Hand Gesture Tracking and Recognition Report

Authors: Kailun Wan, Yanda Li, Yizhou Wang

Introduction

In this assignment, we make a hand gesture tracking and recognition system in Python using OpenCV to detect our hand gestures and control the mouse and keyboard with custom gestures. We apply the feature-engineering and heuristic-based approach to design and implement our system. Our system has the procedures including extracting hand from the video feed, dealing with contours and hulls to get ring gesture and number of fingers, implementing the time window to design recognition system for custom gestures, and controlling mouse and keyboard action with those gestures. We also extend our system to support two-hand gestures. It is divided into six parts, including the setup (part 0) and the bonus part for two hands (part 5). We will explain more of our implementation in each part.

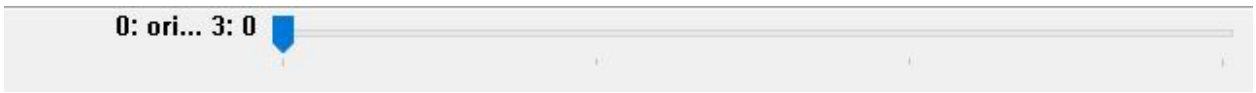
Part 0: Camera Setup and Environment Setting

To extract useful information from the videofeed, we need to set up our camera and the environment. To get better results, we need to make sure that our hands are the only thing the camera can see, with a uniform background contrast. Such a background is to avoid other features' interference in the videofeed instead of our hands.

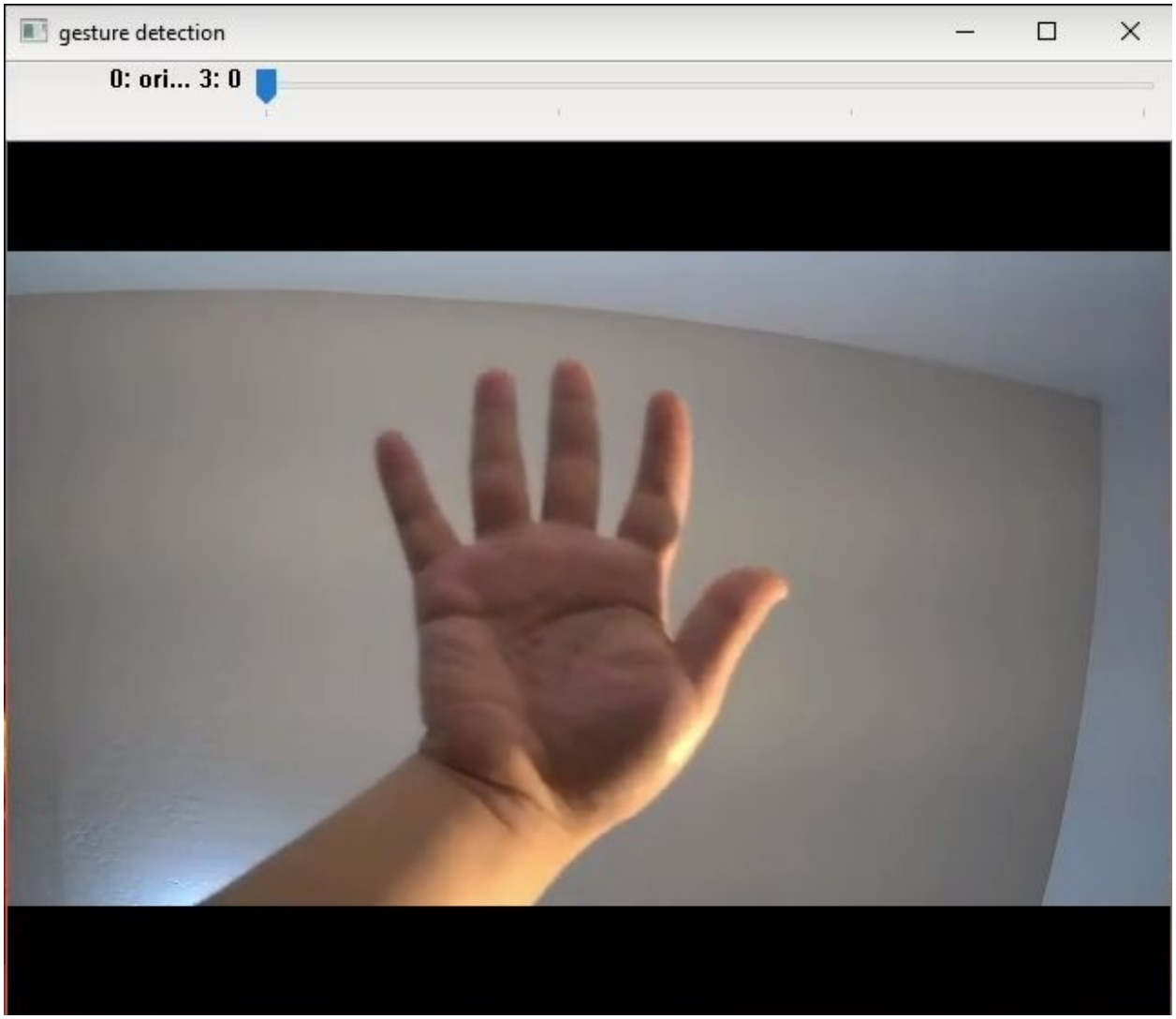
To fulfill this requirement, we point our webcam towards the ceiling and apply multiple lighting effects from different angles to make a suitable ambient environment.

