IVR report 2010

Sean Wilson and Daniel Wren
0831408 0789266

Abstract
This report covers an investigation into an algorithm designed to navigate a Khepera robot around a simple maze from one point to another without colliding with any objects. The algorithm was written in matlab, which has several useful built-in functions that have aided the development of the visual recognition parts of the program. The real world investigations were captured through a simple web camera, taking snapshots of the scene several times throughout the task. The algorithm designed, modified and cleared up the images to aid object isolation and help calculate the correct path through the maze to avoid collision and arrive at the final destination accurately. The investigations showed that the method used was very successful and quick to run. Although the pre-set paths were geometrically accurate, it was observed that any slight bumps or tilt in the maze floor could throw the Khepera robot slightly off-course, and would only correct its course once it had finished running the pre-set path. This meant that until the point of visual servoing, the robot could collide with walls and objects without any detection or correction. One improvement discussed is that of visual servoing the robot as it moves around the maze. This would resolve the collision detection issue, but would increase the run time of the program. This report was produced by Sean Wilson and in part by Daniel Wren. The coding for the algorithm was produced equally by both Sean Wilson and Daniel Wren.

1. The Task
The objective of this task is to design and implement an algorithm that can navigate a Khepera robot via camera snapshots through a maze from a start point to an end point, without colliding with any objects or walls. The robot cannot use any of its sensors to aid in collision-detection. This algorithm must work in a real world and webot world scenario. The world will be contained in white space to prevent any confusion to image capturing. The world will be captured by a web camera, from an unknown angle and origin. In this world there will be a black maze, consisting of several walls and two points, with interchangeable start and finish points. There will also be two objects placed in the maze, which need to be avoided by the robot whilst navigating the maze. A Khepera robot will be placed on the designated start point and, aided only by the algorithm which takes images of the world through a camera, navigate through the maze. It must avoid all walls and objects on its route to the end point. The robot must end within a faint blue line, which circles the end point.

2. Methodology
The method designed to complete the task made use of several functions provided by the University of Edinburgh to aid in vision capturing and manipulation. The code breakdown chart in the appendix clearly shows the provided code used within the program. The program will run from a function called ‘main.m’. There will be six main sections in this program which are explained below. At the top of the main function, two paths are opened to connect matlab to the robot and a file containing drive commands to be used with the Khepera robot.

3. Image Capturing
The process must be ran in its entirety for every image taken by the webcam, in order to ensure that each image is displayed in the same format for the purposes of image manipulation and object detection. A new function will be created called ‘imageCapture.m’ which will run the steps outlined below to capture an image and format it. ‘imageCapture.m’ will take a string, used to save the image to file and 4 numbers used as figure numbers to display the colour image, the raw and smoothed histograms and the final cleaned black and white image. If any of these numbers are set to 0, no figure will be displayed. Three images are taken to detect the following elements: the maze, the unblocked path and the starting point of the robot. Images are taken by a standard web cam, with a dimension of 640 x 480 pixels in 256 colour, via a provided function called ‘liveimagejpg.m’, which takes a filename and uses unix commands to take one shot with the webcam then saves and loads it.

3.1 Thresholding

Each image will be turned into a black and white picture, to remove shadows and glare, using a method called thresholding. The background of the box and maze will be turned to black; the objects and maze walls will be turned to white. This is achieved by setting a value, known as the threshold value, for pixel colour. Everything above the threshold is set to one value, and everything below to another. In this case, black or white, 0 or 1 respectively. This is the first step in image isolation.

The web camera adapts to the light levels in the box; this can be affected by glare from the objects or by brighter and darker areas and produces some very dark or bright images. To ensure that the shadows and glare do not effect which parts are turned into black and white, the level used as the threshold value needs to be adaptive. To do this, several provided functions were used. A function called 'dohist.m' was used to draw a histogram of the colour levels in the image. It ranges from 0 to 255 on the x-axis (representing each colour), and the y-axis represents the amount of pixels with that colour value.

3.2 Smoothing

As can be seen in Figure 2 below, histograms have many small peaks. In order for this histogram to be used to find a suitable threshold value, it must first be smoothed. This will be done with another provided function called 'smooth.m'. This returns the smoothed histogram, with smoothed peaks as seen in Figure 4. This histogram can then be used by 'findthresh.m' to return a suitable thresholding value. The value will then be passed, along with the image, to a function called 'dothresh.m' which applies the threshold to the image pixels, returning the black and white image required.

3.3 Clean-Up

The threshold level cannot remove all shadowing or glare; the shadows can sometimes merge objects, and glare can disjoint parts. To help recreate the image and prevent image distortion, the function 'cleanup.m' is used. This function uses built-in matlab functions called erode and dilate. These help to clean up any breaks caused when the threshold level is applied.

3.4 Homogaphy

The images are taken from an unknown angle and origin, because of this each image needs to be homographs and set to a specific origin before detection can be undertaken. Homography requires a specified number of points from an image, which will be used to manipulate an image by stretching these points to a set of predefined projection points. The predefined projection points are calculated from the figure supplied, which contains co-ordinates of points on the maze. By calculating the x and y ratio between the paper version and a 640 x 480 pixel image, the projection points can be calculated in x-axis and y-axis pixel values. The final image should look like a birds-eye view of the maze, with start and end point on the left-hand side. The image of the maze has 4 large external circles to be used as reference points to calculate homography. In this design, it is assumed that the camera, once the first image is captured, will not move again. Because of this, the homography value given by ‘esthomog.m’ and the rotation check only need to be carried out once. The first image will be of the maze only, therefore it is sensible to use this image to do the calculations to reduce the potential of labelling other objects as the external circles.

3.5 Find Centroids

To obtain accurate coordinates for these circles in the images, the centre points of each must be found. Before the centroids can be calculated, each circle must first be extracted from the image. This will be done in a function called ‘findCentroids.m’ which uses a provided function called ‘mybwlabel.m’ and built-in function regenProp. ‘findCentroids.m’ takes an image and three integers: an upper bound, a lower bound and a figure number. The upper and lower bounds will be used in ‘mybwlabel.m’, which labels any connected areas in the image within the bounds. The x and y values of these centroids are returned along with their size. Through trial and error, moving the distance and angle of the camera results in sensible bounds being found for the circles. The values of the labelled areas are then passed to regenProp to find the centre points.

3.6 Order Centroids

One problem that may arise if the centroids are left unaltered after being returned from findCentroids, is the order of them within the matrix. The areas will not always be found and labelled in the same order, which could cause problems in 'esthomoh.m'. To resolve this, a function called 'orderCentroids.m' should be created. This uses the Euclidean distance to determine which projection centroid the calculated centroids from the image are closest to. The centroids from the image are then ordered from left to right and top to bottom, the same as in the projection centroid matrix.

