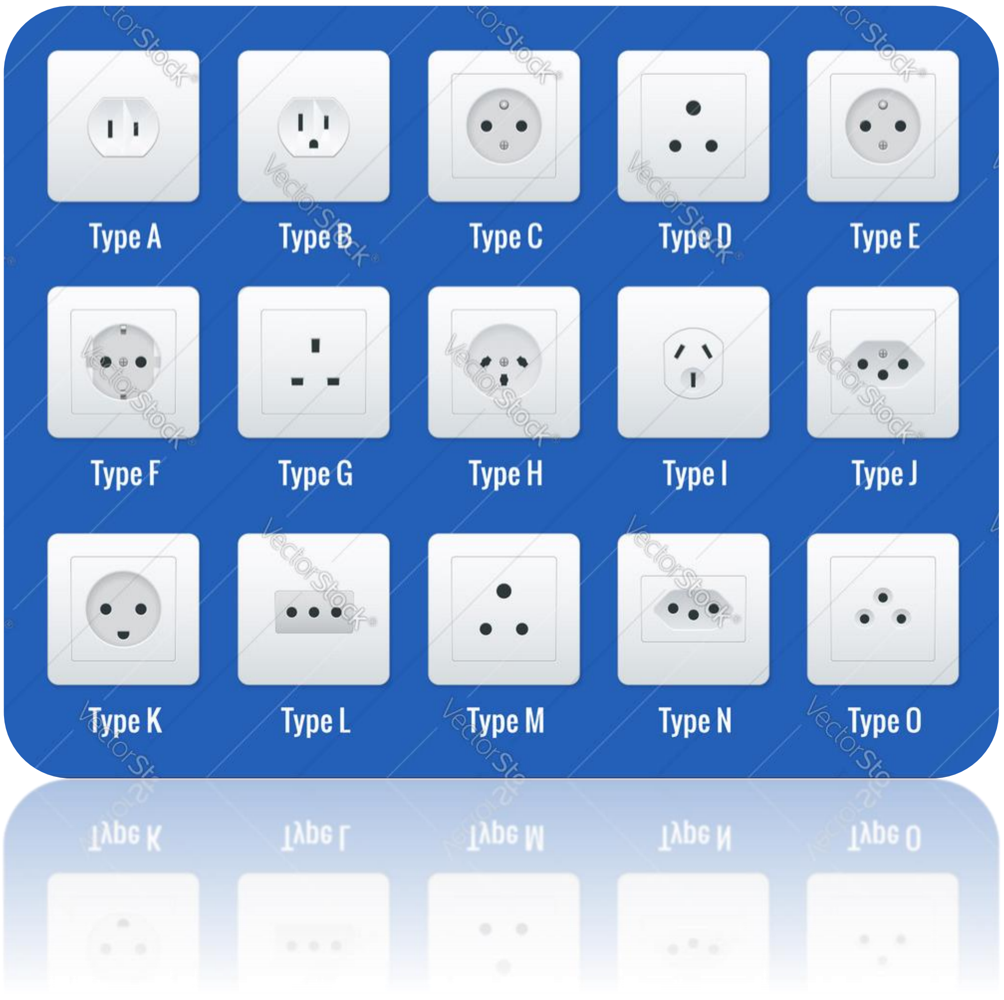
Socket's Types Recognition

Project report



14.3.2023

Amit Shraga - 208993493, Tomer Tal – 316471994

Introduction

This project aims to detect and recognize types of sockets from an image. The following steps outline the flow of our program:

- Step 1 - Resize the image:
  In order to normalize values such as radius, distances etc. between different images we resized all the images to be the same size.

