sriram21_project3

April 22, 2024

Problem 1: Camera calibration

[ ]:
```
import os
from google.colab import drive
from google.colab.patches import cv2_imshow
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
import glob
# Mount your Google Drive for file access
drive.mount('/content/drive/')
import random



# Define the path to your project folder
path_to_folder = "ENPM673"
%cd /content/drive/My\ Drive/{path_to_folder}

# Specify the location for storing output frames
input_folder_path = os.path.join('/content/drive/My Drive/ENPM673/', 'checker␣
↪board')
```

```
Mounted at /content/drive/
/content/drive/My Drive/ENPM673
```

1 Camera Calibration pipeline

1. Capture images of a checkerboard with a 14x14 grid configuration. Ensure that the board occupies more than 60% of each image to enhance the accuracy of the calibration process.
2. Retrieve the images and detect the corners of the checkerboard. This can be efficiently performed using the `findChessboardCorners` function, which internally employs corner detection algorithms to identify the transitions between the black and white squares of the board.
3. With the refined corner coordinates in the image and their corresponding 3D world coordinates (arranged in a sequence of X, Y increments based on the square size of 9mm), input these

data points into a camera calibration function. This function will output the camera matrix, distortion coefficients, as well as translation and rotation vectors for the camera.

4. After obtaining the camera matrix, distortion coefficients, translation vectors, and rotation vectors through calibration, you can reproject the 3D world points back onto the 2D image plane. Reprojection involves using the derived camera parameters to map the known 3D coordinates of the object points onto the 2D plane of the image.

5. Validate the accuracy of the camera calibration by comparing these reprojected image points with the initially detected image points. This step confirms the precision of the calibration process by showing how closely the reprojected points align with the detected points on the image.

Link to the calibrated images = https://drive.google.com/drive/folders/1rZh6Bd_jDrYMZJlih595s-3BFQPWRj8Z?usp=sharing

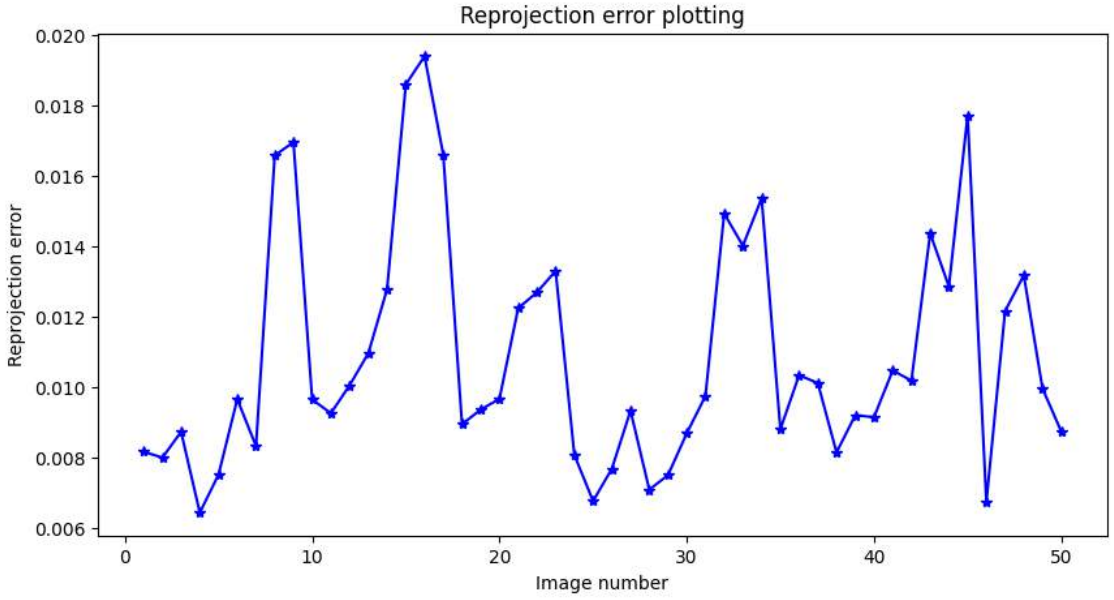
[ ]:
```
import cv2 as cv
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)



sq_size = 9
corner_x = 14
corner_y = 14

checker_board_world_points = []
checker_board_image_points = []

# Simulating the world coordinates by taking camera coordinates and scaling it␣
 ↪up with checkbox size
# grid_points = np.mgrid[0:corner_x, 0:corner_y].T.reshape(-1,2)
checker_board_world = np.zeros((corner_x * corner_y, 3), np.float32)
index = 0
#
for i in range (corner_x):
  for j in range (corner_y):
    checker_board_world[index][0] = i * sq_size
    checker_board_world[index][1] = j * sq_size
    index += 1

# checker_board_world[:, :2] = grid_points * sq_size
# print(checker_board_world)

checker_board_world_points = []
checker_board_image_points = []

# Retrieve the list of TIFF images in the input folder
images_list = glob.glob(os.path.join(input_folder_path, '*.tiff'))
for image in images_list:
  img = cv.imread(image)
```

```
  gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)


  ret,corners = cv.findChessboardCorners(gray, (corner_x,corner_y), cv.
 ↪CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE)
  if ret == True:

        corners2 = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1), criteria)
        checker_board_world_points.append(checker_board_world)


        checker_board_image_points.append(corners2)

        cv.drawChessboardCorners(img, (corner_x,corner_y), corners2, ret)



        # cv2_imshow(img)
        cv.waitKey(1000)


cv.destroyAllWindows()

ret, mtx, dist_coeff, rot_vecs, trans_vecs = cv.
 ↪calibrateCamera(checker_board_world_points,checker_board_image_points, gray.
 ↪shape[::-1], None, None)

reprojection_error = []
for i in range(len(checker_board_image_points)):
    checker_board_image_points2 , _ = cv.
 ↪projectPoints(checker_board_world_points[i], rot_vecs[i], trans_vecs[i],␣
 ↪mtx, dist_coeff)
    error = cv.norm(checker_board_image_points[i], checker_board_image_points2,␣
 ↪cv.NORM_L2)/len(checker_board_image_points2)
    #print("Reprojection error for image  =", error)
    reprojection_error.append(error)

#print("\n")

mean = np.mean(reprojection_error, )
print("Calibration matrix", mtx)
print("Distortion matrix",dist_coeff)
print("Mean reprojection error = ",mean)
print("\n")
image_num = list(range(1,len(reprojection_error)+1))
plt.figure(figsize=(10, 5))
plt.title('Reprojection error plotting')
```

```
plt.xlabel('Image number')
plt.ylabel('Reprojection error')
plt.plot(image_num, reprojection_error, color='blue',marker='*')
plt.show()
```

