RoboArm
Project Report #3

December, 2020

Justin Tam 300026741
Lucas Rahn 300021873

1-Introduction:

One of the most challenging parts of robotics is sensing the environment. The data we collect is never as nice as we would like it to be, and often very difficult to interpret. The purpose of this lab is to describe in detail a method of using computer vision techniques to analyze a single picture of a grid of various objects. Doing so, determine the shape and pose of each object. The data will then be given to a 6 degree of freedom Robix robotic arm to be interacted with. The shapes to be interpreted are either a square, circle, triangle, or rectangle. The main focus of this document will be to introduce the methodology used in the image processing, as well as to document the results. To use the Robix we will be using the forward and inverse models derived from 'CEG4158report_1_Tam_Rahn' and 'CEG4158report_2_Tam_Rahn'.

2-Methodology:

To integrate a computer vision system into the Robix, we use a webcam placed directly above the workspace. The webcam is then used to extract a single frame, which is then processed externally on a desktop computer using Matlab. The Matlab processing design is split into 4 main sections: preprocess the image, determine the object pose(orientation and position), identify the shape, and finally transform the pose into the reference frame of the Robix.

2.1-Preprocessing:

To extract a single frame from the installed webcam we take a screenshot and download this directly to the external desktop. From here, make use of Matlab's image processing toolbox to read the image, convert to grayscale as the color adds no extra information, and invert the pixel values(white becomes black, and black becomes grey, where each pixel value is on the scale of 0-255). The inverted image allows for an easier time processing the image and looking for the objects, as all points of interest correspond to high pixel values. The inverted, and read image is then passed to simultaneously determine the reference frame transformation, and object pose then shape.

2.2-Object Pose:

To calculate the pose of each object we decide to scan the image for objects, crop each object(see Figure 1), and continue to apply a generalized analysis to each object individually that returns the orientation, and position with respect to the image post processing(here on referred to as the original image). The general analysis is composed first determining the position of the shape, then finding the orientation of the given shape both with respect to the cropped image.

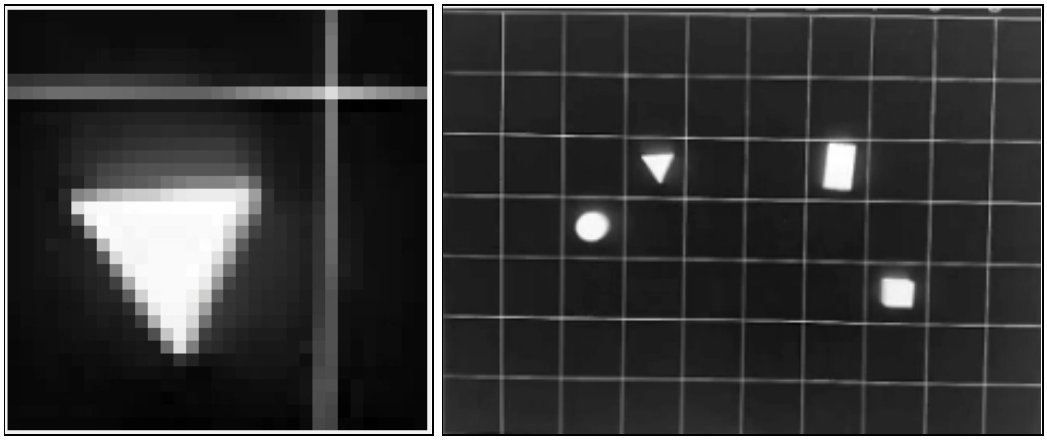
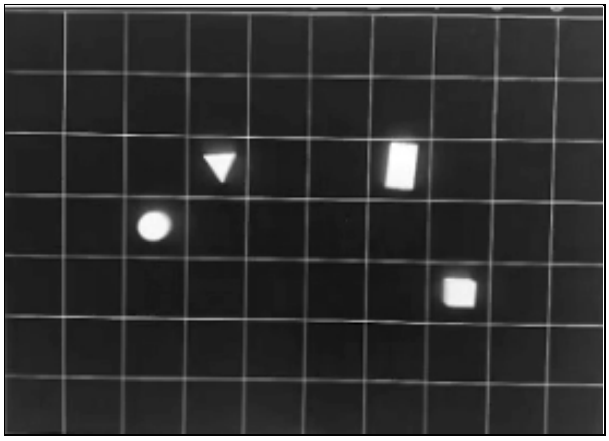


Figure 1: Cropped image versus the image post processing.

2.2.1-Isolating Each Object

To scan the image for objects we convert the image to a double(as so it's not constrained by the 0-255 integer range of ubit8) and convolute the image with a 20x20 array filled with ones, such that the resulting array is the same size as the original image. The convolution is used in order to have the magnitude of each pixel value be proportional to the intensity of itself, as well as the neighbouring pixels. The resulting array could be thought of as a 2D map where the intensity of each pixel indicates how many, and how 'bright' the surrounding pixels are. The array is passed through a threshold, for which pixels below the threshold are set to 0(black) and any remaining pixels are left unchanged. The product is a 2D image where only the objects have non-zero pixel values(see Figure 2). Note the size of the 20x20 filter used was chosen for this step, in order to create a large enough difference in pixel values between the objects and any other interference.



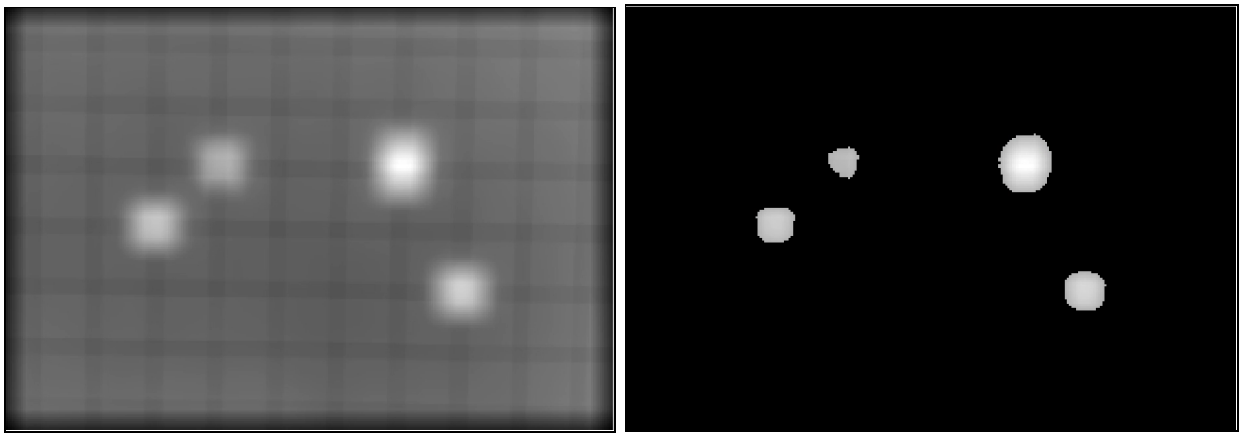


Figure 2: Convoluted original image versus the same image after a threshold filter