- Step 2 - Recognize the object (the socket):
  Convert the resized image to a grayscale image. The purpose is to simplify the image. After that we used "cv2.GaussianBlur", to reduce the amount of noise that could interfere with the subsequent steps in the process.
  Than we created a binary image using "cv2.threshold" to find the bright areas in the image.
  Find the biggest polygon in the binary image.
  Check the angles of the biggest polygon found to see if its angles not too sharp(less than 52 degree), if it is, lower the threshold, and try to find a polygon again.
  It was needed to check the angles because we have encountered cases where the detected polygon passed between crucial parts of the socket.
  So to not miss any information we had to try and find another polygon. Finally, paint the outside of the polygon in white.

- Step 3 - Find a threshold:
  We used the function "cv2.threshold" to convert our grayscale image to a binary image.
  We wanted to separate the black holes of the socket from the rest of the image.
  So we had to find the right threshold that would mark only the holes as white spots, and would refer to all the other image as a black background.
  The threshold needed to be very accurate, not too big so it will not include shadows and unwanted black marks as holes,
  and not too small so the black holes of the socket will still be painted in white.
  Also, different images needed different thresholds. So, the threshold is individual per image.
  For these reasons we used a while loop that decreases the threshold in each loop until we get a list of 2 or 3 contours - the number of holes of a sockets.
  To find the contours We used the function "cv2.findContours" that returns a list of contours in the image.

- Step 4 - Crop the hole:
  We got to a conclusion that to best classify each hole(as a circle or a rectangle), we had to look at it as the main object of the image.
  Therefore, we used "cv2.approxPolyDP" to get a polygon of each contour so we could crop the original image according to the x and y coordinates of each polygon.
  We cropped the holes from the original image (and not from the binary image from previous step 3)
  because we wanted the original and most accurate shape of the holes without the "threshold" function damaging and affecting them.
  After cropping the holes to one image per hole, we threshold each image with a more allowing threshold so that we get an accurate shape for each hole.

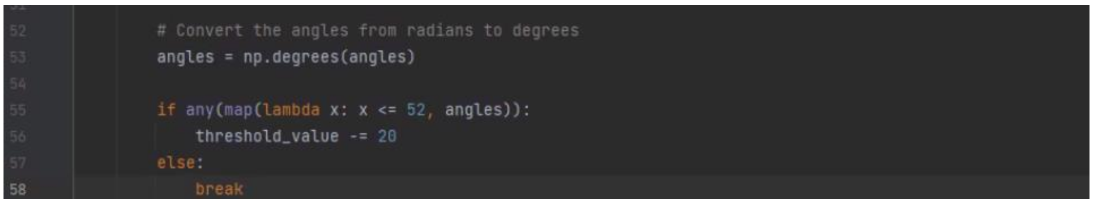
- Step 5 - Classify - circle or rectangle:
  For each image of a single hole we used "cv2.approxPolyDP" function that returns a polygon of the contour in the image.
  The length of the polygon(number of points in the polygon) helped us determine
  Whether it is a polygon of circle (the polygon has more than 8 points) or it is a rectangle (polygon has 2 to 5 points).

- Step 6 - Find relations between the holes:
  After we found the shape for each hole, we now have the whole set of shapes of holes in a socket.
  Some of the types of sockets are easy to detect because their set of shapes is unique. an example is socket type A.
  Type A has 2 rectangles and 0 circles. and is the only type that has these set of shapes.
  But some of the sets of shapes are not unique.
  For an example, the set that consists of 3 circles appears in 4 different socket types.
  So in order to classify these kind of cases we had to figure out the relations between the circles.
  We created functions that calculate the radius of the circles, the relative locations of the circles, and function that calculates the angles
  formed by connecting the 3 points (of the center of the circles) to a triangle.

Explanation about each step:

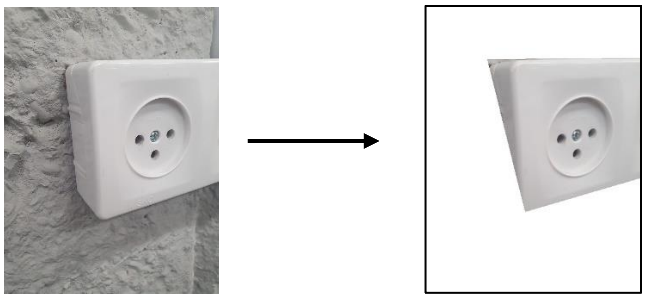
Step 1 – we used cv2.resize function.

