Detecting Shapes

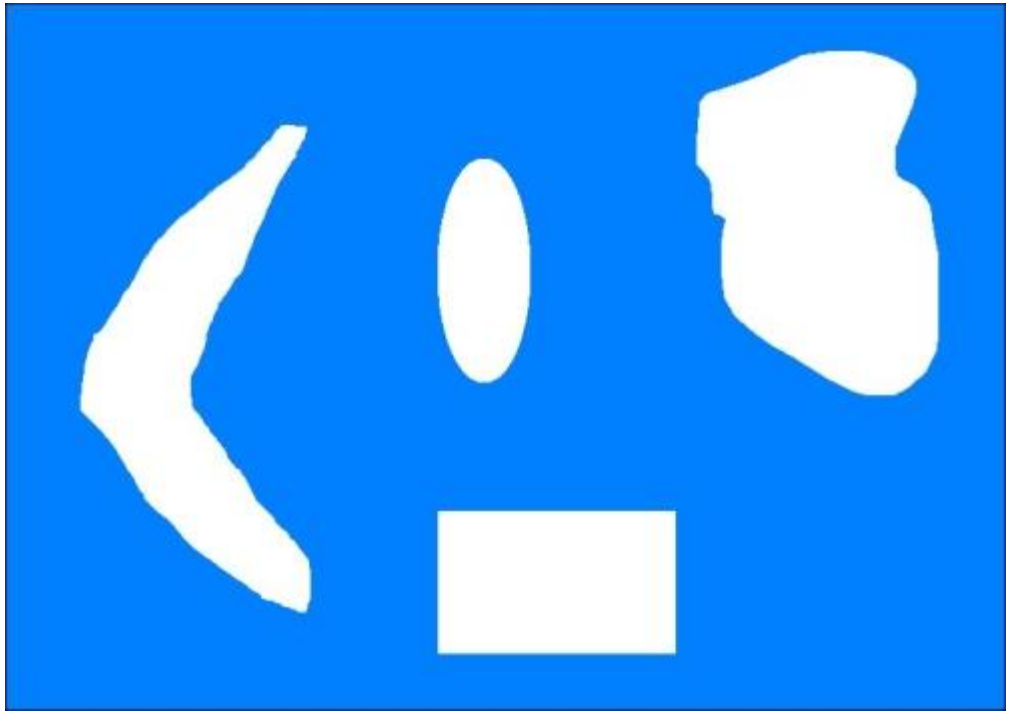
recognize shapes and estimate the exact boundaries

Contour analysis

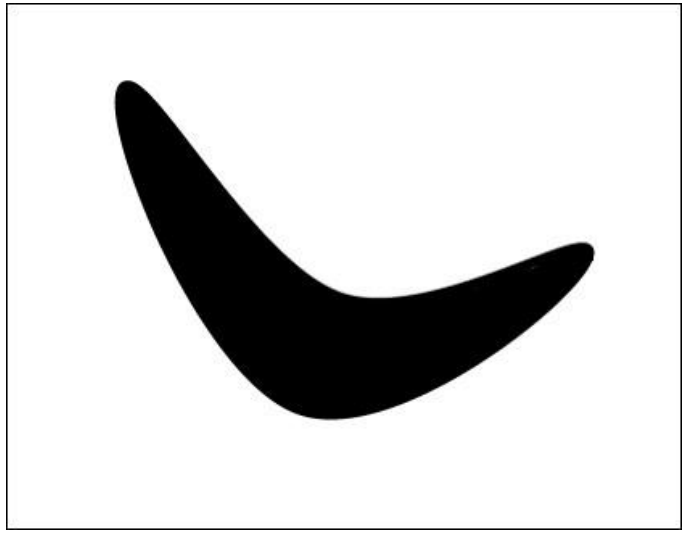
- Contours can be explained simply as a curve joining all the continuous points (along the boundary), having same color or intensity
- Contour analysis is a very useful tool in the field of computer vision
- We deal with a lot of shapes in the real world and contour analysis helps in analyzing those shapes using various algorithms
- When we convert an image to grayscale and threshold it, we are left with a bunch of lines and contours
- Once we understand the properties of different shapes, we will be able to extract detailed information from an image

Contour analysis

- Let's say we want to identify the boomerang shape in the following image:



In order to do that, we first need to know what a regular boomerang looks like:



Contour analysis

- Now using the above image as a reference, can we identify what shape in our original image corresponds to a boomerang?
- If you notice, we cannot use a simple correlation based approach because the shapes are all distorted. This means that an approach where we look for an exact match won't work!
- We need to understand the properties of this shape and match the corresponding properties to identify the boomerang shape
- OpenCV provides a nice shape matcher function that we can use to achieve this
- The matching is based on the concept of Hu moment, which in turn is related to image moments
- The concept of "image moments" basically refers to the weighted and power-raised summation of the pixels within a shape

