sgelam-project3

July 8, 2024

1 Problem 1

[ ]:
```
from google.colab import drive

# Mount Google Drive
drive.mount("/content/drive", force_remount=True)
```

```
Mounted at /content/drive
```

[ ]:
```
import numpy as np
import cv2
import glob
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
import os

# Define the directory containing the images
image_dir = '/content/drive/My Drive/checker board/'


# Get the paths of all image files in the directory
image_paths = glob.glob(os.path.join(image_dir, '*.tiff'))

# Create a directory to save extracted frames
frame_dir = 'frames_2'
if not os.path.exists(frame_dir):
    os.makedirs(frame_dir)

# Extract frames from the images
frame_count = 0
for image_path in image_paths:
    # Read the image
    frame = cv2.imread(image_path)

    # Display the image
    # cv2_imshow(frame)

    # Save the image as a frame
```

```
    frame_count += 1
    frame_name = os.path.join(frame_dir, f'frame_{frame_count:04d}.jpg')
    cv2.imwrite(frame_name, frame)

print("Total frames extracted:", frame_count)
```

```
Total frames extracted: 50
```

The above code segment serves to systematically extract frames from a directory containing TIFF image files. It initiates by specifying the directory path where the images are stored and retrieves all TIFF image file paths within this directory using the glob module. Subsequently, it establishes a new directory labeled "frames_2" to house the extracted frames, should it not already exist. The script then iterates through each image file path, reading the image with OpenCV's cv2.imread function, incrementing a counter to track the number of frames processed, and saving each image as a frame within the designated directory utilizing cv2.imwrite. Upon completion, the total count of extracted frames is printed.

[ ]:
```
# Get a list of extracted frame paths
frame_paths = glob.glob(os.path.join(frame_dir, '*.jpg'))
```

[ ]:
```
# Initialize lists to store object points, image points, and reprojection errors
objpoints = []  # 3D points in real world space
imgpoints = []  # 2D points in image plane
reprojection_errors = []  # List to store reprojection errors

# Define the chessboard pattern size
pattern_size = (14, 14)

# Define the dimensions of the chessboard squares (in meters)
square_size_x = 0.009  # Width of each square along the x-axis (e.g., 9mm)
square_size_y = 0.009  # Width of each square along the y-axis (e.g., 9mm)

# Define the object points (3D coordinates of chessboard corners in real world␣
 ↪space)
objp = np.zeros((np.prod(pattern_size), 3), dtype=np.float32)
objp[:, :2] = np.mgrid[0:pattern_size[0], 0:pattern_size[1]].T.reshape(-1, 2) *␣
 ↪np.array([square_size_x, square_size_y])
# print(objp)
```

This code snippet initializes essential data structures and parameters for camera calibration. It creates three lists to store object points, image points, and reprojection errors. The chessboard pattern size is defined as 14x14 corners, with each square's dimensions set to 0.009 meters along both the x and y axes. Object points (objp) are generated to represent the 3D coordinates of chessboard corners in real-world space, ensuring accurate calibration for subsequent image processing tasks.

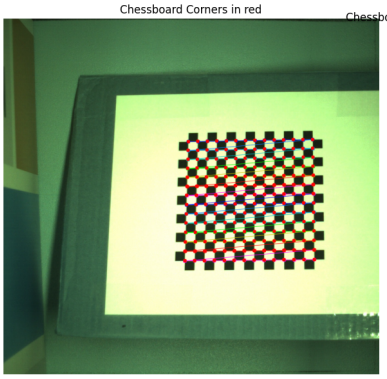
[ ]:
```
# Define the number of columns for the subplot grid
num_columns = 2
```

```
# Initialize a figure and axes for the subplots
fig, axes = plt.subplots(nrows=5, ncols=num_columns, figsize=(10*num_columns,␣
 ↪6*5))

# Initialize a count to keep track of the number of images processed
count = 0

# Loop through each extracted frame to find chessboard corners
for i, frame_path in enumerate(frame_paths):
    img = cv2.imread(frame_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, pattern_size, None)

    # If corners are found, add object points and image points
    if ret:
        objpoints.append(objp)
        imgpoints.append(corners)

        # Draw the corners on the image
        img_with_corners = cv2.drawChessboardCorners(img.copy(), pattern_size,␣
 ↪corners, ret)
        image = img_with_corners

        # Calibrate the camera
        ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,␣
 ↪imgpoints, gray.shape[::-1], None, None)

        # Project object points onto the image to get reprojection
        imgpoints2, _ = cv2.projectPoints(objp, rvecs[-1], tvecs[-1], mtx, dist)

        # Compute reprojection error
        error = cv2.norm(corners, imgpoints2, cv2.NORM_L2) / len(imgpoints2)
        reprojection_errors.append(error)

        # Increment the count of processed images
        count += 1

        if count <= 5:
            # Draw the reprojection points
            for point in imgpoints2:
                cv2.circle(img_with_corners, tuple(point[0].astype(int)), 5,␣
 ↪(255, 0, 0), -1)

            # Display the images with chessboard corners and reprojection␣
 ↪points side by side
```

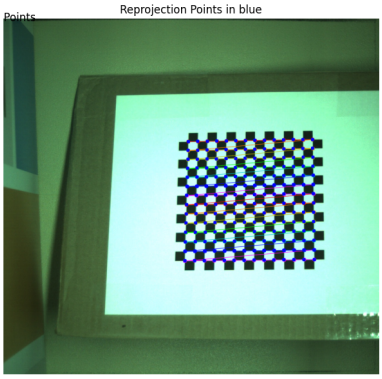
```
            axes[count-1, 0].imshow(image)
            axes[count-1, 0].set_title('Chessboard Corners in red')

            axes[count-1, 1].imshow(cv2.cvtColor(img_with_corners, cv2.
 ↪COLOR_BGR2RGB))
            axes[count-1, 1].set_title('Reprojection Points in blue')

# Set the title for the entire figure
fig.suptitle('Chessboard Corners and Reprojection Points')

# Hide the axes for all subplots
for ax_row in axes:
    for ax in ax_row:
        ax.axis('off')

# Adjust layout to prevent overlap of titles
plt.tight_layout()

# Show the figure
plt.show()
```

Chessboard Corners and Reprojection Points

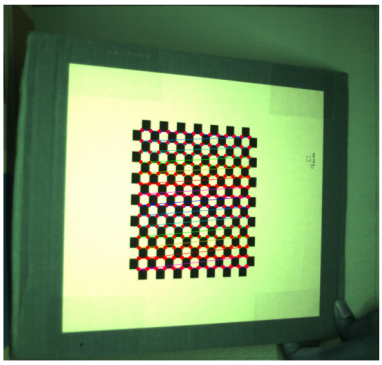
Chessboard Corners in red



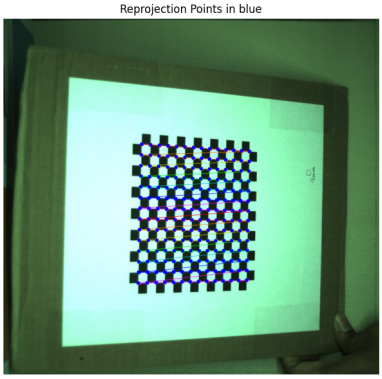
Reprojection Points in blue



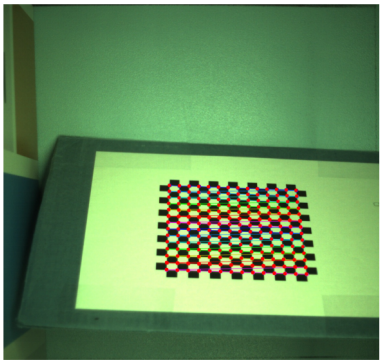
Chessboard Corners in red



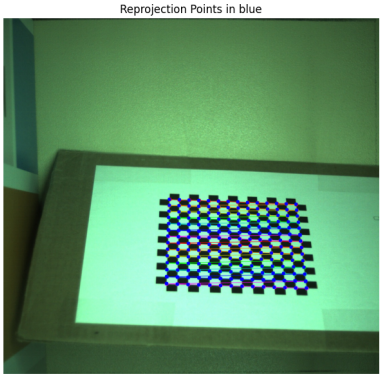
Reprojection Points in blue



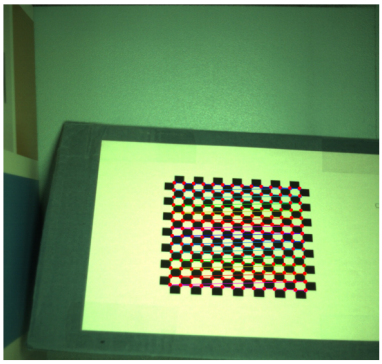
Chessboard Corners in red



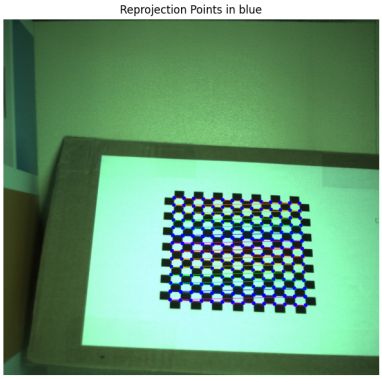
Reprojection Points in blue



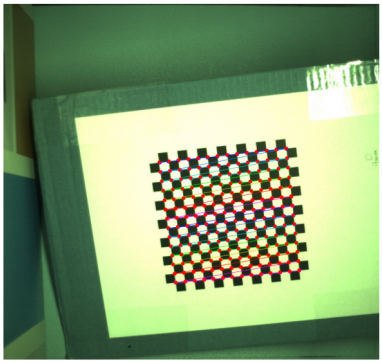
Chessboard Corners in red



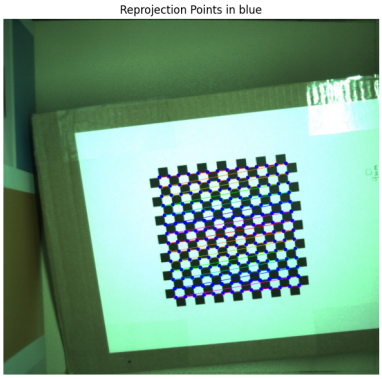
Reprojection Points in blue



Chessboard Corners in red



Reprojection Points in blue



It begins by iterating through each extracted frame from the image paths. For each frame, it converts the image to grayscale and then applies the cv2.findChessboardCorners() function to detect corners based on the specified pattern size.

The cv2.findChessboardCorners() function employs various technical processes to detect the internal corners of a chessboard pattern in an image. It analyzes image gradients to identify regions of sharp intensity changes, utilizes corner response functions like Harris or Shi-Tomasi detectors to score potential corner points based on gradient variations, and applies local maximum suppression to select the most significant corners. Pattern matching is then performed to validate corner positions against the expected pattern size, with iterative refinement techniques enhancing accuracy.

Step-by-step explanation of each stage of the calibration pipeline:

Detection of Calibration Pattern: In this stage, the calibration pattern's keypoints or corners are detected in each acquired image. This is typically done using feature detection algorithms like SIFT. These algorithms identify distinctive points or regions in the image that can be reliably matched between different images.

Matching Keypoints: Once the keypoints are detected in each image, the next step is to match the keypoints between pairs of images. This process establishes correspondences between the same physical points in different images, allowing for accurate calibration.

Estimation of Camera Parameters: Using the matched keypoints, the camera's intrinsic and extrinsic parameters are estimated. The intrinsic parameters include focal length, principal point, and lens distortion coefficients, while the extrinsic parameters describe the camera's position and orientation in the scene.

