Ping-Pong Playing Robot Project: Detailed Explanation Of Work Done In The Computer Vision Field

Written by: Arya Araban

Contents

Introduction

Working on a project with a team consisting of around 20 professors and students across different engineering fields, the main goal of the project is to design and launch a real-world robot which can play ping-pong efficiently. Throughout the many fields contained in this project, one of the main fields is the computer engineering field, which consisted of subfields such as AI and computer vision. For the computer vision side of this project, many tasks were required to be implemented and studied on.
These include:

- Tracking the ping-pong ball and getting the real-world 3D coordinates of it
- Obtaining the speed of the ping-pong ball
- Plotting the trajectory of the moving ping-pong ball
- Predicting the ball's movement based on the current trajectory
- Tracking the ping-pong racket, and getting the Euler angles of the racket

The tasks mentioned above, are the tasks which I have successfully implemented throughout the months working on this project.

The entire code of this section of the project has been open-sourced on Github.
In case any questions arise after studying this document, don’t hesitate to contact me at my email address: arya.araban21@gmail.com

Step 1 - Stereo Camera Calibration Procedure

Stereo Camera Calibration is used to estimate the lens and image sensor parameters of image or video cameras (at least two cameras are required). We can use these parameters to correct for lens distortion and measure the size of an object in world units. In this case, the main object is the ping-pong ball.

Setup prerequisites for Stereo Calibration

1- Checkerboard glued to a flat surface – The checkerboard image can be downloaded from here, which then has to be printed to the corresponding paper, and glued to a flat surface.

2- Two cameras running at the same frame rate and resolution – Ideally, both cameras should be the same model. If either of the two has a higher FPS or resolution compared to the other, we can configure it so that the two cameras have the same properties.

The setup which I used to work on consisted of a 35mm checkerboard printed on an A3 paper:

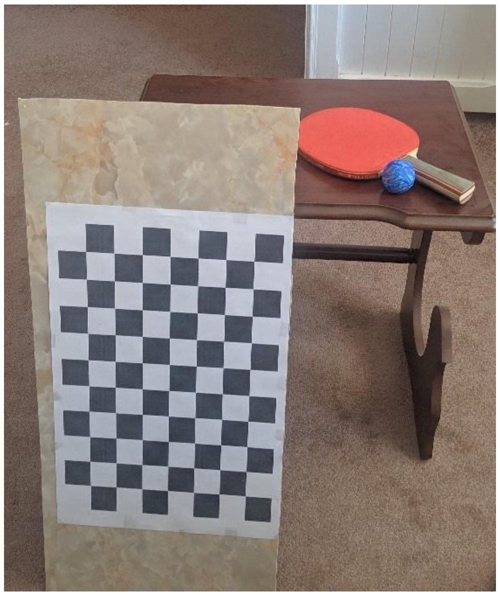


Figure 1 - 35mm A3 checkerboard

As well as two minimal cameras which ran at 30 FPS with an 1280x720 resolution. The first camera being a laptop webcam, and the second one being a mobile device.

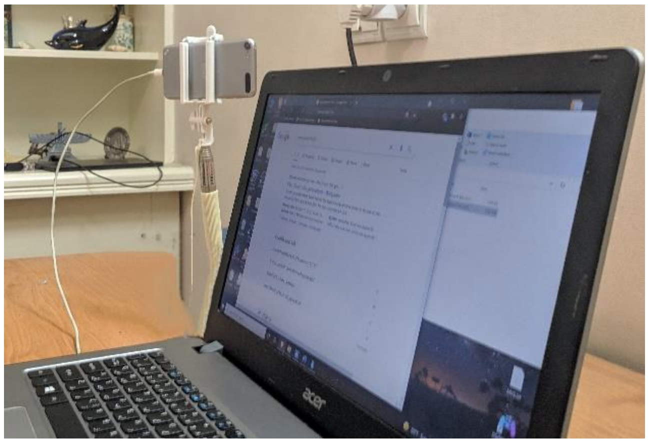


Figure 2 - Stereo Camera Setup

The calibration process

In order to perform calibration, multiple stereo image pairs where the checkerboard is visible in each image pair needs to be captured.
In order to get better results, it's best to hold the checkerboard in different positions for each image pair (e.g., horizontal, vertical, tilted)

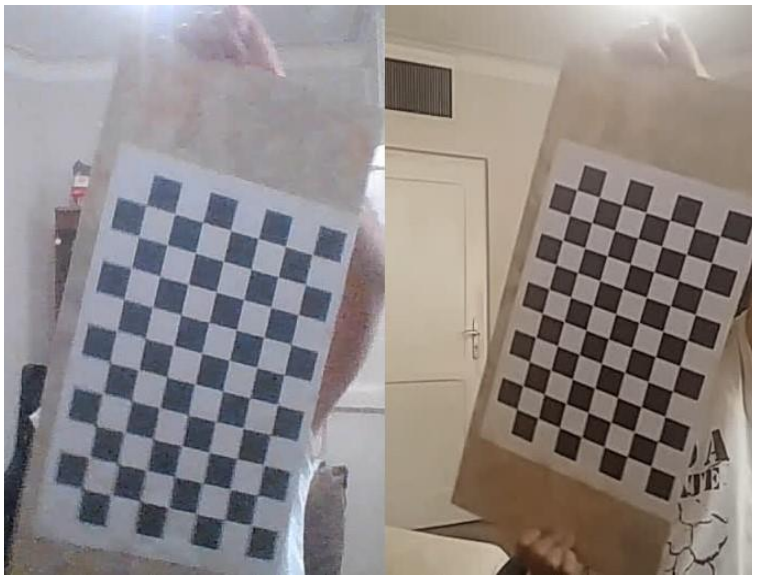
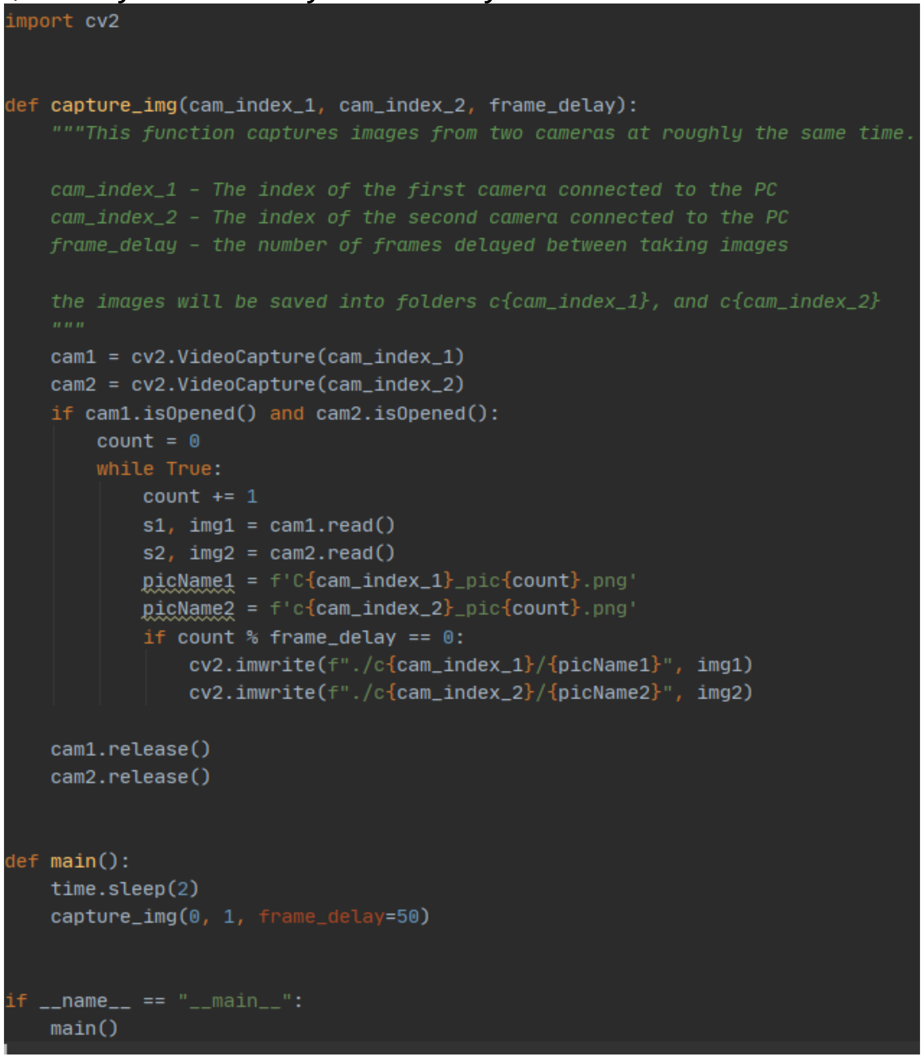


