Introduction to Vision and Robotics
Assessed Practical 1: Robot Tracking

Ran Guan Thea Koutsoukis

4pm March 7, 2013

Contents

1 Introduction

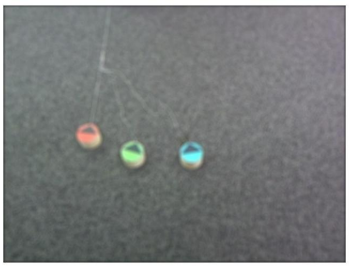
Our strategy for this task was to smooth the image, normalise the colors to account for the difference in illumination, set a threshold for the our definition of red, green and blue and then focus on a small section of the robot to get direction and trajectory

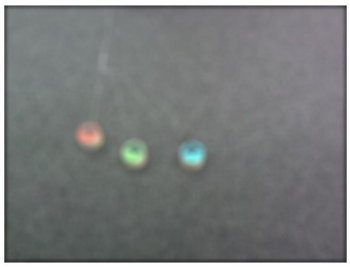
2 Methods

2.1 Smoothing

We smoothed the image to remove noise in the image using two built-in matlab methods, fspecial and imfilter. In detectSingleImage, we used fspecial to create a circular averaging filter with a radius of 10. In deciding whether to use the circular or gaussian filter, two factors came into play. First, the robots we are trying to detect are so distinct from the background, and the nature of the scene is so artificial that the distribution of color pixels is closer to a step function than a gaussian function. This meant the circular filter could perform just as well as the gaussian but potentially faster. Secondly, the circular filter takes less arguments (just a radius size, as opposed to the filter size and sigma arguments needed by gaussian), meaning that experimenting with different values was easier. The radius was found by trial and error, a smaller radius did produce a clearer robot, but caused problems with strings and shadows still being too visible. imfilter then applied this circular filter to the image.

Smoothing with radius 5 and 10:





2.2 Normalisation

To cope with varying lighting, we normalised the image through the function normaliseColor. This takes an image and brightens it up, using the formula for RGB normalisation given in the lectures. A value too small produced a completely black image, whilst a too large value caused the image to be overexposed. Experimentation proved 500 to be the optimal value.

2.3 Robot Detection

First of all, we approximated a background image. This involved finding the average rgb value for the background using meanBG, by summing up the rgb value of each pixel and dividing by the number of pixels. Then we examined each pixel of the image. If the color value of this pixel was close to the mean, we set it as the background. Otherwise it was set to being part of the foreground. The "distance", or similarity of one color to another was computed using our colorSimilarity function. This is a normalised euclidean distance, as each color behaves differently. A small distance between the r values for a particular pixel means more than the same distance between b values, so dividing by a different constant for each returns a more accurate similarity. The foreground image was then passed through the function labelRegion. The image was converted to black and white, and if the black region was large enough it was determined to be a robot. To label each specific color, a pixel in the original image was examined and the highest color channel determined the color of the robot.

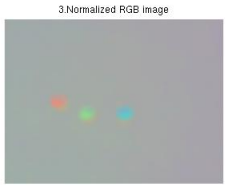
2.4 Direction

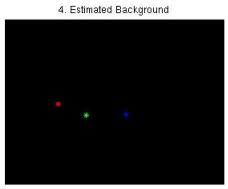
The direction of the robot was found by isolating a small section in the middle of the robot and drawing a straight line from the centre to the midpoint of the black. In most cases, this was accurate enough to correctly determine the direction, however we occasionally ran into trouble when not enough of the black was picked up. This meant the line wasn't being drawn to the tip of the arrow, but to somewhere in the middle.

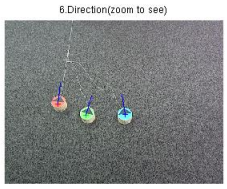
Region of Interest and Computed Direction: (note how not enough black was picked up on the red robot, thus the direction is slightly off)

