Computer Vision Report

Yash Belur, Tam Amabibi
yb521 (02100668,), ta1524 (06011762)

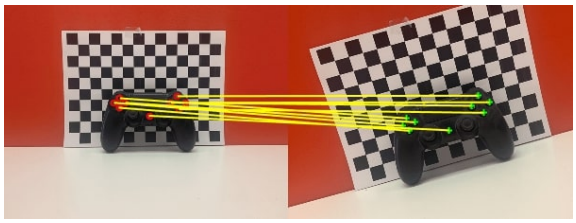

Figure 1: Manual Correspondence of the image

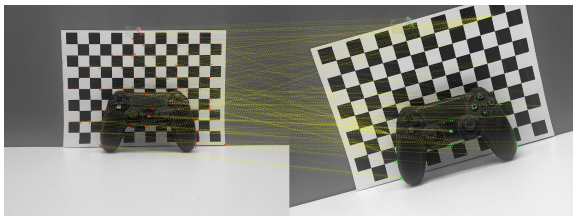

Figure 2: Automatic Correspondence of the image

Method	Mean Squared Error (MSE)	Matched Correspondences
Manual	86366.9062	10
Automatic	67881.4062	88

Table 1: Comparison of Correspondence Methods

1. Collect Data

The collected datasets of the images for both the FD and HG datasets with and without the object are shown in the appendix in figs. 11 to 14. They consist of images of a game controller against the background of a calibration grid and images without the controller.

2. Keypoint Correspondences between images

Firstly, we begin by manually estimating the keypoint correspondences to establish an initial reference before transitioning to an automated approach. This preliminary manual process allows us to ensure accuracy, gain insights into the underlying patterns, and validate the effectiveness of the automatic method.

Fig. 1 shows the image having keypoints that was placed manually under estimation. These were placed under the impression that these would obviously be keypoints when going through automatic process.

Next, now that the keypoints have been manually placed, the keypoints will be automatically placed, showing correspondences between the two images. The surf method was used using the `detectSURFFeatures()` function and they were matched using the `matchFeatures()` function. Listing This will be compared to the manually placed keypoints that correspond with each other.

Fig. 2 shows the image being automatically processed with all the keypoints of both images corresponding with each other. All pairs of correspondence are in fig. 15 in the appendix.

Comparison of Correspondence Methods

Table 1 shows a comparison between the manual and automatic correspondence methods in terms of mean squared error and number of matched correspondences. When compared, the number of correspondences and the MSE reflects the effectiveness of identifying correspondences. So, the lower MSE and higher correspondence in the automatic method suggest that it's more effective in identifying correspondences, this is potentially due to human error when clicking on the images to identify correspondences manually. However, automatic correspondence does give a lot more correspondence, a lot of which may not be very useful.

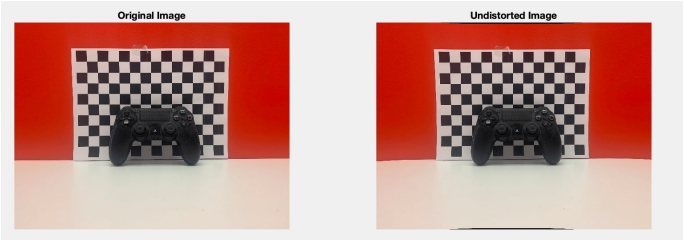

Figure 3: Original and Undistorted Image

Distortion Type	x-axis	y-axis
Tangential distortion (pixels)	0.0012	0.0003
Radial distortion (pixels)	0.1839	-0.5391

Table 2: Distortion Table

3. Camera calibration

The data represents standard errors of the estimated camera parameters at the time the photos were taken. Intrinsic parameters describe internal characteristics such as focal length and principal point, while extrinsic parameters define the camera's position and orientation relative to the scene. Rotation vectors represent angular orientation, and translation vectors describe displacement along the x, y, and z axes.

The MATLAB script calibrates the camera by estimating intrinsic and extrinsic parameters of a pinhole model using chequerboard images. `detectCheckerboardPoints()` automatically detects corner points and board dimensions. `generateCheckerboardPoints()` defines the real-world coordinates of these corners based on known square sizes.

Using these 2D-3D correspondences, `estimateCameraParameters()` computes the intrinsic parameters—focal length, principal point, and distortion coefficients—and the extrinsic parameters. Calibration accuracy is assessed by plotting reprojection errors with `showReprojectionErrors()`, and extrinsics are visualized using `showExtrinsics()`. Finally, the effect of lens distortion is demonstrated by undistorting a calibration image, as shown in fig. 3.

The resulting intrinsic parameters are shown in matrix K below. Additionally, the distortion amounts are shown in 2 Additionally, extrinsic parameters for each image areshown in tables 3 and 4. These parameters yield a calibrated camera model suitable for metric measurements and distortion correction in further imaging tasks.

Fig. 3 illustrates an estimation of the image after applying lens distortion correction, transforming the originally distorted view into a more geometrically accurate representation. By comparing the undistorted image to the original, the effectiveness of the calibration process can be assessed, verifying the accuracy of the intrinsic parameters.

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4606.09 & 28.91 & 2083.85 \\ 0 & 4598.53 & 1592.73 \\ 0 & 0 & 1 \end{bmatrix}$$

In the intrinsic matrix K, f_x and f_y are the focal lengths, s is the skew and c_x and c_y are the principal points.

