Introduction to Vision and Robotics: Assignment 1

Chris Swetenham: Student Num, Daniel Mankowitz: S1128165

3/11/2011

1 Introduction

An overview of the main ideas used in the approach

2 Methods

Describe the vision techniques used

2.1 Initial Processing

The first part of the algorithm, summarised in Figure 1, processes each image to minimise variations both between datasets and across each image. First, the image is blurred using a gaussian kernel of width 5; this removes some of the noise in the image which would cause trouble in later stages. Next, the rgb value of each pixel is normalized; this corrects for variations in illumination across the image. At this stage, the green channel is modified by subtracting 0.2 times the blue channel. This is because the colours of the cars are not a pure value in a single channel; in particular, the blue car shows up brightly on the green channel. This simple operation is equivalent to finding a new "green car channel" attuned particularly to the green car. If the cars were other colours, such as cyan, magenta and yellow, we could have constructed dedicated channels for each car; but in this case a simple tweak to the green channel was enough to disambiguate the cars. Finally, each colour channel is scaled so that the minimum and maximum values present in the channel map to 0 and 1. This helps the thresholding after detection.

The detection algorithm a single image, and identifies the most likely location of the red, green and blue cars. Once these steps have been performed, the algorithm estimates the location of each car in the image using the corresponding colour channel. For the red car for instance, it looks at the red channel, and computes the maximal value for each row, and for each column. It convolves each of these to reduce noise, and finds the maximum along the rows and maximum along the columns. The corresponding x and y coordinate are the identified centre of the car. The algorithm takes a 120x120 bounding box around the centre, and passes the image to the next stage for finer-grained processing. (Is this equivalent to just blurring the image and finding the x, y coord of the maximum, then thresholding at 75% of that value?)

2.2 Linking Algorithm

The linking algorithm is used to link together detections of a single robot in consecutive frames. Since the images provided in the datasets are RGB, the linking algorithm has been developed to run on each of the three colour channels respectively.

The algorithm is shown in *Figure 2*. Initially, the image on the k^{th} iteration will have been split into three colour channels of red, green and blue respectively. Each colour channel is represented as a separate image and each image has been clipped to a dimension of $120x120$ in the detection algorithm. This enables a more accurate bounding box to be calculated for each robot as there

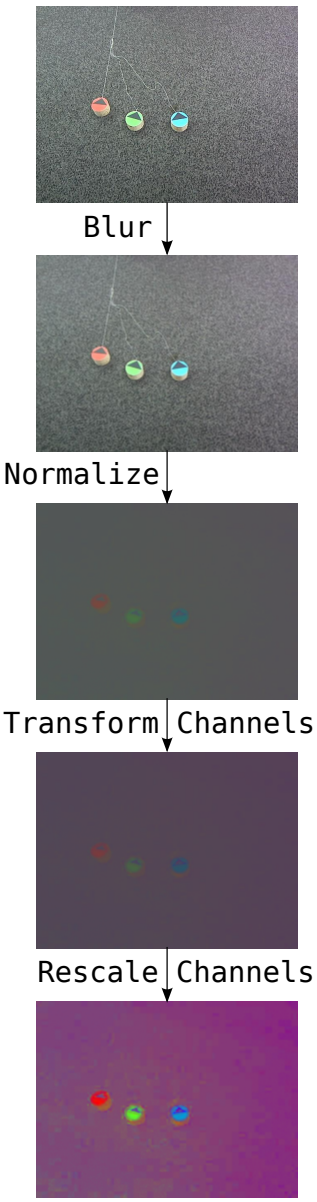

Figure 1: Image Processing Pipeline

will be less image noise in the clipped image.

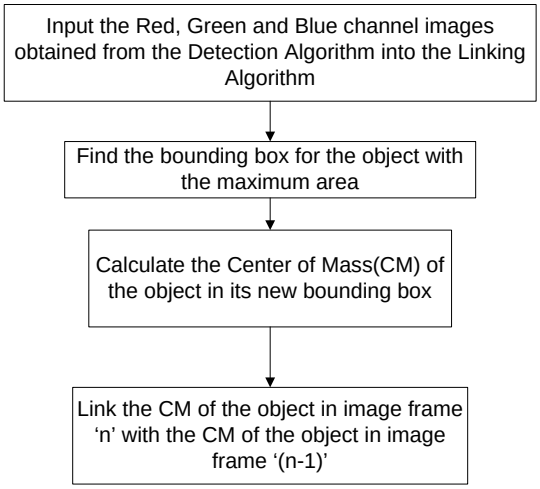

Figure 2: The algorithm developed to link detections of the robot in consecutive frames

2.2.1 Finding the Bounding Box

The bounding box of each robot is calculated using the function *calcBoundingBox*. A flow diagram of the function can be seen in *Figure 3*. This function calculates the bounding box for each robot as well as the box's corresponding centroid in each of the three colour channels respectively.

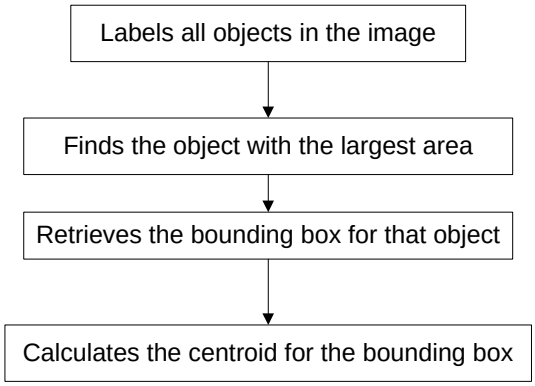

Figure 3: The bounding box calculation for each robot

An excerpt from the *calcBoundingBox* function is shown below.

```
%Excerpts from the bounding box calculation

%Label the image. The labelled image is labelX
[labelX, numXBlobs] = mybwlabel(TImgX);

%Calculate the area for each object found in 'labelX'
allBlobAreas = [];
for i=1:numXBlobs
```

```
    [rows,cols,vals] = find(labelX==i);
    blobSize = sum(vals);
    allBlobAreas = [allBlobAreas blobSize];

end

%Find the object with the maximum area in the image
[maxBlobArea, index] = max(allBlobAreas);

if HaveToolbox==0
%Without using the image toolkit, extract the bounding box
[rows,cols,vals] = find(labelX==index);
x1 = min(cols);
x2 = max(cols);
y1 = min(rows);
y2 = max(rows);
end
```

Each image is labelled using the *mybwlabel* function as seen in the code excerpt. This will identify all of the objects in the image and assign a set value to each pixel associated with a specific object.

The object with the largest area is then identified. A bounding box is subsequently calculated for the object with the largest area (I.e. the largest number of labelled pixels). This is achieved using the *find* Matlab function which identifies the largest object and stores each pixel belonging to the object with its respective row and column position. This is then used to determine the bounding box of the object.

The object will then be surrounded by its corresponding bounding box. The centroid of this bounding box is then calculated. This algorithm is performed on each of the three clipped images corresponding to the three respective colour channels.

It must also be noted that due to bad lighting or image noise, a robot may not be detected in an image frame. This has been catered for in the *calcBoundingBox* function. If the maximum calculated area of an object in the image is 0, then it is assumed that an object has not been detected and the frame is skipped.

2.2.2 Calculate the Center of Mass

Once a bounding box for each robot has been determined, the center of mass of the robot needs to be calculated in relation to this new bounding box. This is achieved by utilising the *calcBoundingBoxCM* function.

