HAND GESTURE RECOGNITION

MAHAVEER VERMA

https://github.com/mahaveerverma/hand-gesture-recognition-opencv

This project implements a hand recognition and hand gesture recognition system using OpenCV on Python 2.7. A histogram based approach is used to separate out a hand from the background image. Background cancellation techniques are used to obtain optimum results. The detected hand is then processed and modelled by finding contours and convex hull to recognize finger and palm positions and dimensions. Finally, a gesture object is created from the recognized pattern which is compared to a defined gesture dictionary.

Contact:

Mahaveer Verma

mahaveer.verma1@gmail.com *https://github.com/mahaveerverma* *http://www.MahaveerVerma.com*

PLATFORM

Python 2.7.6

This project was built on Linux (Elementary OS Freya) but is compatible with Windows as well (verified on Windows 10)

LIBRARIES

- OpenCV 2.4.8 (http://opencv.org/downloads.html)
- Numpy (http://www.numpy.org)

OpenCV and Numpy are available by default on most Linux distros. One must separately download and install these on Windows systems.

HARDWARE

This project requires a camera installed in user's computer. Project tested on Lenovo laptop with webcam.

NOTE FOR WINDOWS USERS

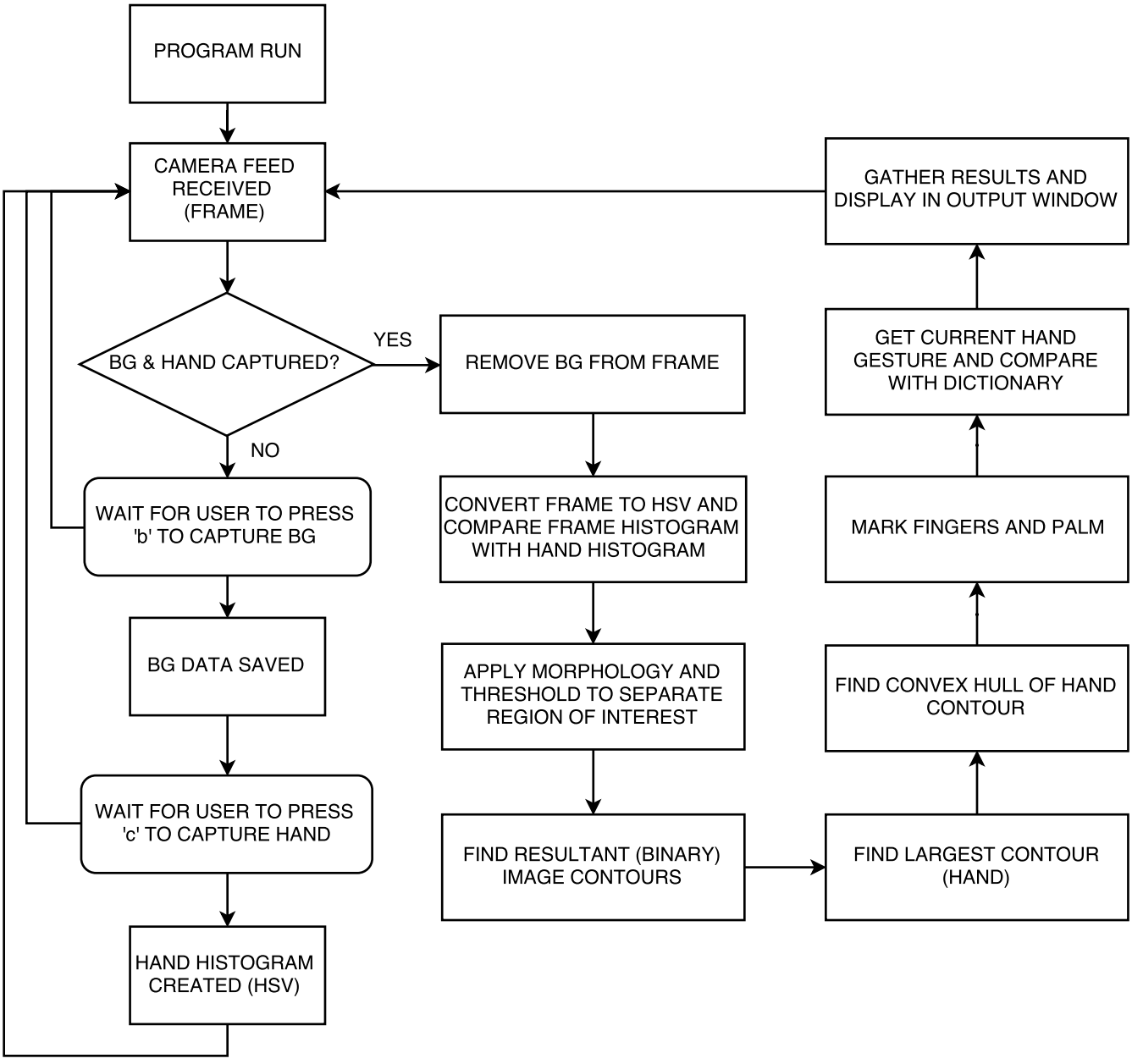
To get this code work on Windows, remove the following line from all .py files:

```
#!/usr/bin/python
```

This line is required for Linux environment to locate Python interpreter and might return some error if left intact while running the program in Windows.