Fig. 4 visualises the camera centric view of each image, which define the position and orientation of multiple cameras in 3D space. Each camera pose is represented by a coloured plane and wireframe box. This illustrates the camera's field of view and alignment relative to the world coordinate system. The X, Y and Z axes indicate spatial positioning, while the numbered labels correspond to different camera positions during calibration. The plot is used to verify the accuracy of the estimated rotation and translation vectors, ensuring proper transformation between local and world coordinates.

x-axis (pixels)	y-axis (pixels)	z-axis (pixels)
0.2035 ± 0.0031	-0.0373 ± 0.0040	-0.2710 ± 0.0003
0.2918 ± 0.0028	0.3102 ± 0.0038	-0.2600 ± 0.0007
0.2150 ± 0.0024	-0.7144 ± 0.0036	-0.3033 ± 0.0010
0.5324 ± 0.0026	-0.0659 ± 0.0038	-0.2254 ± 0.0009
0.5745 ± 0.0027	0.4317 ± 0.0036	-0.1595 ± 0.0011

Table 3: Extrinsic Parameters: Rotation Vectors

x-axis (pixels)	y-axis (pixels)	z-axis (pixels)
-288.2716 ± 3.3466	-160.5518 ± 2.5446	938.1291 ± 2.6357
-346.2158 ± 4.0874	-182.3540 ± 3.0835	1150.7297 ± 2.8144
-266.8685 ± 2.5588	-137.0197 ± 1.9385	712.8194 ± 2.0935
-312.2387 ± 3.8909	-148.9961 ± 2.9537	1093.5681 ± 3.0754
-317.1356 ± 4.6044	-172.4788 ± 3.4446	1299.7005 ± 2.9577

Table 4: Translation Vectors

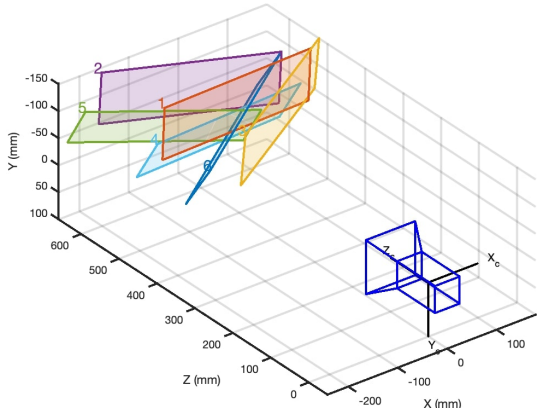

Figure 4: Camera View

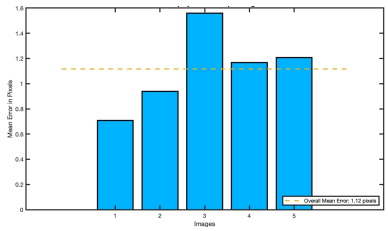

Figure 5: Mean Error Re-projection

Figure 5 illustrates the mean re-projection error per image, a key metric in camera calibration that quantifies how accurate the estimated camera parameters. The y-axis represents the mean error in pixels, while the x-axis corresponds to different images used in the calibration process. The dashed yellow line in the plot signifies the overall mean re-projection error of 1.12 pixels. Since the projection error is relatively small, it means that the calibration process was successful. If it was any higher, it could indicate calibration issues such as poor feature detection, incorrect correspondences, or lens distortions.

4. Transformation estimation

4.1. Homography Matrix

In this section, we estimate the homography matrix between a pair of images taken from the datasets. The homography transformation describes the relationship between two planes in projective space, allowing us to map points from one image to another.

To estimate the homography matrix first the images are converted to greyscale, keypoints are detected and described using the KAZE algorithm using the `detectKAZEFeatures()` function. Feature matches are found via Nearest Neighbour matching with a ratio test to discard ambiguous pairs. Then `estimateGeometricTransform2D` is then used to compute a robust homography matrix, which is visualised to assess accuracy

The computed homography matrix, as derived from our implementation, is as follows:

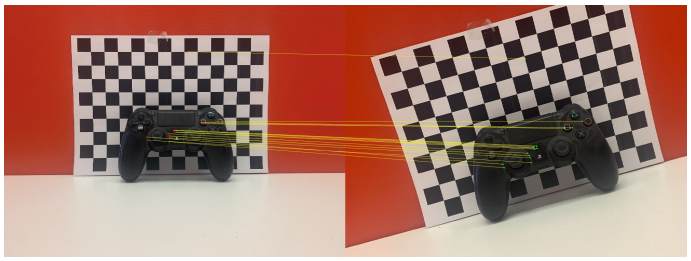

Figure 6: Keypoint correspondences between the two images using the SURF feature detector.

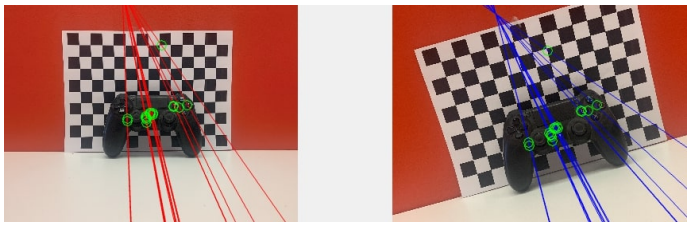

Figure 7: Epipolar lines and matched keypoints overlaid on both images.

$$H = \begin{bmatrix} 0.8717 & -0.2711 & -0.0000 \\ -0.1716 & 0.5799 & -0.0002 \\ 46.4600 & 886.1884 & 1.0000 \end{bmatrix} \quad (1)$$

This transformation matrix encapsulates the projective relationship between the two images, allowing accurate mapping of points from one image to the other.

Figure 6 presents the feature correspondences between the two images, highlighting correctly matched keypoints and the transformations captured by the homography matrix. The red and green markers indicate keypoints from the two images, with white lines representing the correspondences.