The algorithm initially calculates the area of the object within the bounding box using *Equation 1.* This value is used to determine the center of mass of the object within the bounding box.

$$A = \Sigma_r \Sigma_c P_{rc} \tag{1}$$

The center of mass of the object is then determined using *Equation 2.* $(\widehat{r}, \widehat{c})$ are the row and column coordinates of the center of mass of the object respectively.

$$(\widehat{r}, \widehat{c}) = (\frac{1}{A}\Sigma_r \Sigma_c r P_{rc}, \frac{1}{A}\Sigma_r \Sigma_c c P_{rc}) \tag{2}$$

2.2.3 Linking and Plotting

The final step in the linking algorithm is to link the object in image frame n with the object in the previous image frame $(n-1)$. A criteria needs to be developed in order to define when this

'linking' of objects should occur.

The criteria has been defined as follows. It has been previously mentioned that each image has been separated into its R,G and B channels. The R,G,B images have also been cropped into $120X120$ dimension images. Within these cropped images, the object with the largest area of pixels in each channel is chosen to represent the robot. For example, the object with the largest area of red pixels in the red channel cropped image, represents the red robot. Based on this assumption, the largest objects detected in adjacent image frames, in the same colour channel, are connected. This is a safe assumption as the largest concentration of red, green and blue pixels are usually associated with the red, green and blue robots respectively, once the images have been cropped to a $120X120$ dimension.

The center of mass of the red, green and blue robots are stored in 'linker' arrays as shown below. The links are plotted on the estimated background image at the end of the program.

```
%***********************************************
%To link tracks on the estimated background image
XLinkerR = [XLinkerR trueCMXR];
YLinkerR = [YLinkerR trueCMYR];
XLinkerG = [XLinkerG trueCMXG];
YLinkerG = [YLinkerG trueCMYG];
XLinkerB = [XLinkerB trueCMXB];
YLinkerB = [YLinkerB trueCMYB];
```

2.3 Identifying Robot Orientation

Once the positions of the robots have been identified, the next step is to determine the direction in which the robot is currently travelling. The orientation of the robot is identified by an arrow that points in the robot's current direction of movement. At this stage it is important to define the difference between the meaning of an arrow and an arrowhead in this report. An arrow refers to the vector calculated to point in the robot's current direction of movement as shown in *Figure 4*. An arrowhead refers to the arrow attached to the face of the robot.

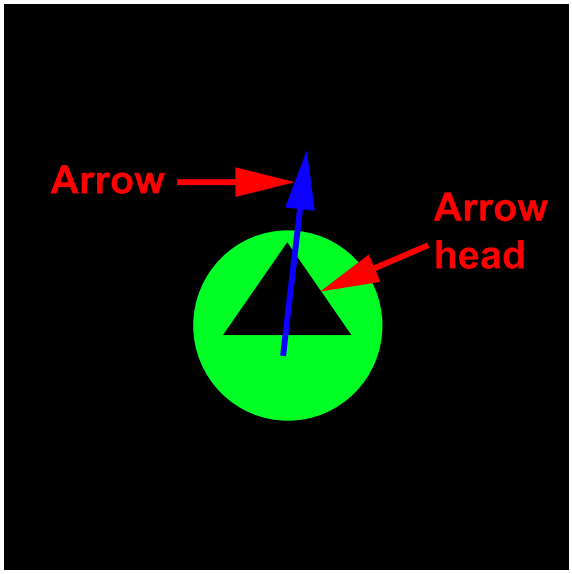

Figure 4: The definition of an arrow and an arrowhead

The orientation algorithm ultimately connects the center of mass of the robot to the centroid of the bounding box surrounding the robot as seen in *Figure 5*. Due to the structure of the robot, the robot's center of mass is found just below the centroid. Connecting the center of mass to the centroid will generally result in an accurate vector describing the robot's orientation.

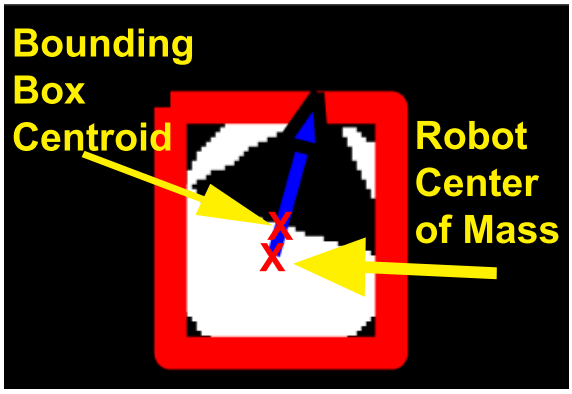

Figure 5: Connecting the center of mass of the robot to the bounding box centroid to determine the robot's orientation

The algorithm initially receives the center of mass coordinates of the robot as well as the bounding box centroid as shown in the code below. Using these values, the algorithm then defines vectors *DR, DG* and *DB* for each of the three colour channels. By normalising these vectors using *Equation 3*, unit vectors are obtained and are then scaled to create the arrows pointing in the direction of the robots current orientation. These arrows are then plotted on the current image frame using a third-party function called *plot_arrow*.

$$D_x = \frac{D_x}{||D_x||} \tag{3}$$

```
%Define unit vectors in the direction of the bounding
%box centroid for each of the rgb channels respectively
CenterMassR = [CenterMassXR CenterMassYR];
DR = (CentroidR - CenterMassR);

CenterMassG = [CenterMassXG CenterMassYG];
DG = (CentroidG - CenterMassG);

CenterMassB = [CenterMassXB CenterMassYB];
DB = (CentroidB - CenterMassB);

%Calculate the unit vectors for each channel
DR = DR/norm(DR);
DG = DG/norm(DG);
DB = DB/norm(DB);

%*****************************************************
%Excerpt of code for plotting the arrows connecting the
%centroids and center of mass of each robot
plot_arrow(trueCMXR,trueCMYR, trueCentroidBBXR+30*DR(1),trueCentroidBBYR+30*DR(2)...
,'linewidth',2,'headwidth',0.25,'headheight',0.33,'color',LineColRArrow,...
'facecolor',LineColRArrow);
hold on
plot_arrow(trueCMXG,trueCMYG, trueCentroidBBXG+30*DG(1),trueCentroidBBYG+30*DG(2)...
,'linewidth',2,'headwidth',0.25,'headheight',0.33,'color',LineColGArrow,...
'facecolor',LineColGArrow);
```

```
hold on
plot_arrow(trueCMXB,trueCMYB, trueCentroidBBXB+30*DB(1),trueCentroidBBYB+30*DB(2)...
,'linewidth',2,'headwidth',0.25,'headheight',0.33,'color',LineColBArrow,...
'facecolor',LineColBArrow);
```

2.4 Background Estimation

Estimation of the background image required the implementation of a variety of different image processing techniques. This is because the datasets provided presented a number of challenges. One such challenge was that of removing robots that were stationary for a significant portion of a frame sequence. This prevented the simple implementation of a median filter to calculate the background image. This is due to the fact that objects that are stationary for a large subset of the frames will be incorporated into the median and subsequently into the background image.

Therefore, in order to solve this problem, an algorithm has been developed to calculate the background image, regardless of stationary robots being in large subsets of the frame sequence.