HOW TO START THE PROGRAM

Run the 'HandRecognition.py' python file to begin. Follow the steps shown in output window for demo. Read the implementation below for detailed description on its working.


PROGRAM RUN
CAMERA FEED RECEIVED (FRAME)
BG & HAND CAPTURED?
YES
NO
WAIT FOR USER TO PRESS 'b' TO CAPTURE BG
BG DATA SAVED
WAIT FOR USER TO PRESS 'c' TO CAPTURE HAND
HAND HISTOGRAM CREATED (HSV)
REMOVE BG FROM FRAME
CONVERT FRAME TO HSV AND COMPARE FRAME HISTOGRAM WITH HAND HISTOGRAM
APPLY MORPHOLOGY AND THRESHOLD TO SEPARATE REGION OF INTEREST
FIND RESULTANT (BINARY) IMAGE CONTOURS
FIND LARGEST CONTOUR (HAND)
FIND CONVEX HULL OF HAND CONTOUR
MARK FINGERS AND PALM
GET CURRENT HAND GESTURE AND COMPARE WITH DICTIONARY
GATHER RESULTS AND DISPLAY IN OUTPUT WINDOW

HAND RECOGNITION

Flowchart explains the complete process of hand gesture extraction in the program.

We will first discuss hand recognition in this section, followed by gesture recognition in the next section.

Running 'HandRecognition.py' presents a window that acts as our output frame, currently displaying the camera feed input.

Camera input is captured using:

```
camera = cv2.VideoCapture(0)
ret, frame = camera.read()
```

'frame' is generated as a Numpy array. Numpy library has a lot of functions for easy access and manipulation of array elements. All that, plus Numpy functions are really fast.

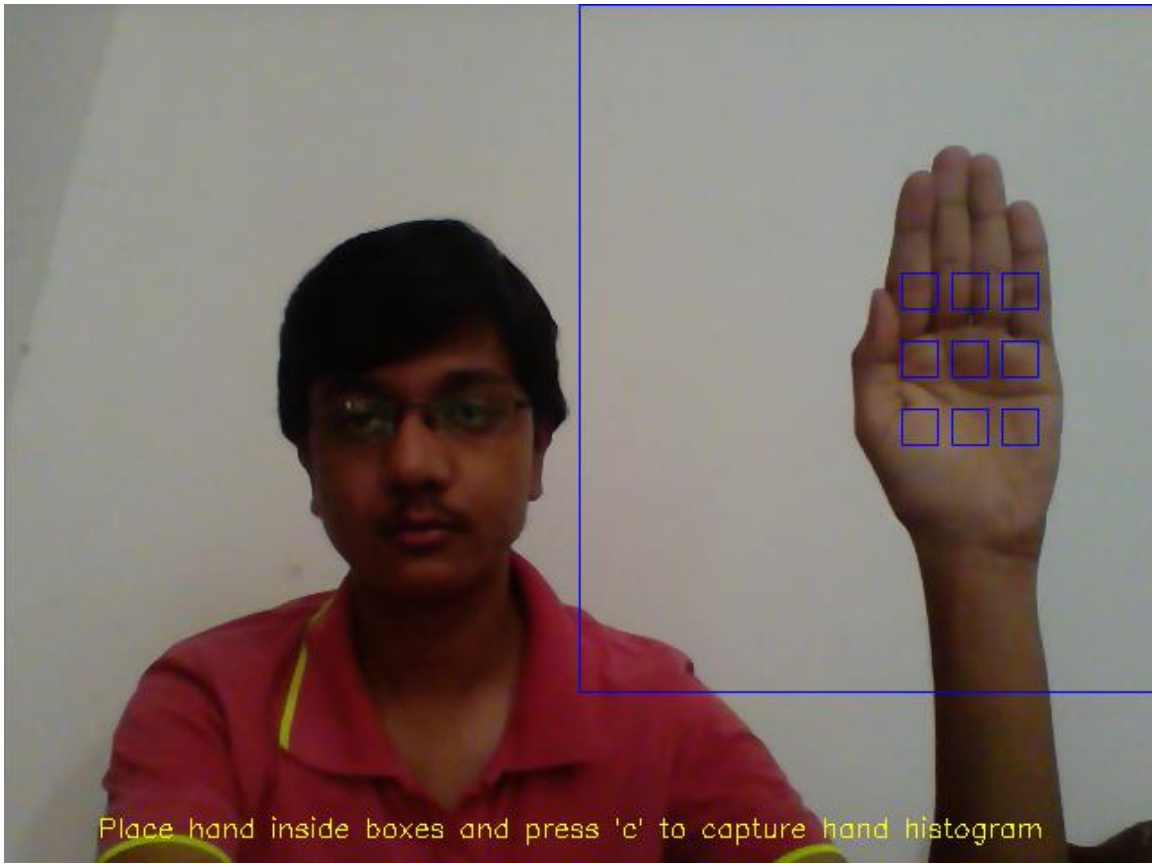
First the program captures the background. This will help reduce detection of false stationary objects which are around the same histogram range as our hand. To capture background, the user will remove his hand or any body part from the rectangle on the right side of the frame and press 'b' on keyboard.

```
bg_model = cv2.BackgroundSubtractorMOG2(0,10)
```

The above line captures the background and creates a background model from our frame.

