

*Vehicle Logo Recognition System*

| **ID** | **Student Name** | **E-mail** |
| --- | --- | --- |
| **196280** | Ashraf Adel | ashraf196280@bue.edu.eg |

Table Of Contents

[**Overview**](#_heading=h.2et92p0) **3**

[**Mapping of User-Defined Functions**](#_heading=h.tyjcwt) **4**

[**Steps**](#_heading=h.3dy6vkm) **4**

[Step 0: Setting up the environment](#_heading=h.1t3h5sf) 4

[getImgs()](#_heading=h.4d34og8) 7

[getImgsLogosGrayed()](#_heading=h.2s8eyo1) 8

[getLogoNames()](#_heading=h.17dp8vu) 8

[imgToLogo()](#_heading=h.3rdcrjn) 10

[Step 1: Grayscale the image then median filter it](#_heading=h.26in1rg) 12

[displayStep()](#_heading=h.lnxbz9) 13

[Step 2: Use canny edge detection](#_heading=h.35nkun2) 13

[Step 3: Close image to connect edges](#_heading=h.1ksv4uv) 14

[Step 4: Remove largest component](#_heading=h.44sinio) 14

[Step 5: Get circle/rectangle filled components](#_heading=h.2jxsxqh) 17

[Step 6: Open image to to get rectangles/small circles](#_heading=h.z337ya) 17

[Step 7: Choose component that resemebles a car license plate](#_heading=h.3j2qqm3) 18

[bestPlateCandidates()](#_heading=h.1y810tw) 19

[Step 8: Circular-mask the image around the area that possibly contains the license plate](#_heading=h.4i7ojhp) 21

[Step 9: choose best subimage around the masked area and return it and the logo’s corresponding name](#_heading=h.2xcytpi) 22

[bestSubImage()](#_heading=h.1ci93xb) 23

[maskAndCropAndZoom()](#_heading=h.3whwml4) 26

[compareImg()](#_heading=h.2bn6wsx) 28

[getFeatures()](#_heading=h.3as4poj) 30

[Program’s Outputs](#_heading=h.49x2ik5) **31**

[Option 1](#_heading=h.2p2csry) 31

[Option 2](#_heading=h.147n2zr) 31

[Option 3](#_heading=h.3o7alnk) 33

[Option 4](#_heading=h.23ckvvd) 33

# 

# Overview

The program presented in this report does the following:

* Given an image of a car, extract a subimage containing the car’s logo and its name (label).
* Assumption made in this program: if the image contains multiple cars, the subimage returned will be related to only one car (mainly the one closer to the center of the image).

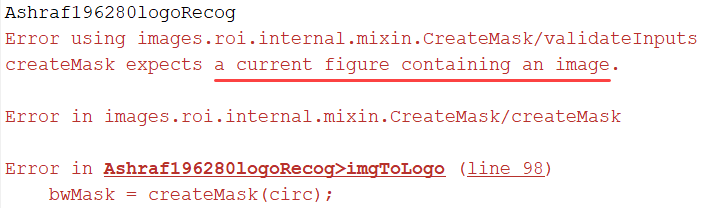
The program’s logical sequence is as follows:

1. Smooth the image
2. Extract edges
3. Modify the edges
4. Extract components
5. Extract only the component similar to a car’s license plate
6. Extract subimages around that component
7. Test and score each subimage with all logos locally stored
8. Display subimage with best score (least differences) along with its logo

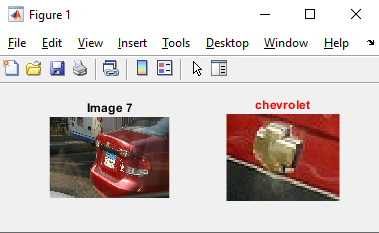
This sequence will be thoroughly explained in steps 1-9 which are mentioned later in this report.

Note that most of the code will be visualized on this example:



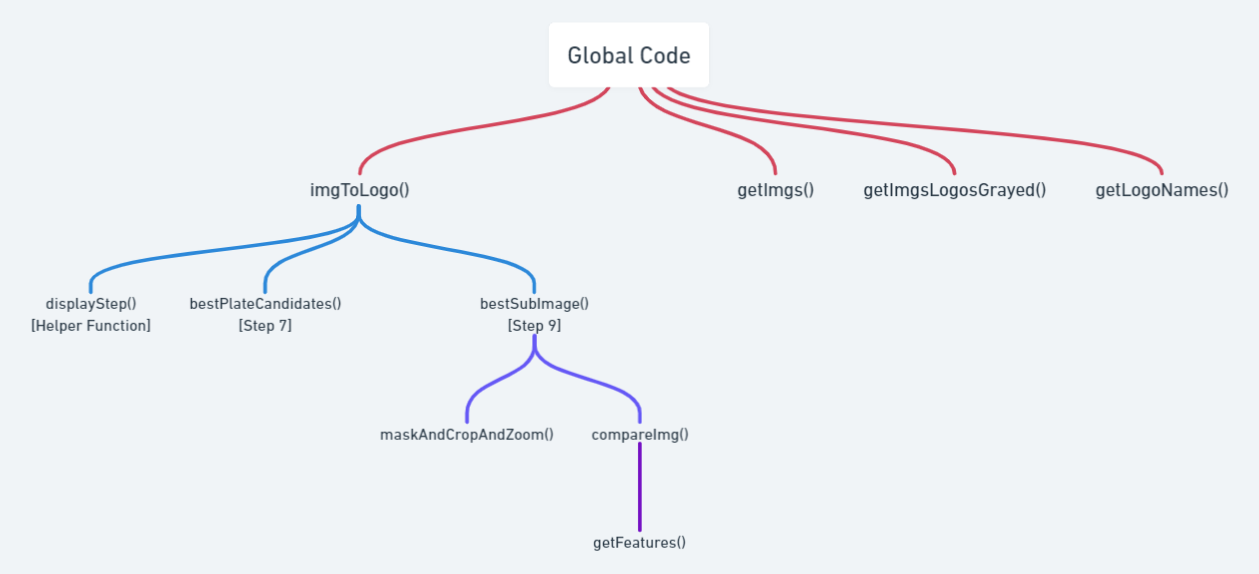
Important notice: In step 8: drawcirlcle() and createMask() require a figure to be displayed to create a circle:  


∴ The output of the program must display the original image before these functions are called, so an example of the program’s output:



# Mapping of User-Defined Functions

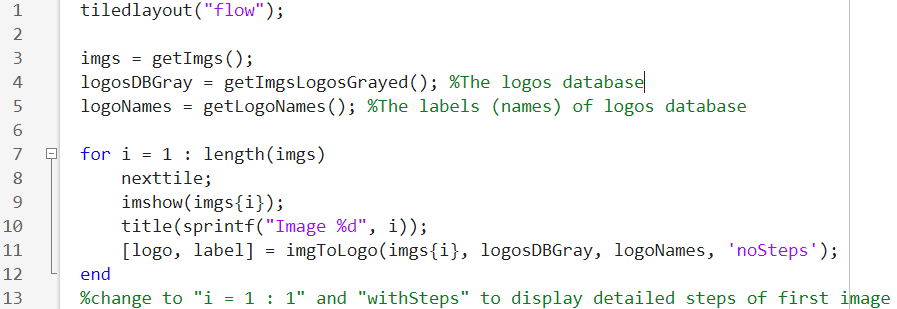
The following is a mind map of the user-defined functions used in this program:

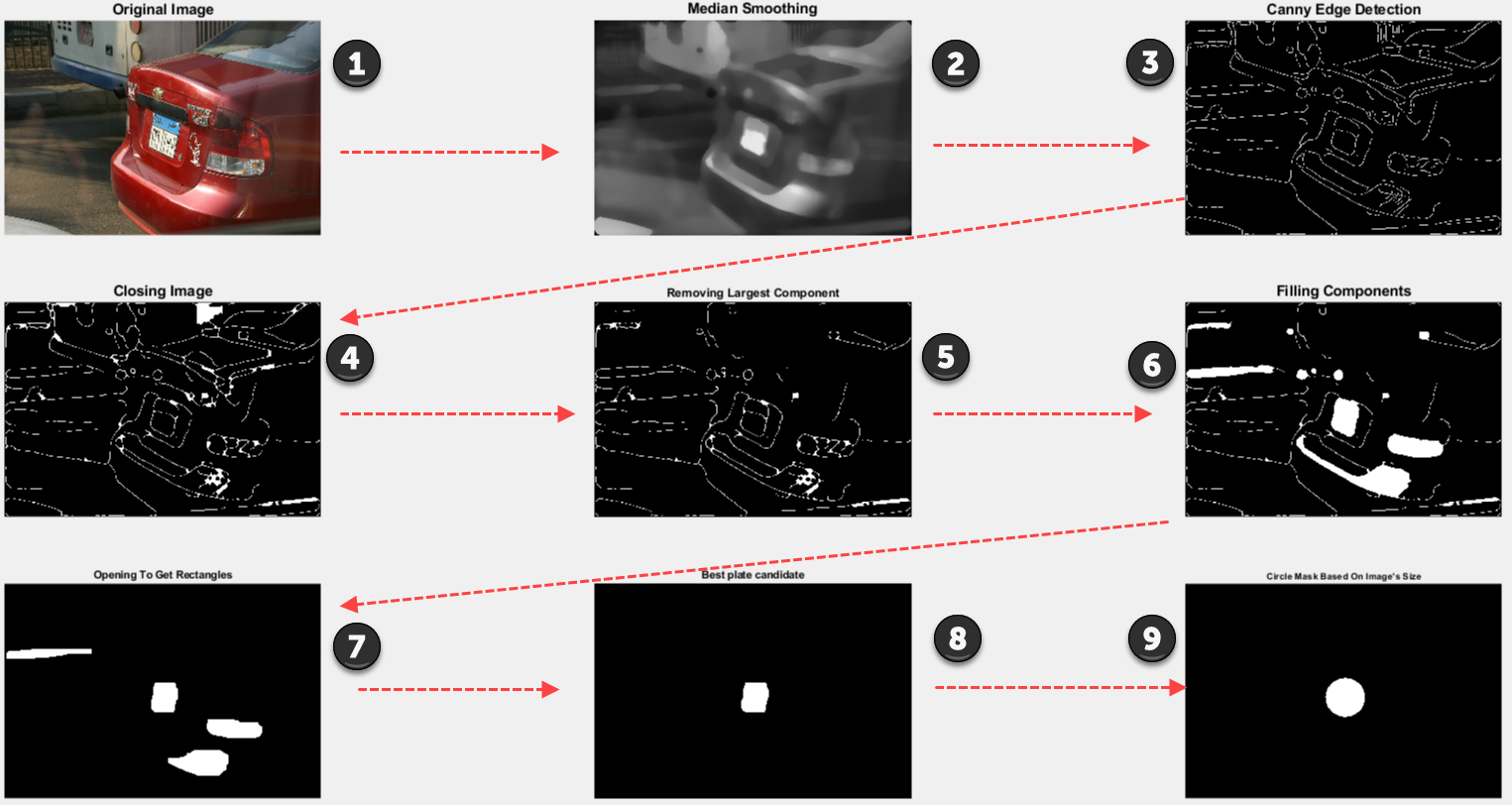
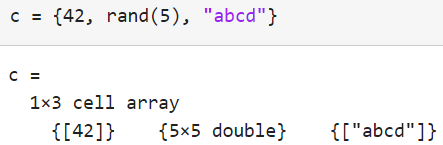
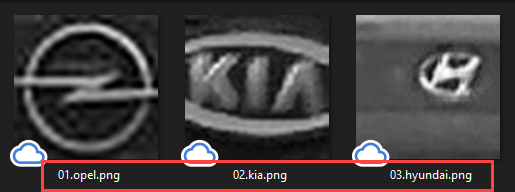
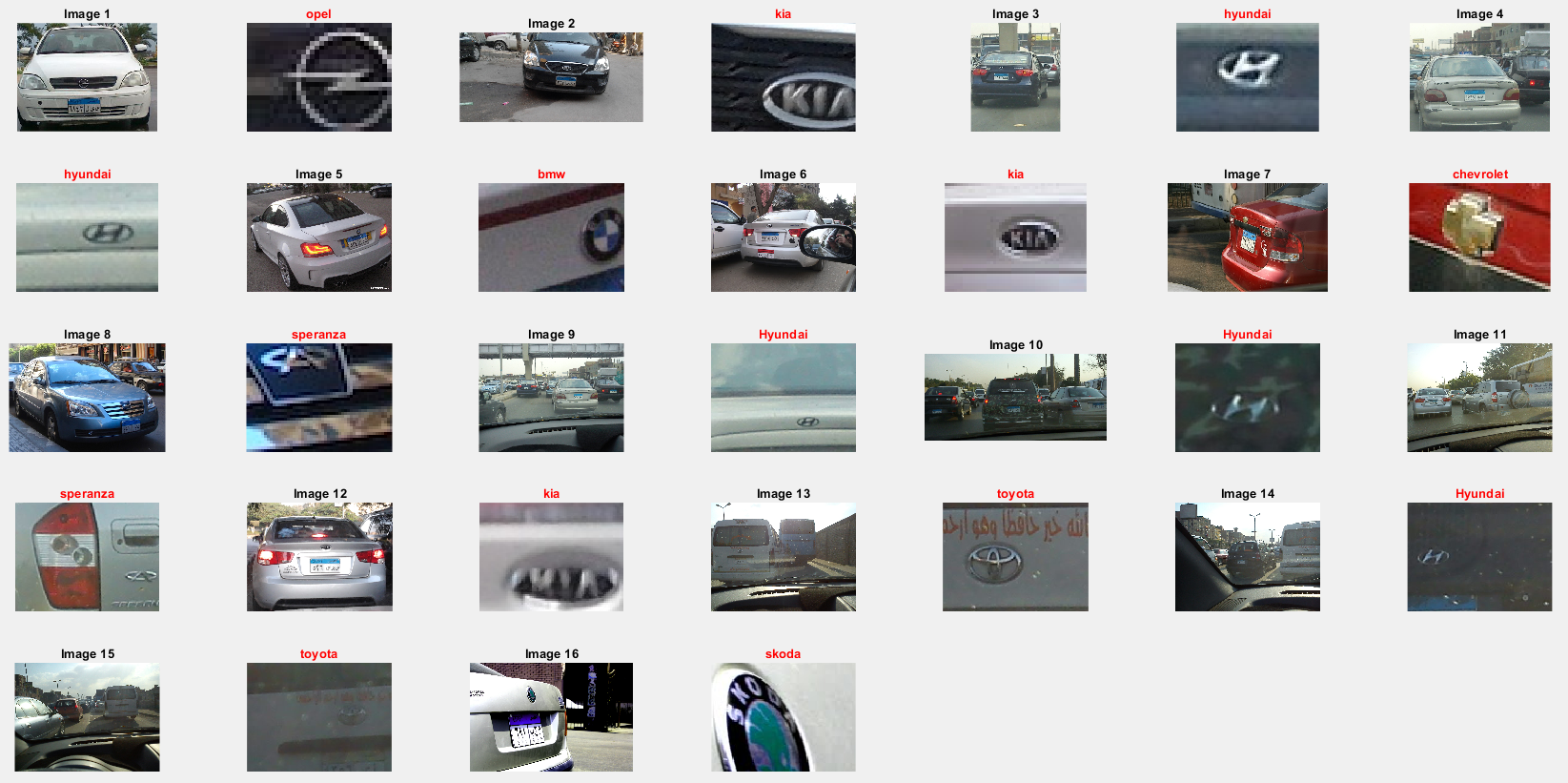
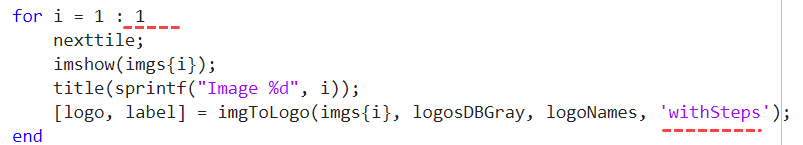
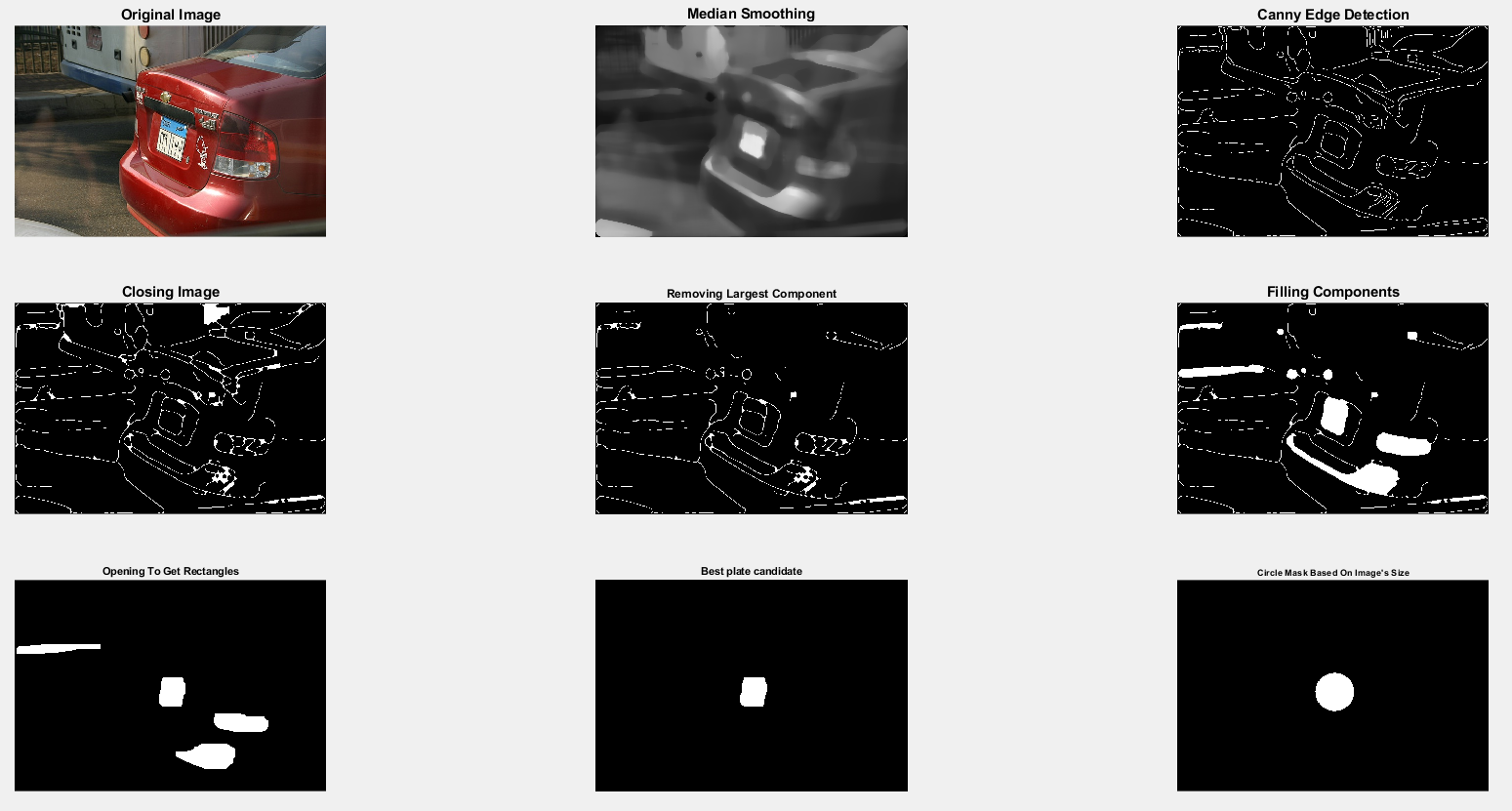
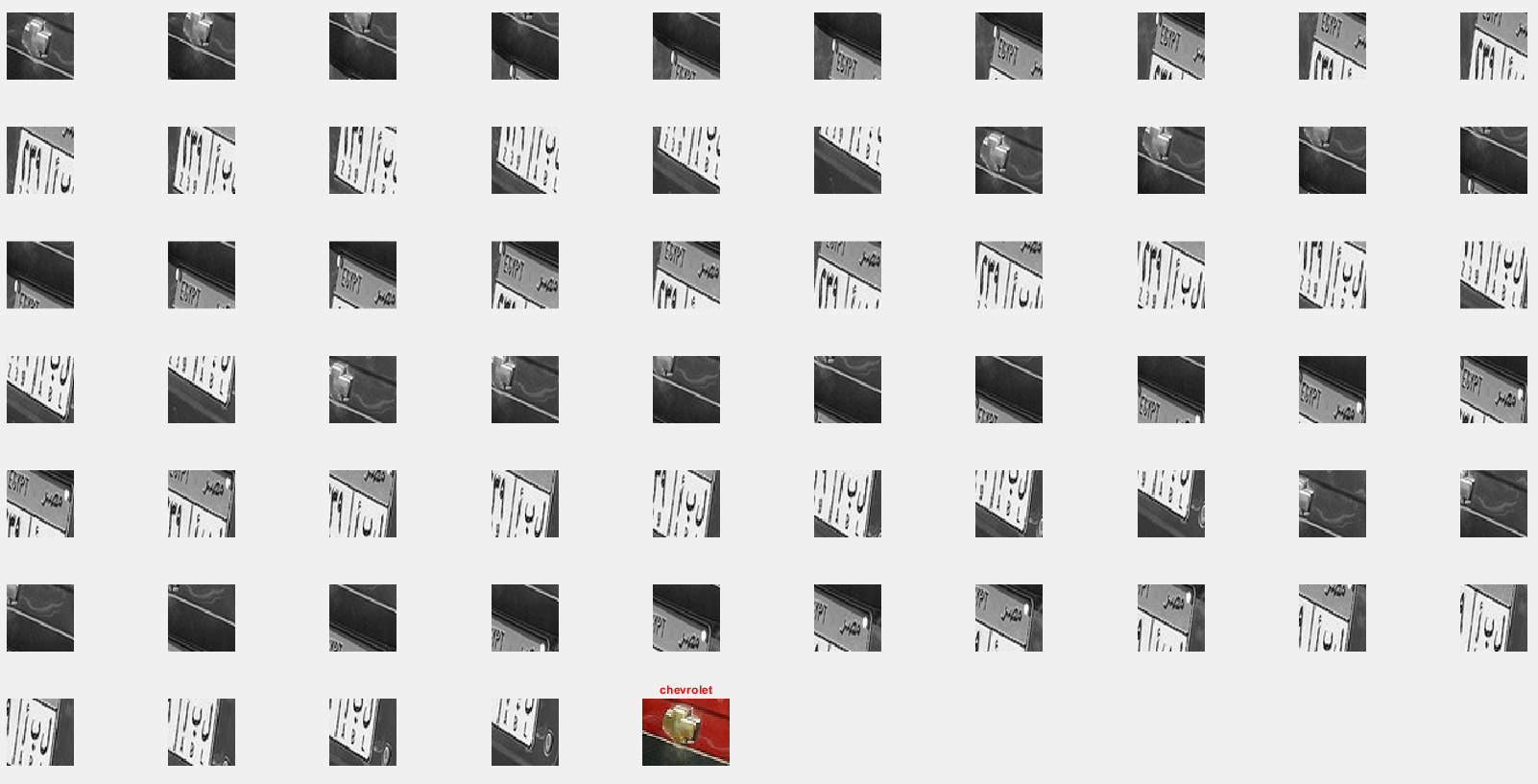


The rest are built in functions that will be explained later in this report.