Additionally, to allow users to adjust some parameters to fit their use environment, we add multiple trackbars to our system for different usages. Users can change some values in the system to make better effects for their specific cases. Different parts have different trackbars, while some of them are common in all parts. We will explain more in each corresponding part.

In this part, we set a main trackbar 'image type' to separately show each following part of our work. Users can choose from part 0 to part 3 to show the results of our system. For part 4 and part 5, the users can stay on either part 2 or 3 to apply corresponding custom gestures to control the mouse and keyboard actions. We will give more details in the latter parts.



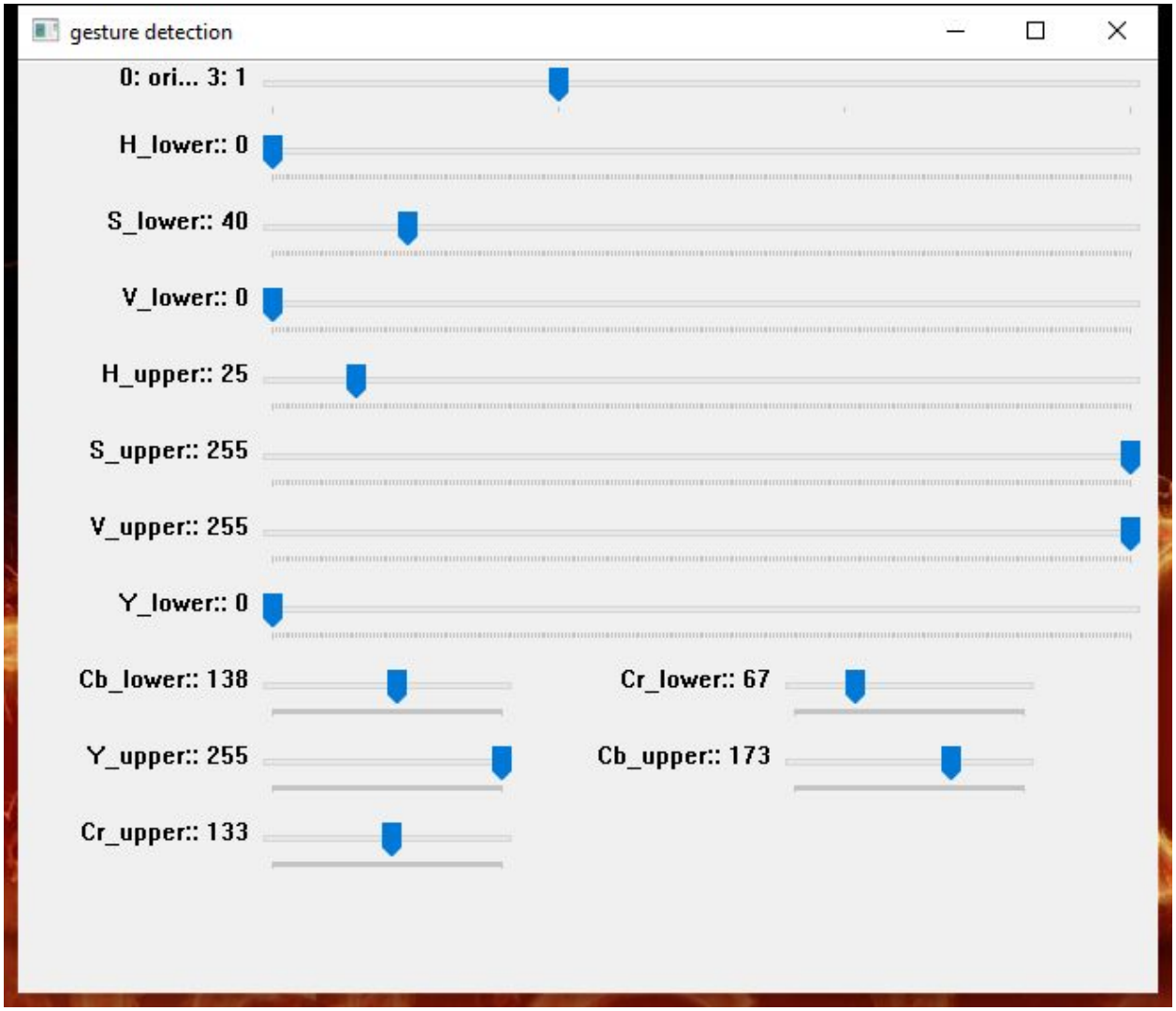
By default, the 'image type' is set to part 0, which shows the videofeed from the webcam. After we set up our environment, we can continue to extract our hands from the feed.



