Computer Vision Course: Mini Project #1

Inside Looking-out Tracking System

Seyed Mohammad Taha Tabatabaei

Deadline: Nov 17, 2023

Report date: Nov 16, 2023

Professor: Dr. Pourreza

Abstract

In this Report, we try to estimate a camera pose in 3D space using photos provided by the camera itself. This procedure is called Inside Looking-Out (ILO). In the Method section, we provide a comparison between the Inside Looking-Out and Outside Looking-In approach. Then we explain the work. In the first phase, we must calibrate the camera. Secondly, we use a photo with valid markers which we know their world coordinates to locate the camera in the world coordinate system. In the Discussion section, we discuss the results. Images from results and visualizations can be found in the Results section. All scripts and plots generated using MATLAB.

Methods

1. A comparison between Inside Looking-Out (ILO) and Outside Looking-In systems

Inside Looking-Out (ILO) and Outside Looking-In (OLI) camera-based tracking systems are two approaches used for tracking objects, determining position and location, or performing Simultaneous Localization and Mapping (SLAM) tasks. Each approach has its own advantages and limitations, making them suitable for different applications.

Perspective and Orientation

Inside Looking-Out: ILO systems are typically installed within the environment or on the tracked object itself. These cameras capture the internal view of the surroundings, allowing for a detailed understanding of the object's immediate environment.

Outside Looking-In: OLI systems, on the other hand, are positioned external to the tracked environment, observing the scene from an external perspective. This provides a global view of the surroundings and the tracked object.

Coverage and Range

Inside Looking-Out: ILO systems excel in providing high-resolution and detailed information about the immediate surroundings. However, they may have limitations in terms of the overall coverage and range, especially in large or open environments.

Outside Looking-In: OLI systems offer broader coverage and are well-suited for tracking objects over larger distances. They are particularly effective in outdoor environments where a global view is essential.

Accuracy and Precision

Inside Looking-Out: Due to the proximity to the tracked object, ILO systems often achieve high accuracy and precision in tracking and positioning. They are suitable for applications that require fine-grained spatial information. Examples like: Robot navigation, Virtual Reality (VR) devices.

Outside Looking-In: OLI systems may face challenges in achieving the same level of accuracy as ILO systems, especially when dealing with small objects or intricate details. However, advancements

in technology are continually improving the accuracy of OLI systems. They have applications like surveillance cameras, reidentification tasks and Motion capture systems.

2. Design a mono-camera ILO tracking system with smartphone camera

To implement a tracking system, we use a smartphone camera, a checkerboard, markers which in this case are QR codes and we use MATLAB as developing environment regarding its reach tools and functions that makes development way more convenient.

Camera Calibration

As First step in building a tracking system, we must calibrate the camera. Following the guidelines in the Homework description, we calibrate the camera using checkerboard. The checkerboard pattern is the most commonly used calibration pattern for camera calibration. The control points for this pattern are the corners that lie inside the checkerboard. Because corners are extremely small, they are often invariant to perspective and lens distortion

To provide a suitable board, we follow these rules[1] in generating the board:

- Use a checkerboard that contains an even number of squares along one edge and an odd number of squares along the other edge, with two black corner squares along one side and two white corner squares on the opposite side. This enables the MATLAB camera calibrator tool to determine the orientation of the pattern and the origin. The calibrator assigns the longer side as the *x*-direction. A square checkerboard pattern can produce unexpected results for camera extrinsics.
- We attach the checkerboard printout to a flat surface. Imperfections on the surface can affect the accuracy of the calibration.
- To improve detection speed, set up the pattern with as little background clutter as possible. (for this, we make sure to left a white margin around the board)

In the image [1] you can see the provided checkerboard.

[1] From MATLAB documentation for having a better calibration

To calibrate the camera, we use the camera calibration tool in MATLAB. This tool can be enabled using “cameraCalibrator” command or via the user interface. MATLAB documentation suggests using at least 10 to 20 photos. Therefore, I captured 24 Images including the checkerboard pattern in different shapes and positions. The calibration final result is shown in image [2]. It is good to mention that 2 of those 24 images had a high mean error, so I removed them. In the final graph, you can see only 22 photos used in calibration and that’s why.

Main goal of calibration is to find the intrinsics of the camera. We use the intrinsics later to undistort images captured by the same camera and to solve the problem of mapping the points in image coordinate system and world coordinate system. The intrinsics parameters are shown in image [3]. You can also see the camera initial settings in image [4].

Finding Camera Position in World Coordinate

Finding the camera position and orientation consists of three main steps: camera calibration, establishing a mapping between world points and their corresponding points in the captured image, and finally, estimating the camera pose. Now we discuss the second step.

Calculating calibration parameters is useful to undistort images captured by the camera with exactly the same parameters. In the procedure of camera pose estimation, first we capture an image with visible markers in it.

The markers I provided for this experiment are 9 QR codes, each 47×47 milimeters. Markers are printed on A3-sized paper in squared orientation. See image [5]. I placed the markers on a wall, assuming that wall would be the XY-plain. The Z-coordinate will be moving from the camera towards the wall, so we expect to have negative values for the camera position because in XY-plain, the Z coordinates are equal to zero. The top left corner of the first QR marker will be the origin.

Let’s move on to the next part. After placing the markers on the wall, we take a photo with the same camera and settings we used earlier in the calibration phase. Using the calibration parameters, it is necessary to undistort the image. From now on, we use this undistorted image instead of the original one. On the undistorted image, we must find the markers. I wrote a function called “barcodeFinder”. Using this function, we can locate the top left corner of each marker. I used this top-left corner as the representation point of the marker. I use top-left, because I consider top-left marker as the origin, so the top-left corner of this marker matches exactly with the (0,0) coordination.

Having marker points on the image, we still need another set of points, the points corresponding to coordinates in the world coordinate system. I calculated the points in the world system by manually

measuring the distances between each marker and marker size. The final coordinates were stored in a matrix. See image [6]. Numbers are in millimeters.

Now, using world points and image points, we can calculate the camera pose by find best vectors that match the corresponding points in the world to those of the image points. MATLAB function “estworldpose” gets image points, world points, and camera intrinsic and calculates the Rotation, Translation, and Affine matrices of the camera with respect to inputs. Using those matrices, it is possible to find the location and orientation of the camera in world coordinate systems.

Discussion

Camera Calibration

To evaluate the accuracy of our calibration, MATLAB provide a useful plot for mean error per photo (image [7]) and a visualization of camera-pattern in 3D space. (image [8])

We can observe from the first plot that the overall mean error for this calibration session is 0.31 pixels. To get a lower error rate, I tried to place pattern in different angles and positions in images. You can see in images [9,10,11] that we have pattern on different sides of the picture frame. As I mentioned before, 2 images have been removed due to high mean error. Achieving such a low error rate is promising. It is worthwhile to mention that it seems using images whose pattern has an angle of more than 45 degrees to the camera image plane is not useful. Those removed images have such high angles. (images [12,13])

Finding Camera Position in World Coordinate

To undistort an image, we load the test image and using ‘undistortImage’ function we can correct the image distortion. (images [14,15])

The next step is finding barcodes using ‘barcodeFinder’ (image [16]). The function uses a for loop to find all 9 markers. Using all those markers will help us achieve better results. Inside the loop, you will find a function called ‘readBarcode’. This function is a MATLAB function to find a bar code in an image. For better results, we use a region of interest (ROI) and we want the function to look for a marker only inside that area (image [17]. Parameter ‘I’ is the input image and “QR-CODE” indicates that the function must look for 2D QR patterns. The output coordinates will be our desired image points.

The last phase is finding the correspondence between image points and world points. As I searched the web, actually we need to solve Perspective N-Points (PNP) problem. The perspective N-points problem, commonly encountered in computer vision and robotics, involves determining the three-dimensional coordinates of a set of points in space by analyzing their two-dimensional projections in an image or a series of images taken from different viewpoints. The challenge arises due to the loss of depth information in the projection process. One popular approach involves using a set of correspondences between the 2D image points and their corresponding 3D world coordinates. It is what we are going to do.

In MATLAB, you can do this by using ‘estworldpose’. This function needs image points, world points and camera intrinsics (See image [18]). As observable in the image [19], the final result contains a projection of 9 markers (shown with cyan) and the camera (shown in red). You can see that there is a reasonable matching between the camera view in image [17] and the projection in image [19]. See final output of ‘estworldpose’ in image [20].

A Test Scenario

To evaluate our results, it is good to inspect some key angles and views to see If the final projection is logical or not. We consider 3 photos from 3 different views. See image [21,22,23]. I will reference image [21] as left, [22] as bottom and [23] as top for short, because they provide views from left, bottom and top respectively. The left and bottom have the same distance from the camera. It helps us to check the relative distance measurement accuracy. The bottom and top tried to be in the same position on the X-axis, providing a difference in height and angle. For the left, you can see that the left edge of the image is parallel to the Y-axis, with a small margin, because the camera lens is not in the center of the phone but on its left side. This difference should be considered in the analysis of all three images. The top edge in the top image and the bottom edge in the bottom image are parallel to the X-axis. The top image tries to provide a top-down view with the upper edge limiting the view of $Y<0$ (a positive pitch), the left image does the same for $X<0$ (a positive yaw).

Now let’s find out whether the projection can provide the same details as we expect or not. The final projection is shown in images [24, 25]. The red camera represents the left view, the green represents the bottom and the blue one is the top. There is also a bunch of other 3D views you can see. Image [26] represents the YX plane view. Image [27] is for the XZ plane. To check the relative distance from the wall, check that and the YZ view in the image [28]. Image [29] provides a view considering an observer Is behind the wall. Images [30, 31, 32] show relevant matrices.

To find all scripts, you can see appendix. Appendix [1] is the main program routine and appendix [2] is the ‘barcodeFinder’ function.