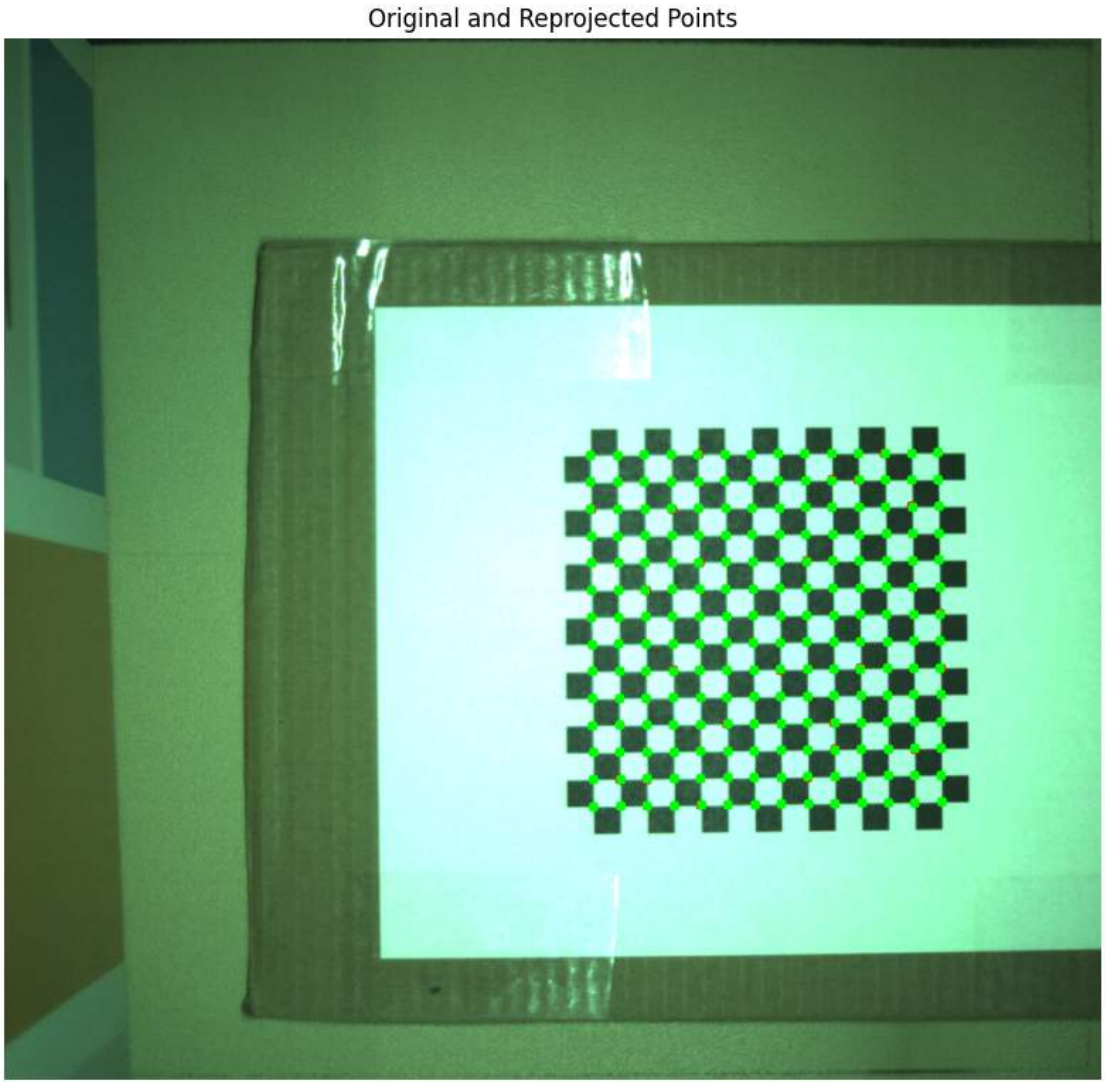
```
Reprojection error for image  = 0.008166640537694076
Reprojection error for image  = 0.007999636855576918
Reprojection error for image  = 0.008731598950925732
Reprojection error for image  = 0.006434758141908699
Reprojection error for image  = 0.007518075792474473
Reprojection error for image  = 0.009658895745934941
Reprojection error for image  = 0.008328392723308459
Reprojection error for image  = 0.016585990833359758
Reprojection error for image  = 0.016960751253470536
Reprojection error for image  = 0.009649153817763106
Reprojection error for image  = 0.009260966687853473
Reprojection error for image  = 0.010055699087605846
Reprojection error for image  = 0.010966905633945033
Reprojection error for image  = 0.012787581950515785
Reprojection error for image  = 0.01860321250461957
Reprojection error for image  = 0.01940523706337789
Reprojection error for image  = 0.01660491069646354
Reprojection error for image  = 0.008948190582835965
Reprojection error for image  = 0.00936278067365248
Reprojection error for image  = 0.00966867582847877
Reprojection error for image  = 0.012245598358232223
Reprojection error for image  = 0.012694127175759306
Reprojection error for image  = 0.013304608797362734
Reprojection error for image  = 0.008074868686563161
Reprojection error for image  = 0.006765326267936118
Reprojection error for image  = 0.007680906785325669
Reprojection error for image  = 0.009331243703575241
Reprojection error for image  = 0.0070900972615019235
Reprojection error for image  = 0.007501084298129083
Reprojection error for image  = 0.008688730121626655
Reprojection error for image  = 0.009749287392883213
Reprojection error for image  = 0.01493358003824238
Reprojection error for image  = 0.014019614986099502
Reprojection error for image  = 0.015369562597409556
Reprojection error for image  = 0.00880680117715179
Reprojection error for image  = 0.010340971927387744
Reprojection error for image  = 0.01011728218468845
Reprojection error for image  = 0.008147258341408564
Reprojection error for image  = 0.00920373086166997
Reprojection error for image  = 0.009149060648308659
Reprojection error for image  = 0.010475560943945965
Reprojection error for image  = 0.010186447322289576
```

```
Reprojection error for image  = 0.014371120755466664
Reprojection error for image  = 0.012861990317992554
Reprojection error for image  = 0.01772118821257876
Reprojection error for image  = 0.006740613083031284
Reprojection error for image  = 0.012163484568418809
Reprojection error for image  = 0.013179362843211691
Reprojection error for image  = 0.009960128068250487
Reprojection error for image  = 0.008741603750160545
Calibration matrix [[2.99385756e+03 0.00000000e+00 5.33349133e+02]
 [0.00000000e+00 2.99584448e+03 5.36314802e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion matrix [[-4.37863768e-01  7.30843653e+00 -2.12707584e-03
4.70379040e-03
  -6.23640468e+01]]
Mean reprojection error =  0.010906265936767467
```



