Static Hand Pose Detection

Ana Rita Torres
up201406093@fe.up.pt
Nuno Martins
up201405079@fe.up.pt
Rui Pedro
up201404965@fe.up.pt

VCOM
Faculdade de Engenharia da Universidade do Porto
Porto, PT

Abstract

As computers become more integrated with human life, humans are encouraged to interact more and more with the systems. To facilitate such an interface between the two, easy interaction could be guaranteed by the recognition of hand gestures - ever so present in human-to-human communication. As such, the tool developed herein based on previous works by other authors serves as a way of having the machine recognize what the human gestures to it. It was developed taking into consideration the first project description of the VCOM subject.

The paper serves as a description of the algorithms employed for the creation of such a tool and to analyze some potential improvements beyond the ones developed.

1 Introduction

Such as described in the VCOM lectures, image segmentation techniques along with some other manipulation tools could lead to powerful results if applied well. The program herein developed can identify hands in an image and count how many fingers they are holding. This was done by a mixture of such image manipulation techniques, that we describe in detail in the next sections.

To run the program, one should have Python3.x installed and open a terminal window in src folder:

- `python main.py camera` to read hand gestures from a picture taken from the computer camera, pressing Space to take a picture on Esc to exit.
- `python main.py file -i 'path'` to read hand gestures from a loaded picture `path`
- Running the previous commands with addition of the parameter `-a` will run a more computationally heavy color segmentation technique and attain better results than default while taking more than ten times the time to run the segmentation.
- `python main.py -h` to get help on how to run the program displayed in the console.

2 Features

The development of this solution took into account the requisites presented in the project proposal, as well as possible improvements mentioned.

In the end, the program is able to:

- allow the acquisition of a color image containing one hand, using a computer connected camera, or the selection of a pre-acquired image;
- segment the acquired image, isolating the hand from the background, as long as the background is not skin colored
- recognize the hand pose from a set of admissible poses, using different datasets, identifying how many fingers are up, including the thumb separately
- detect multiple hands in the same image, as long as they take up at least 20% of total width and height of the image and don't intercept each other
- detect hands in multiple orientations that are not diagonals
- detect image in dynamic backgrounds, as long as no significant skin colored object intercepts the hands
- detect hands with palm facing back or towards the camera

3 Techniques

To achieve the best results, a wide range of techniques were tested to iteratively solve the presented problems that would appear as we segmented the image. The final result consists of the following techniques, in respective order of application

3.1 Image Resize

To make it easier to implement the algorithms and to allow us to use static values when segmenting the image, we set the image to always be resized to 400x400 pixels, serving as the standard resolution to perform the operations.

3.2 Color Smoothing

To make it easier to use color segmentation to detect the hands, we used color smoothing techniques to smooth out the colors. This is needed to remove unwanted lines that would be present in the image and interfere with the detection, like fur and hand lines, as well as noise resulting from the usage of a conventional camera.

To achieve this result, we used Gaussian Blur with a 5x5 kernel, which was the one that produced better results.

3.3 Color Segmentation

To identify the hands in the image, we used color segmentation in different color spaces.

The first method was to use static thresholding in the HSV colorspace, being the thresholds:

- $0<H<35$ (H between 0 and 180)
- $42<S<173$ (S between 0 and 255)
- $60<V<255$ (V between 0 and 255)

The advantage of using the HSV color space over RGB is that it is easier to ignore the brightness(V). This method is fast and very effective against constant backgrounds. But a problem arises when different illumination sources influence the image. Natural light and artificial light produce very different color saturation gaps, effectively rendering this technique ineffective.