Step 2 –

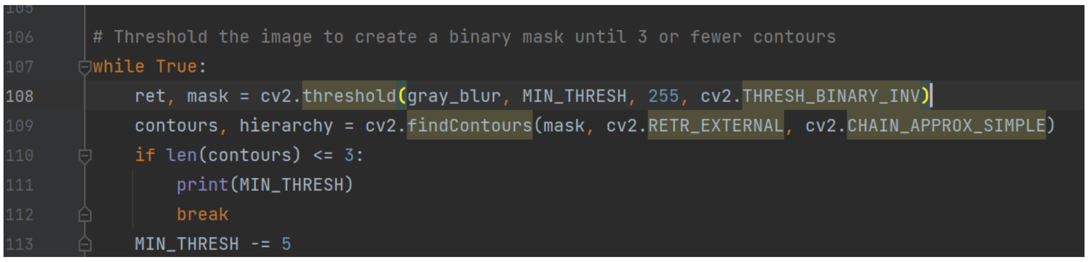


```
        # Convert the angles from radians to degrees
        angles = np.degrees(angles)

        if any(map(lambda x: x <= 52, angles)):
            threshold_value -= 20
        else:
            break
```

We checked the angle of each point of the polygon. If one of the angles is too sharp (less than 52 degrees) then we lower the threshold, to find another polygon. After many tests, we noticed that to crop the socket correctly, the angles of the polygon should be 90 degrees or bigger to avoid cropping the socket itself.



Step 3 –

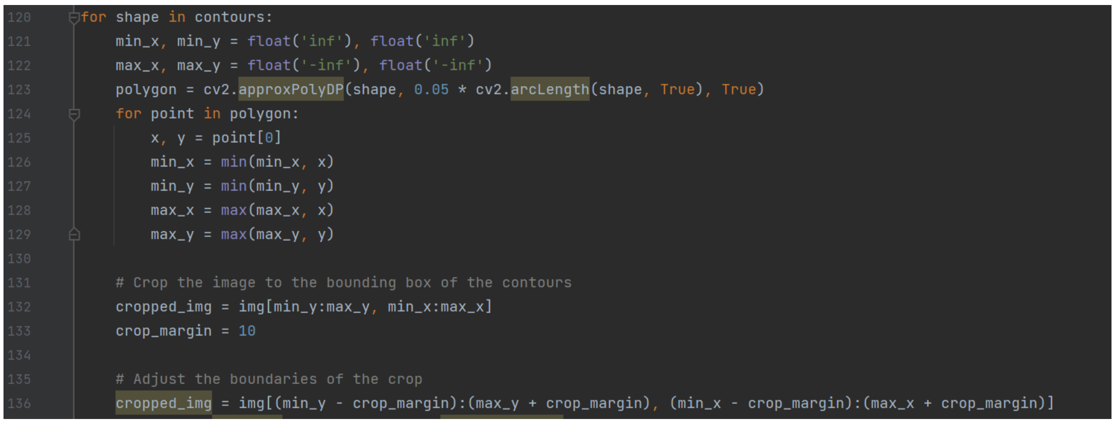


```
# Threshold the image to create a binary mask until 3 or fewer contours
while True:
    ret, mask = cv2.threshold(gray_blur, MIN_THRESH, 255, cv2.THRESH_BINARY_INV)
    contours, hierarchy = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if len(contours) <= 3:
        print(MIN_THRESH)
        break
    MIN_THRESH -= 5
```

Cv2.threshold - In a while loop we used "cv2.threshold" function with the parameter 'MIN_THRESH' that first initialized to 125. This function sets the value of dark areas (that have value of less than 'MIN_THRASH') to be 255 (white), and all the other areas to be 0 (black).