Next we locate, and return the approximate position of each shape by scanning across the convoluted image. We do this by sliding a 30x30 window across, and at each iteration comparing the pixel value at the middle of the array against a threshold value, as well as the values within the window. Thus, only if the middle pixel has the highest value within the window(neighbouring pixels) and the value is higher than a greater threshold value, the pixel will be recorded as the location of the shape. The idea behind this method is that the center of each shape will be the highest value or tied with, and only when the surrounding pixels are less than the current pixel in the slide will the value be recorded as the position of the shape. Note the added constraint of a threshold value is only added to avoid recording windows with all 0s as shapes. The coordinates of each shape are passed to another function to crop out a window surrounding the given pixel, and apply a generalized analysis to determine position and orientation.

2.2.2-Determine Position:

To calculate position we filter and threshold each isolated shape, and proceed to find the center of mass of each object. The filter applied is a 2x2 filled with ones, to again apply a dependence on neighbouring pixels. This image is then thresholded such that all pixels below a certain constant are set to 0, and the pixels remaining are all set to 1(see Figure 3).

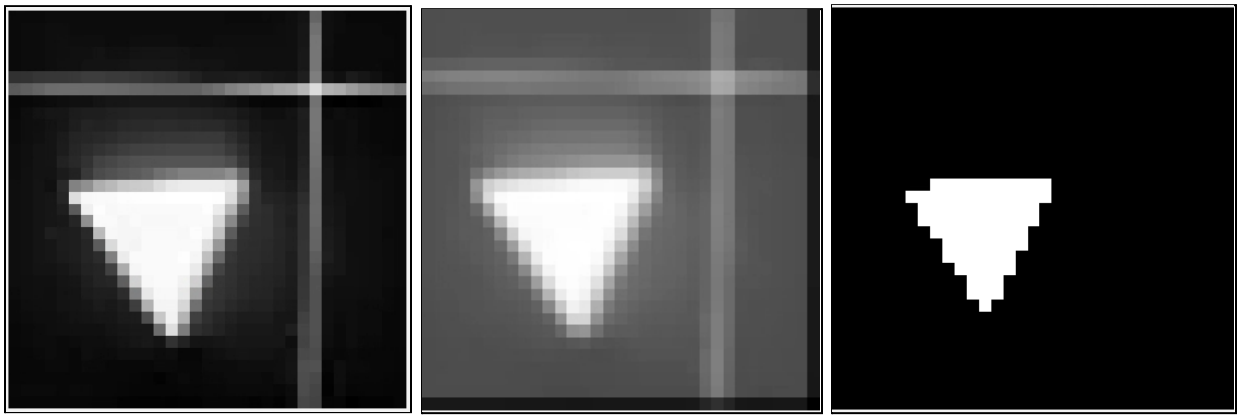


Figure 3: Comparison of the given cropped image(left), convoluted image(middle), and thresholded image(right)

To find the center of mass we independently evaluate with respect to the vertical and horizontal axis. For each axis the image is summed along that opposing axis, producing a 1D array where each value represents the total number of non-zero pixels for that coordinate(see Figure 4). Note that because each non-zero pixel was set to 1, the 1D arrays represent the number of pixels. The respective center of mass is then calculated by multiplying each index of the 1D array by its corresponding value, and then dividing by the total sum of the array. Each calculated center of mass is then adjusted to be with respect to the original image, then returned.

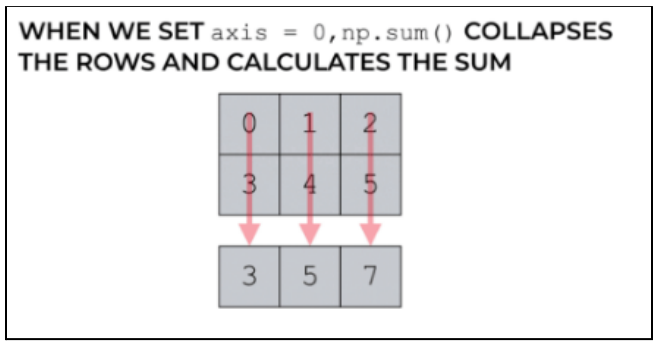


Figure 4: Illustration of summing along a given axis

2.2.3-Determine Orientation:

In order for the Robix to grip each object, the angle the gripper needs to be with respect to the object is different for each shape, and object orientation. We then seek a method to find the rotation of each shape, defined as the angle required for a gripper to be rotated in order to securely grip the object. To find this rotation we use an iterative approach, by rotating the image through 360 degrees, and evaluating the orientation of the rotated shape at each degree, i.e. how well the gripper can pick up the shape. For instance, a rectangle rotated 360 degrees should produce 2 possible ideal orientations, where the longer axis is horizontal, and the shorter axis is vertical(see Figure 5). To evaluate at each degree we evaluate the 'horizontalness'. This is done by applying an edge filter, and convoluting the edge-filtered image with an image containing a horizontal line, then taking the sum over the entire image. This sum will then be dependent on how horizontal the edges are(see Figure 5). To analyze the sums for each rotation we can plot the sum vs degree, and look for peaks. As shown in Figure 6, we see 4 peaks, with 2 of the peaks being noticeably larger. We expect all 4 to correspond to the 4 orientations of the rectangle where a flat edge is horizontal, and the 2 larger peaks corresponding to orientations where the longer edges are horizontal. The rotation with the highest sum(one of the 2 larger peaks) is then taken to be the rotation we're looking for.