3 Results

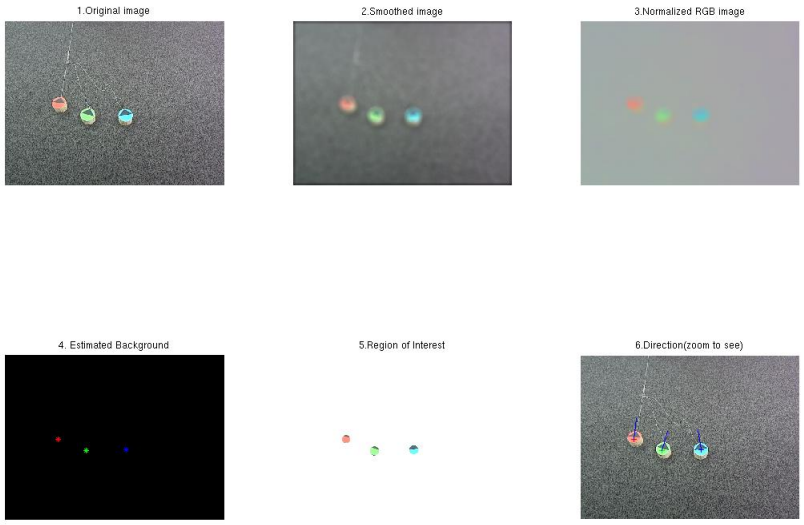
3.1 Dataset 1



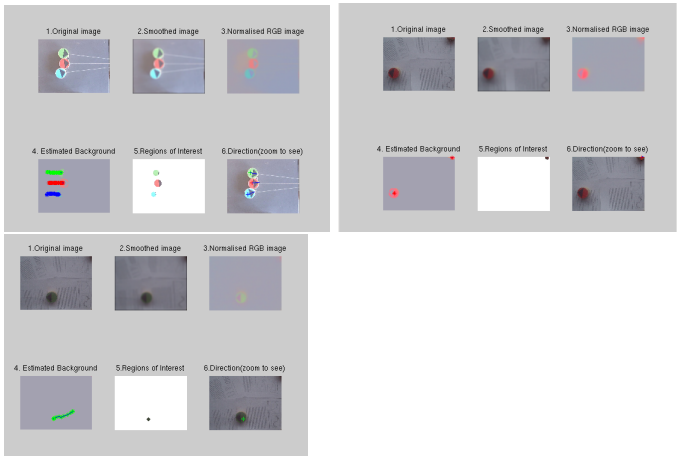




3.2 Dataset 2



3.3 Own Data



Our program worked surprisingly well for our own datasets. A few interesting examples are included here. In the datasets with more than one robot of each color, our program did not succeed for red, but did for green and blue.

4 Discussion

Our program is successful once we fine tune the constants for each particular data set. Unfortunately, we couldn't get adaptive thresholding to accurately work any better than just using the constants that had been set for another set of images. Thus, if our program was tried out on different sized robots, it would likely not perform very well. We would have liked to improve the estimated background image. At the moment, it's only a crude approximation, and only works well on uniform backgrounds. Large shadows and variations in texture and lighting don't show up, so there is definitely room for improvement there.

5 Code

euclidianDist2D

```
function distance = euclideanDist2D(x1,y1,x2,y2)

distance = sqrt(((x1-x2))^2 + ((y1-y2))^2);

end
```

colorSimilarity

```
% Finds the normalised distance between coloured points (r1,g1,b1) and
% (r2,g2,b2)
% Values were found by trial and error

function distance = colorSimilarity(r1,g1,b1,r2,g2,b2)

a = 2.2;
b = 0.75;
c = 0.75;

distance = sqrt(((r1-r2)/a)^2 + ((g1-g2)/b)^2 + ((b1-b2)/c)^2);

end
```

labelRegion

```
% Takes an image and returns the x,y co-ordinates and radius of the red,
% blue and green robots
function [rx,ry,rr,gx,gy,gr,bx,by,br] = labelRegion(img)

% Convert image to grayscale
grayImage = rgb2gray(img);
% Converts grayscale image to black and white using a threshold level of 0.8
thresholdImage = im2bw(grayImage,0.88);
%
L = bwlabel(~thresholdImage,8);
N = max(L(:));

j=1;
rx = 0;
ry = 0;
rr = 0;
gx = 0;
gy = 0;
gr = 0;
```

```
bx = 0;
by = 0;
br = 0;

for i= 1:N
    [r,c]=find(L==i);
    if(size(r,1)>200)
        c1(j)=sum(r)/size(r,1);
        c2(j)=sum(c)/size(c,1);
        if(sum(img(r,c,1))/size(r,1)>200);
        rx=c1(j);
        ry=c2(j);
        plot(c2(j),c1(j),'r*');
        rr=sqrt(size(r,1))/2;
        elseif(sum(img(r,c,3))/size(r,1)>180);
        plot(c2(j),c1(j),'b*');
        bx=c1(j);
        by=c2(j);
        br=sqrt(size(r,1))/2;
        elseif(sum(img(r,c,2))/size(r,1)>180);
        plot(c2(j),c1(j),'g*');
        gx=c1(j);
        gy=c2(j);
        gr=sqrt(size(r,1))/2;
        end
        j=j+1;
    end
end
end
```

