Ping-Pong Project: Detailed Explanation of Code and applications used

Introduction

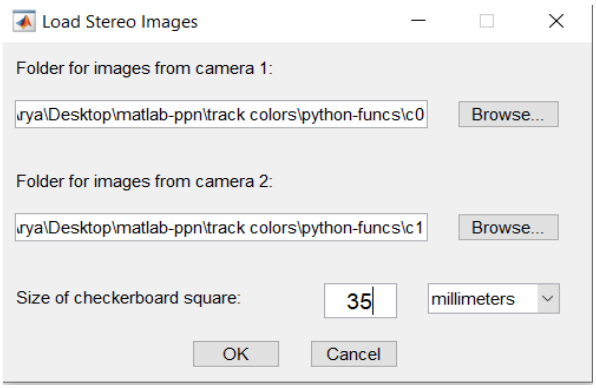
In this document, I'll be going into detail about how the Matlab code I've written should be used, and when to use Matlab's "Apps", as well as the Python scripts that I've written.

Part 1 – Matlab Apps and Python Scripts

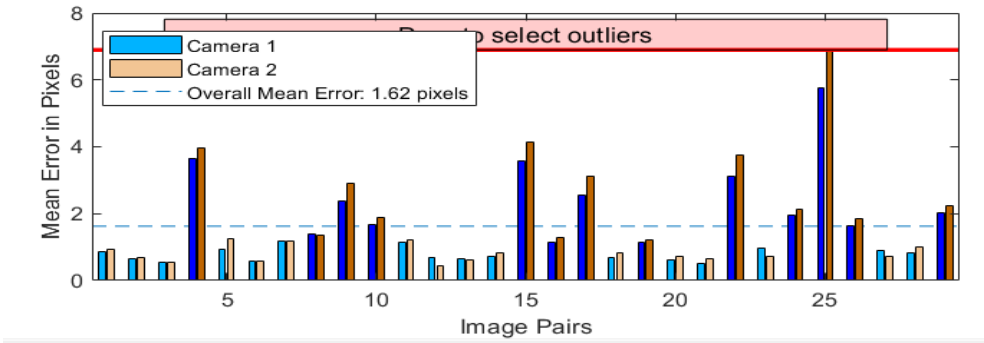
In the first document that I've written for this project, I showed the overall procedure for how to calibrate the stereo cameras. That procedure has been slightly changed throughout the time working on the project.
Firstly, on the official Matlab guide for camera calibration, it has been stated that the optimal number of stereo images to use for calibration is between 10 and 20. I found that having around 18 images gives a rather good "StereoParameters" result.
In order to get the checkerboard images, I use the Python script "stereo_image_capture.py". This script consists of one main function "capture_img"

```python
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()
```

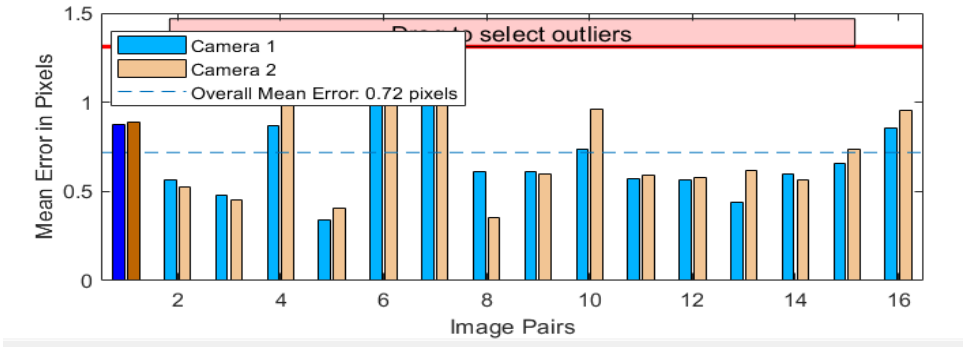
Taking around 30 to 40 images of us holding the checkerboard using this script, is a good amount to give the stereo calibrator app (which we will then remove the high noise pictures, in order to end up with 10-20 pictures).
After loading up the Stereo Calibration Matlab app and hitting the “Add Images” option, we have to specify the folders to load the images, which we specify as the path which the Python script saved it:



After hitting the “calibrate” option, we remove the most noisy images, to finally end up with the 15-20 less noisy images:

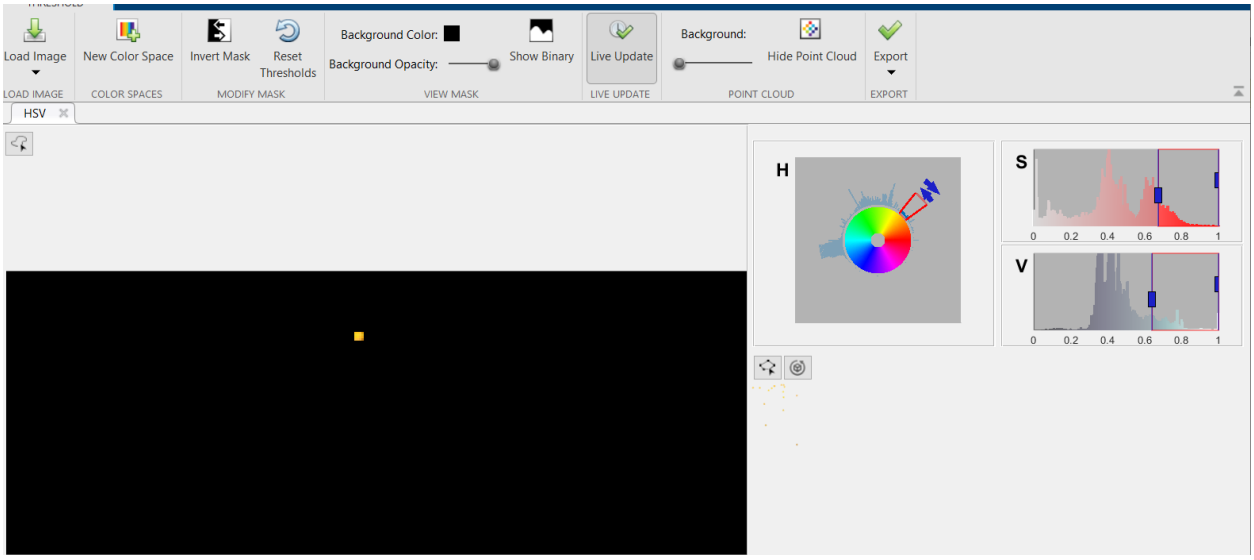


We remove the selected images which have higher mean errors, and get something like the following:



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

The other app that can be used in order to make our work easier, is the "Color Thresholder" App. With this app, we can give an input image, and change the HSV (or LAB) values and see how applying a mask with those values, changes the overall image. For example, if we are looking for an appropriate mask to follow an orange ping-pong ball, we input an image containing the orange ball to this app, and try to modify the values, so that finally we are only left with the ball blob:



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.

throughout the main code, we need two different color masks, the first color mask is used for tracking the ping-pong ball, and the other color mask is for tracking the markers placed on the ping-pong racket. Based on the color of the ball and

markers, we can create the appropriate color masks with the procedure explained above.

Finally, I use another Python script in order to capture stereo video at roughly the same time. This script is called “stereo_video_capture.py”. note that we have to specify the FPS, and number of frames that we want our videos to run for (By setting the parameter NUM_FRAMES), and by running the script, we end up with two files: “./webcam/outputs/output1” and “./webcam/outputs/output1”

We make use of these files, as well as the color masks, in the main Matlab code.

Part 2 – Main Matlab code

We will study the “main.m” file, and then study other files when needed.

The first few lines of my code are pretty self-explanatory, we have a few variables which we are giving initial values to.

```
2 -    v1 = VideoWriter('myFile.avi');
3 -    rd1 = VideoReader('videos/output1.avi');
4 -    rd2 = VideoReader('videos/output2.avi');
5 -    v1.VideoCompressionMethod
6 -    numFrames = ceil(rd1.FrameRate*rd1.Duration)-5;
7 -    open(v1)
8
9 -    worldPoints = [0 0 0];
10 -   prevPoints = [0 0 0];
11 -   overall_worldpoints = zeros(numFrames+10,4); %stores balls location in frame seen: x,y,z, sp
12
13
14 -   FPS = 30.0;
15 -   FTC = FPS/2; %frames to consider per second for speed -- to get every frame, put equal to FP
16     %time_frame = 0; offset = 0;%this is for gravity accaleration
17 -   frames_speed = FPS/FTC; % we want to get each "n" frames, ie each 2 frames at a time.
18 -   spd_total=0; missed_frames = 0;
```

Note that “rd1” and “rd2” are the video files we obtained by using the Python script, “numFrames” is the total number of frames we have in a video,
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, and “overall_worldpoints” is only used to draw the trajectory of the moving ball. Finally, 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.

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

In this part of the code we have started a while loop which will iterate over the frames of the videos. As can be seen, in lines 27 and 28, we are using the binary mask function we created with the Thresholder app, 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. We find the blob’s 2d centers for each of our two frames, and we will be using them to find the 3D coordinations…

```
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 “if” is checking weather we see one 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 the three Euler Angles in “orn”, and an image I1 which marks the racket markers with a + sign.

The insertMarker on line 38 is simply inserting a green marker on the centroid in the first camera's image.

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

In the next few lines of the code, we focus on the speed of the ball which we are tracking.
In line 42 we are checking weather 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)
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.

The next few commented lines are for predicting where the ball will appear in the next frame, which I explained in the 6th document.

```
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 of the if 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). Line 72 is the else part of the condition weather the ball has been seen on the current frame or not, if not increment the “missed_frames” count, which will be used to calculate the speed, as seen on line 46.
At line 75 we are just incrementing the count for our frame variable, and on line 76 we exit the while loop.

```
87      %plot 3d x y z's of the tracked ball
88 -    plot3(overall_worldpoints(:,1),overall_worldpoints(:,2),overall_worldpoints(:,3))
89
90      %don't consider points where speed is zero for showing speed text
91 -    overall_worldpoints = overall_worldpoints(logical(overall_worldpoints(:,4)),:);
92
93
94 -    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.

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

This final line stiches together the first camera’s video, alongside the video we obtained by using the above while loop, which contains the info of the frames. Note that it is required to have installed ffmpeg, which is a powerful CLI tool to handle image and video operations.

Part 3 – Functions used in the main code

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

blob.m

```
function [centroids, angle] = 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);
    angle = regionprops(labeledImage, 'Orientation');
end
```

This function's input is a binary image, which in our case, are the color masked images used for the ball, and outputs the centroid of the largest blob found throughout the image (which is hopefully, the ball 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.

Note that firstly we apply a mask function which should be found based on the color of the rackets' markers.

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. We then simply find the 3D coordinations of each of the points by using the triangulate function.

Finally, we find the rotation matrix of these three points with some calculations, and then we convert the rotation matrix to Euler angles.