This visualisation highlights the effectiveness of RANSAC-based estimation in filtering out outliers and achieving a robust homography transformation. The results show that the estimated homography accurately maps points between images under perspective changes from zoom and slight rotations. RANSAC was crucial in reducing the impact of erroneous correspondences. Overall, the experiment successfully estimated the homography matrix and visualised keypoint correspondences, demonstrating the strength of feature-based homography estimation.

4.2. Fundamental Matrix

In this task, the fundamental matrix $\mathbf{F}$ was computed using matched feature points between the two input images. The process began by detecting KAZ feature points on both images. Feature descriptors were then extracted , followed by matching the features using the `matchFeatures()` function with a Lowe's ratio test threshold of 0.7 to filter out weak matches.

The robust estimation of the fundamental matrix $\mathbf{F}$ was achieved using the RANSAC algorithm and the `estimateFundamentalMatrix()` function, which simultaneously identifies inlier correspondences and reduces the impact of outliers. The estimated fundamental matrix is shown below:

$$\mathbf{F} = \begin{bmatrix} 0.0000 & -0.0000 & 0.0007 \\ 0.0000 & 0.0000 & -0.0013 \\ -0.0013 & 0.0009 & 1.0000 \end{bmatrix}$$

The matrix encapsulates the intrinsic projective geometry between the two views, relating corresponding points across the image pair. To visualise the quality of this estimation, the epipolar lines were computed, using the `epipolarLine()` function and the fundamental matrix, and are overlaid on both images along with the matched keypoints in figure 7.

The plot demonstrates that the epipolar lines accurately align with the matched keypoints, validating the estimated fundamental matrix. Each corresponding keypoint in one image correctly maps to an epipolar line in the other, as expected.

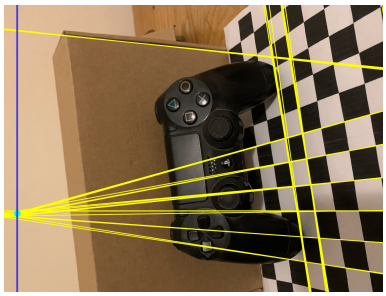

Figure 8: Detected lines and computed vanishing point. Yellow lines represent the extrapolated Hough lines converging to the vanishing point (cyan). The vertical blue line represents the horizon.

Overall, the results confirm that the fundamental matrix correctly captures the epipolar geometry between the two views, with the plotted epipolar lines passing through their corresponding keypoints.

Next, we computed the vanishing point and horizon in one of our images. This process highlights the scene's geometric properties and helps in understanding the 3D layout.

The detection was performed by first applying edge detection to the greyscale version of the image using the Canny method with the `edge()` function. Subsequently, the Hough Transform was computed with the `Hough()` function and then, `houghpeaks()` was used to detect Hough peaks and `houghlines()` to detect prominent lines in the scene. We then extracted the strongest line candidates and extrapolated them to identify where they converge in the image plane.

The vanishing point was then estimated by calculating the intersection of these extrapolated lines. A vertical reference line was drawn through the computed vanishing point to represent the horizon in the image space.

Figure 8 presents the detected lines (in yellow), the computed vanishing point (cyan dot), and the vertical horizon line (blue) overlaid on the scene. The lines clearly converge toward the vanishing point located at the point in image space.

The result confirms that the scene's parallel lines in 3D space converge correctly in the image plane, showcasing the expected perspective geometry. The chequerboard grid significantly helped in reliable line detection. This vanishing point serves as a reference for further 3D understanding of the scene and is essential for tasks such as depth estimation and camera calibration.

4.3. Outlier Tolerance

To evaluate the outlier tolerance of both the Homography and Fundamental matrix estimations, we designed an experiment that progressively injected synthetic outliers into the matched keypoints. Starting from the original correct correspondences, we randomly selected an increasing number of matches and corrupted them by reassigning the correspondence points to random positions.

For each level of outliers, the homography and fundamental matrices were calculated using the same methods as before. Both were configured with a high confidence level (99%) and a maximum of 2000 iterations. We recorded the maximum number of outliers tolerated before the algorithm could no longer find the minimum required inliers (4 for homography, 8 for fundamental matrix) to compute a valid model. The experiment showed a significant difference in the robustness of the two estimations. Over ten rounds the mean results were: **Homography estimation** tolerated up to **480.93 outliers** out of 1162 matches, **Fundamental matrix estimation** only tolerated **17.32 outliers** out of 1162 matches.

This outcome is expected due to the nature of the geometric constraints each method models. Homography operates under a planar scene assumption or pure rotational camera motion, making it less sensitive to outliers when the scene conditions hold. On the other hand, estimating the fundamental matrix involves epipolar geometry, which is sensitive to 3D scene structure and perspective changes. Even a small number of incorrect matches can significantly reduce accuracy. These results highlight the higher robustness of homography estimation to outliers in planar or near-planar scenes. In contrast, the fundamental matrix estimation is less tolerant due to its reliance on precise 3D point correspondences. This reinforces the importance of accurate feature matching and robust outlier rejection.

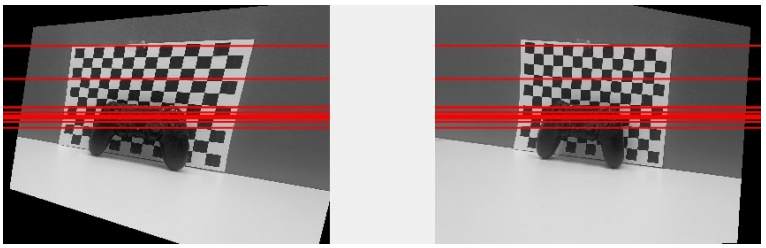

Figure 9: Rectified stereo image pair with epipolar lines overlaid. The lines are now horizontal, ensuring that corresponding points lie along the same row in both images.