normaliseColor

```
% Normalises Colour by brightening up the image

function normalisedImage = normaliseColor(img)
normalisedImage = double(img);

[m,n,l]=size(img);
       for i=1:m
           for j=1:n
               % illuIntensity is the sum of the r, g and b values at each
               % pixel
               illuIntensity = normalisedImage(i,j,1) + normalisedImage(i,j,2) + normalisedImage(i,j
               % the image is normalised by replacing each pixel with one
               % that has been illuminated
               normalisedImage(i,j,1) = normalisedImage(i,j,1)/illuIntensity*500;
               normalisedImage(i,j,2) = normalisedImage(i,j,2)/illuIntensity*500;
               normalisedImage(i,j,3) = normalisedImage(i,j,3)/illuIntensity*500;
            end
       end
end
```

meanBG

```
function [red,green,blue]=meanBG(img)
[m,n,l]=size(img);
img=double(img);
red=0;
green=0;
blue=0;
        for i=1:m
            for j=1:n
                red=red+img(i,j,1);
                green=green+img(i,j,2);
                blue=blue+img(i,j,3);
            end
        end
red=red/(m*n);
green=green/(m*n);
blue=blue/(m*n);
end
```

diffFromBG

```
% Takes an image and returns the background

function [foregroundImage,backgroundImage]=diffFromBG(img)

foregroundImage=zeros(size(img));
backgroundImage=zeros(size(img));
[r,g,b]=meanBG(img);
[m,n,l]=size(foregroundImage);
        for i=1:m
            for j=1:n
                if(colorSimilarity(r,g,b,img(i,j,1),img(i,j,2),img(i,j,3))<45)
                    foregroundImage(i,j,1)=255;
                    foregroundImage(i,j,2)=255;
                    foregroundImage(i,j,3)=255;
                    backgroundImage(i,j,1)=round(r);
                    backgroundImage(i,j,2)=round(g);
                    backgroundImage(i,j,3)=round(b);
                end
                if(colorSimilarity(r,g,b,img(i,j,1),img(i,j,2),img(i,j,3))>=45)
                    foregroundImage(i,j,1)=img(i,j,1);
                    foregroundImage(i,j,2)=img(i,j,2);
                    foregroundImage(i,j,3)=img(i,j,3);
                    backgroundImage(i,j,1)=img(i,j,1);
                    backgroundImage(i,j,2)=img(i,j,2);
                    backgroundImage(i,j,3)=img(i,j,3);
                end
            end
        end
end
```

segmentFromRadius

```
% Isolates the center of mass of the robot
function [centreOfRobot,rdx,rdy,gdx,gdy,bdx,bdy] = centreOfMass(img,rx,ry,rr,gx,gy,gr,bx,by,br)
```

```
rdx=0;
rdy=0;
bdx=0;
bdy=0;
gdx=0;
gdy=0;
rsum=0;
bsum=0;
gsum=0;

centreOfRobot = zeros(size(img));
[m,n,l] = size(newimage);
        for i=1:m
            for j=1:n
                if(euclideanDist2D(rx,ry,i,j)<rr)
                    centreOfRobot(i,j,1)=img(i,j,1);
                    centreOfRobot(i,j,2)=img(i,j,2);
                    centreOfRobot(i,j,3)=img(i,j,3);
                    if(img(i,j,1)<(img(round(rx),round(ry),1)-50))
                        rdx=rdx+i;
                        rdy=rdy+j;
                        rsum=rsum+1;
                    end
                elseif(euclideanDist2D(bx,by,i,j)<br)
                    centreOfRobot(i,j,1)=img(i,j,1);
                    centreOfRobot(i,j,2)=img(i,j,2);
                    centreOfRobot(i,j,3)=img(i,j,3);
                    if(img(i,j,3)<(img(round(bx),round(by),3)-50))
                        bdx=bdx+i;
                        bdy=bdy+j;
                        bsum=bsum+1;
                    end
                elseif(euclideanDist2D(gx,gy,i,j)<gr)
                    centreOfRobot(i,j,1)=img(i,j,1);
                    centreOfRobot(i,j,2)=img(i,j,2);
                    centreOfRobot(i,j,3)=img(i,j,3);
                    if(img(i,j,2)<(img(round(gx),round(gy),2)-50))
                        gdx=gdx+i;
                        gdy=gdy+j;
                        gsum=gsum+1;
                    end
                else
                    centreOfRobot(i,j,1)=255;
                    centreOfRobot(i,j,2)=255;
                    centreOfRobot(i,j,3)=255;
                end
            end
        end
        rdx=rdx/rsum;
        rdy=rdy/rsum;
        gdx=gdx/gsum;
        gdy=gdy/gsum;
        bdx=bdx/bsum;
        bdy=bdy/bsum;
end
```