The background estimation algorithm consists of three main functional blocks. These blocks are *Channel Processing*, *Region Erasing and Filling* and *Median Filtering* respectively. The *Channel Processing* block receives an image frame and processes the image. This includes blurring, normalising, separating the image into its three channels, subtracting the blue channel from the green channel and renormalising the image. This creates an image that is ready for *Region Erasing and Filling*. In order to implement this functional block, the image channels for the current frame are input into a function called *eraseRegion*. A code snippet of the function is shown below.

```
function [Out] = eraseRegion(Img, C, Size)
    Out = Img;
    %Find the pixels on the vertices of the bounding box
    %surrounding the robot. top—t, bottom—b, left—l, right—r
    t = max(1, C(1) — Size/2);
    b = min(size(Img, 1), C(1) + Size/2);
    l = max(1, C(2) — Size/2);
    r = min(size(Img, 2), C(2) + Size/2);
    %Find the average of the four pixels at the vertices of the
    %bounding box
    Avg = Img(t, l, :) + Img(t, r, :) + Img(b, l, :) + Img(b, r, :);
    Avg = Avg / 4;
    %Replace the image segment (I.e. the robot) with the average
    %of the four pixels
    for y = t:b
        for x = l:r
            Out(y, x, :) = Avg;
        end
    end
```

This function calculates the average value of the pixels found at the corner vertices of the bounding boxes surrounding each robot as shown in *Figure 6*. The corner pixels are averaged across all three colour channels and are stored in the variable *Avg*. All of the pixels in the region containing the robot are then filled with this new average value to produce the image shown in *Figure 6*. This is applied to each of the robots in the image.

The *Region Erasing and Filling* functional block is implemented on the three image frames that are used for background estimation. The reason three image frames have been chosen is because this is the minimum number of frames required to effectively use a median filter which is utilised in the *Median Filtering* functional block. In addition to this, using a small number of frames to compute the median minimises the algorithm's processing time. The median of the three images is subsequently computed to produce an estimation of the background image.

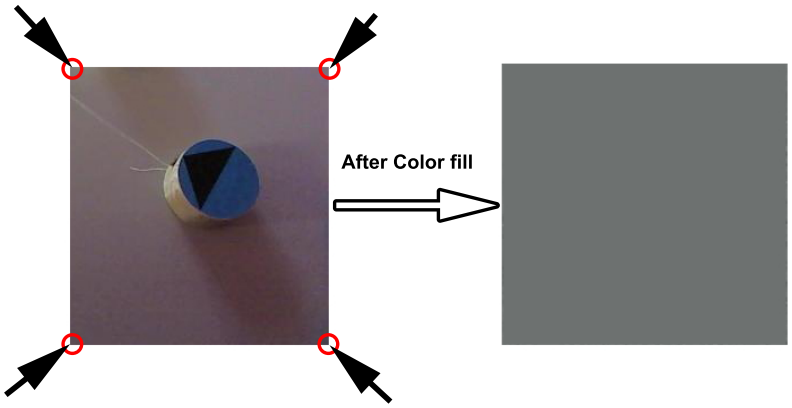

Figure 6: The corner pixel intensities are averaged and are then used to fill the region of the image containing the robot

3 Results

This section details the performance of the image detection algorithms. The algorithms are generally able to detect the robot and determine its orientation. There are some situations where the algorithm cannot achieve this objective. An analysis of the estimated background images are also conducted.

3.1 Robot Detection

Table 1: Results obtained from trying to detect whether or not a robot is in an image frame

	Dataset 1			Dataset 8			Dataset 10		
Type	Red	Green	Blue	Red	Green	Blue	Red	Green	Blue
Correct Detection	83	93	95	95	95	95	95	95	95
Missed Detection	0	0	0	0	0	0	0	0	0
Incorrect Detection	12	2	0	0	0	0	0	0	0

An example of an incorrect detection of a robot due to the robot leaving the image frame.

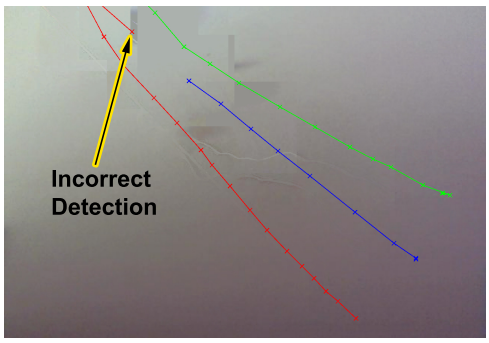

Figure 7: An incorrect detection of the red robot. This track has been linked to the track of the blue robot creating an incorrect track.

Figure 8: The incorrect detection of the red robot as shown in the red channel binary image.

An example of an incorrect detection of a robot due to a region of highly saturated pixels that have a similar colour to that of the robot.

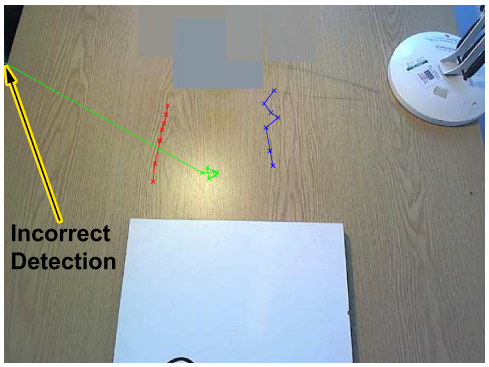

Figure 9: An incorrect detection of the green robot. This results in a broken track

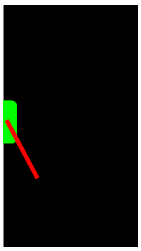

Figure 10: The incorrect detection of the green robot due to a small region of highly saturated pixels.

3.2 Linking Robot Tracks

As detailed in *Section 3.2*, a linking algorithm for the robots has been developed. This algorithm was tested on a variety of datasets and the results are tabulated in *Table 2*. This table details the number of *Correct tracks, Incorrect Tracks* and *Broken Tracks*. A *Broken Track* has been defined as the lack of a connection between two adjacent tracks.

The linking algorithm performed well on dataset 8 and dataset 10. However, dataset 1 had a large number of incorrect and broken tracks, especially on the red channel. These errors occurred as a result of the red robot leaving the image frame. As the robot leaves the frame, the algorithm looks for a set of red pixels that correspond to a red robot. Since the robot is not present, the algorithm identifies the next largest area of red pixels as being the red robot. This is incorrect and causes the results shown in the table.

An example of this is shown in *Figure 7* and *Figure 8*. In this example, the red robot has left the scene and a group of red pixels, as shown in *Figure 8*, have been incorrectly defined as being the red robot. This results in an incorrect linkage between two tracks.

Table 2: Results obtained from linking robot tracks on a variety of different datasets

	Dataset 1			Dataset 8			Dataset 10		
Type	Red	Green	Blue	Red	Green	Blue	Red	Green	Blue
Correct Tracks	83	93	95	95	95	95	95	95	95
Incorrect Tracks	12	2	0	0	0	0	0	0	0
Broken Tracks	0	0	0	0	0	0	0	0	0

Another situation occurs whereby the robot tracks are incorrectly linked. As seen in *Figure 9*, the green robot track has been incorrectly linked to the left hand side of the image frame. This is due to a pixel or group of pixels, p_i having a peak intensity that is higher than the peak intensity of the green robot's pixels. Thus the group of pixels, p_i, will be selected as being the robot object rather than the robot itself. This occurs due to the normalisation of the image as very dark pixels are normalised to high intensity values causing an incorrect linkage between tracks.