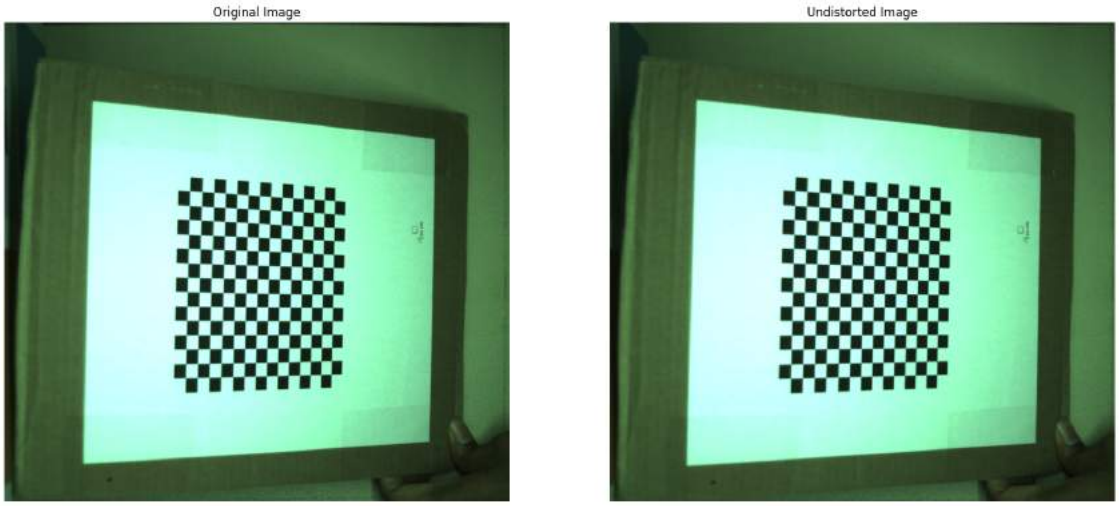
1. Now computing the reprojected points from the 3D world coordinates of the checkerboard using the previously obtained camera calibration parameters (rot_vecs, trans_vecs, mtx, dist_coeff). This transformation simulates how the 3D points should appear in the 2D image plane according to the camera model.
2. Now marking the original detected corners with red circles and the reprojected points with green circles on the color image. This color coding helps differentiate between the observed corners (from image processing) and the expected corners (from camera model predictions).
3. Finally converting the image from BGR to RGB color space for accurate color representation in matplotlib, then displays the image with both sets of points.

[ ]:
```python
import matplotlib.pyplot as plt



# Select the first image to display
img = cv.imread(images_list[0])
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
original_corners = checker_board_image_points[0]
# using
reprojected_points, _ = cv.projectPoints(checker_board_world_points[0],␣
 ↪rot_vecs[0], trans_vecs[0], mtx, dist_coeff)

# Draw original corners in red and reprojected points in green
for corner in original_corners:
    x, y = int(corner[0][0]), int(corner[0][1])
    cv.circle(img, (x, y), 5, (0, 0, 255), -1)  # Red

for point in reprojected_points:
    x, y = int(point[0][0]), int(point[0][1])
    cv.circle(img, (x, y), 5, (0, 255, 0), -1)  # Green

# Convert BGR image to RGB for plotting
img_rgb = cv.cvtColor(img, cv.COLOR_BGR2RGB)

plt.figure(figsize=(10, 10))
plt.imshow(img_rgb)
plt.title('Original and Reprojected Points')
plt.axis('off')
plt.show()
```



[ ]:
```
img = cv.imread(images_list[20])
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

# Undistort the image
undistorted_img = cv.undistort(img, mtx, dist_coeff, None, mtx)

# Plot original and undistorted images
plt.figure(figsize=(20, 10))

# Original image
plt.subplot(1, 2, 1)
plt.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
plt.title('Original Image')
```

```
plt.axis('off')

# Undistorted image
plt.subplot(1, 2, 2)
plt.imshow(cv.cvtColor(undistorted_img, cv.COLOR_BGR2RGB))
plt.title('Undistorted Image')
plt.axis('off')

plt.show()
```



Discuss the significance of the reprojection error and its implications for the accuracy of the calibration: - The reprojection error directly reflects the accuracy of the camera calibration. A low reprojection error indicates that the camera parameters accurately represent the imaging process of the camera, allowing for precise mapping of 3D world coordinates to 2D image coordinates. - It helps validate the camera model used during calibration, including the assumptions made about the lens distortion and the imaging geometry. High errors may suggest that the model does not adequately capture the camera's characteristics or that there were errors in the measurement or estimation processes. - In applications such as 3D reconstruction, augmented reality, and robotics, the quality of the reconstruction or the accuracy of object placement and navigation depends significantly on the precision of the camera calibration. Lower reprojection errors translate into more accurate 3D spatial assessments. - A high reprojection error can lead to poor system performance in applications relying on precise vision measurements, such as robotic surgery, machine vision in manufacturing, or precision agriculture. - In multi-stage vision systems where initial outputs feed into subsequent processing stages, errors introduced by poor calibration can propagate, amplifying the impact on the final outcome. - Significant reprojection errors might indicate the need for frequent recalibrations, especially if the camera setup is prone to mechanical disturbances or environmental changes that could alter its parameters.

2 Problem 2: Stereo Vision

The main objective of this pipeline is to compute depth images from 3 datasets given, each dataset consists of 2 images and information about calibration matrix, baseline, focal lengths.

General pipeline: 1. Identify matching features between the two images in each dataset using any feature matching algorithms. 2. Estimate the Fundamental matrix using RANSAC method based on the matched features. 3. Compute the Essential matrix from the Fundamental matrix considering calibration parameters. 4. Decompose the Essential matrix into rotation and translation matrices. 5. Apply perspective transformation to rectify images and ensure horizontal epipolar lines. 6. Print the homography matrices (H1 and H2) for rectification. 7. Visualize epipolar lines and feature points on both rectified images.