To solve this problem, we integrated an alternative and optional method that is computationally heavier which combines different thresholds in different color spaces(RGB, YCrCb and HSV) that take into account artificial and natural illumination patterns, producing much better results. [2]

If one of the following threshold verifies for a certain pixel on the image, it is considered that it is skin:

- $0.0 <= H <= 50.0$ and $0.23 <= S <= 0.68$ and $R > 95$ and $G >40$ and $B > 20$ and $R > G$ and $R >$ Band $| R - G | > 15$

- R > 95 and G > 40 and B > 20 and R > G and R > Band | R - G | > 15 and A > 15 and Cr > 135 and Cb > 85 and Y > 80 and Cr <= (1.5862*Cb)+20 and Cr>=(0.3448*Cb)+76.2069 and Cr >= (-4.5652*Cb)+234.5652 and Cr <= (-1.15*Cb)+301.75 and Cr <= (-2.2857*Cb)+432.85

As observed in the Figure1(a), this technique produces much better results, having some artifacts nonetheless.

3.4 Morphological transformations

Morphological transformations are also useful to remove unwanted artifacts in the image.

Image erosion followed by dilation is a useful technique to remove unwanted small objects in the image, as well as thin lines connecting diferent objects, for example diferent hands connected by an unwanted artifacts in the image undetected in color segmentation.

Median blur was used to capture small black spots in the white segmentation of the hand.

Connected components technique was used to detect the biggest components of the image, which are presumed to be the hands, followed by removal of smaller components which are likely artifacts captured by color segmentation.

A combination of these 3 techniques to remove unwanted blocks produced very good results in images whose backgrounds are not skin colored, as seen in Figure1(a)

3.5 Hand Orientation

Which way the hand is oriented is important in detecting the hand shape. To check how many fingers are raised, we need to know where to expect them in the image. So, figuring out how the hand was oriented was relevant to the remaining of the process.

Our method has a simple assumption, that is, the hand comes from one of the four sides of the image. That is, the arm containing the hand extends one way, and this is the way to verify how it is oriented. Also, in determining this orientation it doesn't matter if it is a left or a right hand.

We check the first line of pixels in each of the four sides of the image, that is:

- Hand oriented bottom-up: checked by the last row of pixels in the image
- Hand oriented top-down: checked by the first row of pixels
- Hand oriented left to right: checked by the first column of pixels
- Hand oriented right to left: checked by the last column of pixels

After grabbing each of these four values we check the largest one and assume that is the way the hand is facing. This prevents assuming the hand is oriented to one side just because one finger is clipped away from the image, for example.

We used images likes Figure 1(b) to test such orientations, even in the case of multiple hands.

3.6 Image Structural Analysis

In order to check the shape of the hand we decided OpenCV's convex hulls were the way to do this. They create a polygon around the hand, and create vertices on local maximums of the shape. This is relevant because it creates vertices around the fingers and the base of the hand.

However, some grouping of these local maximums was needed in order to trim the polygon into having less vertices and serving more use for the next step.

The convex hull can be seen as the dark blue polygon around the green countour in Figure 1(a) and (c).

3.7 Finger Detection

After grabbing the convex hull that encircles the hand shape, we could apply an algorithm to detect how many fingers were raised. The algorithm applied is based on Meenakshi Panwar and Pawan Mehra's paper [1] on the same subject of this one. The method suggests firstly the calculation of the centroid of the hand shape (the obtained convex hull). This is done by simply averaging the position of each vertex of the convex hull shape. Note that this centroid will adapt to how many fingers were raised by this property alone, given that the amount of vertices on the finger part of the hull shape will amount to the vertical position of the centroid.

After having the centroid's position, what was required was the detection of peaks in height above the centroid. Assuming the hand is presented bottom-up, that is, the fingers on the top and the palm on the bottom, the centroid will be a little above the middle of the image's height. Thus, what comes below the centroid is not relevant since the fingers should be on top. From here, we take each vertex that is above the centroid and we save them (we call them **peaks**). Then, we check which one is farther up and we store the vertical distance between this point and the centroid height. Fingers raised up shouldn't be two distanced vertically, but folded fingers and the thumb should. So, to difference what fingers are raised and folded we take the maximum vertical distance mentioned earlier and we consider as a raised finger only such peaks that are above 75% of this height. The final peaks on the list should be how many fingers we have raised.