3.7 Esthomog and Homographies

Once the circles' centroids have been calculated and ordered, an estimated value to move each pixel by must be worked out. This will be done by a provided function called 'esthomog.m', which takes both sets of points and calculates the value to be used in homography. The value returned by 'esthomog.m' will be used by 'homographise.m', which is a modified version of the supplied function, remap. This is where the actual homography of the image takes place and is saved. It will take both sets of centroids, esthomog's value and the image.

3.8 Rotatecheck and imRotate

As explained in the task description, the camera can have any origin. This means that the map could be upside-down or sideways. This effects how the homographised image is displayed, and the start and finish co-ordinates. To overcome this potential problem, each image will be checked for the homographised image orientation and rotated so that the start and end points are always located on the left side of the image. The centroids of the start and finish locations must be found using the same function as that used to find the centroids of the external circles, but with smaller bounds. 'Rotatecheck.m' will take the centroids of these inner circles and run a simple if loop, which checks which half of the image both centroids are located in and then rotates anti- clockwise accordingly. For example, if both centroids are in the top half of the image (both have y value smaller than 240) then the image is rotated 90 degrees and then overwrites the un-rotated image.

4. Object Detection

Once all three images have been homographised and rotated, object detection can be carried out. In each image a new type of object is added. Knowing this, each time a new image is captured, the previous image can be treated as its background. Subtracting the image matrix of the background image to the new image and taking the absolute value (to remove any negative values that occur), will give a new image matrix which will contain the foreground; in this case, the new object.

4.1 Block detection

The blocks can be detected by doing the above and taking Image 1 (just the maze) as the background and Image 2 (the maze and blocks) as the foreground image. Once the resulting matrix is returned, this image will display only the two white blocks.

4.2 Robot detection

The robot can be detected by doing the above and taking Image 2 (the maze and blocks) as the background and Image 3 (the maze, blocks and robot) as the foreground. Once the resulting matrix is returned, this image will display one large circle.

4.3 Path, Start and Finish Point Detection

The two blocks placed on the maze conceal two out of the three possible routes for the Khepera robot. The robot can also be positioned to start on either point inside the maze. This needs to be identified as it affects which way the robot will navigate the path. As there are three paths and two start points, there are six possible routes the robot can take.

To discover which paths are blocked, using the image isolating the blocks, findCentroids can be applied and will return the centroids of the two blocks. 'findFreePath.m' takes the co-ordinate matrix of the two blocks and carries out a simple if loop to determine within which bounds the two blocks sit. The variable 'paths' will contain the vector matrix with values 1,2 and 3. There will be three bounds that correspond to: path one (closest to the left); path two (the middle), and path three (on the right). If a block's centroid x-value is in one of the bounds corresponding to a path, the number of that path will change to a zero value in the 'paths' variable. As there are two blocks, two values will be changed to zero in the 'paths' variable and returning the max value will reveal the free path. This value is then printed to the command line to inform the user of which path is free.

Finding the robot start point is similar to the algorithm used to find the free path. The top small circle is taken as start point one and the lower as two. A variable called 'roboLoc' holding a vector matrix with values 1 and 2 is instantiated. The y-value of the robot is used in an if loop to determine whether it is smaller than 240 (in the top half of the image) or larger than 240 (in the lower half). If the robot's y-value is smaller than 240, value 2 is changed to zero in 'roboLoc', or value 1 to zero if above 240. The max of this variable will then return the robots start point. The coordinates of the other point are also returned, as these will be used for visual servoing.

5. Moving Through the Maze

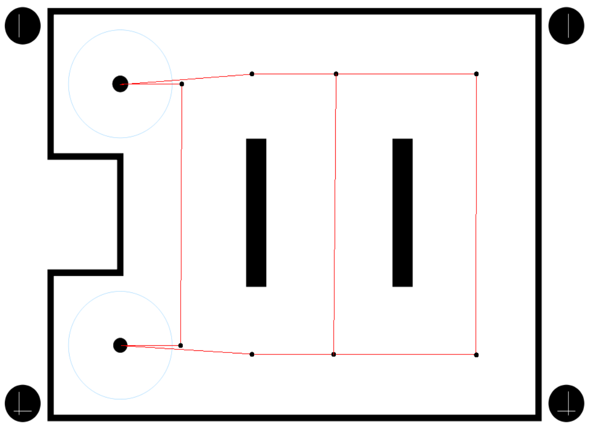
Through repeated test runs between two points on the maze, the average time to travel one pixel can be determined. This is calculated using the pixel distance between the points and recording the time taken to cover the distance several times and taking the average. The time taken to rotate one degree must also be calculated, this is done in the same manor as above but the time to rotate 360 degrees is recorded instead. This is done in both directions to make sure that there are no discrepancies.

The distances between certain points in the maze must be calculated into pixel value, in the same way that the co-ordinates were measured for the projection points. These certain points are points where the robot can/must change direction to carry out a route. The points are:

- The two start points.
- Both ends of the three vertical paths (six points). From examining the maze closely it can be seen that the y-axis value of the start points are different to the mid-point between the outer wall and the ends of the inner wall. This means that the Khepera robot could possibly skim the edges of the inner walls unless rotated.
- To compensate for this possible issue, a further two points will be added: the mid-point between the first inner wall and the outer wall, and the y-axis value of the four further points will be changed to be in line with these values, (see Figure 3).

In total there are 10 destination points on the maze as shown in the diagram below.

The 'doRoute.m' function will take the start point given by 'findStartLocation.m' and the free path number returned by 'findFreePath.m', Via several if loops, the correct pre-planned route will be selected and drive commands in this route will be executed. Each route will, using the calculated times it take to drive between each point, send a drive command, with wheel speeds followed by a pause command of the correct times. For the pre-planned route commands, see the appendix code for 'doRoute.m'. The end of every route should stop the Khepera robot exactly on the end point's centroid, although due to gear slippage and bumps in the box floor this may not be the case. The last drive command will be a rotate to ensure that the robot is facing exactly 90 degrees to the left to prepare for visual servoing.



6. Visual Servoing

The last part of the algorithm is a step to ensure that the robot finishes within the faint blue circle, with the centroid of the robot on the centroid of the end point. A new image is taken of the scene and the robot is isolated in the same way as before. The centroid of the robot is recalculated. The distance and angle between the centroids is calculated via a trigonometric equation. This is possible as the robot is aligned to a 90 degree angle after each path. If the centroids are within a 10 pixel square of each other then they are considered close enough. If they are out of this range, then the distance and angle are multiplied by the time taken to carry out the movement necessary to align the centroids. Once finished, the robot is rotated to face 90 degrees left, in case the process requires reiteration. This is then sent to the robot via the send_command with the calculated pauses in-between. This process will be carried out in a while loop, taking pictures after each visual servoing, and analysing the distance until they are within the set range.