Figure 3 - Example of a stereo image pair

A python script has been coded in order to capture images from two cameras at the same time. In this script, we're able to set intervals as the time between

capturing two images by setting “frame_delay”. For example, if our cameras are set to be 30 FPS, and “frame_delay” is set to 60, there will be a 60/30 = 2 second delay between the capturing of the images. 2 seconds is enough to give a new position to the checkerboard we have at hand.
Note that the “cam_index” inputs are the indexes that OpenCV uses to recognize the camera, and by default they are usually 0 and 1.



```python
import cv2


def capture_img(cam_index_1, cam_index_2, frame_delay):
    """This function captures images from two cameras at roughly the same time.

    cam_index_1 - The index of the first camera connected to the PC
    cam_index_2 - The index of the second camera connected to the PC
    frame_delay - the number of frames delayed between taking images

    the images will be saved into folders c{cam_index_1}, and c{cam_index_2}
    """
    cam1 = cv2.VideoCapture(cam_index_1)
    cam2 = cv2.VideoCapture(cam_index_2)
    if cam1.isOpened() and cam2.isOpened():
        count = 0
        while True:
            count += 1
            s1, img1 = cam1.read()
            s2, img2 = cam2.read()
            picName1 = f'C{cam_index_1}_pic{count}.png'
            picName2 = f'c{cam_index_2}_pic{count}.png'
            if count % frame_delay == 0:
                cv2.imwrite(f"./c{cam_index_1}/{picName1}", img1)
                cv2.imwrite(f"./c{cam_index_2}/{picName2}", img2)

    cam1.release()
    cam2.release()


def main():
    time.sleep(2)
    capture_img(0, 1, frame_delay=50)


if __name__ == "__main__":
    main()
```

Figure 4 - stereo_image_capture.py

After this script has run for a few seconds, we can stop the running of it manually, and the obtained images will be in c0 and c1 folders contained in the folder running the script. A good number of image pairs to have that contain the checkerboard is around 30 to 40

By obtaining these images, we can then use them in order to calibrate our stereo cameras. We will perform the stereo calibration by using the Stereo Camera

Calibrator app in Matlab (which requires the computer vision toolbox of Matlab to be installed). The workflow for stereo calibration in Matlab is as follows:

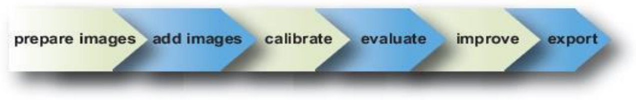
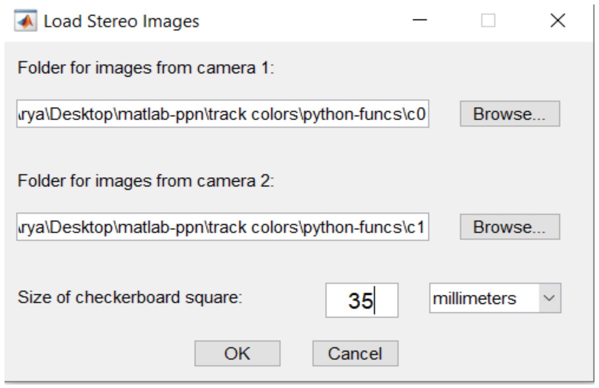


Figure 5 - Camera calibration workflow in Matlab

After loading up the Stereo Calibration Matlab app and hitting the "Add Images" option, we have to specify the folders to load the images, which we specify as the folders which the Python script saved our cameras' images. We also specify the size of each checkerboard square in millimeters, which in our case, was 35mm.



Load Stereo Images

Folder for images from camera 1:

ıya\Desktop\matlab-ppn\track colors\python-funcs\c0

Folder for images from camera 2:

ıya\Desktop\matlab-ppn\track colors\python-funcs\c1

Size of checkerboard square: 35 millimeters

Figure 6 - Loading stereo images for calibration

After adding the images, we press the "Calibrate" option to begin the calibration process. In the official Matlab documentation, it has been stated that using the best 15-20 image pairs gives the best calibration results.

We remove the noisiest images shown in the outlier pane to end up with the best resulting image pairs

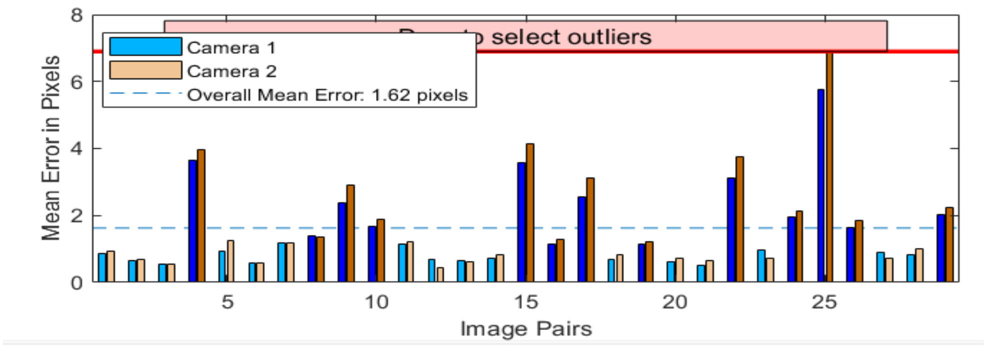


Figure 7 - Selecting the image pairs with higher noise for deletion

It can be seen that before deletion, the Overall Mean Error is 1.62 pixels. After removing the most erroneous image pairs, we have the following:

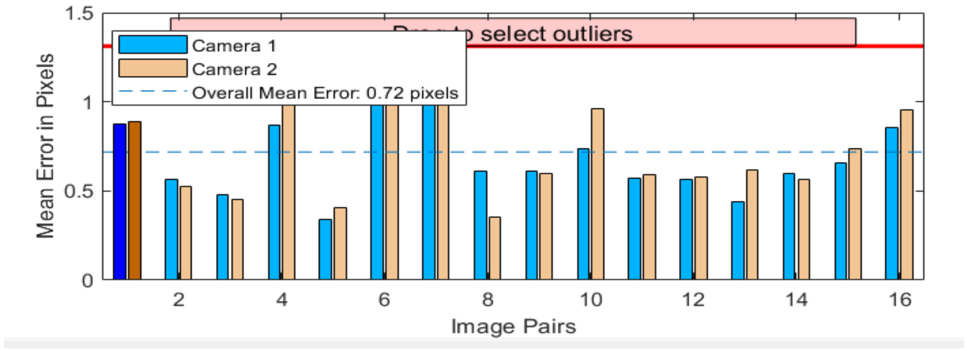


Figure 8 - The 16 best image pairs remaining

As can be seen, we've ended up with a set of images which have a low overall mean error.

After doing so, we select the option to export the stereo calibration parameters in order to use them:

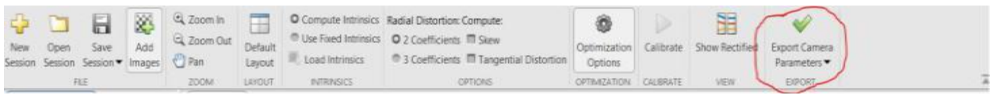


Figure 9 - Export Camera Parameters option

After which, A StereoParameters variable is stored in our workspace. We can also save this parameters variable into a file, so that we can make use of the values for future sessions.