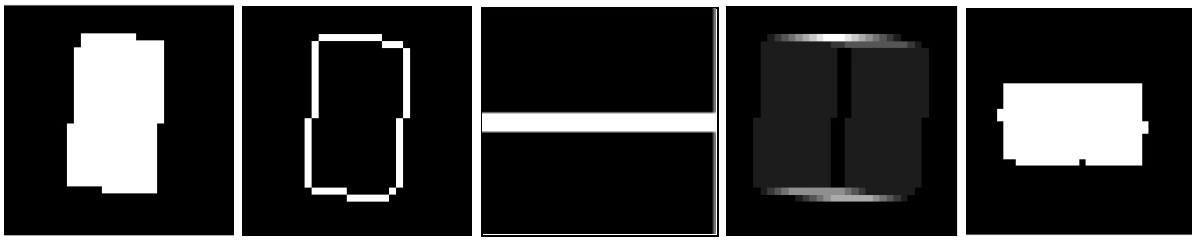


Figure 5: Thresholded image, edge filtered image, horizontal-line filter, horizontal-line-filtered image, and rotated image with the highest 'horizontalness', respectively

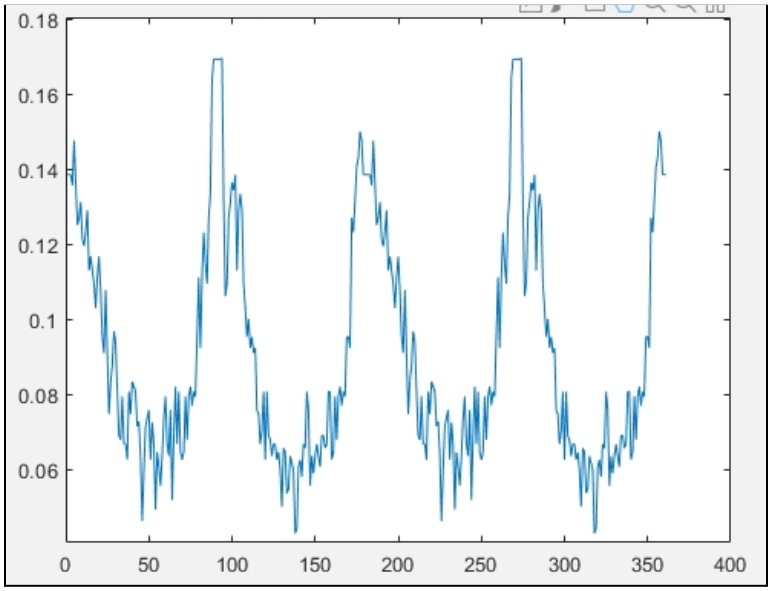


Figure 6: 'Horizontalness' vs degrees for the rectangle in Figure 5

2.3-Shape Identification:

To identify the shape of each object we calculate the dimensions of the rotated and filtered shapes, and use these parameters to calculate an expected area for each possible shape(i.e. πr^2 for a circle), and compare this to the measured area of the filtered image itself. The result is a confidence value corresponding to each shape, where the shape is identified corresponding to the highest confidence value.

2.3.1-Calculating the Dimensions of Each Shape:

To find an expected area for a single object corresponding to multiple shapes, we need to find the dimensions of the object in question. Given the thresholded images of each shape, we simply need to find the lower and upper bounds for each axis. To calculate these values for an axis we collapse the image to a 1D vector along that axis(similar to the center-of-mass calculation, by taking the sum along the opposing axis) and iterate from the lower to upper indices until a non-zero value is found. The 1D vector is then iterated again only from upper to lower indices, again until a non-zero value is found. These 2 values correspond to the upper and lower bounds for the respective axis.

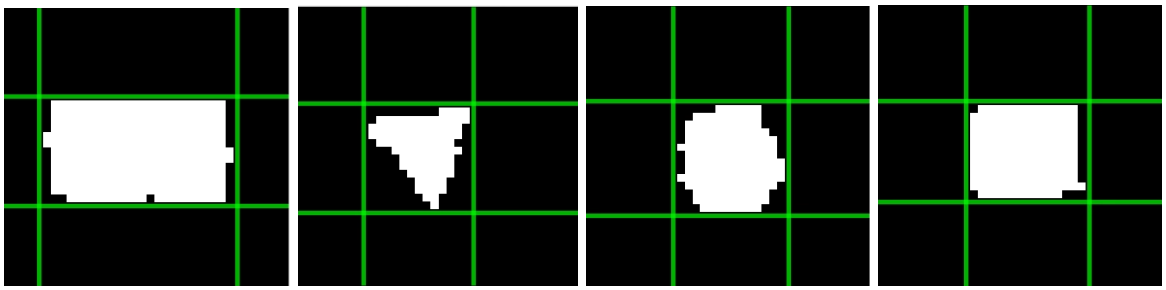


Figure 7: An example of each isolated shape and their calculated dimensions

2.3.2-Calculating Confidence Values:

For a single, isolated, rotated, and thresholded object, we begin by calculating an associated error for each one of the possible shapes it could be. The error calculation for each shape consists of taking the absolute difference between the expected area and the measured area. To calculate the measured area we remember that each pixel is either 0 or 1, and so we sum the entire image into one value, corresponding to our measured area. To find the expected area we use the 2 sets of upper and lower bounds calculated previously to calculate the width and height of the unknown shape. From here we calculate an expected area for each possible shape. The error for each possible shape is then converted into a confidence value by taking the negative, natural exponential of the error, as seen in equation 1.

$$confidence(error) = e^{-error} \quad (1)$$

To calculate the area for a square, circle, and triangle, the dimensions of the object are not always measured to be perfect. For instance, even though the shape is a square the length does not always equal the width exactly. A similar argument can be made for the triangle and circle parameters. To combat this we calculate all possible areas for a given shape, and take the one that has the most error. For example, to calculate the expected area of a square, we compare the absolute difference using the width squared as the expected area, versus the length squared and choose the error to be the value with the most error. We take the larger error since if the shape is what is being calculated, the difference between the 2 calculations should be minimal at most. One other special case does occur when trying to identify a rectangle versus as square using the method proposed. For instance, if the object is a square the expected areas corresponding to a square and rectangle will return the same value. We work around this problem by changing the error function of the square to take the minimal value of possible errors, and then adding a bias term to the error function of the rectangle that is proportional to the difference between width and length. The rectangle bias is then in order to adjust for the case when a rectangle is predicted as a square, due to taking the minimum possible error for the square prediction.

2.4-Reference Frame Transformation:

Here we wish to convert the measured pixel coordinate of each object, to x and y distance measurements as used by the grid, with respect to the base of the Robix. Because the conditions for completing such a task are usually specific to the application and environment in which you're working, it is fitting to have a specific solution. Additionally, for reasons discussed in the discussion, it seemed inappropriate to simply shift the axis of reference from the top left corner of the image to the base of the

robot, then apply a general pixel-to-length scaling factor across the entire grid. Accordingly, the grid had to be rotated such that the vertical and horizontal grid lines were appropriately vertical and horizontal. Additionally, the image was sized so that only the grid fit within the bounds of the image which allowed us to easily recognize each grid line on the image.

It is important to note the axis definition used for the reference frame transformation. As seen in Figure 8 the axis is defined for the Robix such that the positive Y direction is to the left with respect to the positive X direction. The grid used in the labs reverses this so that the positive Y direction is to the right with respect to the positive X direction. In this lab, when applying the method for reference frame transformation, we define the positive Y direction to be the same as that of the grid.