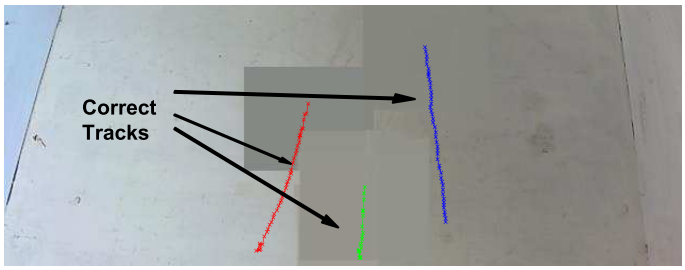

Figure 11: An example of correct robot tracks

3.3 Robot Orientation

The orientation of the robots was tested on a variety of different datasets and the results for each dataset is presented in *Table 3*. There are a number of scenarios whereby the orientation is incorrectly determined causing the arrows to point in incorrect directions. This created a number of missed and incorrect directions as shown in *Table 3*.

An incorrect direction is defined as an arrow pointing at least 45° away, in either direction, from the central vertex of the robot's arrowhead as shown in *Figure 12*. The algorithm is unable to detect that the arrow is pointing in an incorrect direction. An example of this situation is seen in *Figure 13*.

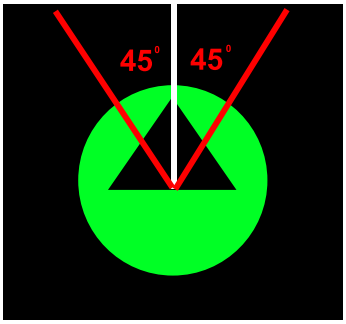

Figure 12: The boundaries for a correct detection

A missed direction is defined as an arrow oriented in an incorrect direction relative to the robot's current direction of motion. However, here the algorithm is able to identify this incorrect orientation and subsequently marks the arrow with a red colour as seen in *Figure 13*.

Figure 14 is an example of one of the shortcomings of the current orientation algorithm. As can be seen in the figure, the bounding box of the robot has been placed around the lower segment of the robot's face. It has completely ignored the arrowhead attached to the robot's upper segment. This is because the upper and lower segments of the robot are separated as there is no 4 or 8 connectivity present between the pixels. The algorithm thus sees the lower segment of the robot as having a larger area and subsequently surrounds that image portion with a bounding box.

This creates another problem. The centroid of the bounding box is now found below the center

Table 3: Results obtained from determining which direction the robot is facing

	Dataset 1			Dataset 8			Dataset 10		
Type	Red	Green	Blue	Red	Green	Blue	Red	Green	Blue
Missed Direction	6	6	2	0	0	0	0	2	4
Incorrect Direction	0	6	0	1	11	3	4	12	2
Correct Direction	89	83	93	94	84	92	91	81	89

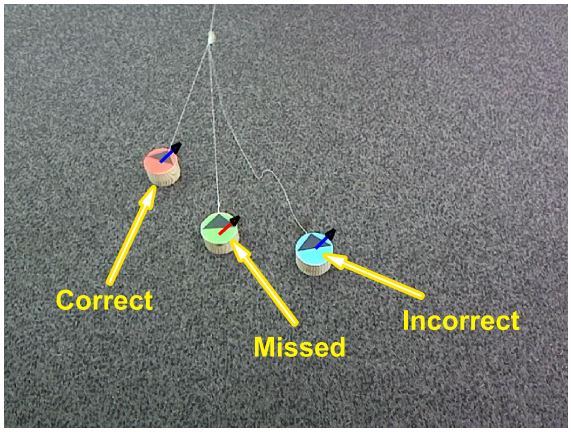

Figure 13: Examples of correct, incorrect and missed arrow orientations respectively

of mass of the robot as seen in the figure. Therefore, when the direction arrow is constructed, the arrow points in the incorrect, and in this case, opposite, direction of the robot's movement.

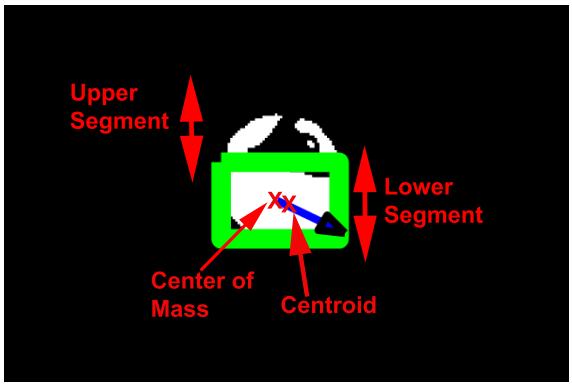

Figure 14: Examples of correct, incorrect and missed arrow orientations respectively

3.4 Background Images

This section analyses the background images produced using the Background Estimation Algorithm detailed in *Section 3.4*. As can be seen in each of the estimated backgrounds from *Figure 15* to *Figure 18*, the presence of the robots have caused a slightly blurred effect on each of the images. This is because the corner pixel averaging, detailed in *Section 3.4*, does not create a perfect representation of the background colour for every pixel that it replaces.

Backgrounds with less texture, such as *Figure 15*, gave better estimation results. This is because these backgrounds have less colour variation and so the corner pixel averaging technique

created a much better representation of the background colour for each pixel.

Figure 15: The estimated background image of the first dataset provided for image processing

Figure 16: The estimated background image of the second dataset provided for image processing

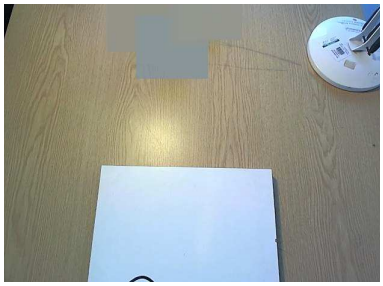

Figure 17: The estimated background image of a new test dataset

Figure 18: The estimated background image of a new test dataset

4 Discussion

This section determines the success of the above-mentioned image processing algorithm. In addition, the limitations and problems present in the current design are discussed, as well as future improvements that could be implemented to enhance the algorithm's performance.

4.1 Success Evaluation

The image processing algorithm performs particularly well on uniform backgrounds. It is able to detect the robot's in almost every frame and maintains a fairly accurate estimate of the robot's current orientation. Textured backgrounds do interfere with processing each robot's orientation. This is partly due to the normalisation procedure performed on each image frame.

The algorithm encounters a problem when a robot leaves the image frame as was the case in dataset 1. Incorrect detections result and incorrect tracks are formed on the image frame. The algorithm also struggles with pixels of high intensity that do not form a part of the robot. If these pixels are the same colour as the robot, they are then incorrectly chosen to represent the robot in the specified image frame.

One of the advantages of the algorithm is that it efficiently processes the images at a relatively high speed. This is due to the efficient implementation of the algorithm in Matlab. This is an important feature if the algorithm is to be exposed to larger datasets.

4.2 Limitations and Problems

A prominent problem presented during this project was that of Image-encoding noise [**?**]. The datasets provided for the project were of a JPEG file type. JPEG images contain compression artefacts which represent data lost due to the compression of the image [**?**]. These artefacts presented a problem as important image information was lost that could have been useful during image processing. The artefacts themselves were also problematic after the image was normalised, producing pixels of high intensity, which was disruptive to the robot detection process.