In Figure 1(c) one can see in the finger vertexes of the convex hull a red color, which indicates that a finger was detected.

3.8 Thumb Detection

Also based on the same paper as the previous section, the authors mention a method of having thumb detection, different from the finger detection. This is relevant because the thumb usually rests much lower than the rest of the fingers, and as such will not be detected by the previous method.

The method developed envolves the scanning of the hand ROI for the thumb. We used a small window with size relative to the image size and we compared the amount of white pixels in that window to those of the complete hand. We then used a percentage as a threshold value for checking the thumb. This percentage is lower when there are less fingers raised (since it would be easier to check when there were no fingers raised). If the amount of white pixels in that window is lower than the threshold, the detector returns true. If not, it assumes there is no raised thumb (See Figure 1 (c) and returns false.

This method is not perfect and it is easily tricked. We figured that given some assumptions about how the hand should be detected, the algorithm works well in most normal (palm facing the camera) cases.

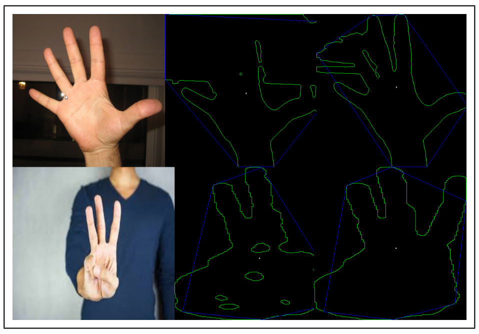
4 Difficulties

Given that the task of detecting human gestures is a very complex one, the solution is based on some assumptions (some discussed earlier). These are:

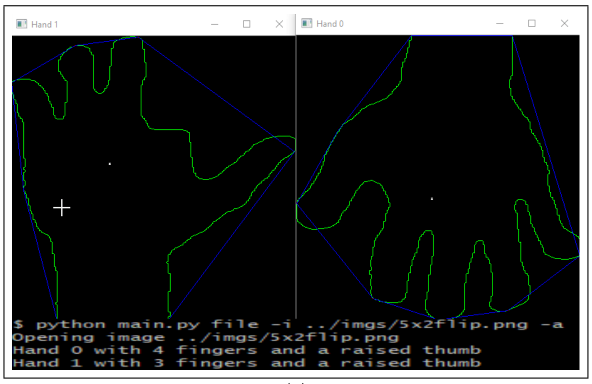
- The hand is oriented along one of the sides, and not in a diagonal.
- If orientation is undetermined, default to bottom-up orientation.
- The background isn't significantly human skin colored.
- Multiple hands in the same image don't intercept.
- Hands occupy a reasonable space in the image.
- Illumination doesn't have unusually colored lights, like blue and green, only yellow and red.
- Raised fingers are raised all the way up.
- Nothing is in front of the hand.
- Gesture to determine is a simple finger count.

4.1 Solution Reached

The final program can easily count how many fingers are raised up and whether the thumb is raised too, and does this reasonably effectively (if taking into consideration the assumptions). It can also detect more than one hand in one picture and can even separate the hand from its background reasonably well.


(a)

(b)



```
$ python main.py file -i ../imgs/5x2flip.png -a
Opening image ../imgs/5x2flip.png
Hand 0 with 4 fingers and a raised thumb
Hand 1 with 3 fingers and a raised thumb
```

(c)

Figure 1: (a) First vs second color segmentation methods applied to the segmentation of images with different backgrounds. Although the first test produces much better results with the second method, they are still not perfect, detecting only 3 fingers and a thumb.(b) (c) This implementation can detect multiple hands in the same image, with different orientations and precisely tell how many fingers are up

4.2 Limitations of the Solution

The solution reached has some limitations, mainly the detection of raised fingers, since it is based on the convex hull shape that is a little unpredictable. Also the method for checking whether the thumb is raised is quite crude, and could be substituted by another, more effective method. It is based on a percentage of pixels, so images that have a percentage of pixels in the search window lower than the one required will flag true to the thumb being raised, even if it isn't.