Now that we have a plain background image detected, we go for capturing a histogram of our hand that we need the program to recognize in each frame. For this the user moves his hand to cover the 9 small boxes in such a way that all or most of his hand shades are covered in them with no air gaps or dark shadows on it. This step is very necessary for a proper recognition.

The user presses 'c' to capture his hand histogram.




```
hand_histogram=hand_capture(frame_original,box_pos_x,box_pos_y)
```

This line is run when the user presses ‘c’. ‘hand_capture()’ is a user defined function which does the following tasks:

1. Convert image frame (with hand on the 9 small boxes) to HSV color space. Why HSV? There are numerous good reasons which one can understand by searching a bit on the internet. Basically we choose HSV because it is a better color space than RGB when it comes to detecting colors because Hue and Saturation channels are not affected by lighting and other image parameters and extracting color information is thus easier.
2. Extract the pixel values from those 9 boxes in HSV color space (our region of interest).
3. Create and return a histogram using those pixel values.

To create a histogram, ‘cv2.calcHist’ function from cv2 library is used.

Once the background and hand histogram are captured, the program proceeds to an infinite while loop which can be interrupted by user by pressing ‘q’. So

that means ‘q’ exits the program. User can also press ‘r’ to reset the program to start everything all over again, in case user wants to recapture background and hand histogram.

In the infinite loop, the program proceeds as follows.

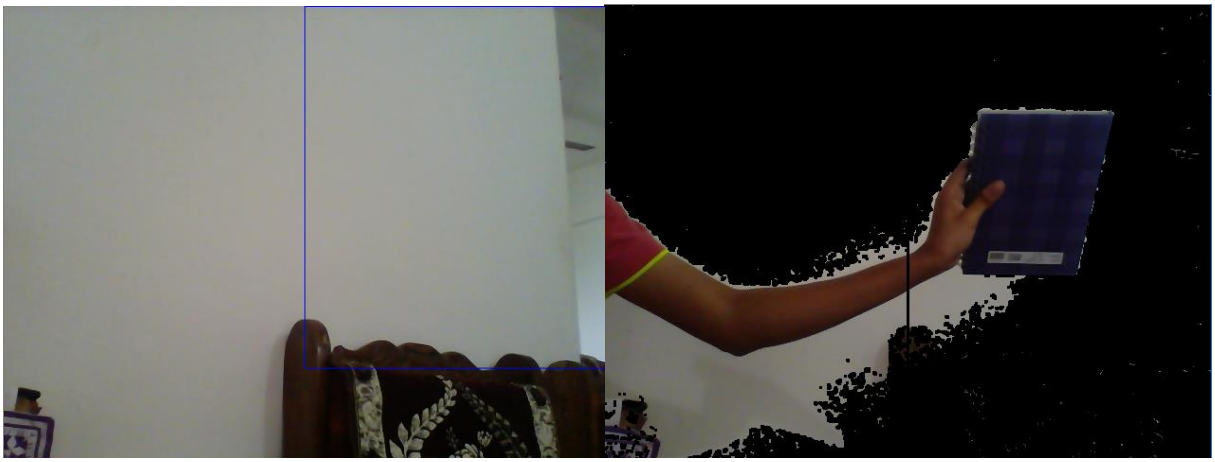
Camera feed input is taken and stored in a numpy array named ‘frame’. Background is removed from the image by taking the background model we just created and running the following line:

```
fg_mask=bg_model.apply(frame)
```

This line will actually just create a mask using the background model. This mask is then applied on the input frame, to result into a frame containing only the foreground. Following line does this task:

```
frame=cv2.bitwise_and(frame,frame,mask=fg_mask)
```

Now, ‘frame’ contains a frame without background.



Screenshot shows image before capturing background and image after bring objects to the frame (after capturing background)

Then we proceed to searching for hand using the histogram we generated.

A user defined ‘hand_threshold’ function does the following tasks:

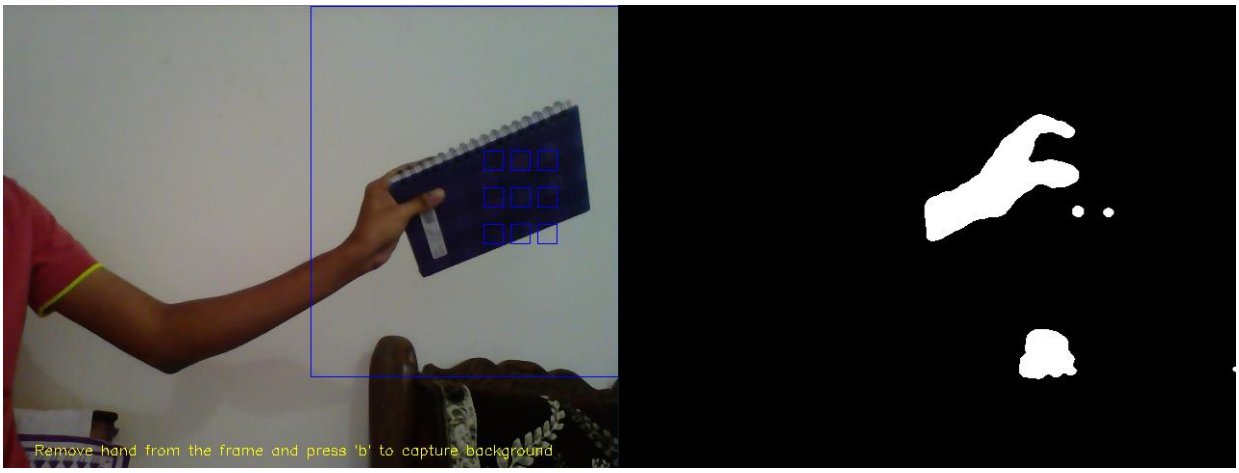
1. Blurs the input frame to reduce noise
2. Convert frame to HSV