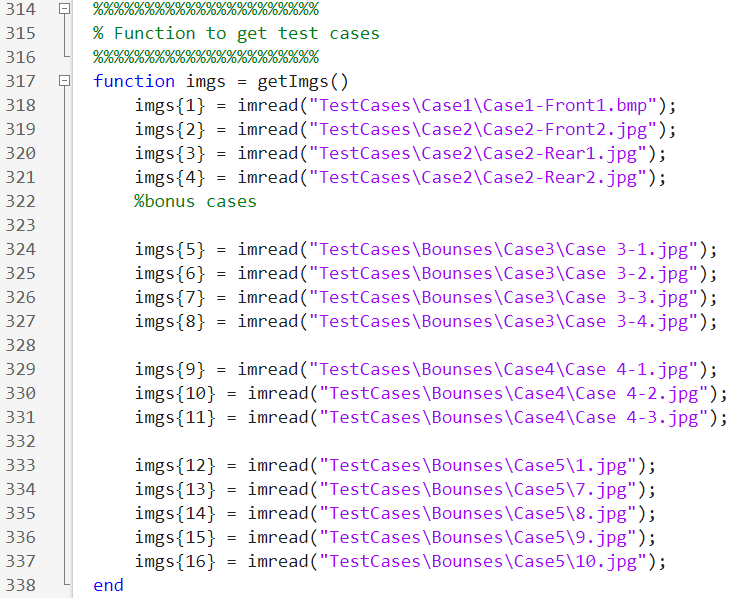
# Steps

## Step 0: Setting up the environment

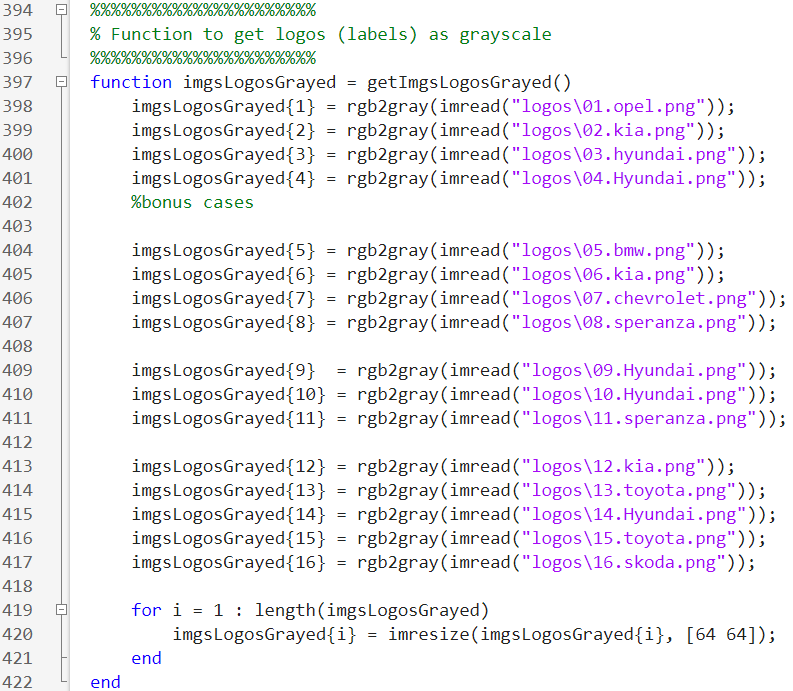


* tiledlayout(“flow”) allows images to appear in a flow manner (from left to right, then top to bottom) instead of manually changing the size of the subplots, for example:  
  
* getImgs() gets the 16 images in a cell array in the folder “TestCases” stored locally
  + A cell array is an array of data that could have different data types
  + Example:  
     
* getImgsLogosGrayed() gets the logos stored locally in “Logos” folder and converts these images to grayscale
* getLogoNames() gets the file names of each image in the “Logos” folder; these file names are the names of the car logos (labels). Example:  
  
* The for loop displays the original image, and then a subimage and label of the logo using the imgToLogo() function.
  + When “noSteps” are displayed:  
    
  + When the images are with steps:  
      
    (instead of length(imgs) → to avoid image cluttering):  
      
    
  + Note: sprintf() is used to format data (e.g. “i” variable) into a string to be displayed as title of an image

### getImgs()

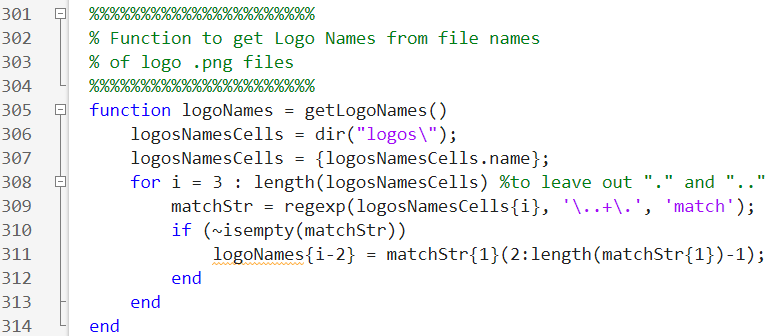


### getImgsLogosGrayed()

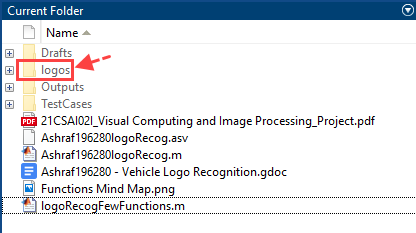


Note that at the end, the images are resized to [64 64], the reason for that is explained in compareImg()

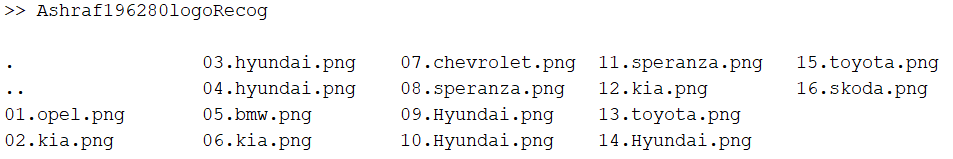
### getLogoNames()



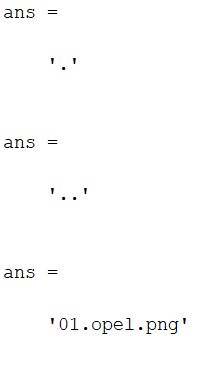
dir(“logos\”) retrieves the terminal’s output of the logos folder:



output:



logosNamesCells.name retrieves names of each output individually. Part of the output:



{logosNamesCells.name} puts these outputs to a cell array:



regexp() written above will only match the dots before and after the logo’s name along with the logo’s name:



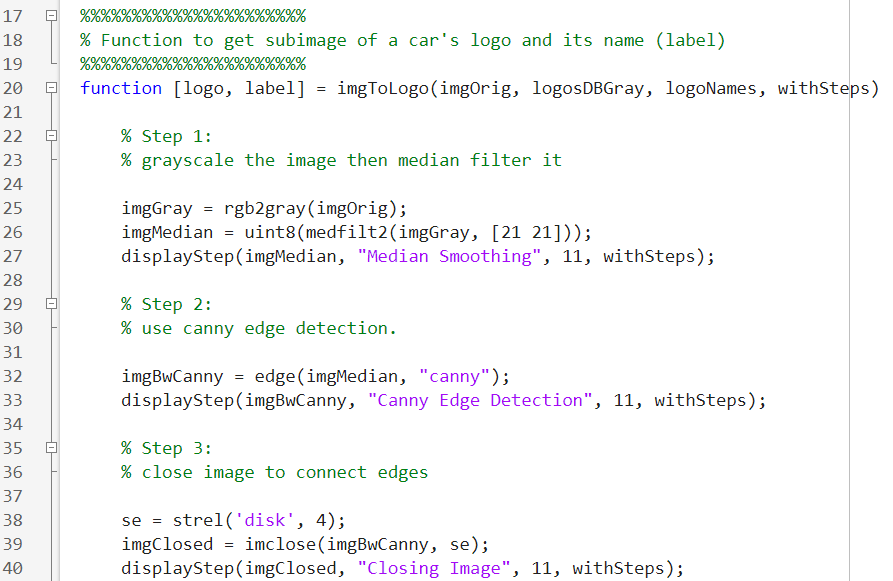
Since regexp returns a cell array containing the stringthat it matched it with (for example: {“.opel.”}, ∴ the line matchStr{1}(2:length(matchStr{1})-1) does the following:

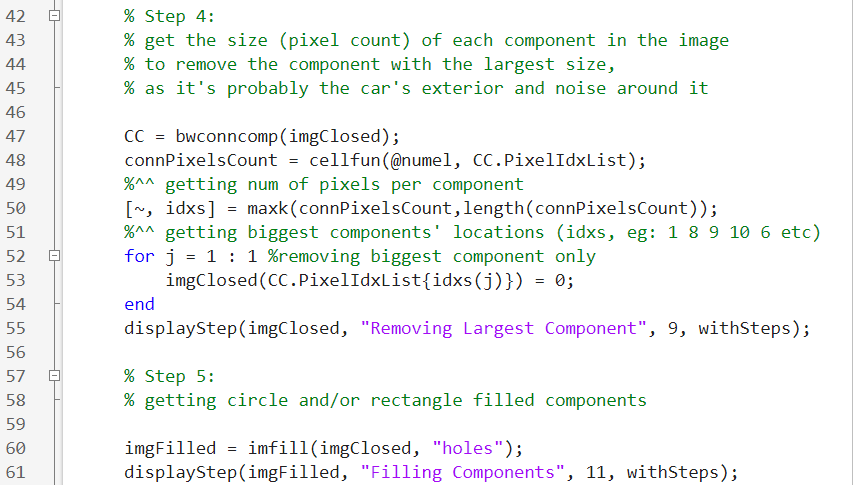
* matchStr{1} → “.opel.”
* matchStr{1}(2:length(matchStr{1})-1) → matchStr{1}(2:5) → “opel”

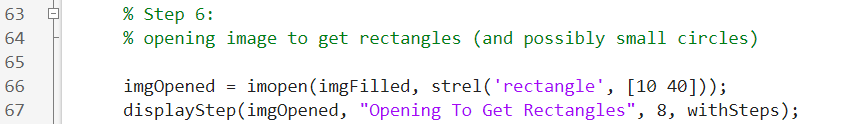
∵ the expression above is stored in logoNames{i-2}

∴ “logoNames” is a cell array containing the names of the car logos.

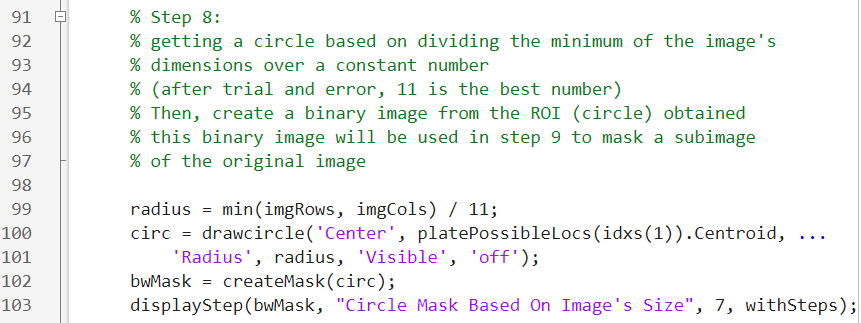
### imgToLogo()

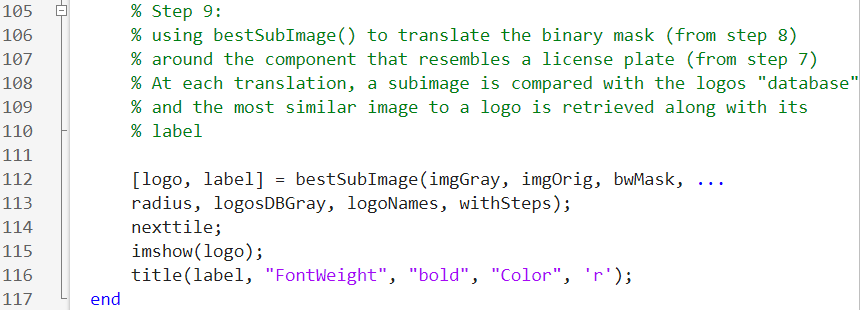






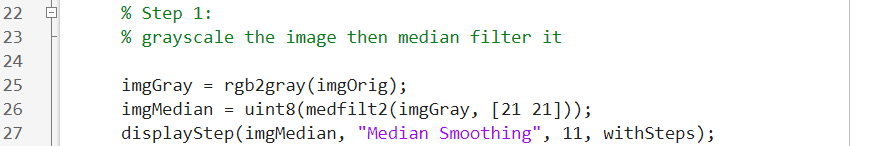






Each of the steps are explained in the following section.