Results

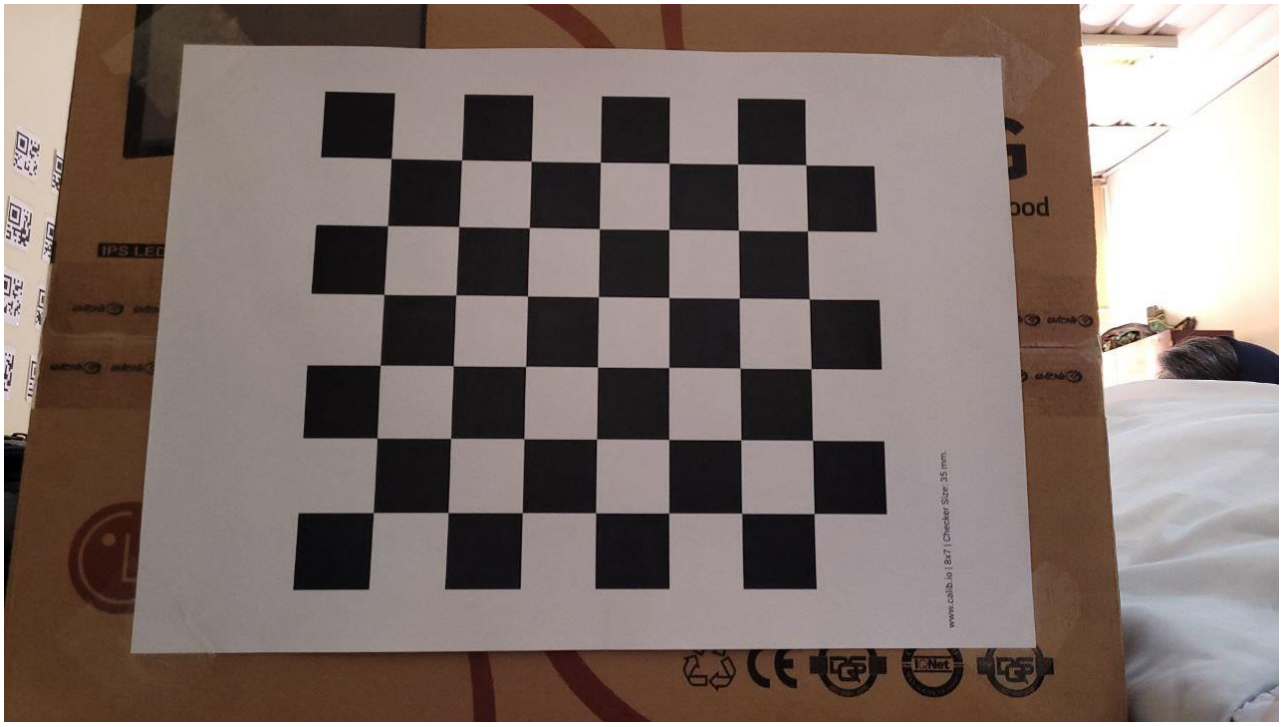

Image 1: Checker board Pattern

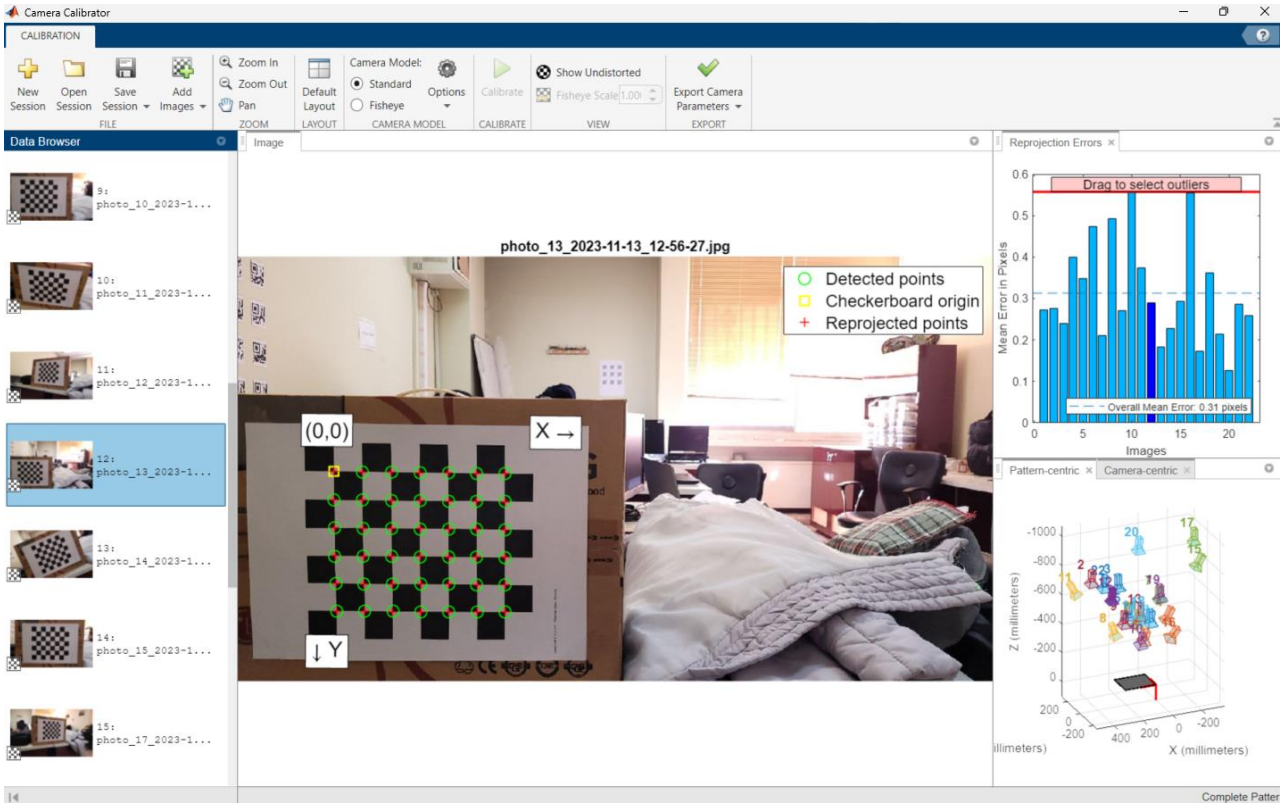

Image 2: Calibration result

cameraParams.Intrinsics

Property	Value
FocalLength	[975.8458,978.2060]
PrincipalPoint	[643.8471,355.5835]
ImageSize	[720,1280]
RadialDistortion	[-0.0265,0.2139,-0.5068]
TangentialDistortion	[0,0]
Skew	0
K	[975.8458,0,643.8471;0,978.2060,355.5835;0,0,1]