The cv2.calibrateCamera() function estimates the intrinsic camera parameters (focal length, optical center) and distortion coefficients by analyzing detected 3D object points and their corresponding 2D image points. It optimizes these parameters to minimize the difference between observed and projected image points, providing the camera matrix (mtx), distortion coefficients (dist), rotation vectors (rvecs), and translation vectors (tvecs).

Refinement of Parameters: After the initial estimation, an optimization process may be employed to refine the camera parameters further. This optimization aims to minimize the reprojection error, which is the discrepancy between the observed keypoints and their corresponding projected points calculated using the estimated parameters.

Validation and Evaluation: Once the calibration is complete, the accuracy of the estimated parameters is validated. This validation may involve assessing the reprojection error, comparing the projected keypoints with manually annotated ground truth points.

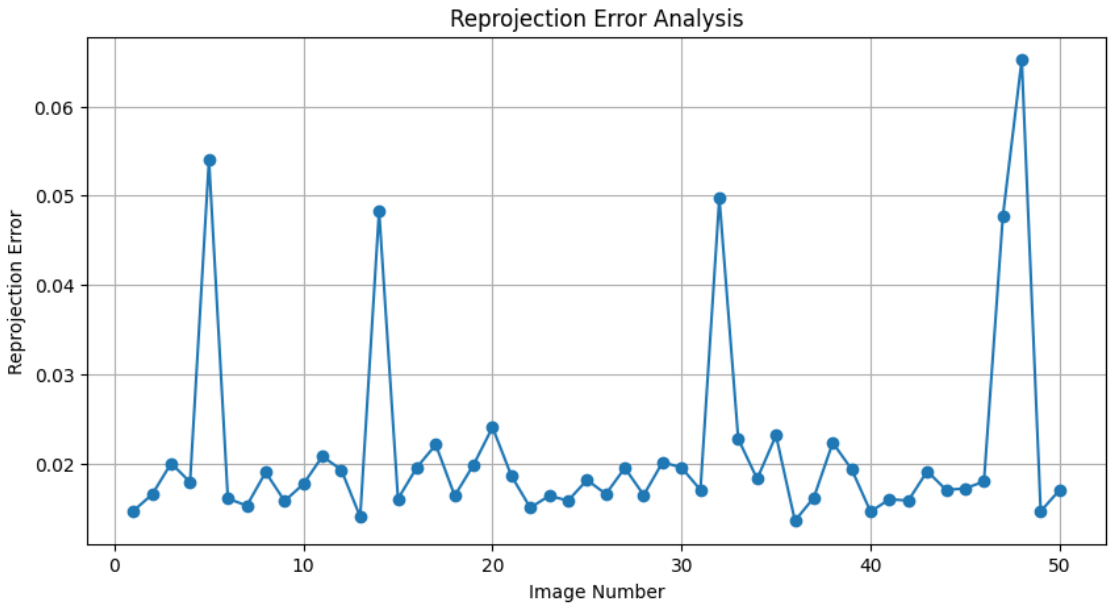
The cv2.projectPoints() function projects the 3D object points (objp) into the 2D image plane using the estimated rotation vectors (rvecs[-1]) and translation vectors (tvecs[-1]), along with the intrinsic camera parameters (mtx) and distortion coefficients (dist). This process calculates the corresponding 2D image points (imgpoints2), providing a mapping of the object's spatial coordinates onto the camera's image plane.

cv2.norm() function calculates the Euclidean distance between the detected corners (corners) and the corresponding reprojection points (imgpoints2). The distance is computed using the L2 norm

(cv2.NORM_L2). Finally, the average distance is computed by dividing the total distance by the number of reprojection points (len(imgpoints2)), yielding the reprojection error. This error metric quantifies the accuracy of the camera calibration process by measuring the deviation between the observed and projected image points.

[ ]:
```
# Calculate and print the mean reprojection error
mean_error = np.mean(reprojection_errors)
print("Mean reprojection error:", mean_error)

# Plot reprojection errors
plt.figure(figsize=(10, 5))
plt.plot(range(1, len(reprojection_errors) + 1), reprojection_errors,␣
 ↪marker='o', linestyle='-')
plt.xlabel('Image Number')
plt.ylabel('Reprojection Error')
plt.title('Reprojection Error Analysis')
plt.grid(True)
plt.show()
```

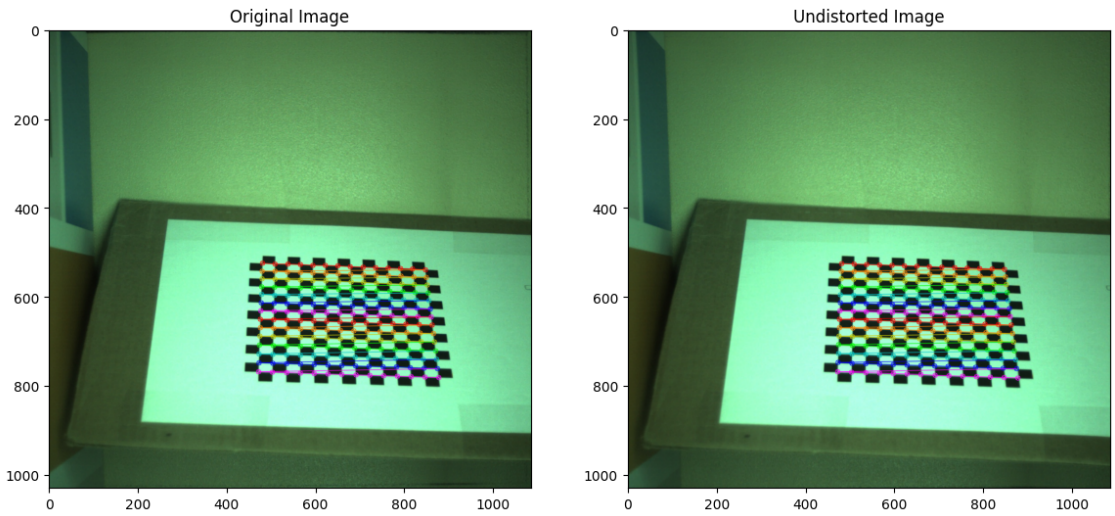
```
Mean reprojection error: 0.02135211514239237
```

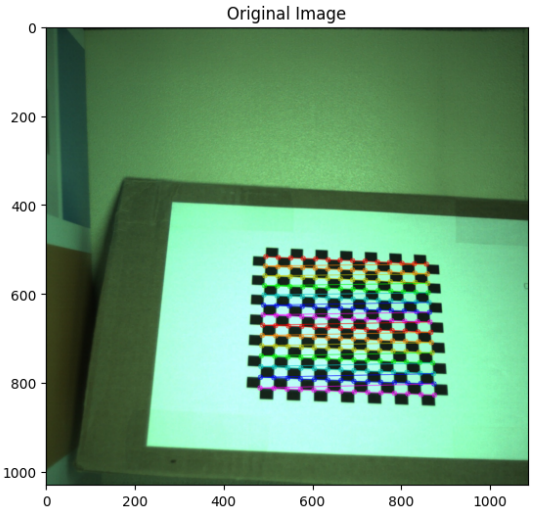


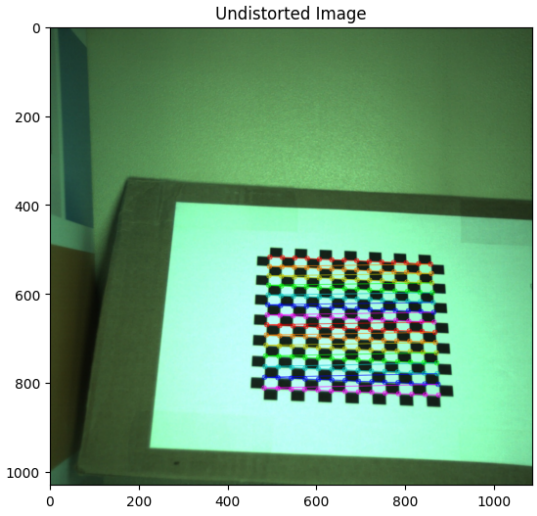
The reprojection error is a critical metric in camera calibration as it quantifies the accuracy of the calibration process. It represents the discrepancy between the observed image points (detected corners) and the projected image points obtained by back-projecting 3D object points onto the image plane using the calibrated camera parameters. A low reprojection error indicates that the calibration accurately models the camera's intrinsic and extrinsic parameters, resulting in precise mapping between 3D world coordinates and 2D image coordinates. Conversely, a high reprojection error suggests inaccuracies in the calibration, which can lead to errors in subsequent computer

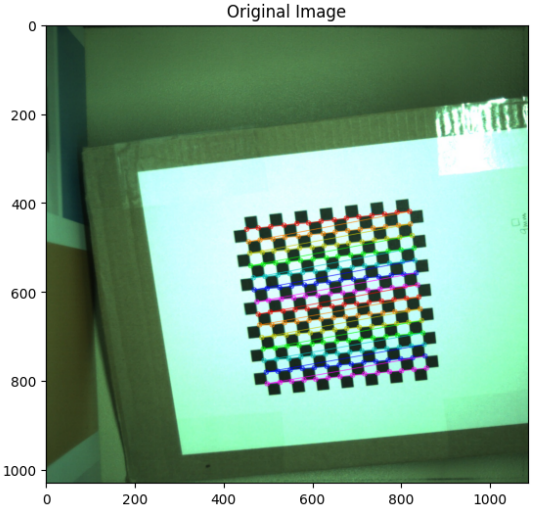
vision tasks such as 3D reconstruction, object tracking, and augmented reality.

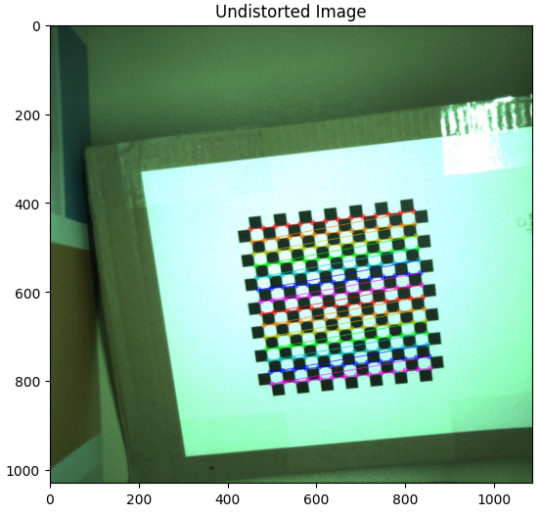
[ ]:
```
# Step 4: Visualization of Calibration Results
# Choose an image to visualize calibration results (e.g., the 12th extracted frame)
im = 1
while im< 5:
  im = im +1
  img = cv2.imread(frame_paths[im])
  h, w = img.shape[:2]

  # Undistort the image
  dst = cv2.undistort(img, mtx, dist, None, mtx)

  # Draw detected corners before calibration
  cv2.drawChessboardCorners(img, (14, 14), imgpoints[im], True)


  # Draw detected corners after calibration
  dst_points, _ = cv2.projectPoints(objp, rvecs[im], tvecs[im], mtx, dist)
  cv2.drawChessboardCorners(dst, (14, 14), dst_points.astype(np.float32), True)
  # Display original and undistorted images
  plt.figure(figsize=(14, 14))
  plt.subplot(1, 2, 1)
  plt.title('Original Image')
  plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
  plt.subplot(1, 2, 2)
  plt.title('Undistorted Image')
  plt.imshow(cv2.cvtColor(dst, cv2.COLOR_BGR2RGB))
  plt.show()
```