8. Calculate the disparity map representing the pixel-wise differences between the two images.
9. Rescale the disparity map and save it as grayscale and color images using heat map conversion.
10. Utilize the disparity information to compute depth values for each pixel.
11. Generate a depth image representing the spatial dimensions of the scene.
12. Save the depth image as grayscale and color using heat map conversion for visualization.

Dataset 1 : The following dataset consists of classroom images

[ ]:
```
import os
from google.colab import drive
from google.colab.patches import cv2_imshow
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
import glob
# Mount your Google Drive for file access
drive.mount('/content/drive/')
import random

path_to_folder = "ENPM673/classroom"
%cd /content/drive/My\ Drive/{path_to_folder}
```

```
Mounted at /content/drive/
/content/drive/My Drive/ENPM673/classroom
```

1. features function: This function processes two images to detect keypoints using the Scale-Invariant Feature Transform (SIFT) and then matches these keypoints using the FLANN-based matcher. It calculates the fundamental matrix using matched points, which is useful in determining the epipolar geometry between the two images.
2. point_match function: This function performs 3D triangulation of matched point pairs (inliers) using the projection matrices of the two camera views. It projects the 2D points into 3D space to reconstruct their positions.
3. decompose_essential_matrix function : This function decomposes the essential matrix to extract possible rotation and translation matrices that describe the camera movement between two views. It uses the essential matrix derived from the fundamental matrix and the camera intrinsic parameters.

```
[ ]: def features(image_1, image_2, x):
        image_1 = cv.imread(image_1)
        image_2 = cv.imread(image_2)
        #x = 200
        gray1 = cv.cvtColor(image_1, cv.COLOR_BGR2GRAY)
        gray2 = cv.cvtColor(image_2, cv.COLOR_BGR2GRAY)

        sift = cv.SIFT_create()
        keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
        keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

        image_with_keypoints1 = cv.drawKeypoints(image_1, keypoints1, None)
        image_with_keypoints2 = cv.drawKeypoints(image_2, keypoints2, None)

        features_both_images = np.concatenate((image_with_keypoints1,␣
     ↪image_with_keypoints2), axis=1)

        #print(features_both_images, "detected features in both the images")

        cv2_imshow(features_both_images)

        index_params = dict(algorithm=1, trees=5)
        search_params = dict(checks=50)
        flann = cv.FlannBasedMatcher(index_params, search_params)
        matches = flann.knnMatch(descriptors1, descriptors2, k=2)

        # Need to draw only good matches, so create a mask
        matchesMask = [[0, 0] for i in range(len(matches))]
        good_matches = []
        for i, (m, n) in enumerate(matches):
            if m.distance < 0.7*n.distance:
               # matchesMask[i] = [1, 0]
                good_matches.append(m)

        points1 = np.float32([keypoints1[m.queryIdx].pt for m in good_matches])
        points2 = np.float32([keypoints2[m.trainIdx].pt for m in good_matches])

        # Compute Fundamental Matrix
        F, mask = cv.findFundamentalMat(points1, points2, cv.FM_RANSAC)

        # We select only inlier points
        inlier_points1 = points1[mask.ravel() == 1]
        inlier_points2 = points2[mask.ravel() == 1]
```

```
    matched_features = cv.drawMatches(image_1, keypoints1,image_2, keypoints2,␣
 ↪good_matches, None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    cv2_imshow(matched_features)
    cv.waitKey(0)
    cv.destroyAllWindows()
    return inlier_points1, inlier_points2, F



img_1 = ('im0.png')
img_2 = ('im1.png')
inlier_points1, inlier_points2, F_matrix = features(img_1, img_2, 250)

print("funda matrix", F_matrix)
def point_match_function(points1, points2, projection_matrix_l,␣
 ↪projection_matrix_r):

     points1_homo = cv.convertPointsToHomogeneous(points1)
     points2_homo = cv.convertPointsToHomogeneous(points2)

     points1_homo = points1_homo.reshape(-1, 3)[:, :2].T  # Take only x, y from␣
 ↪Nx3 and transpose to 2xN
     points2_homo = points2_homo.reshape(-1, 3)[:, :2].T  # Same for points2

    # Triangulate points
     points3D_homo = cv.triangulatePoints(projection_matrix_l,␣
 ↪projection_matrix_r, points1_homo, points2_homo)

    # Convert from homogeneous to 3D points
     points3D = cv.convertPointsFromHomogeneous(points3D_homo.T)

    # points3D is a n x 1 x 3 array, we need to reshape it to n x 3
     return points3D.reshape(-1, 3)


best_points_calc1 = inlier_points1
best_points_calc2 = inlier_points2

K_matrix = np.array([[1746.24, 0, 14.88 ],
                     [0, 1746.24, 534.11],
                     [0, 0, 1]])



E = K_matrix.T @ F_matrix @ K_matrix
```

```
U, S, Vt = np.linalg.svd(E)

E = U @ np.diag([1, 1, 0]) @ Vt.T

print("\n")
print("Essential matrix = \n ", E)
print("\n")

U, S, Vt = np.linalg.svd(E)

W = np.array([[0, -1, 0 ],
              [1, 0, 0],
              [0, 0, 1]])
def decompose_essential_matrix(E):
    retval, R, t, mask = cv.recoverPose(E, best_points_calc1,␣
 ↪best_points_calc2, focal=1746.24, pp=(14.88, 534.11))
    return R, t

Rot_mat, T_mat = decompose_essential_matrix(E)

print("Rotation matrix", Rot_mat)
print("Translation matrix", T_mat)
```

```
funda matrix [[-3.00307938e-08 -3.86035426e-05  2.92561580e-02]
```

```
 [ 3.90141509e-05 -3.68883223e-06 -4.07864237e-01]
 [-2.90919811e-02  4.07106675e-01  1.00000000e+00]]


Essential matrix =
  [[ 0.07799008 -0.13784358  0.04953845]
 [-0.05325756  0.44007104  0.896143  ]
 [-0.46041553  0.77025148 -0.40941624]]


Rotation matrix [[-0.87119336 -0.49090847  0.0055686 ]
 [ 0.44859444 -0.79138972  0.41528946]
 [-0.19946218  0.36429546  0.90967228]]
Translation matrix [[0.9861352 ]
 [0.02070811]
 [0.16464671]]
```

1. drawlines_rectified function : Draws epipolar lines and points on rectified images for visualization, helping validate the stereo rectification and epipolar geometry correctness.
2. rectification function : Computes homography matrices for rectifying stereo images, aligning corresponding epipolar lines horizontally to simplify disparity estimation.
3. rectified_epilines function : Applies rectification transformations to images and points, then calculates and visualizes epipolar lines on these rectified images.
4. drawlines function : Similar to drawlines_rectified but for unrectified images, this function visualizes epipolar lines and points on original images to assess the raw epipolar geometry before rectification.
5. un_rectified_epiplines function : Computes and visualizes epipolar lines on unrectified images, providing a before-and-after comparison relative to rectification results.