7. The Webot World

The webot world uses the same method as the real world above with three changes. Due to these changes the webot world algorithm will be saved in its own main function called ‘main_webots.m’. The changes are:

1. The image capturing is executed via a function called ‘take_snap.m’, which saves the image to the temporary folder on the computer. As the webot world has no shadows or glare, no image clean-up needs to take place.
2. The images are half the size, so any function that is used in both worlds needs to be given a value that changes the sizes used in either world.
3. The distances travelled by the robot change in the webot world, and the speeds are different. A separate function for pre-set paths in the webot world must be created.

8. Predictions

The algorithm above should work quickly and accurately. The only loss in accuracy may occur due to gear slippage, or bumps in the box base. The further the route, the more prone to these inaccuracies the robot will be. Despite taking these external effects into account, the robot should not deviate significantly from the route, and it should not clip the walls. The robot should always finish precisely on the finish point due to the visual servoing.

9. Results

9.1 Real World

24 test were conducted to monitor how four variables would affect the results of the algorithm. Image capturing was not consistently accurate, but as is demonstrated in the sample experiment figures, the cleaning of the images helps to reduce a lot of shadowing. The recorded tests are those which captured usable images to carry out the route. Some images were badly shadowed, which lead to homography failure. As illustrated in the results table below, wall clipping occurred only when path three was the free path. Failure to finish also only happened when running route three. The start and end points had no effect on the results or accuracy of the robot. The maze orientation did not cause any problems and was always correctly rotated with the small internal circles on the left. The camera orientation occasionally resulted in issues to do with finding the thresholding level, with the furthest end of the image becoming white due to shadowing, meaning all four external circles could not be found and homography failure. Each time a new image was captured for servoing the shadowing of the robot would sometimes effect the location of the centroid, causing the robot to carry out the servoing more times than necessary. When the Khepera robot over-ran the wall it was always due to bad servoing. Every time the robot over-ran the wall by a large amount, the image isolation failed and an error would occur.

9.2 Webot World

Image capture was accurate from all angles and orientations because lighting,and glare were not an issue. This resulted in successful homography and object recognition on every test. At no point was the wrong start point or path calculated.

Due to the differences in the webot world, the rotation time and distance travel time were always different. This meant that the approach used did not work consistently. If the times were only slightly incorrect, the robot would still finish in the circle as planned; if differences in timing were large, then the robot could fail and not end up in the finish point at all.

Start Point	**Free Path**	**Camera Angle est.**	**Orientation**	**No. Walls Clipped**	**Walls gone over**	**Finish Correct**	**Number of Servo**
1	1	20	Points at top	0	0	Yes	0
2	1	20	Points at top	0	0	Yes	0
1	2	20	Points at top	0	0	Yes	1
2	2	20	Points at top	0	0	Yes	0
1	3	20	Points at top	1	0	Yes	2
2	3	20	Points at top	0	0	Yes	5
1	1	20	Points at top	0	0	Yes	2
2	1	20	Points at top	0	0	Yes	0
1	2	20	Points at top	0	0	Yes	0
2	2	20	Points at top	0	0	Yes	0
1	3	20	Points at top	0	1	No	3
2	3	20	Points at top	0	0	Yes	1
1	1	45	Points at bot	0	0	Yes	0
2	1	45	Points at bot	0	0	Yes	2
1	2	45	Points at bot	0	0	Yes	3
2	2	45	Points at bot	0	0	Yes	2
1	3	45	Points at bot	1	1	No	0
2	3	45	Points at bot	0	0	Yes	1
1	1	45	Points at bot	0	0	Yes	0
2	1	45	Points at bot	0	0	Yes	1
1	2	45	Points at bot	0	0	Yes	2
2	2	45	Points at bot	0	0	Yes	0
1	3	45	Points at bot	1	0	Yes	0
2	3	45	Points at bot	0	2	No	1

10. Discussion

In general, the algorithm ran well. The robot ended at the finish point during the majority of experiments. An issue that affected our experiment significantly was the light fluctuation and shadows captured by the web cam. If a corner was shadowed badly, the exterior circles were sometimes not captured, causing the homography to fail. Also, if glare was captured from the maze walls, breaks could occur in the wall (as seen in figure 7). Sometimes these areas of connected pixels were large enough to be within the bounds and were calculated as a circle, distorting the homography and skewing the picture. Glare off the shiny robot or blocks could also cause the objects to appear smaller than they were (as seen if figures 7, 8 and 9), resulting in them not being identified as objects. If the lamp and camera were angled in the same direction, this prevented any glare of the objects or maze.

At the larger angle of around 45 degrees, shadows would increase the size of the robot and create an oval-like shape for the robot, affecting the calculated centroid (as seen in figure 12). On the computer it would look like the robot centroid was sat on the finish point, but in reality the robot would be off by a few millimetres. This could also cause the robot to appear to be over the wall of the maze. As image isolation takes one image away from the other, when the robot was on top of the wall, the section over-hanging the wall was removed and the robot's area decreased. Sometimes it was decreased beyond the bounds of detection and an error occurred, sometimes it gave an incorrect centroid, making the robot move further than necessary. To compensate for this, it was best to have the lamp overhead and facing slightly away from the webcam. This reduced glare and the majority of shadows.

Another reason for the robot sometimes clipping the walls of the maze was due to imperfections with the box. Any bump could potentially move the wheels off-line.. However, if the algorithm were to be modified, it would make the collision detection much more accurate and adaptable. For example, if images were captured and visual servoing was used to navigate the robot to the set points one at a time along the route to the finish point.

A problem found with the image servoing used in the algorithm was that it always assumed that the robot was facing 90 degrees to the left. Although this estimation was only wrong by a few degrees,, it had the potential to cause the image servoing calculations to be inaccurate.

Distances of the webcam were also investigated, but not recorded in the results table as it was found that the distances of the webcam had little impact on the results. The only issue detected, was that the bounds of object recognition had to be expanded to account for smaller objects when further away, and larger objects when close up.

To further increase accuracy of image capturing, 'esthomog' could be run for each image, to ensure that any camera movement between the images does not effect the homography. Using the same value from 'esthomog' for an image where the camera or maze is in a slightly different position will give an incorrect homographised image.

The webot world algorithm would benefit from using visual servoing at all points along the path, as the timing of rotation and distance travelled change frequently.