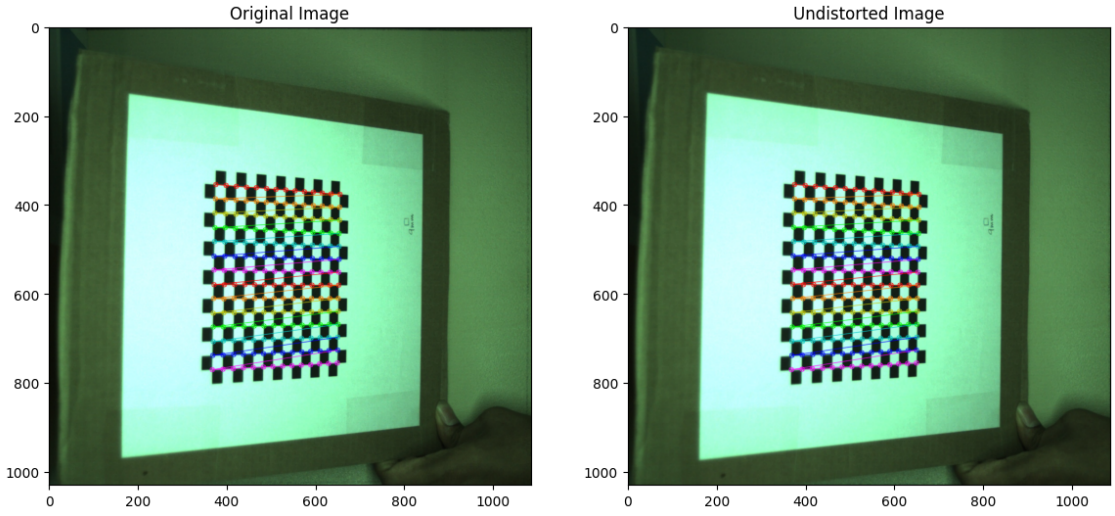



Original Image
0
200
400
600
800
1000
0
200
400
600
800
1000


Undistorted Image
0
200
400
600
800
1000
0
200
400
600
800
1000


Original Image
0
200
400
600
800
1000
0
200
400
600
800
1000


Undistorted Image
0
200
400
600
800
1000
0
200
400
600
800
1000



This code snippet visualizes the calibration results by comparing the original image with its undistorted version. First, it loads the first 5 extracted frames from the list of frame paths and reads its dimensions. Then, it undistorts the image using the camera matrix (mtx), distortion coefficients (dist), and optional rectification transformation (None) obtained during calibration. Next, it draws the detected chessboard corners on the original image using cv2.drawChessboardCorners() before calibration and draws the corresponding corners after calibration on the undistorted image. Finally, it displays both the original and undistorted images side by side using matplotlib.pyplot.

2 Problem 2

3 Class Room

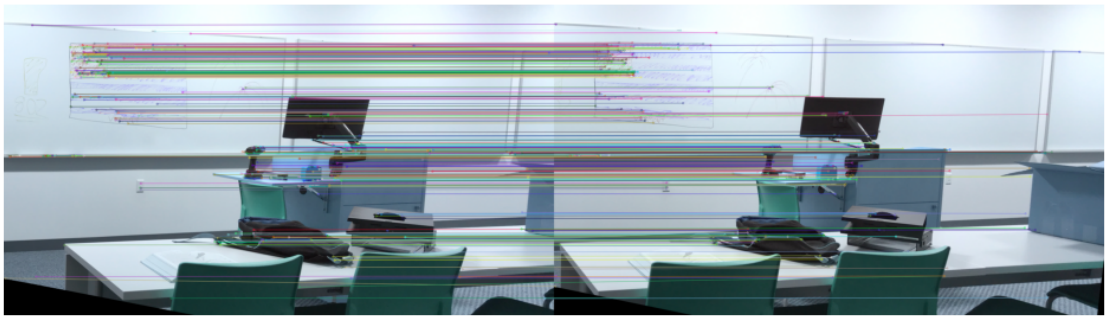
[ ]:
```
import numpy as np
import cv2
import glob
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow  # Importing necessary libraries

frame_count = 0  # Initialize frame count

# Define the directory containing the images
image_dir = '/content/drive/My Drive/problem2_dataset/classroom'

# Get the paths of all image files in the directory
image_paths = glob.glob(os.path.join(image_dir, '*.png'))

frame_dir = 'frames_3'  # Define the directory to save frames

# Create the directory if it doesn't exist
```

```
if not os.path.exists(frame_dir):
    os.makedirs(frame_dir)

# Loop through each image path
for image_path in image_paths:
    # Read the image
    frame = cv2.imread(image_path)

    # Save the image as a frame
    frame_name = os.path.join(frame_dir, f'frame_{frame_count:04d}.png')
    cv2.imwrite(frame_name, frame)

    frame_count += 1  # Increment frame count
```

This code snippet reads images from a directory, saves each image as a frame in a new directory, and increments the frame count. It first defines the image directory and the directory to save frames. Then, it loops through each image path, reads the image, saves it as a frame with a sequential frame name in the specified directory, and increments the frame count.

[ ]:
```
image_paths.sort()  # Sort the image paths

sift = cv2.SIFT_create()  # Create a SIFT object

# Load the first two consecutive images
img1 = cv2.imread(image_paths[0])  # Read the first image
img2 = cv2.imread(image_paths[1])  # Read the second image

# Detect keypoints and compute descriptors for both images
kp1, des1 = sift.detectAndCompute(img1, None)  # Detect keypoints and compute␣
 ↪descriptors for the first image
kp2, des2 = sift.detectAndCompute(img2, None)  # Detect keypoints and compute␣
 ↪descriptors for the second image

# Initialize brute-force matcher
bf = cv2.BFMatcher()

# Match descriptors
matches = bf.knnMatch(des1, des2, k=2)

# Apply ratio test to select good matches
good_matches = []
for m, n in matches:
    if m.distance < 0.5 * n.distance:
        good_matches.append(m)

# Draw matches
```

```
img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good_matches, None,␣
 ↪flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
plt.figure(figsize=(12, 6))
plt.imshow(img_matches)
plt.axis('off')
plt.show()
```



This code snippet sorts the image paths, detects keypoints and computes descriptors for two consecutive images using the SIFT (Scale-Invariant Feature Transform) algorithm, matches descriptors between the images using the brute-force matcher, applies a ratio test to select good matches, draws the matches between the images, and displays the matches.

[ ]:
```
# Convert keypoints to numpy arrays
pts1 = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
pts2 = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)


# pts1 = np.int32(kp1)
# pts2 = np.int32(kp2)


# Estimate the Fundamental matrix using RANSAC
F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_RANSAC)

# Optionally, filter out outliers using the mask
pts1_filtered = pts1[mask.ravel() == 1]
pts2_filtered = pts2[mask.ravel() == 1]
```

converts keypoints from two images to numpy arrays, estimates the Fundamental matrix using the RANSAC algorithm, and optionally filters out outliers using the obtained mask. The Fundamental matrix represents the epipolar geometry between two images and is used for matching keypoints and finding corresponding points between them.

[ ]:
```
# Define the camera matrices and calibration parameters
cam0 = np.array([[1746.24, 0, 14.88],
                 [0, 1746.24, 534.11],
                 [0, 0, 1]])  # Intrinsic matrix for camera 0

cam1 = np.array([[1746.24, 0, 14.88],
                 [0, 1746.24, 534.11],
                 [0, 0, 1]])  # Intrinsic matrix for camera 1

baseline = 678.37  # Baseline (distance between the two cameras)
focal_length = 1746.24  # Focal length of the cameras
width = 1920  # Width of the images
height = 1080  # Height of the images
imgSize = (width, height)  # Image size

# Compute the Essential matrix
E = np.dot(K2.T, np.dot(F, K1))  # Compute the Essential matrix using the␣
 ↪formula E = K2^T * F * K1

# Display the Essential matrix
print("Essential matrix:")
print(E)
```

```
Essential matrix:
[[ 1.17044981e-01 -1.25913694e+02  2.33324657e+01]
 [ 1.19249347e+02 -1.86800737e+00 -8.90317758e+02]
 [-2.27811599e+01  8.85889774e+02 -1.57541175e-01]]
```

Defines camera matrices, calibration parameters, and computes the Essential matrix using the given Fundamental matrix and camera intrinsic matrices. The Essential matrix represents the essential geometric relationship between two calibrated cameras and is used in stereo vision applications for 3D reconstruction and rectification.

[ ]:
```
# Recover the relative pose (rotation and translation) between the two cameras␣
 ↪from the Essential matrix
# The function returns retval, R, t, mask
# - retval: The return value indicating success (True) or failure (False) of␣
 ↪the recovery process
# - R: The 3x3 rotation matrix representing the relative rotation between the␣
 ↪two cameras
# - t: The 3x1 translation vector representing the relative translation between␣
 ↪the two cameras
# - mask: A mask indicating the inliers used for pose recovery

retval, R, t, mask = cv2.recoverPose(E, pts1_filtered, pts2_filtered,␣
 ↪focal=1746.24, pp=(14.88, 534.11))
```

```
# Display the rotation matrix and translation vector
print("Rotation matrix:")
print(R)
print("\nTranslation vector:")
print(t)
```

```
Rotation matrix:
[[ 9.99966918e-01 -5.84126834e-04 -8.11302098e-03]
 [ 5.74348886e-04  9.99999106e-01 -1.20749418e-03]
 [ 8.11371906e-03  1.20279453e-03  9.99966360e-01]]

Translation vector:
[[-0.98970964]
 [-0.02591232]
 [-0.14072451]]
```

[ ]:
```
image_size = (1920, 1080)
retval, H1, H2 = cv2.stereoRectifyUncalibrated(np.float32(pts1_filtered), np.
  ↪float32(pts2_filtered), F, imgSize = image_size)
print("Homography Matrix H1:\n", H1)
print("Homography Matrix H2:\n", H2)


points1_rectified = cv2.perspectiveTransform(pts1_filtered.reshape(-1, 1, 2),␣
  ↪H1).reshape(-1,2)
points2_rectified = cv2.perspectiveTransform(pts2_filtered.reshape(-1, 1, 2),␣
  ↪H2).reshape(-1,2)
```

```
Homography Matrix H1:
 [[-4.69470599e-01  6.12369845e-02 -1.10594843e+01]
 [ 3.68323515e-02 -5.08655822e-01 -3.85022869e+01]
 [ 4.24866349e-05  5.83630096e-07 -5.55281047e-01]]
Homography Matrix H2:
 [[ 9.14993057e-01  2.54779770e-02  6.78485581e+01]
 [-7.54326735e-02  9.98287174e-01  7.33402924e+01]
 [-8.81453092e-05 -2.45440569e-06  1.08594488e+00]]
```

cv2.recoverPose() function to recover the relative pose (rotation and translation) between the two cameras from the Essential matrix and the corresponding filtered keypoints. It then prints the rotation matrix and translation vector representing the relative pose between the two cameras. The rotation matrix R represents how the second camera is rotated relative to the first camera, and the translation vector t represents the translation of the second camera relative to the first camera.