Image 3: Camera intrinsics

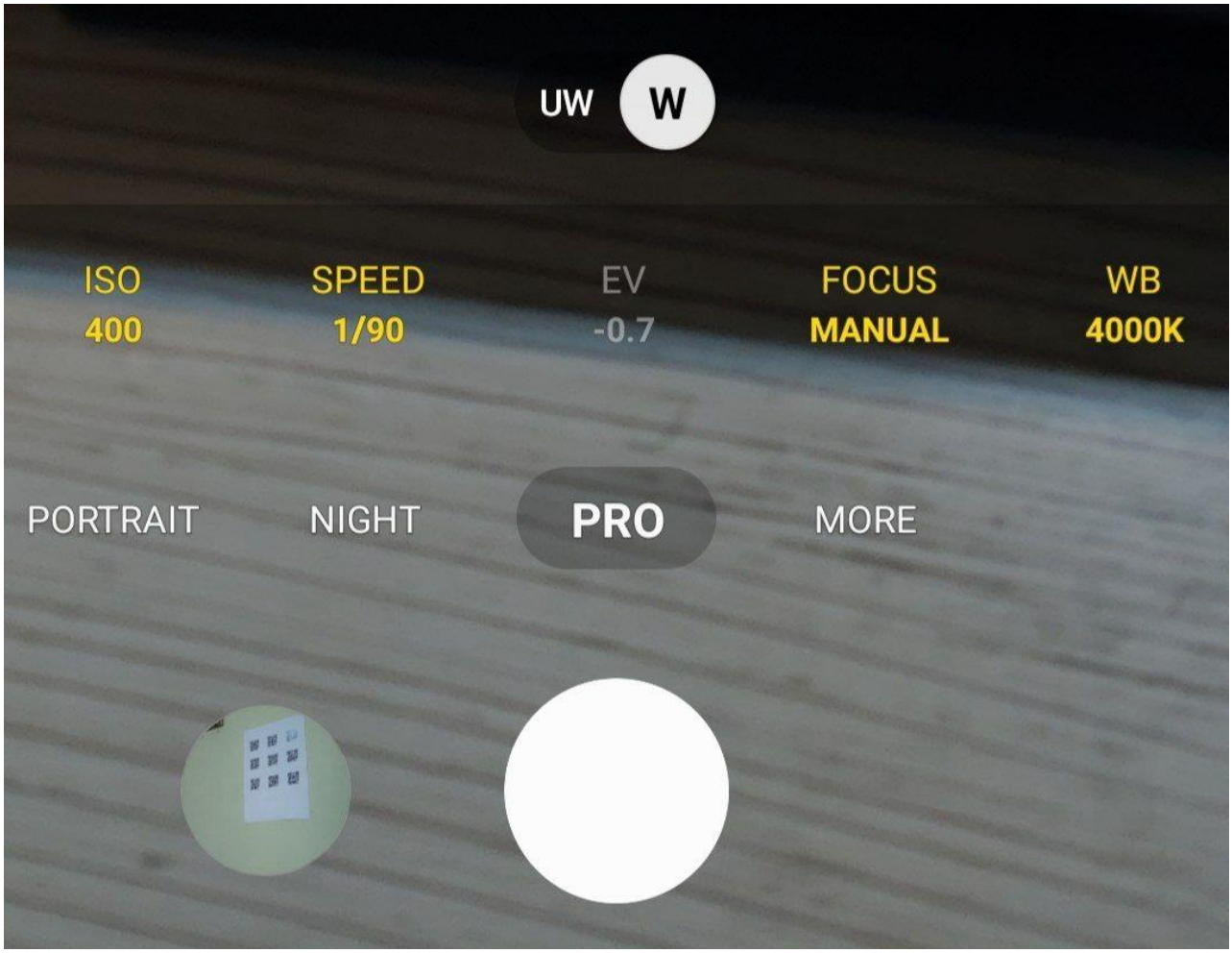

Image 4: Camera settings

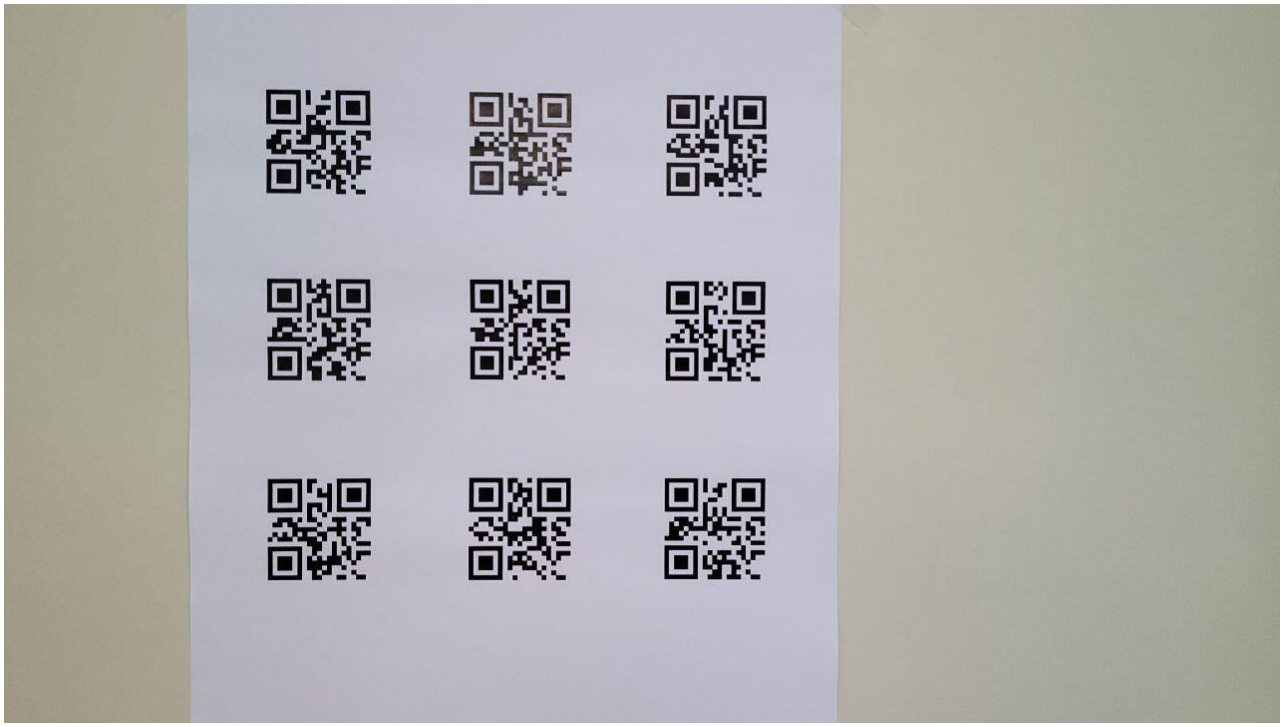

Image 5: QR code markers

```
worldPoints = [0 0;94 0;185 0;0 86;94 86;185 86;0 178;94 178;185 178];

zCoord = zeros(size(worldPoints,1),1);
worldPoints = [worldPoints zCoord];
```

Image 6: World points coordinates (in milimeters)

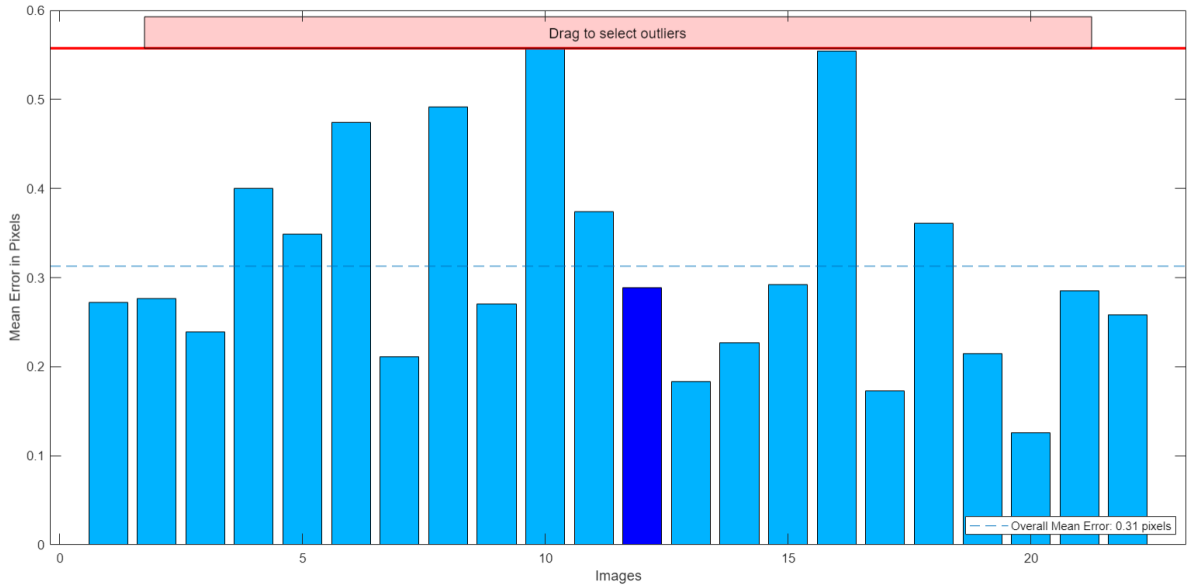

Image 7: Mean error per photo

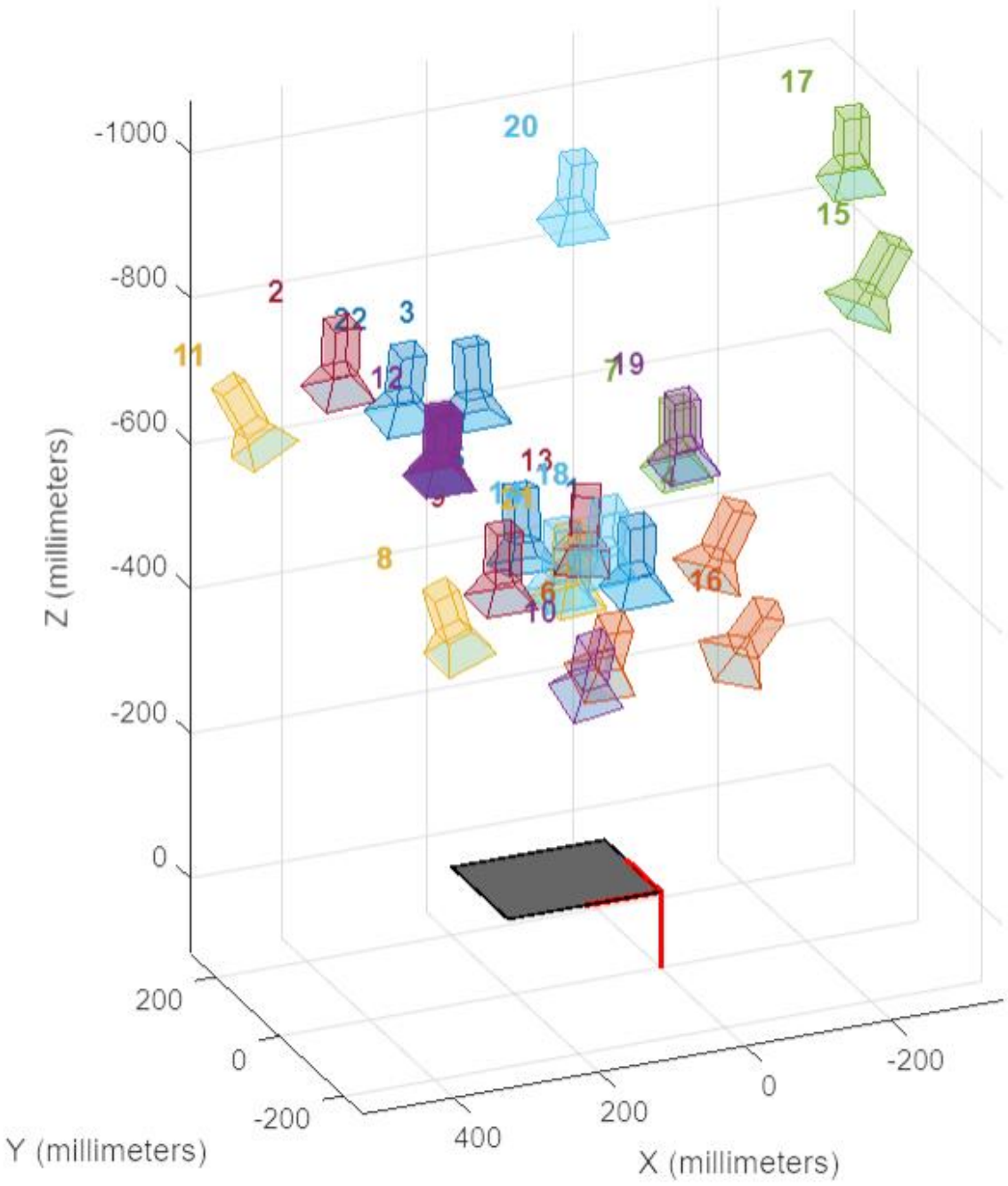

Image 8: Pattern centric view

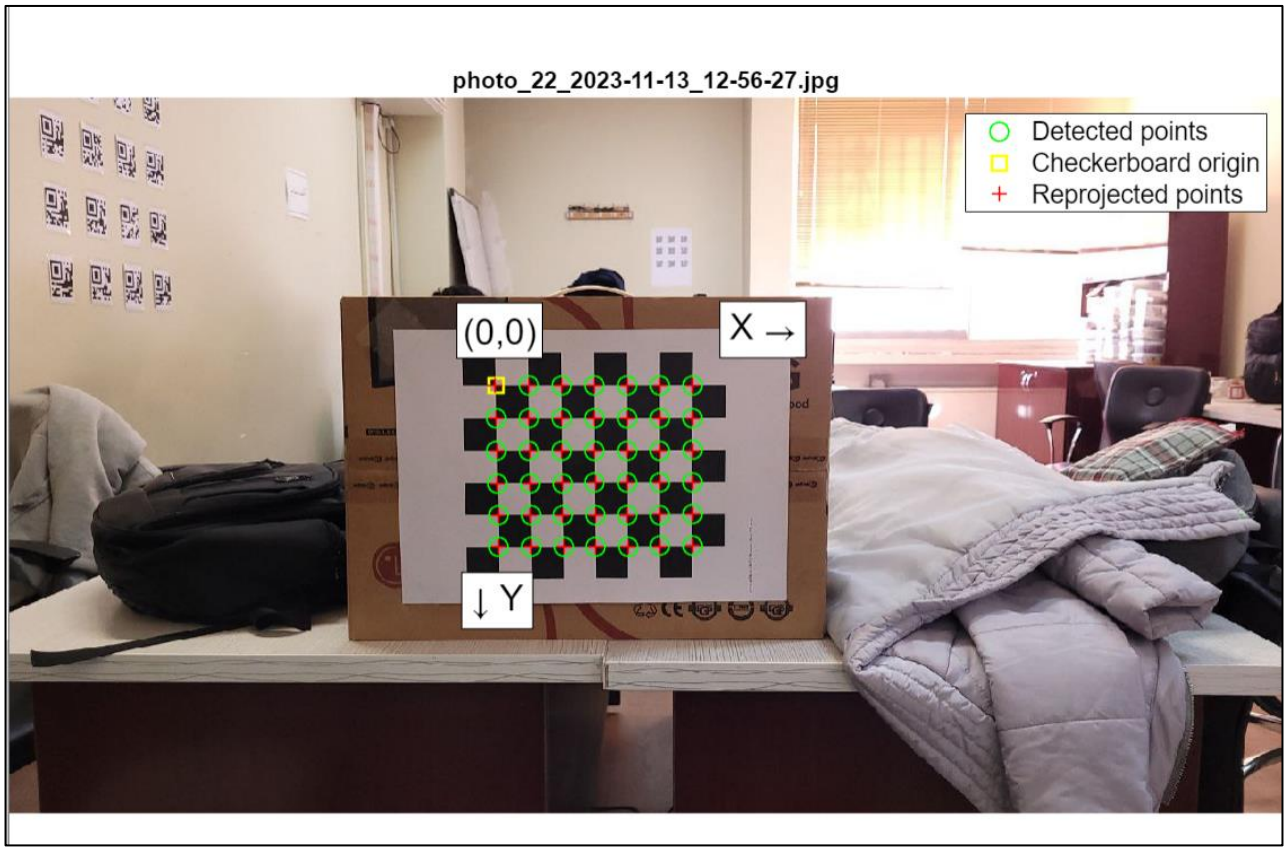
photo_22_2023-11-13_12-56-27.jpg
Detected points
Checkerboard origin
Reprojected points
(0,0)
X →
↓ Y