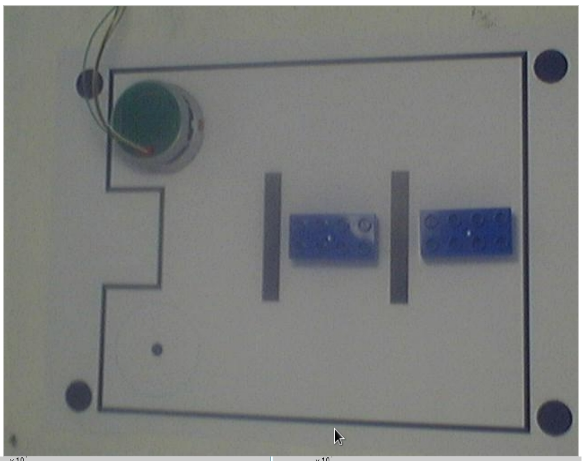
11. Conclusion

In conclusion, the general accuracy in the real world was very good, with few major collisions. The webot world did not work as well with the same approach. If repeated, the process would benefit greatly from constant visual servoing and image capturing to monitor the progress of the robot. All collisions were due to problems with the quality of the box, not slipping of the gears or miscalculated geometry. Although this was not the most effective method to use, it was more rapid than visual servoing throughout the robot movement. If the size of the maze was expanded, then pre-set movements would become less accurate. However, for the size of the maze in this test, the level of accuracy proved to be adequate for guiding the robot around, often not needing visual servoing at the end.

13. Appendix

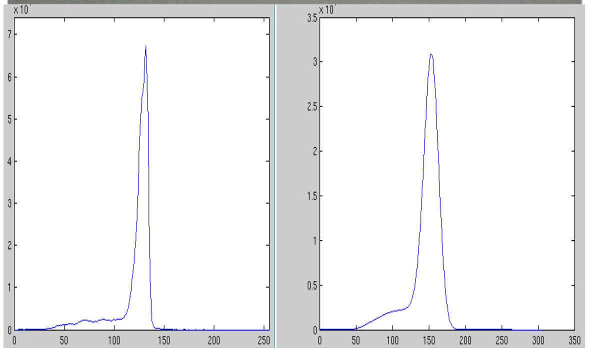
<u>Figure 1</u>

This figure is the raw image captured by the webcam with all objects in place. Glare can be seen on the Duplo block in path two. This will reduce the area of the block once thresholding is applied. This can be see in figures 7 – 9.



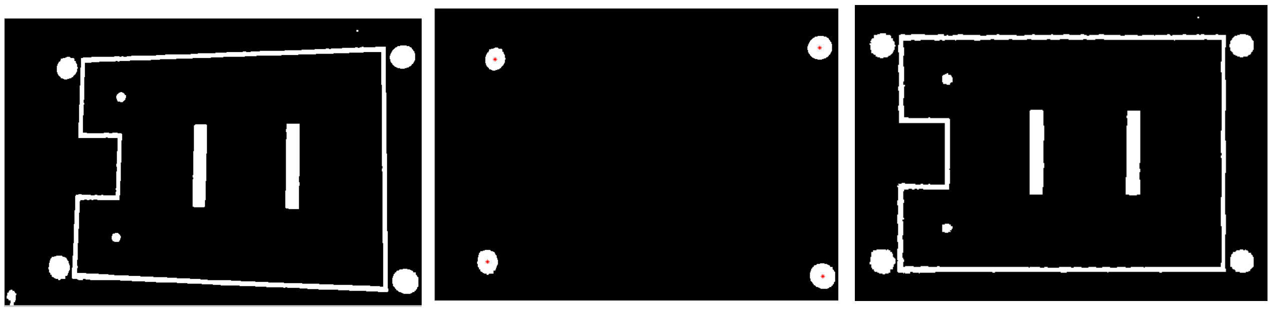
<u>Figure 2 and 3</u>

Figures 2 shows the histogram for image three above. Once smoothing is applied, the small peaks are smoothed out and the larger peaks become smoother, as can be seen in figure 3.



<u>Figure 4, 5 and 6</u>

The figures below show the transformation that takes place between the raw threshold picture and once homography is applied. Figure 5 shows the circles used as reference points along with their calculated centroids.



<u>Figure 7, 8 and 9</u>

The figures below show the stages of object recognition with the blocks. As can be seen in figure 8, the subtraction of the background image is not so clear if the maze is moved slightly between photographs. The glare of the block can be seen in all three images.

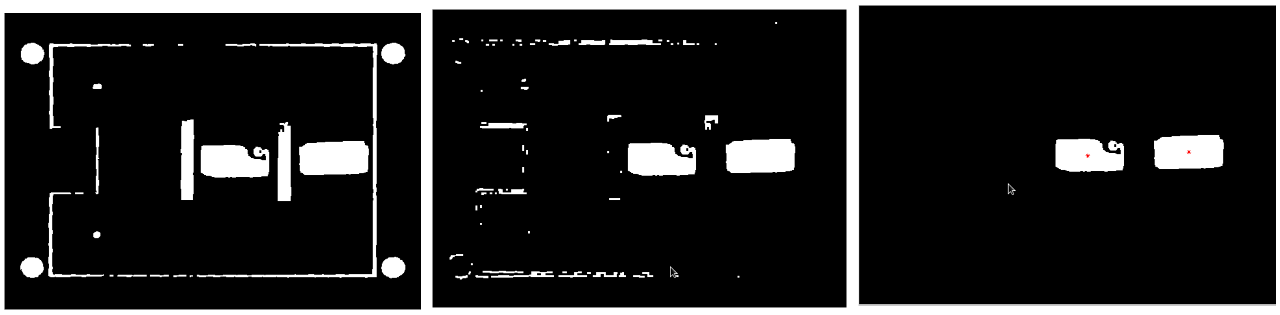


Figure 10, 11 and 12

The figures below show the stages of object recognition with the robot. As can be seen in figure 1, the robot is placed directly over the centroid, yet the robots centroid is calculated behind the point. This is due to the shadow and angle of the camera.

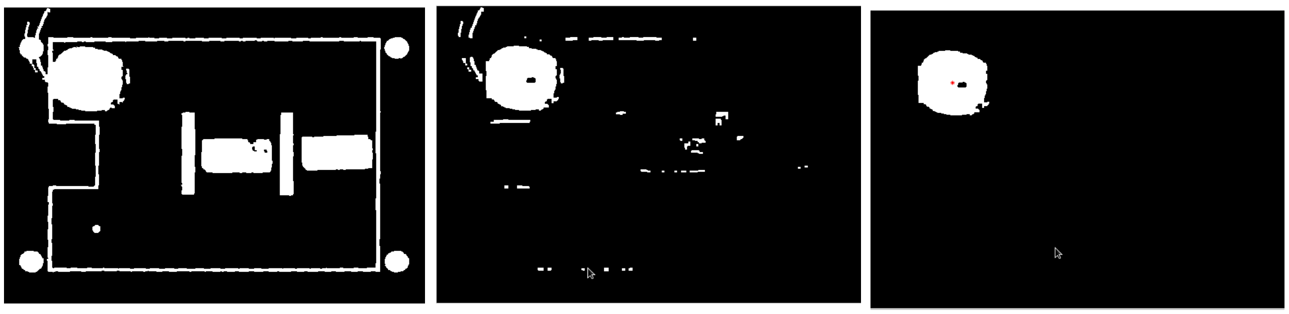


Figure 13

This figure shows the isolated robot and the centroid of the robot and the position of the end point. The robot's calculated centroid is slightly off, even though, from looking at figure 14 below, it is clear that the robot is directly over the finish point. This is because of the camera angle and shadows.

