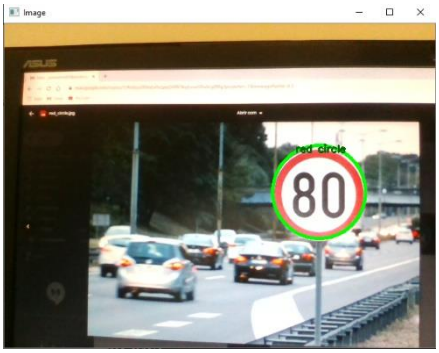
5.3.Results

5.3.1. Red circles

5.3.1.1. Using file



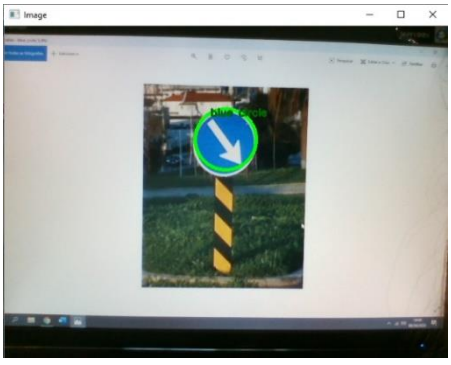
5.3.1.2. Using camera



5.3.2. Blue circles

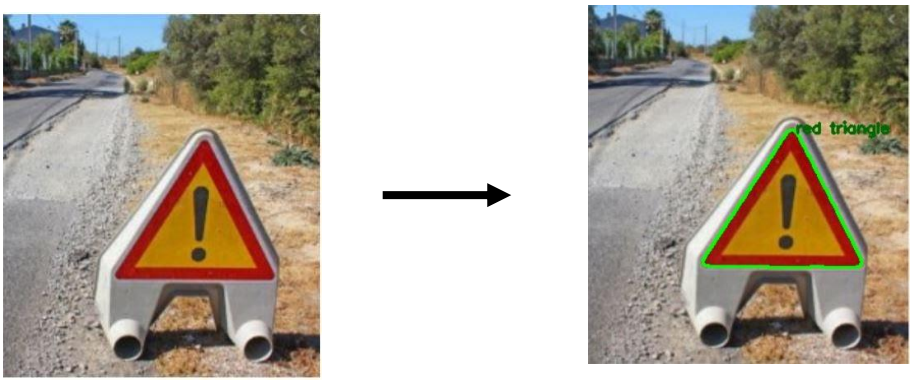
5.3.2.1.Using file

5.3.2.2. Using camera

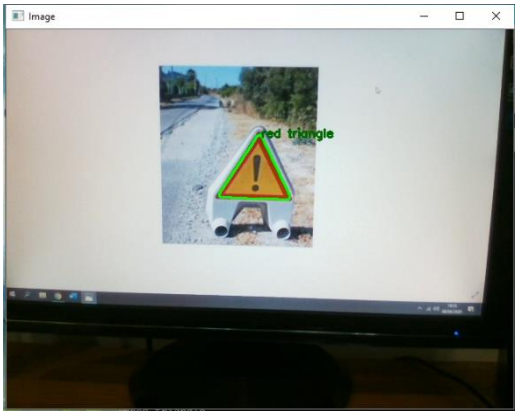


5.3.3. Red triangles

5.3.3.1. Using file

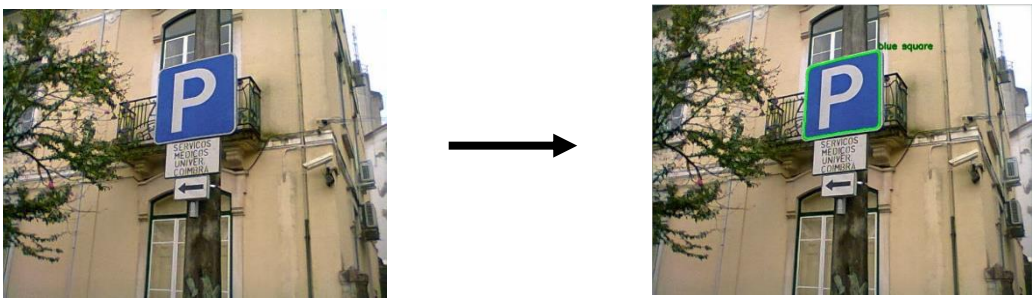


5.3.3.2. Using camera

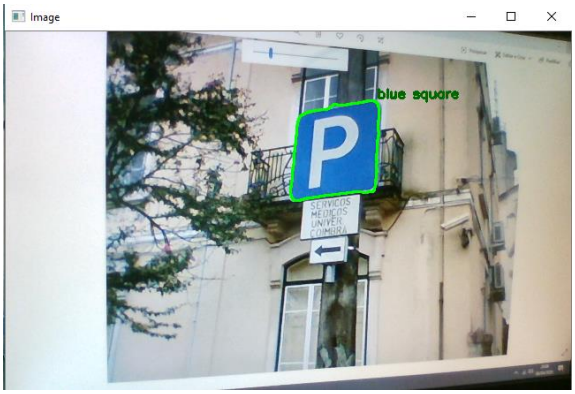