3. Separate out the part contained in the rectangle capture area and discard the rest
4. Calculate back projection of image using ‘cv2.calcBackProject’ function from cv2 library. The histogram generated previously is passed as a parameter to this function call.

```
back_projection = cv2.calcBackProject([hsv], [0,1],hand_hist, [00,180,0,256], 1)
```

5. Apply morphology and smoothening techniques (Gaussian and Median blur) to the back projection generated.
6. Apply threshold to generate a binary image from the back projection. This threshold is used as a mask to separate out the hand from the rest of the frame.

The back projection generated during the process is what separates out the hand from the rest of the image. Morphology and blurring is used to create a proper shape of hand which can be easily used in the next steps.



Screenshot shows foreground image containing hand and a book. The result of histogram comparison leaves only the hand as a large blob.

The next step is to find out contours of the image generated in the previous step. We know that no comparison method is 100% efficient and so no matter how perfect the histogram is or how efficient the back projection algorithm be, there will always be false detections and noise. So we don’t just find the contour of the image but also apply some methods to remove and false detection.

For this, we first find out all contours of the image and then validate the largest contour to verify if it matches the profile of a hand or not.

```
contours,hierarchy=cv2.findContours(contour_frame,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

This line finds the contours and stores them in 'contours' array. 'cv2.findContours' is a function from cv2 library.

Next, 'hand_contour_find', a user defined function is run which finds out the largest contour (by area) in the image, assuming the hand is the large enough and there is no falsely detected object larger than the hand, which is a reasonable assumption for this project. The function returns 'False' if a completely blank frame is provided to it. That happens if there is not a single pixel of useful data detected in back projection. This case is created to prevent crashing of program by stopping execution of further functions on the same frame which work on the detected contour. No contour detected and no such measure would end up crashing the program. The program proceeds working on the current frame only if a 'True' is returned by this function.

This function also returns the largest contour thus found (if any i.e. if 'True')

Now that we have the largest contour, the program then works only on this largest contour found. We first find the convex hull of the contour.

```
hand_convex_hull=cv2.convexHull(hand_contour)
```

We do not go for the more traditional method of finding convex hull defects as used by most other similar projects, because the method used in this project is more effective and useful for what we aim.

Once we have the convex hull of the largest contour, we go for detecting the center of the hand.

```
frame,hand_center,hand_radius,hand_size_score=mark_hand_center(frame_original,hand_contour)
```

'mark_hand_center' is a user defined function that does a lot of crucial things:

1. Finds the range of coordinates that span the largest contour (extreme end points that form a rectangle that bounds the contour).

```
x,y,w,h = cv2.boundingRect(cont)
```

2. In the range determined thus, run a loop that takes every point in the rectangular range and measures the distance from that point to the nearest point on the contour. 'pointPolygonTest' function does this work. This function is really slow and so we try to keep the range over which this loop runs as small and optimized as possible to reduce computation time. Note that the distance returned b pointPolygonTest is negative if the point lies outside the contour which is really good for us because that eliminates any need to check if the point lies outside or inside the contour.