References

[1] M. Panwar and P. Mehra. Hand gesture recognition for human computer interaction. In *Proc. ICIIP*, 2011.

[2] P.Shimpi C.Bapat and J. Jatakia S.Kolkur, D. Kalbande. Human skin detection using rgb, hsv and ycbcr color models.

5 Appendix

.1 main.py - main file

```
import sys
from cv2 import *
from matplotlib import pyplot as plt
import argparse
import algorithm
import image


parser = argparse.ArgumentParser(description="Detect
    number of fingers in hands from image file or
    camera")
parser.add_argument('method' ,help='\'camera\' or
    \'file\'', type=str)
parser.add_argument('-i', '--image', dest='path',
    default='img.jpg',type=str)
parser.add_argument('-a', '--advanced', dest='adv',
    action='store_true')
img = None

args = parser.parse_args()
arg = args.method
if arg == 'camera':
   img = image.CaptureCameraImage()
elif arg == 'file':
   print('Opening image ' + args.path)
   img = image.ReadImageFile(args.path)

if img is None:
   print('Image not found')
   quit()

img = algorithm.ResizeImage(img)
img = algorithm.SmoothImage(img)
imgs = algorithm.DetectHands(img, args.adv)
original = img.copy()
if len(imgs) == 0:
   print('No hands detected')
else:
   print('Detected ' + str(len(imgs)) + ' hand(s)')
for i in range(0, len(imgs)):
   x1,y1,x2,y2 = algorithm.GetRectEdges(imgs[i])
   tl = [x1,y1]
   br = [x2,y2]
   original = algorithm.DrawRectangle(original,tl,br)
   segment = algorithm.GetRectSection(imgs[i], tl, br)
   segment = algorithm.ResizeImage(segment)
   hand, fingers, thumb =
       algorithm.DetectGestures(segment)
   strfingers = 'finger' if fingers == 1 else
       'fingers'
   strthumb = 'a raised thumb' if thumb else 'no
       thumb'
   print('Hand ' + str(i) + ' with ' + str(fingers) +
       ' ' + strfingers + ' and ' + strthumb)
   imshow('Hand ' + str(i),hand)
imshow('Image', original)
sys.stdout.flush()
if len(imgs) > 0:
   waitKey(0)
```

.2 image.py - methods to get images

```
from cv2 import *

"""Returns image from camera when Space is pressed,
    or None when Esc is pressed
"""
def CaptureCameraImage():
   cap = VideoCapture(0)
   img = None
   while(True):
      ret, frame = cap.read()
      cv2.imshow('frame', frame)
      key = waitKey(1)
      space_key = 32
      esc_key = 27
      if key == space_key:
         img = frame
         break
      if key == esc_key:
         break
   cap.release()
   destroyAllWindows()
   return img
"""Reads image from input file
"""
def ReadImageFile(path):
   img = imread(path)
   return img
```

.3 algorithm.py - methods to perform segmentation and detect gestures

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

np.set_printoptions(threshold=np.nan)

def SmoothImage(img):
   blur = GaussianBlur(img,(5,5),0)
   return blur

def ResizeImage(img):
   img2 = resize(img, (400,400))
   return img2

def ErodeAndDilateImg(img):
   kernel1Size_x = 3
   kernel1Size_y = 3
   kernel =
       np.ones((kernel1Size_x,kernel1Size_y),np.uint8)
   erosion = erode(img, kernel, iterations=3)
   dilation = dilate(img, kernel, iterations=3)
   return erosion

def DetectHands(img, advanced=False):
   ranges = None

   if advanced:
      ranges = GetSkin(img)
   else:
      ranges = GetSkinEasy(img)

   ranges = medianBlur(ranges, ksize=9)