Image 9

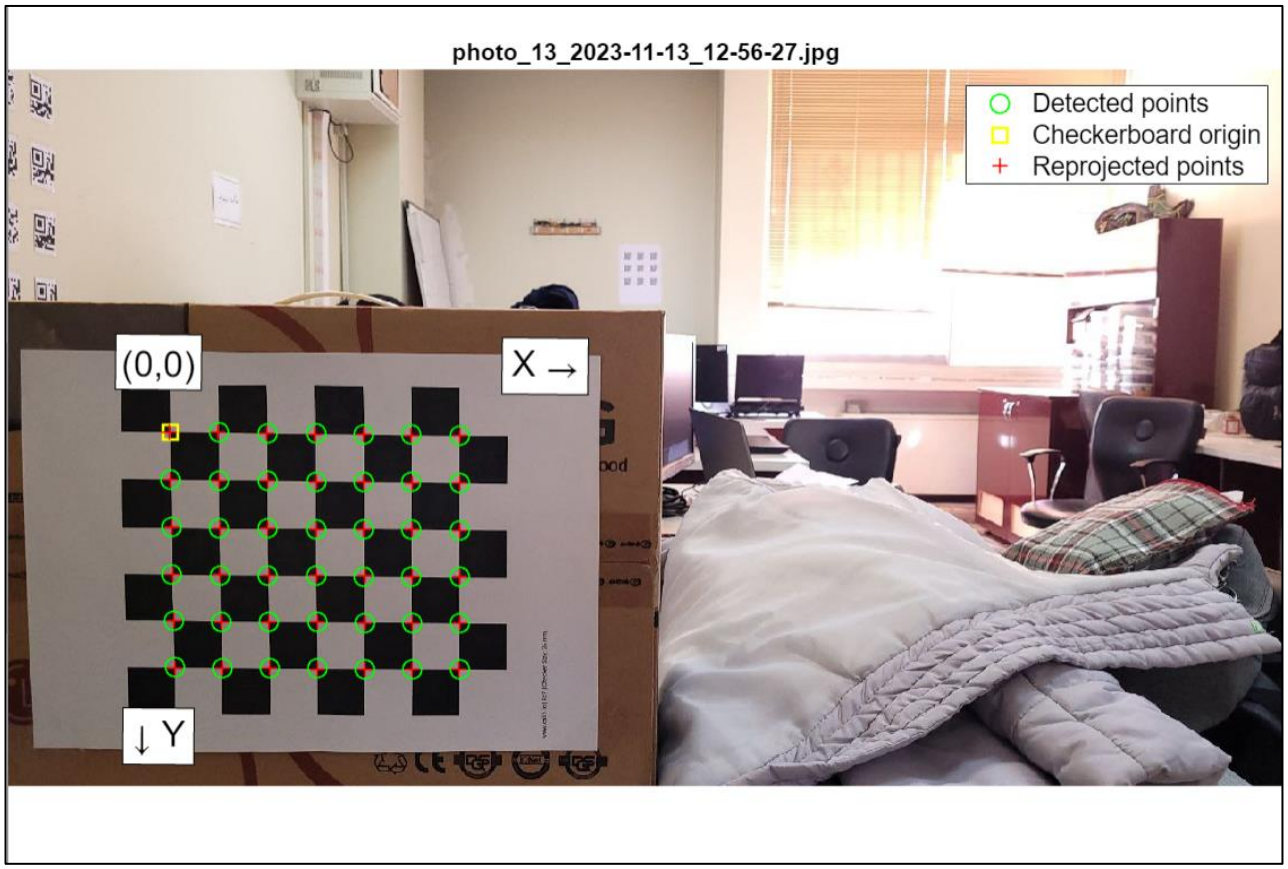
photo_13_2023-11-13_12-56-27.jpg
Detected points
Checkerboard origin
Reprojected points
(0,0)
X →
↓ Y