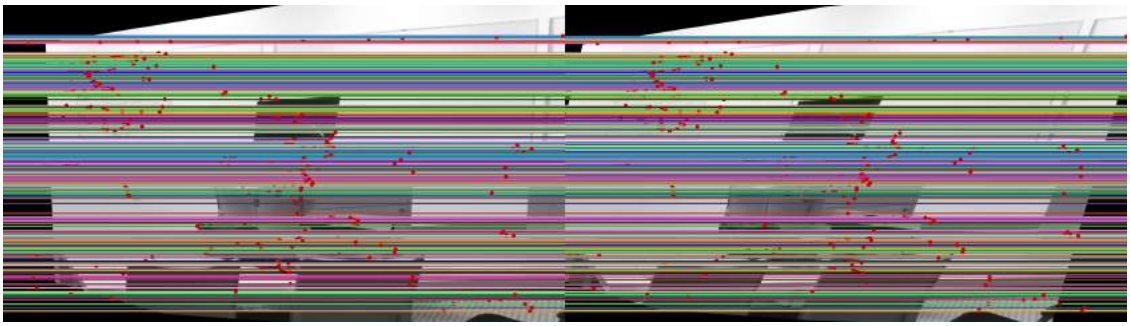
[ ]:
```python
def drawlines_rectified(img1src, img2src, lines, pts1src, pts2src):

    img1color = cv.cvtColor(img1src, cv.COLOR_GRAY2BGR)
    img2color = cv.cvtColor(img2src, cv.COLOR_GRAY2BGR)

    np.random.seed(0)

    r, c = img1src.shape

    for r, pt1, pt2 in zip(lines, pts1src, pts2src):

        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0, y0 = map(int, [0, pt1[1]])
        x1, y1 = map(int, [c, pt1[1]])
        img1color = cv.line(img1color, (x0, y0), (x1, y1), color, 3)
        img1color = cv.circle((img1color), (int(pt1[0]), int(pt1[1])), 7, (0,␣
 ↪0, 200), -1)
```

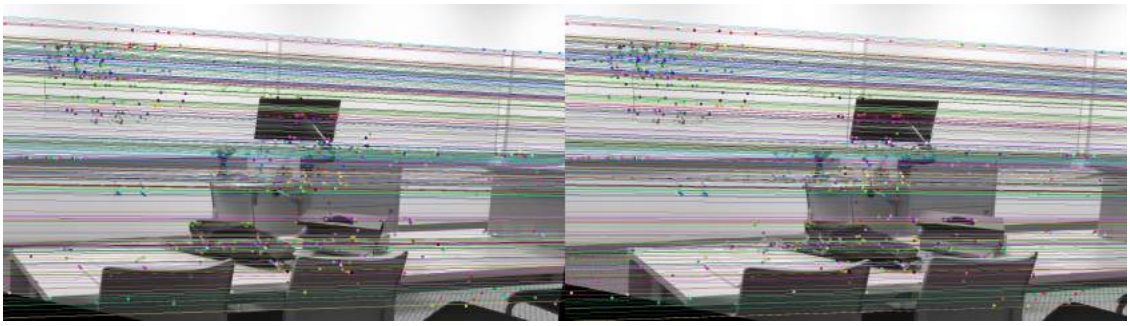
```
        x0, y0 = map(int, [0, pt2[1]])
        x1, y1 = map(int, [c, pt2[1]])
        img2color = cv.line((img2color), (x0, y0), (x1, y1), (color), 3)
        img2color = cv.circle(img2color, (int(pt2[0]), int(pt2[1])), 7, (0, 0,␣
 ↪200), -1)

    return img1color, img2color

def rectification(imag1, imag2):
    h1, w1 = imag1.shape[:2]
    h2, w2 = imag2.shape[:2]

    retval, H1, H2 = cv.stereoRectifyUncalibrated(np.
 ↪float32(best_points_calc1), np.float32(best_points_calc2), F_matrix,␣
 ↪imgSize=(w1, h1))

    print("\n")
    print("Homography matrix for left image  = \n", H1)
    print("\n")
    print("Homography matrix for right image  = \n", H1)
    print("\n")

    return H1, H2, (w1, h1), (w2, h2)
# Convert images to grayscale if not already done in 'features'
gray1 = cv.cvtColor(cv.imread(img_1), cv.COLOR_BGR2GRAY)
gray2 = cv.cvtColor(cv.imread(img_2), cv.COLOR_BGR2GRAY)

# Rectification of Images
H1, H2, image1_size, image2_size = rectification(gray1, gray2)
img1_rectified = cv.warpPerspective(gray1, H1, image1_size)
img2_rectified = cv.warpPerspective(gray2, H2, image2_size)

def rectified_epilines(imag1, imag2):

    img1_rectified = cv.warpPerspective(imag1, H1, image1_size)
    img2_rectified = cv.warpPerspective(imag2, H2, image2_size)

    points1_rectified = cv.perspectiveTransform(best_points_calc1.reshape(-1,␣
 ↪1, 2), H1).reshape(-1,2)
    points2_rectified = cv.perspectiveTransform(best_points_calc2.reshape(-1,␣
 ↪1, 2), H2).reshape(-1,2)

    lines1 = cv.computeCorrespondEpilines(
        points2_rectified.reshape(-1, 1, 2), 2, F_matrix)
    lines1 = lines1.reshape(-1, 3)
```

```
    img5, img6 = drawlines_rectified(img1_rectified, img2_rectified, lines1,␣
 ↪points1_rectified, points2_rectified)

    lines2 = cv.computeCorrespondEpilines(
        points1_rectified.reshape(-1, 1, 2), 1, F_matrix)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines_rectified(img2_rectified, img1_rectified, lines2,␣
 ↪points2_rectified, points1_rectified)

    result = np.concatenate((img3, img5), axis=1)
    horizontal_concat = cv.hconcat([img1_rectified, img2_rectified])

    return result, horizontal_concat, img1_rectified, img2_rectified

# Draw Epipolar Lines on Rectified Images
result, horizontal_concat, img1_rectified, img2_rectified =␣
 ↪rectified_epilines(img1_rectified, img2_rectified)
cv2_imshow(result)
cv.waitKey(0)
cv.destroyAllWindows()
def drawlines(img1, img2, lines, pts1, pts2):
    r, c = img1.shape

    color1 = cv.cvtColor(img1, cv.COLOR_GRAY2BGR)
    color2 = cv.cvtColor(img2, cv.COLOR_GRAY2BGR)

    np.random.seed(0)
    for r, pt1, pt2 in zip(lines, pts1, pts2):

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

        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [c, -(r[2]+r[0]*c)/r[1]])

        image_colour_1 = cv.line(color1, (x0, y0), (x1, y1), color, 1)
        imag1_with_lines = cv.circle(image_colour_1, (int(pt1[0]),␣
 ↪int(pt1[1])), 5, color, -1)
        imag2_with_lines = cv.circle(image_colour_1, (int(pt2[0]),␣
 ↪int(pt2[1])), 5, color, -1)

    return imag1_with_lines, imag2_with_lines

def un_rectified_epiplines(imag1, imag2, src_pts1, src_pts2):

    lines1 = cv.computeCorrespondEpilines(
        src_pts2.reshape(-1, 1, 2), 2, F_matrix)
    lines1 = lines1.reshape(-1, 3)
```

```
    img5, img6 = drawlines(imag1, imag2, lines1, src_pts1, src_pts2)

    lines2 = cv.computeCorrespondEpilines(
        src_pts1.reshape(-1, 1, 2), 1, F_matrix)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines(imag2, imag1, lines2, src_pts2, src_pts1)

    result = np.concatenate((img3, img5), axis=1)

    return result

# Optionally, draw epipolar lines on unrectified images
before_rectified_epiplines = un_rectified_epiplines(gray1, gray2,␣
 ↪inlier_points1, inlier_points2)
cv2_imshow(before_rectified_epiplines)
cv.waitKey(0)
cv.destroyAllWindows()
```

```
Homography matrix for left image  =
 [[-3.68005997e-01  5.16099122e-02 -1.67449645e+01]
 [ 3.21286096e-02 -4.07820523e-01 -3.24258079e+01]
 [ 4.30410050e-05 -3.11272853e-06 -4.50679357e-01]]


Homography matrix for right image  =
 [[-3.68005997e-01  5.16099122e-02 -1.67449645e+01]
 [ 3.21286096e-02 -4.07820523e-01 -3.24258079e+01]
 [ 4.30410050e-05 -3.11272853e-06 -4.50679357e-01]]
```