Cv2.findContours – Gets the binary image and returns 'contours' - a list of contours (outlines) of objects in an image. Each contour is represented as a list of points. We are looking for the contours of the holes. All the socket types have 2-3 holes. So, in each iteration of the while loop we check if 'len(contours)' – number of contours we found, is 3 or less. If the number of contours that we found is bigger than 3, it means that there was a dark area that is certainty not a hole. So, we decrease the 'MIN_THRESH' parameter to find only the darkest spots on the image.

Step 4 –

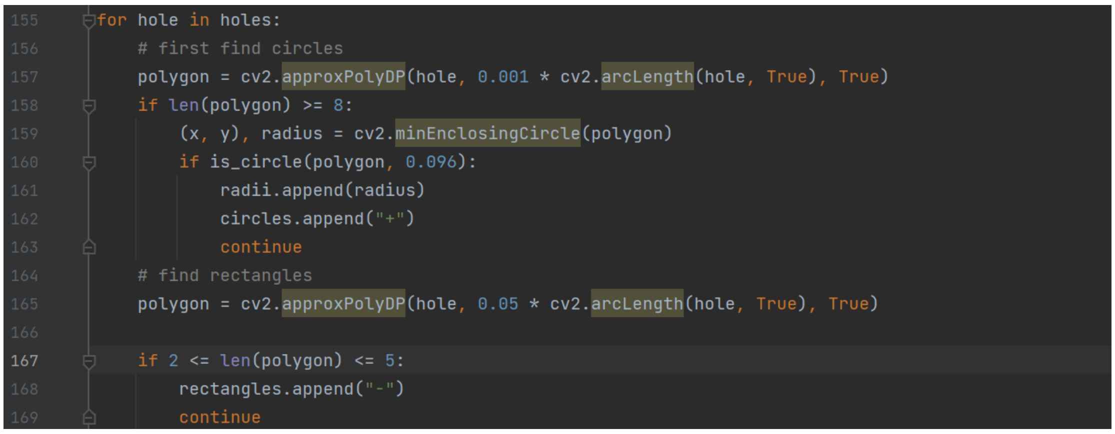


```python
for shape in contours:
    min_x, min_y = float('inf'), float('inf')
    max_x, max_y = float('-inf'), float('-inf')
    polygon = cv2.approxPolyDP(shape, 0.05 * cv2.arcLength(shape, True), True)
    for point in polygon:
        x, y = point[0]
        min_x = min(min_x, x)
        min_y = min(min_y, y)
        max_x = max(max_x, x)
        max_y = max(max_y, y)

    # Crop the image to the bounding box of the contours
    cropped_img = img[min_y:max_y, min_x:max_x]
    crop_margin = 10

    # Adjust the boundaries of the crop
    cropped_img = img[(min_y - crop_margin):(max_y + crop_margin), (min_x - crop_margin):(max_x + crop_margin)]
```

For each contour (hole) in the contours list we found in step 3 we found its polygon using "cv2.approxPolyDP" – gets a contour and returns an approximated polygonal curve with a specified precision.

For each point in the polygon, we find the maximal x,y coordinate and the minimal x,y coordinate and crop the image accordingly (with an addition of 10 pixels for each margin).

Step 5 -



```python
for hole in holes:
    # first find circles
    polygon = cv2.approxPolyDP(hole, 0.001 * cv2.arcLength(hole, True), True)
    if len(polygon) >= 8:
        (x, y), radius = cv2.minEnclosingCircle(polygon)
        if is_circle(polygon, 0.096):
            radii.append(radius)
            circles.append("+")
            continue
    # find rectangles
    polygon = cv2.approxPolyDP(hole, 0.05 * cv2.arcLength(hole, True), True)

    if 2 <= len(polygon) <= 5:
        rectangles.append("-")
        continue
```

The purpose of this step is to classify whether the hole is a circle or a rectangle.