Contour analysis

- In this equation, p refers to the pixels inside the contour, w refers to the weights, N refers to the number of points inside the contour, k refers to the power, and I refers to the moment. Depending on the values we choose for w and k, we can extract different characteristics from that contour

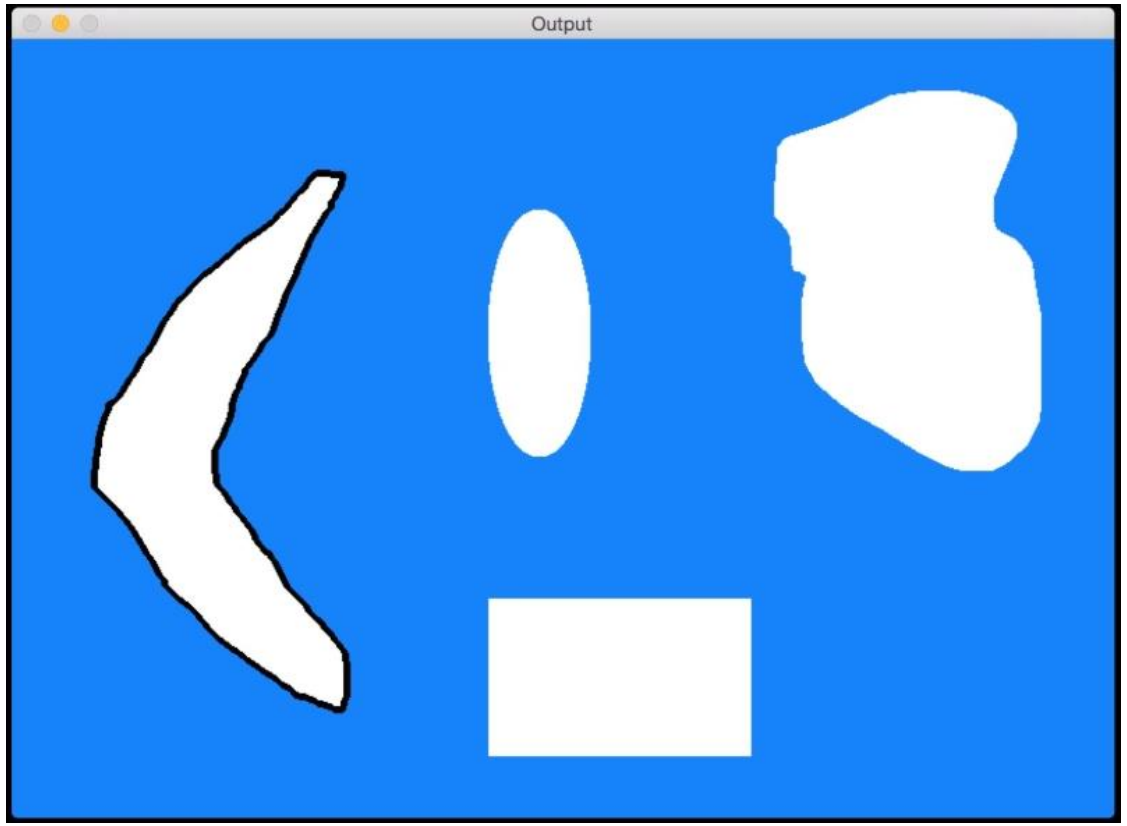
$$I = \sum_{i=0}^{N} w_i p_i^k$$

Contour analysis

- Perhaps the simplest example is to compute the area of the contour
- To do this, we need to count the number of pixels within that region
  - So mathematically speaking, in the weighted and power raised summation form, we just need to set w to 1 and k to 0
  - This will give us the area of the contour
- Depending on how we compute these moments, they will help us in understanding these different shapes
- This also gives rise to some interesting properties that help us in determining the shape similarity metric

Contour analysis

- If we match the shapes, you will see something like this:



OpenCV: cv2.findContours()

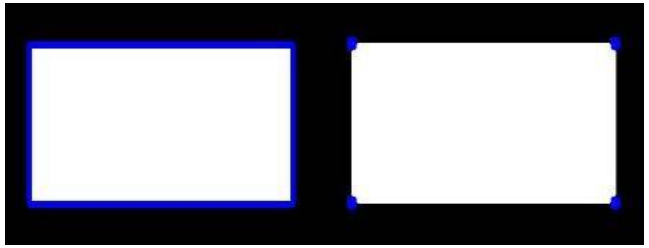
- Three arguments in cv2.findContours() function
  - first one is source image
  - second is contour retrieval mode (RETR_TREE - retrieves all of the contours and reconstructs a full hierarchy of nested contours, RETR_LIST - retrieves all of the contours without establishing any hierarchical relationships)
  - third is contour approximation method
- Outputs: the contours and hierarchy
  - contours is a Python list of all the contours in the image
  - each individual contour is a Numpy array of (x,y) coordinates of boundary points of the object

How to draw the contours?

