

Step 0: Hardware & Software

Computer: **MacBook Pro**, 8GB, 2.4 GHz Intel Core i5 running OS X 10.10.2
Camera: **iPhone 6** rear-facing camera
Software: **MATLAB** R2014b and its Image Processing Toolbox, **Apple's Preview**

Step 1: Domain Engineering

The domain was engineered around the following steps to obtain a binary image of the skin:

1. Accept a color image as input
2. Convert the color image to a grayscale image
3. Convert the grayscale image to binary one where pixels greater than a specified luminance level are considered to be skin.

With this process in mind, the goal was to capture images where the hand and arm are much lighter relative to the background. To accomplish this, black cloth was used as a background while the bare hand made gestures over it. The black cloth was preferably lint free, but discussion around image process can be found in Step 2.

I with a more naive solution that required the user to wear a dark sleeve to cover the entire arm (up to the wrist) so that it would be considered as background in the binary image. However, further work led me to want to find the location of the wrist, which was must easier when a portion of the bare arm was shown (say, at least three inches from the wrist) or when the wrist was close to the edge of the screen (so that the arm "fell off" the image). The reason having a portion of the arm was useful in wrist detection is described in the Step 2.

Although in most cases it was not an issue to capture a large amount of the user's arm in the image, issues did arise when the thickest part of the forearm was present. Specifically, it prevented the ability to capture the palm's center point, which in term prevents the algorithm from finding the correct wrist location. Detecting the palm's center point and the possible issues are discussed in Step 2.

Ultimately, a partially rolled up black sleeve was worn. This eliminated the issues in detecting the palm's center and left enough arm in the binary image to accurately locate the wrist.

I do not consider the room I used to be particularly unique in it's lighting, however it could create shadows when gestures were made with the palm facing up and fingers folding over (such as the "Rock On" gesture seen in Step 3). Because these shadows could be mistaken as the background in gestures such as these, I would use my camera's flash to eliminate them. Any gesture where the back of the hand was facing the camera did not require a flash.

With the above lighting concerns handle, either the back of the hand or its palm can face the camera. However, it is assumed that the hand is positioned in a way that the hand looks down on either side of the hand and any fingers involved in the gesture should be spread outward (i.e. not towards the camera).

Images were captured as JPGs using the iPhone 6's "square" capturing capabilities (simply captures the images in square dimensions). The original images had dimensions in the

thousands of pixels, so I used Apple's Preview to resize images to 350x350. I found it quicker to do this once using Preview than have MATLAB do the resizing on every run. In reality, it would probably be best to have the images come straight from the camera in this format.

After reviewing my iPhone, I realized that I took a (shocking) 179 images over the course of this project. This high number was primarily because of the number of domains I was testing (do I wear a sleeve, no sleeve, or with the sleeve partially rolled up? Should I use a black background or a white background with a glove?). Once I settled on a domain, it was quite easy to get good pictures.

For this documentation, I used the following number of images for each gesture: **Fist** - 11; **Four** - 4; **HangLoose** - 3; **Loser** - 2; **OK** - 1; **Peace** - 5; **Point** - 4; **Rock** - 3; **Splay** - 9; **Three** - 3; **ThumbsUp** - 3.

Note: I will use this original input image as the basis for the step-by-step analysis in the following sections. In other words, most of system generated images shown in previous sections will be based on this image (it will be clear when this is not the case as different gestures will be used).

Example Image (Splay)



Step 2: Data Reduction

Vocabulary

At this point, it's useful to cover the system's vocabulary. Note that the vocabulary is a combination of the where and that what. Examples include [Quadrant 4, RockOn], [Quadrant 5, ThumbsUp], and [Quadrant 7, Fist].

Vocabulary - The Where

The system divides the image into nine quadrants that are padded with a margin to create some "dead-space." This "dead-space" is designed to make sure the user is clear in where he wants the gesture to be recognized. If the gesture falls into the dead-space, the system will return **-1** as the quadrant, ignore the gesture, and ask the user to try again.

The margin, which can be thought of as a border around the quadrants, is 5% of the length of the square image. In this case, where the image is 350 x 350 pixels, the margin is **17.5 pixels**. Note that this margin does not stack. In other words, the distance between all quadrants will be 17.5 pixels, not 35 pixels. Similarly, the distance from any of the outside quadrants to an edge of the image is 17.5.

The square quadrants have a side length based on the pixels not used as a margin. Specifically, each quadrant will have sides of the following length:

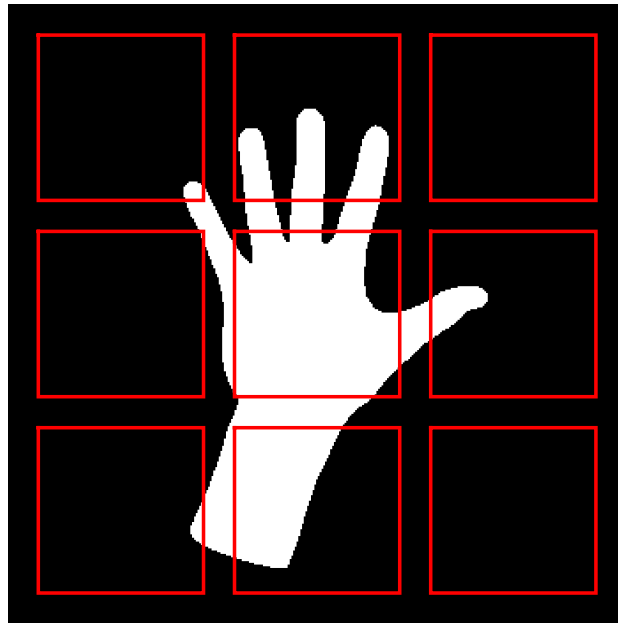
$$\begin{aligned} \text{quadrant side length} &= ([\text{length of image}] - 4 * [\text{margin size}]) / [\# \text{ of quadrants}] \\ &\quad \text{in this case} \\ \text{quadrant side length} &= (350 - 4 * 17.5) / 3 = \mathbf{93.3 \text{ pixels}} \end{aligned}$$

The quadrants are numbered as follows:

Quadrants of square image



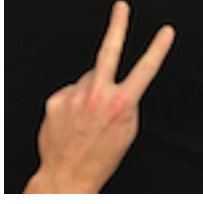
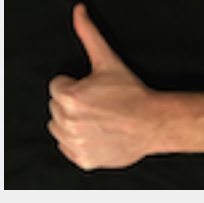
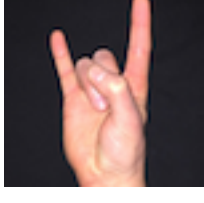
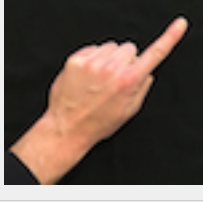
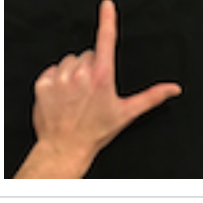
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |




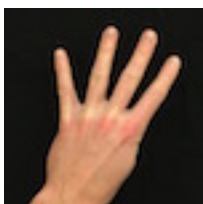
They can further be visualized by the following image. Quadrants are represented by the red rectangles and are buffered by the margin:

Quadrants visualized over a splayVocabulary - The What

The system can recognize the following 12 gestures, with one of those being the “Unsure” gesture. Details of how the system recognizes each of these are revealed later. In all of these cases, the hand can face either way (palm up or down) as long as the lighting does not create internal shadows. Gestures can be made in any direction. In other words, the hand can be right-side up, sideways, upside down, or anything in between. However, it does need to be in the desired quadrant.

Note: It's assumed that the fingers are spread apart from each other by at least a couple pixels' distance.

| Gesture | Description | Example |
|------------------|---|---|
| Fist | All fingers tucked into a closed hand |  |
| Splay | All fingers spread out |  |
| Peace | Only the index and middle fingers are spread outwards while the other fingers are tucked into the palm. |  |
| Thumbs Up | All fingers are tucked into the palm except for the thumb, which extends outward. |  |
| Rock On | The pinky and index fingers extend outward while the other three fingers are tucked into the palm |  |
| Point | Any lone finger extended outwards, with the exception of the thumb, while the other fingers are tucked into the palm. Multiple adjacent fingers (such as an index and middle finger) can be extended as long as they are extended together (touching along the sides). |  |
| Loser | Extending the thumb and the index finger while tucking the other fingers into the palm. |  |

| Gesture | Description | Example |
|-------------------|---|--|
| Hang Loose | Extending the pinky and thumb while tucking the other fingers into the palm. |  |
| OK | The index finger and the thumb come together to form a circle while the other fingers are extended. This gesture is used to “submit” the password for processing. |  |
| Three | Three fingers extended with the others closed into the palm. |  |
| Four | Four fingers extended with the others closed into the palm. |  |
| Unsure | Any gesture that does not fall into one of the above categories and their specifications. | N/A |

Reducing the Data

Every image read into the system goes through a series of steps to determine its “what” and “where.”