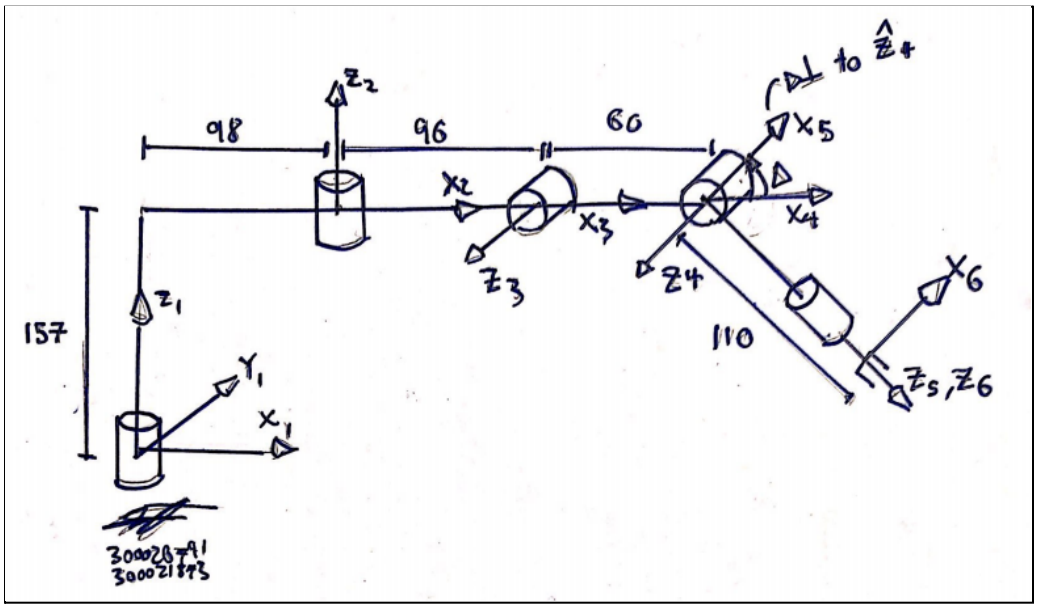


Figure 8: Axis Definition of Robix

For our approach, a pixel coordinate was associated with each grid line. Each vertical grid line was measured in the first row of the grid, starting from the top, and each horizontal grid line was measured in the first column of the grid , starting from the left (Figure 9 red lines). Because the number of grid lines from the top left to the base of the Robix was known, a base coordinate was associated with the location of the grid lines that intersected at the bottom-middle of the image (Figure 9 green dot). From here, each measured grid line was hard set to be a distance of 2 inches away from the previous (Figure 9 blue arrow represents fixed distance), starting from the base, and the scaling factor to be applied between grid lines was calculated by dividing the location of the object pixel with respect to one of the grid lines, by the distance between the two closest grid lines, and finally multiplying by 2 inches. Now, we see that distance from the base can be calculated by taking the hard set value of each grid line being two inches, and adding the specific scaling factor to compensate for the distance between grid lines.

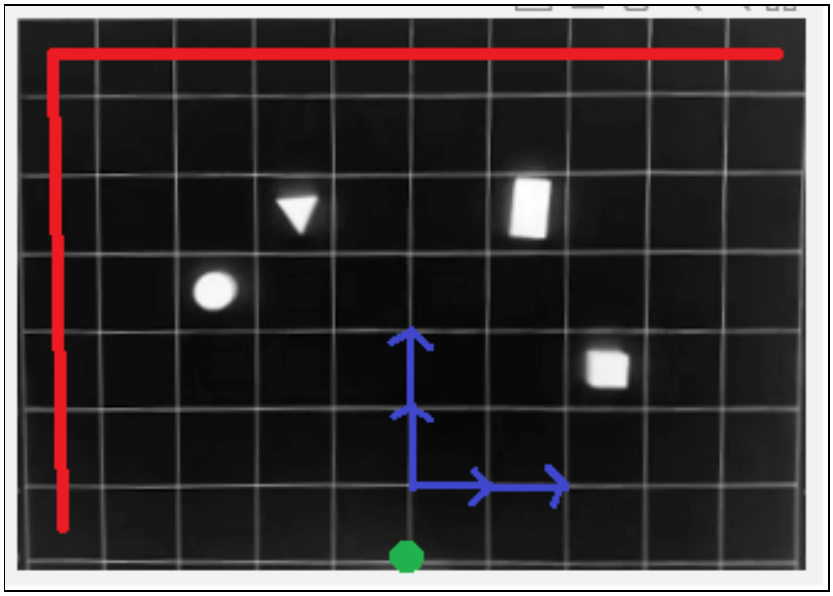


Figure 9: Depiction of Reference Frame Transformation Definitions

3-Implementation:

The implementation is split into 2 sections. The first section corresponds to the image processing program derived in this report, and is concerned with finding the cartesian coordinates of each shape with respect to the Robix, as well the type of shape and what orientation to grip them. The second section corresponds to using the inverse model derived in Lab 2 as well, in order to move the end effector to each of the identified objects. Thus, combining all of the work done from Labs 1-3.

3.1-Image Processing:

To test the image processing we use a screenshot obtained from “Image Processing Lab Project", in order to test our program for all 4 shapes opposed to the imaged used, and obtained during Robot Control. The program produces 4 figures, 1 for each shape, and a single 4x1 Matlab cell where each element is an additional cell corresponding to the position, orientation, shape, and corresponding confidence in that order from top to bottom. These are all shown below in Figures 10-12. Note that in displaying the position of each shape, the produced 1x2 vector is formatted as [x, y], such that it is consistent with the robot axis definition in Figure 8. Also note the confidence produced in Figure 12 for the rectangle. The confidence is clearly greater than 1, as a result of the bias used in differentiating between the rectangle and the square.