[ ]:
```
def drawlines_rectified(img1, img2, lines, pts1, pts2):
    """
    Draw epipolar lines and feature points on both rectified images.
```

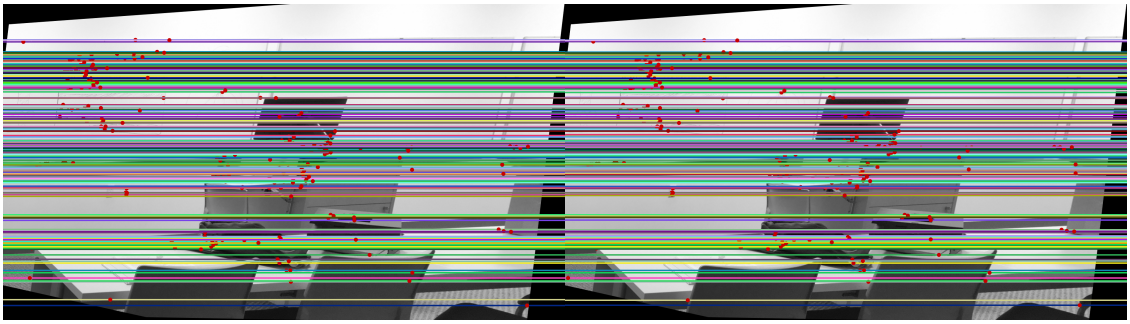
```
    Args:
    - img1: First rectified image (grayscale)
    - img2: Second rectified image (grayscale)
    - lines: Epipolar lines corresponding to keypoints
    - pts1: Keypoints in the first image
    - pts2: Keypoints in the second image

    Returns:
    - img1_color: First rectified image with epipolar lines and feature points␣
↪(color)
    - img2_color: Second rectified image with epipolar lines and feature points␣
↪(color)
    """

    img1_color = cv2.cvtColor(img1, cv2.COLOR_GRAY2BGR)  # Convert first image␣
↪to color
    img2_color = cv2.cvtColor(img2, cv2.COLOR_GRAY2BGR)  # Convert second image␣
↪to color

    np.random.seed(0)  # Set random seed for consistent colors

    r, c = img1.shape  # Get dimensions of the images

    # Iterate over each line, point pair
    for r, pt1, pt2 in zip(lines, pts1, pts2):

        # Generate random color
        color = tuple(np.random.randint(0, 255, 3).tolist())

        # Draw line on first image
        x0, y0 = map(int, [0, pt1[1]])
        x1, y1 = map(int, [c, pt1[1]])
        img1_color = cv2.line(img1_color, (x0, y0), (x1, y1), color, 3)  # Draw␣
↪line
        img1_color = cv2.circle(img1_color, (int(pt1[0]), int(pt1[1])), 7, (0,␣
↪0, 200), -1)  # Draw circle at keypoint

        # Draw line on second image
        x0, y0 = map(int, [0, pt2[1]])
        x1, y1 = map(int, [c, pt2[1]])
        img2_color = cv2.line(img2_color, (x0, y0), (x1, y1), color, 3)  # Draw␣
↪line
        img2_color = cv2.circle(img2_color, (int(pt2[0]), int(pt2[1])), 7, (0,␣
↪0, 200), -1)  # Draw circle at keypoint
```

```
    return img1_color, img2_color  # Return images with epipolar lines and␣
 ↪feature points
```

This function drawlines_rectified() draws epipolar lines and feature points on both rectified images. It takes the rectified images, epipolar lines, and corresponding keypoints as input and returns the images with epipolar lines and feature points drawn in color. Each line is drawn from the left image to the right image, and keypoints are marked with circles. Random colors are assigned to each line for visualization.

[ ]:
```
# Rectify the images using perspective transformation
rectified_img1 = cv2.warpPerspective(img1, H1, (img1.shape[1], img1.shape[0]))
rectified_img2 = cv2.warpPerspective(img2, H2, (img2.shape[1], img2.shape[0]))

# Copy rectified images for further processing
rectified_image1 = rectified_img1
rectified_image2 = rectified_img2

# Display the rectified images
cv2_imshow(rectified_img1)
cv2_imshow(rectified_img2)

# Convert rectified images to grayscale
rectified_img1_gray = cv2.cvtColor(rectified_img1, cv2.COLOR_BGR2GRAY)
rectified_img2_gray = cv2.cvtColor(rectified_img2, cv2.COLOR_BGR2GRAY)

# Compute rectified keypoints
rectified_points1 = cv2.perspectiveTransform(pts1_filtered.reshape(-1, 1, 2),␣
 ↪H1).reshape(-1, 2)
rectified_points2 = cv2.perspectiveTransform(pts2_filtered.reshape(-1, 1, 2),␣
 ↪H2).reshape(-1, 2)

# Compute epipolar lines for rectified points
lines_1 = cv2.computeCorrespondEpilines(rectified_points2.reshape(-1, 1, 2), 2,␣
 ↪F)
lines_1 = lines_1.reshape(-1, 3)

# Draw epipolar lines on rectified images
image_5, image_6 = drawlines_rectified(rectified_img1_gray,␣
 ↪rectified_img2_gray, lines_1, rectified_points1, rectified_points2)

# Compute epipolar lines for rectified points (inverse direction)
lines_2 = cv2.computeCorrespondEpilines(rectified_points1.reshape(-1, 1, 2), 1,␣
 ↪F)
lines_2 = lines_2.reshape(-1, 3)

# Draw epipolar lines on rectified images (inverse direction)
```

```
image_3, image_4 = drawlines_rectified(rectified_img1_gray,␣
 ↪rectified_img2_gray, lines_2, rectified_points2, rectified_points1)

# Concatenate the images horizontally for visualization
result = np.concatenate((image_3, image_5), axis=1)

# Display the result
cv2_imshow(result)
```

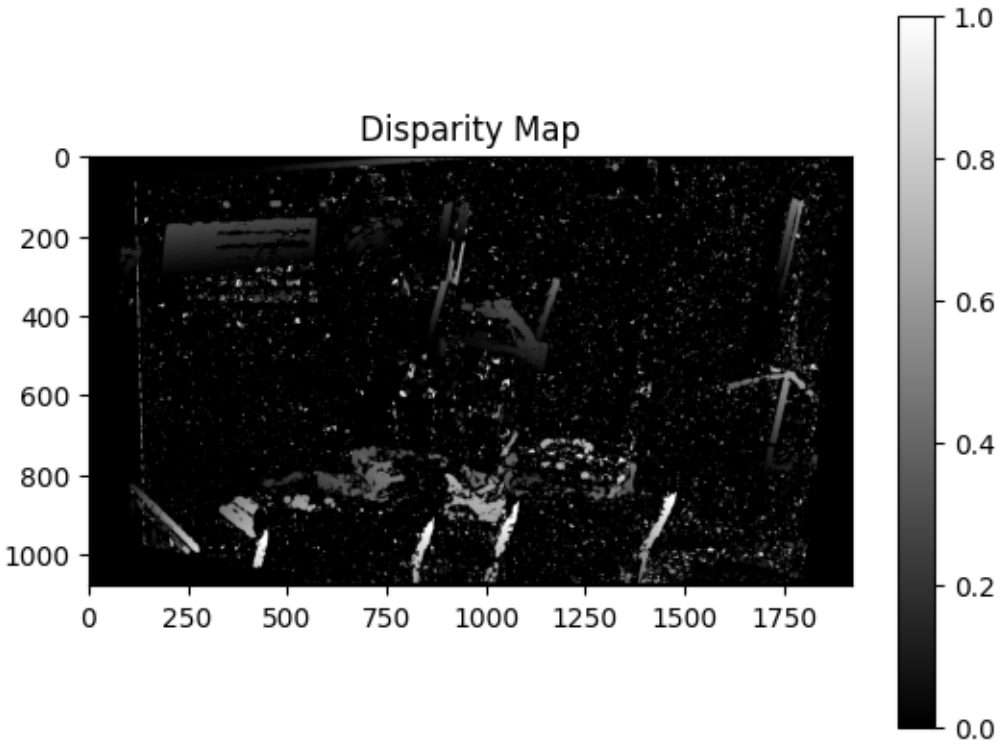


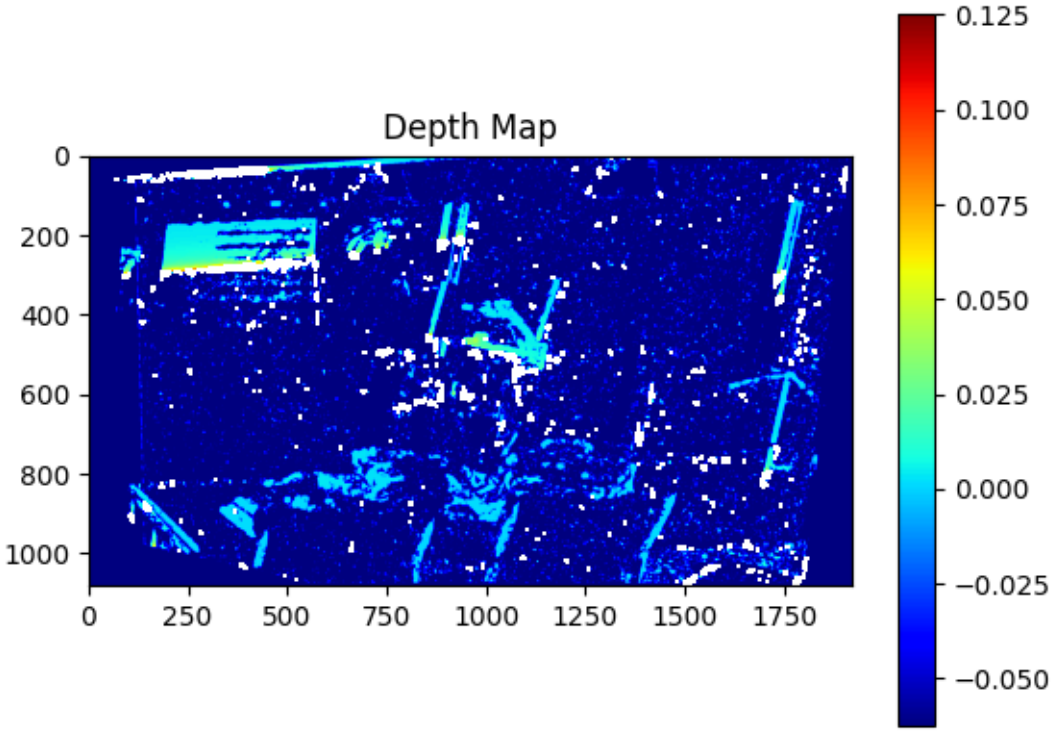
This code snippet rectifies two images using perspective transformation with the provided homography matrices H1 and H2, converts them to grayscale, computes epipolar lines for the rectified keypoints, draws the epipolar lines on the rectified images, and finally concatenates the images horizontally for visualization. The result shows the rectified images with horizontal epipolar lines drawn on them, indicating the correspondence between the points in the rectified images.