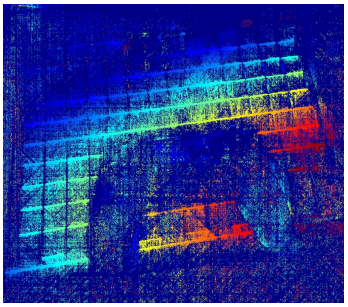

Figure 10: Depth map estimated from the stereo image pair. Closer objects are represented in cooler colors, while distant regions appear warmer.

5. 3D geometry

5.1. Stereo Rectification

Stereo rectification is essential for 3D reconstruction as it aligns epipolar lines horizontally, reducing stereo correspondence to a 1D search. Using the fundamental matrix from Task 4, and the pair of stereo images shown in fig. 17 in the appendix stereo rectification was performed.

The process involved using the Fundamental matrix calculated previously to remove outliers, and computing projective transformations to align the images using the `estimateUncalibratedRectification()`, `projective2d` and `imwarp()` functions. The warped images were then visualised with overlaid epipolar lines to verify alignment.

As shown in fig. 9, the horizontal epipolar lines confirm successful rectification, enabling easier disparity estimation for depth calculations.

The rectified image pair preserves scene geometry while aligning epipolar lines, enabling accurate depth estimation. The uncalibrated rectification matrices maintain spatial consistency between images. Results show effective alignment, with parallel epipolar lines confirming successful rectification. Minor misalignments may remain due to lens distortion or fundamental matrix inaccuracies.

5.2. Depth Map

To estimate the depth map, stereo rectified images obtained in Task 5.1 were used. The process involved computing disparity between corresponding points in the left and right images, as disparity is inversely proportional to depth. The following steps were performed:

1. **Stereo Rectification:** The rectified image pair from task 5.1 was used.
2. **Disparity Calculation:** A Semi-Global Matching (SGM) algorithm was used to compute the disparity map, with a disparity range of 0 to 64 pixels using the function `disparitySGM()`.
3. **Depth Map Visualization:** The computed disparity values were converted into a colour-coded depth map using a jet colourmap.

The estimated depth map is shown in Fig. 10. Warmer colours (red/yellow) indicate objects farther from the camera, while cooler colours (blue) represent closer regions. The structured nature of the scene is visible, with depth variation captured across the different objects.

The depth estimation successfully highlights the structure of the scene, with clear depth gradients visible and the outline of the controller. However, some areas exhibit noise and incorrect depth estimates, likely due to occlusions, textureless regions, or mismatched correspondences.