detectSingleImage

```
% Takes a single image and returns the x,y co—ordinates and radius of the
% red, blue and green robot for use by trackAllImages

function [rx,ry,rr,gx,gy,gr,bx,by,br] = detectSingleImage(img)

% Creates a 2D circular average filter with radius 10 and applies it to the
% image
filter = fspecial('disk',10);
smoothedImage = imfilter(img,filter);

% Normalises the image
normalisedImage = normaliseColor(smoothedImage);

% Isolates the background & foreground images
[foregroundImage,backgroundImage] = diffFromBG(normalisedImage);

% Plots all versions of the image

subplot(2,3,1);
imshow(img);
title('1.Original image')

subplot(2,3,2);
imshow(uint8(smoothedImage));
title('2.Smoothed image')

subplot(2,3,3);
imshow(uint8(normalisedImage));
title('3.Normalised RGB image')

subplot(2,3,4);
imshow(uint8(backgroundImage));
title('4. Tracking')

hold on;
[rx,ry,rr,gx,gy,gr,bx,by,br] = labelRegion(uint8(foregroundImage));
[regionsOfInterest,rdx,rdy,gdx,gdy,bdx,bdy] = centreOfRobot(img,rx,ry,rr,gx,gy,gr,bx,by,br);

subplot(2,3,5);
imshow(uint8(regionsOfInterest));
title('5.Regions of Interest');

subplot(2,3,6);
imshow(img);
title('6.Direction')

%rdx = centre of black part)
hold on;
plot(ry,rx,'r+');
line([(rdy—ry)*60/sqrt((rdy—ry)^2+(rdx—rx)^2)+ry,ry],[(rdx—rx)*60/sqrt((rdy—ry)^2+(rdx—rx)^2)+rx,rx]
plot(by,bx,'b+');
line([(bdy—by)*60/sqrt((bdy—by)^2+(bdx—bx)^2)+by,by],[(bdx—bx)*60/sqrt((bdy—by)^2+(bdx—bx)^2)+bx,bx]
plot(gy,gx,'g+');
line([(gdy—gy)*60/sqrt((gdy—gy)^2+(gdx—gx)^2)+gy,gy],[(gdx—gx)*60/sqrt((gdy—gy)^2+(gdx—gx)^2)+gx,gx]
```

```
subplot(2,3,4);
end
```

trackAllImages

```
% Function takes all images in the same directory as the code and applies
% detectSingleImage to each one, returning the complete trajectory the
% robots take

function [trajArrayRx,trajArrayRy,trajArrayGx,trajArrayGy,trajArrayBx,trajArrayBy] = trackAllImages(

files=dir('*.jpg');
trajArrayRx=zeros(size(files));
trajArrayRy=zeros(size(files));
trajArrayBx=zeros(size(files));
trajArrayBy=zeros(size(files));
trajArrayGx=zeros(size(files));
trajArrayGy=zeros(size(files));

for i=1:size(files)
    img = imread(files(i).name);
    i
    if(i>4)
    figure(1);
    [trajArrayRx(i),trajArrayRy(i),~,trajArrayGx(i),trajArrayGy(i),~,trajArrayBx(i),trajArrayBy(i),~
    end
    hold on

for j=1:size(files)
    if(trajArrayRx(j)>0&&trajArrayRy(j)>0)
    plot(trajArrayRy(j),trajArrayRx(j),'r*');
    if(j>1&&trajArrayRx(j-1)>0&&trajArrayRy(j-1)>0)
    line([trajArrayRy(j-1),trajArrayRy(j)],[trajArrayRx(j-1),trajArrayRx(j)]);
    end
    end
end

for j=1:size(files)
    if(trajArrayGx(j)>0&&trajArrayGy(j)>0)
    plot(trajArrayGy(j),trajArrayGx(j),'g*');
    if(j>1&&trajArrayGx(j-1)>0&&trajArrayGy(j-1)>0)
    line([trajArrayGy(j-1),trajArrayGy(j)],[trajArrayGx(j-1),trajArrayGx(j)]);
    end
    end
end

for j=1:size(files)
    if(trajArrayBx(j)>0&&trajArrayBy(j)>0)
    plot(trajArrayBy(j),trajArrayBx(j),'b*');
    if(j>1&&trajArrayBx(j-1)>0&&trajArrayBy(j-1)>0)
    line([trajArrayBy(j-1),trajArrayBy(j)],[trajArrayBx(j-1),trajArrayBx(j)]);
    end
    end

end
```

```
end
end
```

link to experimental datasets: https://github.com/theakaterina/ivr-practical1

6 Note on Mark Distribution

Ran did the majority of the coding within the first week of the assignment being released. The next two weeks Thea spent cleaning up code, adjusting constants and writing the report. We propose the marks be split 60:40 in Ran's favour.