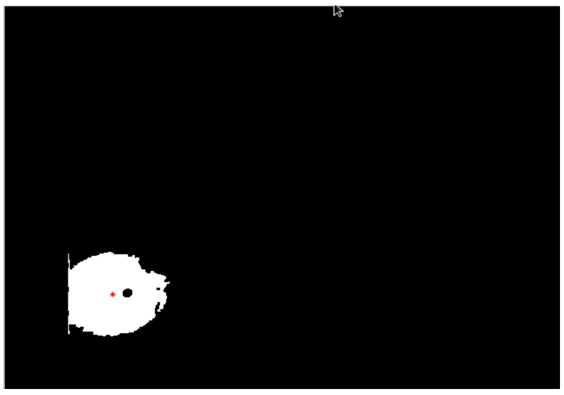
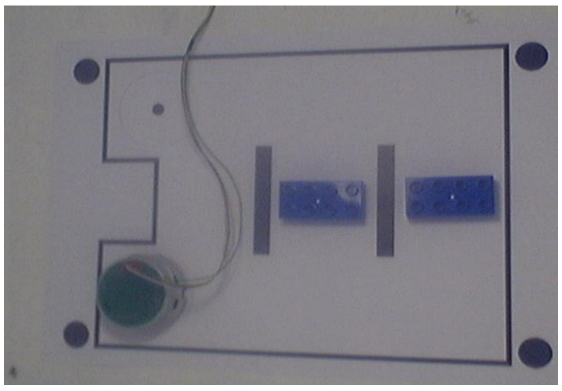
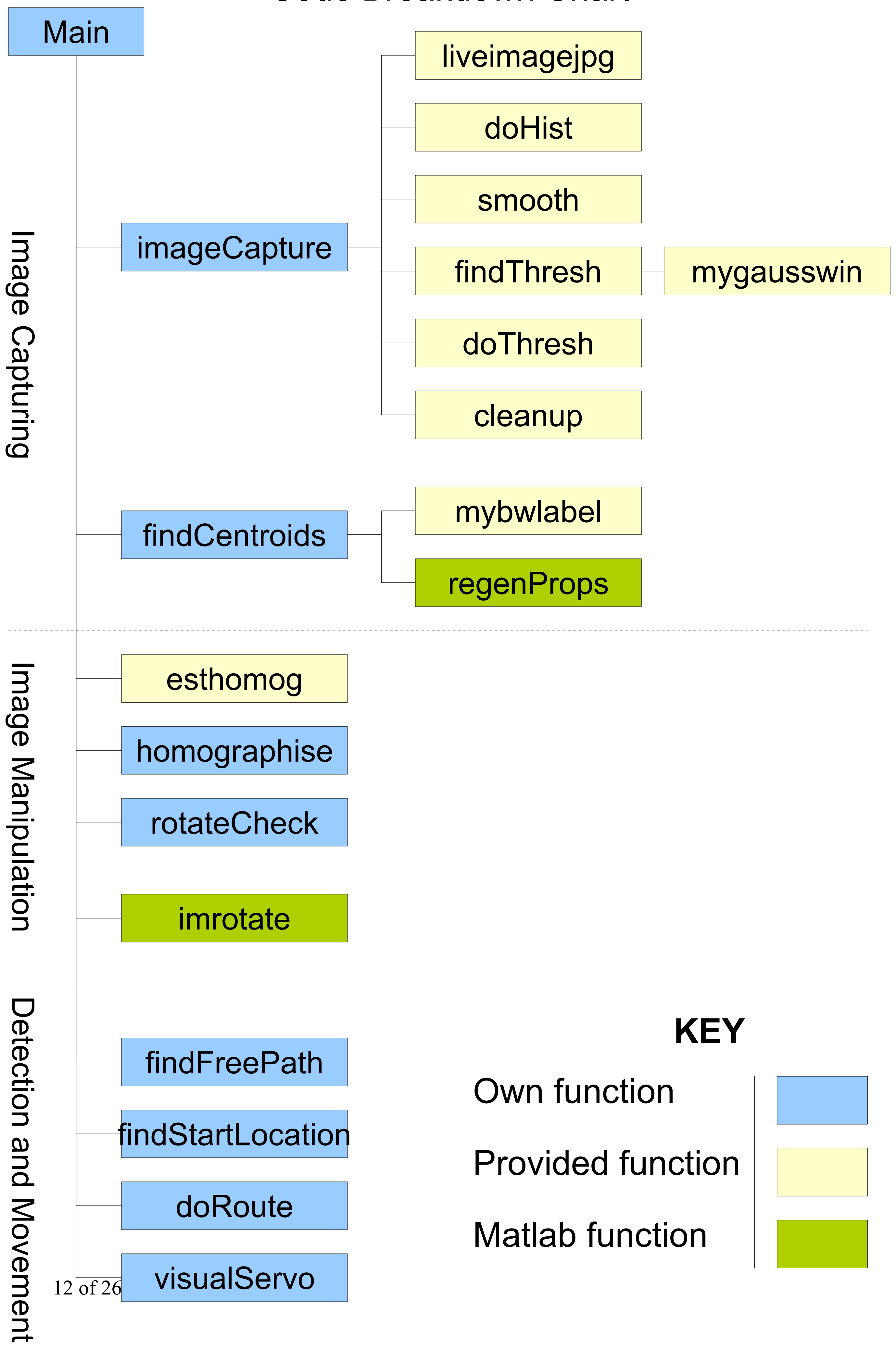


Figure 14

This figure shows the image taken to be used in visual servoing. As can be seen in the image, the robot has already positioned itself perfectly in the centre if the faint blue circle.