The Main Script - HW1c.m

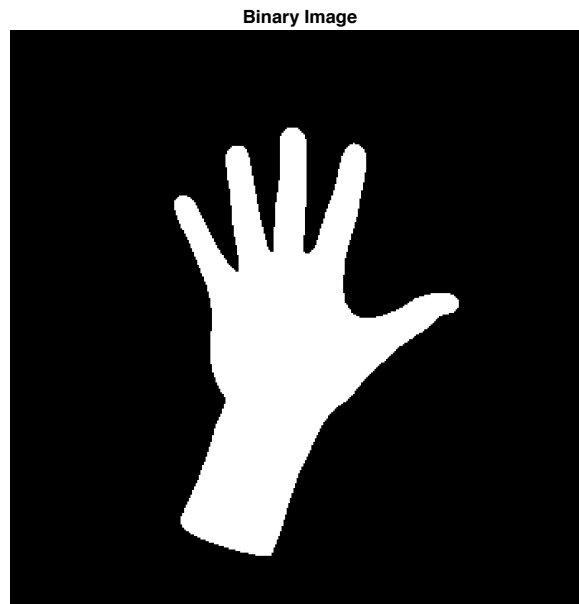
The main script that the system runs from is HW1c.m. It’s responsible for loading the filenames for the images, calling the functions detailed below, and then telling the user if the password matches.

Note: The code to get all filenames from a directory of images was obtained from a Stack Overflow [Eshan 2013].

Getting the Binary Image of Skin - **getBasicImages()**

To get a binary image with 1s for the arm and hand and 0s for the background, the color image is loaded and then converted it to a grayscale. The grayscale image is then converted into a binary image using a threshold luminance level. Specifically, any pixel with a luminance greater than 60/255 is considered skin while anything less is considered the background. 60/255 was chosen by trial and error.

Often times, small specks of lint or dust or simply an odd reflection of light could cause false positives. Tiny connected components of just a few pixels were appearing in the binary image as skin when it was really just background noise. Although this could be avoided by removing the debris and retaking the picture, I decided to make the system more robust by having it remove connected components with an area less than 20 pixels. This resulted in binary images that accurately represented the skin and nothing else.



Detecting the Password Submission Gesture - **isSubmitPasswordGesture()**

At this point, the boundaries of the image are checked to see if an internal hole greater than a certain size exists within the hand. This hole is assumed to be the hole formed by making the "OK" gesture, which the system recognizes as a cue to process previous gestures as the password.

The hole must be large enough for it to be considered the "submit" ("OK") gesture. If it takes more than 75 points to represent the boundary of this internal hole, the gesture is recognized, the analysis stops, and the password sequence is submitted. If not, the system moves on to analyze the gesture like any other (see steps below).

Detecting the Quadrant and Gesture - `getQuadrantAndGesture()`

We now need to determine the location and the gesture in the image. This function calls a number of other functions (below) to determine this information.

Determining Palm Properties - `getPalmProperties()` <- called by `getQuadrantAndGesture()`

The function serves the following purposes:

1. Find the center of the palm
2. Find the radius of the palm
3. Get the boundary points of an approximate mask for the palm

Note: The inspiration for finding the three palm properties listed above came from (Chen, 2014). Although the paper provides no code, it provided the strategies for determining these characteristics as well as an algorithm to find the palm mask. I made very minimal changes to this algorithm and attribute the credit to their work.

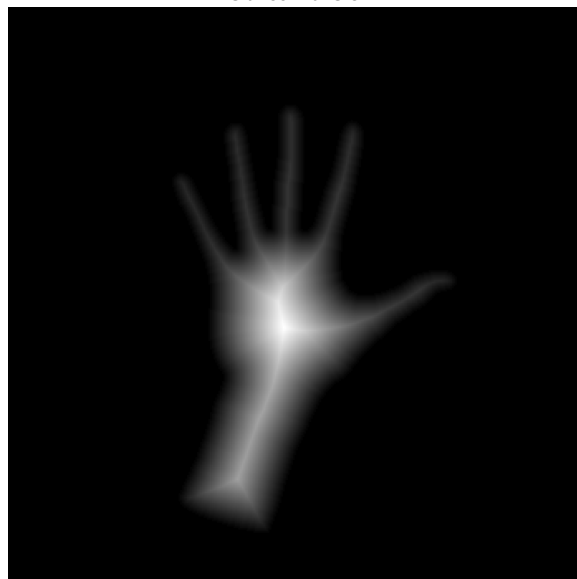
Finding the Palm's Center and Radius

Determining the palm's center and radius is rather easy when you realize that the palm is the "widest" object in the image.

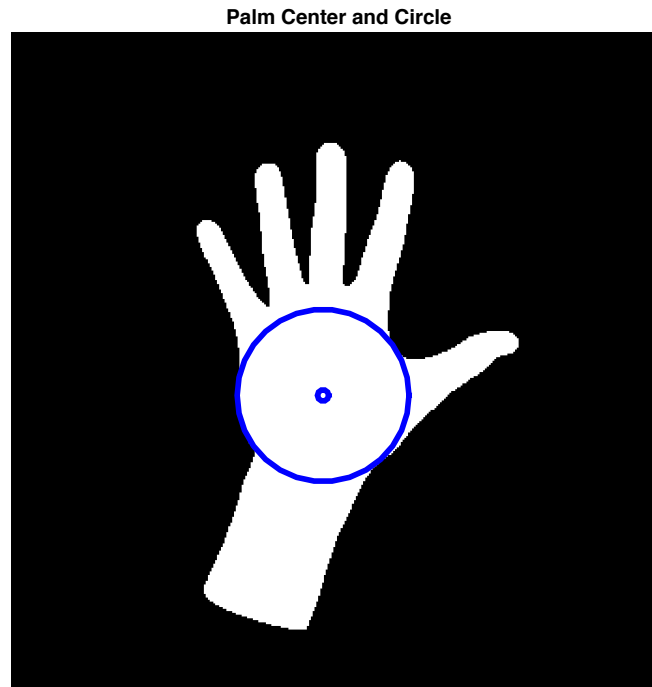
To define the use of the term "widest," lets think of the hand and arm in the binary image as a single object. The object is represented by 1s and the boundary around it (and the rest of the background) with 0s. The pixels within this object (the 1s) can be labeled with a distance to the nearest boundary pixel (the 0s) of the object using a "distance map" (sometimes called a distance transform).

MATLAB has a built in function (`bwdist()`) that computes this distance map and all of the information above, however it finds the distance to the nearest non-zero pixel. Because of that, we pass in the *inverse* of the binary image where the the object is represented by 1s. This makes the background 1s and the object 0 and results in the following data/image:

Distance Transform



With this concept and computed matrix, we can say the palm is the widest object in the image as it contains a pixel that has the greatest distance to the nearest boundary. Specifically, this pixel with the greatest distance to the nearest boundary is the **center of the palm**. Furthermore, the distance from this center point to the boundary is considered the **palm radius**, which lets us form the **palm circle** as shown below.



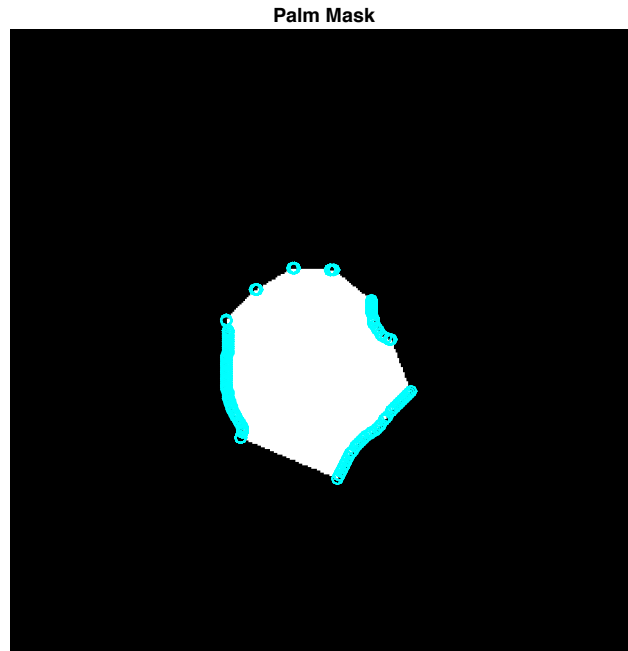
Finding the Approximate Mask for the Palm

The palm mask is represented by 360 boundary points gathered from the algorithm described below. For this procedure, we will make use of the palm center, palm radius, palm circle, and the image's distance map found above.