Step 2 - Stereo video capture

Our main goal is to be able to track a moving ping-pong ball's properties (such as 3D coordinates, speed, etc ...) throughout offline videos, and in order to achieve this goal, we will need a tool to be able to capture those videos!

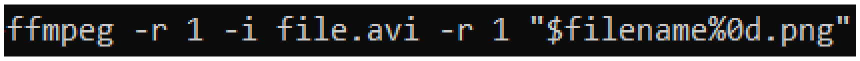
For this purpose, a Python script has been coded which will capture video from the two cameras we set up at the exact same time. This script is called "stereo_video_capture.py". in this script, there are global variables which we set. we have to specify the FPS of the cameras (which in my minimal setup, is 30FPS) and number of frames that we want our videos to run for (By setting the variable NUM_FRAMES).
So for example, if FPS = 30 and NUM_FRAMES = 120, then running the script will make it so that the cameras record video for 4 seconds.
By running the script, we end up with two output files:
"./webcam/outputs/video1.avi" and "./webcam/outputs/video2.avi"

Step 3 - Creating Color Masks

Since most of the object tracking will be handled by color thresholding, we need to come up with an appropriate mask function to be able to separate an object from the environment. For example, if we want to track an orange ping-pong ball throughout a video, for each frame of the video, we need to apply an orange color mask function to each frame, which will make capturing only the ball possible.

Color Mask for tracking the ping-pong ball

In order to make our work easier, we separate a single frame from one of the videos we obtained from the previous section by using FFmpeg (which is a powerful multimedia framework which allows performing various operations on video files) by executing the following command:



```
ffmpeg -r 1 -i file.avi -r 1 "$filename%0d.png"
```

By executing the command above, we extract all the video frames of file.avi (which will be one of the two videos we obtained in the previous section)

And then we can handpick an appropriate frame where the ball is clearly visible.

After this, we can make use of the “Color Thresholder” App of Matlab. With this app, we can give an input image (which in our case will be the handpicked frame), and change the HSV (or LAB) values and see how applying a mask with those values, changes the overall image, and finally we can export that mask as a function, so it will be appliable to every frame from the two videos we have.

For example, if we are looking for an appropriate mask to separate an orange ping-pong ball from the environment, we input one of the frames that clearly contains the orange ball to this app, and try to modify the values, so that finally we are only left with the ball blob:

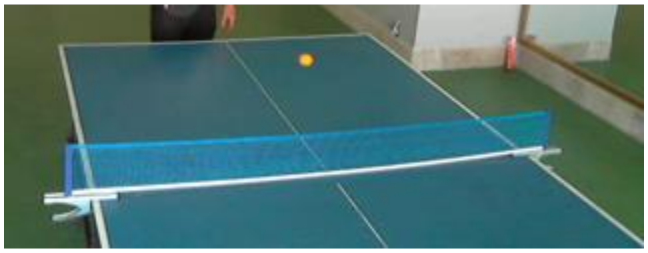


Figure 10 - The selected frame used to create the color mask

By inputting this image to the color Thresholder app and picking the “HSV” option, we tinker with the HSV values by using the side-plane, so that finally, all we are left with is the orange ping-pong ball in the image

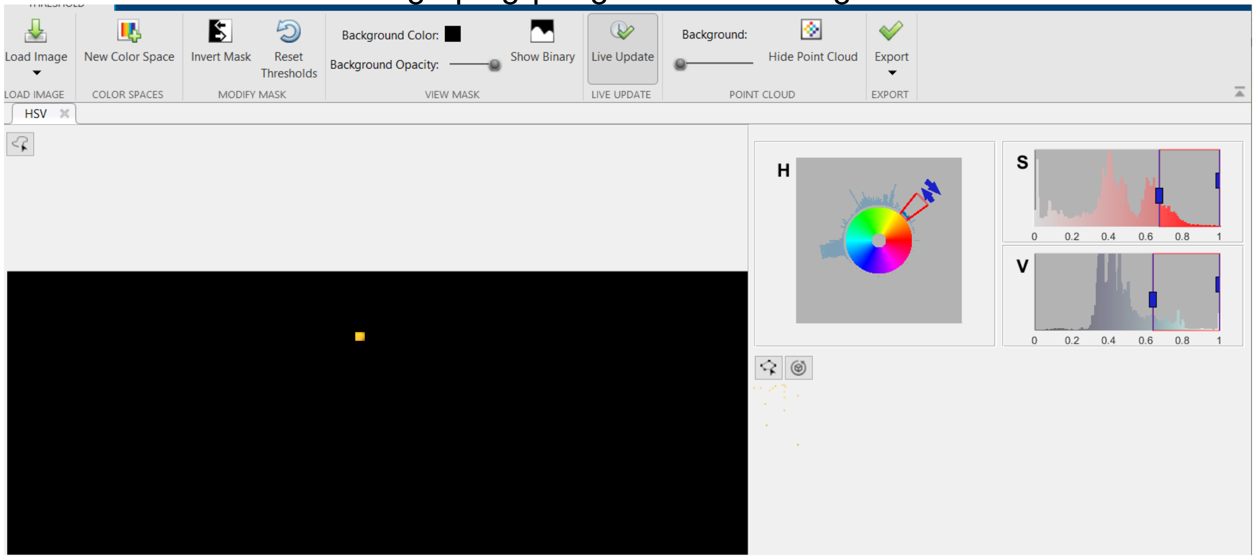
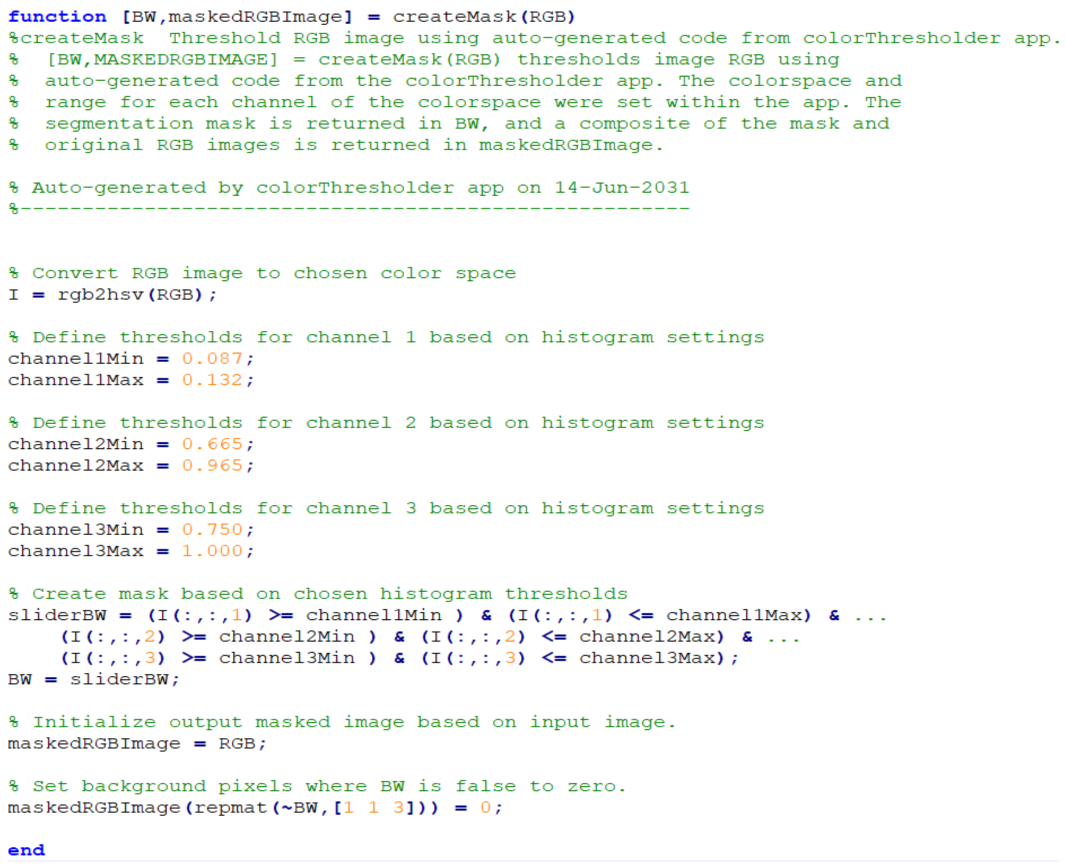


Figure 11 –tinkering with the side-plane options, so that only the ball is visible

After coming up with an appropriate mask, we have to export this mask as a function with the help of the export button on the top.