   #find small components not caught by median blur
   nb_components, output, stats, centroids =
       cv2.connectedComponentsWithStats(ranges,
       connectivity=4)
   #connectedComponentswithStats yields every
       seperated component with information on each
       of them, such as size
   #the following part is just taking out the
       background which is also considered a
       component, but most of the time we don't want
       that.
   #remove background
   nb_components = nb_components - 1
   biggestone = -1
   biggesti = -1
   for i in range(0, len(stats)):
      if stats[i][4] > biggestone:
         biggesti = i
         biggestone = stats[i][4]
   newstats = []
   for i in range(0, len(stats)):
      if i != biggesti:
         newstats.append(stats[i])
   newstats = np.array(newstats)

   # minimum size of particles we want to keep
       (number of pixels)
   #here, it's a fixed value, but you can set it as
       you want, eg the mean of the sizes or whatever
   min_size = 100*100
   imgs = []
   #your answer image
   #img2 = np.zeros((output.shape),np.uint8)
   #for every component in the image, you keep it
       only if it's above min_size
   for i in range(0, nb_components):
        if newstats[i][4] >= min_size:
            hand = np.zeros((output.shape),np.uint8)
            hand[output == i + 1] = 255
            imgs.append(hand)

    return imgs

def GetSkinEasy(img):
    img2 = cvtColor(img, COLOR_BGR2HSV)
    x_min = 0
    x_max = 35
    y_min = 42
    y_max = 173
    z_min = 60
    z_max = 255
    ranges = inRange(img2, np.array([x_min, y_min,
        z_min]), np.array([x_max, y_max, z_max]))
    return ranges

def nmax(x1,x2):
    return x1 if x1 > x2 else x2

def nmin(x1,x2):
    return x1 if x1 < x2 else x2

def TR1(R,G,B,Y,Cr,Cb,H,S,V):
    bol = (H > 0 and H < 50) and (S > 58 and S < 173)
        and R > 95 and G > 40 and B > 20 and R > G and
        R > B and abs(int(R) - int(G)) > 15
    return bol

def TR2(R,G,B,Y,Cr,Cb,H,S,V):
    bol = R > 95 and G > 40 and B > 20 and R > G and R
        > B and abs(int(R) - int(G)) > 15 and Cr > 135
        and Cb > 85 and Y > 80 and Cr <=
        (1.5862*Cb)+20 and Cr>=(0.3448*Cb)+76.2069
    bol2 = Cr >= (-4.5652*Cb)+234 and Cr <=
        (-1.15*Cb)+301 and Cr <= (-2.2857*Cb)+432
    return bol and bol2

def GetSkin(img):
    rgb = img.copy()
    hsv = cvtColor(img, COLOR_BGR2HSV)
    ycrcb = cvtColor(img, COLOR_BGR2YCrCb)

    dest = np.zeros((img.shape[0],img.shape[1]),
        np.uint8)

    for y in range(0,img.shape[0]):
        for x in range(0,img.shape[1]):
            r = rgb[y][x][2]
            g = rgb[y][x][1]
            b = rgb[y][x][0]

            Y = ycrcb[y][x][0]
            cr= ycrcb[y][x][1]
            cb= ycrcb[y][x][2]

            h = hsv[y][x][0]
            s = hsv[y][x][1]
            v = hsv[y][x][2]

            tr1 = TR1(r,g,b, Y, cr,cb,h,s,v)

            tr2 = TR2(r,g,b, Y, cr,cb,h,s,v)

            if tr1 or tr2:
                dest[y][x] = 255
    return dest


def DetectGestures(img):
    _, contours, hierarchy = findContours(img,
        RETR_TREE, CHAIN_APPROX_SIMPLE)
    hull = []
    color_centroids = (255,255,255)
    color_peaks = (0,0,255)
    color_contours = (0,255,0)
    color = (255,0,0)

    for i in range(len(contours)):
        hull.append(RemoveRepeatedPoints(convexHull(contours[i],False)))

    drawing = np.zeros((img.shape[0], img.shape[1],
        3), np.uint8)
    biggest_centroid, bc_index =
        GetBiggestCentroid(hull)
```