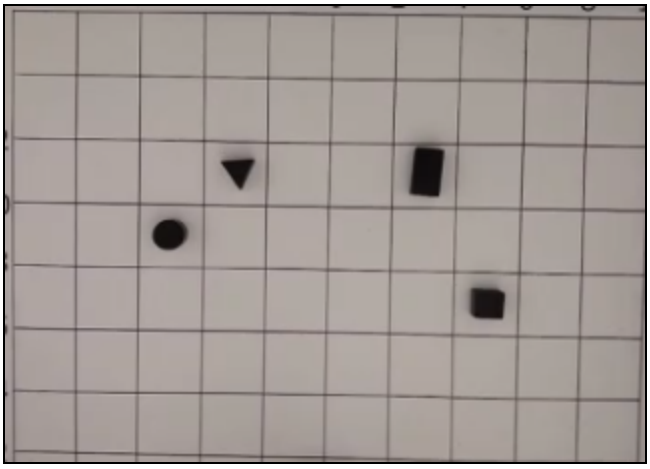


Figure 10: Test image used in the image processing implementation

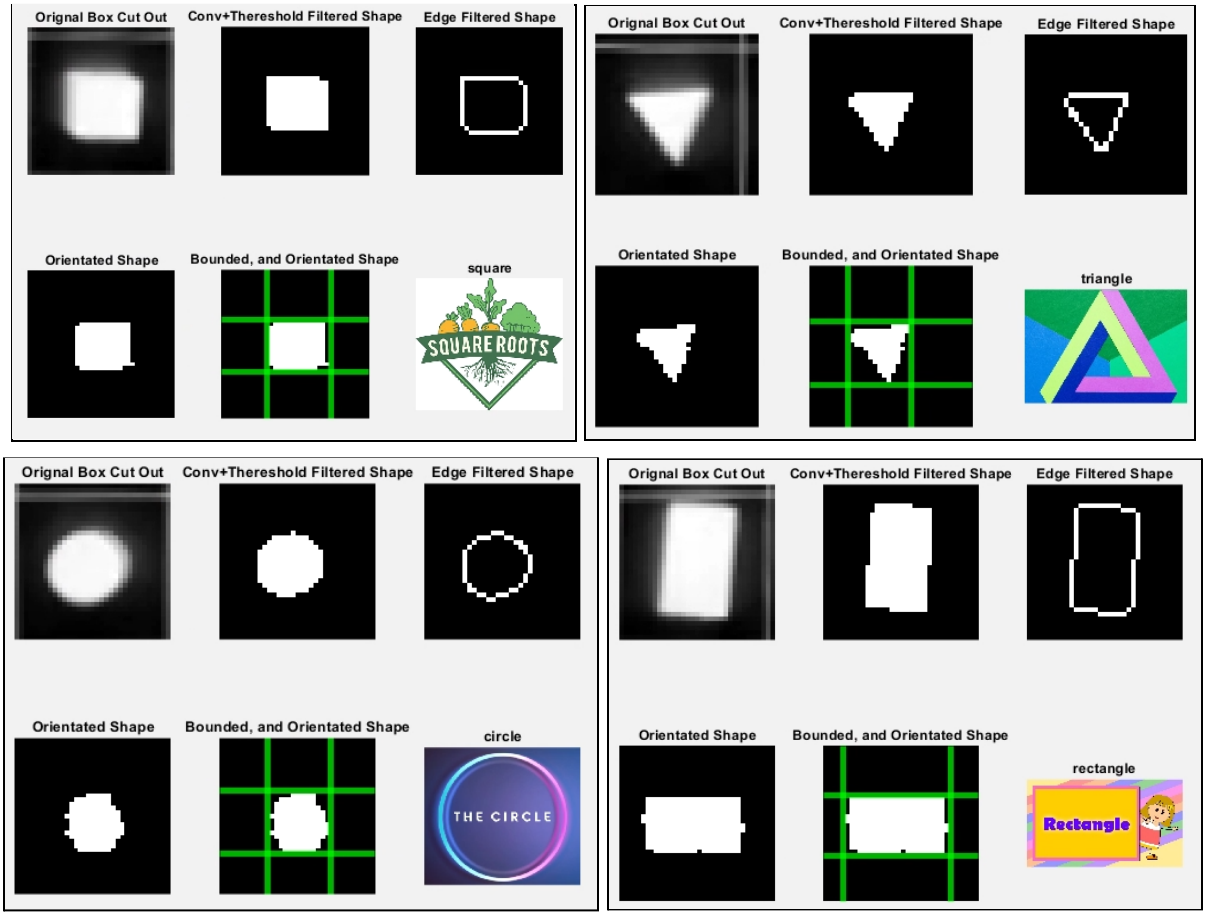


Figure 11: Image progressions for all shapes

shape{1, 1}

	1
1	[11.0742,2.8626]
2	5
3	'triangle'
4	0.9407

shape{2, 1}

	1
1	[11.1228,-3.0676]
2	94
3	'rectangle'
4	2.9810e+03

shape{3, 1}

	1
1	[8.9860,4.9474]
2	83
3	'circle'
4	0.9939

shape{4, 1}

	1
1	[6.9667,-5.0292]
2	6
3	'square'
4	0.8805

Figure 12: Final results from image processing

3.2-Robot Control:

To test the ability of the end effector to reach the shapes predicted by the image processing module, we apply the module and attempt to move the end effector to that position using the previously derived inverse model. To generate a valid Q matrix to give to the model, we set phi = 180, theta = orientation, psi = 0, z = 1, and the x and y coordinates as determined. The results are shown below in Figure 13.