Note that I gave the masks I tested a filename like “OrangeBallMask.m”, and put it in the “private” folder, so that Matlab can recognize the functions, while keeping these files in an organized manner.

The obtained function files’ contents will be similar to the following:



```
function [BW,maskedRGBImage] = createMask(RGB)
%createMask  Threshold RGB image using auto-generated code from colorThresholder app.
%  [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using
%  auto-generated code from the colorThresholder app. The colorspace and
%  range for each channel of the colorspace were set within the app. The
%  segmentation mask is returned in BW, and a composite of the mask and
%  original RGB images is returned in maskedRGBImage.

% Auto-generated by colorThresholder app on 14-Jun-2031
%------------------------------------------------------


% Convert RGB image to chosen color space
I = rgb2hsv(RGB);

% Define thresholds for channel 1 based on histogram settings
channel1Min = 0.087;
channel1Max = 0.132;

% Define thresholds for channel 2 based on histogram settings
channel2Min = 0.665;
channel2Max = 0.965;

% Define thresholds for channel 3 based on histogram settings
channel3Min = 0.750;
channel3Max = 1.000;

% Create mask based on chosen histogram thresholds
sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) & ...
    (I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...
    (I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);
BW = sliderBW;

% Initialize output masked image based on input image.
maskedRGBImage = RGB;

% Set background pixels where BW is false to zero.
maskedRGBImage(repmat(~BW,[1 1 3])) = 0;

end
```

Figure 12 - an example of a mask function file generated by the Color Thresholder app

Color Mask for tracking the ping-pong racket

throughout the main code, beside the color mask which is used for tracking the ping-pong ball, we also need a different color mask which tracks the ping-pong racket. In order to track the racket, colored markers will be sticked on the racket, which the reasons will be explained in the next section.
The marked racket looks as follows:

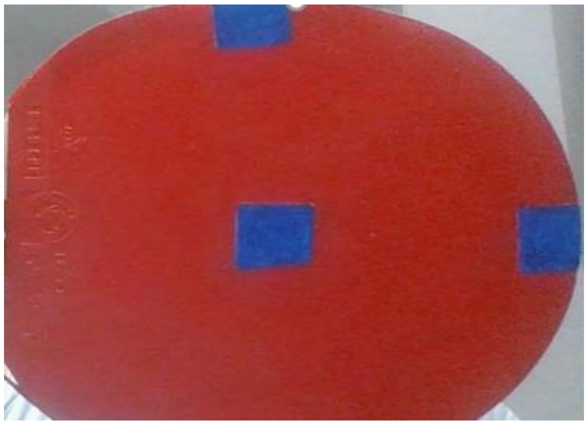


Figure 13 - Color marked racket

Based on the color of the markers (I used blue markers), we can create the appropriate color masks with the procedure explained above.

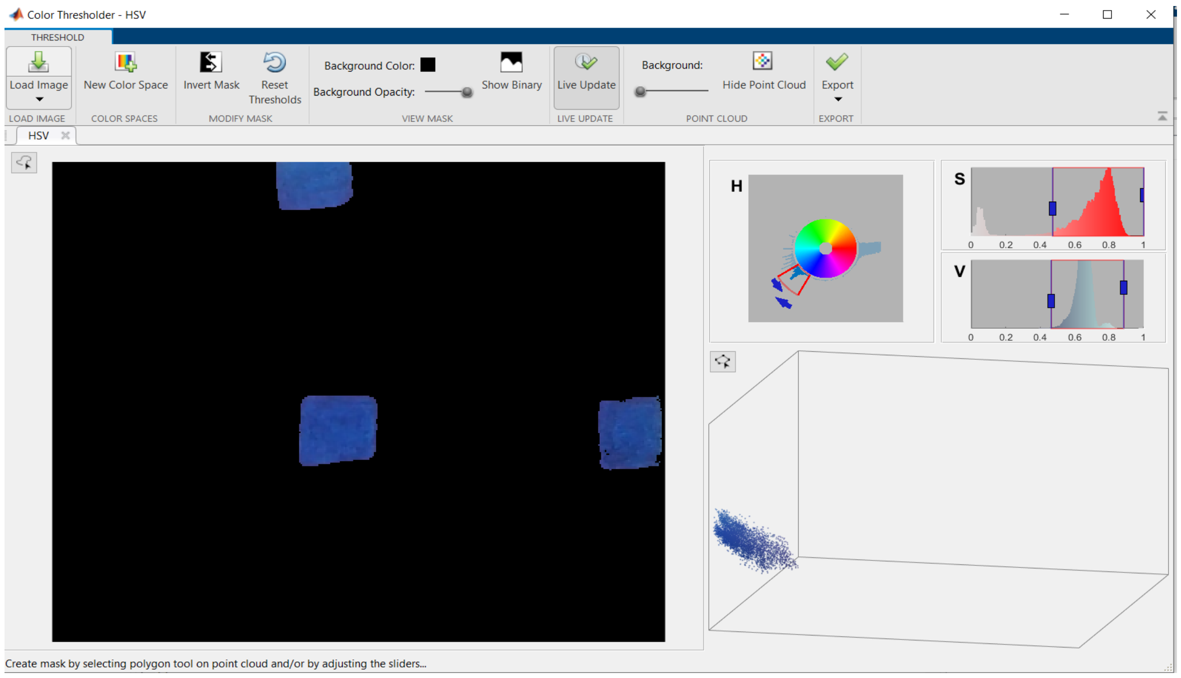


Figure 14 - tinkering with the side-plane options, so that only the racket markers are visible

Function file names such as “BlueMarkerMask.m” have been used for the resulting functions from applying color thresholding to the racket, and have yet again been placed in the “private” folder.

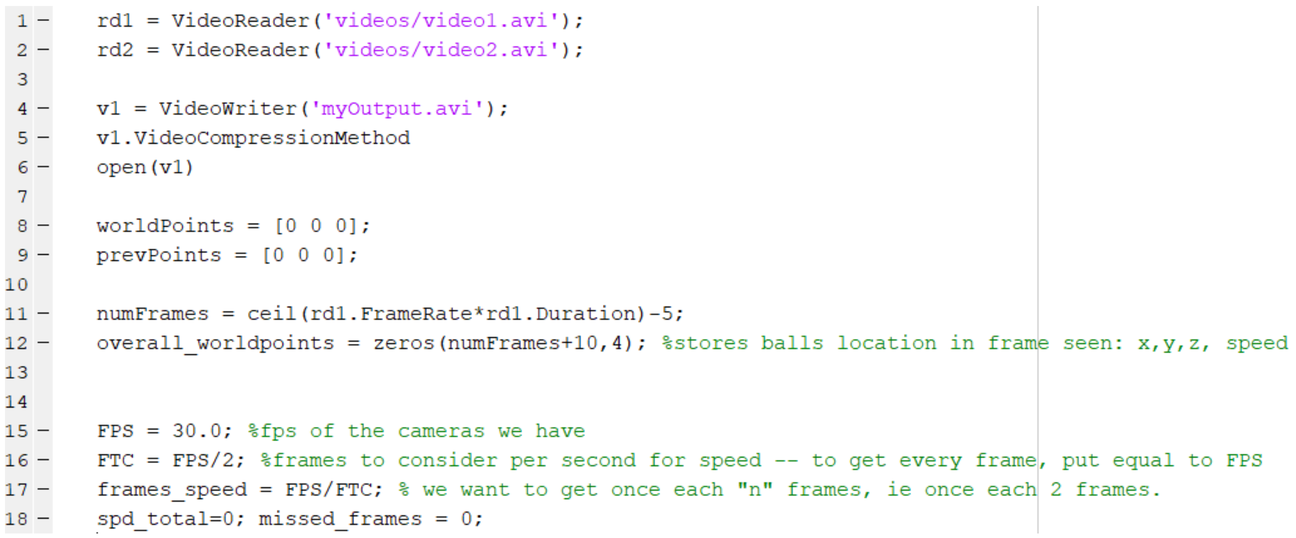
Note that we can either input an image of the marked racket taken directly from one of the cameras, or we can yet again extract frames from the videos and select an appropriate frame where the racket and its markers are clearly visible.