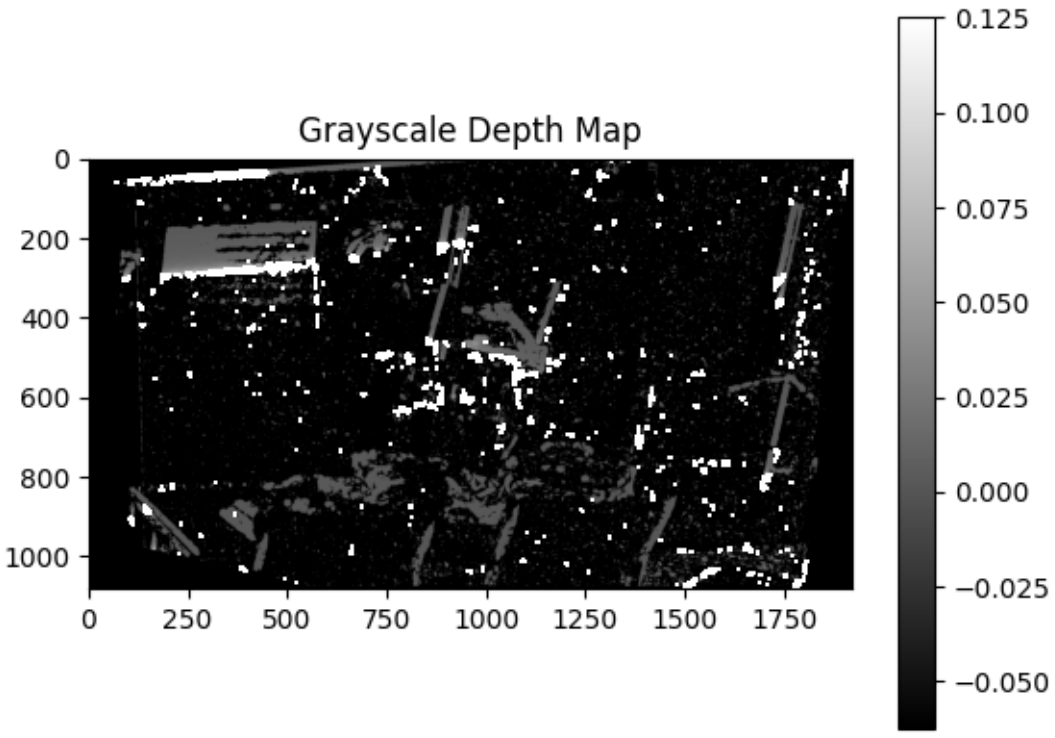
[ ]:
```python
def disparity_and_depth(baseline, f, img):
    """
    Compute the depth map and array depth from the given baseline, focal length, and disparity map.

    Args:
    - baseline: Baseline (distance between the two cameras)
    - f: Focal length of the cameras
    - img: Disparity map

    Returns:
    - depthmap: Depth map computed from the disparity map
    - array_depth: Array depth computed from the disparity map
    """
    depthmap = np.zeros((img.shape[0], img.shape[1]))  # Initialize depth map array
    array_depth = np.zeros((img.shape[0], img.shape[1]))  # Initialize array depth array

    for i in range(depthmap.shape[0]):
        for j in range(depthmap.shape[1]):
            depthmap[i][j] = 1 / img[i][j]  # Compute depth from disparity
            array_depth[i][j] = baseline * f / img[i][j]  # Compute array depth

    return depthmap, array_depth  # Return depth map and array depth

# Convert rectified images to grayscale
```

```
rectified_image1_gray_ = cv2.cvtColor(rectified_image1, cv2.COLOR_BGR2GRAY)
rectified_image2_gray_ = cv2.cvtColor(rectified_image2, cv2.COLOR_BGR2GRAY)

# Compute disparity map using StereoBM
stereo = cv2.StereoBM_create(numDisparities=64, blockSize=13)
disparity_ = stereo.compute(rectified_image1_gray_, rectified_image2_gray_)

# Normalize the disparity map
disparity_normalized = cv2.normalize(disparity_, None, alpha=0, beta=1,␣
 ↪norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

# Compute depth map and array depth
depthmap, _ = disparity_and_depth(baseline, focal_length, disparity_)

# Display the disparity map
plt.imshow(disparity_normalized, cmap='gray')
plt.title('Disparity Map')
plt.colorbar()
plt.show()

# Display the depth map (color)
plt.imshow(depthmap, cmap='jet')
plt.title('Depth Map')
plt.colorbar()
plt.show()

# Display the depth map (grayscale)
plt.imshow(depthmap, cmap='gray')
plt.title('Grayscale Depth Map')
plt.colorbar()
plt.show()
```

```
<ipython-input-140-640fbfd0f55d>:19: RuntimeWarning: divide by zero encountered
in divide
  depthmap[i][j] = 1 / img[i][j]  # Compute depth from disparity
<ipython-input-140-640fbfd0f55d>:20: RuntimeWarning: divide by zero encountered
in divide
  array_depth[i][j] = baseline * f / img[i][j]  # Compute array depth
```


Disparity Map
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
1.0
0.8
0.6
0.4
0.2
0.0


Depth Map
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
0.125
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050



Tis code snippet defines a function disparity_and_depth() to compute the depth map and array depth from a given baseline, focal length, and disparity map. It then computes the disparity map using the StereoBM algorithm, normalizes it, and computes the depth map and array depth. Finally, it displays the disparity map, depth map in color, and depth map in grayscale for visualization.

#Storage Room

[ ]:
```
frame_count = 0  # Initialize frame count

# Define the directory containing the images
image_dir = '/content/drive/My Drive/problem2_dataset/storageroom'

# Get the paths of all image files in the directory
image_paths = glob.glob(os.path.join(image_dir, '*.png'))

frame_dir = 'frames_4'  # Define the directory to save frames

# Create the directory if it doesn't exist
if not os.path.exists(frame_dir):
    os.makedirs(frame_dir)

# Loop through each image path
```

```
for image_path in image_paths:
    # Read the image
    frame = cv2.imread(image_path)

    # Save the image as a frame
    frame_name = os.path.join(frame_dir, f'frame_{frame_count:04d}.png')
    cv2.imwrite(frame_name, frame)

    frame_count += 1  # Increment frame count
```

[ ]:
```
image_paths.sort()  # Sort the image paths

sift = cv2.SIFT_create()  # Create a SIFT object

# Load the first two consecutive images
img1 = cv2.imread(image_paths[0])  # Read the first image
img2 = cv2.imread(image_paths[1])  # Read the second image

# Detect keypoints and compute descriptors for both images
kp1, des1 = sift.detectAndCompute(img1, None)  # Detect keypoints and compute␣
 ↪descriptors for the first image
kp2, des2 = sift.detectAndCompute(img2, None)  # Detect keypoints and compute␣
 ↪descriptors for the second image

# Initialize brute-force matcher
bf = cv2.BFMatcher()

# Match descriptors
matches = bf.knnMatch(des1, des2, k=2)

# Apply ratio test to select good matches
good_matches = []
for m, n in matches:
    if m.distance < 0.5 * n.distance:
        good_matches.append(m)

# Draw matches
img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good_matches, None,␣
 ↪flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
plt.figure(figsize=(12, 6))
plt.imshow(img_matches)
plt.axis('off')
plt.show()
```

[ ]:
```python
# Convert keypoints to numpy arrays
pts1 = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
pts2 = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)


# pts1 = np.int32(kp1)
# pts2 = np.int32(kp2)


# Estimate the Fundamental matrix using RANSAC
F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_RANSAC)

# Optionally, filter out outliers using the mask
pts1_filtered = pts1[mask.ravel() == 1]
pts2_filtered = pts2[mask.ravel() == 1]
```

[ ]:
```python
import numpy as np

# Define the camera matrices and calibration parameters
cam0 = np.array([[1742.11, 0, 804.9],
                 [0, 1742.11, 541.22],
                 [0, 0, 1]])  # Intrinsic matrix for camera 0

cam1 = np.array([[1742.11, 0, 804.9],
                 [0, 1742.11, 541.22],
                 [0, 0, 1]])  # Intrinsic matrix for camera 1

baseline = 221.6  # Baseline (distance between the two cameras)
focal_length = 1746.24  # Focal length of the cameras
width = 1920  # Width of the images
height = 1080  # Height of the images
imgSize = (width, height)  # Image size

# Compute the Essential matrix
```

```
E = np.dot(K2.T, np.dot(F, K1))  # Compute the Essential matrix using the␣
 ↪formula E = K2^T * F * K1

# Display the Essential matrix
print("Essential matrix:")
print(E)
```

```
Essential matrix:
[[ 1.22143829e-01 -2.02105683e+02  4.70869369e+01]
 [ 2.01535301e+02 -3.25048274e+00 -5.97986507e+02]
 [-4.72276759e+01  5.95420424e+02  5.19498423e-01]]
```

[ ]:
```
# Recover the relative pose (rotation and translation) between the two cameras␣
 ↪from the Essential matrix
# The function returns retval, R, t, mask
# - retval: The return value indicating success (True) or failure (False) of␣
 ↪the recovery process
# - R: The 3x3 rotation matrix representing the relative rotation between the␣
 ↪two cameras
# - t: The 3x1 translation vector representing the relative translation between␣
 ↪the two cameras
# - mask: A mask indicating the inliers used for pose recovery

retval, R, t, mask = cv2.recoverPose(E, pts1_filtered, pts2_filtered,␣
 ↪focal=1746.24, pp=(14.88, 534.11))

# Display the rotation matrix and translation vector
print("Rotation matrix:")
print(R)
print("\nTranslation vector:")
print(t)
```

```
Rotation matrix:
[[ 9.99995952e-01 -6.74574742e-04 -2.76435114e-03]
 [ 6.68553976e-04  9.99997404e-01 -2.17834881e-03]
 [ 2.76581342e-03  2.17649188e-03  9.99993807e-01]]

Translation vector:
[[-0.94417164]
 [-0.07462518]
 [-0.32089094]]
```

[ ]:
```
image_size = (1920, 1080)
retval, H1, H2 = cv2.stereoRectifyUncalibrated(np.float32(pts1_filtered), np.
 ↪float32(pts2_filtered), F, imgSize = image_size)
print("Homography Matrix H1:\n", H1)
print("Homography Matrix H2:\n", H2)
```

```
points1_rectified = cv2.perspectiveTransform(pts1_filtered.reshape(-1, 1, 2),␣
 ↪H1).reshape(-1,2)
points2_rectified = cv2.perspectiveTransform(pts2_filtered.reshape(-1, 1, 2),␣
 ↪H2).reshape(-1,2)
```

```
Homography Matrix H1:
 [[-2.67131520e-01 -2.42403991e-02 -5.32664858e+01]
 [ 7.72818718e-02 -3.42608080e-01 -7.86345701e+01]
 [ 8.18140408e-05  6.51198069e-06 -4.31814666e-01]]
Homography Matrix H2:
 [[ 7.68585311e-01  7.33605179e-02  1.82543422e+02]
 [-2.22642741e-01  9.83293931e-01  2.22758309e+02]
 [-2.36344111e-04 -2.25587532e-05  1.23907207e+00]]
```

[ ]:
```
def drawlines_rectified(img1, img2, lines, pts1, pts2):
    """
    Draw epipolar lines and feature points on both rectified images.

    Args:
    - img1: First rectified image (grayscale)
    - img2: Second rectified image (grayscale)
    - lines: Epipolar lines corresponding to keypoints
    - pts1: Keypoints in the first image
    - pts2: Keypoints in the second image

    Returns:
    - img1_color: First rectified image with epipolar lines and feature points␣
 ↪(color)
    - img2_color: Second rectified image with epipolar lines and feature points␣
 ↪(color)
    """

    img1_color = cv2.cvtColor(img1, cv2.COLOR_GRAY2BGR)  # Convert first image␣
 ↪to color
    img2_color = cv2.cvtColor(img2, cv2.COLOR_GRAY2BGR)  # Convert second image␣
 ↪to color

    np.random.seed(0)  # Set random seed for consistent colors

    r, c = img1.shape  # Get dimensions of the images

    # Iterate over each line, point pair
    for r, pt1, pt2 in zip(lines, pts1, pts2):

        # Generate random color
```

```
        color = tuple(np.random.randint(0, 255, 3).tolist())

        # Draw line on first image
        x0, y0 = map(int, [0, pt1[1]])
        x1, y1 = map(int, [c, pt1[1]])
        img1_color = cv2.line(img1_color, (x0, y0), (x1, y1), color, 3)  # Draw␣
 ↪line
        img1_color = cv2.circle(img1_color, (int(pt1[0]), int(pt1[1])), 7, (0,␣
 ↪0, 200), -1)  # Draw circle at keypoint

        # Draw line on second image
        x0, y0 = map(int, [0, pt2[1]])
        x1, y1 = map(int, [c, pt2[1]])
        img2_color = cv2.line(img2_color, (x0, y0), (x1, y1), color, 3)  # Draw␣
 ↪line
        img2_color = cv2.circle(img2_color, (int(pt2[0]), int(pt2[1])), 7, (0,␣
 ↪0, 200), -1)  # Draw circle at keypoint

    return img1_color, img2_color  # Return images with epipolar lines and␣
 ↪feature points
```