```
dist= cv2.pointPolygonTest(cont,(ind_x,ind_y),True)
```

3. Find the point with the largest such distance. The point is the center of the largest circle that can be inscribed inside the contour and the distance is the radius of the circle. And because the value would be negative for a point outside the contour, the point with the largest distance is definitely inside the contour. Thus, the point is the center of our hand and the radius is the dimension of our palm.
4. Check if the hand radius is large enough because very small radius might suggest falsely detected objects in case the frame is actually kept empty. Again, a healthy condition for this project. Return palm coordinates and radius

Once we have the palm center coordinates and radius, we go for finding fingers in our hands.

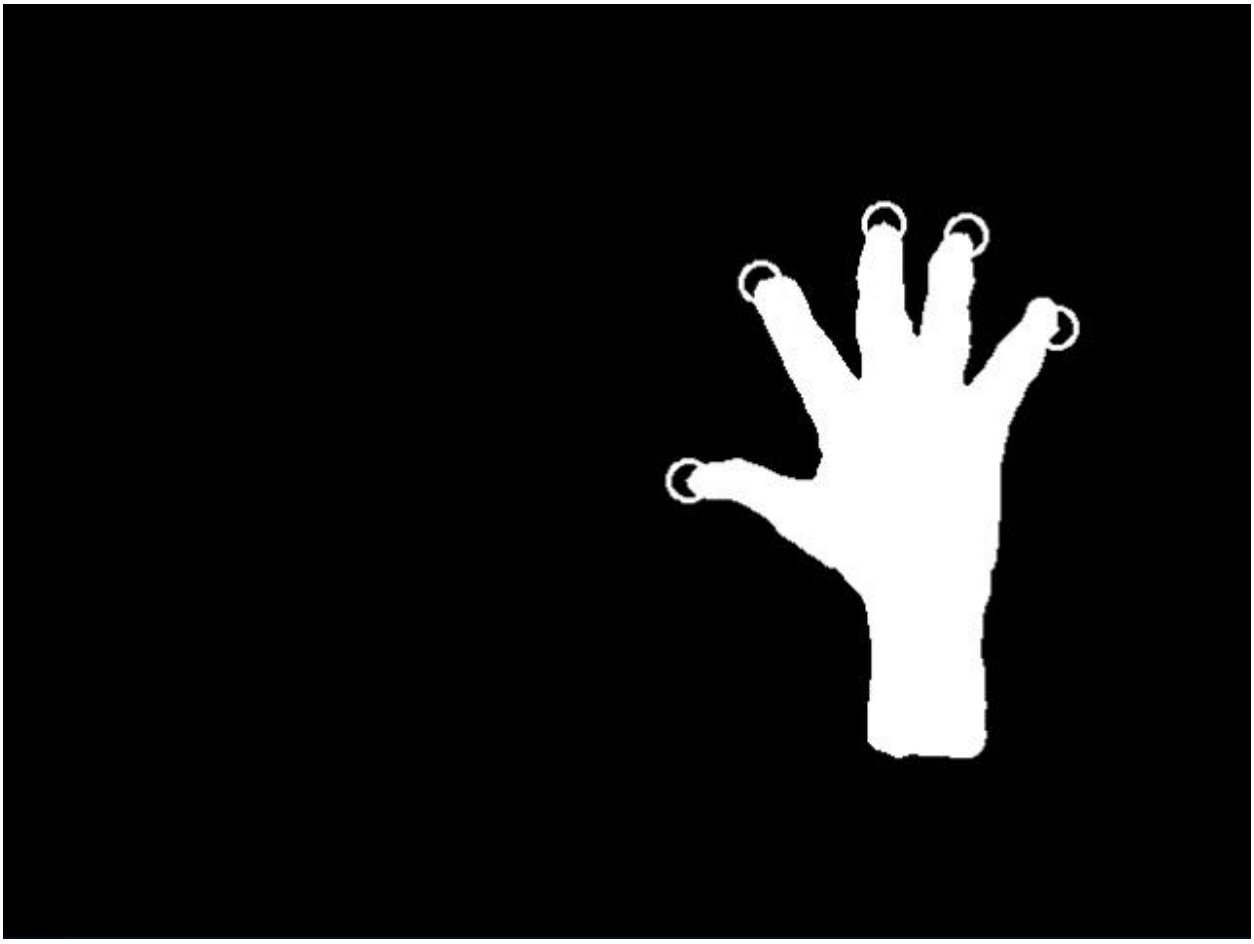
frame,finger,palm=mark_fingers(frame,hand_convex_hull,hand_center,hand_radius)

'mark_fingers' is a very important user defined function which does the following tasks:

1. Go counter clockwise direction and eliminate all convex hull points that are very near to each other. This helps in achieving '1 point per finger'. Otherwise the original convex hull would show about 10 points near every finger. The traditional method of finding convex hull defects does this automatically in the process but we still won't prefer that way for this project.

2. Eliminate all convex hull points too far or too near the hand center obtained in the ‘mark_hand_center’ function. This will leave us with just a single point per every finger that is stretched out.
3. Optimizes the results for best results and so that we don’t get more than 5 fingers.

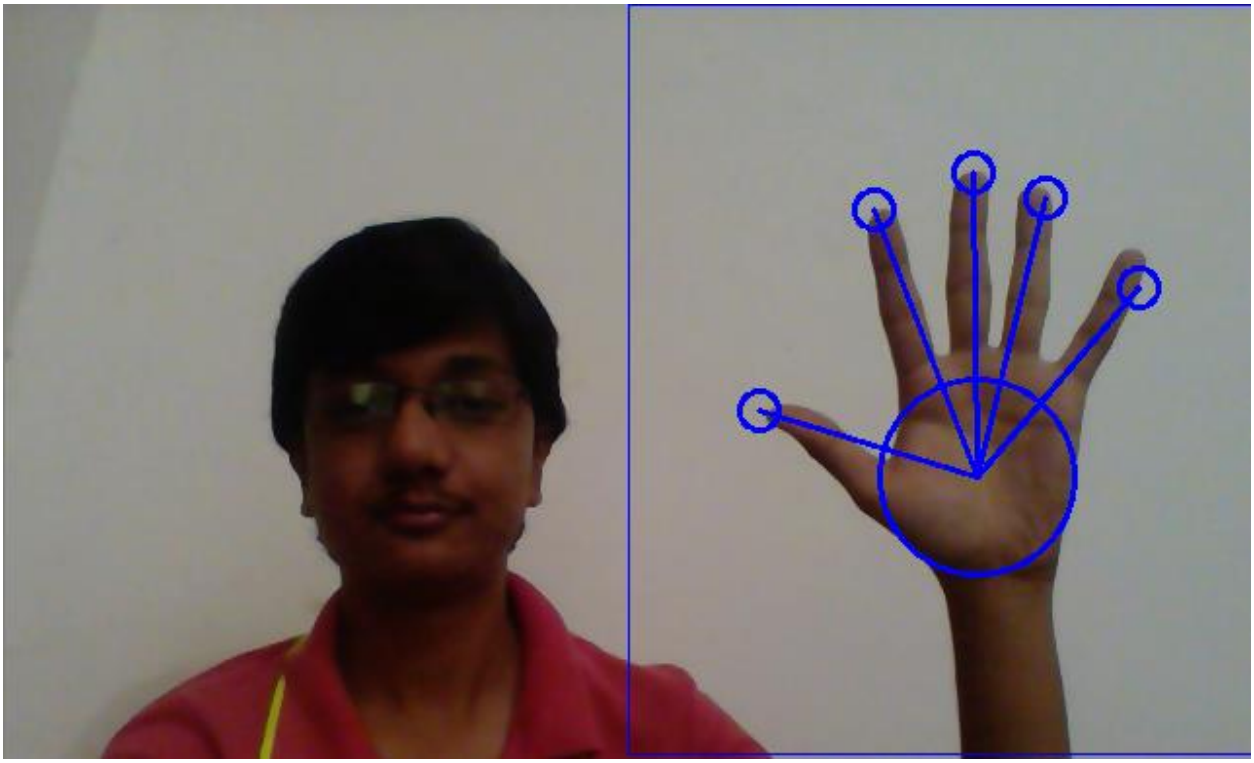
What we have after this, is a palm center, radius and exactly only those many points in a convex hull that belong to our fingers. The plus point is, we have proper coordinates of each and every finger after this function.



This makes it easy for us to model our hand based on the following parameters:

- Palm center position
- Palm radius
- Finger positions in coordinate system

From these parameters, we can then figure out the angles between different fingers, the count of fingers shown and ultimately the gesture defined with those parameters.



Once we have the hand modelled we go for gesture recognition.

HAND GESTURE RECOGNITION

From the hand recognition section, we have the following end results with us:

- Palm center
- Palm radius
- Finger positions (and thus count as well)

This project creates a gesture recognition system using a 'Gesture' class. The class variables and functions are defined in 'GestureAPI.py'

Every instance of the 'Gesture' class has following attributes:

- name – A name given to the gesture
- hand_center – Center of palm
- hand_radius – Radius of palm
- finger_pos – Array of position of fingers
- finger_count – Total number of fingers
- angle – Angle between each finger and X axis

Out of these, when creating a new instance, we supply name, hand_center, hand_radius and finger_pos by use of class functions. The remaining are calculated by calling class functions.

'GestureAPI.py' file contains a dictionary of gestures that are used when searching for a gesture found in current input frame. One example on how to define a gesture in this dictionary is:

```
# BEGIN ----------------------------------#
  V=gesture("V")
  V.set_palm((475,225),45)
  V.set_finger_pos([(490,90),(415,105)])
  V.calc_angles()
  dict[V.getname()]=V
# END ------------------------------------#
```

A gesture with name "V" is defined. The name is later used as identifier or unique id to the gesture. Palm center is set as (475,225) with a radius of 45 pixels. This gesture, as the name suggests is a hand showing 2 fingers making a 'V'. So we have the coordinates of those 2 fingers as (490,90) and (415,105). The finger count will be generated automatically as we pass the finger coordinates. V.calc_angles() will calculate angles of each finger for instance V.

The last line dict[V.getname()]=V will append this gesture to a dictionary 'dict'.

This dictionary is generated when 'DefineGestures()' function is called. So we place the following line somewhere at the beginning of our 'HandRecognition.py' code:

```
GestureDictionary=DefineGestures()
```

Now we have a dictionary we can use to compare our detected gesture with.

Now, we need a 'Gesture' instance to store information about the gesture made by the hand in each input frame.

```
frame_gesture=Gesture("frame_gesture")
```

Every time the hand recognition code is run and every time we have a new coordinate for our hand and fingers, we update the attributes of this instance.

```
frame_gesture.set_palm(palm[0],palm[1])
frame_gesture.set_finger_pos(finger)
frame_gesture.calc_angles()
```

Then we run the 'DecideGesture' function defined in GestureAPI.py