- To draw the contours, cv2.drawContours function is used.
  - It can also be used to draw any shape provided you have its boundary points.
  - Its first argument is source image
  - second argument is the contours which should be passed as a Python list
  - third argument is index of contours (useful when drawing individual contour)
  - To draw all contours, pass -1 and remaining arguments are color, thickness etc.
- To draw all the contours in an image:
  - cv2.drawContours(img, contours, -1, (0,255,0), 3)
- To draw an individual contour, say 4th contour:
  - cv2.drawContours(img, contours, 3, (0,255,0), 3)
- But most of the time, below method will be useful:
  - cnt = contours[4]
  - cv2.drawContours(img, [cnt], 0, (0,255,0), 3)

Contour Approximation Method

- This is the third argument in cv2.findContours function. What does it denote actually?
- Above, we told that contours are the boundaries of a shape with same intensity. It stores the (x,y) coordinates of the boundary of a shape. But does it store all the coordinates? That is specified by this contour approximation method.
- If you pass cv2.CHAIN_APPROX_NONE, all the boundary points are stored. But actually do we need all the points? For eg, you found the contour of a straight line. Do you need all the points on the line to represent that line? No, we need just two end points of that line. This is what cv2.CHAIN_APPROX_SIMPLE does. It removes all redundant points and compresses the contour, thereby saving memory.
- Below image of a rectangle demonstrate this technique. Just draw a circle on all the coordinates in the contour array (drawn in blue color). First image shows points I got with cv2.CHAIN_APPROX_NONE (734 points) and second image shows the one with cv2.CHAIN_APPROX_SIMPLE (only 4 points). See, how much memory it saves!!!

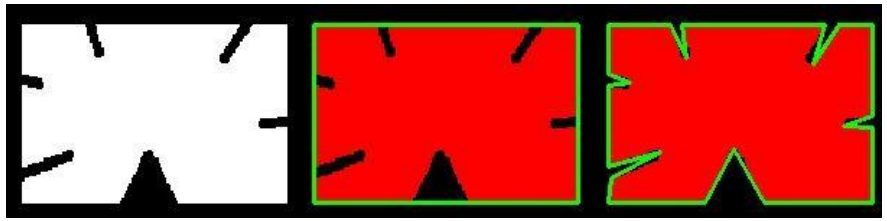


OpenCV: cv2.matchShapes()

- cv2.matchShapes(contour1, contour2, method, parameter)
  - object1 – First contour or grayscale image.
  - object2 – Second contour or grayscale image.
  - method – Comparison method: CV_CONTOURS_MATCH_I1 , CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3
  - parameter – Method-specific parameter (not supported now).
- The function compares two shapes. All three implemented methods use the Hu invariants

OpenCV: cv2.approxPolyDP()

- It approximates a contour shape to another shape with less number of vertices depending upon the precision we specify
- To understand this, suppose you are trying to find a square in an image, but due to some problems in the image, you didn't get a perfect square, but a "bad shape" (As shown in first image below). Now you can use this function to approximate the shape. In this, second argument is called epsilon, which is maximum distance from contour to approximated contour. It is an accuracy parameter. A wise selection of epsilon is needed to get the correct output.
  - epsilon = 0.1*cv2.arcLength(cnt,True)
  - approx = cv2.approxPolyDP(cnt,epsilon,True)
- Below, in second image, green line shows the approximated curve for epsilon = 10% of arc length. Third image shows the same for epsilon = 1% of the arc length. Third argument specifies whether curve is closed or not