## Step 1: Grayscale the image then median filter it

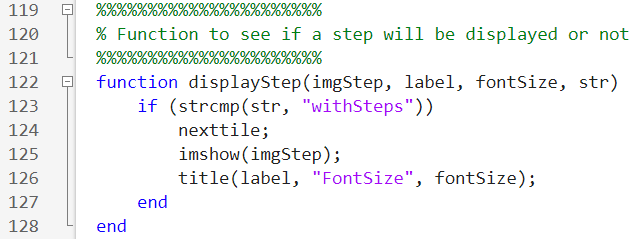


* After trial and error, [21 21] is the best grid size to filter the image.
* uint8() is written to avoid precision errors.
* Whenever displayStep() is written, this means that the current step could be displayed on the current figure (tiled layout “flow”) if the final argument of imgToLogo was the string “withSteps”

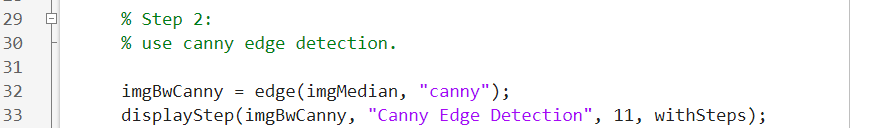
Example of output:



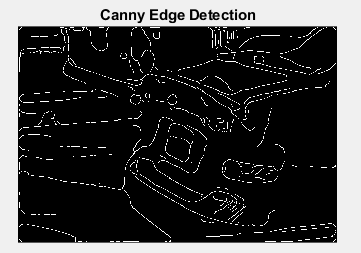
### displayStep()



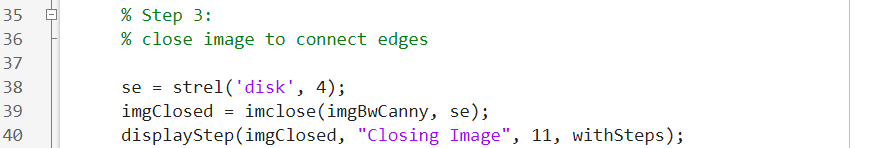
## Step 2: Use canny edge detection



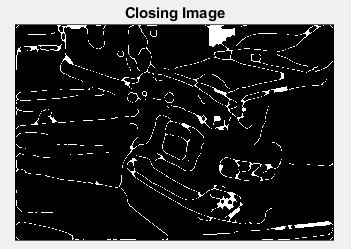
Example of output:



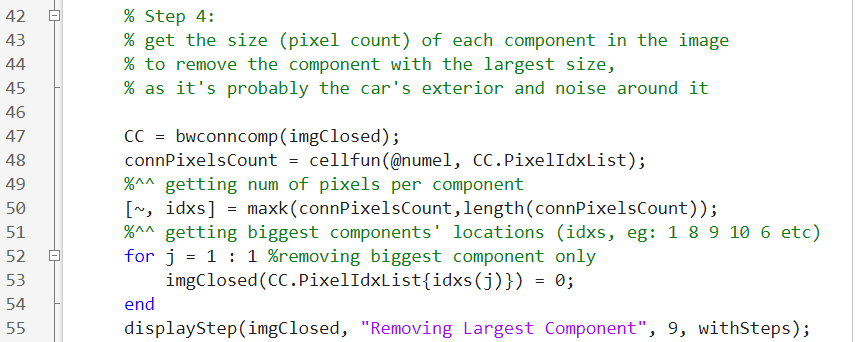
## Step 3: Close image to connect edges

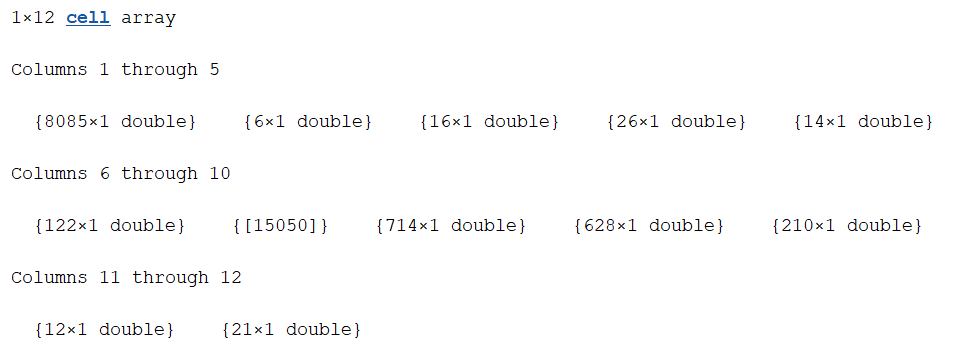
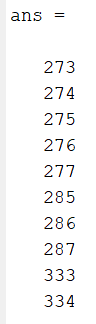
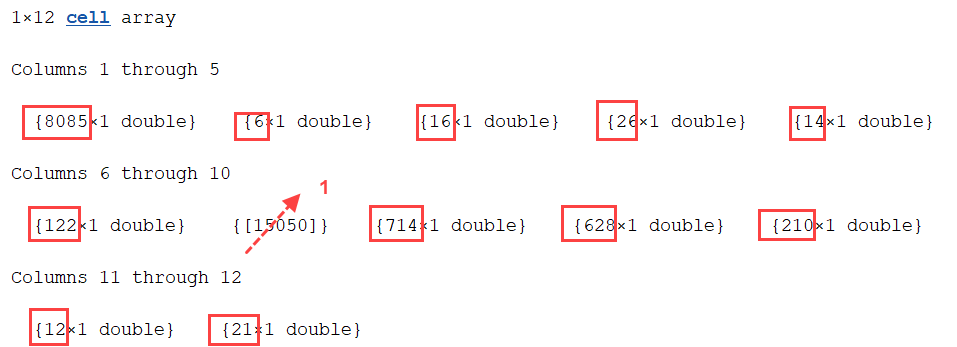
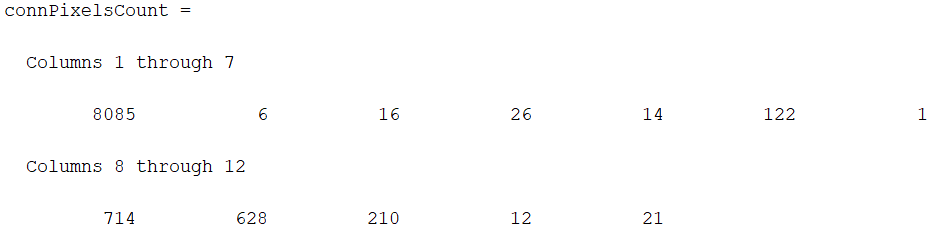
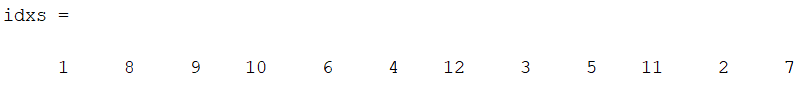
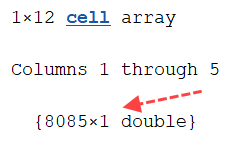


The value “4” is obtained after trial and error:

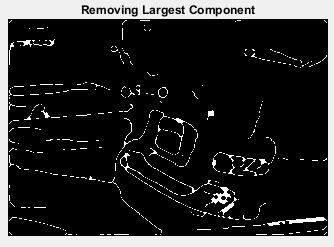


## Step 4: Remove largest component

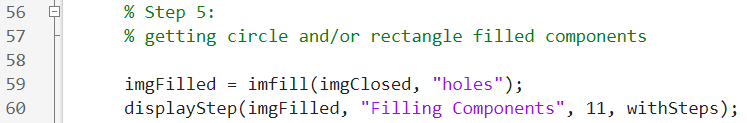


* bwconncomp() returns the objects (components/ROIs) in the image in a cell array where each object has some properties including “PixelIdxList” which contains the linear indices of the pixels in the image (i.e. vector of pixel locations of that object in the image).  
  Example of output of CC.PixelIdxList:  
    
  Example of output of CC.PixelIdxList{1}(1:10) returns the first 10 pixel locations of component 1:  
  
* cellfun(@numel, CC.PixelIdxList) does the operation of the first argument (numel function) on each element in the second argument and returns the result for each element.
  + @ is a function handle which basically tells matlab that the following variable is a function. In short, it is usually used to pass a function in another function
  + numel() is a built-in matlab function that counts the number of elements in a data structure (cell array, vector, etc). Therefore, when used on CC.PixelIdxList, it returns the following numbers:  
      
    Proof: Example output of connPixelsCount:  
    
* maxk(arr, k) gets the maximum k elements in arr, where here, arr is connPixelsCount and k is the number of components (12 in the example above), so it is similar to just sorting connPixelsCount decendingly, but this is done to also retrieve the indices of the components that are larger in size. For example, the output of the example above:  
    
  Which is true because for example, 8085 (idx = 1) is the largest component.
* In the for loop, imgClosed(vec) = 0 means that that we’re setting the locations of vector “vec” in image “imgClosed” to 0, so here, the vector is CC.PixelIdxList{idxs(j)} = 0 which is this vector:  
  
  + Note that a for loop could be avoided and “j” could be written as “1”, but the for loop is left in case of further future modifications

Example output of this step:

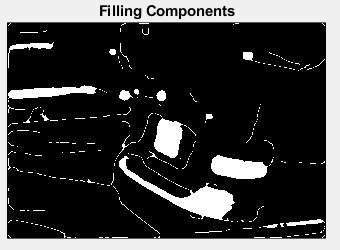


## Step 5: Get circle/rectangle filled components

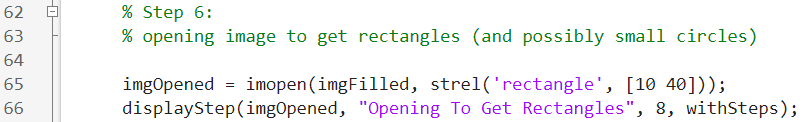


* If “holes” argument is not specified, you will have to interact with the image and click on the areas that you want to fill, which is not what we want, so we add “holes” argument.