Part 1: Extracting Hand from the feed

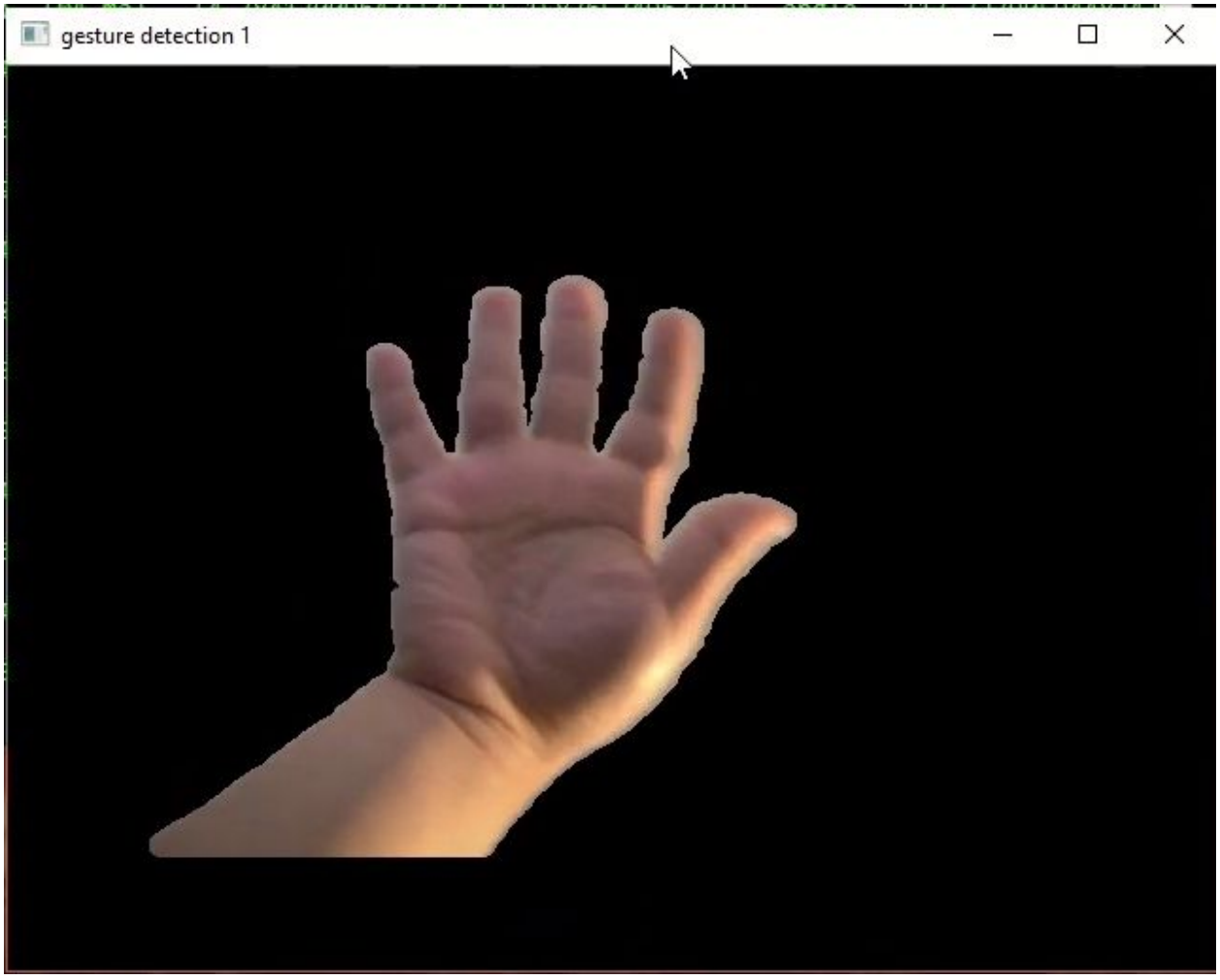
To extract the hand from the videofeed, we use a simple skin color matching technique. Since RGB color space has non-uniform characteristics, it does not work well in color-based detection. The HSV based detection, however, discriminates color and intensity information under uneven illumination conditions and is better to use with uniform background. Moreover, YCbCr color space is effective for the separation of image pixels in terms of color in color images. In this part, we apply both HSV and YCbCr skin mask to segment the hand skin from the videofeed. We tried several different values for our case and chose appropriate lower and upper bound values for our HSV and YCbCr detection to get the skin mask.