[ ]:
```
# Rectify the images using perspective transformation
rectified_img1 = cv2.warpPerspective(img1, H1, (img1.shape[1], img1.shape[0]))
rectified_img2 = cv2.warpPerspective(img2, H2, (img2.shape[1], img2.shape[0]))

# Copy rectified images for further processing
rectified_image1 = rectified_img1
rectified_image2 = rectified_img2

# Display the rectified images
cv2_imshow(rectified_img1)
cv2_imshow(rectified_img2)

# Convert rectified images to grayscale
rectified_img1_gray = cv2.cvtColor(rectified_img1, cv2.COLOR_BGR2GRAY)
rectified_img2_gray = cv2.cvtColor(rectified_img2, cv2.COLOR_BGR2GRAY)

# Compute rectified keypoints
rectified_points1 = cv2.perspectiveTransform(pts1_filtered.reshape(-1, 1, 2),␣
 ↪H1).reshape(-1, 2)
rectified_points2 = cv2.perspectiveTransform(pts2_filtered.reshape(-1, 1, 2),␣
 ↪H2).reshape(-1, 2)

# Compute epipolar lines for rectified points
lines_1 = cv2.computeCorrespondEpilines(rectified_points2.reshape(-1, 1, 2), 2,␣
 ↪F)
```

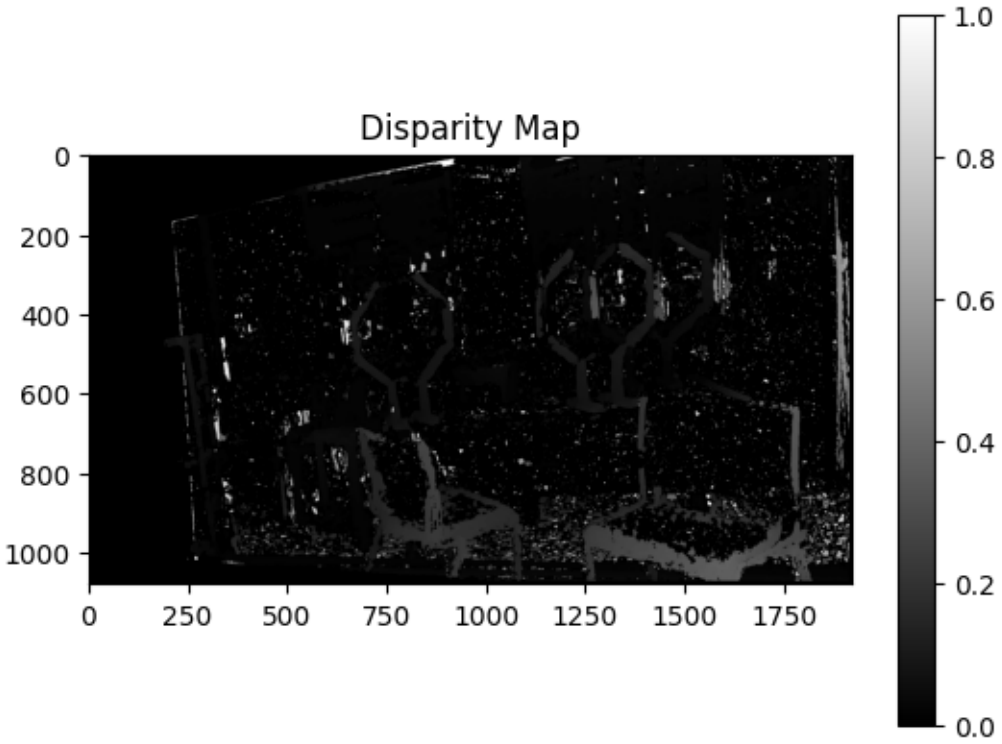
```
lines_1 = lines_1.reshape(-1, 3)

# Draw epipolar lines on rectified images
image_5, image_6 = drawlines_rectified(rectified_img1_gray,␣
 ↪rectified_img2_gray, lines_1, rectified_points1, rectified_points2)

# Compute epipolar lines for rectified points (inverse direction)
lines_2 = cv2.computeCorrespondEpilines(rectified_points1.reshape(-1, 1, 2), 1,␣
 ↪F)
lines_2 = lines_2.reshape(-1, 3)

# Draw epipolar lines on rectified images (inverse direction)
image_3, image_4 = drawlines_rectified(rectified_img1_gray,␣
 ↪rectified_img2_gray, lines_2, rectified_points2, rectified_points1)

# Concatenate the images horizontally for visualization
result = np.concatenate((image_3, image_5), axis=1)

# Display the result
cv2_imshow(result)
```

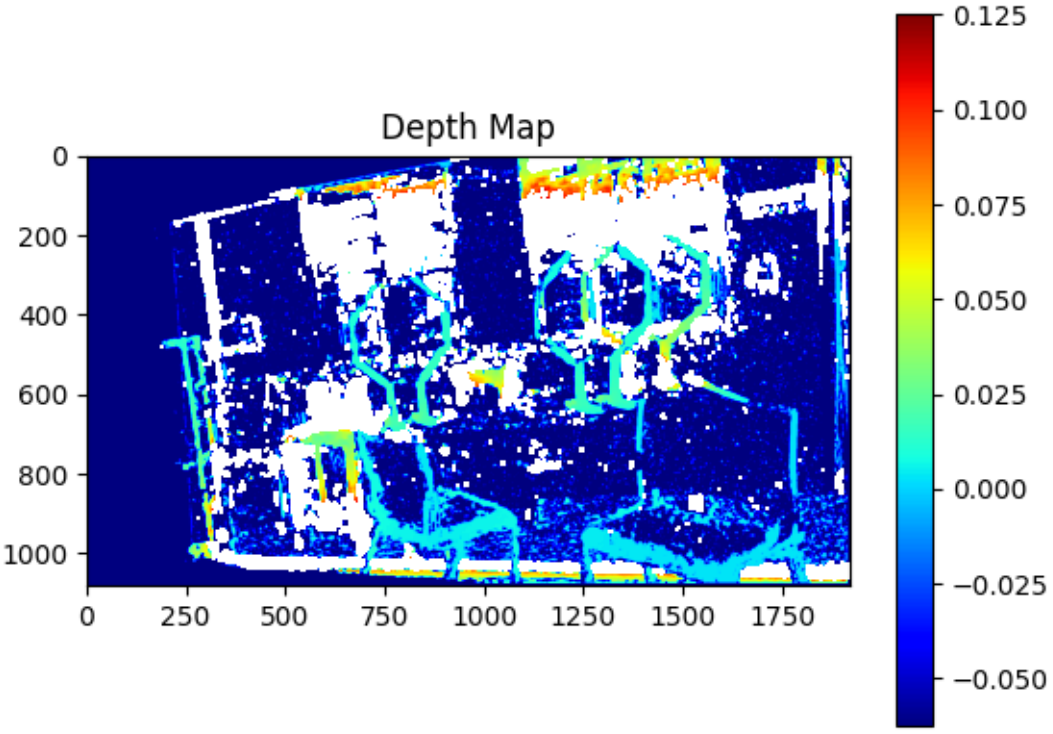
[ ]:
```
def disparity_and_depth(baseline, f, img):
    """
    Compute the depth map and array depth from the given baseline, focal length, and disparity map.

    Args:
    - baseline: Baseline (distance between the two cameras)
    - f: Focal length of the cameras
    - img: Disparity map

    Returns:
    - depthmap: Depth map computed from the disparity map
    - array_depth: Array depth computed from the disparity map
    """
```

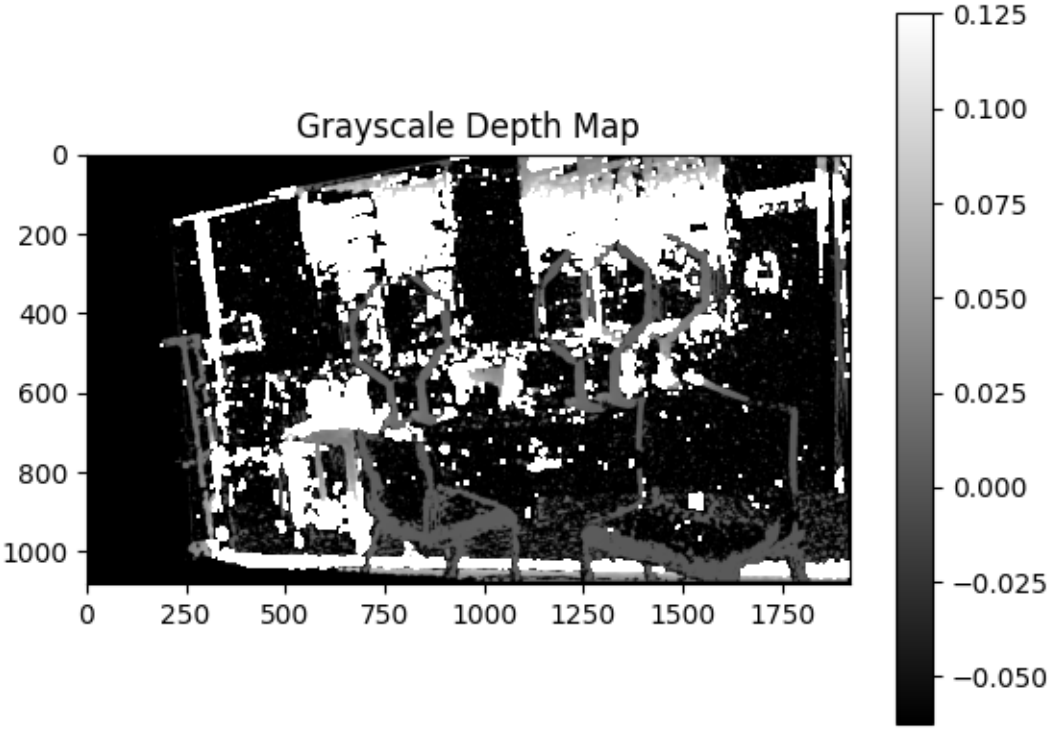
```
    depthmap = np.zeros((img.shape[0], img.shape[1]))  # Initialize depth map␣
 ↪array
    array_depth = np.zeros((img.shape[0], img.shape[1]))  # Initialize array␣
 ↪depth array

    for i in range(depthmap.shape[0]):
        for j in range(depthmap.shape[1]):
            depthmap[i][j] = 1 / img[i][j]  # Compute depth from disparity
            array_depth[i][j] = baseline * f / img[i][j]  # Compute array depth

    return depthmap, array_depth  # Return depth map and array depth

# Convert rectified images to grayscale
rectified_image1_gray_ = cv2.cvtColor(rectified_image1, cv2.COLOR_BGR2GRAY)
rectified_image2_gray_ = cv2.cvtColor(rectified_image2, cv2.COLOR_BGR2GRAY)

# Compute disparity map using StereoBM
stereo = cv2.StereoBM_create(numDisparities=64, blockSize=13)
disparity_ = stereo.compute(rectified_image1_gray_, rectified_image2_gray_)

# Normalize the disparity map
disparity_normalized = cv2.normalize(disparity_, None, alpha=0, beta=1,␣
 ↪norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

# Compute depth map and array depth
depthmap, _ = disparity_and_depth(baseline, focal_length, disparity_)

# Display the disparity map
plt.imshow(disparity_normalized, cmap='gray')
plt.title('Disparity Map')
plt.colorbar()
plt.show()

# Display the depth map (color)
plt.imshow(depthmap, cmap='jet')
plt.title('Depth Map')
plt.colorbar()
plt.show()

# Display the depth map (grayscale)
plt.imshow(depthmap, cmap='gray')
plt.title('Grayscale Depth Map')
plt.colorbar()
plt.show()
```

```
<ipython-input-149-640fbfd0f55d>:19: RuntimeWarning: divide by zero encountered
in divide
```

```
  depthmap[i][j] = 1 / img[i][j]  # Compute depth from disparity
<ipython-input-149-640fbfd0f55d>:20: RuntimeWarning: divide by zero encountered
in divide
  array_depth[i][j] = baseline * f / img[i][j]  # Compute array depth
```