Example output of this step:

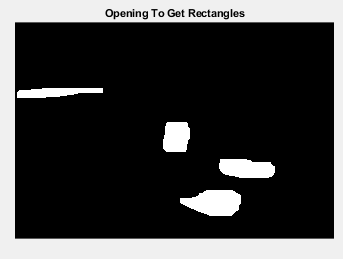


## Step 6: Open image to to get rectangles/small circles



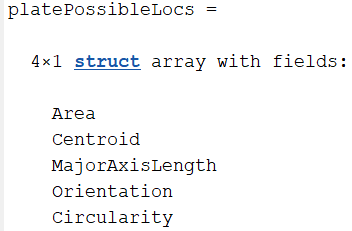
* The numbers [10 40] are obtained after trial and error

Example output of this step:

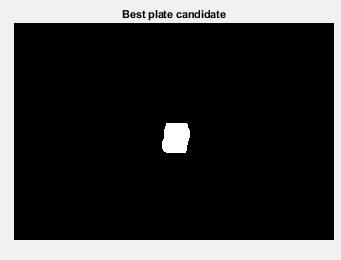


## Step 7: Choose component that resemebles a car license plate

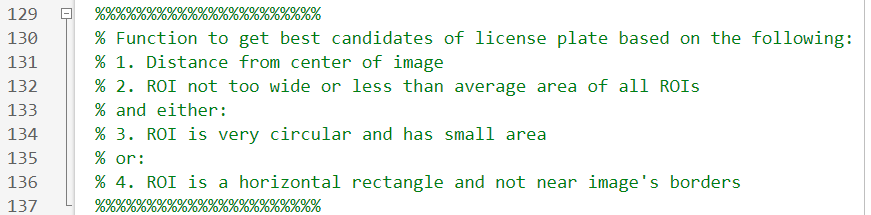


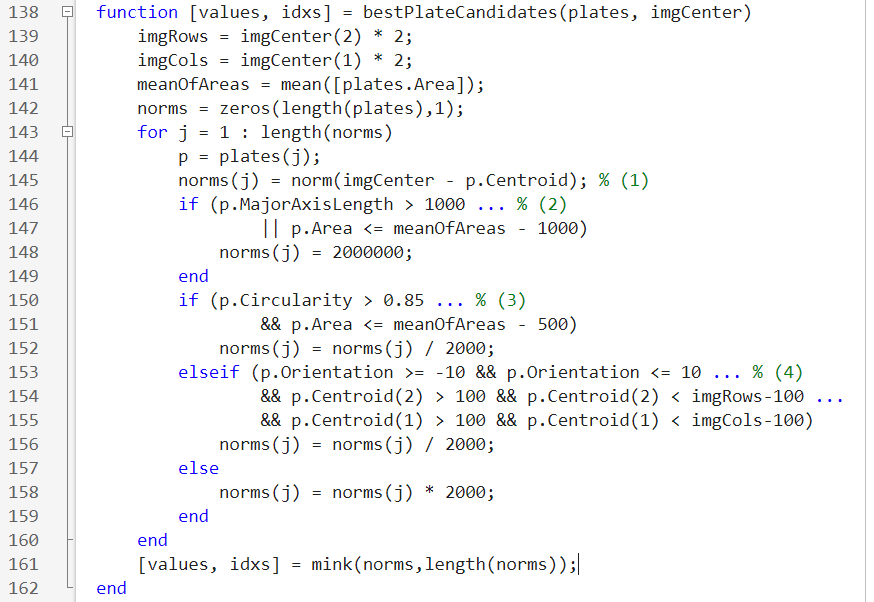
* regionprops() gets the properties stated in the arguments for each component:  
  
  + Area: number of pixels in each component
  + Centroid: [x-coordinate y-coordinate] (equivalent to [columnNum rowNum]) of the center of a component with respect to the image
  + MajorAxisLength: Length of the major axis of the ellipse that encompasses the component
  + Orientation: angle between the general direction of the component and the the horizontal line
  + Circularity: how much (from 0 to 1) is the component similar to a circle
* imgCenterPoint is obtained in the format [x-coordinate y-coordinate] to be able to subtract it from the component’s centroid in bestPlateCandidates() function
* bestPlateCandidates() returns index of component that is most likely a license plate in as first index of “idxs” vector (i.e. idxs(1))
* The for loop removes all components except the one that is most likely a license plate

Example of Output Image:

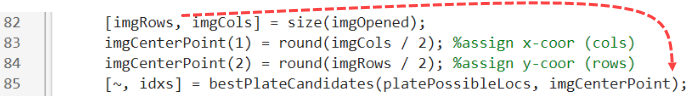
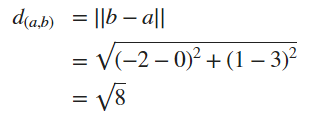


### bestPlateCandidates()

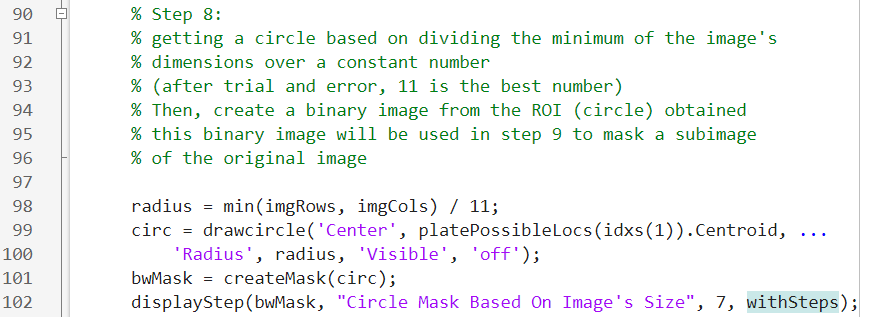




The description of the function is stated in the comments at the top of the function.

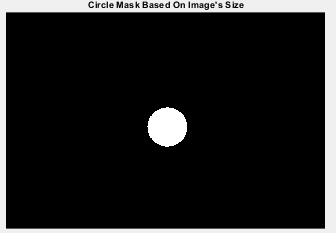
* imgRows and imgRows are the dimensions of the input image that are obtained by doing the reverse algebraic operations of this:  
    
  It could also be obtained by passing the imgRows and imgCols as arguments here:  
  
* [plates.Area] returns all the components’ areas in a vector which is passed in mean() to get the average of all the areas and assign it in meanOfAreas
* Initializing norms vector with zeros for performance efficiency
* p = plates(j) is just to write the shorthand variable “p” in the rest of the for loop
* norm() is a built-in function that gets the euclidean distance between two points (with the format [x-coordintae y-coordinate]
  + Example, if imgCenter is (0, 3) and p.Centroid is (-2, 1), then the euclidean distance is calculated like this:  
    
  + ∴ In essence, we’re seeing how far each component is from the image’s center
* The first if condition excludes components that are very wide or that are much less than the average size (area) of components. This exclusion happens by assigning that component a very high value to avoid it being selected as the least distance component (which happens at line 161)
* The second if condition checks if the component is very circular and small, then automatically assign it a very small value (i.e. Imagine shifting it to the image’s center  
  To make it a better candidate for selection). This assumption is made because a component with this shape is most likely the logo itself. Therefore, this will be a better candidate than the license plate
* The third condition is executed if the second one didn’t execute. It also puts horizontal components that are not around 100 pixels from the image’s boundaries as good license plate candidates
  + The boundaries condition is done because it is safe to assume that the license plate won’t be at the edge of the image, so any components obtained at that part are probably noise
* If all the previous conditions weren’t satisfied, increase the distance because this is probably not a component that resembles a license plate or a logo
* After getting all the distances (more accurately, scores), we return the best (least) scores along with their indices using mink(). The indices are what we’re interested in, as they’re used in line 87 as previously explained

## Step 8: Circular-mask the image around the area that possibly contains the license plate

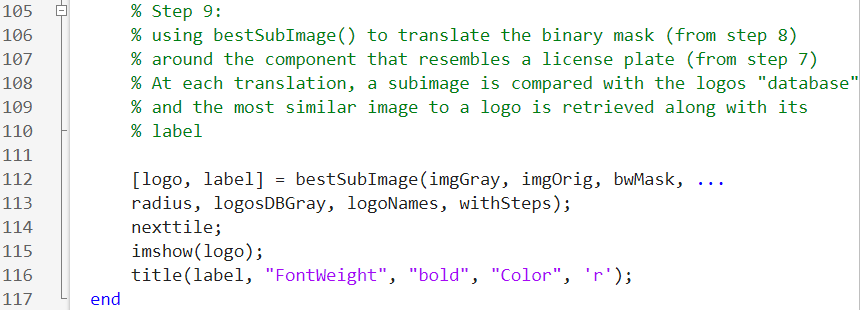


* drawcircle() is a built-in function that creates a circle based on its center point location on the image, and its radius. It returns a circle ROI object
  + The ‘Visible’ argument is to prevent a blue circle from being overlayed on the original image that is being displayed. Example:  
    
* createMask() creates a binary image where the “on” pixels are based on the passed ROI object (in our case, “circ”)

Example of Output Image:

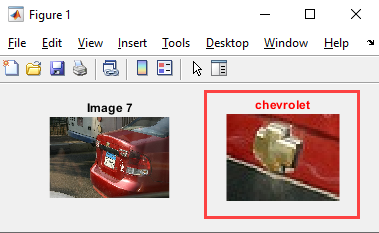


## Step 9: choose best subimage around the masked area and return it and the logo’s corresponding name

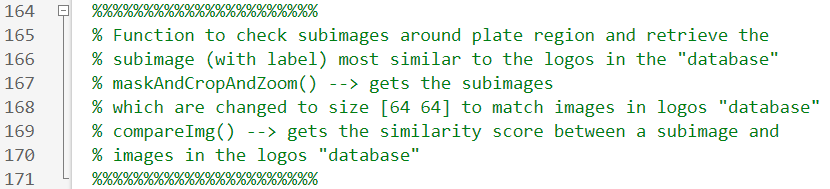


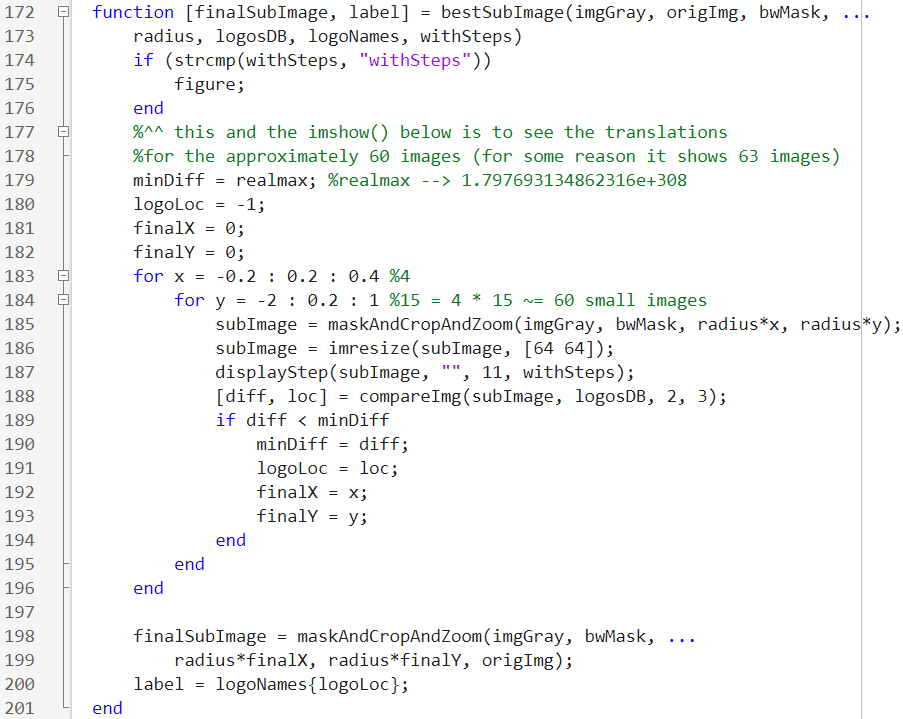
* bestSubImage() returns the subimage that most likely contains the logo and the label (name) of the logo