A. Additional Figures

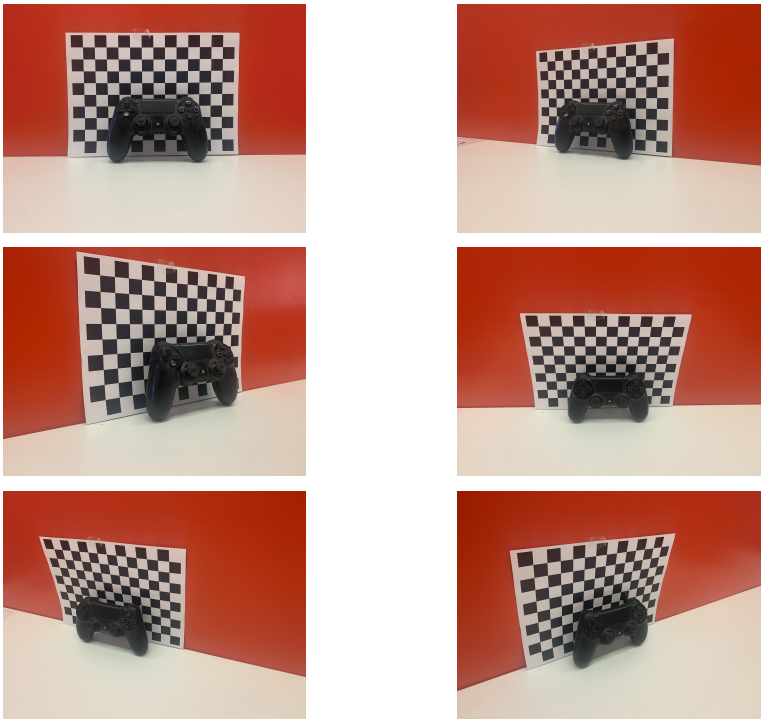

Figure 11: FD Object Images

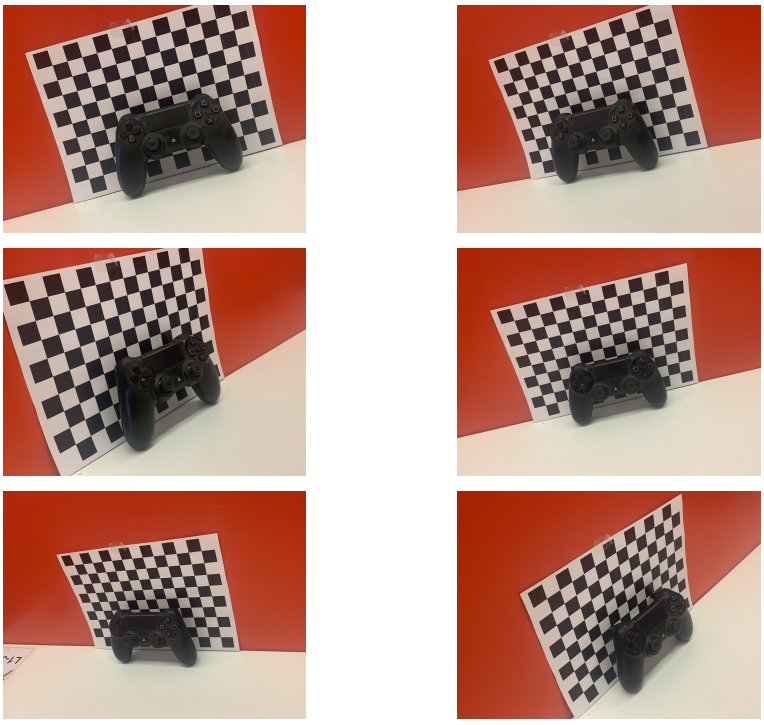

Figure 12: HG Object Images

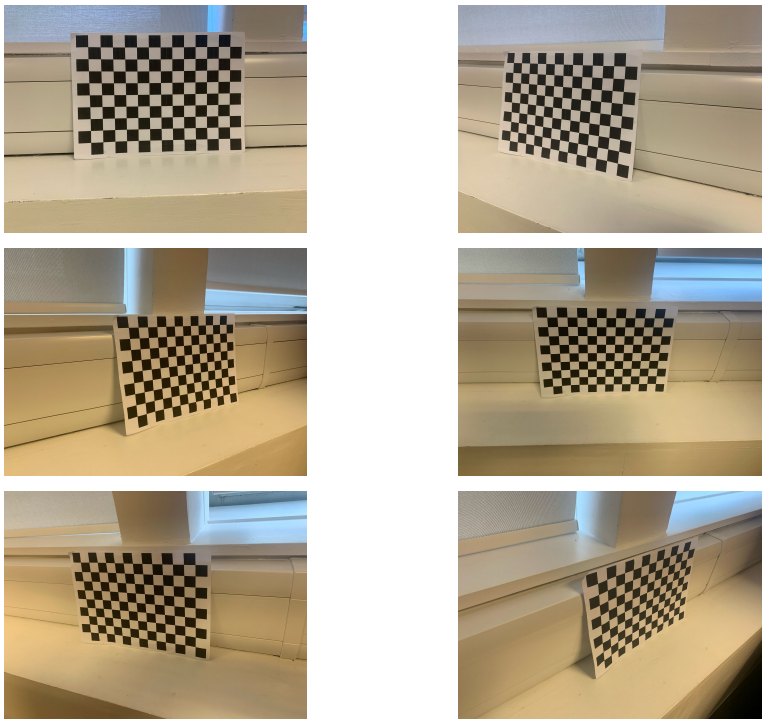

Figure 13: FD Grid Images

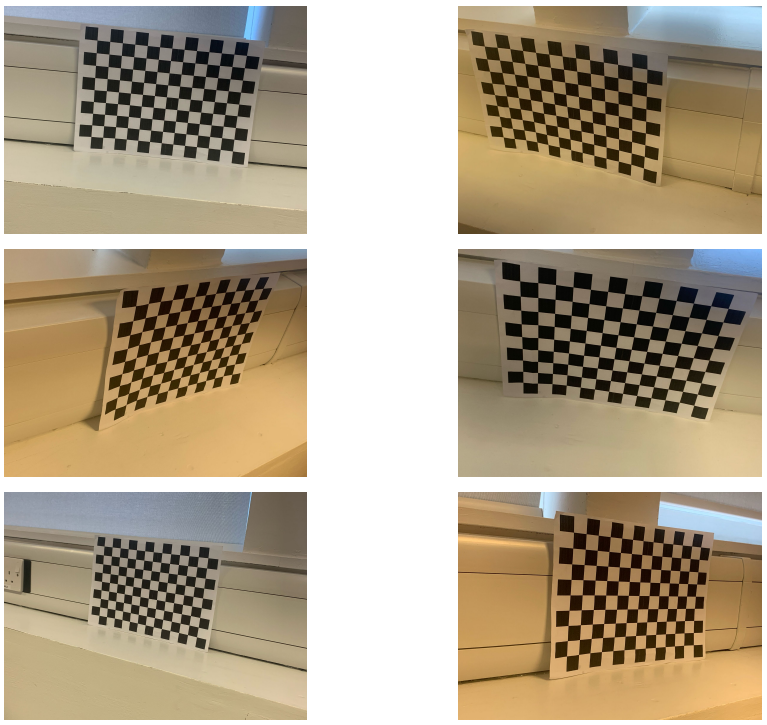

Figure 14: HG Grid Images

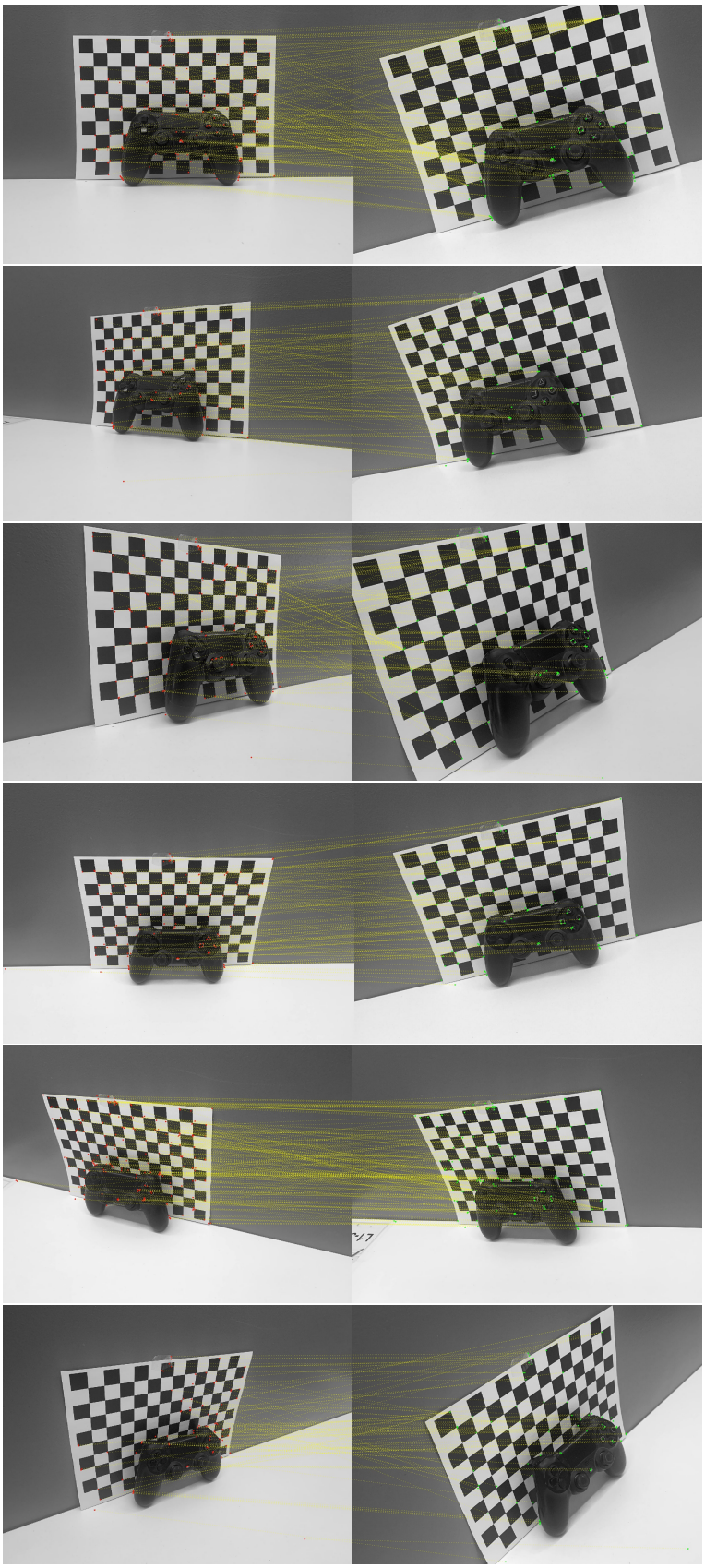

Figure 15: All FD and HG Images pairs with mapped correspondences

Figure 16: Chequerboard detection for camera calibration

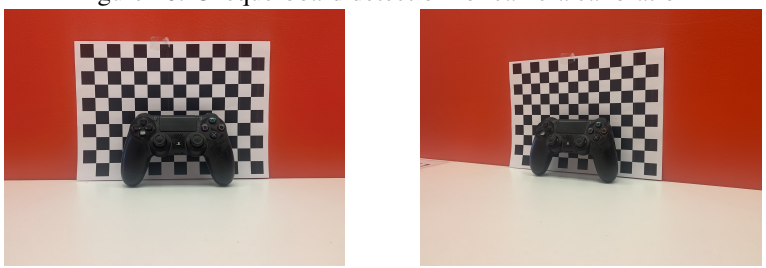

Figure 17: Original Stereo Images

B. Matlab Code

```
function find_correspondence(fd_images, hg_images)
    % FIND_CORRESPONDENCE Matches features between FD and HG image sets
    % fd_images: cell array of file paths to FD images
    % hg_images: cell array of file paths to HG images

    % Ensure the sets have the same number of images
    numImages = min(length(fd_images), length(hg_images));

    for i = 1:numImages
        % Read images
        imgFD = imread(fd_images{i});
        imgHG = imread(hg_images{i});

        % Convert to grayscale if necessary
        if size(imgFD, 3) == 3
            imgFD = rgb2gray(imgFD);
        end
        if size(imgHG, 3) == 3
            imgHG = rgb2gray(imgHG);
        end

        % Detect features and extract descriptors
        pointsFD = detectSURFFeatures(imgFD);
        pointsHG = detectSURFFeatures(imgHG);

        % pointsFD = detectKAZEFeatures(imgFD);
        % pointsHG = detectKAZEFeatures(imgHG);

        [featuresFD, validPointsFD] = extractFeatures(imgFD, pointsFD);
        [featuresHG, validPointsHG] = extractFeatures(imgHG, pointsHG);