1. Function: StereoBM_create:

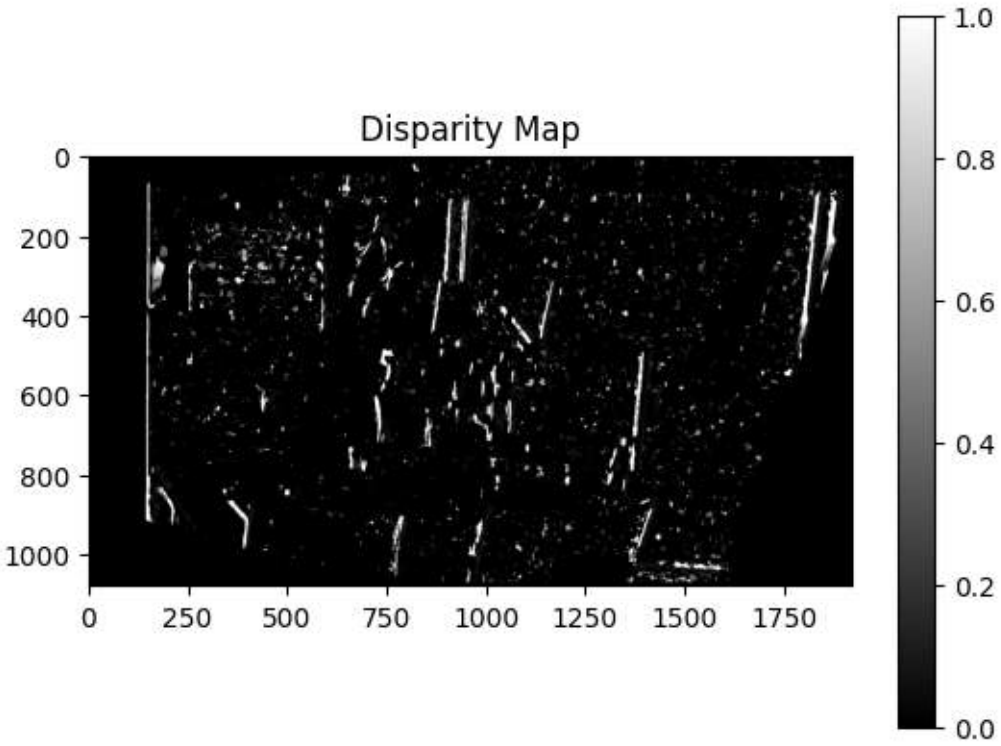
- This function initializes a block matching stereo disparity algorithm. StereoBM (Block Matching) is one of the simplest and fastest disparity algorithms provided by OpenCV but is often sensitive to noise and texture.
- Normalizes the disparity values to a 0-1 range for visualization purposes. Normalization makes it easier to observe the disparity variations in a visible format.

2. disparity_to_depth function:

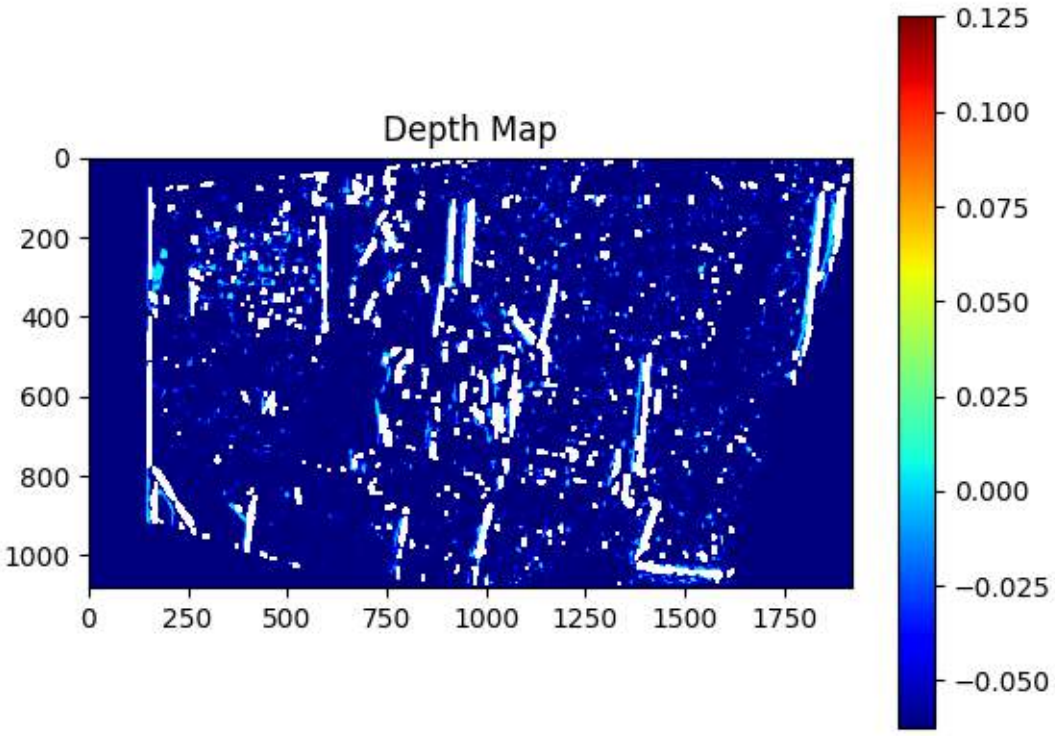
- Converts disparity values to depth measurements using the camera's baseline and focal length. This function directly relates the disparity map to physical distances, providing a depth map that represents real-world measurements. **Visualization**: Shows disparity maps and depth maps in both grayscale and heatmap color mappings to highlight depth variations visually.

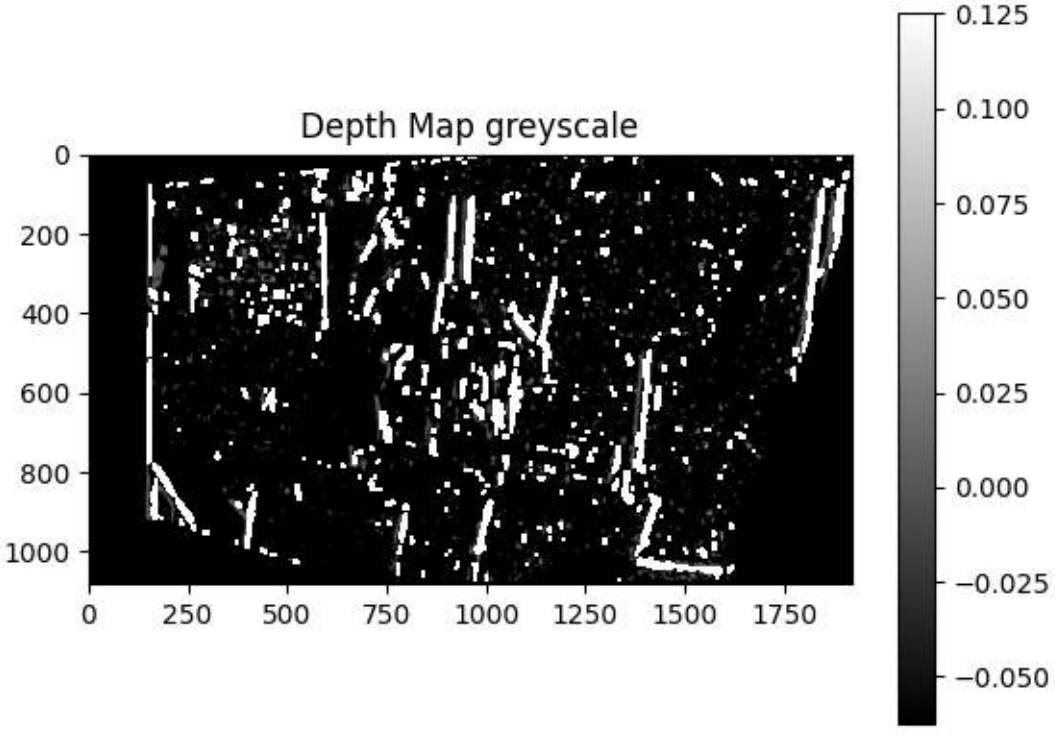
[ ]:
```
# Assuming the camera has square pixels, so focal lengths are equal
focal_length = 1746.24

baseline = 678.37
# Create StereoBM object
stereo = cv.StereoBM_create(numDisparities=16, blockSize= 17)

# Compute disparity map
disparity = stereo.compute(img1_rectified, img2_rectified)

# Normalize disparity map for visualization
disparity_normalized = cv.normalize(disparity, None, alpha=0, beta=1,␣
 ↪norm_type=cv.NORM_MINMAX, dtype=cv.CV_32F)

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

def disparity_to_depth(baseline, f, img):
    """This is used to compute the depth values from the disparity map"""
```

```
    depth_map = np.zeros((img.shape[0], img.shape[1]))
    depth_array = np.zeros((img.shape[0], img.shape[1]))

    for i in range(depth_map.shape[0]):
        for j in range(depth_map.shape[1]):
            depth_map[i][j] = 1/img[i][j]
            depth_array[i][j] = baseline*f/img[i][j]


    return depth_map, depth_array




depth_map,_ = disparity_to_depth(baseline ,focal_length, disparity)

# Show the depth map as heatmap
plt.imshow(depth_map, cmap='jet')
plt.title('Depth Map')
plt.colorbar()
plt.show()

plt.imshow(depth_map, cmap='gray')
plt.title('Depth Map greyscale')
plt.colorbar()
plt.show()
```




```
<ipython-input-71-505e5db91e65>:29: RuntimeWarning: divide by zero encountered
in divide
  depth_map[i][j] = 1/img[i][j]
<ipython-input-71-505e5db91e65>:30: RuntimeWarning: divide by zero encountered
in divide
  depth_array[i][j] = baseline*f/img[i][j]
```


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


Depth Map greyscale
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

Dataset 2

[ ]:
```
import os
from google.colab import drive
from google.colab.patches import cv2_imshow
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
import glob
# Mount your Google Drive for file access
drive.mount('/content/drive/')
import random



path_to_folder = "ENPM673/storageroom"
%cd /content/drive/My\ Drive/{path_to_folder}
```

```
Drive already mounted at /content/drive/; to attempt to forcibly remount, call
drive.mount("/content/drive/", force_remount=True).
/content/drive/My Drive/ENPM673/storageroom
```