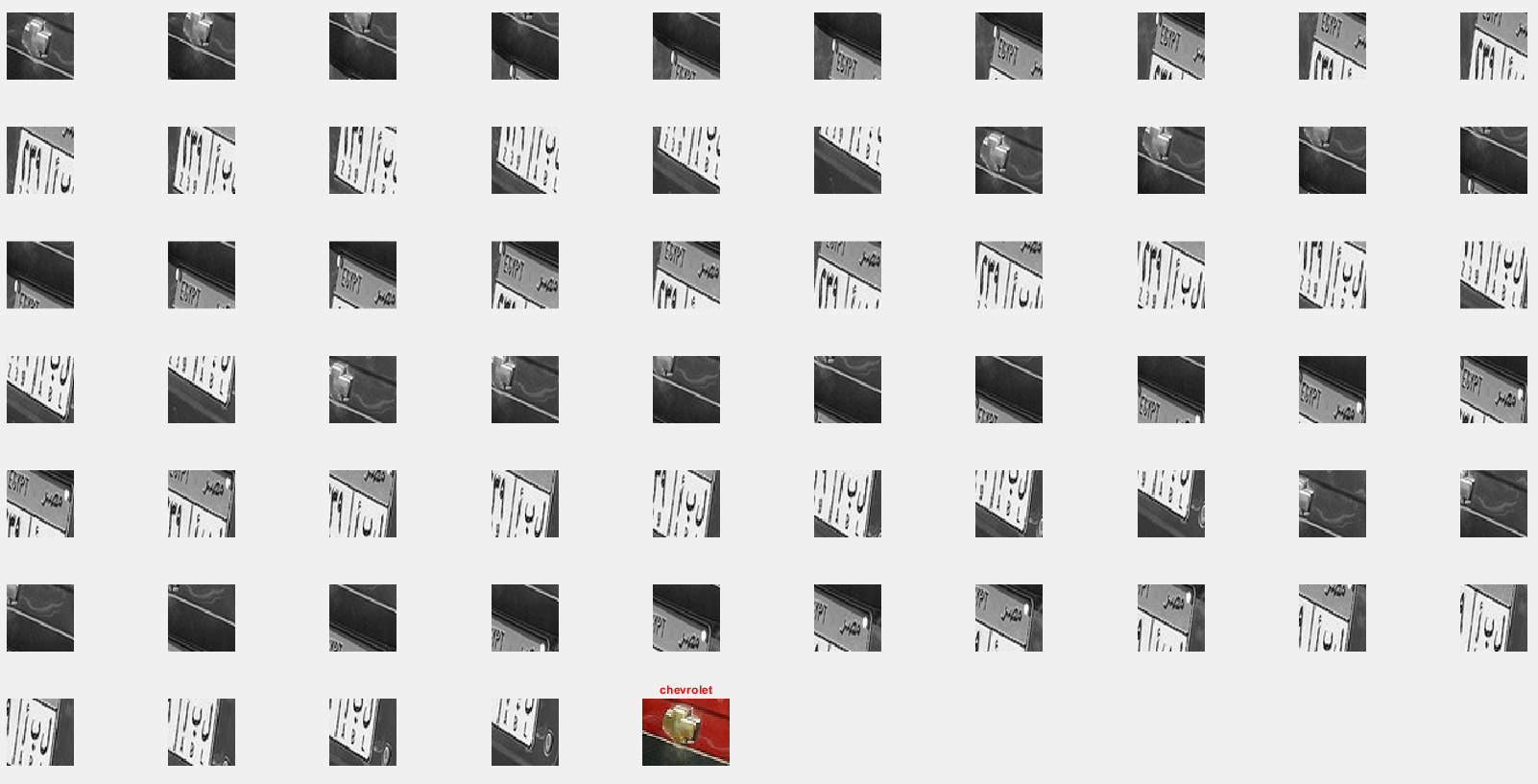
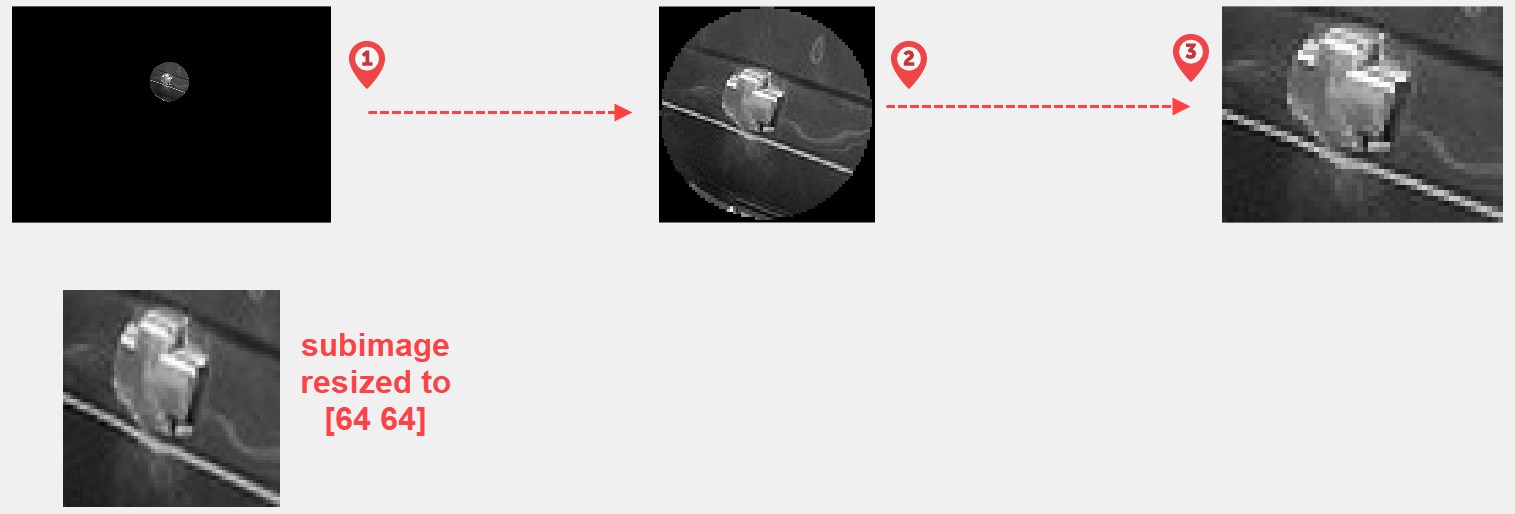
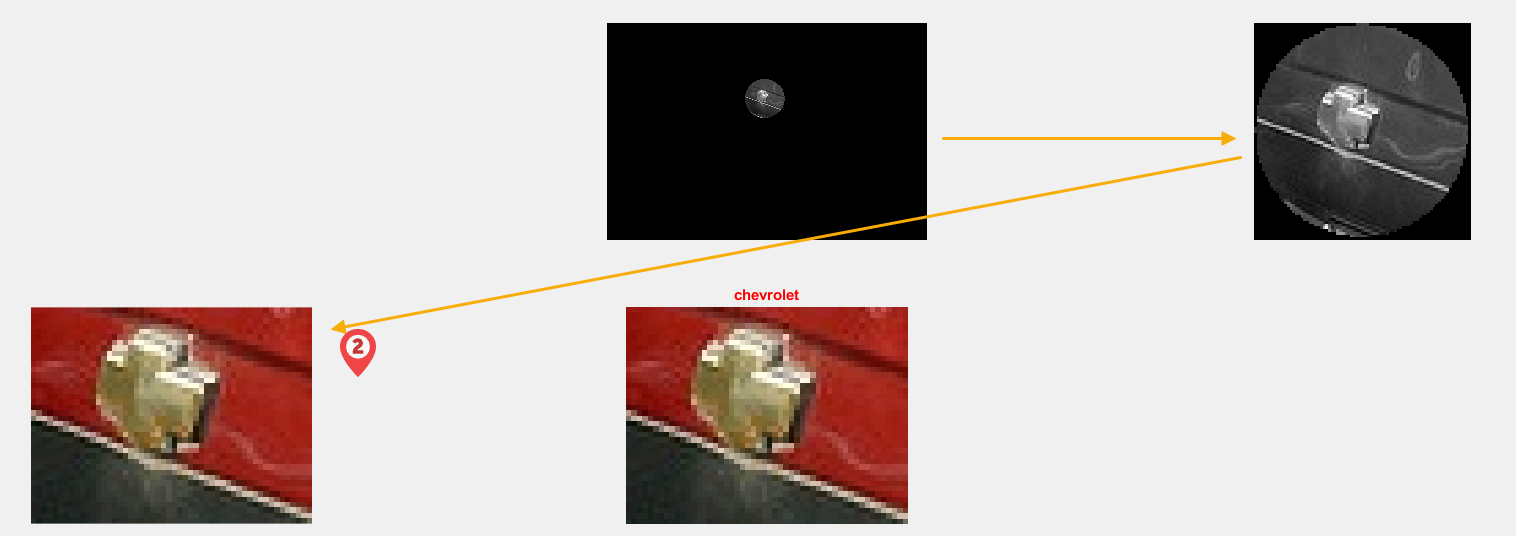
Example of Output Image:



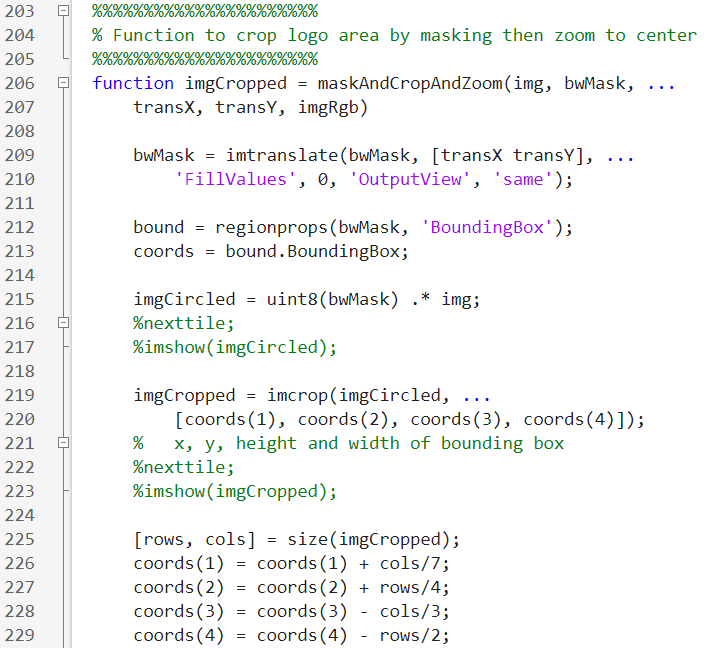
### bestSubImage()

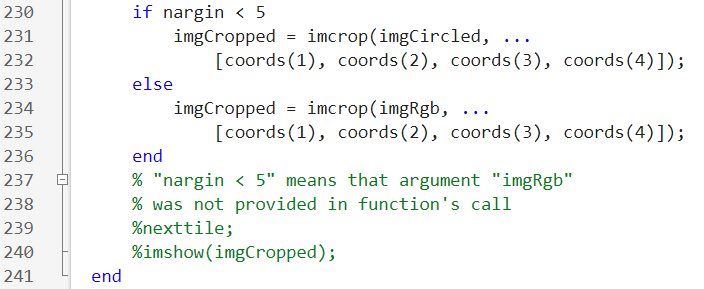


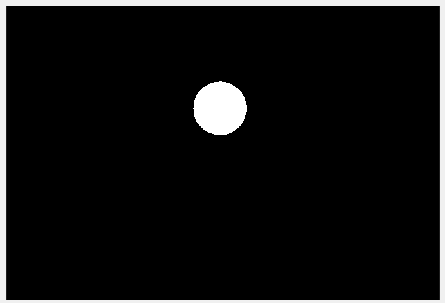
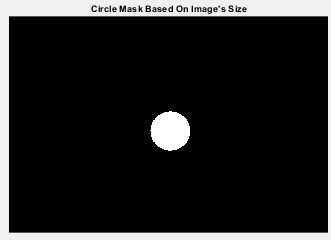
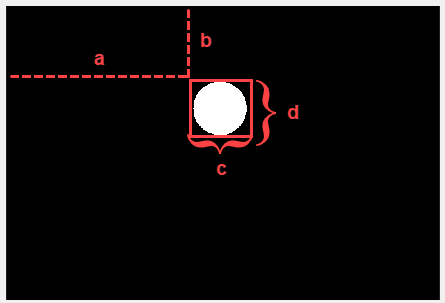
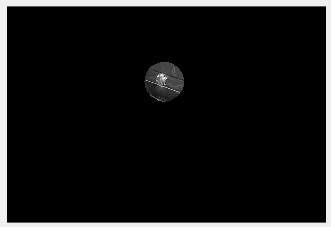
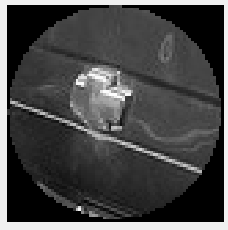
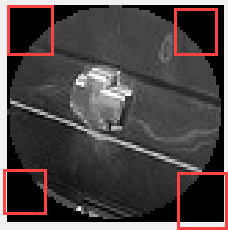
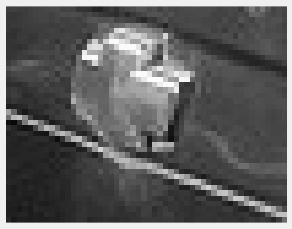