Depth Map
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
0.125
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050


Grayscale Depth Map
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
0.125
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050

#Traproom

[ ]:
```
frame_count = 0  # Initialize frame count

# Define the directory containing the images
image_dir = '/content/drive/My Drive/problem2_dataset/traproom'

# Get the paths of all image files in the directory
image_paths = glob.glob(os.path.join(image_dir, '*.png'))

frame_dir = 'frames_4'  # Define the directory to save frames

# Create the directory if it doesn't exist
if not os.path.exists(frame_dir):
    os.makedirs(frame_dir)

# Loop through each image path
for image_path in image_paths:
    # Read the image
    frame = cv2.imread(image_path)

    # Save the image as a frame
    frame_name = os.path.join(frame_dir, f'frame_{frame_count:04d}.png')
    cv2.imwrite(frame_name, frame)

    frame_count += 1  # Increment frame count
```

[ ]:
```
image_paths.sort()  # Sort the image paths

sift = cv2.SIFT_create()  # Create a SIFT object

# Load the first two consecutive images
img1 = cv2.imread(image_paths[0])  # Read the first image
img2 = cv2.imread(image_paths[1])  # Read the second image

# Detect keypoints and compute descriptors for both images
kp1, des1 = sift.detectAndCompute(img1, None)  # Detect keypoints and compute␣
 ↪descriptors for the first image
kp2, des2 = sift.detectAndCompute(img2, None)  # Detect keypoints and compute␣
 ↪descriptors for the second image

# Initialize brute-force matcher
bf = cv2.BFMatcher()

# Match descriptors
matches = bf.knnMatch(des1, des2, k=2)
```

```
# Apply ratio test to select good matches
good_matches = []
for m, n in matches:
    if m.distance < 0.5 * n.distance:
        good_matches.append(m)

# Draw matches
img_matches = cv2.drawMatches(img1, kp1, img2, kp2, good_matches, None,␣
 ↪flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

# Display the matches
plt.figure(figsize=(12, 6))
plt.imshow(img_matches)
plt.axis('off')
plt.show()
```

[ ]:
```
# Convert keypoints to numpy arrays
pts1 = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
pts2 = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)


# pts1 = np.int32(kp1)
# pts2 = np.int32(kp2)


# Estimate the Fundamental matrix using RANSAC
F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_RANSAC)

# Optionally, filter out outliers using the mask
pts1_filtered = pts1[mask.ravel() == 1]
pts2_filtered = pts2[mask.ravel() == 1]
```

[ ]:
```
# Define the camera matrices and calibration parameters
cam0 = np.array([[1769.02, 0, 1271.89],
                 [0, 1746.24, 527.17],
```

```
                  [0, 0, 1]])  # Intrinsic matrix for camera 0

cam1 = np.array([[1769.02, 0, 1271.89],
                  [0, 1746.24, 527.17],
                  [0, 0, 1]])  # Intrinsic matrix for camera 1

baseline = 295.44  # Baseline (distance between the two cameras)
focal_length = 1769.02  # Focal length of the cameras
width = 1920  # Width of the images
height = 1080  # Height of the images
imgSize = (width, height)  # Image size

# Compute the Essential matrix
E = np.dot(K2.T, np.dot(F, K1))  # Compute the Essential matrix using the␣
 ↪formula E = K2^T * F * K1

# Display the Essential matrix
print("Essential matrix:")
print(E)
```

```
Essential matrix:
[[ 2.13168014e+01 -2.21540811e+04 -1.91770953e+02]
 [ 2.17925223e+04 -1.38514022e+03  2.43054396e+04]
 [ 1.90085926e+02 -2.48318196e+04 -2.94908894e+01]]
```

[ ]:
```
# Recover the relative pose (rotation and translation) between the two cameras␣
 ↪from the Essential matrix
# The function returns retval, R, t, mask
# - retval: The return value indicating success (True) or failure (False) of␣
 ↪the recovery process
# - R: The 3x3 rotation matrix representing the relative rotation between the␣
 ↪two cameras
# - t: The 3x1 translation vector representing the relative translation between␣
 ↪the two cameras
# - mask: A mask indicating the inliers used for pose recovery

retval, R, t, mask = cv2.recoverPose(E, pts1_filtered, pts2_filtered,␣
 ↪focal=1746.24, pp=(14.88, 534.11))

# Display the rotation matrix and translation vector
print("Rotation matrix:")
print(R)
print("\nTranslation vector:")
print(t)
```

```
Rotation matrix:
[[ 0.99991297 -0.01300614 -0.00221376]
```

```
 [ 0.01304126  0.99977597  0.01667117]
 [ 0.00199643 -0.01669859  0.99985858]]

Translation vector:
[[-0.74604382]
 [-0.00507838]
 [ 0.66587749]]
```

[ ]:
```
image_size = (1920, 1080)
retval, H1, H2 = cv2.stereoRectifyUncalibrated(np.float32(pts1_filtered), np.
 ↪float32(pts2_filtered), F, imgSize = image_size)
print("Homography Matrix H1:\n", H1)
print("Homography Matrix H2:\n", H2)


points1_rectified = cv2.perspectiveTransform(pts1_filtered.reshape(-1, 1, 2),␣
 ↪H1).reshape(-1,2)
points2_rectified = cv2.perspectiveTransform(pts2_filtered.reshape(-1, 1, 2),␣
 ↪H2).reshape(-1,2)
```

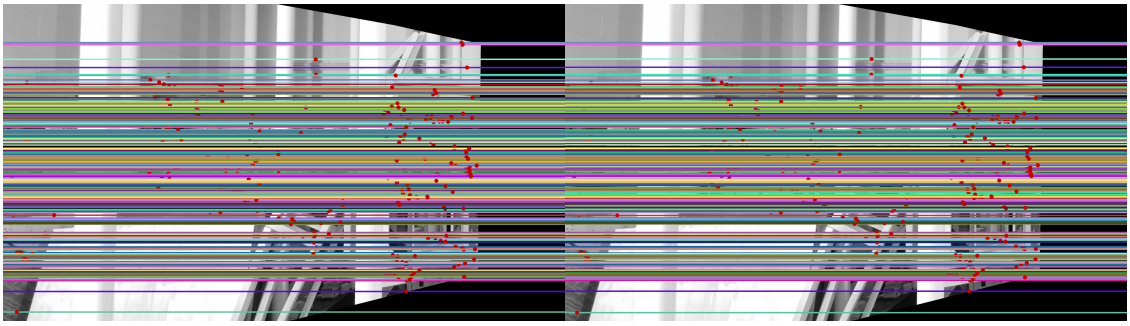
```
Homography Matrix H1:
 [[-1.89447276e+01  6.54474538e-01  4.98616222e+03]
 [-2.56434696e+00 -1.44584265e+01  2.51779281e+03]
 [-4.95092353e-03  2.81366524e-04 -9.71959894e+00]]
Homography Matrix H2:
 [[ 1.33081629e+00  8.80969106e-03 -3.22340874e+02]
 [ 1.79476869e-01  1.00121001e+00 -1.72951197e+02]
 [ 3.44623128e-04  2.28132411e-06  6.67929882e-01]]
```

[ ]:
```
def drawlines_rectified(img1, img2, lines, pts1, pts2):
    """
    Draw epipolar lines and feature points on both rectified images.

    Args:
    - img1: First rectified image (grayscale)
    - img2: Second rectified image (grayscale)
    - lines: Epipolar lines corresponding to keypoints
    - pts1: Keypoints in the first image
    - pts2: Keypoints in the second image

    Returns:
    - img1_color: First rectified image with epipolar lines and feature points␣
 ↪(color)
    - img2_color: Second rectified image with epipolar lines and feature points␣
 ↪(color)
    """
```

```
    img1_color = cv2.cvtColor(img1, cv2.COLOR_GRAY2BGR)  # Convert first image␣
↪to color
    img2_color = cv2.cvtColor(img2, cv2.COLOR_GRAY2BGR)  # Convert second image␣
↪to color

    np.random.seed(0)  # Set random seed for consistent colors

    r, c = img1.shape  # Get dimensions of the images

    # Iterate over each line, point pair
    for r, pt1, pt2 in zip(lines, pts1, pts2):

        # Generate random color
        color = tuple(np.random.randint(0, 255, 3).tolist())

        # Draw line on first image
        x0, y0 = map(int, [0, pt1[1]])
        x1, y1 = map(int, [c, pt1[1]])
        img1_color = cv2.line(img1_color, (x0, y0), (x1, y1), color, 3)  # Draw␣
↪line
        img1_color = cv2.circle(img1_color, (int(pt1[0]), int(pt1[1])), 7, (0,␣
↪0, 200), -1)  # Draw circle at keypoint

        # Draw line on second image
        x0, y0 = map(int, [0, pt2[1]])
        x1, y1 = map(int, [c, pt2[1]])
        img2_color = cv2.line(img2_color, (x0, y0), (x1, y1), color, 3)  # Draw␣
↪line
        img2_color = cv2.circle(img2_color, (int(pt2[0]), int(pt2[1])), 7, (0,␣
↪0, 200), -1)  # Draw circle at keypoint

    return img1_color, img2_color  # Return images with epipolar lines and␣
↪feature points
```

[ ]:
```
# Rectify the images using perspective transformation
rectified_img1 = cv2.warpPerspective(img1, H1, (img1.shape[1], img1.shape[0]))
rectified_img2 = cv2.warpPerspective(img2, H2, (img2.shape[1], img2.shape[0]))

# Copy rectified images for further processing
rectified_image1 = rectified_img1
rectified_image2 = rectified_img2

# Display the rectified images
cv2_imshow(rectified_img1)
cv2_imshow(rectified_img2)
```