Step 4 – Analyzing Matlab code (usage and details)

In the previous sections, we created a few different items.
Firstly, we calibrated our stereo setup to create the Stereo Parameters,
Then, we created a stereo video pair showcasing an actual ping-pong game (where the colored ping-pong ball, and optionally the marked racket is visible),
And finally, we created color mask functions based on the color of the objects.
In this section, we will be studying the Matlab code which makes use of these items in order to achieve the main goals that were specified in the introduction section.

Main Code

In the first few lines of this code, we are just initializing the variables and parameters.



```
rd1 = VideoReader('videos/video1.avi');
rd2 = VideoReader('videos/video2.avi');

v1 = VideoWriter('myOutput.avi');
v1.VideoCompressionMethod
open(v1)

worldPoints = [0 0 0];
prevPoints = [0 0 0];

numFrames = ceil(rd1.FrameRate*rd1.Duration)-5;
overall_worldpoints = zeros(numFrames+10,4); %stores balls location in frame seen: x,y,z, speed


FPS = 30.0; %fps of the cameras we have
FTC = FPS/2; %frames to consider per second for speed -- to get every frame, put equal to FPS
frames_speed = FPS/FTC; % we want to get once each "n" frames, ie once each 2 frames.
spd_total=0; missed_frames = 0;
```

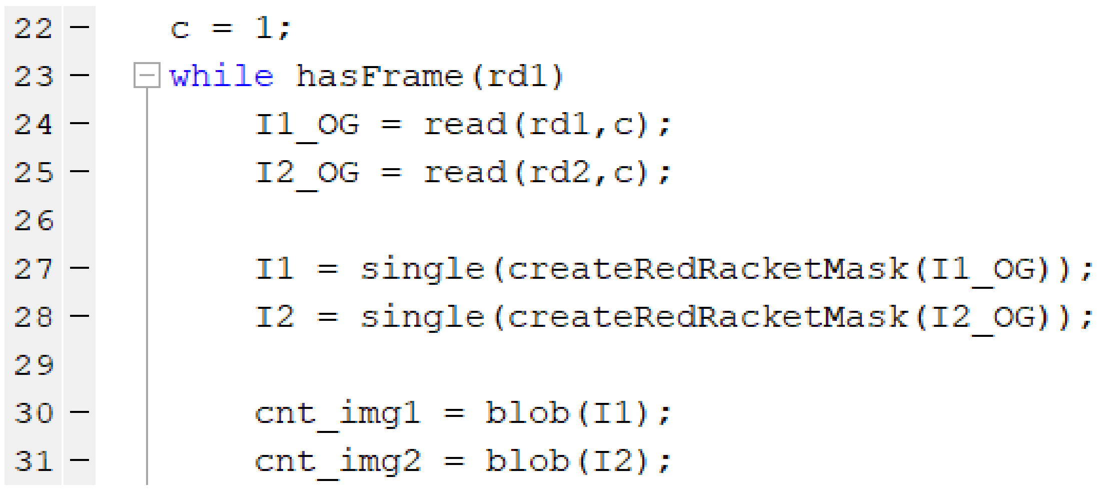
- In the first two lines, we are inputting the videos which we obtained in step 2 into “rd1” and “rd2” so that we are able to read their frames

- “v1” Is the main video output of executing the code (which will show the properties of tracking the ping-pong ball such as speed, and will also optionally show the properties of the racket). This file will be called myOutput.avi

- We use “worldPoints” (which are the current frame’s world points) and “prevPoints” (which are the previous frame’s world points) in order to find the speed

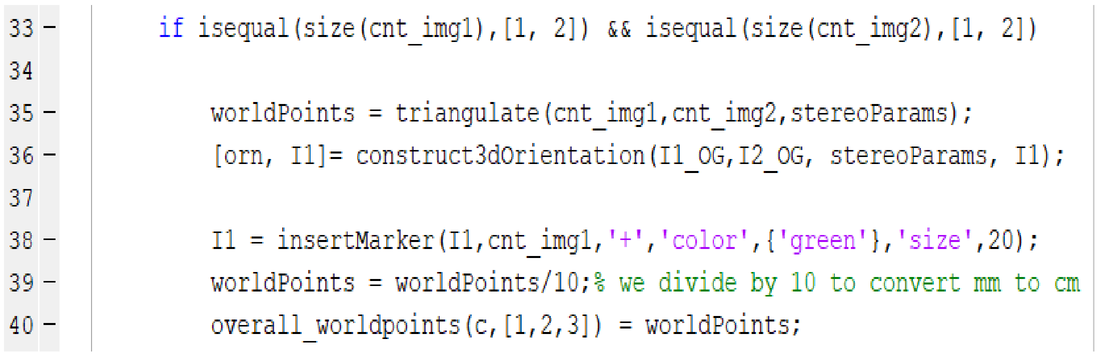
- “overall_worldpoints” is used to draw the trajectory of the moving ball. It stores the 3D world points and the speed of the ball in such a way that the i’th row will contain these properties for the i’th frame

- The variables seen throughout lines 15 to 18 are used to handle getting speed based on a lesser number of frames of our camera’s FPS, as well as dealing with frames which couldn’t capture the ball’s location.



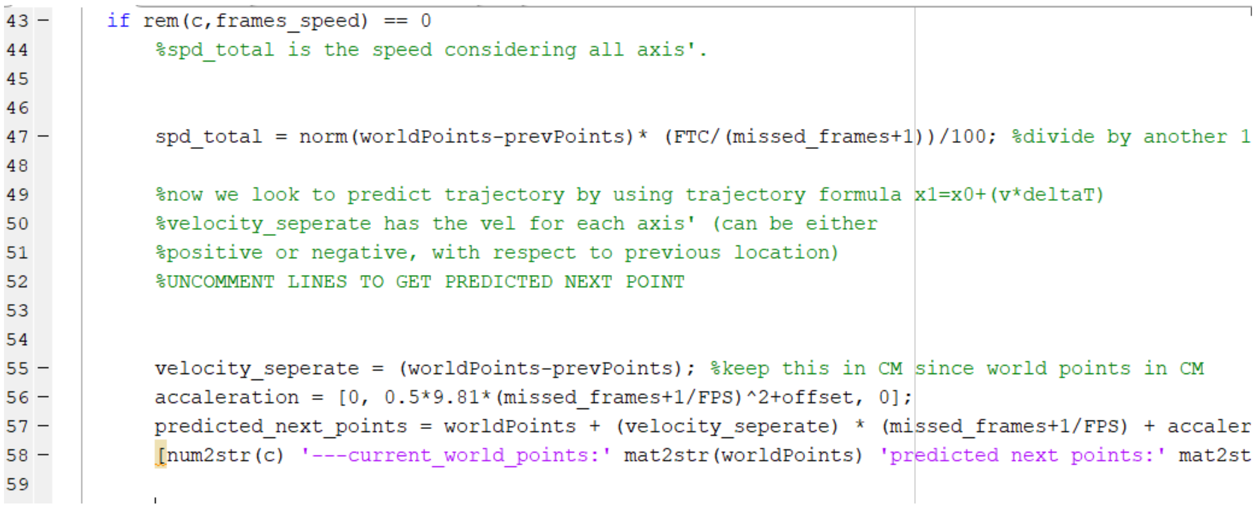
```
22 -     c = 1;
23 -   while hasFrame(rd1)
24 -         I1_OG = read(rd1,c);
25 -         I2_OG = read(rd2,c);
26
27 -         I1 = single(createRedRacketMask(I1_OG));
28 -         I2 = single(createRedRacketMask(I2_OG));
29
30 -         cnt_img1 = blob(I1);
31 -         cnt_img2 = blob(I2);
```