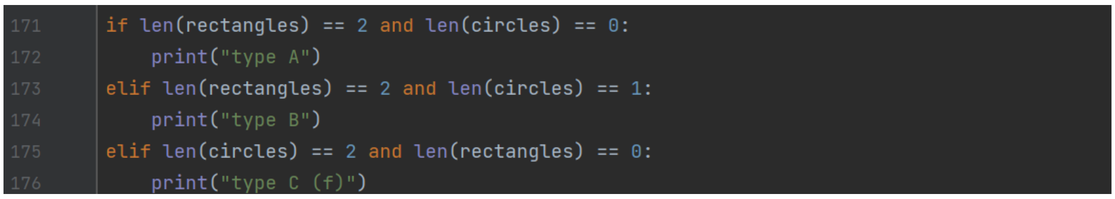
We found that one of the differences between a shape of a circle and a shape of a rectangle is the number of points of a polygon. A rectangle's polygon has 4 points according to its 4 edges (straight lines), whereas a circle has curved boundary – hence, circle has an infinite number of points on their boundary.

For that, we used the **"cv2.approxPolyDP"** function with different value of the precision parameter. When trying to detect circles we used higher precision to get as many points as we can, and only those with number of points bigger than 8 is a candidate to be classified as a circle. When trying to classify a rectangle we used a lower precision to ignore small interferences and noises in the contour that may alter the outcome. While trying to ignore noises, we found that the number of the points the "cv2.approxPolyDP" function finds for a rectangle is not always 4. So, to allow mistakes we enlarged the range of acceptable points.

Is_circle – Function we created that gets a polygon and a threshold parameter and returns true if the calculated match score is less than the threshold, false otherwise. We calculate the perimeter (using "cv2.arcLength" function) and the area of the polygon (using "cv2.contourArea" function). Then, using the area we found before we calculated the perimeter it should have as a circle.

$desirable_{perimeter} = 2 * \pi * \sqrt{\frac{area}{\pi}}$. Now we check the Δ (difference) between the desirable perimeter and the found perimeter. if the Δ is less than the threshold parameter then we return True, False otherwise.

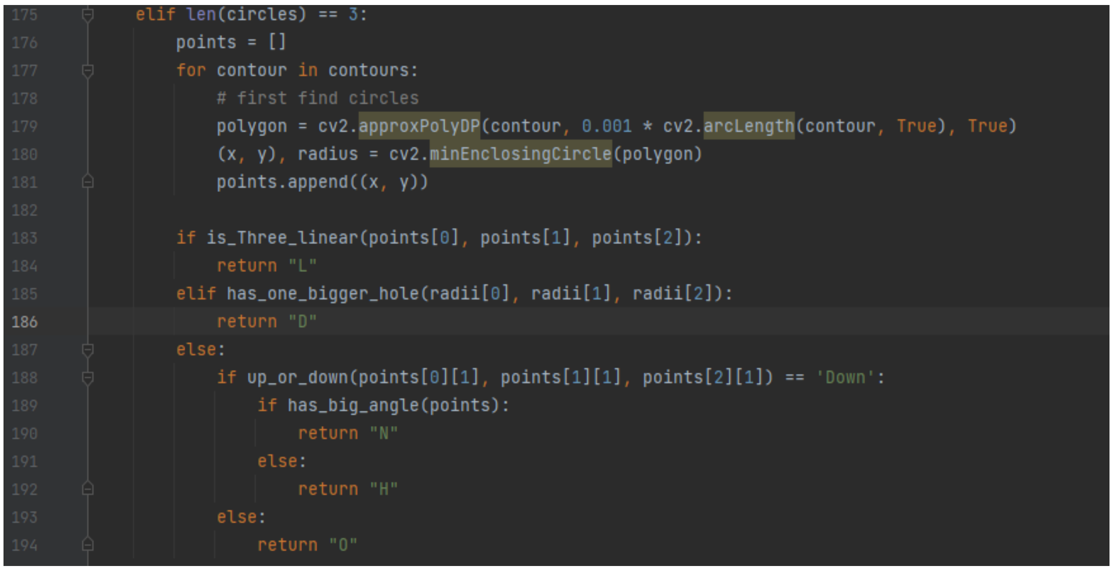
Step 6 –



```
if len(rectangles) == 2 and len(circles) == 0:
    print("type A")
elif len(rectangles) == 2 and len(circles) == 1:
    print("type B")
elif len(circles) == 2 and len(rectangles) == 0:
    print("type C (f)")
```