```
    orientation = GetOrientation(img)

    if biggest_centroid == []:
        return img, 0, False
    peaks = GetPeaks(biggest_centroid, hull[bc_index],
        orientation)
    # Draw Centroid
    drawing[biggest_centroid[1]][biggest_centroid[0]]
        = color_centroids
    # To see it slightly better, draw pixels around it
        in same color
    drawing[biggest_centroid[1] +
        1][biggest_centroid[0]] = color_centroids
    drawing[biggest_centroid[1] -
        1][biggest_centroid[0]] = color_centroids
    drawing[biggest_centroid[1] +
        1][biggest_centroid[0] + 1] = color_centroids
    drawing[biggest_centroid[1] +
        1][biggest_centroid[0] - 1] = color_centroids
    drawing[biggest_centroid[1] -
        1][biggest_centroid[0] + 1] = color_centroids
    drawing[biggest_centroid[1] -
        1][biggest_centroid[0] - 1] = color_centroids

    drawContours(drawing,contours,bc_index,color_contours,1,8,h
    drawContours(drawing,hull,bc_index,color,1,8)

    for i in range(len(peaks)):
        drawing[peaks[i][1]][peaks[i][0]] = color_peaks

    window_x = 0
    window_y = 0

    thumb = False

    if orientation == 'TOPDOWN' or orientation ==
        'BOTTOMUP':
        window_size_x = int(math.floor(0.2 *
            img.shape[1]))
        window_size_y = img.shape[0]

        for i in range(0, 5):
            thumb = GetThumb(img, i*window_size_x +
                window_x, window_y, window_size_x,
                window_size_y, len(peaks))
            if thumb:
                break
    else:
        window_size_x = img.shape[1]
        window_size_y = int(math.floor(0.2 *
            img.shape[0]))
        for i in range(0, 5):
            thumb = GetThumb(img, window_x,
                i*window_size_y + window_y,
                window_size_x, window_size_y, len(peaks))
            if thumb:
                break

    nfingers = len(peaks)
    return drawing, nfingers, thumb

def RemoveRepeatedPoints(hull):
    if len(hull) == 1:
        return hull
    groups = []
    current_group = []
    max_dist = 70
    for i in range(0, len(hull)):
        if len(current_group) == 0:
            current_group.append(hull[i])
            continue
        currx = hull[i][0][0]
        curry = hull[i][0][1]
        latestx =
            current_group[len(current_group)-1][0][0]
        latesty =
            current_group[len(current_group)-1][0][1]
        #do manhattan distance to check if points are
            close
        if abs(abs(currx - latestx) + abs(curry -
            latesty)) < max_dist:
            current_group.append(hull[i])
        #discard close points and keep only the one in
            the center
        else:
            groups.append(current_group[math.floor(len(current_g
         current_group = []
         current_group.append(hull[i])
   if len(current_group) > 0:
      currx = hull[0][0][0]
      curry = hull[0][0][1]
      latestx =
          current_group[len(current_group)-1][0][0]
      latesty =
          current_group[len(current_group)-1][0][1]
      if len(groups) > 0 and abs(abs(currx - latestx)
          + abs(curry - latesty)) >= max_dist:
         groups.append(current_group[math.floor(len(current_group)/2)])
      elif len(groups) == 0:
         groups.append(current_group[math.floor(len(current_group)/2)])
   return np.array(groups, dtype=np.int32)

def GetCentroid(hull):
   centroids = []
   for h in hull:
      x = []
      y = []

      for point in h:
         x.append(point[0][0])
         y.append(point[0][1])
      centroids.append((int(round(np.mean(x))),
          int(round(np.mean(y)))))
   return centroids