The algorithm accrues a list of boundary points for the mask by making a decision on 360 points evenly-spaced apart on the palm circle. To accomplish this, it iterates from angle = [1 up to 360] (the degrees in a circle). At each iteration, it uses the palm center, palm radius, and angle to determine the point to be examined using a bit of trigonometry. For each of these 360 points, it does the following:

- If the pixel at this point is the background, that point is added to the list of boundary points being accrued.
- If the pixel at this point is a part of the hand, the distance map is used to find the nearest background pixel. This boundary pixel is then added to the boundary points being accrued instead.

At the end, the algorithm has accrued 360 boundary points that represents the palm mask (shown below with points as teal 'o's). This mask is going to be used to determine the location of the wrist and to isolate the fingers.



Locating the Wrist - **maxDistanceBetweenSequentialPoints()** <- *called by*
getQuadrantAndGesture()

Note: Although no code was provided, the idea for how to identify the wrist as described below came from [Chen 2014].

To locate the wrist, the system determines the endpoints of the wrist line and then calculates its midpoint as the center of the wrist.

The two wrist endpoints are determined by using the 360 mask points accrued for the palm. If we look closely at the algorithm, we'll notice that the majority of sequential mask points around the palm are relatively close together. However, the exception to this case is when we start to look at points along the palm circle that are close to the wrist. Since these points are not a part of the background, the algorithm will instead acquire the nearest boundary point in its list.

In most cases around the palm, the algorithm will grab the nearest boundary point by reaching farther away from the palm center. However, when looking at points on the palm circle near the wrist, the algorithm won't just reach further away from the palm center because that would take it further down the arm. Instead, since it goes towards the nearest boundary, it will go to one side of the wrist (specifically, whichever side is closest). What's key here is that as we move around the palm circle, the algorithm switches from going to one side of the wrist for the nearest

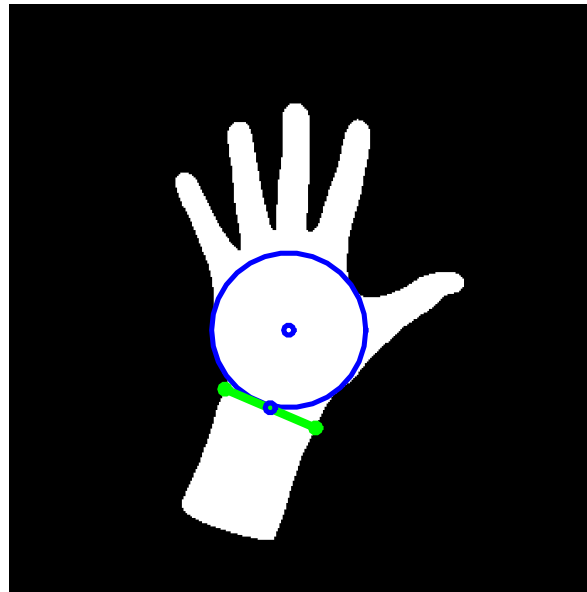
boundary point to going to the other. Because of this switch, the distance between these two sequential points will be noticeably larger than the rest. This can be seen in the image under "Finding the Approximate Mask for the Palm."

With that said, we can determine the two wrist endpoints by finding the sequential points representing this switch. To do this, we traverse the the 360 boundary points and determine which two have the furthest distance between them. For completeness, we also need to compare point 1 to point 360. The two points we find as a result of this are the wrist endpoints.

The wrist midpoint is determined by simply finding the middle point between the two endpoints (averaging the x and y coordinates).

Note: the algorithm really isn't "reaching out" in a particular example by searching the space. I just explain it this way as it makes things a little more intuitive. In reality, it knows exactly which point it is going to grab by using the distance map. See the previous section for more details.

The image below shows this wrist information, as well as the previous characteristics we've been capturing so far.

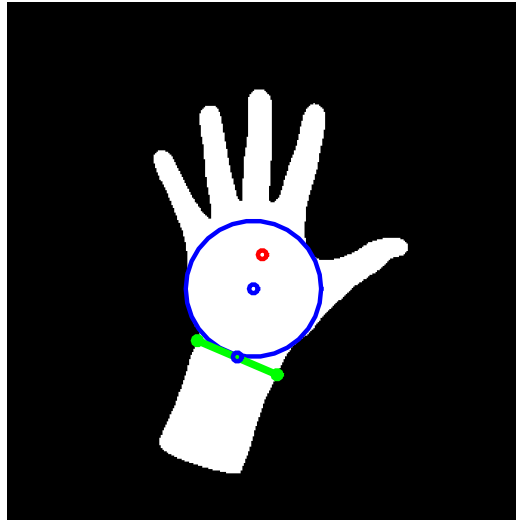


Locating the Hand Centroid - **getApproximateHandCentroid()** <- called by *getQuadrantAndGesture()*

The center of palm is typically not the centroid for the hand as any extended fingers pulls the center of mass towards them. Even if no fingers are extended, the centroid is slightly different than the palm center due to the added width of the enclosed fingers.

To accommodate this, I approximate the centroid by first finding the orientation of the hand relative to the wrist midpoint. Specifically, this is the directed vector from the wrist to the palm

center. The approximate centroid is then found by continuing in this direction by have the length of this vector from the palm center. This centroid is shown below in red.

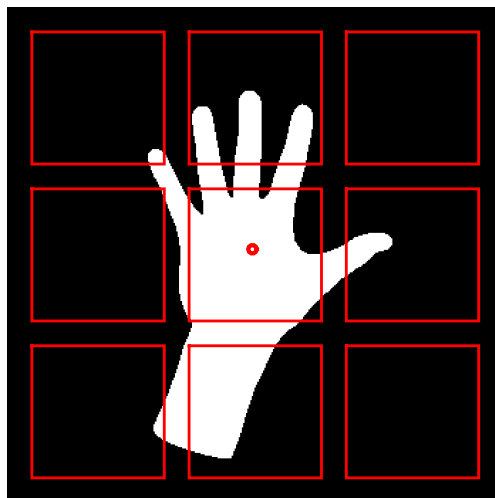


Locating the Gesture - **getQuadrant()** <- called by *getQuadrantAndGesture()*

To determine which quadrant the gesture is in, I use the approximated centroid which was determined when getting the palm properties.

With knowledge of the margin and quadrant size as described in the “Where” vocabulary above, finding out which quadrant the gesture falls in is pretty straight forward. Please see the attached code.

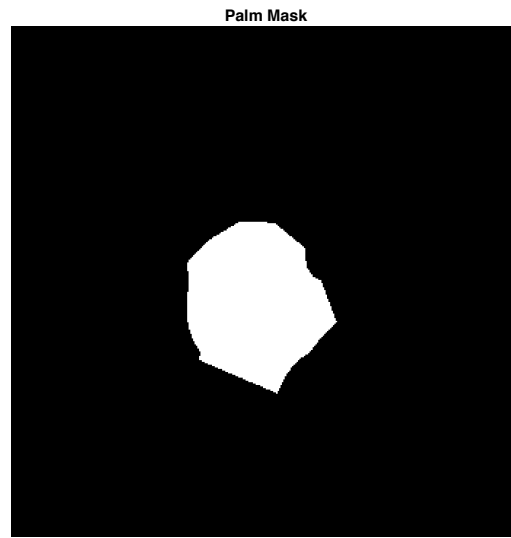
The number returned by this function is equal to the quadrant number the palm center lands in. If the palm center lands in the margin space, a -1 is returned so the system knows the gesture could not be placed. If the gesture’s location could not be placed, the gesture is ignored and the user is prompted to try again.