After classified each hole (as a circle or a rectangle). We now have the set of the shapes. We created a list of cases to check and determine the type of the socket. For a unique set of shapes, we determined the type without any other tests.

For the case of 3 circles, there are 5 relevant types of sockets. So, we used this block of code below:



```
    elif len(circles) == 3:
        points = []
        for contour in contours:
            # first find circles
            polygon = cv2.approxPolyDP(contour, 0.001 * cv2.arcLength(contour, True), True)
            (x, y), radius = cv2.minEnclosingCircle(polygon)
            points.append((x, y))

        if is_Three_linear(points[0], points[1], points[2]):
            return "L"
        elif has_one_bigger_hole(radii[0], radii[1], radii[2]):
            return "D"
        else:
            if up_or_down(points[0][1], points[1][1], points[2][1]) == 'Down':
                if has_big_angle(points):
                    return "N"
                else:
                    return "H"
            else:
                return "O"
```

Because 3 circles is not a unique set of shapes, we need to know more information about the relations between the circles. We used **"cv2.minEnclosingCircle"** that gets a polygon and returns the radius and the center of the circle.

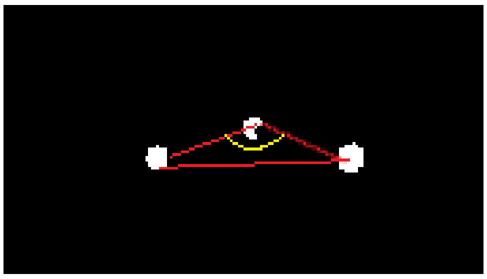
After discovering the locations and radiuses of the circles in the image, we created our own functions to determine the socket's type.

"is_Three_Linear" – gets 3 points in R^2 and returns True if they are located on the same line. Returns False otherwise. We check the slopes between point 1 to point 2 and between point 2 and point 3. And check if the difference between the slopes is less than a value we chose (0.1). if True – the type is L. if false, we continue to check other tests.

"has_one_bigger_hole" – gets 3 radiuses (one of each circle) and returns True if only one hole is bigger than the other two. In this function, we used constant values to check the difference between the radiuses of the circles. It was possible to use constant values because we resized the image (step 1) to be a specific size. First, we checked which 2 circles have similar radius, and then checked the difference between the sizes of one of them and the third circle's radius.

"up_or_down" – gets the y coordinates of 3 circles. Returns "up" if the single hole is above the two others, and "down" otherwise.

has_big_angle – gets the center point of each one of the 3 circles. Returns True if one of the angles is bigger than 100 degrees, returns False otherwise.



For the case of 3 rectangles, there are 2 options (type G and type I). the difference between them is the relative location of the single rectangle (whether it is above the 2 rectangles, or below them. We check this with our function "up_or_down".