In this part of the code, we have started the main while loop which will iterate over the frames of the videos. We also represent the current frame with the “c” variable. As can be seen, in lines 27 and 28, we are using the binary mask function we created with the Thresholder app to output binary images I1 and I2, and what the blob function does is simply find the largest matching blob in the image (which is our ball), and returns the centroid of it as an [X, Y] point. We find the blob’s 2D centers for each of our two frames, and we will be using them to find the 3D coordinates…



```
33 -     if isequal(size(cnt_img1),[1, 2]) && isequal(size(cnt_img2),[1, 2])
34
35 -         worldPoints = triangulate(cnt_img1,cnt_img2,stereoParams);
36 -         [orn, I1]= construct3dOrientation(I1_OG,I2_OG, stereoParams, I1);
37
38 -         I1 = insertMarker(I1,cnt_img1,'+','color',{'green'},'size',20);
39 -         worldPoints = worldPoints/10;% we divide by 10 to convert mm to cm
40 -         overall_worldpoints(c,[1,2,3]) = worldPoints;
```

In the next part, the condition is checking whether we see a centroid in each of the two images. if so, we continue on with the code.
The triangulate function finds the 3D world points based on the two centroids we input to it, and the stereo parameters we found with the Stereo Calibrator app.
We then use the construct3dOrientation function which returns the three Euler Angles, and stores them in “orn” by using the three markers that are on the racket. An image I1 is also returned, which marks the racket markers with a + sign
insertMarker on line 38 is simply inserting a green marker on the centroid in the first camera’s image.

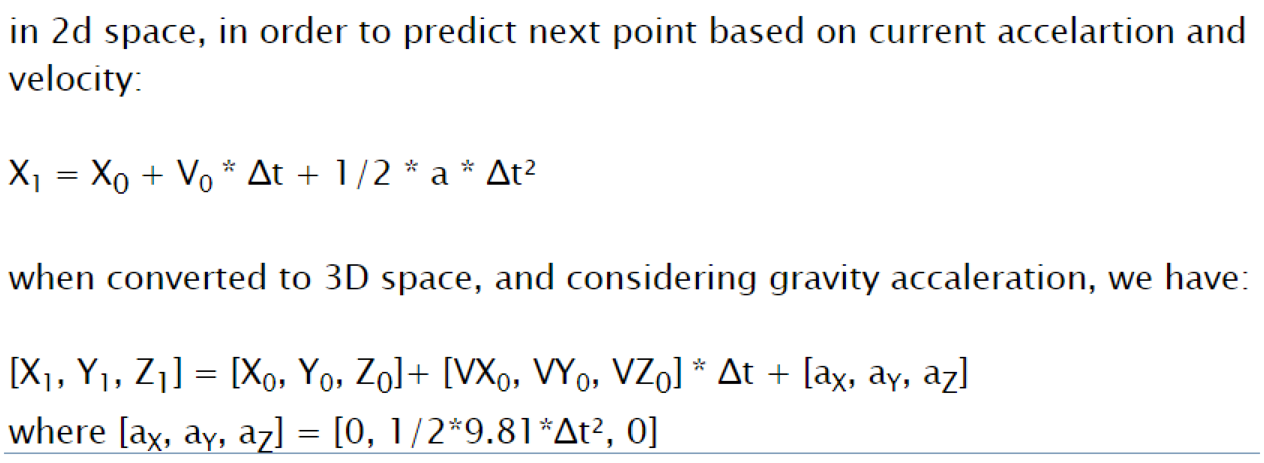


```
43 -   if rem(c,frames_speed) == 0
44         %spd_total is the speed considering all axis'.
45
46
47 -       spd_total = norm(worldPoints-prevPoints)* (FTC/(missed_frames+1))/100; %divide by another 1
48
49         %now we look to predict trajectory by using trajectory formula x1=x0+(v*deltaT)
50         %velocity_seperate has the vel for each axis' (can be either
51         %positive or negative, with respect to previous location)
52         %UNCOMMENT LINES TO GET PREDICTED NEXT POINT
53
54
55 -       velocity_seperate = (worldPoints-prevPoints); %keep this in CM since world points in CM
56 -       accaleration = [0, 0.5*9.81*(missed_frames+1/FPS)^2+offset, 0];
57 -       predicted_next_points = worldPoints + (velocity_seperate) * (missed_frames+1/FPS) + accaler
58 -       [num2str(c) '---current_world_points:' mat2str(worldPoints) 'predicted next points:' mat2st
59
```

In the next few lines of the code, we focus on the speed of the ball which we are tracking.
In line 43 we are checking whether the current frame should be considered based on the FTC we set (example: FPS is 30 and FTC is 15, then “frames_speed” will be 2, meaning each two frames we will get the speed by comparing the ball’s location in frame i+2 to frame i)

The way that speed is found is that we first find the distance between the current frame and the previous observed frame (which is based on “frames_speed” and “missed_frames”), and then we multiply this number with the FTC.

Line 56 and 57 are printing the next predicted points based on the last world points by using the following manner:



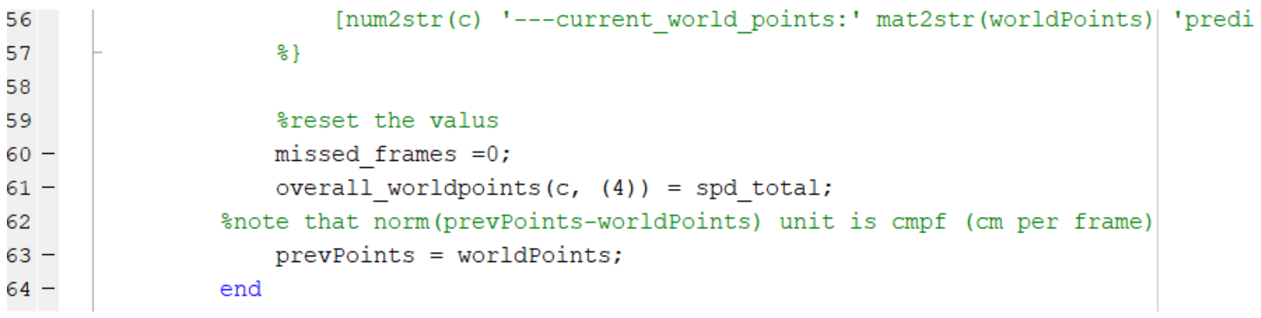
in 2d space, in order to predict next point based on current accelartion and velocity:

$$X_1 = X_0 + V_0 * \Delta t + 1/2 * a * \Delta t^2$$

when converted to 3D space, and considering gravity accaleration, we have:

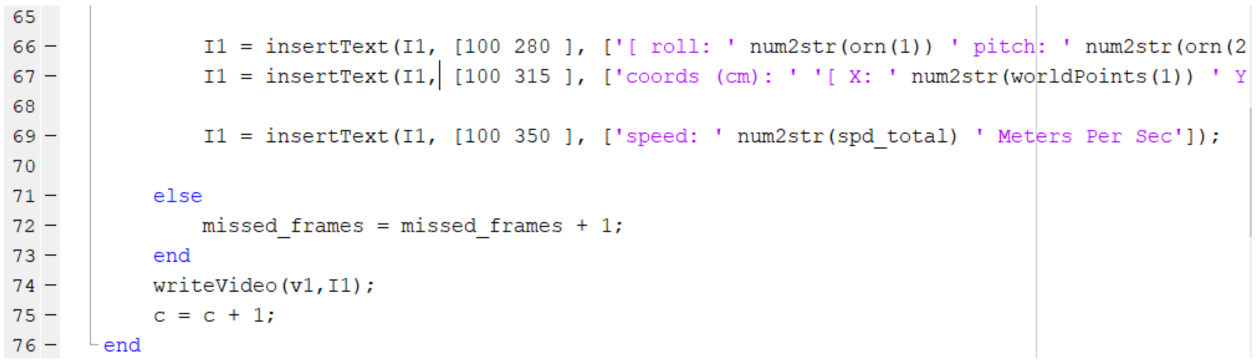
$$[X_1, Y_1, Z_1] = [X_0, Y_0, Z_0] + [VX_0, VY_0, VZ_0] * \Delta t + [a_X, a_Y, a_Z]$$

where $[a_X, a_Y, a_Z] = [0, 1/2*9.81*\Delta t^2, 0]$



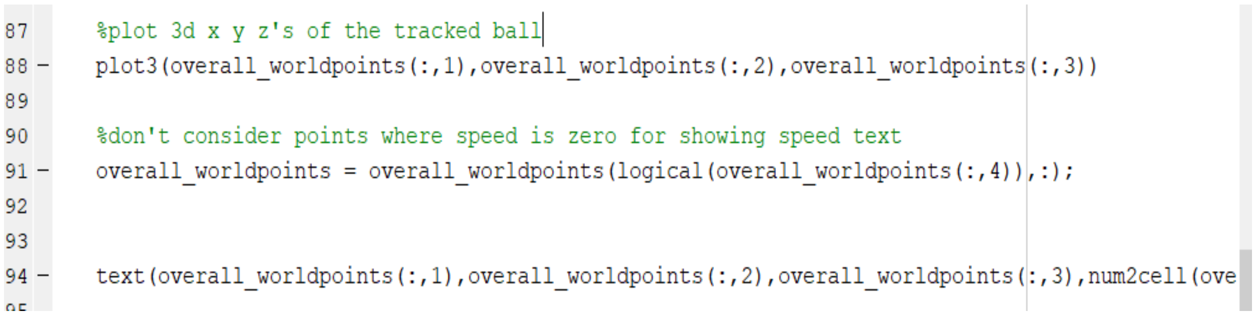
```
56                  [num2str(c) '---current_world_points:' mat2str(worldPoints) 'predi
57 -            %}
58
59              %reset the valus
60 -            missed_frames =0;
61 -            overall_worldpoints(c, (4)) = spd_total;
62          %note that norm(prevPoints-worldPoints) unit is cmpf (cm per frame)
63 -            prevPoints = worldPoints;
64 -        end
```

in the final lines inside the condition, we reset the “missed_frames” count, and add the current frame’s speed to the corresponding row in the final column of “overall_worldpoints”



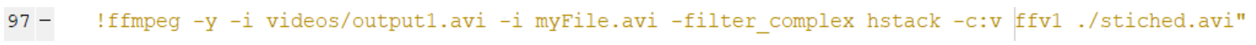
```
65
66 -           I1 = insertText(I1, [100 280 ], ['[ roll: ' num2str(orn(1)) ' pitch: ' num2str(orn(2
67 -           I1 = insertText(I1, [100 315 ], ['coords (cm): ' '[ X: ' num2str(worldPoints(1)) ' Y
68
69 -           I1 = insertText(I1, [100 350 ], ['speed: ' num2str(spd_total) ' Meters Per Sec']);
70
71 -       else
72 -           missed_frames = missed_frames + 1;
73 -       end
74 -       writeVideo(v1,I1);
75 -       c = c + 1;
76 -   end
```

In the next couple of lines, we are inserting all the info we gathered for the current frame on to the first binary masked image (which we will be displaying). If no ball was identified, the “missed_frames” count is incremented, which will be used to calculate the speed for the next frame, as seen on line 46.
Line 74 is writing the current frame which we have to the output video
At line 75 we are just incrementing the count for our frame variable (Reminder: This while loop ends when all the frames have been iterated)



```
%plot 3d x y z's of the tracked ball
plot3(overall_worldpoints(:,1),overall_worldpoints(:,2),overall_worldpoints(:,3))

%don't consider points where speed is zero for showing speed text
overall_worldpoints = overall_worldpoints(logical(overall_worldpoints(:,4)),:);


text(overall_worldpoints(:,1),overall_worldpoints(:,2),overall_worldpoints(:,3),num2cell(ove
```

These few lines of code are used to plot the trajectory of the moving ball.



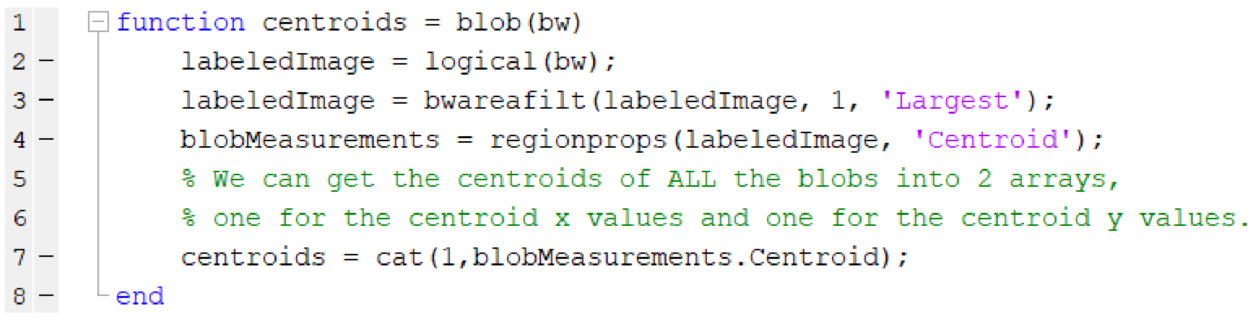
```
!ffmpeg -y -i videos/output1.avi -i myFile.avi -filter_complex hstack -c:v ffv1 ./stiched.avi"
```

By using FFmpeg, this final line stiches together the first camera’s video alongside the video we created by using the above while loop, which contains the ping-pong objects information.

Helper Functions

Throughout the main code a few functions have been used, which have been coded separately. These functions have pretty simple purposes:

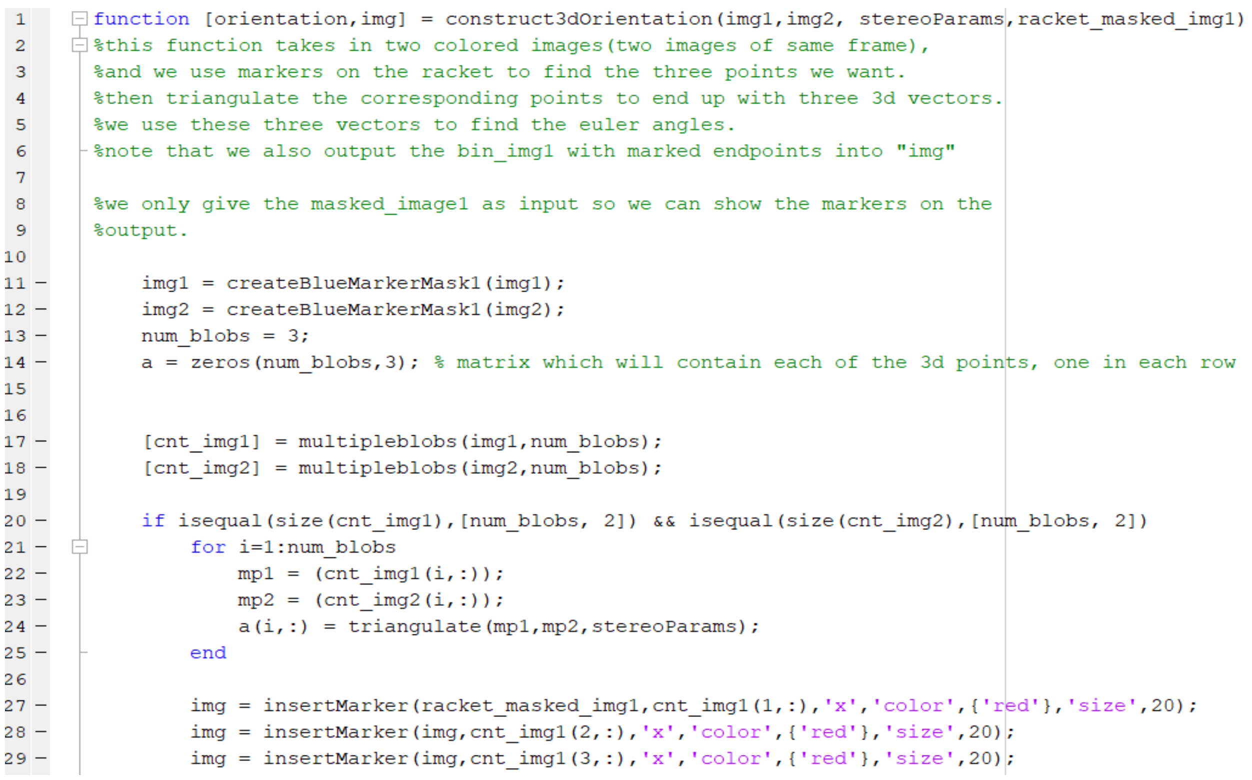
blob.m



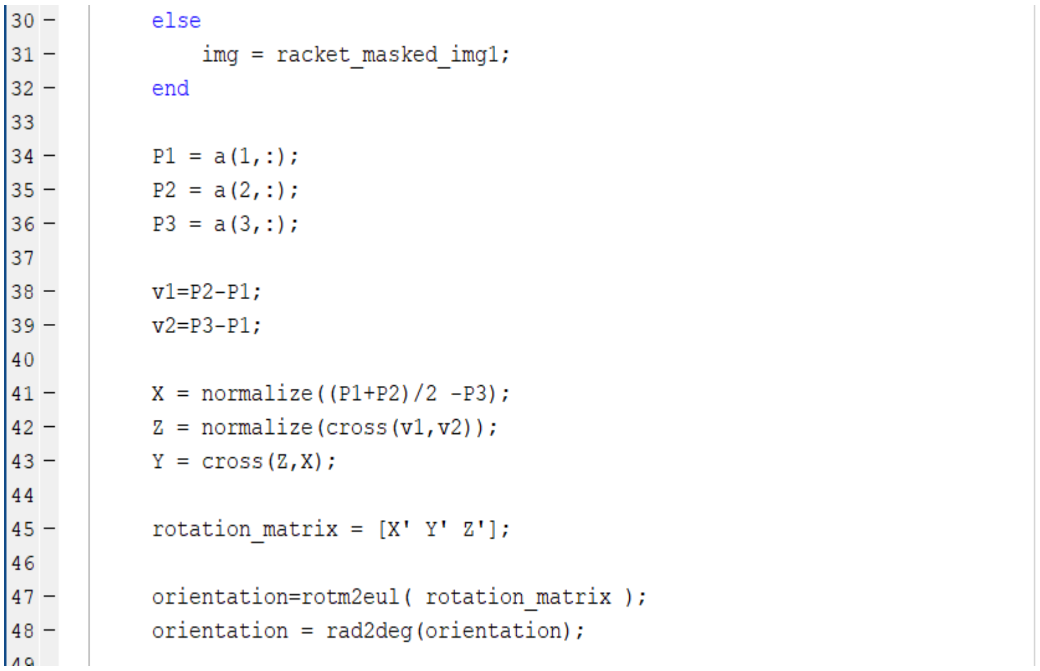
```
function centroids = blob(bw)
    labeledImage = logical(bw);
    labeledImage = bwareafilt(labeledImage, 1, 'Largest');
    blobMeasurements = regionprops(labeledImage, 'Centroid');
    % We can get the centroids of ALL the blobs into 2 arrays,
    % one for the centroid x values and one for the centroid y values.
    centroids = cat(1,blobMeasurements.Centroid);
end
```

This function's input is a binary image, which in our case, is the image after applying the colored mask. The output of this function is the centroid of the largest blob found throughout the image (which is hopefully, the ball which we want to track).

Construct3dOrientation.m



```
function [orientation,img] = construct3dOrientation(img1,img2, stereoParams,racket_masked_img1)
%this function takes in two colored images(two images of same frame),
%and we use markers on the racket to find the three points we want.
%then triangulate the corresponding points to end up with three 3d vectors.
%we use these three vectors to find the euler angles.
%note that we also output the bin_img1 with marked endpoints into "img"

%we only give the masked_image1 as input so we can show the markers on the
%output.

    img1 = createBlueMarkerMask1(img1);
    img2 = createBlueMarkerMask1(img2);
    num_blobs = 3;
    a = zeros(num_blobs,3); % matrix which will contain each of the 3d points, one in each row


    [cnt_img1] = multipleblobs(img1,num_blobs);
    [cnt_img2] = multipleblobs(img2,num_blobs);

    if isequal(size(cnt_img1),[num_blobs, 2]) && isequal(size(cnt_img2),[num_blobs, 2])
        for i=1:num_blobs
            mp1 = (cnt_img1(i,:));
            mp2 = (cnt_img2(i,:));
            a(i,:) = triangulate(mp1,mp2,stereoParams);
        end

        img = insertMarker(racket_masked_img1,cnt_img1(1,:),'x','color',{'red'},'size',20);
        img = insertMarker(img,cnt_img1(2,:),'x','color',{'red'},'size',20);
        img = insertMarker(img,cnt_img1(3,:),'x','color',{'red'},'size',20);
```



```
30 -        else
31 -            img = racket_masked_img1;
32 -        end
33
34 -        P1 = a(1,:);
35 -        P2 = a(2,:);
36 -        P3 = a(3,:);
37
38 -        v1=P2-P1;
39 -        v2=P3-P1;
40
41 -        X = normalize((P1+P2)/2 -P3);
42 -        Z = normalize(cross(v1,v2));
43 -        Y = cross(Z,X);
44
45 -        rotation_matrix = [X' Y' Z'];
46
47 -        orientation=rotm2eul( rotation_matrix );
48 -        orientation = rad2deg(orientation);
```

This function takes in the two original(non-masked) images, and will find the Euler angles of the marked racket, if it exists in the images.

If no marked racket is detected, the Euler angles will simply have “NaN” values.

In this function, we first apply the racket mask function which we obtained in step 2, We then find the centroids of each one of the three markers separately with the help of “multipleBlobs.m” which is very similar to the “blob.m” function, the only difference being that it allows tracking multiple blobs. After which, we find the 3D coordinates of each of the points by using the triangulate function.

We then find the rotation matrix of these three points with the help of the calculations throughout line 34 to 43. Note that P1, P2, P3 are the centroid points of each of the markers.

Finally, this rotation matrix is converted to Euler angles with the help of the built-in “rotm2eul” function.

Conclusion

By using a simple stereo setup and color thresholding for object tracking, we were able to achieve many of the tasks required for the computer vision side of the ping-pong playing robot project. By coding the appropriate functionalities in Matlab, we were able to get satisfying results for the tasks at hand. However, these tasks were performed on off-line videos, whereas in the real world, we want to be able to track the ping-pong objects in an on-line manner. For this use-case, built-in Matlab functions might not be appropriate since some of these functions yield overhead. Future work on this project may include implementing the achieved vision tasks on-line. For this, it may be wise to look into OpenCV, which is a strong framework mainly aimed at real-time computer vision. It can either be attempted to integrate OpenCV in Matlab, or just use a programming language which supports OpenCV natively, such as Python or C++. This would require the work done here to be implemented in the selected programming language.

References

https://au.mathworks.com/help/vision/ug/using-the-stereo-camera-calibrator-app.html

https://au.mathworks.com/help/vision/opencv-interface-support-package.html

https://au.mathworks.com/help/images/ref/colorthresholder-app.html

https://www.gamedeveloper.com/programming/movement-prediction