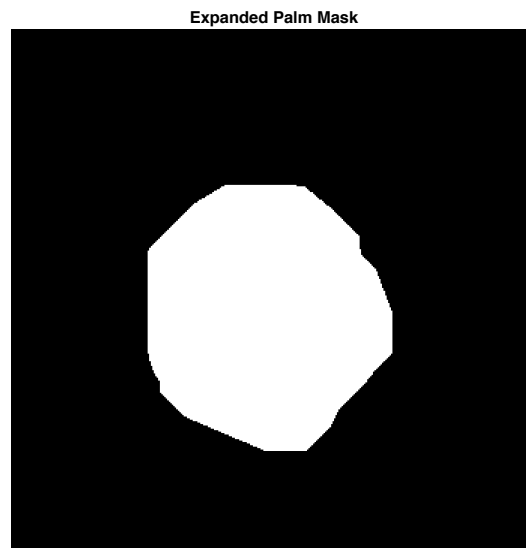
Separating the Fingers - **getFingers()** <- called by *getQuadrantAndGesture()*

To obtain the fingers as distinct connected components, the system uses the palm mask it determined in obtaining the palm properties.

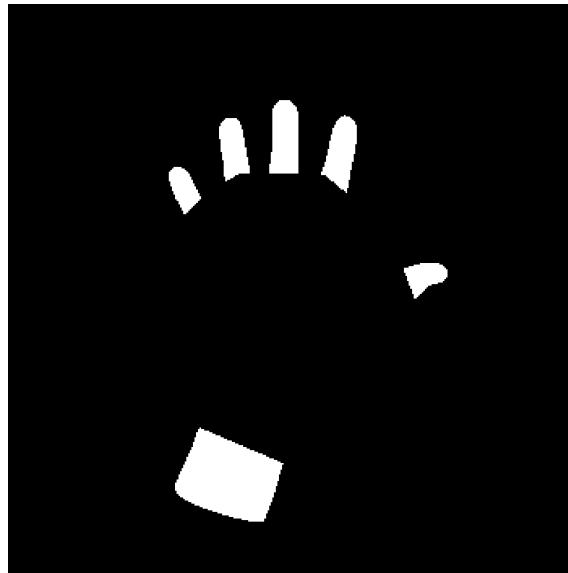
First, it creates a new binary image of the same size as the input (350 x 350) that only contains the palm mask (previously shown under “Finding the Approximate Mask for the Palm”).



Second, to make sure that it encompasses the entire palm, it is dilated by a disk structure with a radius of the palm radius / 1.5. This ratio was found by trial and error, and it creates an object that is almost certain to encompass all of the palm and more.



A new binary image is then created by taking the original one and subtracting the dilated mask. This subtraction leaves only extended fingers and the arm, but they have all been disconnected from each other. Subtracting the dilated palm mask can also remove pieces of the fingers and arm, but this is fine as we will not need entire fingers/arms for our analysis. In fact, it's actually safer to subtract a little more than a little less to ensure that the entire palm has been removed in the new binary image. If pieces of the palm are left in the image, it will adversely affect the analysis.



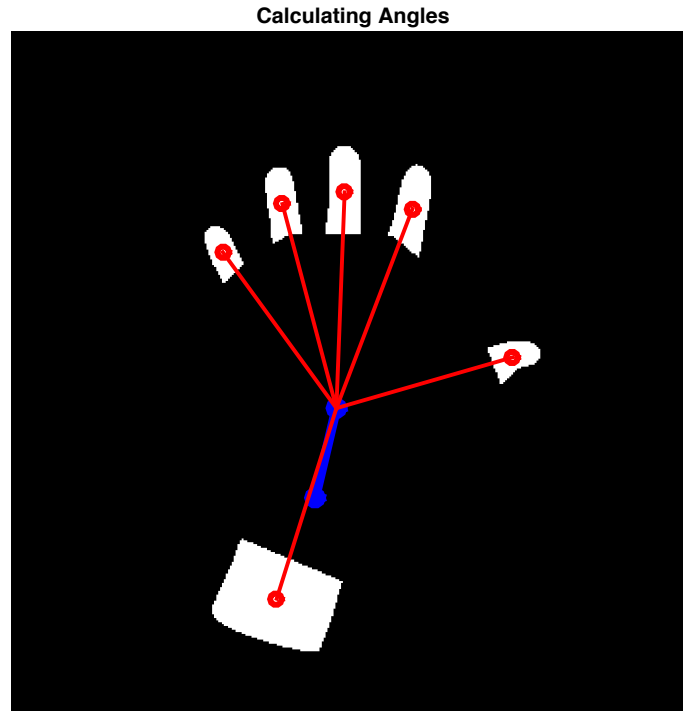
Recognizing the Gesture - **getGesture()** <- called by *getQuadrantAndGesture()*

Now that we have the palm center, the wrist center, and a binary image with the arm and individual fingers as distinct connected components (no palm), we can begin to analyze the image.

We begin by getting the centroid for the connected components. This will give us a centroid for each of the extended fingers and the arm in the image with the palm removed. This is done using MATLAB's `regionprops()` function.

Note: When I use the term palm, wrist, and finger below, I'm referring to the center of the palm, center of the wrist, and centroid of the finger connected component, respectively.

We'll then calculate the counterclockwise angles formed by the palm-to-wrist vector to the palm-to-finger vectors by iterating over these centroids. The angle to the arm will be ignored (see the **getFingerAngles()** function described next). The angles are calculated based on the vectors in the image below. The blue line represents the palm-to-wrist vector, while the red lines represent the palm-to-finger vectors. Again, there is a line from the palm to the arm, but this will be ignored.



The number of angles the function gets back will tell it the number of fingers that are extended for the gesture. Certain gestures can be determined simply by looking at the number of fingers. Specifically:

- **0 fingers** present implies the gesture is a **fist**
- **3 fingers** present implies the gesture is a **three**
- **4 fingers** present implies the gesture is a **four**
- **5 fingers** present implies the gesture is a **splay**

However, the presence of 1 or 2 fingers is where things get more interesting. Two functions, **getOneFingerGesture()** and **getTwoFingerGesture()** are detailed below and describe how the angles are useful for determining more complex gestures.

Once the gesture is determined by one of the above scenarios, it is returned to be added to the password sequence.

Calculating the Angles to the Fingers - **getFingerAngles()** <- called by *getGesture()*

*Note: The line of code that initially calculates the **angle** variable in this function was found on the MATLAB Central forums [Stafford 2010]. A thread existed where a user posted a question about finding the counterclockwise angle formed by two directed vectors that shared an endpoint, and another user posted a solution for use.*

*Note: When I use the terms **palm**, **wrist**, and **finger** below, I'm referring to the center of the palm, center of the wrist, and centroid of the finger connected component, respectively.*

This function calculates the angle formed by the palm-to-wrist vector to the palm-to-finger vectors by iterating over the finger centroids. It uses a bit of linear algebra and trigonometry, which can be seen in the line of code that initially sets the **angle** variable in the **for** loop.

Note, the angle to the arm is filtered out and is not returned in the **angles** array. The centroid that belongs to the arm is easily distinguished as the palm-to-arm vector creates a very small angle with the palm-to-wrist vector. To handle this, I filter out any centroid that creates a palm-to-centroid vector with an angle less than 45° to the palm-to-wrist vector (in either direction - clockwise or counterclockwise). This is a fairly safe threshold as I cannot bend my fingers anywhere close to that degree.

And the end, the angles are returned to the **getGestures()** function for analysis.

Identifying Single Finger Gestures - **getOneFingerGesture()** <- called by **getGesture()**

There are only two single finger gestures recognized by the system: the **Point** and the **Thumbs Up** sign. To distinguish the two, I simply determine if a thumb is present by using the finger angles (see the **containsThumb()** function described next). If it does, the gesture must be a Thumbs Up. If it doesn't the gesture is a Point. The correct one is then returned by the function.

Check for a Thumb - **containsThumb()** <- called by **getOneFingerGesture()** and **getTwoFingerGesture()**

This function returns true if a thumb is present in the image based on the angles passed in as an argument.

Because of the direction the thumb extends in relative to the other fingers, it is quite easy to identify when given the angles calculated above. Specifically, this function checks to see if any of the angles formed by the palm-to-wrist and palm-to-finger vectors is less than 135° (clockwise or counterclockwise). If such an angle exists in the array that was passed in as an argument, then the image contains a thumb. Otherwise, the image doesn't contain a thumb.

Identifying Two Finger Gestures - **getTwoFingerGesture()** <- called by **getGesture()**

Again, the angles that were calculated above will be used to determine which two finger gesture is being made in the image.