* The first if condition checks if “withSteps” was in the last argument in imgToLogo() or not
  + If “withSteps” was written, the create a new figure to display the subimages generated around the masked area
* Lines 178 to 182 initializes some variables that will be used in the nested for loop
  + realmax is a built-in constant that is very high
* The nested for loop is used to slightly translate the mask in the x and y directions by ratios of the mask’s radius (for example, translating by 40% of the radius’ length, then by 60%, etc)
  + The reason for this is that if the mask is around the license plate, we want to translate the mask around the license plate until a subimage that has the logo is found. Checking the subimage for a found logo is done using compareImg() function. Example:  
    
    - The “chevrolet” image at the end is actually related to the program’s output, not to the nested for loop
  + However, before calling compareImg(), the function maskAndCropAndZoom() is called to do the following:
    - Masks the image using the new translated mask (where the values of the translation are passed in the last two arguments)
    - Crops the mask to be a rectangle instead of a circle
    - Zooms in the masked image to get rid of black borders
    - All these steps are thoroughly discussed later in the report
    - Example:  
      
  + Also, before calling compareImg(), the subimage obtained from maskAndCropAndZoom() is resized to [64 64] in order to be properly compared with the images in the logos “database”, as they’re also stored as 64x64 images
  + ∵ compareImg() returns the best (least) score and the index of the logo (in the logos “database”) that is the nearest match to the given subimage, the if condition is used to store the score of the best subimage and location of the best logo match in “logoLoc”. It also stores the x and y translations done to obtain this best subimage in “finalX” and “finalY” respectively
* After the nested loop finishes, we can now use maskAndCropAndZoom() one last time with “finalX” and “finalY” to obtain the subimage that most likely contains the logo
  + Notice that an extra argument “origImg” is passed in the function this time. That’s because we now want to return the original colored image as the subimage to be displayed in the final output. Example:  
    
* logoNames{logoLoc} fetches the name of the logo that matched best with the subimage. Example:  
  

#### maskAndCropAndZoom()

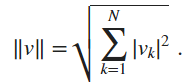




* imtranslate() translates the mask based on the x and y directions passed in the arguments
  + ‘FillValues’ determines the intensity that you want to fill the area that is now not occupied with the mask with. For our purpose, it is “0” as we want the rest of the image mask to be black
  + ‘OutputView’ = same is to make sure that the new image mask has the same dimensions as the original image mask
  + Example:  
    
* ‘BoundingBox’ is a property obtained from regionprops() that consists of info best explained with the following hypothetical illustration:  
    
  ∴ “BoundingBox” returns [a b c d] where “a” and “b” are the (x,y) coordinate of the top-left corner of the box, and “c” and “d” are the horizontal width and vertical height of the bounding box respectively
* imgCircled = uint8(bwMask) .\* img; multiplies each pixel in the original gray scale image “img” by either 0 or 1 to create the masked (filtered) image. Example:  
  
* imcrop() crops the image based on the bounding box information provided in the second argument. Example:  
  
* Lines 225 to 229 zooms in the image to remove these black borders:  
   → 
  + The values used to zoom in are obtained after trial and error
* “nargin” is a built-in matlab variable that counts the number of arguments passed, so the if condition checks if the original colored image is passed in the last argument or not
  + If it is passed, return the colored subimage
  + If it is not passed, return the gray scale subimage

#### compareImg()

#### 

* getFeatures() returns a vector of features if ‘single’ is passed in the first argument and a matrix (vector of vectors) if ‘multiple’ is passed in the first argument
  + This is because “reqImg” is the current subimage that we want to compare, and “imgsToCompareWith” (same as “logosDB”) is a cell array of images that are in the logos “database”
  + Note that the structure of “logosFeatures” matrix is as follows:
    - Each column represents an image in the logos “database”
    - Each row represents a specific feature
    - So if we have 16 images in the logos “database” and “numFeatures” is 3, then the matrix will be 3 rows by 16 columns
* The nested for loop along with the sqrt() at line 260 gets the euclidean norm:  
    
  Where  = “sqrdDiff(i)” of the i’th image and  = “reqImgFeatures(j) - logosFeatures(j, i)” of the kth (in the code, jth) feature.
  + ∵ the outer for loop iterates over the matrix’s columns (images) and the inner for loop iterates over the matrix’s rows (features). ∴ if, for example, i = 3 and j = 2, then “reqImgFeatures(j) - logosFeatures(j, i)” means we’re subtracting the second feature of the subimage with the third image in the logos “database”.
* mink() is used to get the best feature match (least score) along with the index of the logo in the logos “database” that best matched the subimage

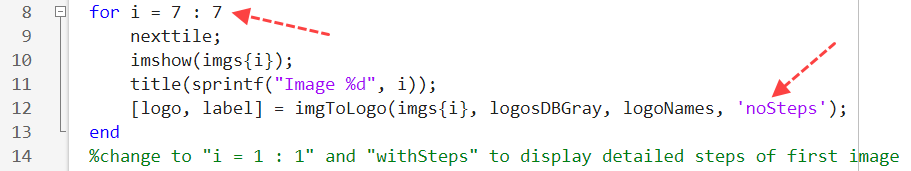
##### getFeatures()

##### 

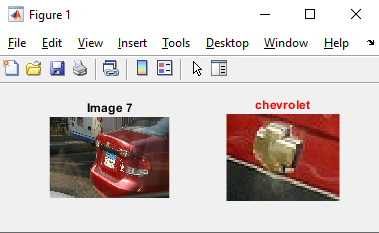
* This function’s returned value (i.e. “logosFeaturesVectorOrMatrix) is either a vector (if “singleOrMultiple” is “single”) or a matrix (if “singleOrMultiple” is “multiple”)
* The features are obtained using the following steps:
  + Convert intensity values of the current image to double to avoid precision errors
  + Get the two-dimensional Fourier transform of the current image (matrix) using a fast Fourier transform algorithm (fft2())
    - Note that the values of the fourier transform are chosen to be the features of the image
  + Get the absolute value of these fourier transform values using abs()
    - The “(:)” in “fourier(:)” is to change the vector of fourier transform values to be from a x 1 to 1 x a (where “a” is all the values returned from fft2())
  + Sort the “fourier” vector in descending order
  + Get the first “numFeatrues” features in the vector
  + Assign the selected features (values) to “logosFeaturesVectorOrMatrix”
    - In line 293, the “(:, i)” is best explained with a hypothetical example:  
      If numFeatures is 3, i = 1, and we perceive “logosFeaturesVectorOrMatrix” to be initially an empty matrix like this:  
      **0** 0 0 0  
      **0** 0 0 0  
      **0** 0 0 0  
      And “fourier” has these values:  
      3  
      6  
      8  
      Then “logosFeaturesVectorOrMatrix” will be:  
      **3** 0 0 0  
      **6** 0 0 0  
      **8** 0 0 0

# Program’s Outputs

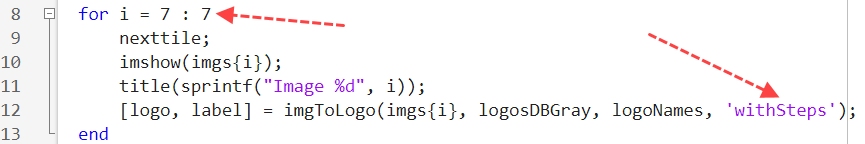
## Option 1



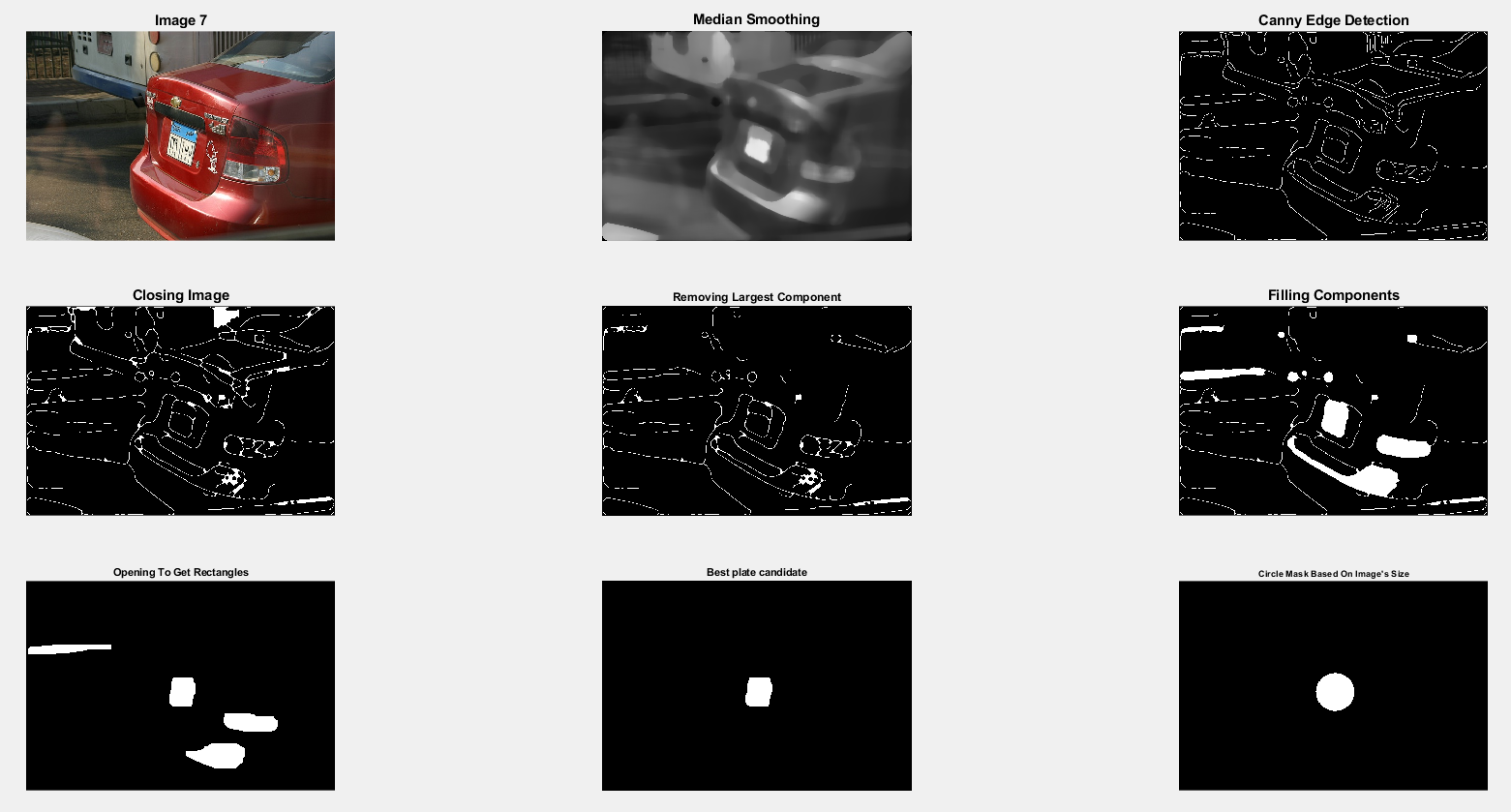
Output:

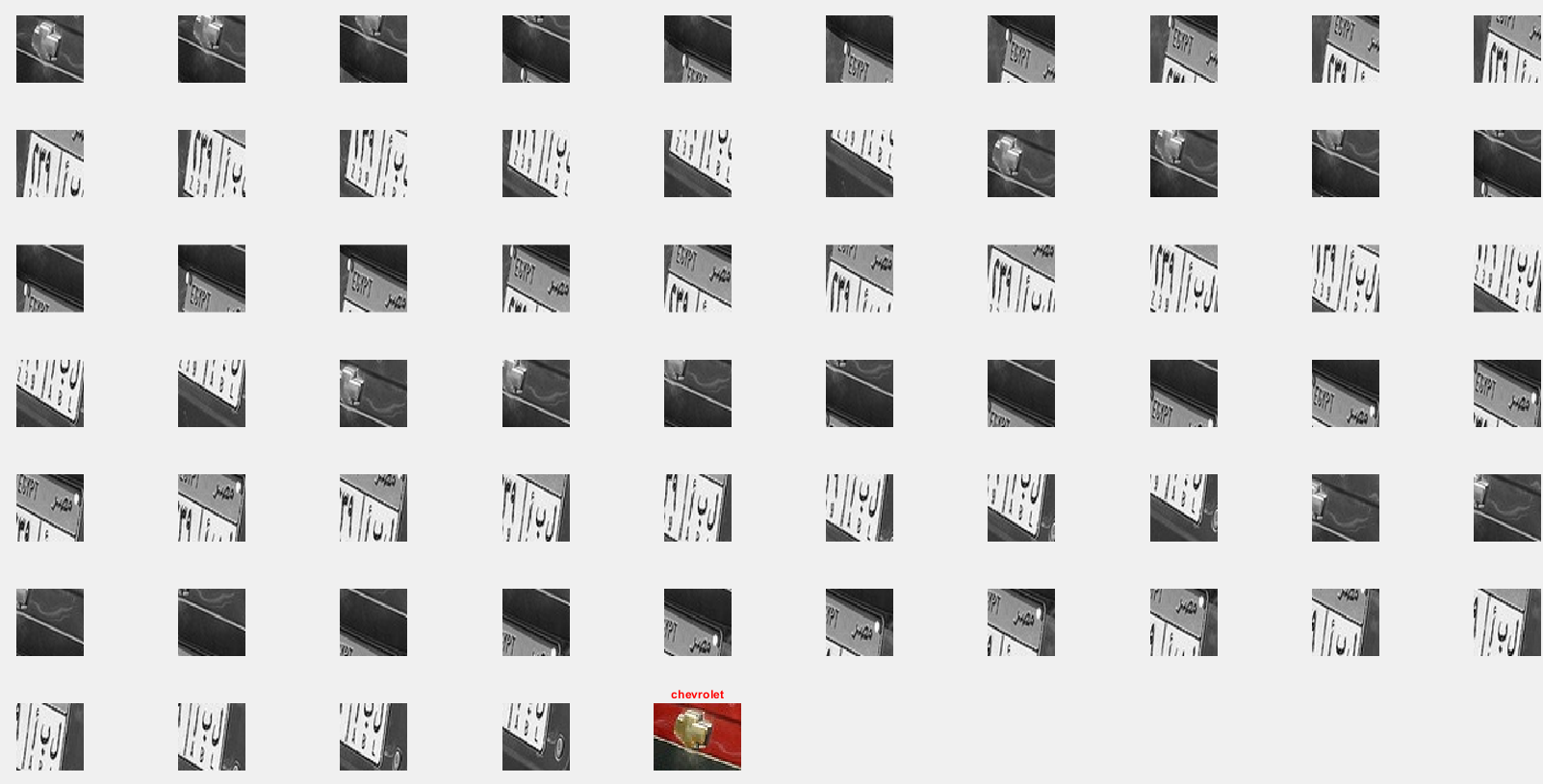


## Option 2

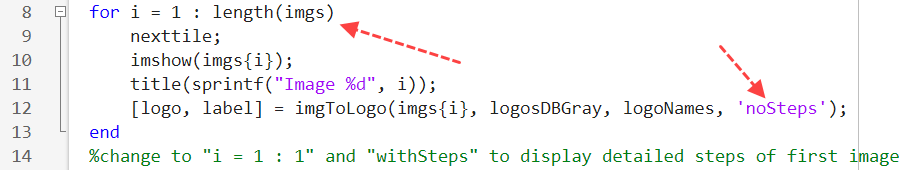


Output:

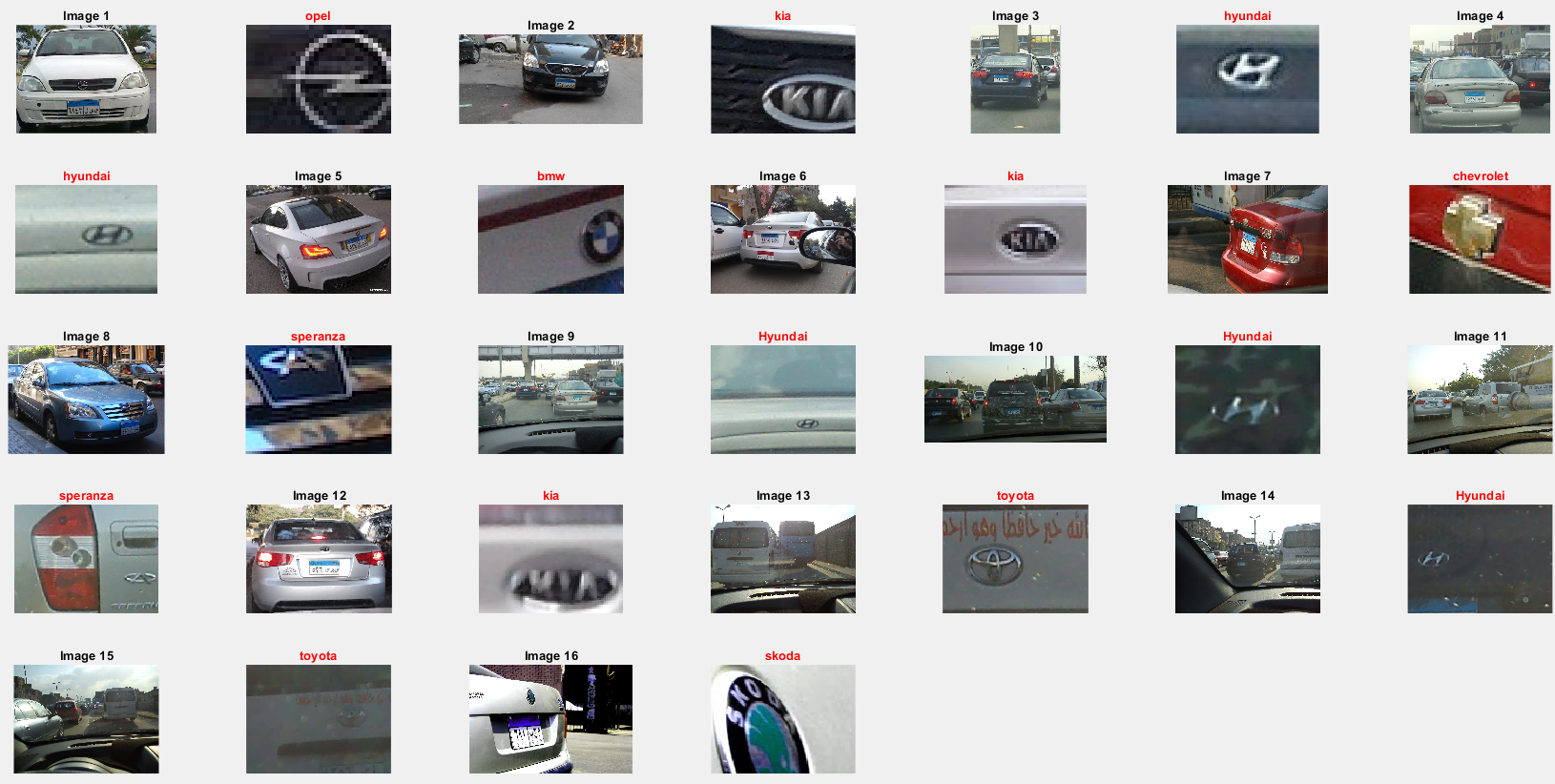




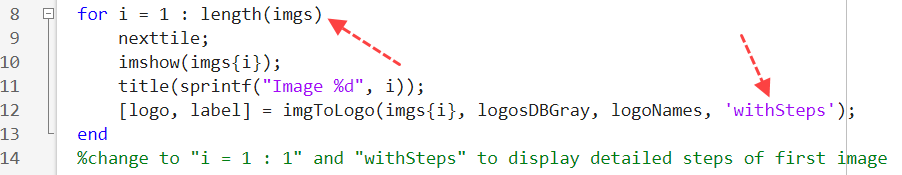
## Option 3



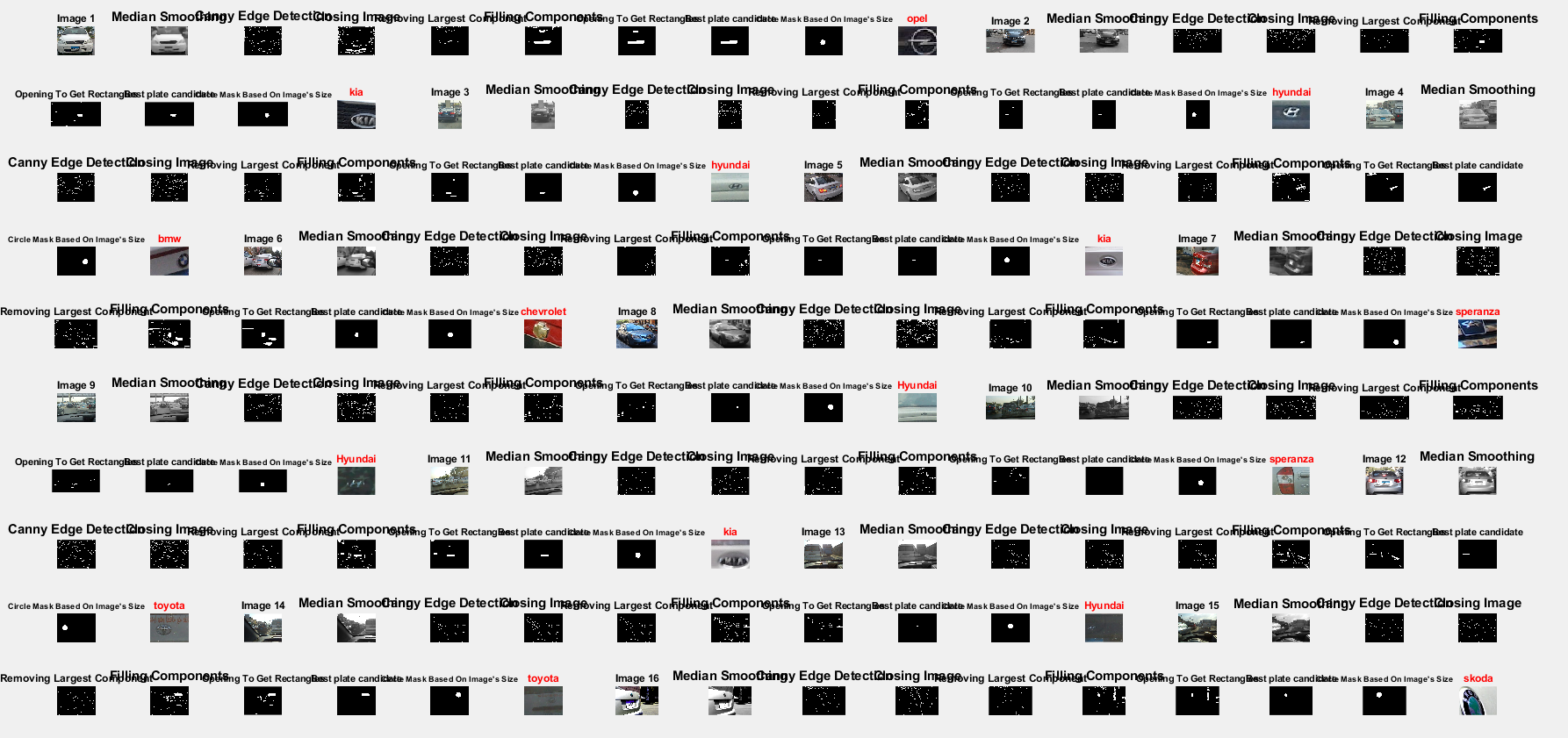
Output:



## Option 4



Output:



Note:

No output of bestSubImage() was displayed to avoid cluttering:

