Kalin Markov - s1339217 25.02.2016
Tomislav Jovanovic - s1341139
Introduction To Vision And Robotics

Introduction:

In this assignment, we were given three datasets with images, labelled datasets 1, 2 and 3. They each have a bunch of consecutive image frames, so when we run them to display one after the other quickly, we have effectively simulated motion of the robots. The background in each frame stays the same, but the three robots change positions. The task we were faced with was to identify these robots on the screen, distinguish between them – they were three colours – red, green, and blue, draw their trajectory – the path each robot had taken so far, and also draw a small line displaying the orientation of the robot. Our main approach to conquering this task first involved removing the background from the images so that we could focus on the robots. Then we processed the image by converting it to a binary thresholded image that contained two colours – white or black, where the robots were white and the rest of the background – black. We then used library Matlab methods in order to get certain properties of the blobs (robots) – such as centre, range of pixels, etc. We then used this information and functions we wrote in order to determine the colour of each robot, and then we drew all boxes, trajectories, and orientation lines in this colour in order to make the distinguishing between robots easy. The next section of this report goes into detail on how exactly all of this was done.

Methods:

Getting the Background

The first step to getting our vision system to work was getting the background for each of the four datasets. This would then help us easily extract the robots. We wrote a Matlab file called get_background.m in order to complete this task. In the start of this file, we specify what dataset it is that we want to find the background for. The datasets are huge, and we do not need to use all images for finding the background. We experimented with different numbers of images chosen for the filtering, and we noticed that having more images does not make the background extraction any better. Also, it is possible that the get_background.m file takes longer to run if we give it too many images. Thus, we decided that 20 images would be the best for completing the task. It is best if the 20 images are not consecutive but are instead taken at random from the entire dataset. We had a loop in the code that extracted 20 random images from the dataset. The 20 images chosen were saved in the chosen_images matrix. The technique for actually finding the background was called median filtering. The idea it is based on certainly makes sense in real life – given a set of images of the same exact place over a given time, everything that remains the same is likely to be the background, while objects that keep changing their position do not make up this background. It is the same in this case – the robots keep moving, so the background should not consist of them. The median averaging we do refers to averaging corresponding pixels in the 20 given images. Because the robots will not stay in one place over the 20 frames, getting the median of corresponding pixels over the 20 frames will result in the pixel colours of the background and not the robots. This technique is performed in just one line of code by calling the median function and passing it the set of 20 images chosen. When we chose the 20 random images, we only start from the 100^{th} frame of the dataset onward. The reason for this lies in how the median filtering works. The robots do not move at all for the first 100 or so frames of each dataset. Thus, taking into account the method just described above, it we were to pass 20 of these images to the median function, it would consider the robots as background as well. Also, now that we described the algorithm, we can justify why the 20 images taken were not consecutive. If we take 20 consecutive images, and if the robots move very slowly, there is a chance that the average pixel value of some corresponding pixels in the frames will include some bits of the robots. Obviously this is not desired, so therefore we take 20 random images and solve this problem. The resulting background image is then saved in a file called

“background” followed by the number of the dataset for which this is the background of.

Binary Thresholding:

The next step in our vision system is called binary thresholding, and lies in the get_binary.m file. The purpose of this is to create a black and white segmented image, where the white represents foreground objects (robots) and the black is the background. Given an image and its background, we first call the imsubtract function to subtract the background. This should result in a mostly black background, and it will keep the robots the same colour. We then get the binary image for each channel (red, green, and blue) – this will be a 1 (white) if the the channel value for that pixel is over 50, and 0 (black) if the channel value for that pixel is less than 50 (50 on the scale from 0 to 255). We want to combine our results for the three channels, so we just “or” the images per channel and get the final image. This basically says that if one of the three channels had a value above 50 for a pixel, then we will make that pixel white. Removing the background from the original image basically means making it black, so none of those pixel RGB values will be over 50, and the final image will remain black. The robots in the foreground certainly have RGB values of many pixels over 50, so they will become white. We noticed that there were some gaps in the white blobs that represented the robots – these were probably some pixel values that didn’t pass the threshold test and remained black. We closed these gaps by using the matlab functions strel and imclose. We also removed the very small white regions within the background by using erode and dilate. This resulted in a properly segmented image that distinguished the robots and the background quite well.

Getting Average Colour:

The third “helper” matlab file that we created was called get_Average_RGB.m. The purpose of this piece of code was quite simple. Given an image and a list of certain pixels in that image that we want to examine, this function returned the average RGB value of those pixels. Its methodology was quite simple – just loop through those pixels in the image, extract the red, green, and blue values, keep a running sum of each one, and then divide by the number of pixels taken to get the average. We tried to do this directly on the input image, but it did not work as expected. The reason for this was that the matrix representing the image holds uint8 values, which can only go up to 255, so the sum gets cut off at this number and the results make no sense. That is why we had to convert the image pixel values to double first before running the loop.

Finding Closest Robot:

We also had a find_closest_robot.m file. This takes in a centre point, a list of all centre x coordinates for each robot so far, a list of all centre y coordinates for each robot so far, and then the instance in time we want to compare the centre point to (basically, when we actually use this method, this “instance in time” will be the previous frame, as we want to compare the current centre of an object to all centres observed in the previous frame). This function basically loops through the robots so far, finds the distance from the given centre point to each centre of each robot at the frame specified by the variable filled_centers_num (in our code), and then updates the minimum distance. At the end, we return the index of the robot that was closest to the point we passed to the function. The distance is just a euclidean distance, and is calculated by the function “distance” in the file distance.m. In that file, we chose to use the method of finding the squared distance and not going the extra step of square-rooting and finding the exact distance. This is because finding the square root of a number can be a very expensive and slow operation, and it is not needed – we never work with actual distance but only need them for comparison, and if a distance is smaller than another one, then its square will also be smaller.

Putting Everything Together:

The file main.m puts everything together, using all of the above functions, and represents the core of the whole vision system. Running it will bring up the frames and identify all key components. Before we do this, we must run get_background.m in order to get and save the

background for that particular set of images in an image file. At first we had a method that had one main loop that read all of the frames, found the robot colours for each frame, etc. However, we soon realized that this was not the best approach, and we could optimize our code in some ways. Since the first frame will always contain all robots that will ever be present throughout the dataset, it is much better and more efficient to assign each robot an index, calculate its colour, and save this information in a matrix, for only one frame instead of repeating this same process for every single image. Thus, the first part of the main.m code deals with just the first frame, and then the second half is a loop that goes through all other frames and processes them. The first step is to read the correct background image based on the data set specified, and then read the first frame of the dataset. We then call the get_binary function, which subtracts the background and creates a black and white segmented image. There is a library matlab function called regionprops, and when we pass the binary image to it and give it a parameter "all", it gives us a bunch of information on each white blob it detects on the image – which would correspond to our robots. We initialise two matrices that will hold the all the x-coordinates of the robot centres for each frame (center_Xs) and all the y-coordinates of the robot centres for each frame (center_Ys). We then loop through all regions detected in this frame, and add each centre to the center_Xs and center_Ys matrices. Now we need to fill out the matrix that holds the information on each robot and its colour. The region_props method returns a list of all pixels occupied by the that blob/robot. We pass this list to the get_Average_RGB function, along with the original, unprocessed frame, and it returns to us the average RGB colour of that robot. We first tried drawing the box around the robot in this exact colour, but is was not very clear and did not stand out from the background to well, so we decided to convert this colour to pure red, blue, or green, depending on which of these colours it was closest to. We did this by taking the maximum of the red, green, and blue – whichever of these three numbers was highest, that is the colour we said that particular robot was. Then we stored this colour at the index of the robot which got assigned to it. There was one dataset which caused a tie between the red and the green components for one of the robots, so we made a separate if statement to deal with that case.

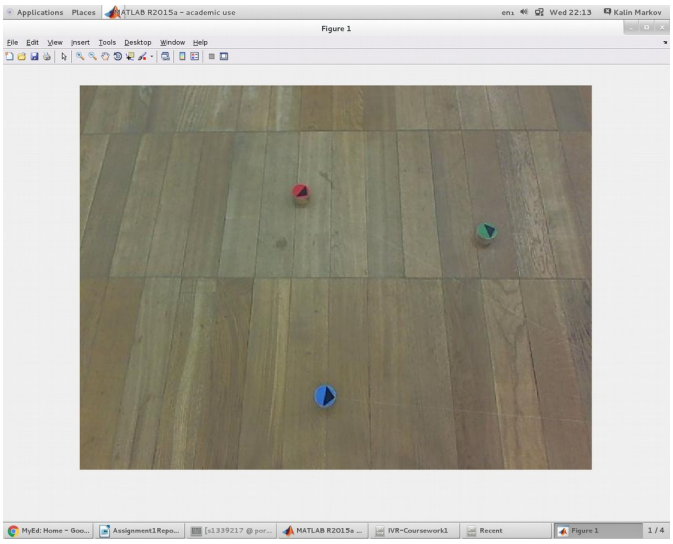
The second half of the main.m code loops through all other frames in the dataset. We read the current frame and show it, so that we can draw boxes, paths, and orientations onto it. We then get the binary segmented image and use regionprops to get information about all of the white blobs, which should be robots. In the first frame, we set the num_robots variable to be the number of robots detected there, and we assume that the vision correctly saw all robots there and that it didn't pick up any extra objects. In the loop through all remaining frames, we initialise num_regions to be the number of regions detected in that frame, which should not be more than the number of robots initially present, but could be less, as some robots may go off of the frame. As a result, when drawing the bounding boxes and calculating centres of regions of a new frame, we loop from 1 to min(num_regions, num_robots). In this loop, we get the centre of the region (by indexing into the region_props matrix we creating after reading the frame), and then we use the find_closest_robot function described earlier in order to find the closest robot to this centre, based on the robot positions from the previous frame. This is basically like matching up the region we detect with the correct robot. In order to determine what colour it is, and thus in what colour to draw the bounding box in, we indexed into the holdRobotInformation matrix that was created using the first frame of the dataset, in the top part of the code. We finally draw the bounding box as a rectangle in that colour, and we add to both lists of x and y coordinates so far the new centre we detected. Drawing the trajectories of the robots was quite straight-forward because the center_Xs and center_Ys hold all of the previous x and y coordinates of all robots. All we had to do was call the plot functions and give it all x and y coordinates encountered so far of a particular robot.