Code Breakdown Chart



Main.m

```
%-----------------------Connect to Robot----------------------------------

path(path,'/afs/inf.ed.ac.uk/user/s08/s0831408/Thirdyear/IVR/Matlab/Webot_interf
ace/tcp_udp_ip');
path(path,'/afs/inf.ed.ac.uk/user/s08/s0831408/Thirdyear/IVR/Matlab/Webot_interf
ace/robot_commands');

%-------------------------------------------------------------------------
%-------------------------Image Capturing---------------------------------

%Use this image to help reduce the effects of shadowing
%blankImage = imageCapture('Blank',0,0,0,1);
%prompt = input('Is the maze in place? Press return to continue.', 's');

cleanimage1 = imageCapture('Maze',1,2,3,4);
%cleanimage1 = abs(cleanimage1 - blankImage);
prompt = input('Are the obstacles in place? Press return to continue.', 's');

cleanimage2 = imageCapture('Scene',1,2,3,4);
prompt = input('Is the robot in place? Press return to continue.', 's');

cleanimage3 = imageCapture('Robot',1,2,3,4);
prompt = input('Good to go? Press return to continue.', 's');

%-------------------------------------------------------------------------
%------------------------Calculate Homography-----------------------------

[centroids bigSize] = findCentroids(cleanimage1, 450, 1500, 0, 5);    %Points
from image
projections = [43 67; 597 67; 43 413; 597 413];            %Move to points

orderedCentroids = orderCentroids(centroids, projections);

estimation = esthomog(projections,orderedCentroids,4);
rawHomography = homographise(estimation,cleanimage1,6);

%-------------------------------------------------------------------------
%------------------------Calculate Image Rotation-------------------------

[smallCircleCentroids smallSize] = findCentroids(rawHomography, 100, 250, 0, 7);

imsize = size(cleanimage1);
rotateDeg = rotatecheck(smallCircleCentroids,imsize);

mazeHomography = imrotate(rawHomography,rotateDeg);
figure(7)
imshow(mazeHomography);

%-------------------------------------------------------------------------
%--------------------------Detect Free Path-------------------------------

duploHomography = homographise(estimation,cleanimage2,8);
duploHomography = imrotate(duploHomography,rotateDeg);
figure(8)
imshow(duploHomography);

onlyDuplo = abs(duploHomography - mazeHomography);
```

```
cleanDuplo = cleanup(onlyDuplo,1,1,9);

[duplo_centroids duploSize] = findCentroids(cleanDuplo, 4000, 7250, 0, 10);

freePath = findFreePath(duplo_centroids,1); %world = 1 (real world khepara)

%---------------------------------------------------------------------------
%------------------------Find Robot------------------------------------------

roboHomography = homographise(estimation,cleanimage3,11);
roboHomography = imrotate(roboHomography,rotateDeg);
figure(11)
imshow(roboHomography);

onlyRobot = abs(roboHomography - duploHomography);
cleanRobot = cleanup(onlyRobot,1,1,12);

[start_centroid roboSize] = findCentroids(cleanRobot, 6000, 11500, 0, 13);

%---------------------------------------------------------------------------
%-------------------------Plan Route------------------------------------------

%Find starting point

[start finishLoc] = findStartLocation(start_centroid, smallCircleCentroids, 1);


%---------------------------------------------------------------------------
%---------------------------Execute route-------------------------------------

doRoute(start, freePath);

%---------------------------------------------------------------------------
%---------------------------Servoing--------------------------------------
servoImage = imageCapture('Servo',19,0,0,14);
servoHomography = homographise(estimation,servoImage,0);
servoHomography = imrotate(servoHomography,rotateDeg);
onlyServoRobot = abs(servoHomography - duploHomography);

%onlyServoRobot = cleanup(onlyServoRobot,1,1,15); %further clean up
figure(15)
imshow(onlyServoRobot);

[roboLoc finishSize] = findCentroids(onlyServoRobot, 5000, 11500, 0, 16);

while (abs(roboLoc(1) - finishLoc(2)) > 4) && (abs(roboLoc(2) - finishLoc(1)) >
4)

    servoImage = imageCapture('Servo',0,0,0,17);
    servoHomography = homographise(estimation,servoImage,0);
    servoHomography = imrotate(servoHomography,rotateDeg);
    onlyServoRobot = abs(servoHomography - duploHomography);

    onlyServoRobot = cleanup(onlyServoRobot,1,0,17); %further clean up

    [roboLoc finishSize] = findCentroids(onlyServoRobot, 7000, 11500, 0, 18);

    prompt = input('Do you want to continue servoing? Press return to
continue.', 's');
```

```
    visualServo(roboLoc, finishLoc);
end
disp('Visual Servoing Complete')
```

imageCapture.m

```
function cleanedImage = imageCapture(filename, fig1, fig2, fig3, fig4)

    image = liveimagejpg(filename);
    if fig1 > 0
      figure(fig1)
      imshow(image)
    end
    hist = doHist(image, fig2);
    smoothedHist = smooth(hist);
    threshold = findThresh(smoothedHist, 4, fig3);
    threshedImage = doThresh(image,threshold,0);
    cleanedImage = cleanup(threshedImage,1,1,fig4);
end
```

findCentroids.m

```
function [centroids, size] = findCentroids(image, lower, upper, num, show)

[cc,size,points] = mybwlabel(image, lower, upper, num);

% Find centroids of the large circles
s = regionprops(cc, 'centroid');
centroids = cat(1, s.Centroid);

% Show centroids of objects within lower and upper bounds
if show > 0
    figure(show)
    imshow(cc)
    hold on
    plot(centroids(:,1), centroids(:,2), 'r*') %(in red)
    hold off
end
end
```

orderCentroids.m

```matlab
function orderedC = orderCentroids(c, p)

%Point a distances
A(1) = sqrt((c(1) - p(1))^2 + (c(5) - p(5))^2);
A(2) = sqrt((c(1) - p(2))^2 + (c(5) - p(6))^2);
A(3) = sqrt((c(1) - p(3))^2 + (c(5) - p(7))^2);
A(4) = sqrt((c(1) - p(4))^2 + (c(5) - p(8))^2);

w = 1;
for i = 2 : 4
    if A(i) < A(w)
        w = i;
    end
end

if w == 1
    topleft = c([1 5]);
end
if w == 2
    topleft = c([2 6]);
end
if w == 3
    topleft = c([3 7]);
end
if w == 4
    topleft = c([4 8]);
end
%Point b distances
B(1) = sqrt((c(2) - p(1))^2 + (c(6) - p(5))^2);
B(2) = sqrt((c(2) - p(2))^2 + (c(6) - p(6))^2);
B(3) = sqrt((c(2) - p(3))^2 + (c(6) - p(7))^2);
B(4) = sqrt((c(2) - p(4))^2 + (c(6) - p(8))^2);

w = 1;
for i = 2 : 4
    if B(i) < B(w)
        w = i;
    end
end
if w == 1
    topright = c([1 5]);
end
if w == 2
    topright = c([2 6]);
end
if w == 3
    topright = c([3 7]);
end
if w == 4
    topright = c([4 8]);
end
%Point c distances
C(1) = sqrt((c(3) - p(1))^2 + (c(7) - p(5))^2);
C(2) = sqrt((c(3) - p(2))^2 + (c(7) - p(6))^2);
C(3) = sqrt((c(3) - p(3))^2 + (c(7) - p(7))^2);
C(4) = sqrt((c(3) - p(4))^2 + (c(7) - p(8))^2);
```

```
w = 1;
for i = 2 : 4
    if C(i) < C(w)
        w = i;
    end
end

if w == 1
    bottomleft = c([1 5]);
end
if w == 2
    bottomleft = c([2 6]);
end
if w == 3
    bottomleft = c([3 7]);
end
if w == 4
    bottomleft = c([4 8]);
end
%Point d distances
D(1) = sqrt((c(4) - p(1))^2 + (c(8) - p(5))^2);
D(2) = sqrt((c(4) - p(2))^2 + (c(8) - p(6))^2);
D(3) = sqrt((c(4) - p(3))^2 + (c(8) - p(7))^2);
D(4) = sqrt((c(4) - p(4))^2 + (c(8) - p(8))^2);

w = 1;
for i = 2 : 4
    if D(i) < D(w)
        w = i;
    end
end
if w == 1
    bottomright = c([1 5]);
end
if w == 2
    bottomright = c([2 6]);
end
if w == 3
    bottomright = c([3 7]);
end
if w == 4
    bottomright = c([4 8]);
end

orderedC = ([topleft; topright; bottomleft; bottomright]);

end
```