5.3.4. Blue squares / rectangles

5.3.4.1. Using file



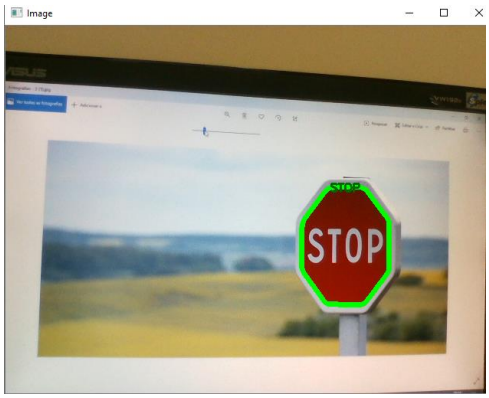
5.3.4.2. Using camera

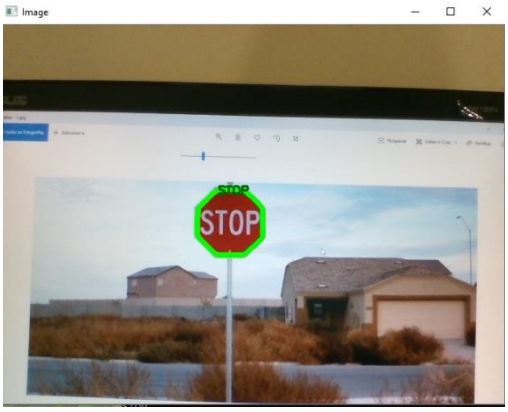


5.3.5. STOP signs

5.3.5.1. Using file

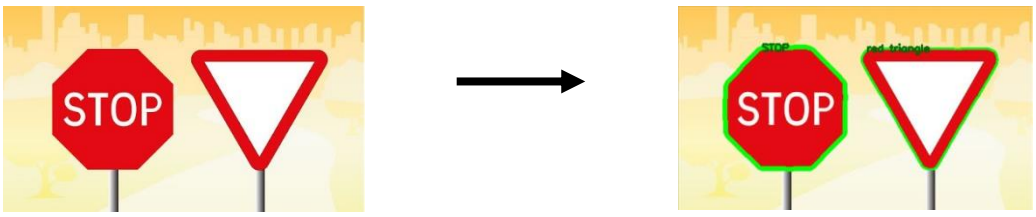
5.3.5.2. Using camera

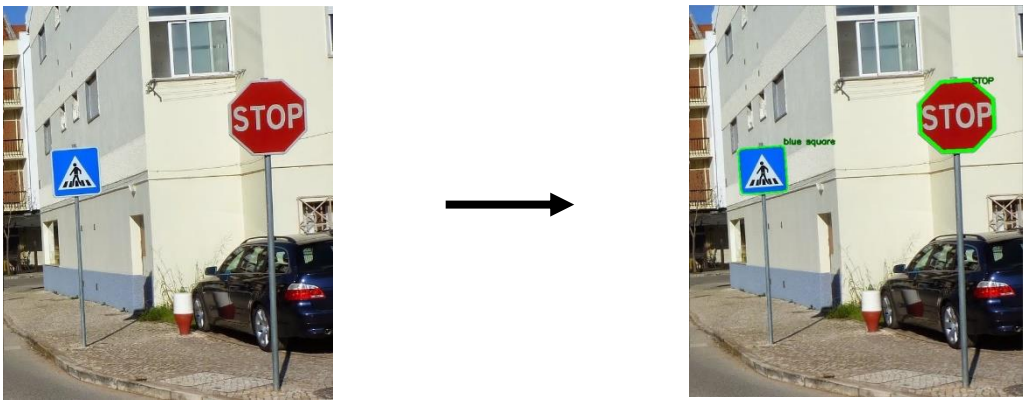




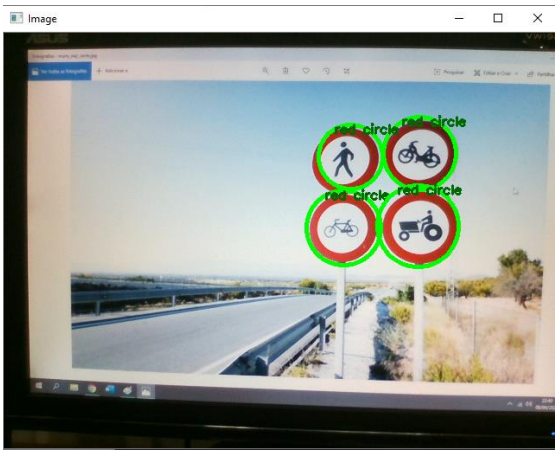
5.3.6. Multiple signs

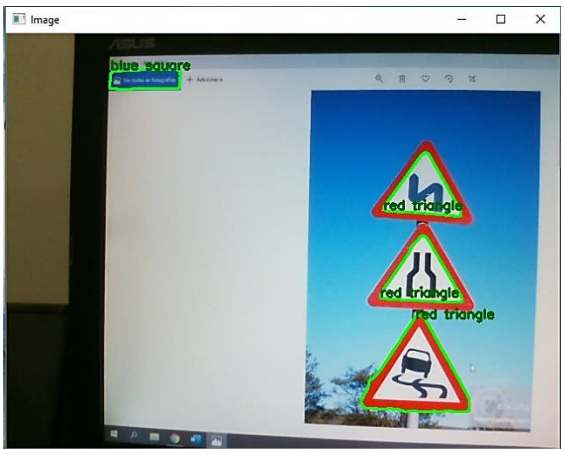
5.3.6.1. Using file