The last main task the vision system had to perform was to draw the orientations of the robots. The first way we tried to attempt this was using the arrowhead, but we ran into great difficulty when identifying this arrowhead on each robot. As a result, we decided to use a simpler approach. This involved using the current centre of robot and then a previous centre to write an equation of a line that would represent the orientation of the robot. We could use this approach

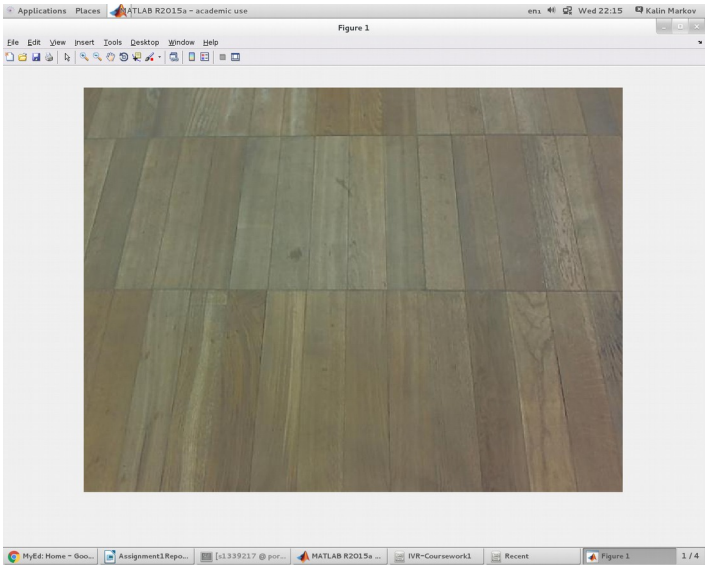
because we know that the robot can only move in the direction its arrowhead is pointing. Because there is quite a lot of noise in the images, if we use the current robot centre and the previous one, we could get quite inaccurate orientations. That is why we used the current frame and the one from 10 frames before in order to write the equations of the line. After finding the slope and y-intercept of the line, we determine if the robot is going left or right on the screen, then pick a new x coordinate that is 40 away from the current centre's x-coordinate, and then we calculate the corresponding y-coordinate using the equation of the line. The point of this is so that the line segment we draw can extend enough in front of the robot so that it is easily visible. We also take care to avoid attempting to draw lines outside of the image.

Results:

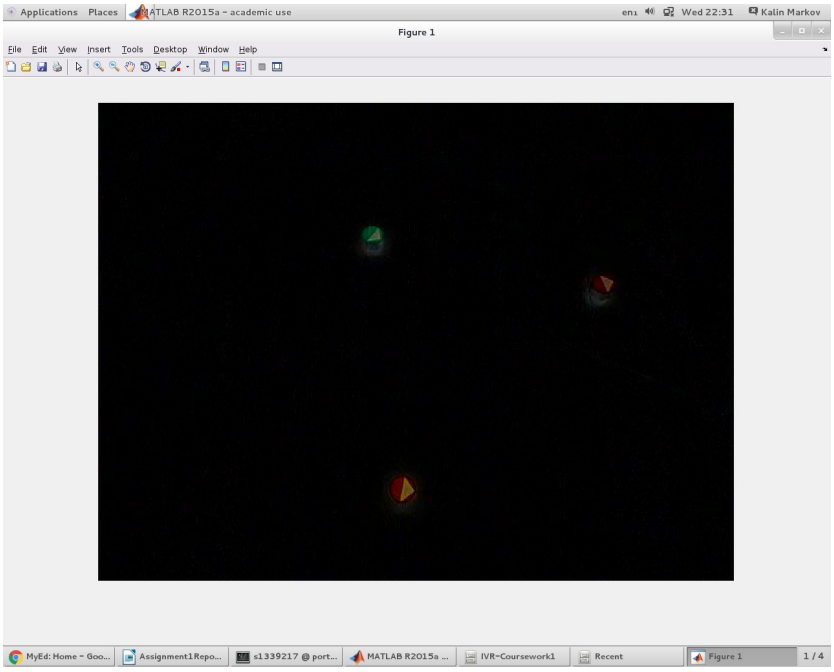
Here are the results after each stage in the image processing.
This is what a sample frame could look like:



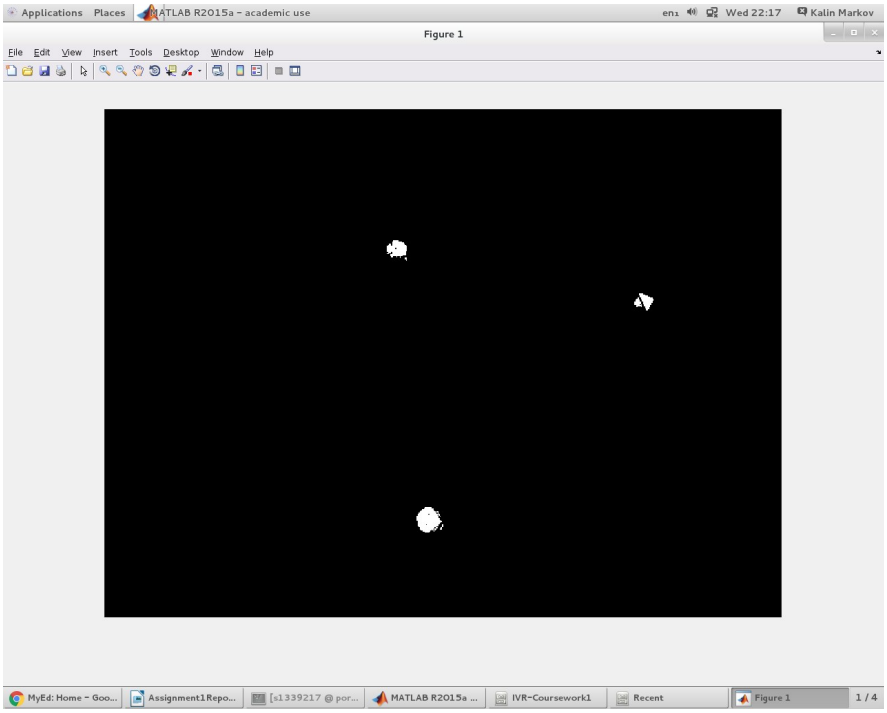
And here is what we get when we get the background of this image:



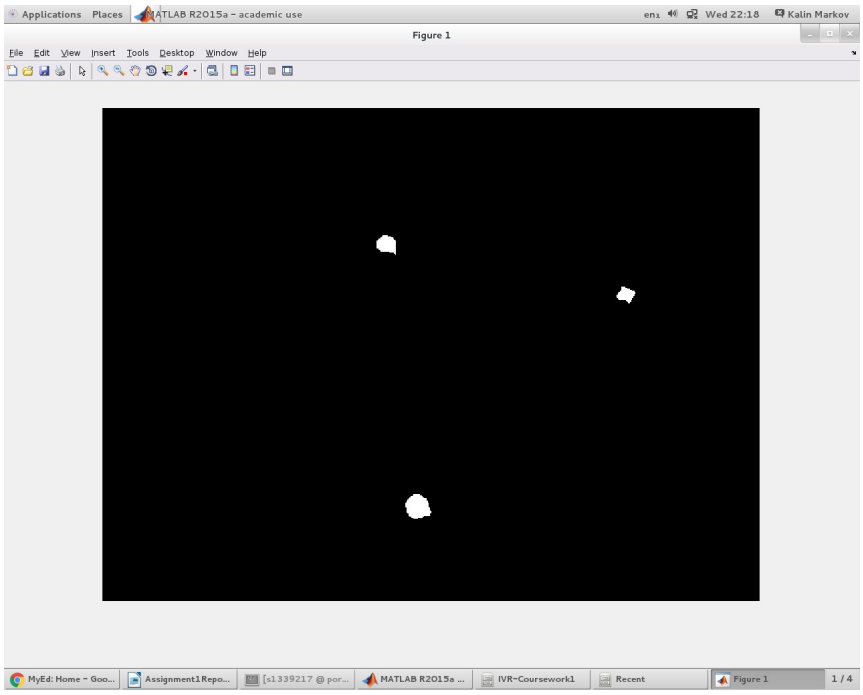
As you can see, getting the background seems to work quite well. As described earlier, the next step in the image processing cycle would be to remove the background from the original image. This is done using subtraction of pixel values, so anything originally in the background should get mapped to something 0 or close to 0 – so the colour black. Here is what subtracting the background from the image above results in:



It is good that we get a black background, and we can also see how the robots are still there. Their colours are quite changed and different from the original image, but this is not a problem, as we only need the locations of their pixels. We access these locations in the original image to get the colour, so the fact that the colour here is distorted is not a problem. We can also see the shadows of each robot. Our code did not deal with removing them. They may cause the boxes drawn around each robot to be slightly off centre, but this should not be a big problem. The next step is to do binary thresholding:

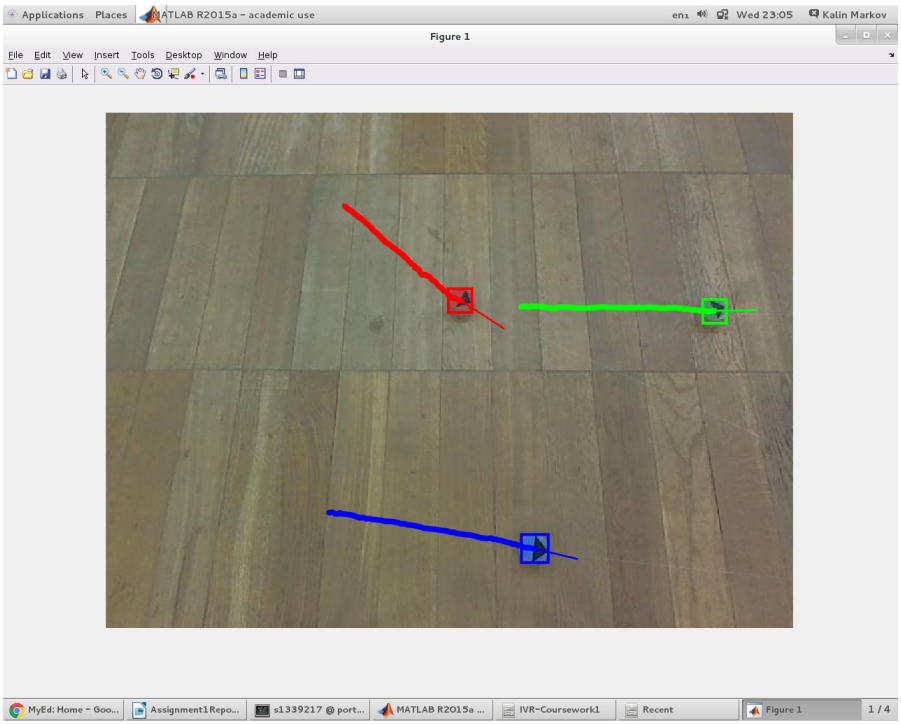


The resulting image contains three white blobs – which directly correspond to the positions of the robots. There are some black gaps within the white blobs which we want to remove, and once we do this, the processed frame looks like this:

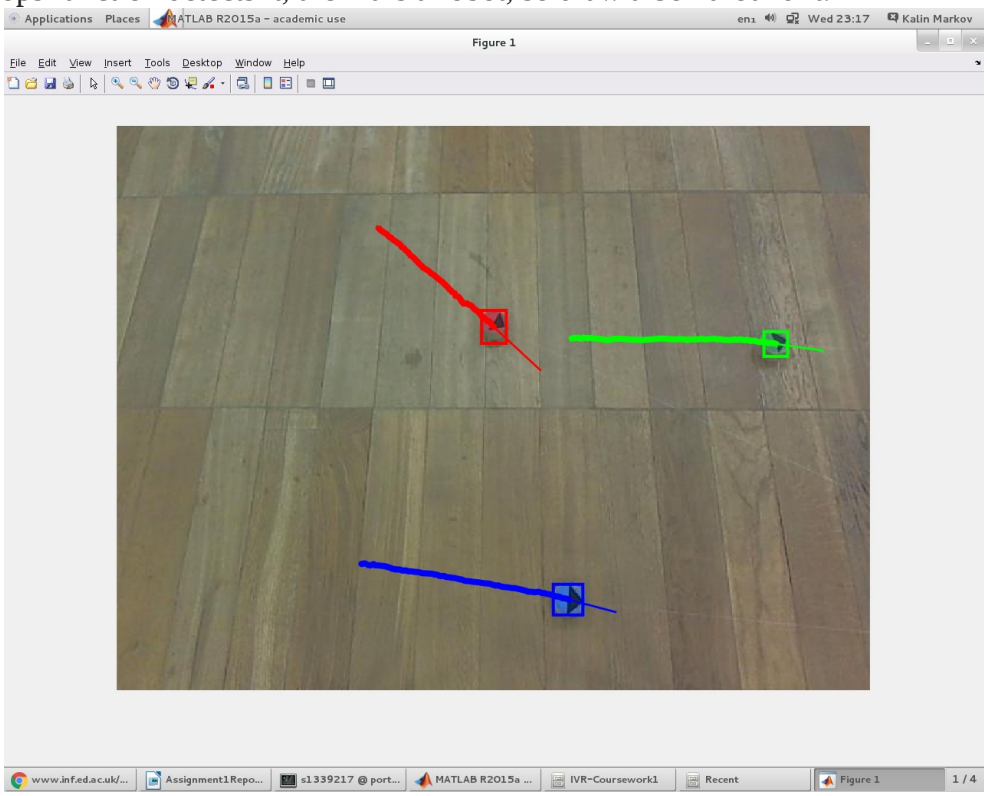


The erode and dilate do not have any visible effect here, as even if there are any very small white regions remaining, they are too small to see with the human eye.