We can limit the possible gestures to **Hang Loose** and **Loser** if we can determine that a thumb is present. For this, we can again use the **containsThumb()** function. If a thumb exists, then we simply need to analyze the angle between the thumb and the other finger to make our selection. For this, I determined that if this angle is greater than 85°, then it is the Hang Loose gesture. If it isn't, then it's the Loser gesture.

If no thumb is present, then we need to distinguish between the **Peace** sign and **Rock On** gesture. Again, the angles are quite useful. I determined that if the angle between the two fingers are less than 30° , then the gesture is the **Peace** sign. Otherwise, it is the **Rock On** gesture.

The chosen gesture is then returned.

End of Gesture Analysis

After the previous steps are completed, we'll have the quadrant that the hand is located in and the gesture that the user is showing. These two features come together to provide a single "word" in the vocabulary (i.e. [Quadrant 5, First]). A sequence of this vocabulary is then built up and checked against the correct password, as described in the next step.

Step 3: Parsing and Performance Step

The grammar consists of the following: any of the gestures can be made in any order, with repetitions allowed, and in any quadrant. However, the submitted gestures and quadrants they are located in must correspond to the predefined password. For example, the predefined password might be a fist gestured in quadrant five, followed by a splay displayed in quadrant 7. If this is the case, then the user must submit the same gestures in the same quadrants ([fist, 5], [splay, 7]) to be granted access.

To submit a password represented by a sequence of gestures, the user must gesture “OK” anywhere in the image (see **Step 2** for a reminder of this vocabulary). It does not have to be placed in a specific quadrant.

All gestures from the start up to but not including the “OK” gesture is considered a part of the password. Their location and type of gesture are analyzed and compared to the predefined password to determine a match.

The gestures entered by the user are compared to the predefined password in the **passwordMatches()** function. If all quadrants and gestures match up, the user is granted access. If not, the user is denied access.

For the sequences below, please use the following definitions:

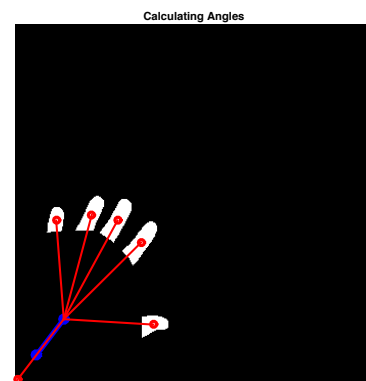
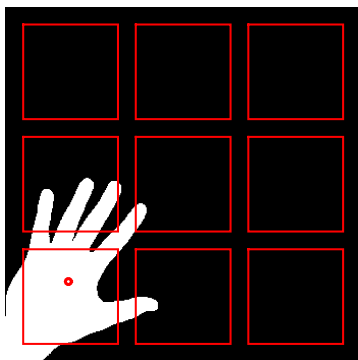
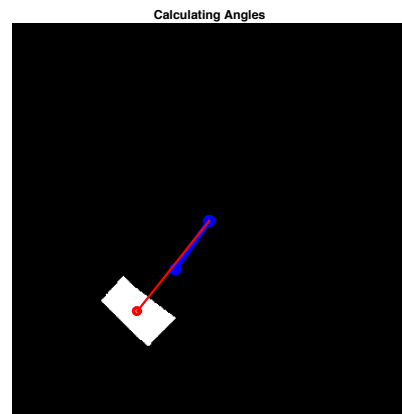
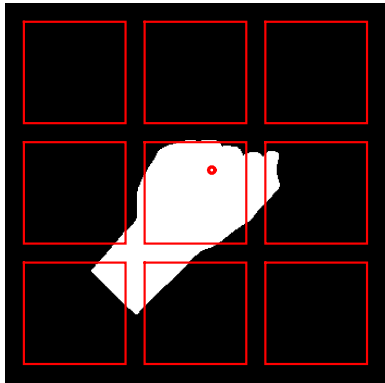
- **System Password:** the password the system is looking for
- **User Entered:** the sequence the user intended to enter
- **System Recognized:** the sequence the system recognized

Recall: The system sees the **OK** gesture as the “Submit” button. Although it is included at the end of every user entered sequence, it does not count as a symbol in the password.

Sequence 1: Successful

This is the minimal feature set required by the assignment.

System Password: [Fist, 5]; [Splay, 7]
User Entered: [Fist, 5]; [Splay, 7]; [OK, 5]
System Recognized: [Fist, 5]; [Splay, 7]; [OK, 5]
Response: Access Granted

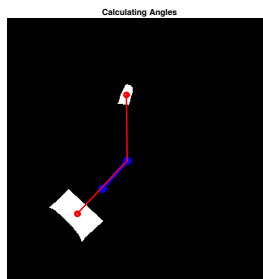
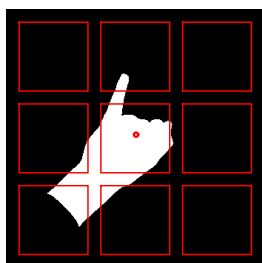
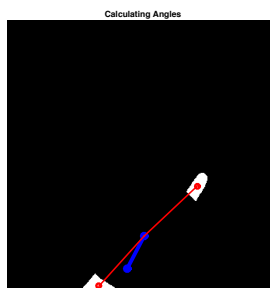
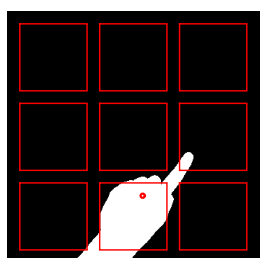
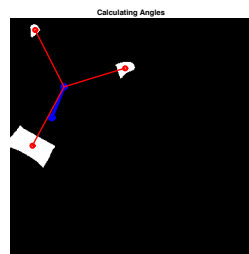
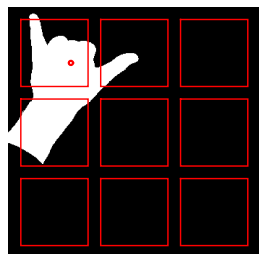
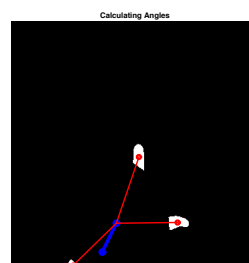
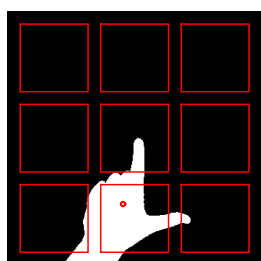


Sequence 2: Successful

This sequence is meant to demonstrate more complicated gestures in more quadrants.

System Password: [Loser, 8]; [HangLoose, 1]; [Point, 8]; [Point, 5]
User Entered: [Loser, 8]; [HangLoose, 1]; [Point, 8]; [Point, 5]; [OK, 5]
System Recognized: [Loser, 8]; [HangLoose, 1]; [Point, 8]; [Point, 5]; [OK, 5]
Response: Access Granted

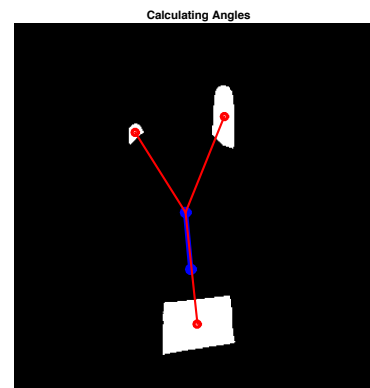
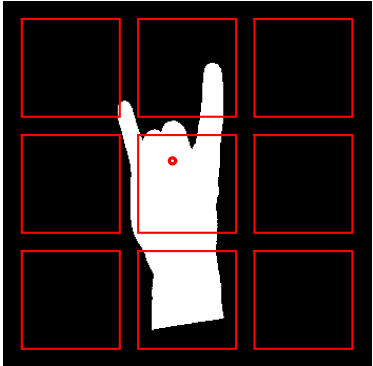
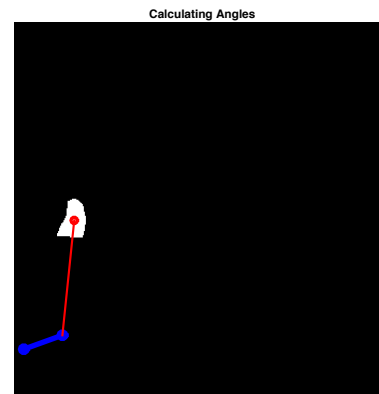
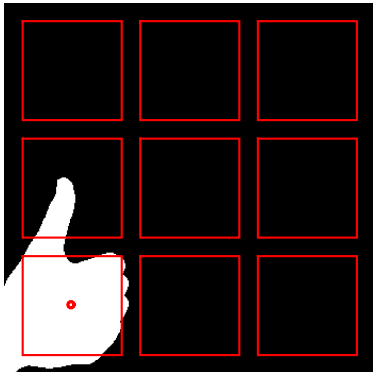
Note, the Point gesture in quadrant five was actually made with the pinky rather than index finger.