homopgraphise.m

```
function fliphomog = homographise(projection,image,show)
    [IR,IC,D]=size(image);

homog=zeros(IC,IR);   % destination image
v=zeros(3,1);

% loop over all pixels in the destination image, finding
% corresponding pixel in source image
for r = 1 : IC
    for c = 1 : IR
        v=projection*[r,c,1]';        % project destination pixel into source
        x=round(v(1)/v(3));  % undo projective scaling and round to nearest
integer
        y=round(v(2)/v(3));
        if (y >= 1) && (y <= IR) && (x >= 1) && (x <= IC)
          homog(r,c,:)=image(y,x,:);   % transfer colour
        end
    end
end

fliphomog = flipdim(homog,2);

if show > 0
    figure(show)
    imshow(fliphomog) % /255 if colour image
end

% save transfered image
imwrite(uint8(fliphomog),'Homographised.jpg','jpg');
```

rotateCheck.m

```
function rotateDeg = rotatecheck(sc,num)

    y = num(1)/2;
    x = num(2)/2;

    %Detect the origin of image via start point centroid location
    %And rotate image so that start points are on on the left

    if (sc(3) < y) && (sc(4) < y)       %Start points located at top
        rotateDeg = 90;

    elseif (sc(1) < x) && (sc(2) < x)   %Start points located on left
        rotateDeg = 0;

    elseif (sc(1) > x) && (sc(2) > x)   %Start points located on right
        rotateDeg = 180;

    else                                    %Start points located at bottom
        rotateDeg = 270;
    end
```

findFreePath.m

```
function freepath = findFreePath(cc, world)

if world == 1
    a = 273;
    b = 153;
    c = 416;
    d = 296;
    e = 558;
    f = 428;

elseif world == 2
    a = 136;
    b = 76;
    c = 208;
    d = 143;
    e = 279;
    f = 214;
end

routes = [1,2,3];                       %Route 1 is closest to start points

    if (cc(1) < a) && (cc(1) > b)   %If block 1 is in route 1..
        routes(1) = 0;                  %Remove this route
    end

    if (cc(2) < a) && (cc(2) > b)   %If block 2 is in route 1..
        routes(1) = 0;                  %Remove this route
    end

    if (cc(1) < c) && (cc(1) > d)   %If block 1 is in route 2..
        routes(2) = 0;                  %remove this route
    end

    if (cc(2) < c) && (cc(2) > d)   %If block 2 is in route 2..
        routes(2) = 0;                  %Remove this route
    end

    if (cc(1) < e) && (cc(1) > f)   %If block 1 is in route 3..
        routes(3) = 0;                  %Remove this route
    end

    if (cc(2) < e) && (cc(2) > f)   %If block 2 is in route 3..
        routes(3) = 0;                  %Remove this route
    end

    freepath = max(routes)             %Free path is remaining route

end
```

findStartLocation.m

```
function [start, finish] = findStartLocation(cc, smallCentroids, world)

if world == 1
    y = 240;
elseif world == 2
    y = 120;
end


locations = [1,2];

    if cc(2) < y           %If y value of robot centroid is in top half
        locations(2) = 0;   %Remove start point 2 from locations
        finish = smallCentroids([2 4]);
    else                    %Else robot is in bottom half
        locations(1) = 0;   %Remove start point 1 from locations
        finish = smallCentroids([1 3]);
    end

    start = max(locations)  %Start is the remaining location
end
```

doRoute.m

```
function doRoute(start, path)

deg3 = 0.16487;    %Turn 2.865983 degrees
deg90 = 5.1775;
degpause = 0.5;
a = 5.65;
b = 10.95;
c = 5.57;
d = 12.66;
A = 20.24;
B = 21.29;

if path == 1
    if start == 1               %Path 1 from point 1 to 2
        send_command('D,1,1');
        pause(a);
        send_command('D,0,0');
        pause(degpause);
        send_command('D,1,-1');
        pause(deg90);

        send_command('D,1,1');
        pause(A);
        send_command('D,0,0');
        pause(degpause);
        send_command('D,1,-1');
        pause(deg90);

        send_command('D,1,1');
        pause(a);
        send_command('D,0,0');
    elseif start == 2                           %Path 1 from point 2 to 1
        send_command('D,1,1');
        pause(a);
        send_command('D,0,0');
        pause(degpause);
        send_command('D,-1,1');
        pause(deg90);

        send_command('D,1,1');
        pause(A);
        send_command('D,0,0');
        pause(degpause);
        send_command('D,-1,1');
        pause(deg90);

        send_command('D,1,1');
        pause(a);
        send_command('D,0,0');
    end
end

if path == 2
    straight = c;
elseif path == 3
    straight = c + d;
end

if (start == 1) && (path == 2 || path == 3)
    send_command('D,-1,1');
    pause(deg3);
    send_command('D,1,1');
    pause(b);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,1,-1');
    pause(deg3);
    send_command('D,1,1');
    pause(straight);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,1,-1');
    pause(deg90);

    send_command('D,1,1');
    pause(B);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,1,-1');
    pause(deg90);

    send_command('D,1,1');
    pause(straight);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,1,-1');
    pause(deg3);
    send_command('D,1,1');
    pause(b);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,-1,1');
    pause(deg3);
    send_command('D,0,0');
elseif (start == 2) && (path == 2 || path == 3)
    send_command('D,1,-1');
    pause(deg3);
    send_command('D,1,1');
    pause(b);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,-1,1');
    pause(deg3);
    send_command('D,1,1');
    pause(straight);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,-1,1');
    pause(deg90);

    send_command('D,1,1');
    pause(B);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,-1,1');
    pause(deg90);

    send_command('D,1,1');
    pause(straight);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,-1,1');
    pause(deg3);
    send_command('D,1,1');
    pause(b);
    send_command('D,0,0');
    pause(degpause);
    send_command('D,1,-1');
    pause(deg3);
    send_command('D,0,0');
end
end
```