However, this technique has some potential issues. Different users may have different environment settings, lighting conditions, and skin colors. The system with simple implementation with the skin color matching technique does not work well with all users. We understand that those upper and lower bound values for skin detection need to change for any specific use case under different conditions. Thus, to enable our system to perform reliably for all users, we add several trackbars for this part.



In this part, we allow users to adjust any lower or upper bound values for both HSV and YCbCr color spaces and reach their best results. In total, we build 12 trackbars, one for each value of either lower or upper bound for HSV or YCbCr color space, ranges from 0 to 255. The users can change these values by themselves when using our system and adapt to their skin colors and environment. Thus, we extend our system to be reliable for all users under different conditions.

After we get the extracted skin mask, we find that it is quite noisy for our hand. We apply morphological transforms - erosion and dilation to remove the noisy pixels around our single hand to reduce such noises. We also use a standard gaussian blur to remove those noisy pixels in the skin mask region. As a result, we get better image frames with our hand's extraction for use in the latter parts.



Part 2: Hand Image with Connected Component Analysis

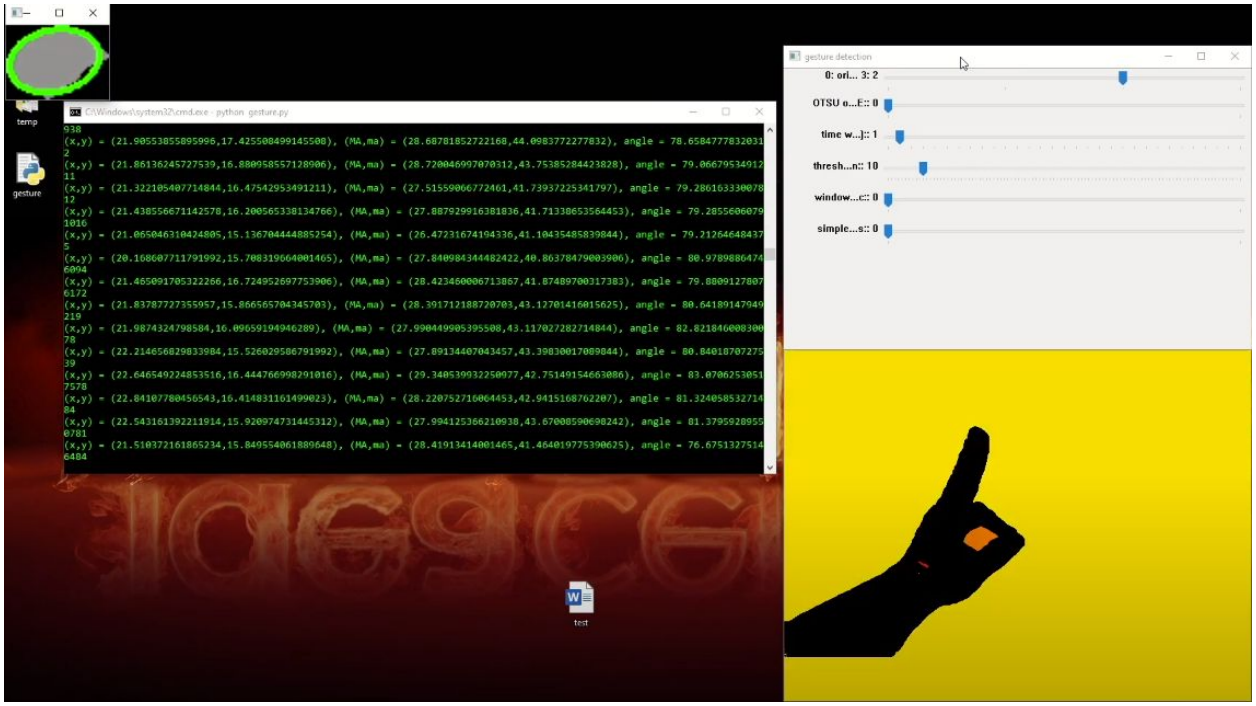
With the working system to extract hand from the feed, we try to detect a specific hand gesture, a ring formed by our fingers, to achieve the first gesture detection.