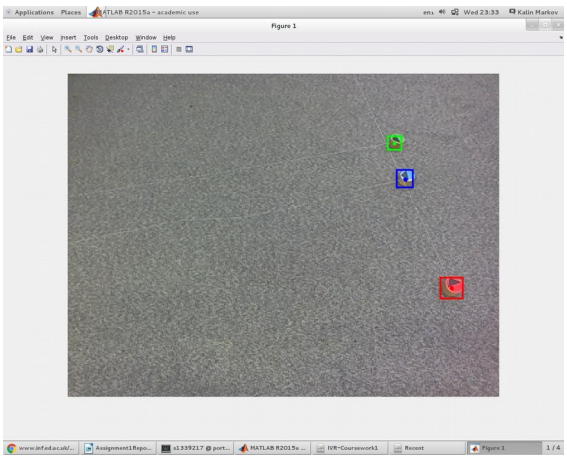
We call the regionprops function on this binary image, and then use the information it returns and the original frame in order to identify the robots, their tracks, and orientations. Here is a sample of how this works:



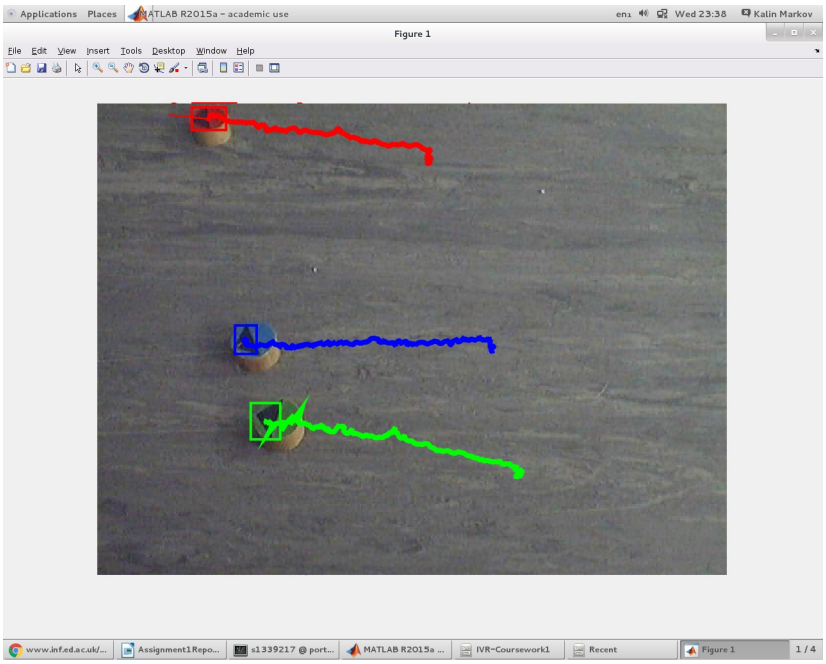
This sample shows our vision system working quite well. The colours, tracks, and orientations are all correct in this case.
The robot detection was working very well, and never caused any invalid detections on the 4 datasets we were given. Also, it never failed to detect any robot present, and it always got the robot colours correctly as well. One small problem it had was that occasionally the rectangle it drew around a robot was slightly off centre or a bit larger than it should be. I think this was because of the shadows of the robots. For example, in the image below, the box around the red robot extends a slight bit too far below the robot. Our criteria for a detection was simple – if there is a white spot and the regionprops function detects it, then it is a robot, so draw a box around it.



The robot tracking was also working very well on the 4 datasets we were given. The tracks always seemed to be correct and continuous for each robot, with no breaks or gaps. To draw the track, we simply kept joining the robot centres from the start frame onward and kept extending it with each new processed frame.
The orientation of the robots was working decently, but definitely was the least reliable out of the three task we had. When the robot has not moved at all, no orientation is drawn. This is because our orientation depends on the robots centres changing position by at least a certain distance, and this change is used to draw the orientation line. When the robots are static, the centres do not change at all or by very little (due to noise in camera), so no orientation line is drawn:



Also, the orientation sometimes is not correct or disappears for a short amount of time. This is when the centres calculated are at an angle that is not the same as the robot's actual motion. When the orientation disappears, this could be because the robot has not moved enough distance between ten frames.



For example, in this frame the blue and green robot orientations have disappeared. Also, the paths here are very choppy. This is because the robot centres kept shifting to the side during general track of motion. However, the general direction is still correct.

Discussion:

I think our program was quite successful as a whole. The robot detection and tracking was working very well and did not have any limitations. One way we could improve it to make it even better is to handle shadows in some way, so that the detection box always stays the exact same size. The orientation detection of the robots was working well when the robots were moving, but it certainly has some problems and limitations. It did not show any orientation when the robots were static, and sometimes it got the orientation wrong because the centres taken did not correspond to the direction the robot was actually going in. This was due to noise in the frames. One way to improve this would be to use a completely different technique, one that is more complicated and we were considering using before. This technique could be locating and using the arrowhead to form a triangle, then finding its tip and centre of the side opposite this tip, and drawing a line with this orientation. This would always draw the orientation, even when the robot is static, and is less likely to go wrong in the calculation as well.