Sequence 3: Successful

Further demonstrations of different gestures.

System Password: [ThumbsUp, 7]; [RockOn, 5]
User Entered: [ThumbsUp, 7]; [RockOn, 5]; [OK, 5]
System Recognized: [ThumbsUp, 7]; [RockOn, 5]; [OK, 5]
Response: Access Granted



Sequence 4: Successful

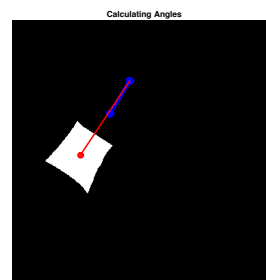
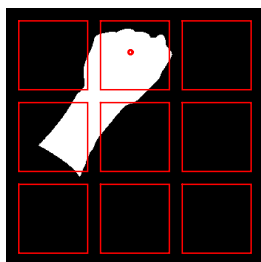
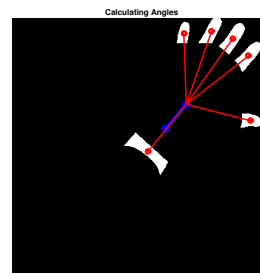
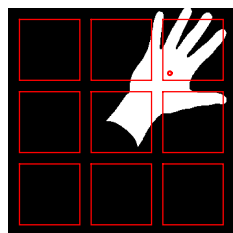
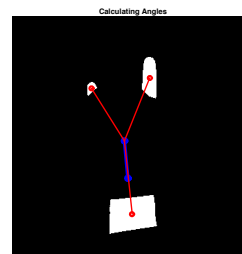
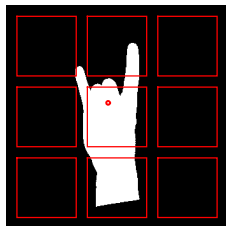
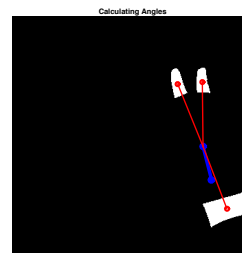
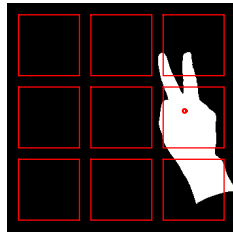
This sequence covers additional gestures and quadrants.

System Password: [Peace, 6]; [RockOn, 5]; [Splay, 3]; [Fist, 2]

User Entered: [Peace, 6]; [RockOn, 5]; [Splay, 3]; [Fist, 2]; [OK, 5]

System Recognized: [Peace, 6]; [RockOn, 5]; [Splay, 3]; [Fist, 2]; [OK, 5]

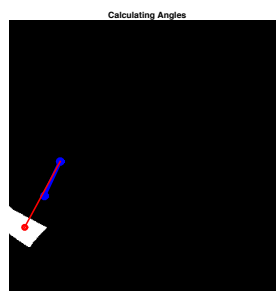
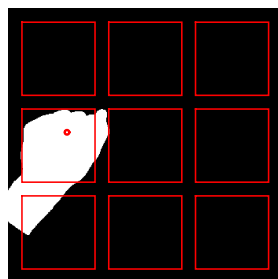
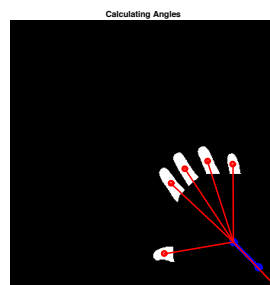
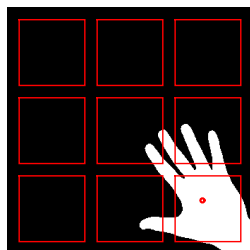
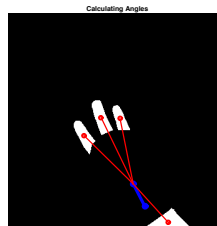
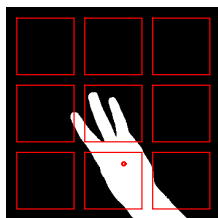
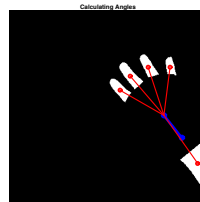
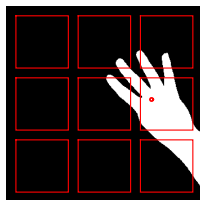
Response: Access Granted



Sequence 5: Successful

Demonstrating the remaining gestures and quadrants not covered in the previous sequences.

System Password: [Four, 6]; [Three, 8]; [Splay, 9]; [Fist, 4]
User Entered: [Four, 6]; [Three, 8]; [Splay, 9]; [Fist, 4]; [OK, 5]
System Recognized: [Four, 6]; [Three, 8]; [Splay, 9]; [Fist, 4]; [OK, 5]
Response: Access Granted

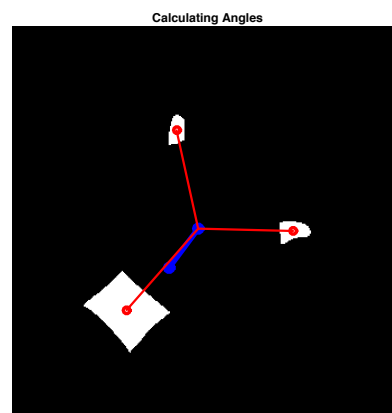
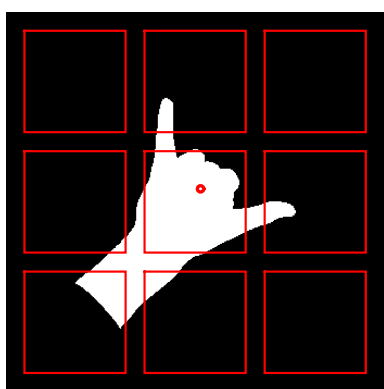
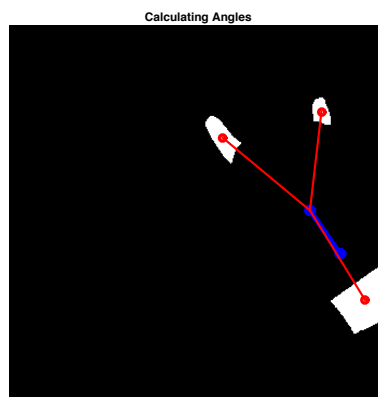
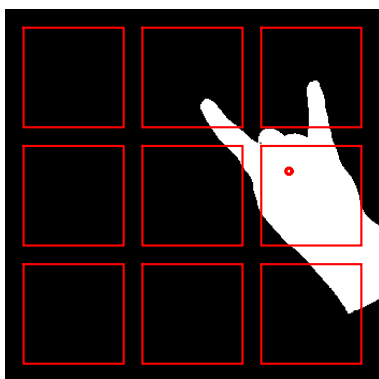


Sequence 6: Successful

Demonstrating the systems ability to recognize wrong passwords when distinguishing between four different gestures that use only two fingers.