Image 10

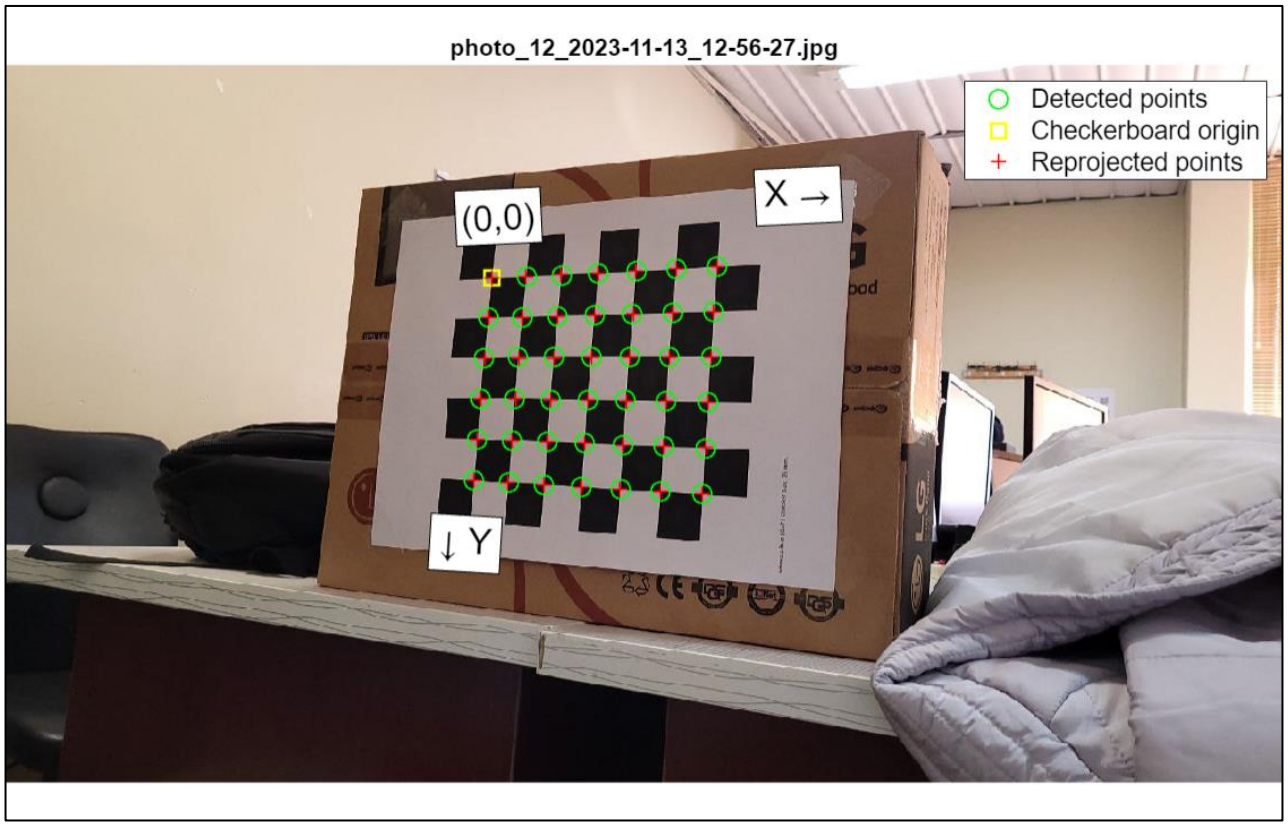

Image 11

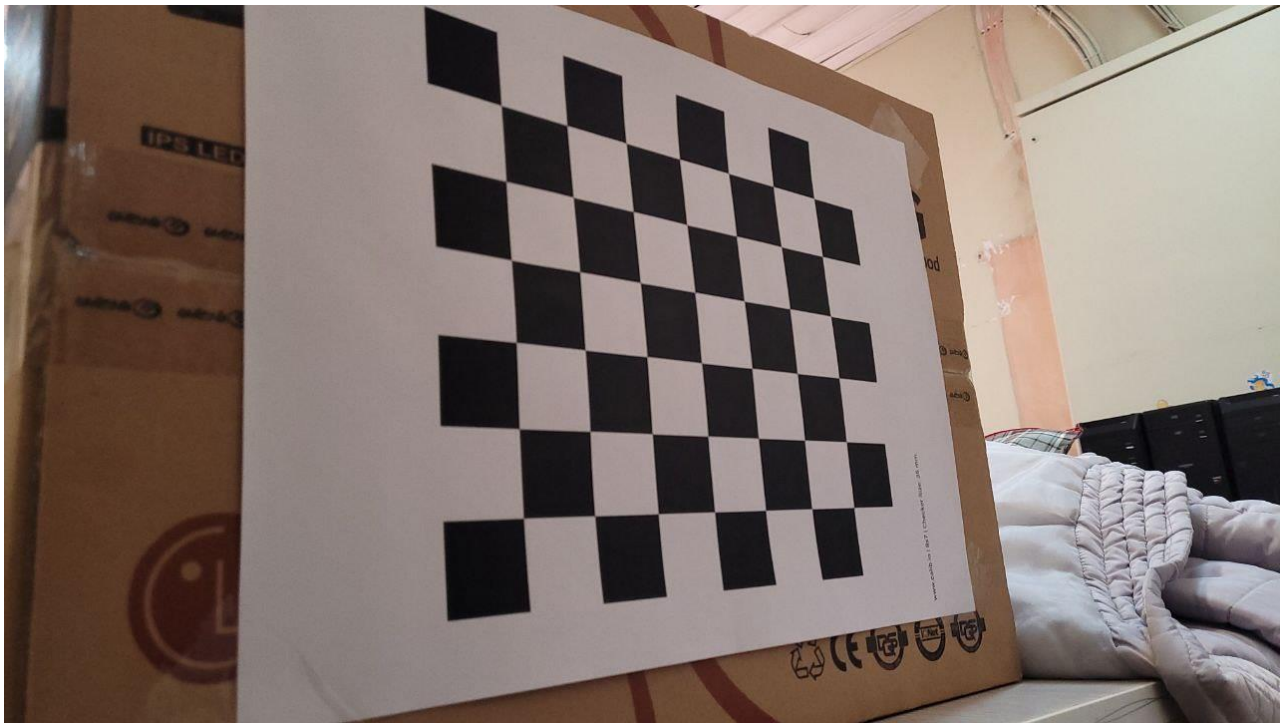

Image 12: Removed image

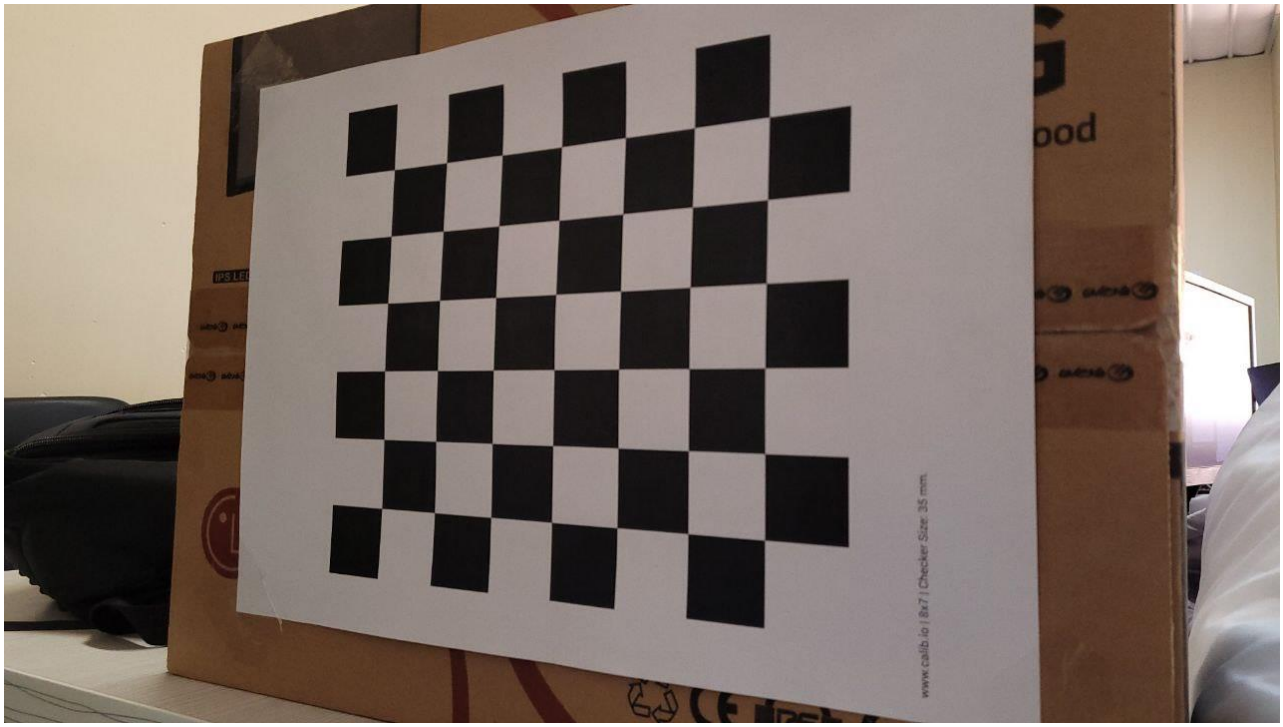

Image 13: Removed image

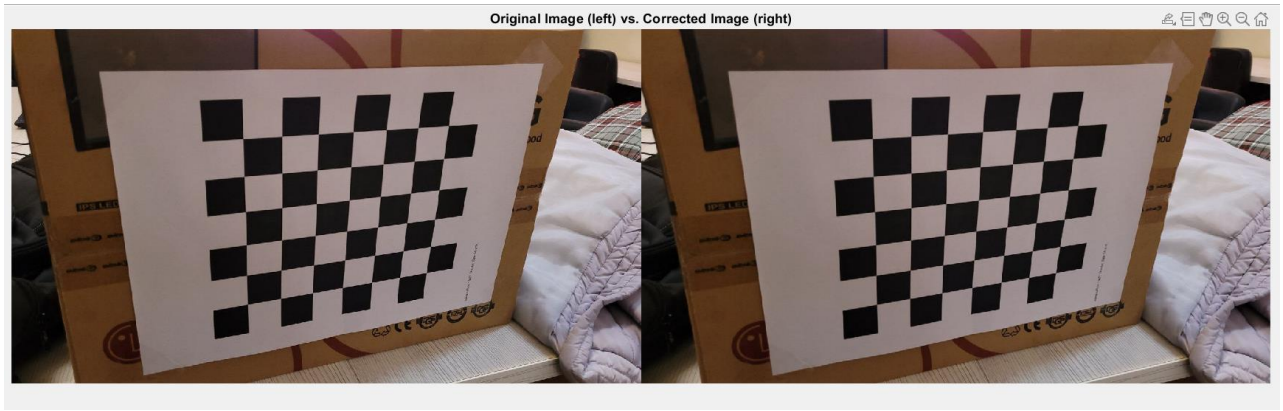

Image 14: Original(left) vs undistorted(Right)

```
oriImage = imread("qr\new\test5.jpg");

% You can use the calibration data to remove effects of lens distortion.
undistortedImage = undistortImage(oriImage, cameraParams);
```

Image 15: MATLAB code

barcodeFinder.m

```
function points = barcodeFinder(I)
    points = zeros(9,2);
    for i = 1:9
        imshow(I);

        roi1 = drawrectangle;
        roi = roi1.Position;

        % Search the image for a QR Code.
        [msg, format, loc] = readBarcode(I,roi(1,:),"QR-CODE");
         disp("Decoded format and message: " + format + ", " + msg)

        % Annotate the image with the decoded message.
        xyText =  loc(2,:);
        Imsg = insertText(I, xyText, msg, "BoxOpacity", 1, "FontSize", 25);

        % Insert filled circles at the finder pattern locations.
        Imsg = insertShape(Imsg, "filled-circle", [loc, ...
            repmat(10, length(loc), 1)], "Opacity", 1);

        points(i,:) = xyText;
        save("imagePoints.mat","points");

        % Display image.
        imshow(Imsg)
    end
end
```

Image 16: barcodeFinder

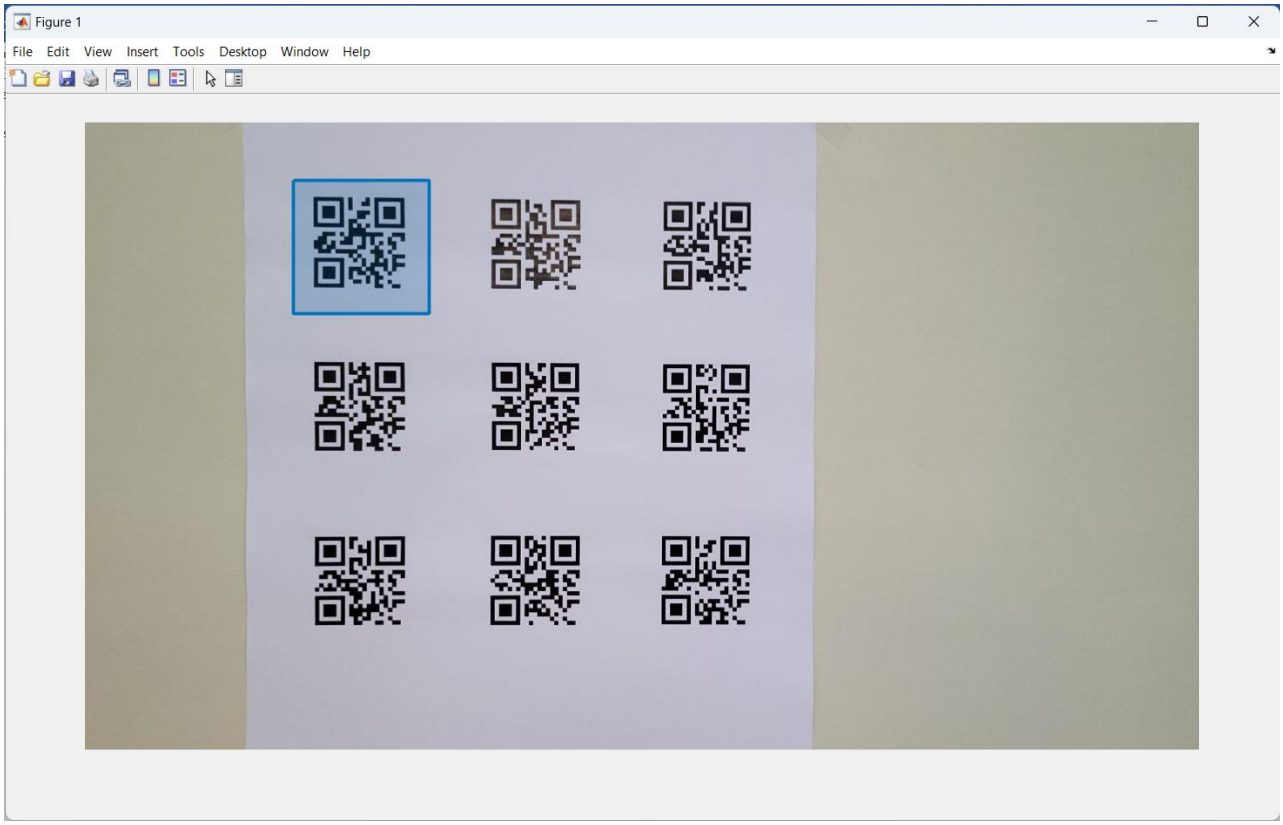

Image 17: ROI selection

```
worldPose = estworldpose(imagePoints,worldPoints,cameraParams.Intrinsics);

pcshow(worldPoints,VerticalAxis="Y",VerticalAxisDir="down", ...
    MarkerSize=40);
hold on
plotCamera(Size=10,Orientation=worldPose.R', ...
    Location=worldPose.Translation);
hold off
```