Main flow of the program:

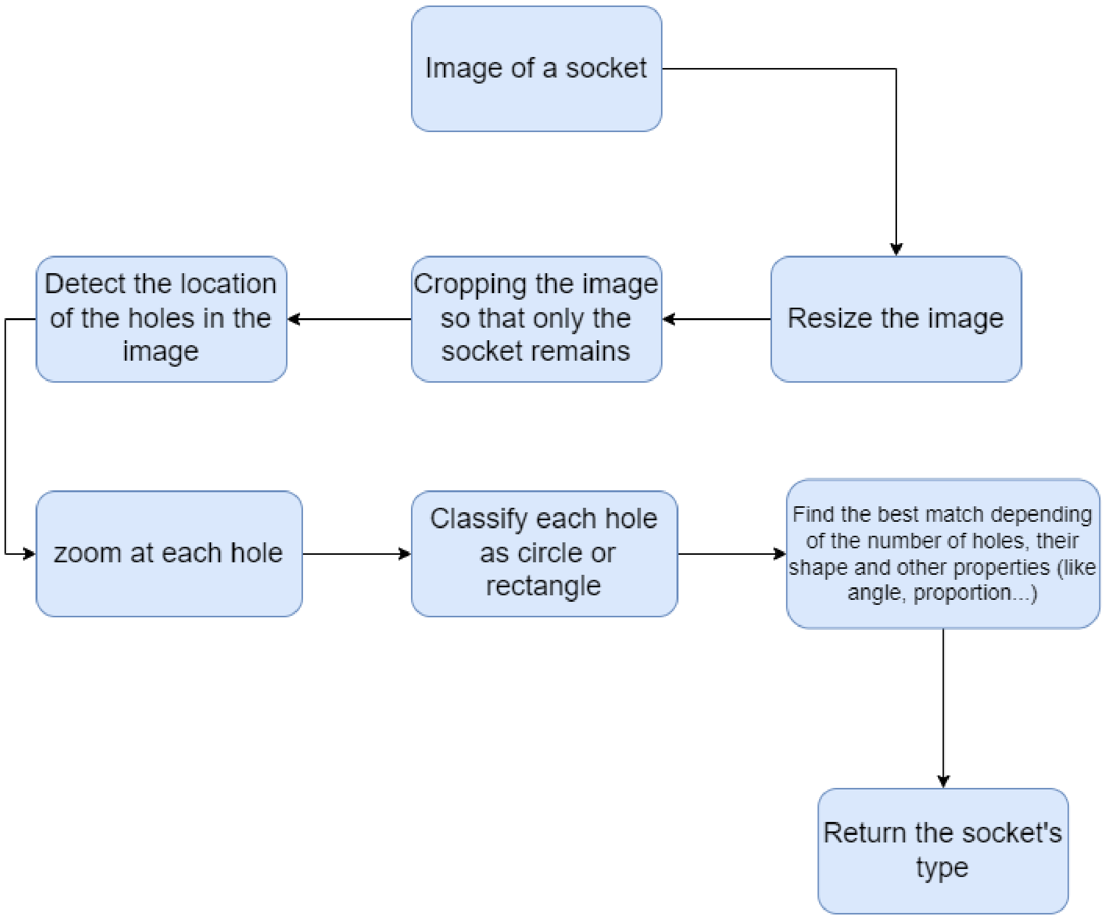
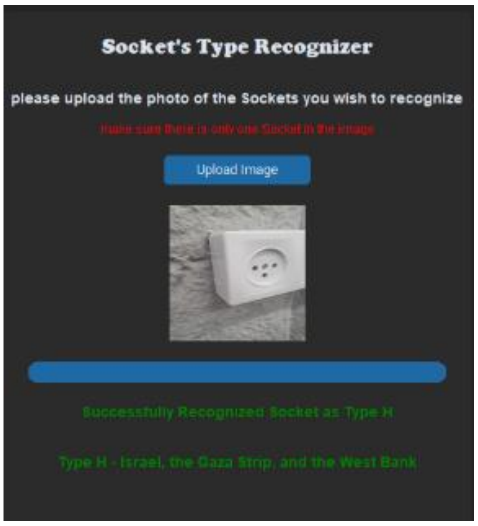


Image of a socket
Resize the image
Cropping the image so that only the socket remains
Detect the location of the holes in the image
zoom at each hole
Classify each hole as circle or rectangle
Find the best match depending of the number of holes, their shape and other properties (like angle, proportion...)
Return the socket's type

Results:

Out of the 15 types of sockets exist, we classified 12 of them, with a success rate of 84 precent.
The types that we did not deal with had the same holes shape as other plugs but had a difference at the surface of the socket.
In this project we preferred to focus on the differences between the holes of a socket.
Among the types we have dealt with, we checked 3 images per type. 36 images total. 6/36 images were not classified correct, because of shading that was too dark, and because difficulties cropping the wall.