4.3 Future Improvements

The *normChannel* Matlab function currently performs contrast stretching on each image. This technique is used to stretch the range of pixel intensity values which ultimately increases the contrast of the image. However, this technique is sensitive to pixel outliers [**?**]. The algorithm does not currently take this into account. Doing so in a future implementation would result in an improvement in image contrast.

The current algorithm used to determine the orientation of the robot could also be improved. An alternative approach would be to calculate the center of mass of the arrowhead situated on the robot. Connecting the robot's center of mass with the arrowhead center of mass may provide an improved estimate of the robot's current orientation.

There are also alternative techniques to detecting the robot's in a given image. Once such technique is that of region splitting [**?**]. This technique treats an image as a single region and iteratively splits the image into sub-regions that share the same, or similar, characteristics. This is achieved using a split-and-merge algorithm. As the process iterates, regions that contain similar characteristics are also merged together. This is an effective way to detect different regions as well as objects in an image.

Another effective method that can be used to detect the robots is edge detection. By taking a Laplacian of a Gaussian (LoG), it is possible to detect the edges of an object [**?**]. This technique could be applied to the robot images to effectively detect the robot edges.

5 Conclusion

The image processing algorithm developed to detect robots in an image sequence has been detailed. Various techniques such as normalisation, blurring, contrast stretching and histogram smoothing have been implemented in this algorithm. Various sub-algorithms have been defined such as robot detection, linking tracks in successive image frames as well as determining the robot's current orientation. The algorithm was able to maintain near perfect performance in detecting robots. However, if a robot leaves an image frame or there are highly saturated pixels of the same colour as the robot, then the algorithm struggles to detect the robot creating incorrect tracks. An estimated image of the background has been processed. Blurred patches are present on some of the background images, but this can be minimised. Various improvements have also been suggested such as region splitting for detecting the robots as well as finding the center of mass of the arrowhead in order to create a better orientation arrow.

References

APPENDIX A

A.1 Program Code

A.1.1 Read Image Sequences into Matlab

```
function Imgs = myreadfolder(Folder, Count)
    % Load a first image to use for format and size
    TestImg = myreadimg(Folder, 1);
    % zero-initialise an input matrix of the right size and type
    Imgs = zeros(size(TestImg,1), size(TestImg,2), size(TestImg,3), Count, class(TestImg));
    % read in all the images
    for I = 1:Count
        Imgs(:, :, :, I) = myreadimg(Folder, I);
    end
```

```
function I = myreadimg(Folder, Num)
    I = double(imread([Folder, sprintf('%08d', Num), '.jpg']))/255;
```

A.1.2 Background Estimation Functions

```
function [Out] = eraseRegion(Img, C, Size)
    Out = Img;
    %Find the pixels on the vertices of the bounding box
    %surrounding the robot. top-t, bottom-b, left-l, right-r
    t = max(1, C(1) - Size/2);
    b = min(size(Img, 1), C(1) + Size/2);
    l = max(1, C(2) - Size/2);
    r = min(size(Img, 2), C(2) + Size/2);
    %Find the average of the four pixels at the vertices of the
    %bounding box
    Avg = Img(t, l, :) + Img(t, r, :) + Img(b, l, :) + Img(b, r, :);
    Avg = Avg / 4;
    %Replace the image segment (I.e. the robot) with the average
    %of the four pixels
    for y = t:b
        for x = l:r
            Out(y, x, :) = Avg;
        end
    end
```

A.1.3 Detection Algorithm Helper Functions

```
function [ImgR, ImgG, ImgB] = processChannels(Img, FilterSize, FilterWidth)


    % Blur the image
    FImg = myimgblur(Img, FilterSize, FilterWidth);

    % Normalise the image
    NImg = normalize_rgb(FImg);

    % Split the image into its three colour channels
    ImgR = NImg(:,:,1);
    ImgG = NImg(:,:,2);
```

```
    ImgB = NImg(:,:,3);

    % Subtract the blue channel from the green channel
    % to remove some of the excess blue in the green channel
    ImgG = ImgG - 0.2 * ImgB;

    % Renormalise each channel
    ImgR = normchannel(ImgR);
    ImgG = normchannel(ImgG);
    ImgB = normchannel(ImgB);
```

```
function [Out] = myimgblur(Img, WindowSize, BlurWidth)
    global HaveToolbox;
    if HaveToolbox
        Filter = fspecial('gaussian', [WindowSize WindowSize], BlurWidth);
        Out = imfilter(Img, Filter, 'symmetric', 'conv');
    else
        % Correct between the width parameter of fspecial and the one of
        % mygausswin :(
        BlurWidth = WindowSize / BlurWidth;

        Filter = mygausswin(WindowSize, BlurWidth) * mygausswin(WindowSize, BlurWidth)';
        Filter = Filter / sum(reshape(Filter,1,numel(Filter)));

        % crop the convolved image to original size
        HalfWidth = floor(WindowSize/2);

        MinX = HalfWidth + 1;
        MaxX = size(Img, 2) + HalfWidth;

        MinY = HalfWidth + 1;
        MaxY = size(Img, 1) + HalfWidth;

        Out = Img;
        for i = 1:3
            F = conv2(Img(:,:,i), Filter);
            Out(:,:,i) = F(MinY:MaxY, MinX:MaxX);
        end
    end
```

```
function [OutImg] = normalize_rgb(Img)
    OutImg = zeros(size(Img));
    % SumImg = sum(Img, 3);
    [H,W,C] = size(Img);
    SumImg = zeros(H, W);
    % ColW = [0.3 0.59 0.11] * 3;
    ColW = [1.0 1.0 1.0];
    for i = 1:3
        SumImg = SumImg + Img(:,:,i) .* ColW(i);
    end
    SumImg = SumImg + (SumImg == 0);
    for i = 1:3
        OutImg(:,:,i) = Img(:,:,i) ./ SumImg;
    end
```

```
function [Out] = normchannel(Img)
    Pixels = reshape(Img, 1, numel(Img));
    %Avg = mean(Pixels);
    %Std = std(Pixels);
    %Out = 0.1 * (Img - Avg) / Std;
    Max = max(Pixels);
    Min = min(Pixels);
```

```
    Out = (Img - Min) / (Max - Min);
```

```
function [C, T] = xyhistmax(Img)
    % find the maximum value along each axis
    XHist = max(Img, [], 1)';
    YHist = max(Img, [], 2);
    % convolve it to get some smoothing
    XHist = convhist(XHist, 20, 5);
    YHist = convhist(YHist, 20, 5);

    C = [0 0];
    Max = [0 0];

    % find the maximum along each axis
    Max(1) = max(YHist);
    Max(2) = max(XHist);

    % find the range around this value where the axis max is within some
    % threshold of the max

    YExtent = find(YHist > 0.9 * Max(1));
    XExtent = find(XHist > 0.9 * Max(2));
    C(1) = round(mean(YExtent));
    C(2) = round(mean(XExtent));
    T = 0.75 * min(Max(1), Max(2));

%     hFig = figure(2);
%     clf();
%     set(hFig, 'Position', [0 0 640 480]);
%     axis([0, 640, 0, 1]);
%     hold on;
%     plot(XHist, 'r');
%
%     line([0 640], [Max(2) Max(2)], 'Color', [0 0 0]);
%     line([0 640], [0.9*Max(2) 0.9*Max(2)]);
%     line([0 640], [T T], 'Color', [0 0 0]);
%     line([C(2) C(2)], [0 0.9*Max(2)]);
%     line([min(XExtent) min(XExtent)], [0 0.9*Max(2)]);
%     line([max(XExtent) max(XExtent)], [0 0.9*Max(2)]);
%     pause(3);
```

```
% find a threshold from a histogram by smoothing with a gaussian with
% standard deviation sigma and find the low valley location
% sizeparam should be at least 4, with larger giving less smoothing
function thresh = findthresh(thehist)

  [len,x] = size(thehist);

  tmp1=thehist;

  % find largest peak
  peak = find(tmp1 == max(tmp1));

  % find highest peak to left
  xmaxl = -1;
  pkl = -1;
  for i = 2 : peak-1
    if tmp1(i-1) < tmp1(i) & tmp1(i) ≥ tmp1(i+1)
      if tmp1(i) > xmaxl
        xmaxl = tmp1(i);
        pkl = i;
      end
    end
  end
```

```
  if pkl == -1
    pkl = 1;
    xmaxl = 1;
  end
%    [pkl,xmaxl]

  % find deepest valley between peaks
  xminl = max(tmp1)+1;
  vall = -1;
  for i = pkl+1 : peak-1
    if tmp1(i-1) > tmp1(i) & tmp1(i) <= tmp1(i+1)
      if tmp1(i) < xminl
        xminl = tmp1(i);
        vall = i;
      end
    end
  end
  if vall == -1
    vall = 2;
    xminl = 2;
  end
%    [vall,xminl]

  % find highest peak to right
  xmaxr = -1;
  pkr = -1;
  for i = peak+1 : len-1
    if tmp1(i-1) < tmp1(i) & tmp1(i) >= tmp1(i+1)
      if tmp1(i) > xmaxr
        xmaxr = tmp1(i);
        pkr = i;
      end
    end
  end
  if pkr == -1
    pkr = len;
    xmaxr = 1;
  end
%    [pkr,xmaxr]

  % find deepest valley between peaks
  xminr = max(tmp1)+1;
  valr = -1;
  for i = peak+1 : pkr-1
    if tmp1(i-1) > tmp1(i) & tmp1(i) <= tmp1(i+1)
      if tmp1(i) < xminr
        xminr = tmp1(i);
        valr = i;
      end
    end
  end
  if valr == -1
    valr = len-1;
    xminr = 2;
  end
%    [valr,xminr]

  % find lowest point between peaks
  if xmaxr > xmaxl
      thresh = valr;
  else
      thresh = vall;
  end


  thresh = thresh-1;    % subtract 1 as histogram bin 1 is for value 0
```

```
function [Out] = cliprect(Img, Centre, Size)
    YMin = round(max(1, Centre(1) - Size/2));
    YMax = round(min(size(Img, 1), Centre(1) + Size/2));
    XMin = round(max(1, Centre(2) - Size/2));
    XMax = round(min(size(Img, 2), Centre(2) + Size/2));
    Out = Img(YMin:YMax, XMin:XMax);
end
```