Image 18: estworldpose function

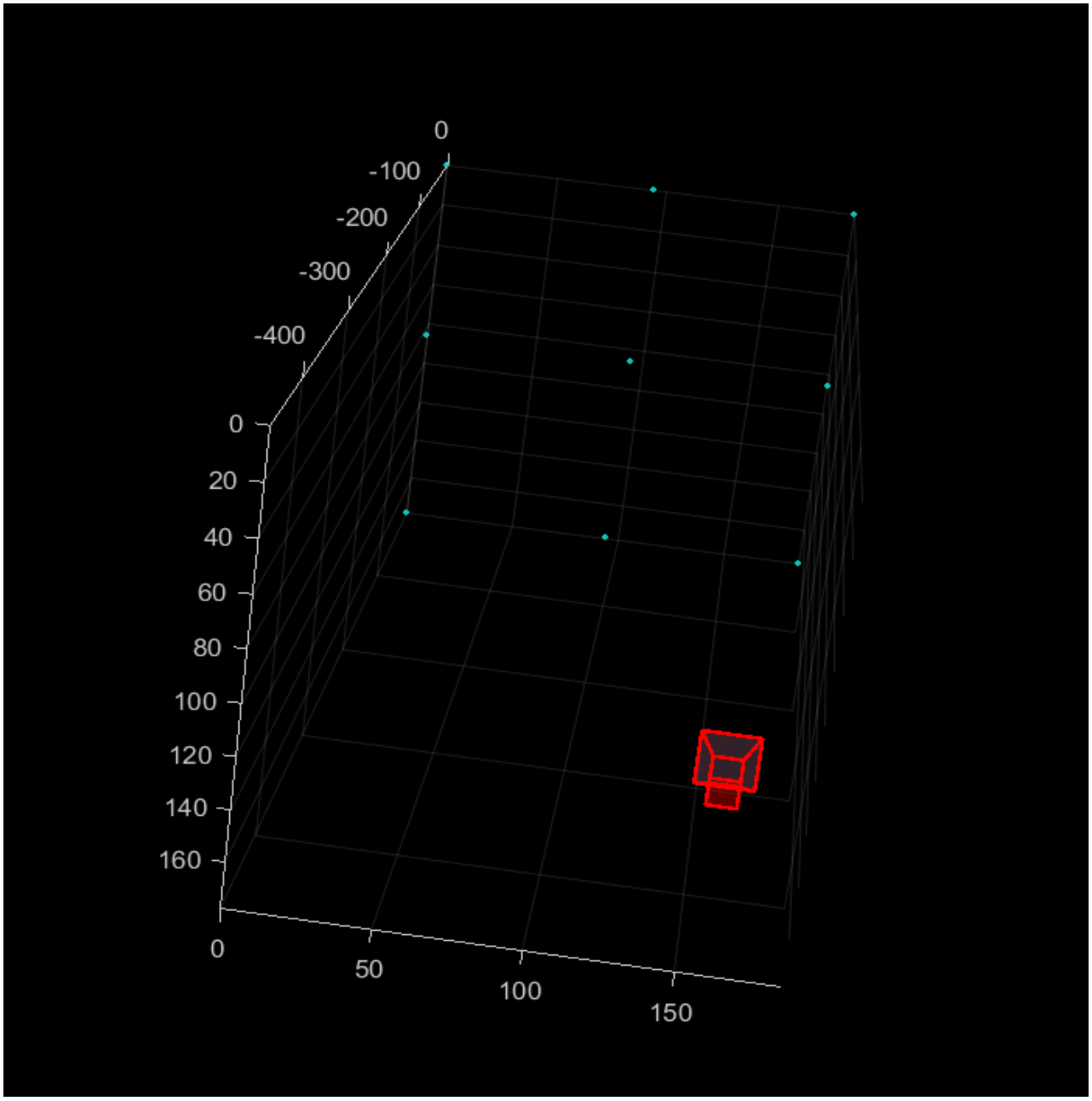

Image 19: Final camera position

worldPoseLeft | worldPose | worldPoseBottom | worldPoseTop

1x1 rigidtform3d

Property	Value
Dimensionality	3
Translation	[153.9558,102.7069,-453.7663]
R	[0.9996,0.0032,0.0290;-0.0039,0.9997,0.0243;-0.0289,-0.0244,0.9993]
A	*4x4 double*