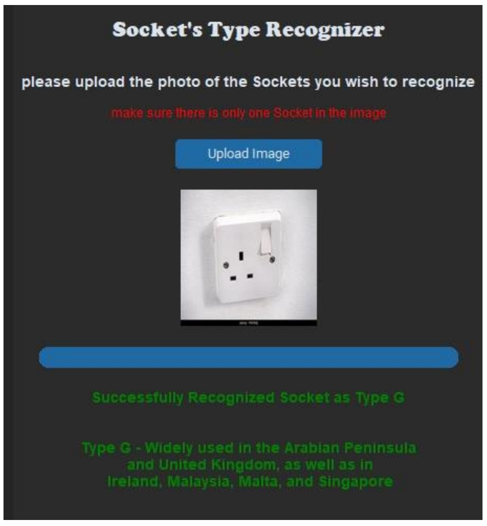
Socket's Type Recognizer

please upload the photo of the Sockets you wish to recognize

make sure there is only one Socket in the image

Successfully Recognized Socket as Type H

Type H - Israel, the Gaza Strip, and the West Bank



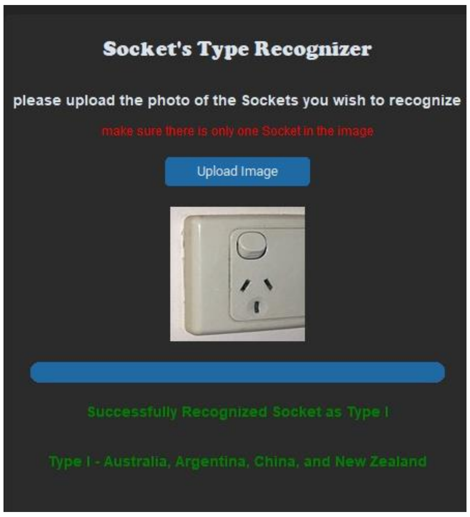
Socket's Type Recognizer

please upload the photo of the Sockets you wish to recognize

make sure there is only one Socket in the image

Successfully Recognized Socket as Type G

Type G - Widely used in the Arabian Peninsula
and United Kingdom, as well as in
Ireland, Malaysia, Malta, and Singapore



Socket's Type Recognizer

please upload the photo of the Sockets you wish to recognize

make sure there is only one Socket in the image

Successfully Recognized Socket as Type I

Type I - Australia, Argentina, China, and New Zealand

Conclusions:

Our goal was to succeed to classify all the 13 types of sockets that have difference between the holes in them.
However, when facing socket type K, we found it difficult to deal with the bottom hole that has a shape of a semi-circle.
One of our goals was to deal with images that were taken from different angles. for that reason, some images of holes with circle shape
that was taken from certain angles, classified by mistake as a semi-circle.
We realized there is a trade-off between dealing with angles, and classifying semi-circles.
so we chose to deal with angles instead of recognizing the k type. As a result, instead of dealing with 13 types of sockets, we dealt with 12 types.
This computer vision project was very educational and challenging to build.
as we used classic techniques learned during the course to perform detection and recognition, getting satisfying results even when using difficult images.
Even though the application is good, there are still improvements to be made:

- Deal with socket types K, E and F.
- Improve the Cropping algorithm.
- Enable images of multiple sockets.
- Use of ANN and Deep Learning (not learned during the course).