        % Match features
        indexPairs = matchFeatures(featuresFD, featuresHG);

        % Get matching points
        matchedPointsFD = validPointsFD(indexPairs(:,1), :);
        matchedPointsHG = validPointsHG(indexPairs(:,2), :);

        % Display matches without lines
        figure;
        showMatchedFeatures(imgFD, imgHG, matchedPointsFD, matchedPointsHG, 'montage', 'PlotOptions', {'r.','g.','w:'});
        title(['Matched Features between FD and HG images - Pair ', num2str(i)]);
    end
end
```

Listing 1: Code for Finding Correspondances

```

    [imagePoints, boardSize, imagesUsed] = detectCheckerboardPoints(imageFileNames);
    fprintf('Detected a %d x %d checkerboard pattern.\n', boardSize(1), boardSize(2));

    % Optionally, display the detected points on one of the images.
    I = imread(imageFileNames{find(imagesUsed,1)});
    figure; imshow(I); hold on;
    plot(imagePoints(:,1), imagePoints(:,2), 'ro');
    title('Detected Checkerboard Points');

    %% 3. Define World Coordinates for the Checkerboard Corners
    % Specify the size of one square in world units (e.g., millimeters).
    squareSize = 25;  % Change to your actual square size
    worldPoints = generateCheckerboardPoints(boardSize, squareSize);

    %% 4. Estimate Camera Parameters
    % Read one image to get the image size.
    I = imread(imageFileNames{find(imagesUsed,1)});
    imageSize = [size(I,1), size(I,2)];

    % Estimate the intrinsic and extrinsic camera parameters.
    cameraParams = estimateCameraParameters(imagePoints, worldPoints, ...
        'ImageSize', imageSize);
```