```
gesture_found=DecideGesture(frame_gesture,GestureDictionary)
```

This will return the name of gesture detected and will return "NONE" if no suitable match is found. The 'DecideGesture' function works in the following algorithm:

1. Run a loop to compare current frame gesture passed as 1st argument in the function with every element in the dictionary passed as 2nd argument in the function.

2. While comparing, first check if the number of fingers in the current frame gesture (src1) and gesture being compared (src2) are same.

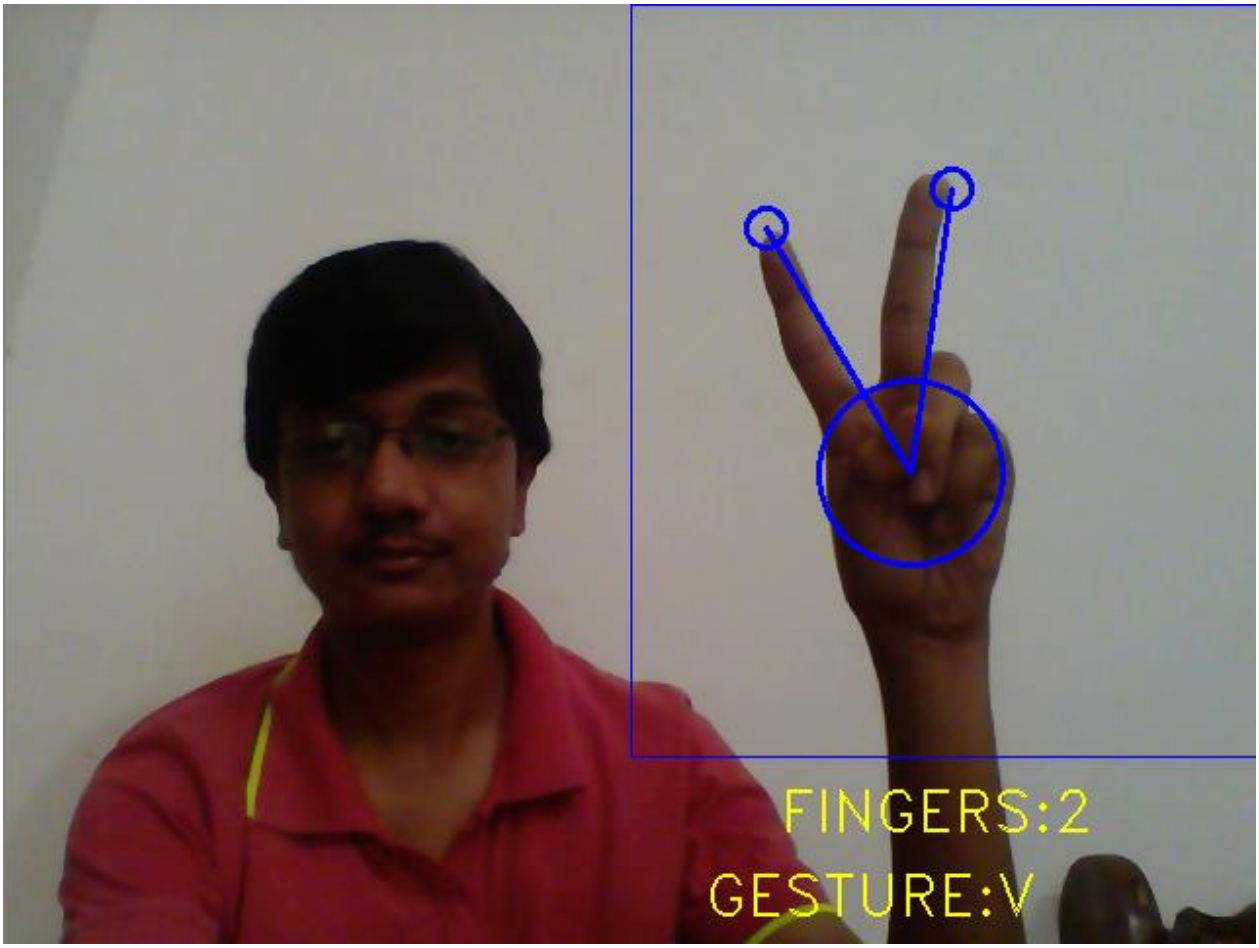
3. If the number of fingers are same and more than 1, generate an array of differences in angles made by each finger of src1 and that of src2 i.e. angle by finger 1 (src1) – angle by finger 1 (src2), angle by finger 1 (src1) – angle by finger 1 (src2) and so on. If the values turn out to be nearly constant, we can say the hand has a constant tilt and the gestures might be same. Example a "V" with hand kept straight and a "V" with a tilted hand.

4. If comparison passes the angle stage, we go for doing the same for fingers. But here we take the ratio of finger lengths instead of difference. So that we know if the hand has shrunk or grown by a constant factor. If the array thus obtained has absurd variations, it means the wrong fingers are making the correct angles and this will again reject the gesture from list.

5. If both the stages are passed, return the gesture thus obtained.

6. If the number of fingers is 1, compare angle and length of finger to radius of hand ratio. In this case, a tilted hand with single finger making the same gesture also gets rejected. By comparing ratios of length of finger to radius of hand, we get to know whether the correct finger is being used in the gesture. Return if match found.

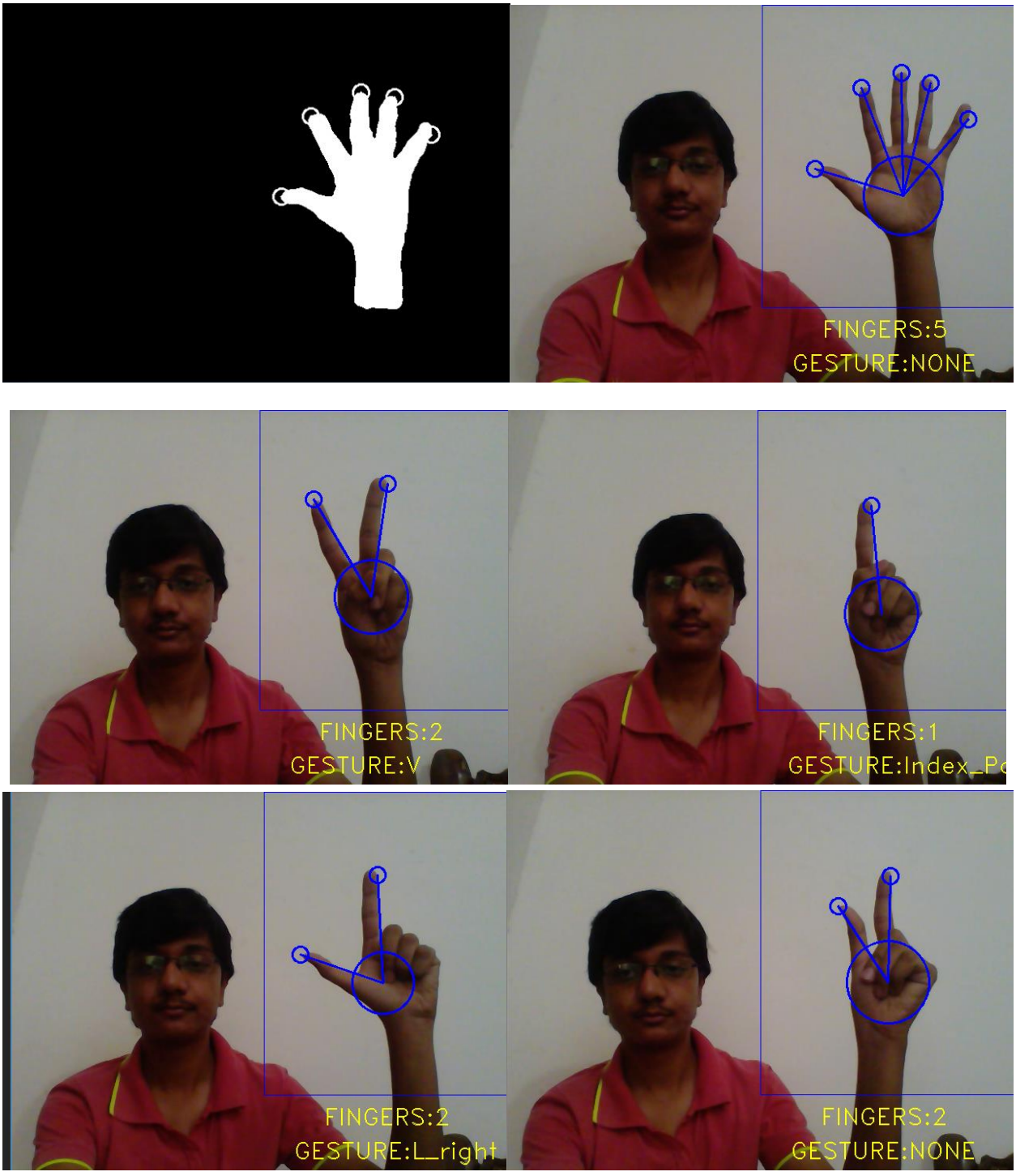
The name of the gesture thus detected is displayed on the output screen.