# Gets biggest centroid based on average distance to
    vertices
# --> ONLY RELEVANT IF ASSUMING HAND AS BIGGEST
    CENTROID
def GetBiggestCentroid(hull):
   centroids = GetCentroid(hull)
   max_avg = 0
   biggest_centroid = []
   max_i = 0
   for i in range(len(centroids)):
      dist_x = 0
      dist_y = 0
      x_centroid = centroids[i][0]
      y_centroid = centroids[i][1]
      for point in hull[i]:
         dist_x += abs(x_centroid - point[0][0])
         dist_y += abs(y_centroid - point[0][1])
      cur_avg = (dist_x / len(hull[i]) + dist_y /
          len(hull[i])) / 2
      if cur_avg > max_avg:
         max_avg = cur_avg
         biggest_centroid = [x_centroid, y_centroid]
         max_i = i
   return biggest_centroid, max_i

# Given the centroid and the hull, calculate peaks
    and filter irrelevant peaks (below 25% of dist
    between centroid and the biggest peak)
# centroid = [x_centroid, y_centroid]
# points = [ [[x_point1, y_point1]], [[x_point2,
    y_point2]], ...]
def GetPeaks(centroid, points, orientation):
   peaks = []
   # Get max VERTICAL peak
   max_peak = 0
   if orientation == 'BOTTOMUP':
      for i in range(len(points)):
         if points[i][0][1] < centroid[1]:
            if abs(points[i][0][1] - centroid[1]) >
                max_peak:
               max_peak = abs(points[i][0][1] -
                   centroid[1])
      for i in range(len(points)):
         if (centroid[1] - points[i][0][1]) >= 0.3 *
             max_peak:
            peaks.append(points[i][0])
      GetPeakPlot(centroid, peaks)
      return peaks
   elif orientation == 'RIGHTLEFT':
      for i in range(len(points)):
         if points[i][0][0] < centroid[0]:
            if abs(points[i][0][0] - centroid[0]) >
                max_peak:
               max_peak = abs(points[i][0][0] -
                   centroid[0])
      for i in range(len(points)):
         if (centroid[0] - points[i][0][0]) >= 0.3 *
             max_peak:
            peaks.append(points[i][0])
      GetPeakPlot(centroid, peaks)
      return peaks
   elif orientation == 'TOPDOWN':
      for i in range(len(points)):
         if points[i][0][1] > centroid[1]:
            if abs(centroid[1] - points[i][0][1]) >
                max_peak:
               max_peak = abs(points[i][0][1] -
                   centroid[1])
      for i in range(len(points)):
         if (points[i][0][1] - centroid[1]) >= 0.3 *
             max_peak:
            peaks.append(points[i][0])
      GetPeakPlot(centroid, peaks)
      return peaks
   elif orientation == 'LEFTRIGHT':
      for i in range(len(points)):
         if points[i][0][0] > centroid[0]:
            if abs(centroid[0] - points[i][0][0]) >
                max_peak:
               max_peak = abs(points[i][0][0] -
                   centroid[0])
      for i in range(len(points)):
         if (points[i][0][0] - centroid[0]) >= 0.3 *
             max_peak:
            peaks.append(points[i][0])
      GetPeakPlot(centroid, peaks)
      return peaks

def GetPeakPlot(centroid, peaks):
   N = len(peaks)
   peaksx = []
   peaksy = []
   for i in range(N):
      peaksx.append(peaks[i][0])
      peaksy.append(peaks[i][1])

   colors = [[0,0,0]]
   area = np.pi*3
   plt.scatter(peaksx, peaksy, s=area, c=colors,
       alpha=0.5)
   plt.title('Scatter Plot of Peaks')
   plt.xlabel('x')
   plt.ylabel('y')
   plt.gca().invert_yaxis()
   #plt.show()

   return centroid

def GetOrientation(img):
   left_count = 0
   right_count = 0
   top_count = 0
   bot_count = 0

   for y in range(0, img.shape[0]):
      if img[y][0] == 255:
         left_count += 1
      if img[y][img.shape[1]-1] == 255:
         right_count += 1