Image 20: Camera position

Image 21: The left

Image 22: The bottom

Image 23: The top

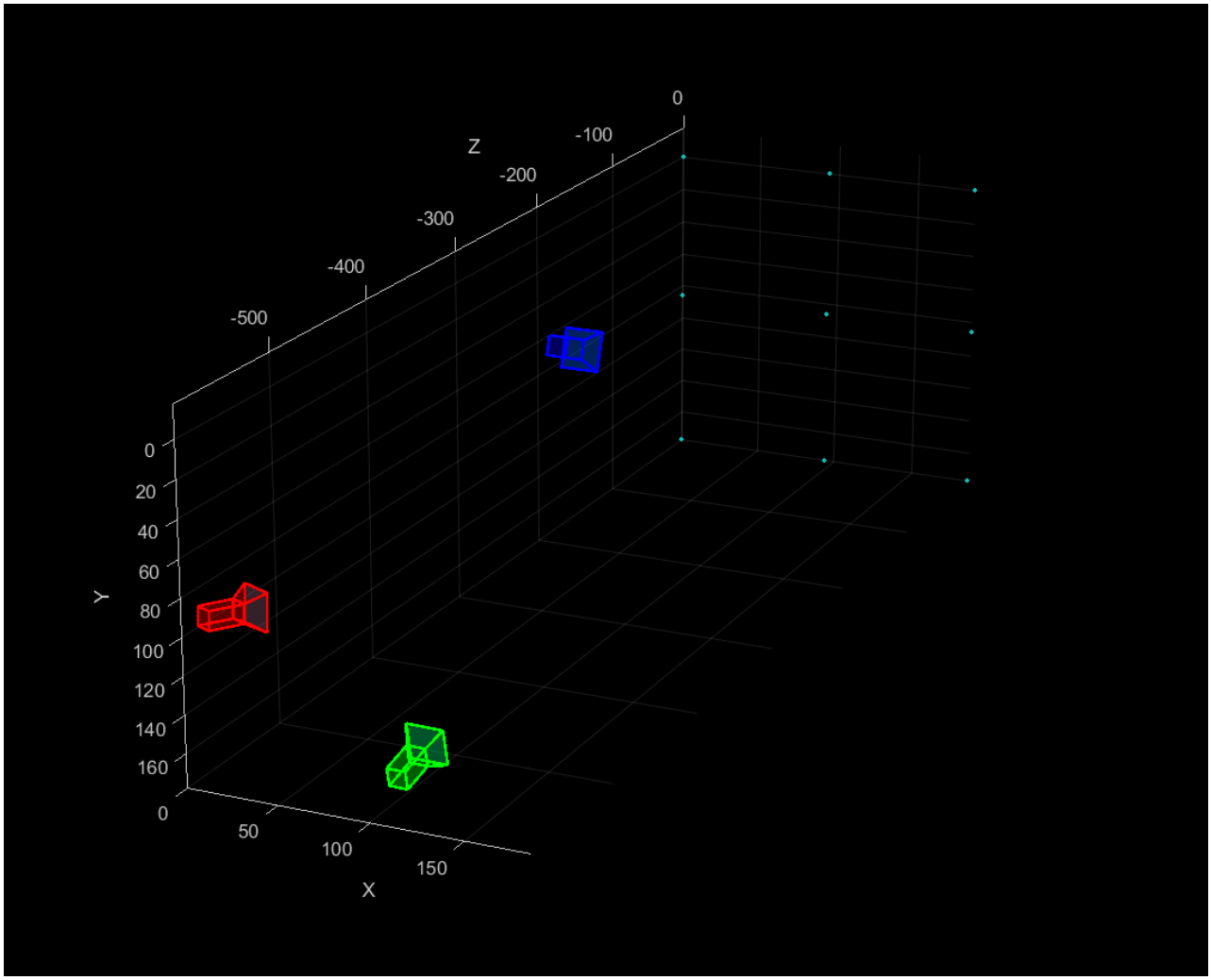

Image 24: Final projection view 1

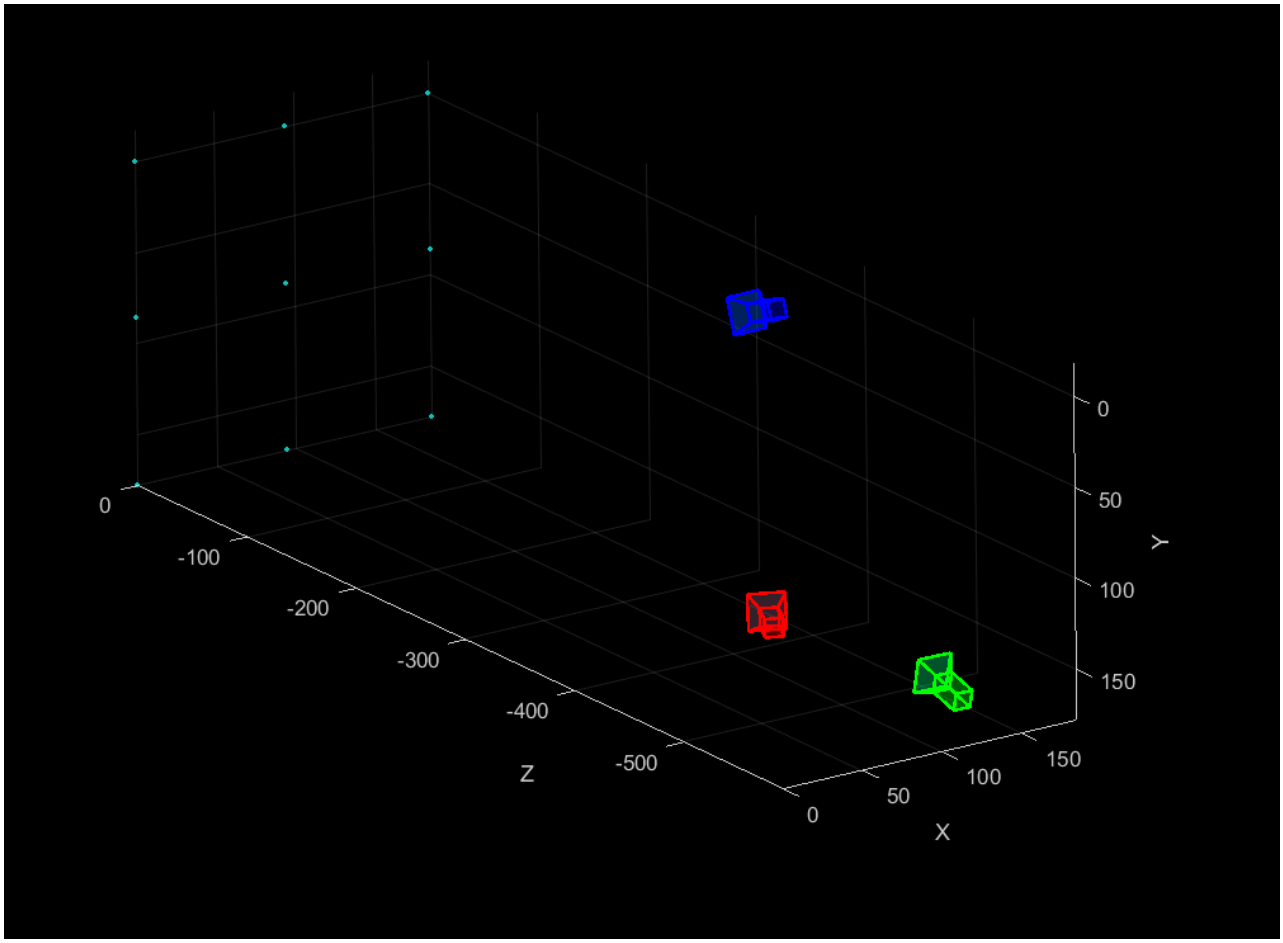

Image 25: Final projection view 2

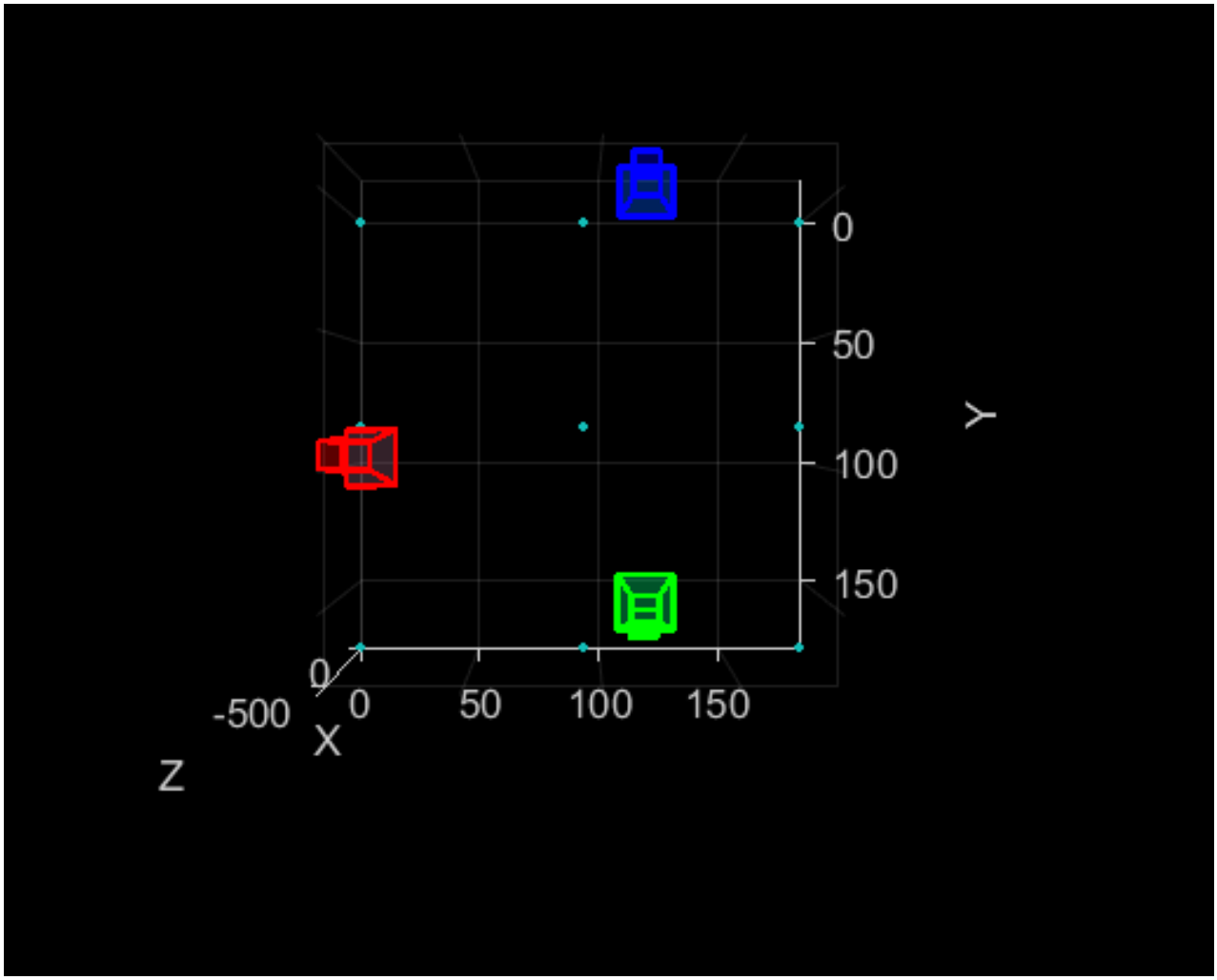

Image 26: YX plane

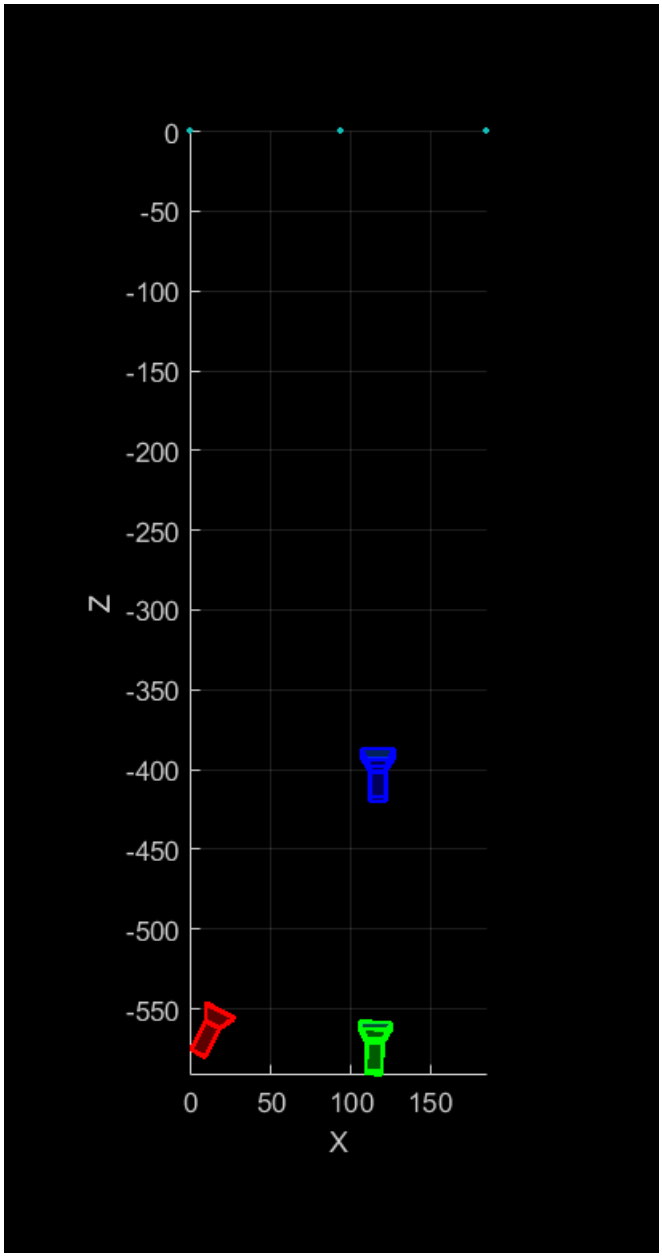

Image 27: XZ plane

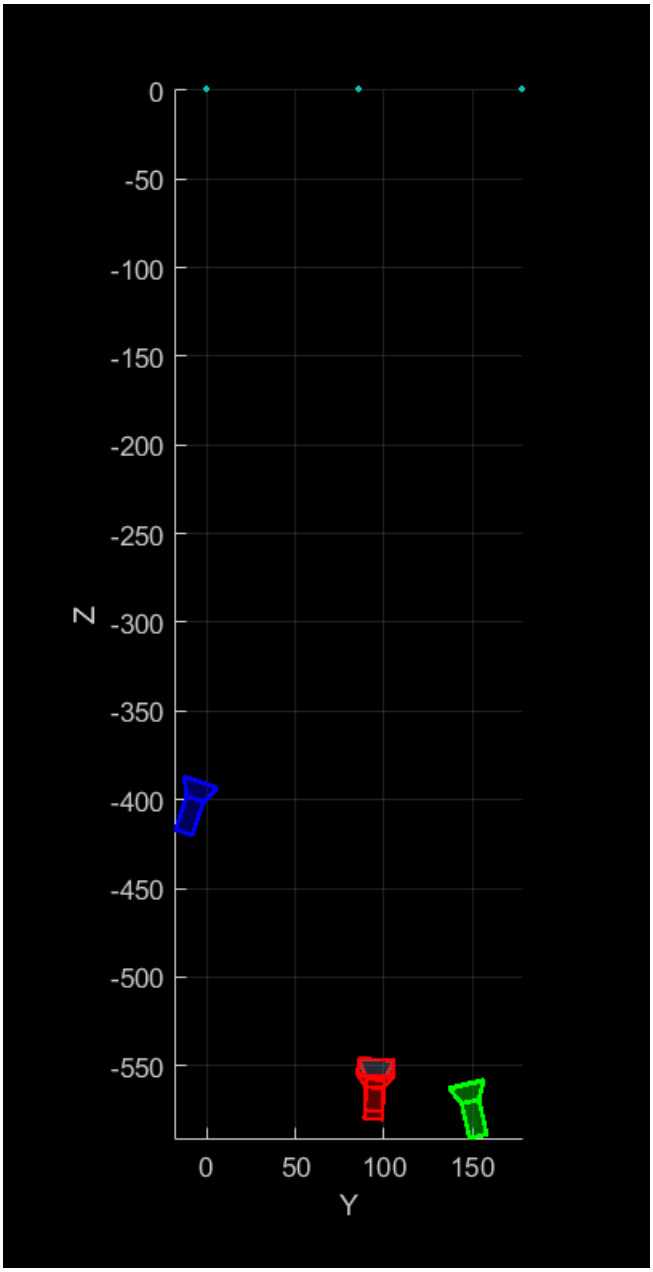

Image 28: YZ plane

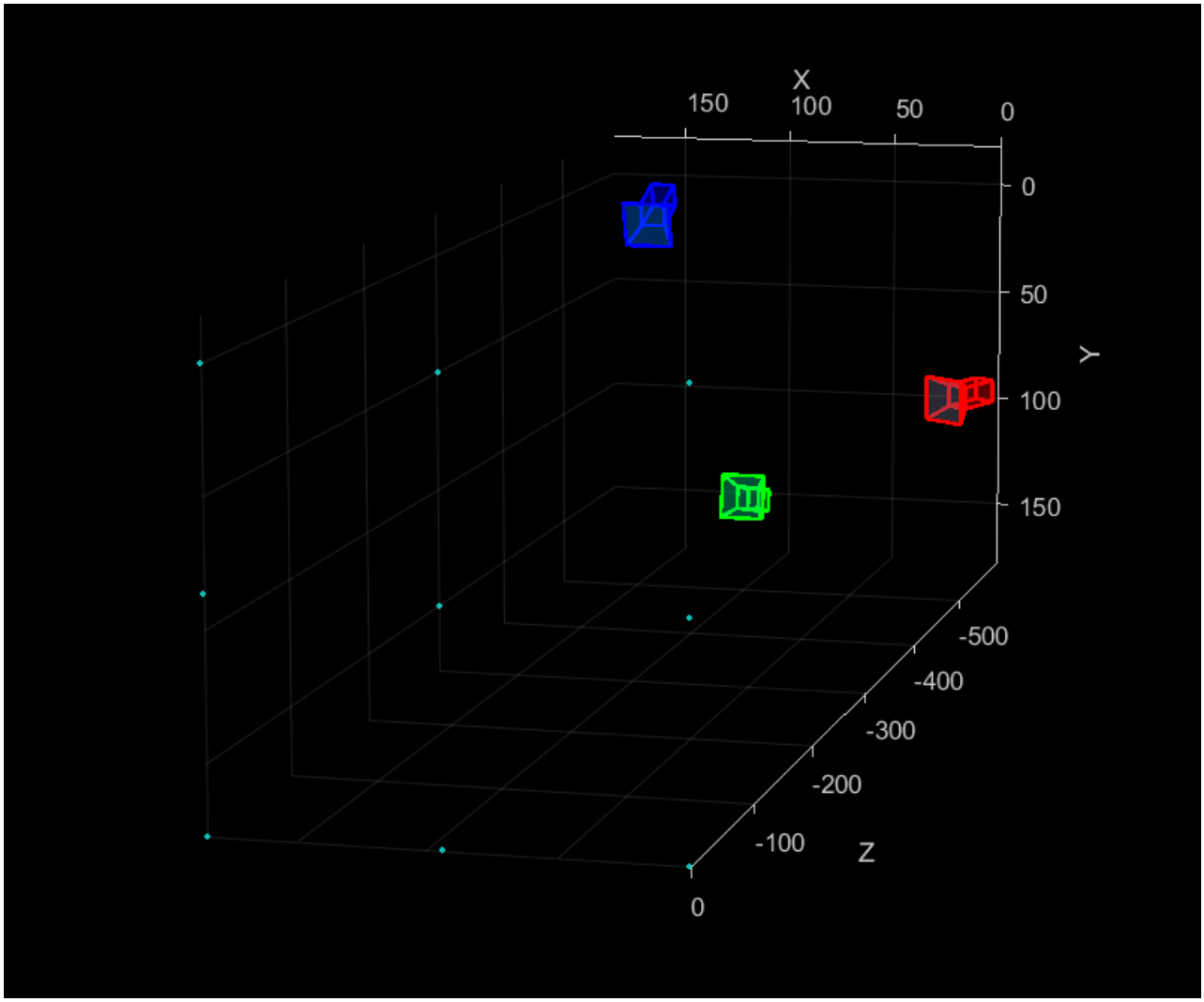

Image 29: Behind the wall view

worldPoseLeft | worldPose | worldPoseBottom | worldPoseTop

1x1 rigidtform3d

Property	Value
Dimensionality	3
Translation	[3.1517,116.4739,-566.1061]
R	[0.8836,0.0038,0.4682;-0.0103,0.9999,0.0114;-0.4682,-0.0149,0.8835]
A	*4x4 double*

Image 30: The left pose parameters

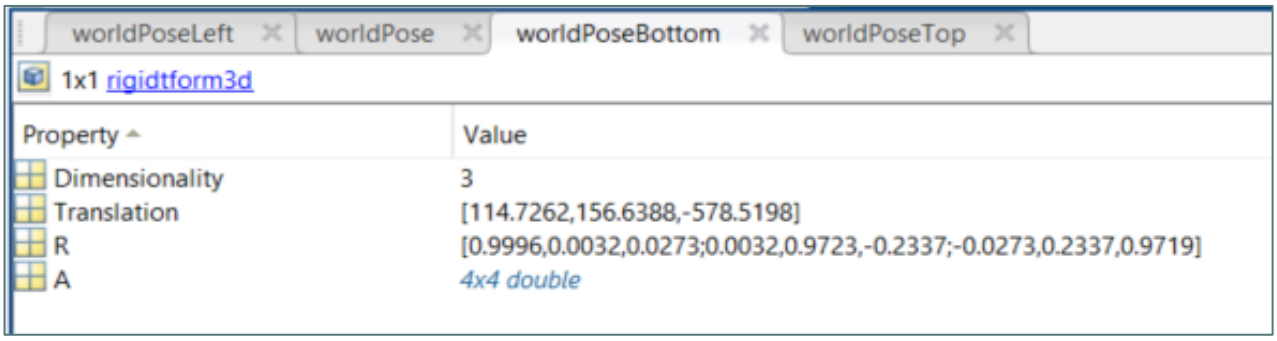

Image 31: The bottom pose parameters

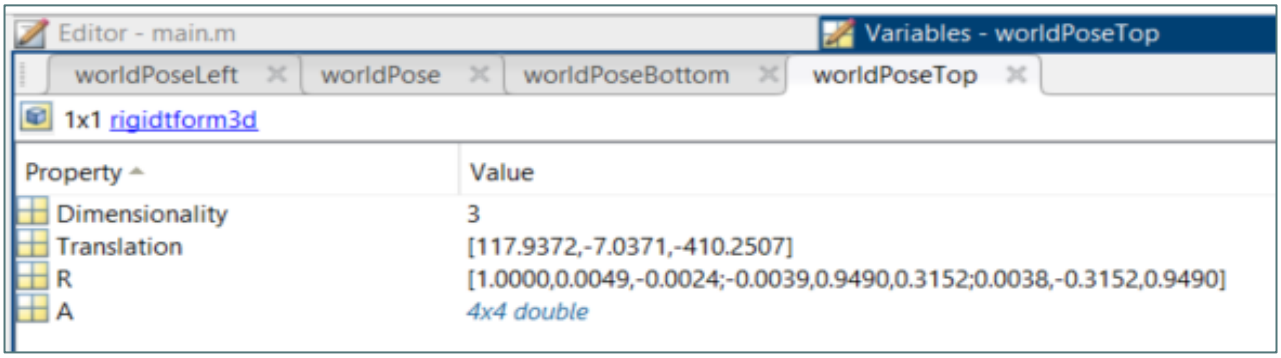

Image 32: The top pose parameters

Appendix

[1] ‘main’ script:

```
% Define images to process
imageFileNames = {'E:\taha\code\camera-calibration\mono-calibration\checkerboard-
pr\photo_1_2023-11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_2_2023-11-
13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_3_2023-11-
13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_4_2023-11-
13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_5_2023-11-
13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_6_2023-11-
13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_7_2023-11-
13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_8_2023-11-
13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_10_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_11_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_12_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_13_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_14_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_15_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_17_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_18_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_19_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_20_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_21_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_22_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_23_2023-
11-13_12-56-27.jpg',...
    'E:\taha\code\camera-calibration\mono-calibration\checkerboard-pr\photo_24_2023-
11-13_12-56-27.jpg',...
    };
% Detect calibration pattern in images
detector = vision.calibration.monocular.CheckerboardDetector();
[imagePoints, imagesUsed] = detectPatternPoints(detector, imageFileNames);
imageFileNames = imageFileNames(imagesUsed);
```

```
% Read the first image to obtain image size
originalImage = imread(imageFileNames{1});
[mrows, ncols, ~] = size(originalImage);

% Generate world coordinates for the planar pattern keypoints
squareSize = 35;  % in units of 'millimeters'
worldPoints = generateWorldPoints(detector, 'SquareSize', squareSize);

% Calibrate the camera
[cameraParams, imagesUsed, estimationErrors] = estimateCameraParameters(imagePoints,
worldPoints, ...
    'EstimateSkew', false, 'EstimateTangentialDistortion', false, ...
    'NumRadialDistortionCoefficients', 3, 'WorldUnits', 'millimeters', ...