```
gesture_text="GESTURE:"+str(gesture_found)
cv2.putText(frame_in,gesture_text,(int(0.56*frame_in.shape[1]),int(0.97*frame_in.shape[0])),cv2.FONT_HERSHEY_DUPLEX,1,(0,255,255),1,8)
```

The whole process of hand recognition and gesture recognition repeats with every input frame.

RESULTS


FINGERS:5
GESTURE:NONE
FINGERS:2
GESTURE:V
FINGERS:1
FINGERS:2
GESTURE:L_right
FINGERS:2
GESTURE:NONE

Illustrated are some result screenshots of hand recognition and gesture being detected as required.

The last image is an example of how measuring finger length ratio eliminates a gesture made with incorrect combination of fingers. Here the difference between ratio of lengths of index to middle finger compared to that of thumb and index finger helped eliminate this false detection.

The end results show an output window with the input image overlaid by the hand model generated and displays the finger count and gesture detected.

APPENDIX

- *Variables and Parameters:*

Some variables are considered for optimization purposes. Also, there are variables that define certain parameters such as capture box area, threshold values, threshold for finger lengths, threshold for hand radius, etc.

All the parameters are named in such a way that it would be easy to understand what they do in a function. Also, all of them are defined at the very beginning of 'HandRecognition.py' for easy access.

In most cases, for tricky variables, I have placed comments near their declaration or use. Also, most variables are used in form of ratio of something. Example, radius_thresh=0.04 is the threshold for minimum tolerable radius of hand detected. The value 0.04 implies that (0.04)x(Frame Width) will be used as minimum threshold, to keep a relative measure of threshold compared to input frame rather than keeping it a constant value.

In case you need to know what a certain parameter does, let me know and I will be happy to explain.

- *References & Tutorials:*

1. OpenCV documentation: http://docs.opencv.org/2.4.8/
2. This tutorial helped me understand basics of histogram and served as a guide to finger recognition: http://www.benmeline.com/finger-tracking-with-opencv-and-python/
3. HTML book series, full of basic tutorials on Python. This particular article is very useful to understand how to work with classes in Python: http://learnpythonthehardway.org/book/ex40.html
4. Filtering and smoothening images: http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html
5. Contours in OpenCV: http://docs.opencv.org/master/d4/d73/tutorial_py_contours_begin.html

And *Google* of course. Internet has a solution to every problem. Google is the key to find the right one.

Find Project here: https://github.com/mahaveerverma/hand-gesture-recognition-opencv