System Password: [Peace, 6]; [Loser, 5];
User Entered: [RockOn, 6]; [HangLoose, 5]; [OK, 5]
System Recognized: [RockOn, 6]; [HangLoose, 5]; [OK, 5]
Response: Access Denied

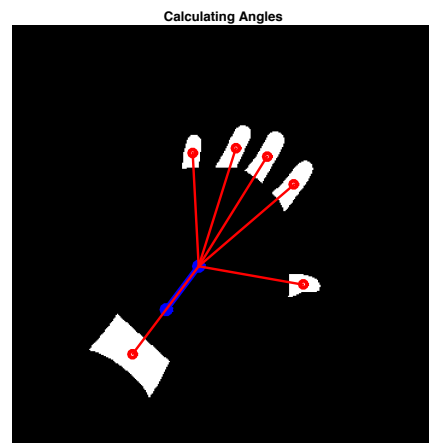
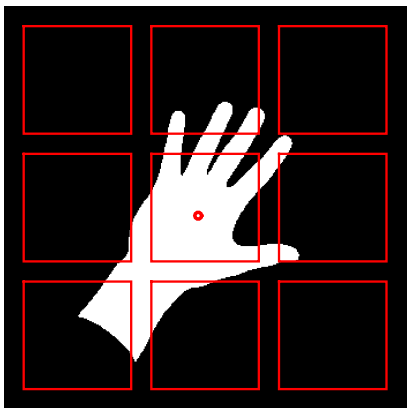
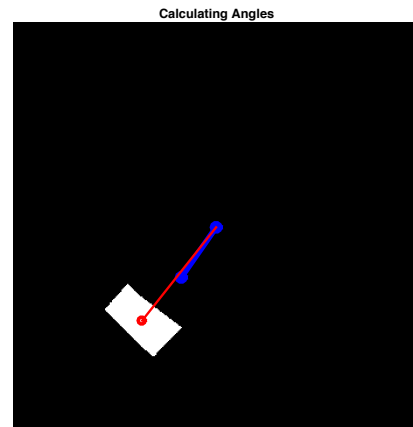
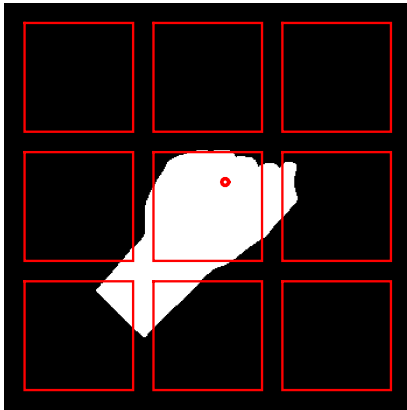
Notice that even though the RockOn is similar to Peace and HangLoose is similar to Loser, the system recognized that the gestures entered didn't match the password.



Sequence 7: Successful

Demonstrating the systems ability to recognize wrong passwords due to quadrant mismatches.

System Password: [Fist, 4]; [Splay, 8]
User Entered: [Fist, 5]; [Splay, 5]; [OK, 5]
System Recognized: [Fist, 5]; [Splay, 5]; [OK, 5]
Response: Access Denied

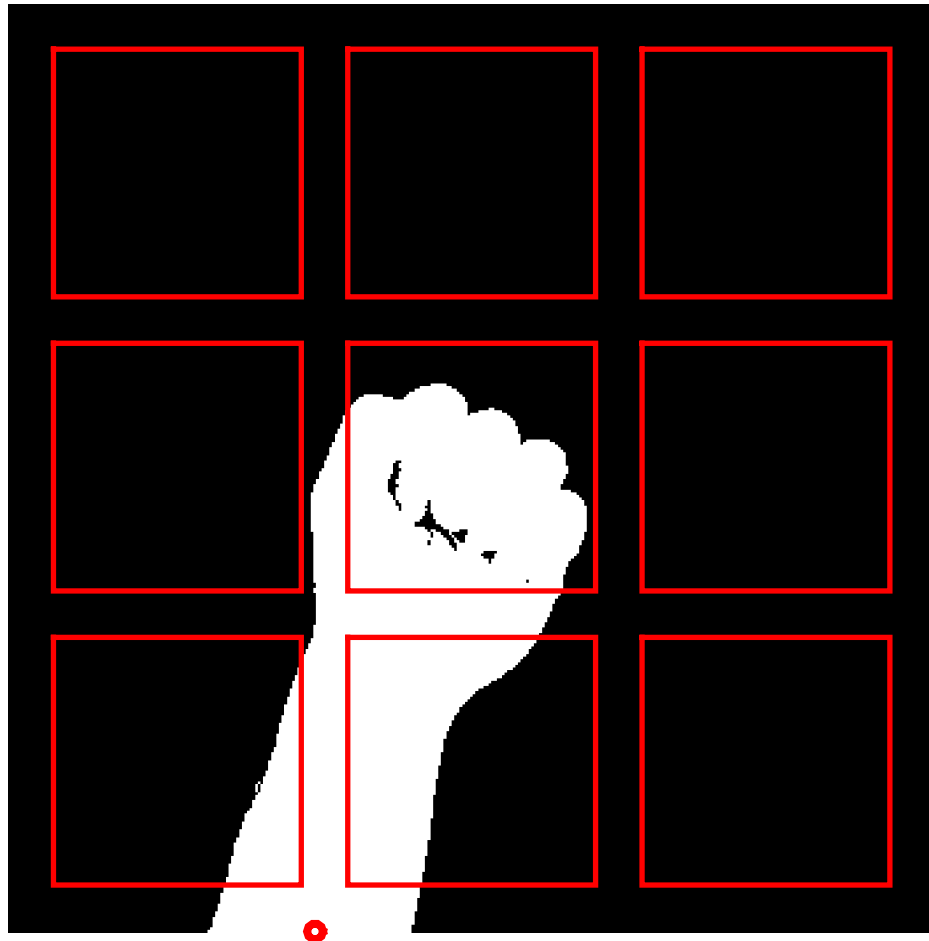


Sequence 8: Failure

To focus in on the issue, I'm going to use single gesture/quadrant passwords to demonstrate some failures.

System Password: [Fist, 5];
User Entered: [Fist, 5]; [OK, 5]
System Recognized: [Unsure, -1]; [OK, 5]
Response: Access Denied

As we can see from the image below, the intended gesture was clearly a fist in the 5th quadrant. However, because of the shadows created by the fingers, the widest part of the object determined by the distance map was the base of the arm (the red dot). Because of this, the system placed the object outside of any quadrant and rejected the gesture (labeled it as unsure).

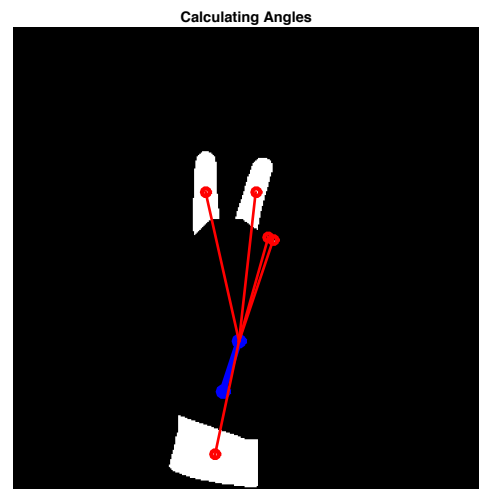
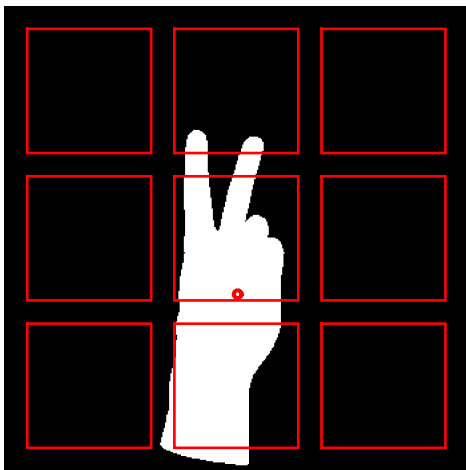


Sequence 9: Failure

To focus in on the issue, I'm going to use single gesture/quadrant passwords to demonstrate failures.

System Password: [Peace, 5];
User Entered: [Peace, 5]; [OK, 5]
System Recognized: [Four, 5]; [OK, 5]
Response: Access Denied

In this particular instance, the expanded palm mask did not completely remove the palm. Because of this, a couple of object points that were actually knuckles of the pinky and ring finger tucked into the palm remained after the mask removal. The system viewed these "stubs" as fingers and determined that there were four extended. The gestured was identified as a "four" rather than "peace" and the password was rejected.

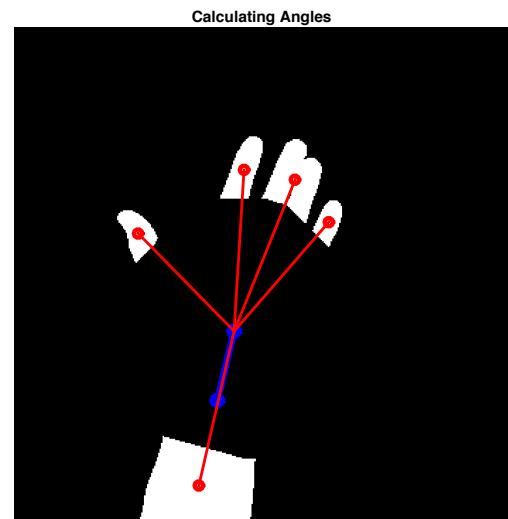
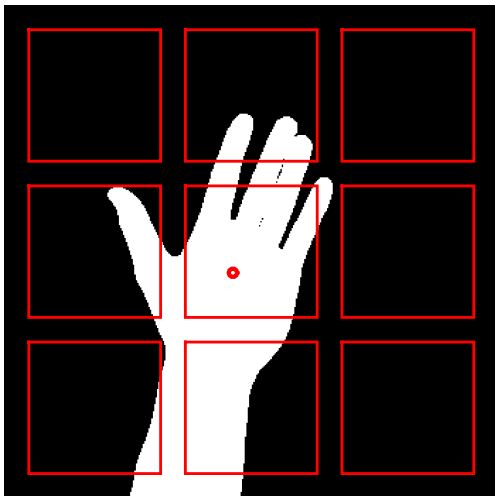


Sequence 10: Failure

To focus in on the issue, I'm going to use single gesture/quadrant passwords to demonstrate failures.