A.1.4 Linking Algorithm Functions

```
function [verticesX, verticesY, centroidX, falseImageX] = calcBoundingBox(TImgX)
global HaveToolbox;
%Set the image thresholds used to determine
%if the detected object is indeed a robot
thresholdUpper=4000;
thresholdLower=1500;
falseImageX=0;

%Label the image
[labelX, numXBlobs] = mybwlabel(TImgX);

%Initialise the centroid variable to hold the coordinates
%of the bounding box
centroidX = zeros(1,2);

%Then extract the various properties from the image

%if the image toolbox is available
if HaveToolbox==1
blobMeasurements = regionprops(labelX, ['basic']);
if size(blobMeasurements,1) == 0
    verticesX = [0 0 0 0 0];
    verticesY = [0 0 0 0 0];
    centroidX = [0 0];
    return;
end
end

%Calculate the area for each blob obtained from 'labelX'
allBlobAreas = [];
for i=1:numXBlobs
    [rows,cols,vals] = find(labelX==i);
    blobSize = sum(vals);
    allBlobAreas = [allBlobAreas blobSize];

end

%Find the blob with the maximum area in the image
[maxBlobArea, index] = max(allBlobAreas);

%If there are no 'blobs'(objects) in the image
if numXBlobs == 0 || maxBlobArea==0
    verticesX = [0 0 0 0 0];
    verticesY = [0 0 0 0 0];
    centroidX = [0 0];
    return;
end


if HaveToolbox==1
%Extract the bounding box from the image toolbox
boundingBox = blobMeasurements(index).BoundingBox;
x1 = boundingBox(1);
```

```
y1 = boundingBox(2);
x2 = x1 + boundingBox(3) - 1;
y2 = y1 + boundingBox(4) - 1;
elseif HaveToolbox==0
%Without using the image toolkit, extract the bounding box
[rows,cols,vals] = find(labelX==index);
x1 = min(cols);
x2 = max(cols);
y1 = min(rows);
y2 = max(rows);
end
%determine the vertices of the bounding box
verticesX = [x1 x2 x2 x1 x1];
verticesY = [y1 y1 y2 y2 y1];
%Determine the centroid of the bounding box surrounding the robot
centroidX(1,1) = (x2+x1)/2;
centroidX(1,2) = (y2+y1)/2;

%************************************************
%Not fully implemented as it was found that this
%algorithm depends on the camera position

%Check the area of the blob to determine
%whether the blob falls within the correct region.
    if(maxBlobArea> thresholdUpper || ...
            maxBlobArea<thresholdLower)
        %If the blob is greater than the predetermined
        %threshold size of the robot, then the blob
        %is probably not a robot
        falseImageX= 1;
    else
        falseImageX=0;
    end
 %************************************************


end
```

```
function [centerMassX, centerMassY] = calcBoundingBoxCM(verticesX, verticesY, TImg)

%Initialise Area counter to determine how many pixels
%the robot in the image contains
areaCounter = 0;

%The center of mass coordinates for the robot are initialised
centerMassX = 0;
centerMassY = 0;

%Convert vertices of type double to integers
verticesX = round(verticesX);
verticesY = round(verticesY);

%Calculate the area of the object within the bounding Box

for x = verticesX(1):verticesX(2)
    for y = verticesY(1):verticesY(3)
        TImg(y,x);
        if(TImg(y,x)==1)

            areaCounter  = areaCounter+1;
        else
            areaCounter;
        end
    end
end

```

```
%Calculate the center of mass for the image
centerMass=zeros(1,2);
for r=verticesX(1):verticesX(2)
    for c=verticesY(1):verticesY(3)
        centerMass(1,1) = centerMass(1,1)+r*TImg(c,r);
        centerMass(1,2) = centerMass(1,2)+c*TImg(c,r);
    end
end
%The x coordinate center of mass
centerMassX= centerMass(1,1)/areaCounter;
%The y coordinate center of mass
centerMassY= centerMass(1,2)/areaCounter;
```