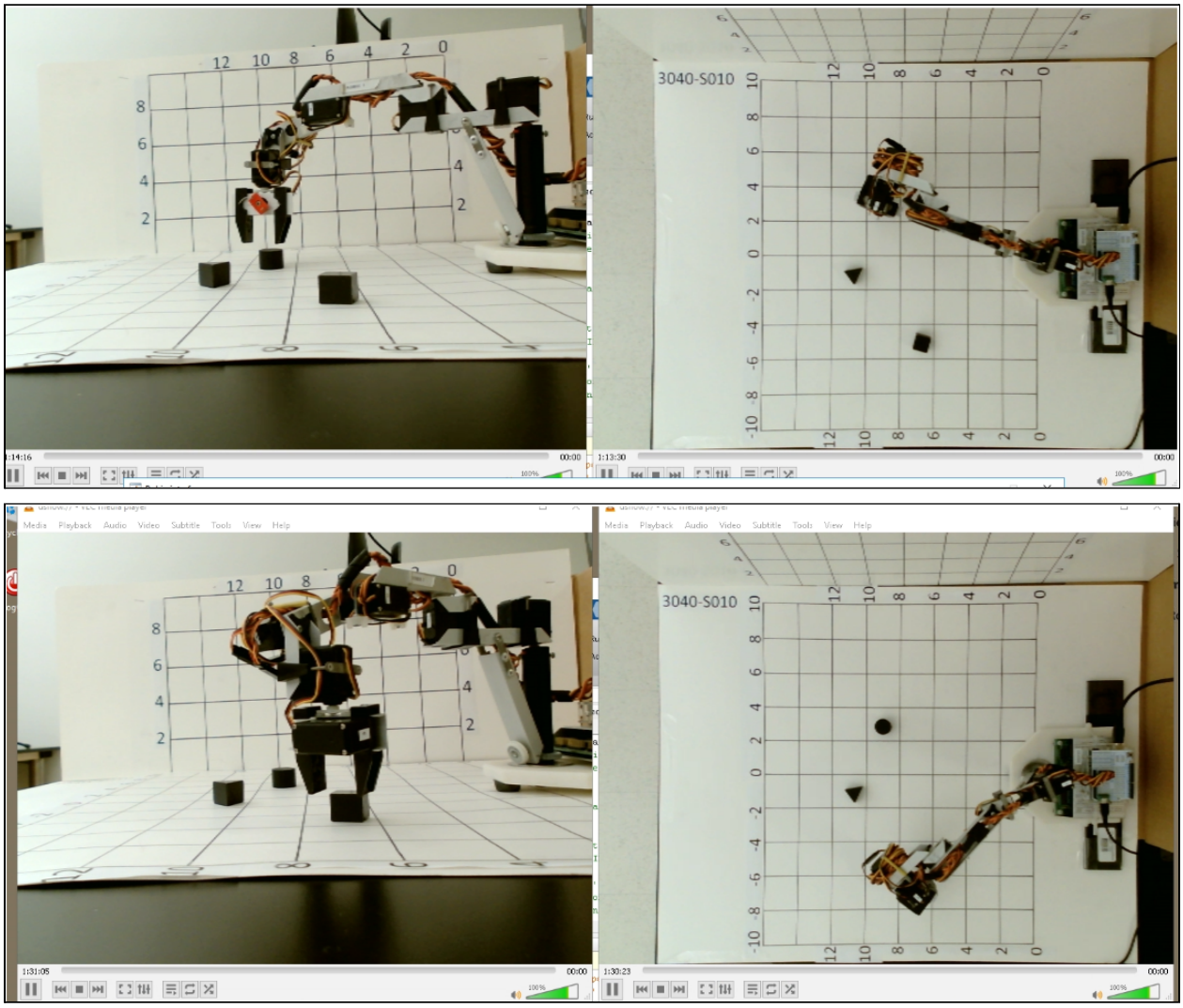


Figure 13: Implementation of the inverse model using image processing

4-Discussion and Analysis:

4.1-Error Measurements:

$$Percent\ Difference\ =\ \left(\frac{|Object - End\ Effector|}{Object}\right)100\% \quad (2)$$

Table 1: Error measurements:

Object distance (x, y)	End effector Distance (x, y)	Distance Percent difference(%)
(9.21, 2.89)	(9.9, 3.0)	(7.5, 3.8)
(10.76, -1.18)	(10.9, -1.2)	(1.3, 1.7)
(6.82, -5.94)	(7.1, -6.1)	(4.1, 2.7)

Average percent difference including both x and y percent difference: 3.5%

4.2-Implementation:

There were a number of things that we thought could have been done to improve accuracy, and usability of the lab. First, some of the input parameters had to be input manually. Namely the robot was not set up to when run, immediately identify objects, and move them to desired locations. Preprocessing steps preceded this. To set up the robot, a figure was captured (it was decided to take screenshots of the webcam view) and imported by the program. This image had to be further rotated and cropped if necessary, manually. In industry, it would be required that this task is automated, and vision was updated real time. As this lab is for learning purposes and geared more towards vision – identifying objects, their orientation, and respective location – it is fitting to focus on this.

When picking up objects we thought it would be necessary to have the arm pointing down. Firstly, it makes picking up objects more stable – we do not run the risk of approaching the objects at a bad phi or psi angle, in which you would be approaching the object with the gripper on the edge or corner of the object. Additionally it makes calculations much easier. The initial inverse model does not support predefining values for phi and psi. This is because not all joint angle solutions satisfy any arbitrary value for phi and psi, and it is difficult to know the possible phi and psi solutions with any given end effector position. When deriving the inverse model, we took an approach where the equations parameters were the index elements of the forward model. By defining phi = 180 and psi = 0, we ensure the end effector is directed downwards, and also simplify the forward model seen in Figure 14. Here we know the desired location and theta value for the end effector, which was found using the vision program, so the forward model becomes trivial, and we can use these index elements to calculate the inverse model equations. This approach of picking up objects leaves out complex analysis of defining and choosing one of the possible phi and psi solutions.

$$Q_{TRPY} = \begin{bmatrix} \cos\theta\cos\varphi & \cos\theta\sin\varphi\sin\psi - \sin\theta\cos\psi & \cos\theta\sin\varphi\cos\psi + \sin\theta\sin\psi & x_T \\ \sin\theta\cos\varphi & \sin\theta\sin\varphi\sin\psi + \cos\theta\cos\psi & \sin\theta\sin\varphi\cos\psi - \cos\theta\sin\psi & y_T \\ -\sin\varphi & \cos\varphi\sin\psi & \cos\varphi\cos\psi & z_T \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 14: General forward Q matrix for the TRPY convention

When replotting the points that are measured by the vision system the points get replotted exactly where the object is located. Additionally, we showed in the last lab that the joint angles received when using the inverse model are exactly the same as the ones used to generate the forward model. This shows that any error in the displacement of the end effector is not due to vision recognized position, or discrepancies in the joint angles but rather to another factor. It was difficult to calibrate the robot arm, as there was an initial shift and scaling problem with the arm. Additionally, looking at it on a computer screen made it difficult to confirm certain angles of the arm when testing the calibration. Had the lab been done in person it may have been easier to confirm certain calibration angles giving a more accurate span. It is also important to note that the calculated errors are approximate errors. When measuring the end effector distance, it is difficult to see where the actual end effector is, especially when the arm points down and is blocking the camera view. Again, had the lab been done in person, it would be easier to measure the distance of the end effector and would provide more precise error measurements. The positional error is not very high, 3.8% and it is reasonable to associate the lack of perfect calibration and imprecise end effector distance measurements with this error.

4.3-Methodology:

The methodology employed in this document to correctly identify the parameters of each shape was successful based on the test image used in both the robot control and image processing, but there are alternatives. To calculate the orientation we went with an iterative approach, independent of the shape. While this did lead to a relatively correct orientation, they were not always the most optimal. For example, the calculated orientation of the triangle in Figure 11 rotated the triangle an extra 5 degrees, when the triangle already appeared to be in its horizontal state. However, a different strategy could be optimized to instead identify the shape first, and calculate the remaining parameters more analytically.

Initially it was planned that the reference frame transformation would consist of shifting the pixel coordinate to the base of the Robix which would provide nice pixel indexing for scaling purposes. Translating to the right would make any points to the left of the new base appropriately negative, vertical coordinates could be flipped such that the bottom represented the zero pixel and the top of the image would correspond to the height of the resolution. From this, we could have easily assigned a pixel-to-length scaling factor that was general for the entire image. The problem we ran into when attempting this method was 2 fold – actual length became smaller with respect to the length measured by the pixels when moved closer to the edge of the image, and lines were uneven throughout the image, giving inaccurate results with respect to the grid. Measuring points close to the base gave decently accurate results, but moving farther away from the base, the error could be as great as 1 inch. The alternative, as described in the methodology, has its own drawbacks. Namely it reduces the workspace by one column and one row. Since pixel coordinates are determined for each line, placing objects in the first column or row would interfere with this measurement. It was planned to convolute and remove the

objects from the image so that the first a second row could be used, but depending on where the object was placed, sometimes the intensity of the grid line was lowered or removed entirely, so this was discarded.