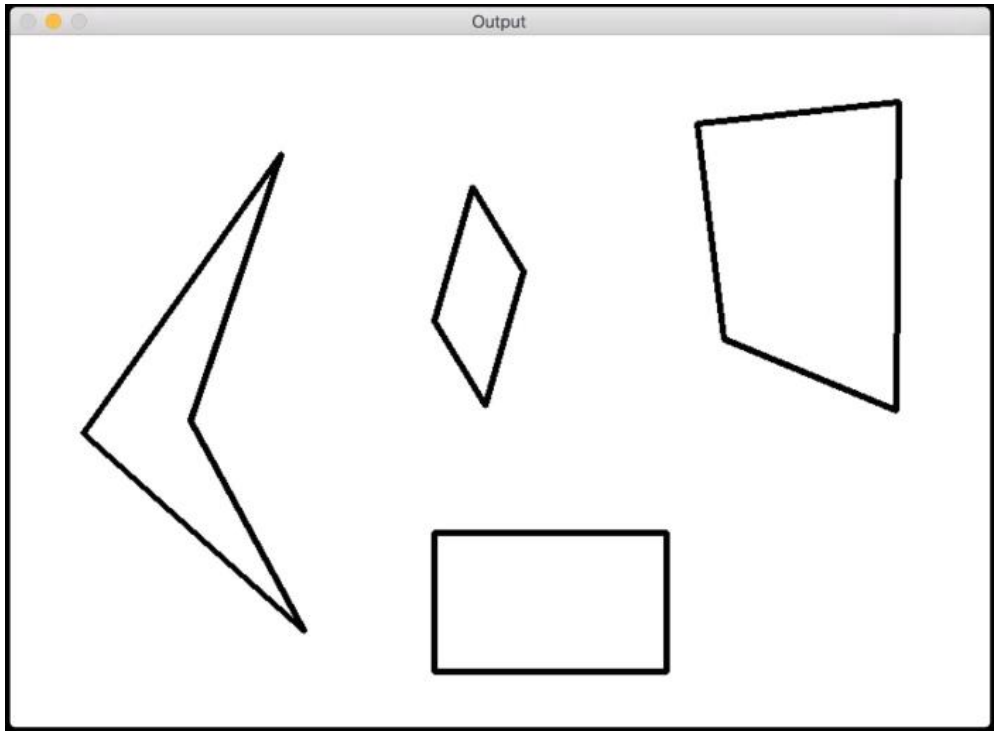


Approximating a contour

- A lot of contours that we encounter in real life are noisy
- This means that the contours don't look smooth, and hence our analysis takes a hit
- So how do we deal with this?
- One way to go about this would be to get all the points on the contour and then approximate it with a smooth polygon

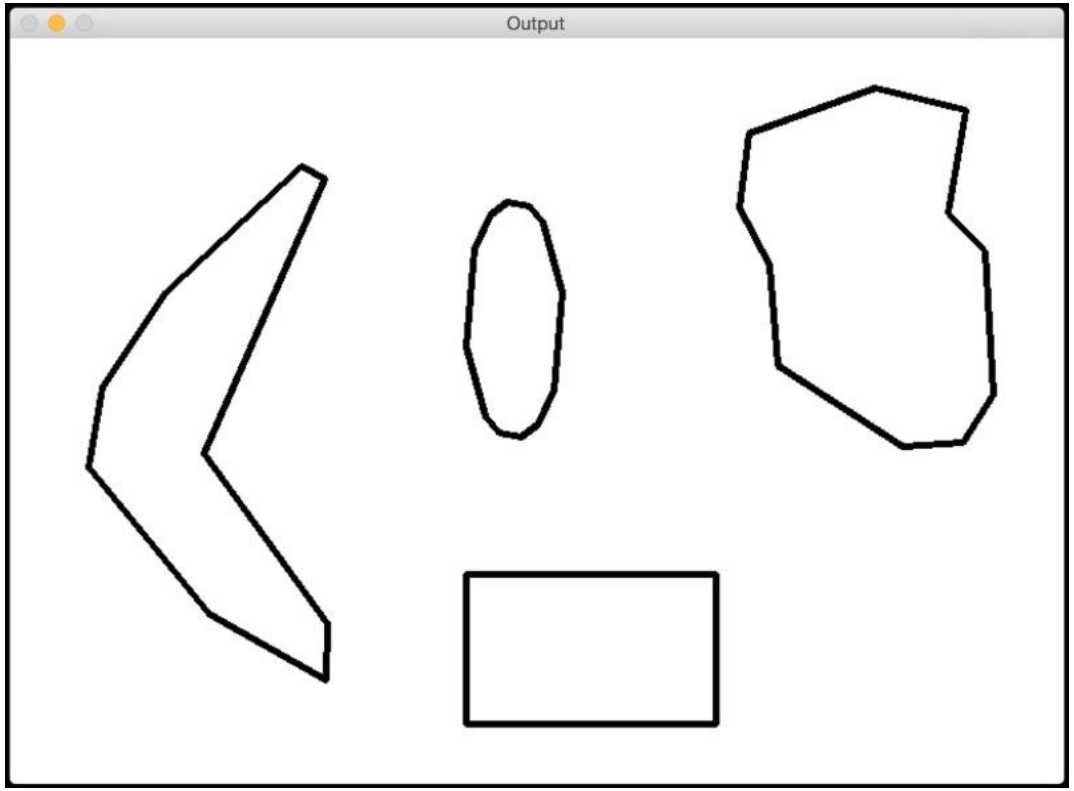
Approximating a contour

- Let's consider the boomerang image again. If you approximate the contours using various thresholds, you will see the contours changing their shapes. Let's start with a factor of 0.05:



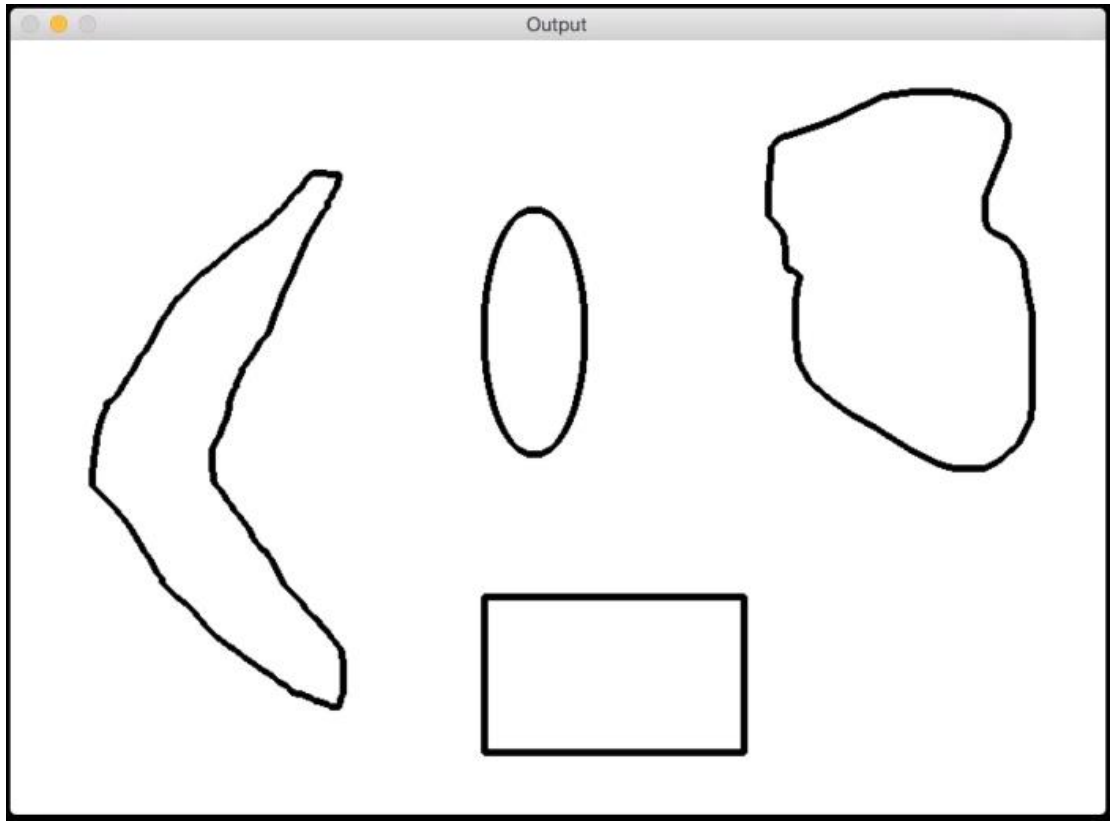
Approximating a contour

- If you reduce this factor, the contours will get smoother. Let's make it 0.01:



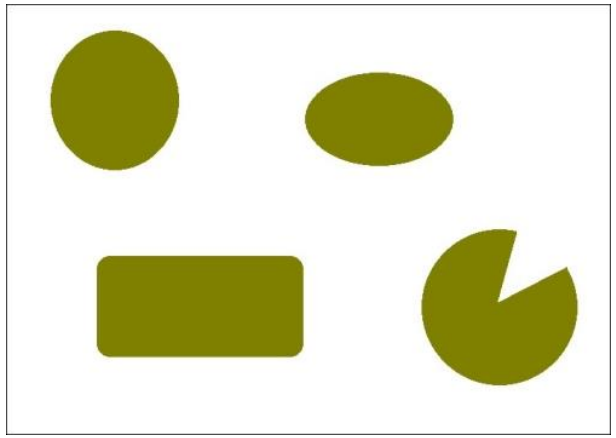
Approximating a contour

- If you make it really small, say 0.00001, then it will look like the original image:



Identifying the pizza with the slice taken out

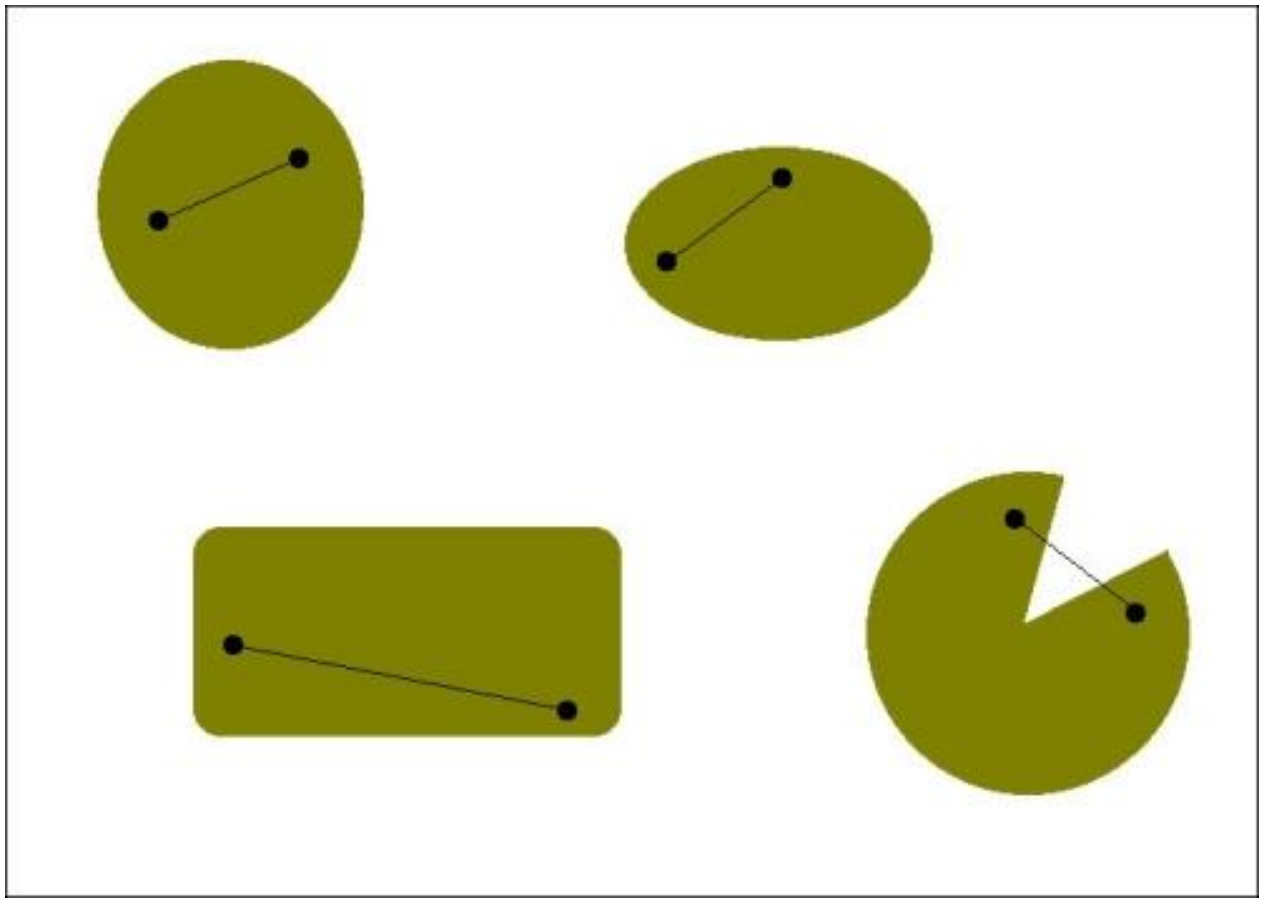
- We cannot take the approach we took earlier because we don't know what the shape looks like.
- So we don't have any template. We are not even sure what shape we are looking for, so we cannot build a template based on any prior information. All we know is the fact that a slice has been taken from one of the pizzas. Let's consider the following image:



Identifying the pizza with the slice taken out

- It's not exactly a real image, but you get the idea
- You know what shape we are talking about. Since we don't know what we are looking for, we need to use some of the properties of these shapes to identify the sliced pizza
- If you notice, all the other shapes are nicely closed. As in, you can take any two points within those shapes and draw a line between them, and that line will always lie within that shape. These kinds of shapes are called **convex shapes**
- If you look at the sliced pizza shape, we can choose two points such that the line between them goes outside the shape as shown in the figure that follows:

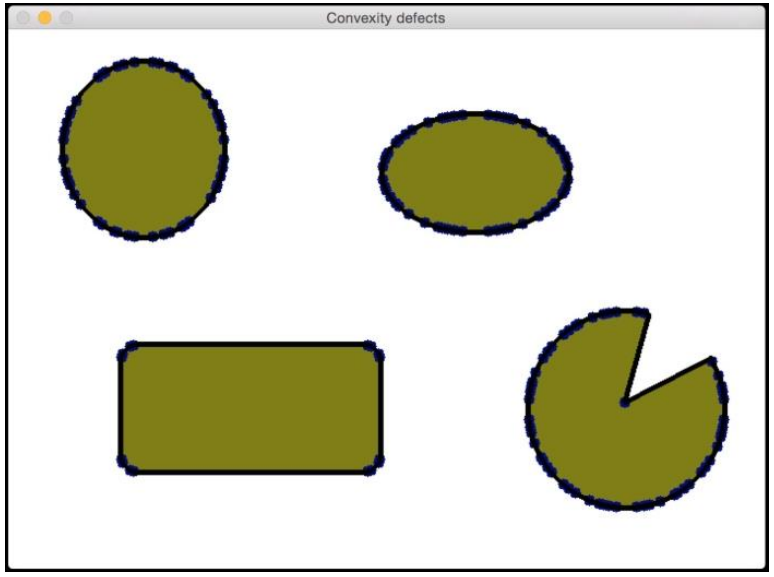
Identifying the pizza with the slice taken out



So, all we need to do is detect the non-convex shape in the image and we'll be done!

Identifying the pizza with the slice taken out

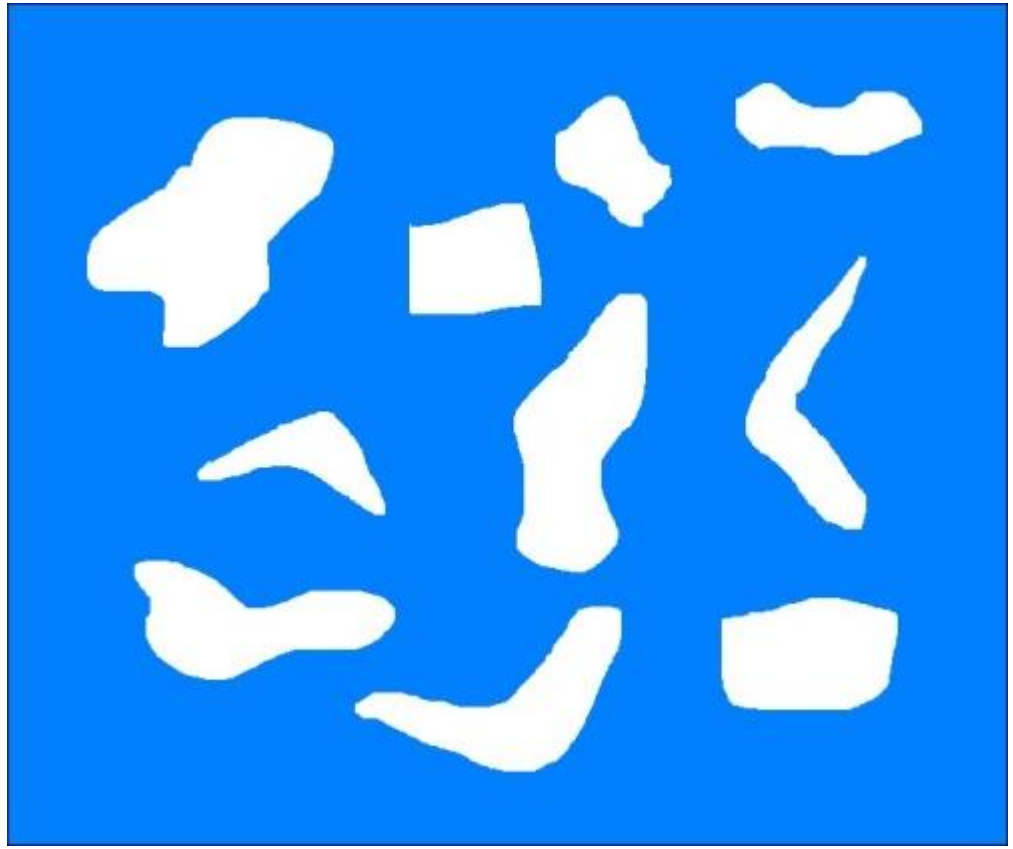
- The curves are not really smooth. If you observe closely, there are tiny ridges everywhere along the curves. So, if you just run your convexity detector, it's not going to work. This is where contour approximation comes in really handy. Once we've detected the contours, we need to smoothen them so that the ridges do not affect them.



How to censor a shape?