visualServo.m

```
function visualServo(roboC, finishC)

    lineTime = 0.089175;        %Time to move one pixel
    rotTime = 0.16487;          %Time to rotate one deg

    %roboC(x,y) and finishC(y,x)

    x = roboC(1) - finishC(2);
    if x < 0                   %Robot has travelled further than destination
        x = abs(x);
        goingBack = 1;
    else
        goingBack = 0;
    end
    y = finishC(1) - roboC(2);
    if y < 0                    %Robot is lower than destination
        y = abs(y);
        goingUp = 1;
    else
        goingUp = 0;
    end

    distance = sqrt(x^2 + y^2);
    radangle = asin(y / distance);

    angle = radangle*(180/pi);

    pauseRot = rotTime * angle;
    pauseLine = lineTime * distance;

    %Rotate required amount
    if goingUp == 1;
        if goingBack == 1;
            send_command('D,-1,1'); %Left
            pause(pauseRot);
            send_command('D,-1,-1'); %Reverse
            pause(pauseLine);
            send_command('D,0,0');
            send_command('D,1,-1'); %Face front
            pause(pauseRot);
        elseif goingBack == 0;
            send_command('D,1,-1'); %Right
            pause(pauseRot);
            send_command('D,1,1'); %Forward
            pause(pauseLine);
            send_command('D,0,0');
            send_command('D,-1,1'); %Face front
            pause(pauseRot);
        end
    end
    send_command('D,0,0');

    if goingUp == 0;
        if goingBack == 1;
            send_command('D,1,-1'); %Right
            pause(pauseRot);
            send_command('D,-1,-1'); %Reverse
            pause(pauseLine);
            send_command('D,0,0');
            send_command('D,-1,1'); %Face front
            pause(pauseRot);
        elseif goingBack == 0;
            send_command('D,-1,1'); %Left
            pause(pauseRot);
            send_command('D,1,1'); %Forward
            pause(pauseLine);
            send_command('D,0,0');
            send_command('D,1,-1'); %Face front
            pause(pauseRot);
        end
    end
    send_command('D,0,0');
```

main_webot.m

```
%-----------------------Connect to Robot-----------------------------------

path(path,'/afs/inf.ed.ac.uk/user/s08/s0831408/Thirdyear/IVR/Matlab/Webot_interf
ace/tcp_udp_ip');
path(path,'/afs/inf.ed.ac.uk/user/s08/s0831408/Thirdyear/IVR/Matlab/Webot_interf
ace/robot_commands');
path(path,'/tmp/');

%--------------------------------------------------------------------------
%-------------------------Image Capturing----------------------------------

snapEmpty = webotsCapture('snapshot',1,2,3,4);
prompt = input('Snapped Maze!', 's');

snapElements = webotsCapture('snapshot',4,5,6,7);
prompt = input('Snapped Elements!', 's');

%--------------------------------------------------------------------------
%------------------------Calculate Homography------------------------------

[centroids bigSize] = findCentroids(snapElements, 50, 120, 0, 8);  %Points from
image
projections = [21 33; 299 33; 21 207; 299 207];                      %Move to
points

orderedCentroids = orderCentroids(centroids, projections);

estimation = esthomog(projections,orderedCentroids,4);
rawHomography = homographise(estimation,snapEmpty,10);
homoSnap = homographise(estimation,snapEmpty,11);

%--------------------------------------------------------------------------
%------------------------Calculate Image Rotation--------------------------

[smallCircleCentroids smallSize] = findCentroids(homoSnap, 10, 75, 0, 12);

imsize = size(snapEmpty);
rotateDeg = rotatecheck(smallCircleCentroids,imsize);

mazeHomoSnap = imrotate(homoSnap,rotateDeg);
figure(13)
imshow(mazeHomoSnap);

%--------------------------------------------------------------------------
%--------------------------Detect Free Path--------------------------------

elementsHomoSnap = homographise(estimation,snapElements,0);
elementsHomoSnap = imrotate(elementsHomoSnap,rotateDeg);
figure(14)
imshow(elementsHomoSnap);

onlyElements = abs(elementsHomoSnap - mazeHomoSnap);
figure(15)
imshow(onlyElements);

[block_centroids blocksSize] = findCentroids(onlyElements, 100, 1000, 0, 16);
```

```
freePath = findFreePath(block_centroids,2); %world=2 (webots)

%-----------------------------------------------------------------------------
%------------------------Find Robot-------------------------------------------

[botCentroid botSize] = findCentroids(onlyElements, 10, 50, 0, 17);

%-----------------------------------------------------------------------------
%-------------------------Plan Route------------------------------------------

[start finishLoc] = findStartLocation(botCentroid, smallCircleCentroids, 2);

%-----------------------------------------------------------------------------
%---------------------------Execute route-------------------------------------

webotRoute(start, freePath);
```

webotsCapture.m

```
function threshBots = webotsCapture(filename, fig1, fig2, fig3, fig4)

    take_snap();

    pause(1)

    image = importdata([filename, '.ppm'], 'ppm');

    if fig1 > 0
      figure(fig1)
      imshow(image)
    end

    hist = doHist(image, fig2);
    smoothedHist = smooth(hist);
    threshold = findThresh(smoothedHist, 4, fig3);
    image = myrgb2gray(image);
    threshBots = doThresh(image,threshold,fig4);
end
```

webotsRoute.m

```
function webotRoute(start, freePath)

a = 10;
b = 17.5;
A = 22.5;

rot = 10;


if(freePath == 1)
    if(start == 1)
```

```
        send_command('D,20,20')
        pause(a)
        send_command('D,10,-10')
        pause(rot)
        send_command('D,20,20')
        pause(A)
        send_command('D,10,-10')
        pause(rot)
        send_command('D,20,20')
        pause(a)
        send_command('D,0,0')
    elseif(start == 2)
        send_command('D,20,20')
        pause(a)
        send_command('D,-10,10')
        pause(rot)
        send_command('D,20,20')
        pause(A)
        send_command('D,-10,10')
        pause(rot)
        send_command('D,20,20')
        pause(a)
        send_command('D,0,0')
    end
end

if freePath == 2
    longLen = a+b;
elseif freePath == 3
    longLen = a+(b*2);
end

if(start == 1) && (freePath == 2 || freePath == 3)
        send_command('D,20,20')
        pause(a+b)
        send_command('D,10,-10')
        pause(rot)
        send_command('D,20,20')
        pause(A)
        send_command('D,10,-10')
        pause(rot)
        send_command('D,20,20')
        pause(longLen)
        send_command('D,0,0')
elseif(start == 2) && (freePath == 2 || freePath == 3)
        send_command('D,20,20')
        pause(longLen)
        send_command('D,-10,10')
        pause(rot)
        send_command('D,20,20')
        pause(A)
        send_command('D,-10,10')
        pause(rot)
        send_command('D,20,20')
        pause(longLen)
        send_command('D,0,0')
end
end
```