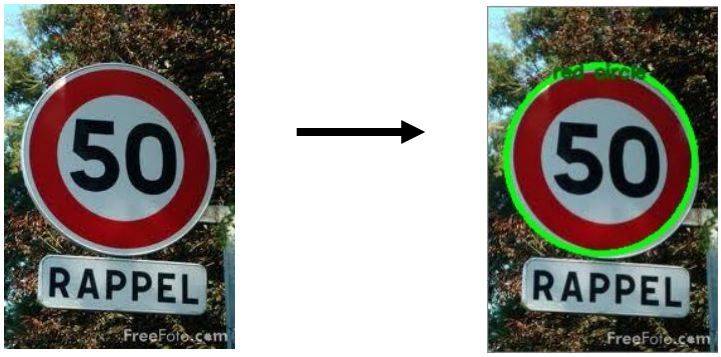
5.3.6.2. Using camera



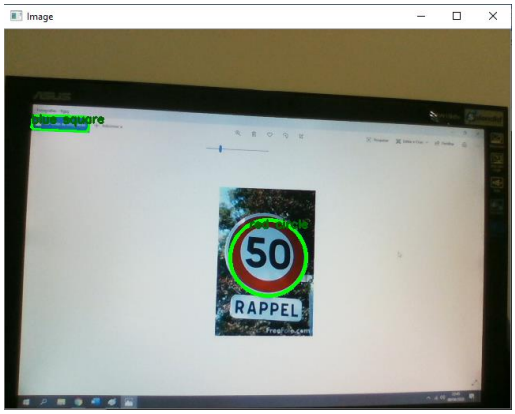


5.3.7. Poorly illuminated signs

5.3.7.1. Using file

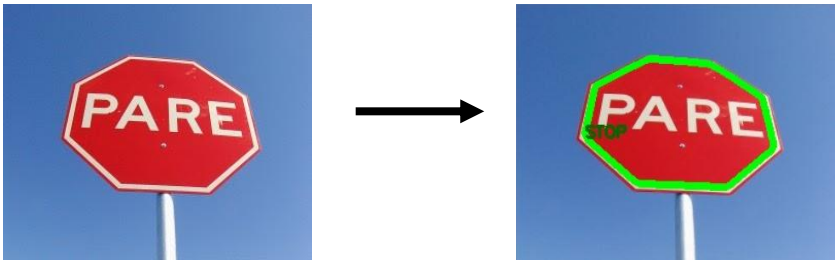


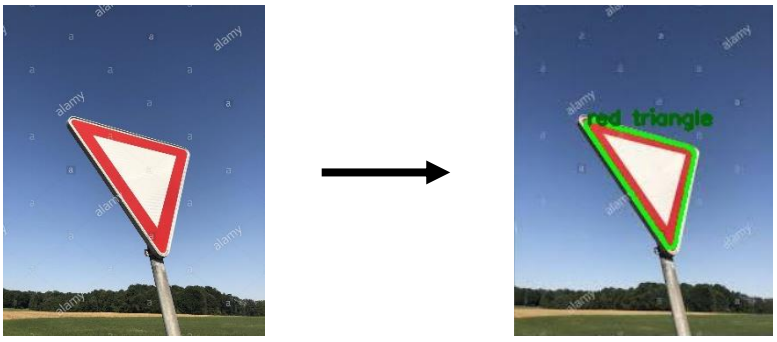
5.3.7.2. Using camera



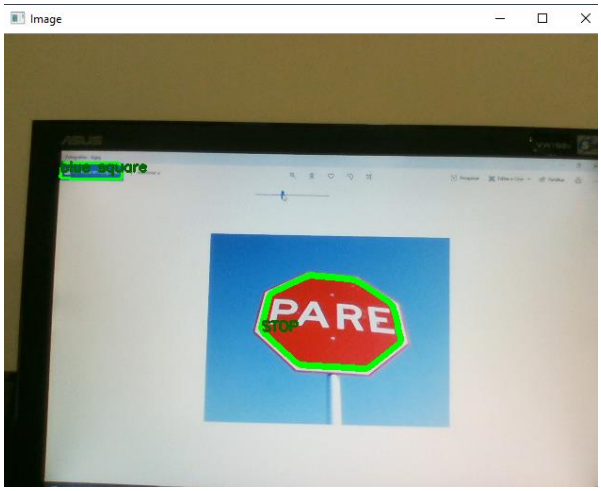
5.3.8. Slanted signs

5.3.8.1. Using file





5.3.8.2. Using camera



5.3.9. Partially occluded circular signs

5.3.9.1. Using file

5.3.9.2. Using camera