- Let's say you are dealing with images and you want to block out a particular shape.
- Now, you might say that you will use shape matching to identify the shape and then just block it out, right?
- But the problem here is that we don't have any template available. So, how do we go about doing this?
- Shape analysis comes in various forms, and we need to build our algorithm depending on the situation. Let's consider the following figure:

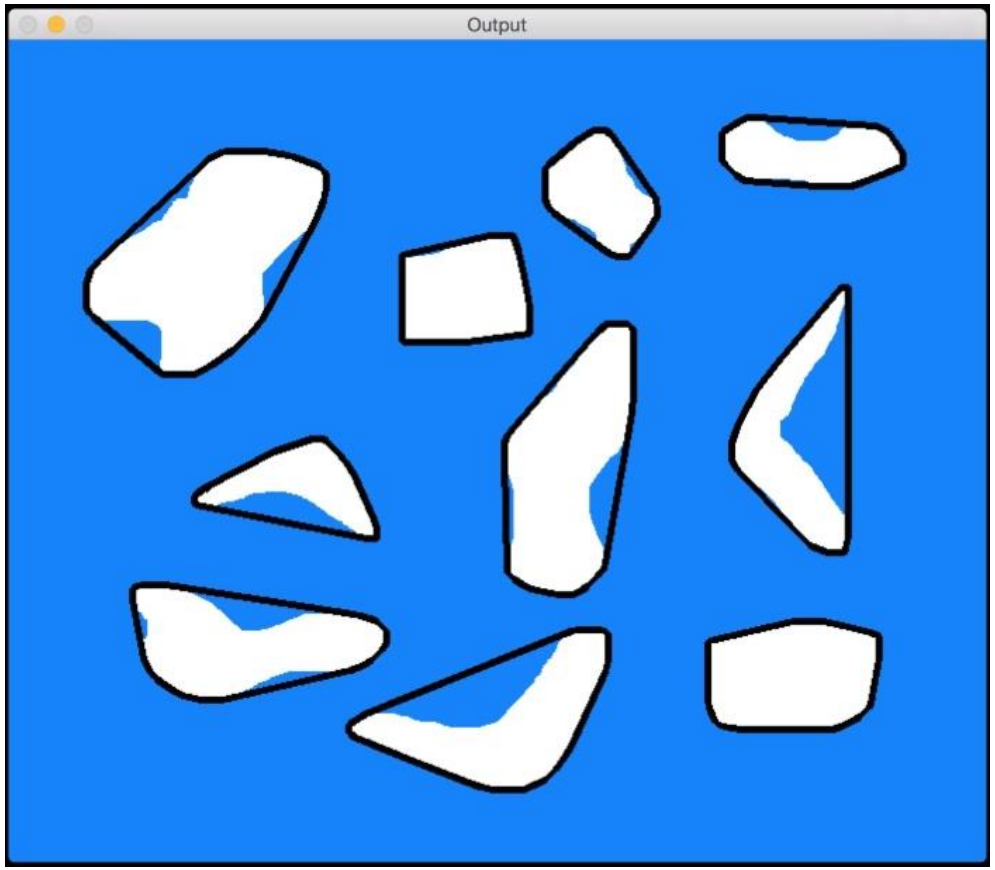
How to censor a shape?



How to censor a shape?

- Let's say we want to identify all the boomerang shapes and then block them out without using any template images
- As you can see, there are various other weird shapes in that image and the boomerang shapes are not really smooth
- We need to identify the property that's going to differentiate the boomerang shape from the other shapes present
- Let's consider the convex hull. If you take the ratio of the area of each shape to the area of the convex hull, we can see that this can be a distinguishing metric
  - This metric is called solidity factor in shape analysis. This metric will have a lower value for the boomerang shapes because of the empty area that will be left out, as shown in the following figure:

How to censor a shape?



How to censor a shape?

- The black boundaries represent the convex hulls
- Once we compute these values for all the shapes, how do separate them out? Can we just use a fixed threshold to detect the boomerang shapes? Not really! We cannot have a fixed threshold value because you never know what kind of shape you might encounter later
- So, a better approach would be to use **K-Means clustering**. K-Means is an unsupervised learning technique that can be used to separate out the input data into K classes

How to censor a shape?

- We know that we want to separate the shapes into two groups, that is, boomerang shapes and other shapes.
  - So, we know what our *K* will be in K-Means
- Once we use that and cluster the values, we pick the cluster with the lowest solidity factor and that will give us our boomerang shapes.
- Bear in mind that this approach works only in this particular case. If you are dealing with other kinds of shapes, then you will have to use some other metrics to make sure that the shape detection works. As we discussed earlier, it depends heavily on the situation. If you detect the shapes and block them out, it will look like this:

How to censor a shape?