Listing 2: Code to Callibrate Camera

```

    img1 = imread('Photos/Grid/FD/1.jpg');
    img2 = imread('Photos/Grid/HG/1.jpg');

    gray1 = rgb2gray(img1);
    gray2 = rgb2gray(img2);

    % Detect feature points
    points1 = detectKAZEFeatures(gray1);
    points2 = detectKAZEFeatures(gray2);

    % Extract descriptors
    [features1, validPoints1] = extractFeatures(gray1, points1);
    [features2, validPoints2] = extractFeatures(gray2, points2);

    % Match features
    indexPairs = matchFeatures(features1, features2, 'MaxRatio', 0.7, 'Unique', true);
    matchedPoints1 = validPoints1(indexPairs(:,1), :);
    matchedPoints2 = validPoints2(indexPairs(:,2), :);

    % Compute Homography using RANSAC
    [H, inliers] = estimateGeometricTransform2D(matchedPoints1, matchedPoints2, 'projective', 'MaxNumTrials', 4000);
```

Listing 3: Code to find Homography Matrix

```
    points1 = detectKAZEFeatures(gray1);
    points2 = detectKAZEFeatures(gray2);

    % Extract features
```

```
    [features1, validPoints1] = extractFeatures(
        gray1, points1);
    [features2, validPoints2] = extractFeatures(
        gray2, points2);

    % Match features
    indexPairs = matchFeatures(features1, features2,
         'MaxRatio', 0.7, 'Unique', true);
    matchedPoints1 = validPoints1(indexPairs(:,1),
        :);
    matchedPoints2 = validPoints2(indexPairs(:,2),
        :);

    % Compute Fundamental Matrix using RANSAC
    [F, inliers] = estimateFundamentalMatrix(
        matchedPoints1, matchedPoints2, 'Method', '
        RANSAC');
```

Listing 4: Code to find Fundamental Matrix

```

    % Prepare test points for Homography and
        Fundamental matrix
    numMatches = size(matchedPoints1, 1);
    maxOutliersAllowed_H = 0;
    maxOutliersAllowed_F = 0;

    % Convert matched points to locations for
        testing
    pts1 = matchedPoints1.Location;
    pts2 = matchedPoints2.Location;

    % HOMOGRAPHY Outlier Tolerance Test
    for numOutliers = 0:numMatches
        % Inject synthetic outliers by randomly
            shuffling some matches
        idx = randperm(numMatches);
        corrupted_pts2 = pts2;
        if numOutliers > 0
            corrupt_idx = idx(1:numOutliers);
            corrupted_pts2(corrupt_idx, :) =
                corrupted_pts2(randperm(numOutliers),
                 :);
        end

        try
            % Attempt to estimate Homography with
                corrupted points
            [~, inlierIdx] =
                estimateGeometricTransform2D(pts1,
                corrupted_pts2, 'projective', '
                MaxNumTrials', 2000, 'Confidence',
                99);
            % If enough inliers, the method still
                tolerates these outliers
            if sum(inlierIdx) >= 4  % Minimum
                inliers for homography
                maxOutliersAllowed_H = numOutliers;
            else
                break; % Cannot tolerate this many
                    outliers
            end
        catch
            break;
        end
    end

    % FUNDAMENTAL MATRIX Outlier Tolerance Test
    for numOutliers = 0:numMatches
        idx = randperm(numMatches);
        corrupted_pts2 = pts2;
        if numOutliers > 0
            corrupt_idx = idx(1:numOutliers);
            corrupted_pts2(corrupt_idx, :) =
                corrupted_pts2(randperm(numOutliers),
                 :);
        end

        try
            % Attempt to estimate Fundamental Matrix
                 with corrupted points
            [~, inlierIdx] =
                estimateFundamentalMatrix(pts1,
                corrupted_pts2, 'Method', 'RANSAC', '
                NumTrials', 2000, 'Confidence', 99);
            if sum(inlierIdx) >= 8  % Minimum
                inliers for 8-point algorithm
                maxOutliersAllowed_F = numOutliers;
            else
                break;
            end
        catch
            break;
        end
    end

    fprintf('Maximum tolerated outliers for
        Homography estimation: %d out of %d matches\n
        ', maxOutliersAllowed_H, numMatches);
    fprintf('Maximum tolerated outliers for
        Fundamental matrix estimation: %d out of %d
        matches\n', maxOutliersAllowed_F, numMatches)
        ;
```