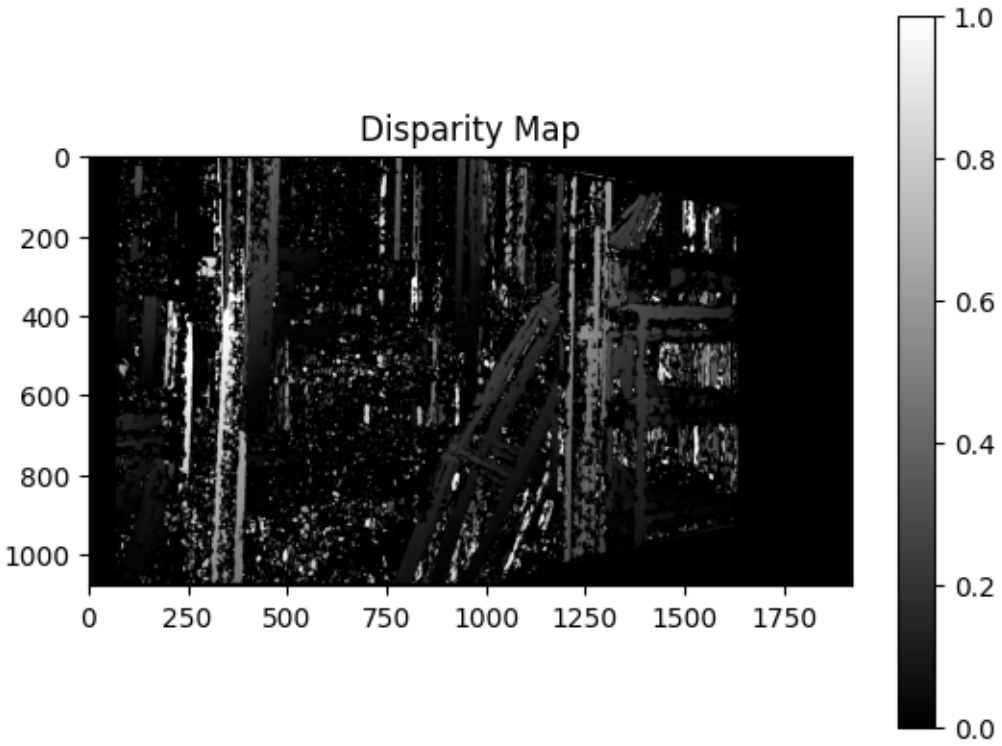
```
# Convert rectified images to grayscale
rectified_img1_gray = cv2.cvtColor(rectified_img1, cv2.COLOR_BGR2GRAY)
rectified_img2_gray = cv2.cvtColor(rectified_img2, cv2.COLOR_BGR2GRAY)

# Compute rectified keypoints
rectified_points1 = cv2.perspectiveTransform(pts1_filtered.reshape(-1, 1, 2),␣
 ↪H1).reshape(-1, 2)
rectified_points2 = cv2.perspectiveTransform(pts2_filtered.reshape(-1, 1, 2),␣
 ↪H2).reshape(-1, 2)

# Compute epipolar lines for rectified points
lines_1 = cv2.computeCorrespondEpilines(rectified_points2.reshape(-1, 1, 2), 2,␣
 ↪F)
lines_1 = lines_1.reshape(-1, 3)

# Draw epipolar lines on rectified images
image_5, image_6 = drawlines_rectified(rectified_img1_gray,␣
 ↪rectified_img2_gray, lines_1, rectified_points1, rectified_points2)

# Compute epipolar lines for rectified points (inverse direction)
lines_2 = cv2.computeCorrespondEpilines(rectified_points1.reshape(-1, 1, 2), 1,␣
 ↪F)
lines_2 = lines_2.reshape(-1, 3)

# Draw epipolar lines on rectified images (inverse direction)
image_3, image_4 = drawlines_rectified(rectified_img1_gray,␣
 ↪rectified_img2_gray, lines_2, rectified_points2, rectified_points1)

# Concatenate the images horizontally for visualization
result = np.concatenate((image_3, image_5), axis=1)

# Display the result
cv2_imshow(result)
```

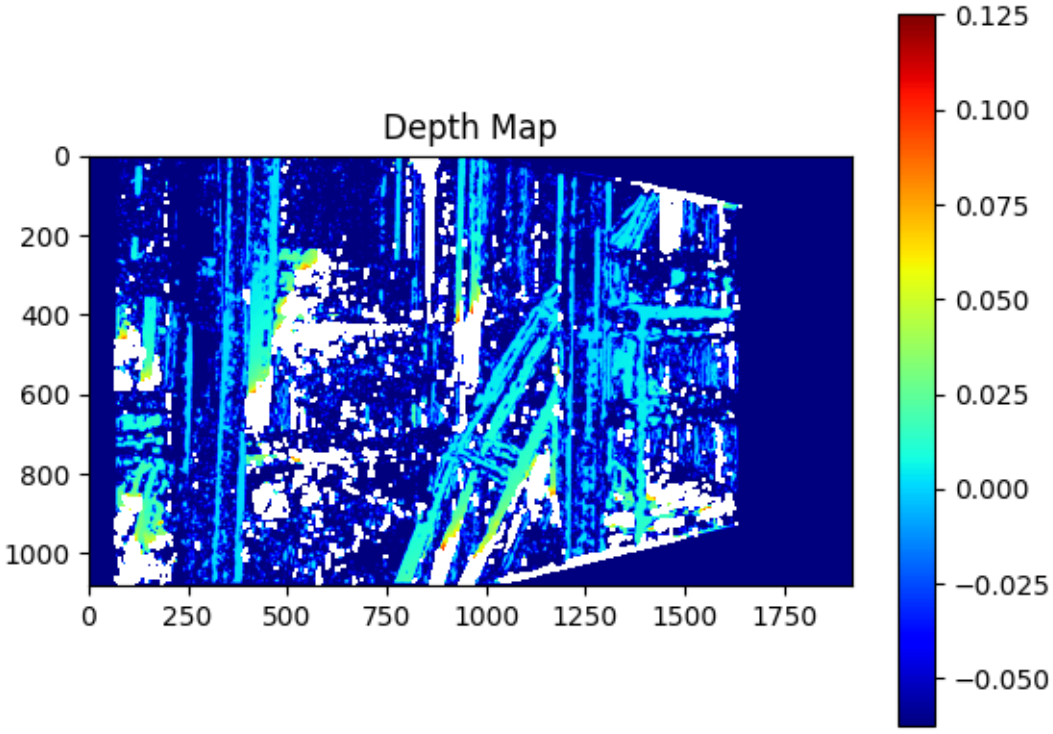


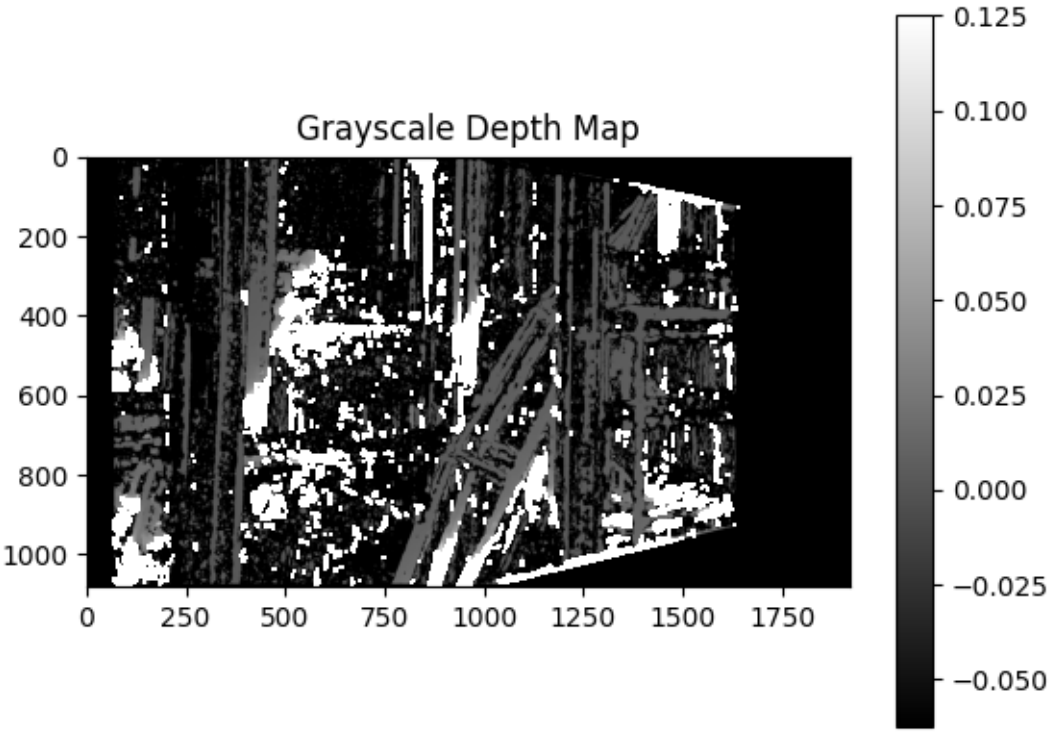
[ ]:
```python
def disparity_and_depth(baseline, f, img):
    """
    Compute the depth map and array depth from the given baseline, focal length, and disparity map.

    Args:
    - baseline: Baseline (distance between the two cameras)
    - f: Focal length of the cameras
    - img: Disparity map

    Returns:
    - depthmap: Depth map computed from the disparity map
    - array_depth: Array depth computed from the disparity map
    """
    depthmap = np.zeros((img.shape[0], img.shape[1]))  # Initialize depth map array
    array_depth = np.zeros((img.shape[0], img.shape[1]))  # Initialize array depth array

    for i in range(depthmap.shape[0]):
        for j in range(depthmap.shape[1]):
            depthmap[i][j] = 1 / img[i][j]  # Compute depth from disparity
            array_depth[i][j] = baseline * f / img[i][j]  # Compute array depth

    return depthmap, array_depth  # Return depth map and array depth

# Convert rectified images to grayscale
rectified_image1_gray_ = cv2.cvtColor(rectified_image1, cv2.COLOR_BGR2GRAY)
rectified_image2_gray_ = cv2.cvtColor(rectified_image2, cv2.COLOR_BGR2GRAY)

# Compute disparity map using StereoBM
stereo = cv2.StereoBM_create(numDisparities=64, blockSize=13)
disparity_ = stereo.compute(rectified_image1_gray_, rectified_image2_gray_)

# Normalize the disparity map
```

```
disparity_normalized = cv2.normalize(disparity_, None, alpha=0, beta=1,␣
 ↪norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

# Compute depth map and array depth
depthmap, _ = disparity_and_depth(baseline, focal_length, disparity_)

# Display the disparity map
plt.imshow(disparity_normalized, cmap='gray')
plt.title('Disparity Map')
plt.colorbar()
plt.show()

# Display the depth map (color)
plt.imshow(depthmap, cmap='jet')
plt.title('Depth Map')
plt.colorbar()
plt.show()

# Display the depth map (grayscale)
plt.imshow(depthmap, cmap='gray')
plt.title('Grayscale Depth Map')
plt.colorbar()
plt.show()
```

```
<ipython-input-158-640fbfd0f55d>:19: RuntimeWarning: divide by zero encountered
in divide
  depthmap[i][j] = 1 / img[i][j]  # Compute depth from disparity
<ipython-input-158-640fbfd0f55d>:20: RuntimeWarning: divide by zero encountered
in divide
  array_depth[i][j] = baseline * f / img[i][j]  # Compute array depth
```


Disparity Map
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
1.0
0.8
0.6
0.4
0.2
0.0


Depth Map
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
0.125
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050


Grayscale Depth Map
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
0.125
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050