To achieve this, we use the connected component analysis in OpenCV to get the ring from our hand's "ok" gesture. However, such an algorithm needs a binary image for input. Thus, we threshold and binarize the images of part 1 to fulfill the requirement. To have a better result, we apply two types of binarization methods, "Otsu" and "Triangle," and use the one with better outcomes for our case. Similar to skin extraction in the last part, we allow the users to choose either "Otsu" or "Triangle" binarization in their cases with a trackbar for better results.

Additionally, we also invert the binary image to acquire the black hand with a white background to use the

connected component analysis. We can use this technique to identify and label the connected components in the image under such conditions.

From this algorithm, we can get multiple components for our image. The largest one is the background, then the hand, and the next is the ring of our hand gesture. We need the third largest component, the ring, to become our region of interest (ROI). We extract ROI from the binary image, then apply contour analysis on the ring to fit an ellipse to the largest contour, draw and get the related information, such as the center's coordinates, major and minor axes, and the orientation of the ellipse for our hand gesture data. We also collect the area of the ellipse from the ROI information for future use. To avoid any issues in contour analysis, we adjust our codes to avoid and ignore the contour analysis if there is no hand in the videofeed.



The data we collect from the ellipse can be useful in our custom gesture design in part 4 and part 5. For example, we use the ellipse center's coordinate to move our mouse and use the ellipse area to achieve some functions in the photo app. We will explain more details in the latter parts.

Meanwhile, we add multiple trackbars to adjust some values, such as time window, the threshold for comparison, machine type, and gesture type for our custom gestures. We add these trackbars for both parts 2 and 3, and we will use them in parts 4 and 5 when we explain our custom gestures. More details will come below.

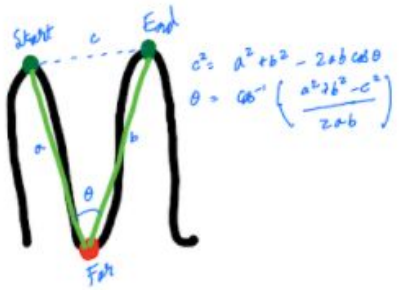
Part 3: Tracking 2D Finger Positions

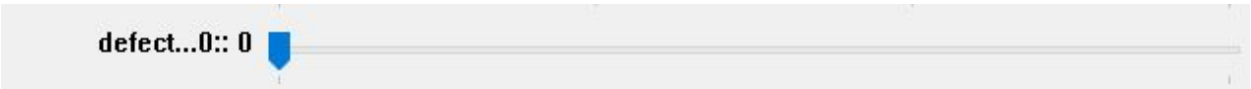
Another type of hand gestures we use in our system is finger detection. We use the contour, convex hull, and convexity defect analysis to get the correct number of fingers on a hand.