Code:

distance.m

```
% Euclidian distance between two points of form [x y]
function dist = distance(a, b)

dist = sum((a - b) .^ 2);
```

---

find_closest_robot.m

```
% Finds the closest robot to the point given their last known position
function index = find_closest_robot(point, center_Xs, center_Ys, filled_centers_num)

min_dist = 99999999999;
index = -1;

% Total number of robots on the field
total_robots = length( center_Xs(:,1) );

for i = 1:total_robots
    center_i = [center_Xs(i,filled_centers_num) center_Ys(i, filled_centers_num)];
    dist = distance(point, center_i);
    if dist < min_dist
        min_dist = dist;
        index = i;
    end
end
```

---

get_Average_RGB.m

```
function averageColor = get_Average_RGB(image,pixelList)
lengthPixelList = length(pixelList);
h = fspecial('gaussian',10,10);
blurredImage = imfilter(image,h);
sumRed = 0;
sumGreen = 0;
sumBlue = 0;
image_double = im2double(blurredImage);
for i=1:lengthPixelList
    red = image_double(pixelList(i,2),pixelList(i,1),1) * 255;
    green = image_double(pixelList(i,2),pixelList(i,1),2) * 255;
    blue = image_double(pixelList(i,2),pixelList(i,1),3) * 255;
    sumRed = sumRed + red;
    sumGreen = sumGreen + green;
    sumBlue = sumBlue + blue;
end
averageColor =
[uint8(sumRed/lengthPixelList),uint8(sumGreen/lengthPixelList),uint8(sumBlue/lengthPixelList)];
```

---

get_background.m

```
% Dataset .. Possible values (1, 2, 3,4)
dir_num = 4;

% Directory path of the images
dir_name = ['data' int2str(dir_num) '/']

% Get the list of all the image file names from the chosen directory
image_names = dir([dir_name '*.jpg']);
num_images = length(image_names);

% Total number of frames we want for the median filtering
num_filter_frames = 20;

chosen_images = zeros(480,640,3,num_filter_frames, 'uint8');
```

```
for i=1:num_filter_frames

    % Get the index of the next_frame, starting with 100
    image_index = 100 + (num_images-100) * i / ( num_filter_frames + 1 );
    image_index = round( image_index );

    % Normalize the image after reading
    image = imread( [dir_name image_names(image_index).name] );
    chosen_images(:,:,:,i) = image;
    imshow(chosen_images(:,:,:,i));
end


% Get the background as the median over all the chosen images
background = median(chosen_images, 4);

% Write background as "background$i.jpg"
imwrite(background, ['background' int2str(dir_num) '.jpg']);
```

---

get_binary.m

```
function ret = get_binary(image, background)

% Subtract background from the original image
no_background = abs(imsubtract(background, image));

% Get the binary image for each channel
thresholded_image = no_background > 50;

% Or the images per channel to get the final image
binary_image = thresholded_image(:,:,1) | thresholded_image(:,:,2) | thresholded_image(:,:,3);

% Close the gaps to get clear blobs
se = strel('disk', 6);
binary_image = imclose(binary_image, se);

% Mitigates the issue of super small regions by removing them with erode
binary_image = bwmorph(binary_image, 'erode', 2);
binary_image = bwmorph(binary_image, 'dilate', 3);

% Return labeled image with the connected regions
ret = bwlabel(binary_image,8);
```