System Password: [Splay, 5];
User Entered: [Splay, 5]; [OK, 5]
System Recognized: [Four, 5]; [OK, 5]
Response: Access Denied

In this failure, the user presented a splay in quadrant five but his middle and ring fingers were touching. This caused the system to recognize these fingers as one object and declare the gesture a four. Access was denied.



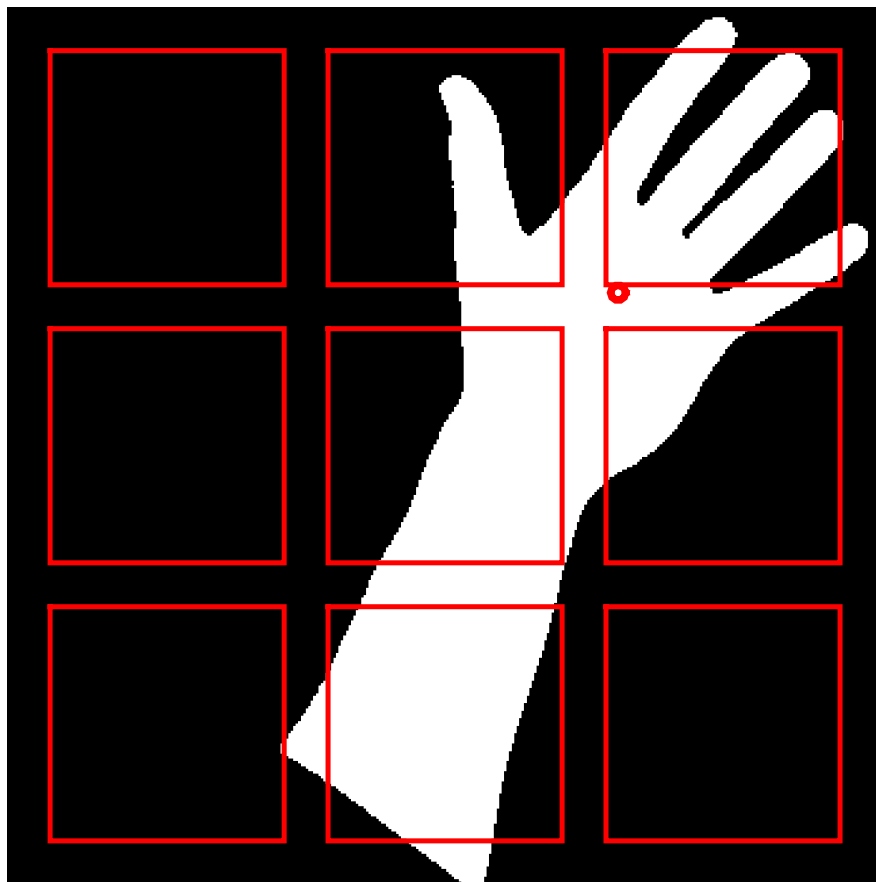
Sequence 11: Failure

To focus in on the issue, I'm going to use single gesture/quadrant passwords to demonstrate failures.

System Password: [Splay, 3];
User Entered: [Splay, 3]; [OK, 5]
System Recognized: [Unsure, -1]; [OK, 5]
Response: Access Denied

In this failure, the user meant to display a splay in the third quadrant. His fingers are up at the edge of the image, but the centroid approximated fell just outside of the desired area. Because the location couldn't be placed with confidence, the gesture was rejected.

This could potentially be resolved by pulling the camera further away. The hand would then shrink and, if the fingers were still touching the top right corner, the centroid would fall in the appropriate quadrant.

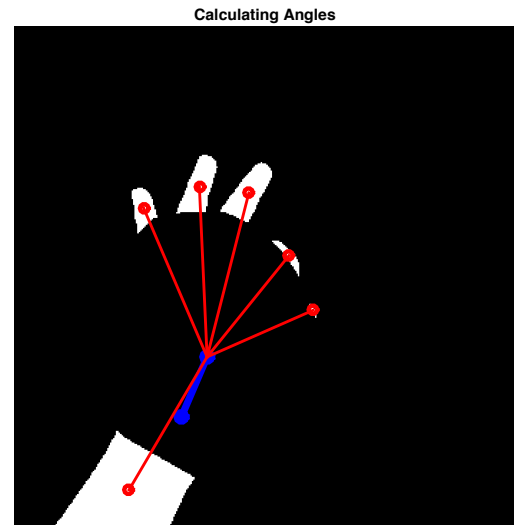
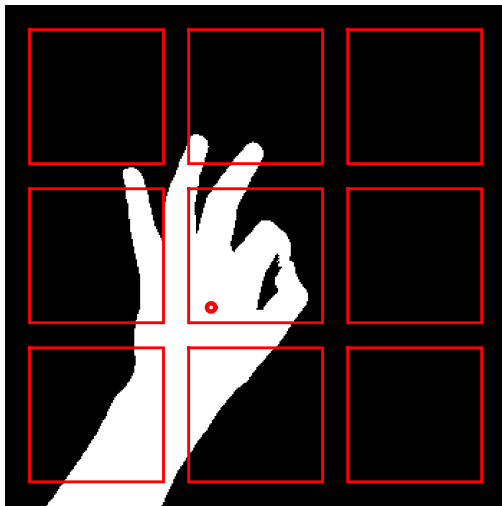


Bonus Sequence 12: Failure (just to make sure I got the “failure” category right)

To focus in on the issue, I’m going to use single gesture/quadrant passwords to demonstrate failures.

System Password: [Splay, 3];
User Entered: [Splay, 3]; [OK, 5]
System Recognized: [Splay, 3]; [Splay, 5]
Response: Access Denied

In this failure, the user attempted to submit his password using the “OK” symbol, but the hole formed by the index and thumb fingers was not large enough to trigger the “submit” action. This was due to the hole being at an angle to the camera rather than directly facing it.



Step 4: Creativity

The creative was built into the previous step, but I thought I'd comment on it at a high level here.

My primary advancements were in two areas. The first and easier one was in separating the palm from the fingers. Once you can do this in an acceptable manner, you can count the number of fingers and create six acceptable gestures from this alone (the fist and then five possibilities from extending a certain number of fingers).

The second and more interesting was on identifying the orientation by location the center of the palm and the wrist, and then determining gestures based on the angle formed with the fingers. This made it possible to identify the arm and thumbs, and distinguish between what might be considered similar gestures (both the "peace" and "rock on" gestures extend two fingers, but form noticeably different angles).

Although determining the orientation and angles was a bit more complicated, it isn't too difficult to follow the process. Furthermore, it made the system significantly more robust than just counting how many fingers. For example, instead of considering all two finger gestures to be the same, the system can distinguish between four of them.

I considered using the orientation further to create more complicated gestures, but I noticed that it wasn't too comfortable to continue to turn my hand/wrist to indicate direction.

References

Chen, Kim, Liang, Zhang, Yuan. "Real Time Hang Gesture Recognition Using Finger Segmentation." April 2014. <http://www.hindawi.com/journals/tswj/2014/267872/>

Ehsan (username). Stack Overflow. "Load all the images from a directory." March 2013. <http://stackoverflow.com/questions/15655177/load-all-the-images-from-a-directory>

Stafford, Roger. "Finding angle between two lines. MATLAB Central. March 2010 http://www.mathworks.com/matlabcentral/newsreader/view_thread/276582

I want to specifically comment that the paper [Chen, 2014] was particularly influential. I really liked what they did and tried to replicate several aspects of it (they provided no code, just a discussion on how they solved certain problems).

Finally, I used MATLAB's documentation extensively.