Listing 5: Code to test Outlier Tolerance

```
    img = imread('Photos/vanishing.jpg');
    padding = 0;
    img_padded = padarray(img, [padding, padding],
        255);

    % Convert to grayscale and threshold
    gray_img = rgb2gray(img_padded);
    binary_img = imbinarize(gray_img, 80/255);

    % Edge detection using Canny
    edge_img = edge(binary_img, 'canny', 0.8);

    % Compute Hough Transform
    [H, theta, rho] = hough(edge_img);

    % Detect peaks in the Hough transform
    num_peaks = 10;
    peak_thresh = ceil(0.3 * max(H(:)));
    peaks = houghpeaks(H, num_peaks, 'Threshold',
        peak_thresh);

    % Extract lines from the Hough transform
    min_length = 50;
    lines = houghlines(edge_img, theta, rho, peaks,
        'FillGap', 5, 'MinLength', min_length);

    % Plot the original image
    figure, imshow(img_padded), hold on
```

```
    title('Detected Lines and Vanishing Point');

    % Helper function to plot lines
    plot_lines(lines, size(edge_img));

    % Plot fixed vertical line and vanishing point marker
    x_vanish = 285.091;
    y_vanish = 1973.39;
    xline(x_vanish, 'LineWidth', 4, 'Color', 'b');
    plot(x_vanish, y_vanish, '.', 'MarkerSize', 50, 'Color', 'c');

    hold off;

    %% Function to plot lines with extrapolation
    function plot_lines(lines, img_size)
        % Group lines based on index ranges if needed
        total_lines = length(lines);
        for k = 1:total_lines
            % Extract the endpoints
            pt1 = lines(k).point1;
            pt2 = lines(k).point2;

            % Calculate slope and intercept
            if pt2(1) ~= pt1(1)
                m = (pt2(2) - pt1(2)) / (pt2(1) - pt1(1));
                b = pt1(2) - m * pt1(1);
                % Extrapolate the line across the image width
                x_vals = linspace(1, img_size(2), 500);
                y_vals = m * x_vals + b;
            else
                % Vertical line case
                x_vals = pt1(1) * ones(1, 2);
                y_vals = [1, img_size(1)];
            end
            plot(x_vals, y_vals, 'y-', 'LineWidth', 2);
        end
    end
```

Listing 6: Code to find Horizon & Vanishing Points

```

    leftImg = imread('Photos/Grid/FD/1.jpg');
    rightImg = imread('Photos/Grid/FD/2.jpg');

    % Convert to grayscale (if necessary)
    if size(leftImg, 3) == 3
        leftImgGray = rgb2gray(leftImg);
        rightImgGray = rgb2gray(rightImg);
    else
        leftImgGray = leftImg;
        rightImgGray = rightImg;
    end

    % Detect and extract features
    points1 = detectSURFFeatures(leftImgGray);
    points2 = detectSURFFeatures(rightImgGray);
    [features1, validPoints1] = extractFeatures(leftImgGray, points1);
    [features2, validPoints2] = extractFeatures(rightImgGray, points2);

    % Match features
    indexPairs = matchFeatures(features1, features2, 'Unique', true);
    matchedPoints1 = validPoints1(indexPairs(:,1));
    matchedPoints2 = validPoints2(indexPairs(:,2));

    % Estimate fundamental matrix
    [fMatrix, inliers] = estimateFundamentalMatrix(matchedPoints1, matchedPoints2, ...
        'Method', 'RANSAC', 'NumTrials', 2000, 'DistanceThreshold', 1.5);

    % Select inlier matches
    inlierPoints1 = matchedPoints1(inliers);
    inlierPoints2 = matchedPoints2(inliers);

    % Stereo Rectification (uncalibrated)
    imageSize = size(leftImgGray);
    [t1, t2] = estimateUncalibratedRectification(fMatrix, ...
        inlierPoints1.Location, inlierPoints2.Location, imageSize);

    % Convert to projective transformations
    tform1 = projective2d(t1);
    tform2 = projective2d(t2);

    % Warp images using projective transformation
    leftRectified = imwarp(leftImgGray, tform1, 'OutputView', imref2d(imageSize));
    rightRectified = imwarp(rightImgGray, tform2, 'OutputView', imref2d(imageSize));

    % Rectify the inlier points as well
    inlierPoints1Rect = transformPointsForward(tform1, inlierPoints1.Location);
    inlierPoints2Rect = transformPointsForward(tform2, inlierPoints2.Location);
```

Listing 7: Code for Stereo Rectification

```
    leftImg = imread('Photos/Grid/HG/1.jpg');
    rightImg = imread('Photos/Grid/FD/1.jpg');

    % Convert to grayscale (if necessary)
    if size(leftImg, 3) == 3
        leftImgGray = rgb2gray(leftImg);
        rightImgGray = rgb2gray(rightImg);
    else
        leftImgGray = leftImg;
        rightImgGray = rightImg;
    end

    % Detect and extract features
    points1 = detectSURFFeatures(leftImgGray);
    points2 = detectSURFFeatures(rightImgGray);
    [features1, validPoints1] = extractFeatures(leftImgGray, points1);
    [features2, validPoints2] = extractFeatures(rightImgGray, points2);

    % Match features
    indexPairs = matchFeatures(features1, features2, 'Unique', true);
    matchedPoints1 = validPoints1(indexPairs(:,1));
    matchedPoints2 = validPoints2(indexPairs(:,2));
```

```
23
24      % Estimate fundamental matrix
25      [fMatrix, inliers] = estimateFundamentalMatrix(
            matchedPoints1, matchedPoints2, ...
26          'Method', 'RANSAC', 'NumTrials', 2000, '
                DistanceThreshold', 1.5);
27
28      % Select inlier matches
29      inlierPoints1 = matchedPoints1(inliers);
30      inlierPoints2 = matchedPoints2(inliers);
31
32      % Stereo Rectification
33      imageSize = size(leftImgGray);
34      [t1, t2] = estimateUncalibratedRectification(
            fMatrix, inlierPoints1.Location,
            inlierPoints2.Location, imageSize);
35
36      % Convert to projective transformations
37      tform1 = projective2d(t1);
38      tform2 = projective2d(t2);
39
40      % Warp images using projective transformation
41      leftRectified = imwarp(leftImgGray, tform1, '
            OutputView', imref2d(imageSize));
42      rightRectified = imwarp(rightImgGray, tform2, '
            OutputView', imref2d(imageSize));
43
44      % Compute disparity map
45      disparityRange = [0 64]; % Adjust range based on
             scene
46      disparityMap = disparitySGM(leftRectified,
            rightRectified, 'DisparityRange',
            disparityRange);
```

Listing 8: Code for Depth Estimation