---

main.m

```
dir_num = 4;
dir_name = ['data' int2str(dir_num) '/'];

image_files = dir([dir_name '*.jpg']);
num_images = length(image_files);

% Which frame we want to start tracking the robots from
starting_frame = 1;

% Fetch the background for a particular dataset
background = imread(['background' int2str(dir_num) '.jpg']);
```

```
%%%%%%%%%%%%%%%%% Set the initial parameters %%%%%%%%%%%%%%%%%
frame = imread( [dir_name image_files(starting_frame).name] );

% Convert to frame to binary thresholded image
binary_image = get_binary(frame, background);

% Get the inital white blobs
region_props = regionprops(binary_image, 'all');

% Get the number of robots
num_robots = length(region_props); % ASSUMING ALL ROBOTS ARE PRESENT IN THE INITIAL
FRAME

% Keep the positions of all the centers in each frame (can be optimized using only 1 matrix --
TODO )
center_Xs = zeros(num_robots, num_images);
center_Ys = zeros(num_robots, num_images);

% Holds the color for each robot
holdRobotInformation = ['a','a','a'];

% Set the initial centers
for j=1:num_robots
    center_j = region_props(j).Centroid;
    center_Xs(j, starting_frame) = center_j(1);
    center_Ys(j, starting_frame) = center_j(2);

    % Get the average color in each region (which should correspond to a robot)
    pixelList = region_props(j).PixelList;
    averageColor = get_Average_RGB(frame,pixelList);

    % Find the index of the max value in the averageColor array
    [maxValue,maxValueIndex] = max(averageColor);

    % Threshold based on colors
    if(maxValueIndex == 1 && (averageColor(1) ~= averageColor(2)))
        color = 'r';
    end
    if(maxValueIndex == 1 && (averageColor(1) == averageColor(2)))
        color = 'g';
    end
    if(maxValueIndex == 2)
        color = 'g';
    end
    if(maxValueIndex == 3)
        color = 'b';
    end

    % Assign to array that will hold the colors of each robot
    holdRobotInformation(j) = color;
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%

for i=(starting_frame+1):num_images
    % Get the current frame and normalize the RGB values
    frame = imread( [dir_name image_files(i).name] );

    % Show it so we can draw boxes + path onto it
    imshow(frame)
```

```
% Convert the frame to binary thresholded image
binary_image = get_binary(frame, background);

% Get the regions(contours) of the white blobs
region_props = regionprops(binary_image, 'all');

% Sort the regions in a descending fashion based on region AREA
[blah, order] = sort([region_props(:).Area],'descend');
region_props = region_props(order);

% Get the number of regions
num_regions = length(region_props);

% Holds the indicator of the center being present for a particular robot
found_center = zeros(1, num_robots);

% Get the bounding boxes and centers of the regions from the NEW frame
for j=1:min(num_regions, num_robots)
    % J-th center from the list of white blobs
    center_j = region_props(j).Centroid;

    bounding_box = region_props(j).BoundingBox;

    hold on

    % Closest robot to center_j
    index_closest = find_closest_robot(center_j, center_Xs, center_Ys, i-1);

    % Get color of robot and draw a rectangle of that color around it
    color = holdRobotInformation(index_closest);
    rectangle('Position', bounding_box,'EdgeColor',color,'LineWidth',3);

    % Set the new position of the robot to center_j
    center_Xs(index_closest, i) = center_j(1);
    center_Ys(index_closest, i) = center_j(2);
    found_center(index_closest) = 1;
end

% If the center hasn't been set, copy the previous value
for j=1:num_robots
    if found_center(j) == 0
        center_Xs(j, i) = center_Xs(j, i - 1);
        center_Ys(j, i) = center_Ys(j, i - 1);
    end
end

% Draw the orientations of the robots
for k=1:num_robots
    % Use two points to calculate the equation of the line that will
    % represent the robot's orientation

    point1 = zeros(1,2);
    point1(1,1) = center_Xs(k,max((i-10),1));
    point1(1,2) = center_Ys(k,max((i-10),1));

    point2 = zeros(1,2);
    point2(1,1) = center_Xs(k,i);
    point2(1,2) = center_Ys(k,i);

    % Only draw orientation if the centers have changed and the robot
    % has moved a little bit
    if(not(point1(1,1) ==  point2(1,1)) && not(point1(1,2) == point2(1,2)) &&
```

```
distance(point1,point2) > 100)
            % Find the slope and the y-intercept of the line
            if(not((point2(1,1)-point1(1,1)) == 0))
                slope = (point2(1,2)-point1(1,2)) / (point2(1,1)-point1(1,1));
                yintercept = point1(1,2) - (slope * point1(1,1));
            end

            % Case when the robot is going left on the screen
            if(center_Xs(k,i) < center_Xs(k,i-1))
                if(center_Xs(k,i) - 40 > 0)
                    newPointX = center_Xs(k,i) - 40;
                end
                if(center_Xs(k,i) - 40 <= 0)
                    newPointX = 0;
                end
            end

            % Case when the robot is going right on the screen
            if(center_Xs(k,i) > center_Xs(k,i-1))
                if(center_Xs(k,i) + 40 > 640)
                    newPointX = 640;
                end
                if(center_Xs(k,i) + 40 <= 640)
                    newPointX = center_Xs(k,i) + 40;
                end
            end
            % Find new point that will be on the other end of the line segment
            if(not(slope * newPointX + yintercept < 0) && not(slope * newPointX + yintercept >
480))
                newPointY = slope * newPointX + yintercept;
            end
            if(slope * newPointX + yintercept < 0)
                newPointY = 0;
            end
            if(slope * newPointX + yintercept > 480)
                newPointY = 480;
            end

            newPoint = zeros(1,2);
            newPoint(1,1) = newPointX;
            newPoint(1,2) = newPointY;

            % Get color so that the trajectory can be drawn in that color
            color = holdRobotInformation(k);
            plot([point2(1), newPoint(1)], [point2(2), newPoint(2)], color, 'LineWidth', 2);
        end
    end

    % Draw the paths of the robots
    for k=1:num_robots
        % Get color so that the trajectory can be drawn in that color
        color = holdRobotInformation(k);
        plot(center_Xs(k,starting_frame:i),center_Ys(k,starting_frame:i),color,'LineWidth',6);
    end
    hold on;

    % Pause the time between iterations to keep boxes from flickering
    pause(0.000000000001);
end
```