5-Conclusion:

In this lab, we implemented a vision system that acted as an addition to the previous lab of defining the inverse model. The vision system consisted of analyzing a grid of objects identifying and differentiating between a square, circle, and triangle, determining their orientation, and respective location. This allowed us to use our inverse model calculations to direct the end effector of the Robix to the given objects and pick them up. There was little error in the displacement of the end effector, 3.8% which is associated with imperfect calibration and imprecise end effector measurements. Had the lab been done in person instead of over a remote connection, we expect this error could be minimized. The vision system was successful in identifying the correct object, and there was no measurable error in the orientation of the end effector. For the future we might improve this by implementing more automation in some tasks, as described in the discussion, but thought that for the purpose of the focus of the lab (vision system), and for learning purposes, what was implemented was ample, and we are satisfied with the results.

Appendix A

```
clear;
close all;
%\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Main\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

%Read image:
img_grey_ubit8 = rgb2gray(imread('Lab3_test_img.PNG'));
img_grey = double(img_grey_ubit8);
img_grey = 255 - img_grey; %invert colors
img_grey_ubit8 = 255 - img_grey_ubit8;

img_rotate = imrotate(img_grey_ubit8,1,'bilinear','crop');
figure, img_final = imcrop(img_rotate);
img_fin_doub = double(img_final);
figure, imshow(img_fin_doub, [])

pixel = shape_coordinate(img_fin_doub);
object_num = size(pixel);
shape = cell(object_num(1), 1);

for i = 1:object_num(1)
    shape_temp = identify(pixel(i, :), img_fin_doub);
    locxy = shape_temp{1}; locxx = locxy(1); locyy  = locxy(2);
    [locy, locx] = get_length(img_final, locyy, locxx);
    shape_temp{1} = [locx locy];
    shape{i} = shape_temp;
end


%\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Functions\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
%
%Outlining Structure:
%
%Returns the pixel coordinates, orientation and shape
% for each object within the grid.
%The orientation for each shape is defined by the angle wrt to the robix
%that the gripper must be in to grasp the object.
%


function bound_arr = bounds(box_ori_bin)
    sz = size(box_ori_bin);
    x_sum = sum(box_ori_bin, 1); %collapse to a column vector
    for i = 1:sz(2)
```

```
40      function bound_arr = bounds(box_ori_bin)
41 -        sz = size(box_ori_bin);
42 -        x_sum = sum(box_ori_bin, 1); %collapse to a column vector
43 -        for i = 1:sz(2)
44 -            if x_sum(i) > 0
45 -                xL = i-1;
46 -                break
47 -            end
48 -        end
49 -        for i = sz(2):-1:1
50 -            if x_sum(i) > 0
51 -                xU = i+1;
52 -                break
53 -            end
54 -        end
55 -        y_sum = sum(box_ori_bin, 2); %collapse to a row vector
56 -        for i = 1:sz(1)
57 -            if y_sum(i) > 0
58 -                yL = i-1;
59 -                break
60 -            end
61 -        end
62 -        for i = sz(1):-1:1
63 -            if y_sum(i) > 0
64 -                yU = i+1;
65 -                break
66 -            end
67 -        end
68
69          %Output Values
70 -        bound_arr = [yL yU; xL xU];
71 -    end
72
73
74      function pixel = shape_coordinate(img_grey)
75 -        shape_filter = ones(20);
76 -        img_conv = conv2(img_grey, shape_filter, 'same');
77 -        figure, imshow(img_conv, [])
78 -        sz = size(img_conv);
79
80          %Isolate shapes
81 -        for j = 1:sz(1)
82 -            for i = 1:sz(2)
```

```
80          %Isolate shapes
81 -        for j = 1:sz(1)
82 -            for i = 1:sz(2)
83 -                if img_conv(j, i) < 54000
84 -                    img_conv(j, i) = 0;
85 -                end
86 -            end
87 -        end
88 -        figure, imshow(img_conv, [])
89          %figure, imshow(img_conv, [])
90
91          %Reduce shapes to single coordinates:
92 -        window_sz = 30; %n x n window size
93 -        shape = 0;
94 -        img_conv_pad = img_conv; %Add padding
95 -        img_conv_pad(sz(1)+window_sz-1, sz(2)+window_sz-1) = 0;
96 -        for j = 1:sz(1)
97 -            for i = 1:sz(2)
98 -                l = j+round(window_sz/2); %Define middle
99 -                m = i+round(window_sz/2);
100 -               window_arr = img_conv_pad(j:j+window_sz-1, i:i+window_sz-1);
101 -               maximum = max(max(window_arr));
102 -               [y, x] = find(window_arr == maximum); aa = [y, x];
103 -               if (aa(1, 1) == l-j) && (aa(1, 2) == m-i) &&(maximum > 1000)
104 -                   shape = shape + 1;
105 -                   pixel(shape, :) = [l m];
106 -               end
107 -           end
108 -       end
109 -   end
110
111
112 - function idy = identify(pixel_cor, img_grey)
113 -       idy = cell(3, 1); %pos, orientation, shape, and confidence
114         %cut out a box arund the coordinate
115 -       x = pixel_cor(2); y = pixel_cor(1);
116 -       b_sz = 16;
117 -       box = img_grey(y-b_sz:y+b_sz, x-b_sz:x+b_sz);
118 -       box_org = box;
119
120         %Isolate shape/Apply threshold
121 -       shape_filter = ones(2);
122 -       box_conv = conv2(box, shape_filter, 'same');
```

```
120         %Isolate shape/Apply threshold
121 -       shape_filter = ones(2);
122 -       box_conv = conv2(box, shape_filter, 'same');
123 -       figure, imshow(box_conv, [])
124 -       sz = size(box_conv);
125 -       for j = 1:sz(1)
126 -           for i = 1:sz(2)
127 -               if box_conv(j, i) < 880
128 -                   box(j, i) = 0;
129 -               else
130 -                   box(j, i) = 1;
131 -               end
132 -           end
133 -       end
134
135         %Find center of mass
136 -       total_mass  = sum(box, 'all');
137 -       yaxis_sum = sum(box, 1);
138 -       xaxis_sum = sum(box, 2);
139 -       for i = 1:sz(2)
140 -           x_weighted_sum(i) = i*yaxis_sum(i);
141 -       end
142 -       for j = 1:sz(1)
143 -           y_weighted_sum(j) = j*xaxis_sum(j);
144 -       end
145 -       y_cm = sum(y_weighted_sum)/total_mass; %Center of mass wrt to box:
146 -       x_cm = sum(x_weighted_sum)/total_mass;
147 -       y_cm_img = y_cm - b_sz + y; %Center of mass wrt to img:
148 -       x_cm_img = x_cm - b_sz + x;
149
150         %Detect orientation
151 -       box_edge = edge(box,'sobel'); %Convert to edge detect img
152 -       n = 11; horz_line = zeros(n); %nxn horizontal line
153 -       horz_line(round(n/2), :) = ones(1, n);
154 -       figure, imshow(horz_line, [])
155 -       rot_sum  = zeros(360, 1);
156 -       rot_conv_max = zeros(360, 1);
157
158 -       for deg = 0:360 %Rotate image through [-180, 180), dtheta = 1
159 -           idx = deg + 1;
160 -           box_edge_rot = imrotate(box_edge, deg, 'bilinear','crop');
161 -           rot_sum(idx) = sum(box_edge_rot, 'all');
162 -           box_edge_rot = box_edge_rot/rot_sum(idx);
```

```
        box_edge_rot = imrotate(box_edge, deg, 'bilinear','crop');
        rot_sum(idx) = sum(box_edge_rot, 'all');
        box_edge_rot = box_edge_rot/rot_sum(idx);
        box_edge_conv = conv2(box_edge_rot, horz_line, 'same');
        rot_conv_max(idx) = max(box_edge_conv, [], 'all');
        %norm_rot_conv_sum(idx) = rot_conv_sum(idx)/rot_sum(idx);
    end
    [Max, I_deg] = max(rot_conv_max);
    rot_deg = I_deg(1);
    S = std(rot_conv_max);
    box_filt_rot = imrotate(box, rot_deg);
    figure, imshow(box_edge_conv, [])
    figure, plot(rot_conv_max)
    %Detect Shape and Confidence
    bound_arr = bounds(box_filt_rot);
    length = bound_arr(1, 2) - bound_arr(1, 1)-1;
    width = bound_arr(2, 2) - bound_arr(2, 1)-1;
    area = sum(box_filt_rot, 'all');
    %area = area - area*0.08;%final term is 4noise
    possible_shape = {'circle', 'triangle', 'square', 'rectangle'};
    shape_conf = [0 0 0 0]; %indexed off of possible shape cell
    err = [0 0 0 0];
    alpha = 3;
    err(1) = (max(abs(area-(pi*(width/2)^2)), ...
        abs(area-(pi*(length/2)^2))))/area;
    err(2) = abs(area-(width*length/2))/area;
    err(3) = min(abs(area-(width^2)), abs(area-(length^2)))/area+...
        abs(width-length)*0;
    err(4) = abs(area-(width*length))-abs((width-length)*alpha);
    for i = 1:4
        shape_conf(i) = exp(-err(i));
    end

    [confidence, shape_idx] = max(shape_conf);
    shape = possible_shape(shape_idx);


    %Set final values to output
    idy{1} = [x_cm_img y_cm_img];
    idy{2} = rot_deg;
    idy{3} = shape;
    idy{4} = confidence;

```

```
        idy{3} = shape;
        idy{4} = confidence;

        %Plot:
        triangle = imread('Lab3_triangle.PNG');
        circle = imread('Lab3_circle.PNG');
        square = imread('Lab3_square.png');
        rectangle = imread('Lab3_rectangle.jpg');
        shape_img_idx = {circle triangle square rectangle};
        shape_img = cell2mat(shape_img_idx(shape_idx));
        figure,
        subplot(2,3,1)
        imshow(box_org, []);
        title('Orignal Box Cut Out')
        subplot(2,3,2)
        imshow(box, []);
        title('Conv+Thereshold Filtered Shape')
        subplot(2,3,3)
        imshow(imrotate(box_edge, 0), []);
        title('Edge Filtered Shape')
        subplot(2,3,4)
        imshow(box_filt_rot, []);
        title('Orientated Shape')
        subplot(2,3,5)
        imshow(box_filt_rot)
        hold on
        xline(bound_arr(2), 'g', 'LineWidth', 4);
        xline(bound_arr(4), 'g', 'LineWidth', 4);
        yline(bound_arr(3), 'g', 'LineWidth', 4);
        yline(bound_arr(1), 'g', 'LineWidth', 4);
        title('Bounded, and Orientated Shape')
        subplot(2,3,6)
        imshow(shape_img, []);
        title(char(shape))
    end


function scale = get_scale(index)

        scale = zeros(8,2);

        for ix = 1:6
            scale(ix,2) = index(ix+1,2) - index(ix,2);
```

```
240
241 -     for ix = 1:6
242 -         scale(ix,2) = index(ix+1,2) - index(ix,2);
243 -         scale(ix,2) = 2/scale(ix,2);
244 -     end
245
246 -     for iy = 1:8
247 -         scale(iy,1) = index(iy+1,1) - index(iy,1);
248 -         scale(iy,1) = 2/scale(iy,1);
249 -     end
250
251
252 - end
253
254
255   function [locx, locy] = get_length(img, x, y)
256 -     [numRows,numCols] = size(img);
257 -     ycounter = 0;
258 -     xcounter = 0;
259 -     i = 10;
260 -     index = zeros(9,2);
261
262 -     while i <= numCols
263 -         if img(10,i) > 115
264 -             ycounter = ycounter + 1;
265 -             index(ycounter, 1) = i;
266 -             i = i + 3;
267 -         end
268 -         i = i + 1;
269 -     end
270 -     i = 10;
271 -     while i <= numRows
272 -         if img(i,15) > 115
273 -             xcounter = xcounter + 1;
274 -             index(xcounter, 2) = i;
275 -             i = i + 3;
276 -         end
277 -         i = i + 1;
278 -     end
279
280 -     ycounter = 0;
281 -     xcounter = 0;
282 -     scale = get_scale(index);
```

```
280 -       ycounter = 0;
281 -       xcounter = 0;
282 -       scale = get_scale(index);
283 -       i=7;
284 -
285 -       while x < index(i, 2)
286 -           i = i - 1;
287 -           xcounter = xcounter + 2;
288 -       end
289 -       i=1;
290 -       while y > index(i, 1)
291 -           ycounter = ycounter + 2;
292 -           i = i + 1;
293 -       end
294 -
295 -       ycounter = ycounter - 10;
296 -
297 -       locx = 2 + xcounter + scale(7-xcounter/2,2)*(index(7-xcounter/2, 2)- x);
298 -       locy = ycounter + scale(5+ycounter/2, 1)*(y-index(5+ycounter/2, 1));
299 -
300 -   end
```