Before detecting the number of fingers, we need to process the hand image with contour and hull analysis. We use the thresholded binary image from part 2, without inverted (black background and white

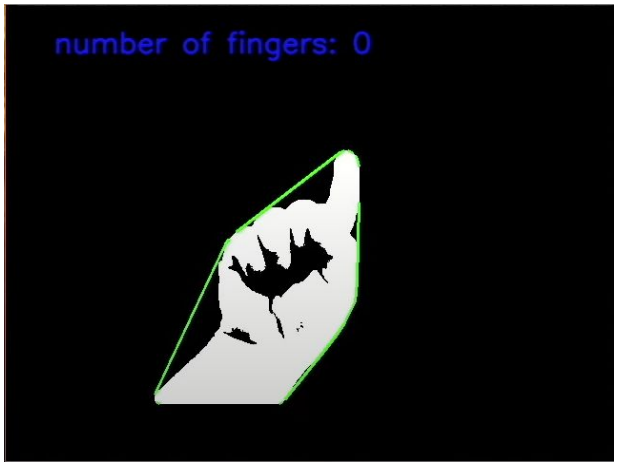
hand), to apply contour analysis similar to that in part 2. The largest contour, however, is that of the hand. Next, we compute a convex hull from this contour and get related convexity defects. Those defects are the furthest points in the hull, where the convexity does not hold. After contour and hull analysis, we convert the thresholded binary image to a BGR color image and draw the convex hull of our hand with green lines, with the defects as red dots.

From the result, we can see that there are many defects in our single hand. Some of the defects are reasonable (defects at the far end between two fingers), while other defects are noises. We find a relationship between the number of reasonable defects and the number of fingers, but we need to filter out those defect noises first. So how can we achieve this? We apply a specific heuristic algorithm to calculate the angle at each defect point and constrain it to some value. The far point forms the angle with the start and end points for each defect. The angle needs to be less than 60 degrees to be a reasonable defect. Our experiment and literature review found that 90 degrees may also be a useful constraint under some circumstances. Thus, we add 60 and 90 degrees for better user experience under different environments as one of the user's choice as a trackbar in this part.

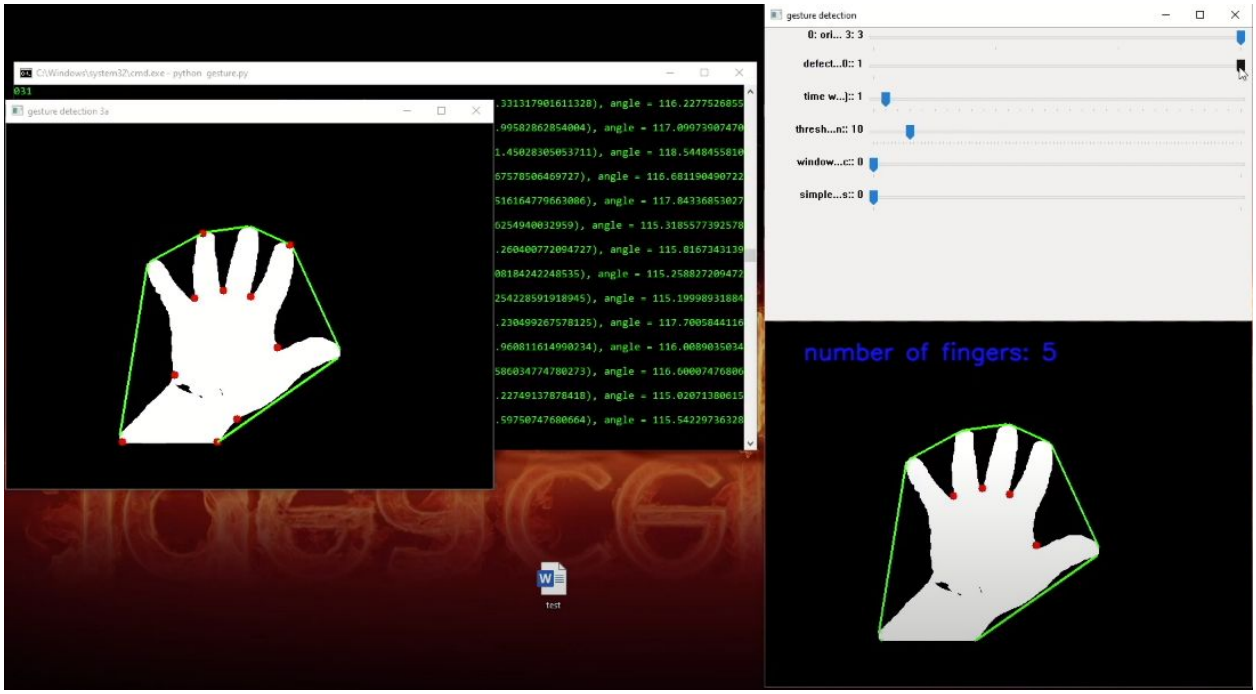




After we filter out noisy defects, we leave only those located at the valley between our fingers. It is easier to find that the number of fingers is one more than the number of reasonable defects (number of finger valleys). However, this algorithm has a significant issue: it cannot distinguish 0 fingers (a fist) from one finger. Neither has any reasonable defects (no finger valleys when our hand has one finger). Thus, if we use them in our system, we will regard them as the same gesture.



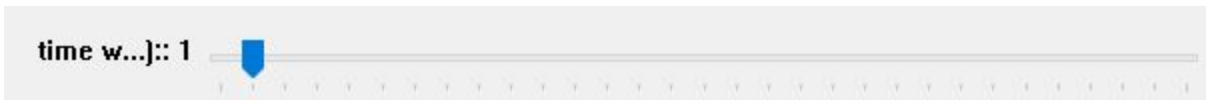
Finally, we get the contour and convex hull analysis before and after the heuristic algorithm to remove noisy defects. We also print the number of fingers detected on the image after applying the heuristic algorithm. We can get the center coordinates of our hand contour and the number of fingers on our hand from this part. We will use these data in future parts to achieve custom gestures.



Part 4: Control mouse and keyboard using custom gestures

From the last two parts, we get much data that we can use to design our custom gestures and control our input devices. To reiterate, we get the area of the ring, the center coordinates, major and minor axes, and the orientation of our ellipse from part 2, and we have the center coordinates of our convex hull as well as the number of fingers in part 3. We use some of them and the pyautogui API to allow our python script to control our mouse and keyboard input.

Before using these data, we find that the information is susceptible to noises, making the input control difficult. Thus, we need to find a way to reduce such noises and use some stable values for our custom gestures. We apply a time window, which collects those variables in multiple frames, and then takes the average to get stable values. The different size of the time window has a different effect – a larger time window reduces many noises, but it increases the system's latency. To allow the users to use the best time window for their environments, we add a trackbar to enable the users to adjust the time window size – from 1 frame to 30 frames – for the window of parts 2 and 3.

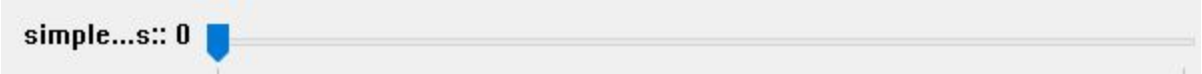


We design three simple custom gestures to control our mouse on the desktop. First, when we set the image type to part 3, we can use the center coordinates of our hand's convex hull to control our mouse's movement, if the number of fingers is 0 (a fist, for example). Otherwise, when the number of fingers is 2,

the mouse makes a double click on an icon, and when the ring from part 2 points downward, the mouse gives a single right click.

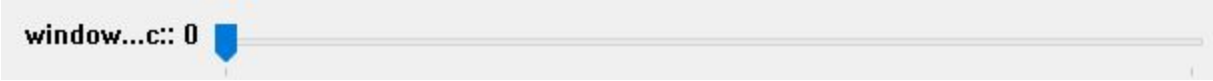
What about complex gestures? The complex gesture needs multi-frame pose processing for simple gestures. In short, it compares the current data of a gesture with the previous data of a gesture and executes some functions if they fulfill certain conditions.

First, we need to change the switch from simple gestures to complex gestures to use complex gestures. There is a trackbar of "simple/complex gestures" with the default value of 0 (simple gestures) in the window of parts 2 and 3. The users need to switch its value to 1 to allow complex gestures to work.



Moreover, there are two more trackbars for users to control in the window of parts 2 and 3. The first one is called "threshold for comparison," which takes a value between 1 to 100. It uses to set the threshold between the previous data and current data for complex gestures. For example, if an action happens when the orientation of our ellipse changes more than 10, we need to set the "threshold for comparison" to be 10.

The second trackbar we add, "windows/mac," is for machine specific. By default, it sets to 0 (windows), and the users can freely change it to 1 (mac) if they run our system on a mac laptop. Since the same keyboard hotkeys in an app may have different buttons for different machines (e.g., cut is "ctrl+c" in windows and "command+c" in mac), we add this trackbar for our system to support various system users.



In our system, we design four complex gestures for our hand in the windows photo app. First, when the ring (ROI) area from part 2 decreases, exceeding the threshold value set in the trackbar, the photo app opens the "save as" function to save the current photo at a chosen location. Next, if the orientation of the ellipse changes, the current picture rotates to the right. When the number of fingers changes from 2 to 4, we add the current photo to the album. Otherwise, if the number of fingers keeps at 5 while the hand moves to the right, the photo app goes back to the collection mode from the current photo view. The demonstration of our simple and complex gestures is shown in our demo video.

Part 5: Hand Gesture Tracking and Recognition for Two Hands

Now, we have a working system for hand gestures for only one hand. But what if we want to use both of our hands to control the mouse and keyboard? The above techniques in part 2 and part 3 look good, but we need to modify them to support two hands. Fortunately, they are not complicated to change to allow bimanual gestures. We will explain the ideas here.

First, we talk about the convex hull and finger tracking from part 3. In that part, we know that the largest contour is for our single hand, and we get the convex hull and related defects for it. For two hands, the idea is quite simple: we take the largest two contours, one for each hand, and then we can get two convex hulls and multiple defects for both hands. After applying the same heuristic algorithm to remove noisy defects, we can get reasonable defects for each hand (each hull) and the number of fingers. Still, this algorithm can not distinguish one finger from 0 fingers, but two fingers to ten fingers work well for both hands finger detection.

Next, we consider the ring from part 2. In that part, we use the connected component analysis to get our ROI from a single hand. We know that the largest component is the background, the second is our hand, and the third is the ROI we want. For two hands, the idea is also simple: the largest one is the background, the second and the third are two hands, and the fourth and fifth are the two rings on both hands. We can get two ROIs through this, and after applying the same remaining technique of part 2, we can get the data for two ellipses.

However, there exists an issue. When our rings change even a little bit in two frames, the order of ROIs may change as well. For example, in frame 1, our left hand's ring is the fourth largest component, and our right hand's ring is the fifth one; while in frame 2, our right hand's ring may become the fourth largest, with the left hand's ring fifth. It may cause problems in custom gestures. For example, suppose we want to enable an action when the left ring's area decreases and the right ring's area increases. In that case, the order-switch will make the recognition of each hand unavailable.

So how can we solve this problem? We use a simple method: split the thresholded image between our two hands into two halves, and fill the removal part of each half with background pixels. Then, we can detect the two images, with the hand on the left half be the left hand, and the hand on the right half be the right hand. This technique has an issue when the two hands switch sides (left hand on the right half and right hand on the left half), but we leave it for future work since the users may seldom have their ring hands switched in real cases.

By applying our simple method, we can get stable ellipses information for both rings, and we also get the distance between the centers of our two rings. We use this information for our custom gestures, as explained below.

Finally, we can design custom gestures for bimanual. The time window and other trackbars mentioned in part 4 work well since our modified algorithm detects the number of hands to activate corresponding gesture controls.

We design three simple custom gestures for bimanual in the word app. First, when the number of fingers is 8, the system strokes “space” of the keyboard. When the two rings from both hands point upward (to express a flower like gesture) at the same horizontal level, we select all content in the word app. Otherwise, when one ring is significantly above the other, we close the word document.

For our four complex bimanual gestures, we still apply them in the windows photo app. When the distance of two rings changes, the current photo zooms in or zooms out. Otherwise, when the areas of both rings decrease, we go into the selection mode from the photo collection. Additionally, when the number of fingers changes from 4 to 8, the current photo's file info is open, and we open the photo view of the selected photo from the photo collection when we move our two “V” shape hands (two fingers on each hand) together. Please consult our demo video for these demonstrations of custom gestures in part 4 and 5.

Conclusion

This assignment gives us how to achieve hand gesture tracking and recognition using Python and OpenCV. By adding multiple trackbars and implementing some algorithms, we use a system to control the input devices with one and two hands. There are still some questions unanswered: How do we detect two hands with "ok" gestures if they switch sides? How can we optimize our system to reduce more noises and make it better to use and control? How many other custom gestures can we add to our system to achieve more input control functions in different apps? We will leave them for future discussion and work.

Appendix

We have a demo video of our system to explain all parts and custom gestures. You can look it from here:

https://youtu.be/leWFPPML5fY

References

Homework Instructions:
https://docs.google.com/document/d/11zTezw9TxPfq2eKa_JUZjzYNFpwolm3Qtf1syXR_eJk