A.1.5 Orientation Algorithm Third Party Helper Function

```
function handles = plot_arrow( x1,y1,x2,y2,varargin )
%
% plot_arrow - plots an arrow to the current plot
%
% format:   handles = plot_arrow( x1,y1,x2,y2 [,options...] )
%
% input:    x1,y1   - starting point
%           x2,y2   - end point
%           options - come as pairs of "property","value" as defined for "line" and "patch"
%                     controls, see matlab help for listing of these properties.
%                     note that not all properties where added, one might add them at the end of this file
%
%                     additional options are:
%                     'headwidth':  relative to complete arrow size, default value is 0.07
%                     'headheight': relative to complete arrow size, default value is 0.15
%                     (encoded are maximal values if pixels, for the case that the arrow is very long)
%
% output:   handles - handles of the graphical elements building the arrow
%
% Example:  plot_arrow( -1,-1,15,12,'linewidth',2,'color',[0.5 0.5 0.5],'facecolor',[0.5 0.5 0.5] );
%           plot_arrow( 0,0,5,4,'linewidth',2,'headwidth',0.25,'headheight',0.33 );
%           plot_arrow;   % will launch demo

% =============================================
% for debug - demo - can be erased
% =============================================
if (nargin==0)
    figure;
    axis;
    set( gca,'nextplot','add' );
    for x = 0:0.3:2*pi
        color = [rand rand rand];
        h = plot_arrow( 1,1,50*rand*cos(x),50*rand*sin(x),...
            'color',color,'facecolor',color,'edgecolor',color );
        set( h,'linewidth',2 );
    end
    hold off;
    return
end
% =============================================
% end of for debug
% =============================================


% =============================================
% constants (can be edited)
% =============================================
alpha       = 0.15;   % head length
beta        = 0.07;   % head width
```

```
max_length  = 22;
max_width   = 10;

% =====================================================
% check if head properties are given
% =====================================================
% if ratio is always fixed, this section can be removed!
if ¬isempty( varargin )
    for c = 1:floor(length(varargin)/2)
        try
            switch lower(varargin{c*2−1})
                % head properties − do nothing, since handled above already
            case 'headheight',alpha = max( min( varargin{c*2},1 ),0.01 );
            case 'headwidth', beta = max( min( varargin{c*2},1 ),0.01 );
            end
        catch
            fprintf( 'unrecognized property or value for: %s\n',varargin{c*2−1} );
        end
    end
end

% =====================================================
% calculate the arrow head coordinates
% =====================================================
den         = x2 − x1 + eps;                                % make sure no devision by zero occurs
teta        = atan( (y2−y1)/den ) + pi*(x2<x1) − pi/2;      % angle of arrow
cs          = cos(teta);                                    % rotation matrix
ss          = sin(teta);
R           = [cs −ss;ss cs];
line_length = sqrt( (y2−y1)^2 + (x2−x1)^2 );                % sizes
head_length = min( line_length*alpha,max_length );
head_width  = min( line_length*beta,max_length );
x0          = x2*cs + y2*ss;                                % build head coordinats
y0          = −x2*ss + y2*cs;
coords      = R*[x0 x0+head_width/2 x0−head_width/2; y0 y0−head_length y0−head_length];

% =====================================================
% plot arrow  (= line + patch of a triangle)
% =====================================================
h1          = plot( [x1,x2],[y1,y2],'k' );
h2          = patch( coords(1,:),coords(2,:),[0 0 0] );

% =====================================================
% return handles
% =====================================================
handles = [h1 h2];

% =====================================================
% check if styling is required
% =====================================================
% if no styling, this section can be removed!
if ¬isempty( varargin )
    for c = 1:floor(length(varargin)/2)
        try
            switch lower(varargin{c*2−1})

             % only patch properties
            case 'edgecolor',   set( h2,'EdgeColor',varargin{c*2} );
            case 'facecolor',   set( h2,'FaceColor',varargin{c*2} );
            case 'facelighting',set( h2,'FaceLighting',varargin{c*2} );
            case 'edgelighting',set( h2,'EdgeLighting',varargin{c*2} );

            % only line properties
            case 'color'    , set( h1,'Color',varargin{c*2} );

            % shared properties
            case 'linestyle', set( handles,'LineStyle',varargin{c*2} );
```

```
            case 'linewidth', set( handles,'LineWidth',varargin{c*2} );
            case 'parent',    set( handles,'parent',varargin{c*2} );

            % head properties - do nothing, since handled above already
            case 'headwidth',;
            case 'headheight',;

            end
        catch
            fprintf( 'unrecognized property or value for: %s\n',varargin{c*2-1} );
        end
    end
end
```

A.1.6 Main Program Code

```
% Clear any existing state
clear all;
clc;
clf;
tic

% Configuration section
DATA_FOLDER = 'data10/';
ENABLE_TOOLBOX = 1;
IMG_SKIP = 5;
IMG_STEP = 1;
MAX_IMG_COUNT = 100;
FILTER_SIZE = 5;
FILTER_WIDTH = 5;
BB_SIZE = 120;

% Attempt to acquire a toolbox license
global HaveToolbox;
HaveToolbox = ENABLE_TOOLBOX && license('checkout', 'Image_Toolbox');

% Load in the dataset
ImgData = myreadfolder(DATA_FOLDER, MAX_IMG_COUNT);

% Compute the median image
MedianIndices = [5 85 95];
MedianImgs = zeros(size(ImgData,1),    ...
                   size(ImgData,2),    ...
                   size(ImgData,3),    ...
                   size(MedianIndices, 2), ...
                   class(ImgData));


for Idx = 1:size(MedianIndices,2)
    ImgIdx = MedianIndices(Idx);

    Img = ImgData(:,:,:,ImgIdx);

    [ImgR, ImgG, ImgB] = processChannels(Img, FILTER_SIZE, FILTER_WIDTH);

    % Marginal Histogram along X-axis, Y-axis
    [CR, ThreshR] = xyhistmax(ImgR);
    [CG, ThreshG] = xyhistmax(ImgG);
    [CB, ThreshB] = xyhistmax(ImgB);

    Img = eraseRegion(Img, CR, BB_SIZE);
    Img = eraseRegion(Img, CG, BB_SIZE);
    Img = eraseRegion(Img, CB, BB_SIZE);