    'InitialIntrinsicMatrix', [], 'InitialRadialDistortion', [], ...
    'ImageSize', [mrows, ncols]);

% View reprojection errors
%h1=figure; showReprojectionErrors(cameraParams);

% Visualize pattern locations
h2=figure; showExtrinsics(cameraParams, 'CameraCentric');

% Display parameter estimation errors
%displayErrors(estimationErrors, cameraParams);

oriImage = imread("qr\new\test5.jpg");

% You can use the calibration data to remove effects of lens distortion.
undistortedImage = undistortImage(oriImage, cameraParams);

imagePoints = barcodeFinder(undistortedImage);

worldPoints = [0 0;94 0;185 0;0 86;94 86;185 86;0 178;94 178;185 178];

zCoord = zeros(size(worldPoints,1),1);
worldPoints = [worldPoints zCoord];

% Save cameraParams
paramStruct = toStruct(cameraParams);
save("camera-params.mat", 'paramStruct');

worldPose = estworldpose(imagePoints,worldPoints,cameraParams.Intrinsics);

pcshow(worldPoints,VerticalAxis="Y",VerticalAxisDir="down", ...
    MarkerSize=40);
hold on
plotCamera(Size=10,Orientation=worldPose.R', ...
    Location=worldPose.Translation);
hold off

orileft = imread("qr\new\left.jpg");
oribottom = imread("qr\new\bottom.jpg");
oritop = imread("qr\new\top.jpg");

%cameraParams = load("camera-params.mat","paramStruct");
```

```
% You can use the calibration data to remove effects of lens distortion.
undistortedTop = undistortImage(oritop, cameraParams);
undistortedLeft = undistortImage(orileft, cameraParams);
undistortedBottom = undistortImage(oribottom, cameraParams);

topPoints = barcodeFinder(undistortedTop);
leftPoints = barcodeFinder(undistortedLeft);
bottomPoints = barcodeFinder(undistortedBottom);

worldPoints = [0 0;94 0;185 0;0 86;94 86;185 86;0 178;94 178;185 178];

zCoord = zeros(size(worldPoints,1),1);
worldPoints = [worldPoints zCoord];

worldPoseTop = estworldpose(topPoints,worldPoints,cameraParams.Intrinsics);
worldPoseLeft = estworldpose(leftPoints,worldPoints,cameraParams.Intrinsics);
worldPoseBottom = estworldpose(bottomPoints,worldPoints,cameraParams.Intrinsics);


pcshow(worldPoints,VerticalAxis="Y",VerticalAxisDir="down", ...
    MarkerSize=50);
hold on
plotCamera(Size=10,Orientation=worldPoseLeft.R', ...
    Location=worldPoseLeft.Translation, Color=[1 0 0]);
plotCamera(Size=10,Orientation=worldPoseBottom.R', ...
    Location=worldPoseBottom.Translation, Color=[0 1 0]);
plotCamera(Size=10,Orientation=worldPoseTop.R', ...
    Location=worldPoseTop.Translation, Color=[0 0 1]);
hold off
```

[2] ‘barcodeFinder’ function:

```
function points = barcodeFinder(I)
    points = zeros(9,2);
    for i = 1:9
        imshow(I);

        roi1 = drawrectangle;
        roi = roi1.Position;

        % Search the image for a QR Code.
        [msg, format, loc] = readBarcode(I,roi(1,:),"QR-CODE");
         disp("Decoded format and message: " + format + ", " + msg)

        % Annotate the image with the decoded message.
        xyText =  loc(2,:);
        Imsg = insertText(I, xyText, msg, "BoxOpacity", 1, "FontSize", 25);

        % Insert filled circles at the finder pattern locations.
        Imsg = insertShape(Imsg, "filled-circle", [loc, ...
            repmat(10, length(loc), 1)], "Opacity", 1);

        points(i,:) = xyText;
        save("imagePoints.mat","points");

        % Display image.
        imshow(Imsg);
    end
end
```