   for x in range(0, img.shape[1]):
      if img[0][x] == 255:
         top_count += 1
      if img[img.shape[0] - 1][x] == 255:
         bot_count += 1

   if left_count > right_count and left_count >
       top_count and left_count > bot_count:
      return 'LEFTRIGHT'
   if right_count > left_count and right_count >
       top_count and right_count > bot_count:
      return 'RIGHTLEFT'
   if top_count > right_count and top_count >
       left_count and top_count > bot_count:
      return 'TOPDOWN'
   if bot_count > right_count and bot_count >
       top_count and bot_count > left_count:
      return 'BOTTOMUP'

def GetWhitePixelCount(img):
   count = 0
   height, width = img.shape[0], img.shape[1]
   for y in range(height):
      for x in range(width):
         if img[y][x] == 255:
            count += 1
   return count

def GetThumb(img, window_x, window_y, window_size_x,
    window_size_y, npeaks):
   totalCount = GetWhitePixelCount(img)
   count = 0

   for y in range(window_y, window_y + window_size_y):
      for x in range(window_x, window_x +
          window_size_x):

         if y >= img.shape[0]:
            continue
         if x >= img.shape[1]:
            continue

         if img[y][x] == 255:
            count += 1

   per = 0.0069 + 0.2 * abs((1 - npeaks) / 5)
   if count < per * totalCount:
      return True
   return False

def GetWindowSize(points, npeaks):
   min_x = 99999
   min_y = 99999

   max_x = 0
   max_y = 0

   for i in range(len(points)):
      if points[i][0][0] > max_x:
         max_x = points[i][0][0]
      if points[i][0][1] > max_y:
         max_y = points[i][0][1]
      if points[i][0][0] < min_x:
         min_x = points[i][0][0]
      if points[i][0][1] < min_y:
         min_y = points[i][0][1]

   # window_x, window_y, window_size_x, window_size_y
   return min_x, min_y, int((max_x - min_x) /
       npeaks), max_y - min_y

def SweepTopBottom(img):
   height, width = img.shape[0], img.shape[1]
   for y in range(0,height):
      for x in range(0,width):
         if img[y][x] == 255:
            return x,y
   return -1,-1

def SweepBottomTop(img):
   height, width = img.shape[0], img.shape[1]
   for y in range(height-1,-1,-1):
      for x in range(0,width):
         if img[y][x] == 255:
            return x,y
   return -1,-1

def SweepLeftRight(img):
   height, width = img.shape[0], img.shape[1]
   for x in range(0,width):
      for y in range(0,height):
         if img[y][x] == 255:
            return x,y
   return -1,-1

def SweepRightLeft(img):
   height, width = img.shape[0], img.shape[1]
   for x in range(width-1,-1,-1):
      for y in range(0,height):
         if img[y][x] == 255:
            return x,y
   return -1,-1

def GetRectEdges(img):
   x1,_ = SweepLeftRight(img)
   x2,_ = SweepRightLeft(img)
   _,y3 = SweepTopBottom(img)
   _,y4 = SweepBottomTop(img)
   topleft_x = x1
   topleft_y = y3
   bottomright_x = x2
   bottomright_y = y4
   return topleft_x, topleft_y, bottomright_x,
       bottomright_y

def GetRectSection(img,tl,br):
   #rectangle(img, (x1,y1), (x2,y2), 128, 1)
   clone = img.copy()
   roi = clone[tl[1]:br[1],tl[0]:br[0]]
   return roi

def DrawRectangle(img,tl,br):
   #top row
   for i in range(tl[0], br[0]):
      img[tl[1]][i] = [0,0,255]
   #bot row
   for i in range(tl[0], br[0]):
      img[br[1]][i] = [0,0,255]
   #left column
   for i in range(tl[1], br[1]):
      img[i][tl[0]] = [0,0,255]
   #right column
   for i in range(tl[1], br[1]):
      img[i][br[0]] = [0,0,255]
   img[br[1]][br[0]] = [0,0,255]
   return img
```