```

```
    MedianImgs(:,:,:,Idx) = Img;
end

MedImg =  median(MedianImgs, 4);

OldDirR = [0 0];
OldDirG = [0 0];
OldDirB = [0 0];

%*****************************************************
%The variables used to link the objects movement

XLinkerR = [];
YLinkerR = [];
XLinkerG = [];
YLinkerG = [];
XLinkerB = [];
YLinkerB = [];

MissCount = 0;

for ImgIdx = IMG_SKIP:IMG_STEP:MAX_IMG_COUNT
    Img = ImgData(:,:,:,ImgIdx);

    [ImgR, ImgG, ImgB] = processChannels(Img, FILTER_SIZE, FILTER_WIDTH);

    % Marginal Histogram along X-axis, Y-axis
    [CR, ThreshR] = xyhistmax(ImgR);
    [CG, ThreshG] = xyhistmax(ImgG);
    [CB, ThreshB] = xyhistmax(ImgB);

    TImgR = cliprect(ImgR, CR, BB_SIZE)>ThreshR;
    TImgG = cliprect(ImgG, CG, BB_SIZE)>ThreshG;
    TImgB = cliprect(ImgB, CB, BB_SIZE)>ThreshB;

    %*****************************************************
    %Calculate and display the bounding box for the image
    %Relative to the clipped image from the cliprect function
    [VerticesXR, VerticesYR, CentroidR, FalseImageR] = calcBoundingBox(TImgR);
    [VerticesXG, VerticesYG, CentroidG, FalseImageG] = calcBoundingBox(TImgG);
    [VerticesXB, VerticesYB, CentroidB, FalseImageB] = calcBoundingBox(TImgB);

    %*****************************************************
    %If no image is detected, skip the current frame
    if min(VerticesXR) == 0 || min(VerticesXG) == 0 || min(VerticesXB) == 0
        MissCount = MissCount + 1;
        continue;
    end
    %Calculate the center of mass of the robot within a
    %bounding box
    [CenterMassXR,CenterMassYR] = calcBoundingBoxCM(VerticesXR, VerticesYR, TImgR);
    [CenterMassXG,CenterMassYG] = calcBoundingBoxCM(VerticesXG, VerticesYG, TImgG);
    [CenterMassXB,CenterMassYB] = calcBoundingBoxCM(VerticesXB, VerticesYB, TImgB);

    %Add variables for the center of mass of the
    %robot relative to the original image
    trueCMXR = (CenterMassXR + CR(2) - size(TImgR, 2)/2);
    trueCMYR = (CenterMassYR + CR(1) - size(TImgR, 1)/2);
    trueCMXG = (CenterMassXG + CG(2) - size(TImgG, 2)/2);
    trueCMYG = (CenterMassYG + CG(1) - size(TImgG, 1)/2);
    trueCMXB = (CenterMassXB + CB(2) - size(TImgB, 2)/2);
    trueCMYB = (CenterMassYB + CB(1) - size(TImgB, 1)/2);

    %Store the true position of the bounding box centroid
    %relative to the coordinates of the original image
    trueCentroidBBXR = (CentroidR(1) + CR(2) - size(TImgR, 2)/2);
    trueCentroidBBYR = (CentroidR(2) + CR(1) - size(TImgR, 1)/2);
```

```
116      trueCentroidBBXG = (CentroidG(1) + CG(2) - size(TImgG, 2)/2);
117      trueCentroidBBYG = (CentroidG(2) + CG(1) - size(TImgG, 1)/2);
118      trueCentroidBBXB = (CentroidB(1) + CB(2) - size(TImgB, 2)/2);
119      trueCentroidBBYB = (CentroidB(2) + CB(1) - size(TImgB, 1)/2);
120
121      %*********************************************
122      %To link tracks on the estimated background image
123      XLinkerR = [XLinkerR trueCMXR];
124      YLinkerR = [YLinkerR trueCMYR];
125      XLinkerG = [XLinkerG trueCMXG];
126      YLinkerG = [YLinkerG trueCMYG];
127      XLinkerB = [XLinkerB trueCMXB];
128      YLinkerB = [YLinkerB trueCMYB];
129
130      %Define unit vectors in the direction of the bounding
131      %box centroid for each of the rgb channels respectively
132      CenterMassR = [CenterMassXR CenterMassYR];
133      DR = (CentroidR - CenterMassR);
134
135      CenterMassG = [CenterMassXG CenterMassYG];
136      DG = (CentroidG - CenterMassG);
137
138      CenterMassB = [CenterMassXB CenterMassYB];
139      DB = (CentroidB - CenterMassB);
140
141      %Calculate the unit vectors for each channel
142      DR = DR/norm(DR);
143      DG = DG/norm(DG);
144      DB = DB/norm(DB);
145
146      %Colour for the orientation arrows in the bouding box
147      LineColR = 'b-';
148      LineColG = 'b-';
149      LineColB = 'b-';
150
151      %Colour for the orientation arrows on the main image
152      LineColRArrow = [0 0 1];
153      LineColGArrow = [0 0 1];
154      LineColBArrow = [0 0 1];
155   %Calculate the dot product between the arrows current
156   %orientation and its previous orientation. If the value
157   %is less than 0.5, then the arrow is pointing in an incorrect
158   %direction
159      if dot(DR, OldDirR) < 0.5
160    %Arrow is pointing in an incorrect direction,
161    %therefore make the arrow red to indicate this.
162          LineColR = 'r-';
163          LineColRArrow = [1 0 0];
164      end
165      if dot(DG, OldDirG) < 0.5
166          LineColG = 'r-';
167          LineColGArrow = [1 0 0];
168      end
169      if dot(DB, OldDirB) < 0.5
170          LineColB = 'r-';
171          LineColBArrow = [1 0 0];
172      end
173      OldDirR = DR;
174      OldDirG = DG;
175      OldDirB = DB;
176
177      figure(1);
178      clf();
179      %*****************************************************
180      %Plotting the main images
181
182      subplot(3,3,1:6);
```

```
    %myimshow(MedImg);
    myimshow(Img);
    hold on
    plot(XLinkerR, YLinkerR, 'xr-', XLinkerG, YLinkerG, 'xg-', XLinkerB, YLinkerB, 'xb-');
    xlabel(ImgIdx);
    myimshow(Img);
    colormap('default');
    hold on;
    %Plot the arrow on each of the cars to indicate their
    %orientation
    plot_arrow(trueCMXR,trueCMYR, trueCentroidBBXR+30*DR(1),trueCentroidBBYR+30*DR(2),'linewidth',2,'headw
    hold on
    plot_arrow(trueCMXG,trueCMYG, trueCentroidBBXG+30*DG(1),trueCentroidBBYG+30*DG(2),'linewidth',2,'headw
    hold on
    plot_arrow(trueCMXB,trueCMYB, trueCentroidBBXB+30*DB(1),trueCentroidBBYB+30*DB(2),'linewidth',2,'headw

    %Plot the red channel with its corresponding bounding
    %box
    subplot(3,3,7);
    myimshow(TImgR);
    colormap('gray');
    hold on;
    plot(VerticesXR, VerticesYR, 'r-', 'LineWidth', 5);
    plot_arrow(CenterMassR(1),CenterMassR(2), CentroidR(1)+30*DR(1),CentroidR(2)+30*DR(2),'linewidth',2,'h
    %plot([CenterMassR(1),CentroidR(1)+30*DR(1)], [CenterMassR(2), CentroidR(2)+30*DR(2)], LineColR, 'Line
    xlabel('Red Channel');

    %Plot the green channel with its corresponding bounding
    %box
    subplot(3,3,8);
    myimshow(TImgG);
    colormap('gray');
    hold on;
    plot(VerticesXG, VerticesYG, 'g-', 'LineWidth', 5);
    plot_arrow(CenterMassG(1),CenterMassG(2), CentroidG(1)+30*DG(1),CentroidG(2)+30*DG(2),'linewidth',2,'h
    %plot([CenterMassG(1),CentroidG(1)+30*DG(1)], [CenterMassG(2), CentroidG(2)+30*DG(2)], LineColG, 'Line
    xlabel('Green Channel');

    %Plot the blue channel with its corresponding bounding
    %box
    subplot(3,3,9);
    myimshow(TImgB);
    colormap('gray');
    hold on;
    plot(VerticesXB, VerticesYB, 'b-', 'LineWidth', 5);
    plot_arrow(CenterMassB(1),CenterMassB(2), CentroidB(1)+30*DB(1),CentroidB(2)+30*DB(2),'linewidth',2,'h
    %plot([CenterMassB(1),CentroidB(1)+30*DB(1)], [CenterMassB(2), CentroidB(2)+30*DB(2)], LineColB, 'Line
    xlabel('Blue Channel');
    %*****************************************************
    %pause(0.1);
    input('...');
end
%*****************************************************
%Once the algorithm is finished, plot the background
%image with the linked tracks
clf();
myimshow(MedImg);
hold on
plot(XLinkerR, YLinkerR, 'xr-', XLinkerG, YLinkerG, 'xg-', XLinkerB, YLinkerB, 'xb-');
xlabel(ImgIdx);
input(int2str(MissCount));
toc
```