[14]:
```
def features(image_1, image_2, x = 150):
    image_1 = cv.imread(image_1)
    image_2 = cv.imread(image_2)
    #x = 200
    gray1 = cv.cvtColor(image_1, cv.COLOR_BGR2GRAY)
    gray2 = cv.cvtColor(image_2, cv.COLOR_BGR2GRAY)

    sift = cv.SIFT_create()
    keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

    image_with_keypoints1 = cv.drawKeypoints(image_1, keypoints1, None)
    image_with_keypoints2 = cv.drawKeypoints(image_2, keypoints2, None)

    features_both_images = np.concatenate((image_with_keypoints1,␣
 ↪image_with_keypoints2), axis=1)

    #print(features_both_images, "detected features in both the images")

    cv2_imshow(features_both_images)

    index_params = dict(algorithm=1, trees=5)
    search_params = dict(checks=30)
    flann = cv.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(descriptors1, descriptors2, k=2)
```

```
    # Need to draw only good matches, so create a mask
    matchesMask = [[0, 0] for i in range(len(matches))]
    good_matches = []
    for i, (m, n) in enumerate(matches):
        if m.distance < 0.5*n.distance:
           # matchesMask[i] = [1, 0]
            good_matches.append(m)

    points1 = np.float32([keypoints1[m.queryIdx].pt for m in good_matches])
    points2 = np.float32([keypoints2[m.trainIdx].pt for m in good_matches])

    # Compute Fundamental Matrix
    F, mask = cv.findFundamentalMat(points1, points2, cv.FM_RANSAC)

    # We select only inlier points
    inlier_points1 = points1[mask.ravel() == 1]
    inlier_points2 = points2[mask.ravel() == 1]



    matched_features = cv.drawMatches(image_1, keypoints1,image_2, keypoints2,␣
 ↪good_matches, None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    cv2_imshow(matched_features)
    cv.waitKey(0)
    cv.destroyAllWindows()
    return inlier_points1, inlier_points2, F



img_1 = ('im0.png')
img_2 = ('im1.png')
inlier_points1, inlier_points2, F_matrix = features(img_1, img_2, 150)

print("funda matrix", F_matrix)
def triangulation(points1, points2, projection_matrix_1, projection_matrix_r):

     points1_homo = cv.convertPointsToHomogeneous(points1)
     points2_homo = cv.convertPointsToHomogeneous(points2)

     points1_homo = points1_homo.reshape(-1, 3)[:, :2].T  # Take only x, y from␣
 ↪Nx3 and transpose to 2xN
     points2_homo = points2_homo.reshape(-1, 3)[:, :2].T  # Same for points2

    # Triangulate points
     points3D_homo = cv.triangulatePoints(projection_matrix_1,␣
 ↪projection_matrix_r, points1_homo, points2_homo)
```

```
    # Convert from homogeneous to 3D points
     points3D = cv.convertPointsFromHomogeneous(points3D_homo.T)

    # points3D is a n x 1 x 3 array, we need to reshape it to n x 3
     return points3D.reshape(-1, 3)


best_points_calc1 = inlier_points1
best_points_calc2 = inlier_points2

K_matrix = np.array([[1742.11, 0, 804.90 ],
                  [0, 1742.11, 541.22],
                  [0, 0, 1]])




E = K_matrix.T @ F_matrix @ K_matrix

U, S, Vt = np.linalg.svd(E)

E = U @ np.diag([1, 1, 0]) @ Vt.T

print("\n")
print("Essential matrix = \n ", E)
print("\n")

U, S, Vt = np.linalg.svd(E)

W = np.array([[0, -1, 0 ],
              [1, 0, 0],
              [0, 0, 1]])
def decompose_essential_matrix(E):
    retval, R, t, mask = cv.recoverPose(E, best_points_calc1,␣
 ↪best_points_calc2, focal=1746.24, pp=(14.88, 534.11))
    return R, t

Rot_mat, T_mat = decompose_essential_matrix(E)

print("Rotation matrix", Rot_mat)
print("Translation matrix", T_mat)
```

```
funda matrix [[-2.61137688e-11  3.48594073e-06 -8.54414506e-03]
 [-3.58672226e-06  3.94596931e-07  4.59878477e-01]
 [ 8.78866175e-03 -4.60941855e-01  1.00000000e+00]]


Essential matrix =
  [[-0.01670634  0.00314559  0.00995008]
 [ 0.95892482 -0.19139772  0.20885041]
 [ 0.20733902 -0.02716115 -0.97780183]]


Rotation matrix [[ 0.18310223  0.98308814 -0.00335907]
 [-0.20471516  0.04147011  0.9779427 ]
 [ 0.96154317 -0.17837584  0.20884632]]
Translation matrix [[0.99980598]
 [0.01454693]
 [0.0132811 ]]
```

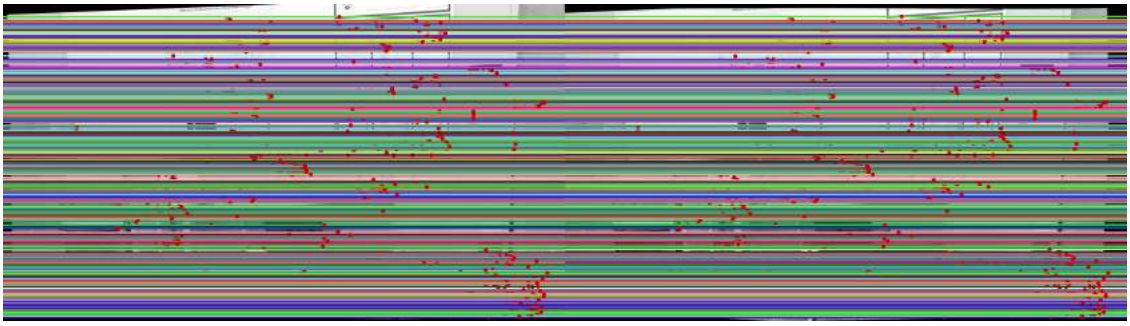
[15]:
```python
def drawlines_rectified(img1src, img2src, lines, pts1src, pts2src):

    img1color = cv.cvtColor(img1src, cv.COLOR_GRAY2BGR)
    img2color = cv.cvtColor(img2src, cv.COLOR_GRAY2BGR)

    np.random.seed(0)
```

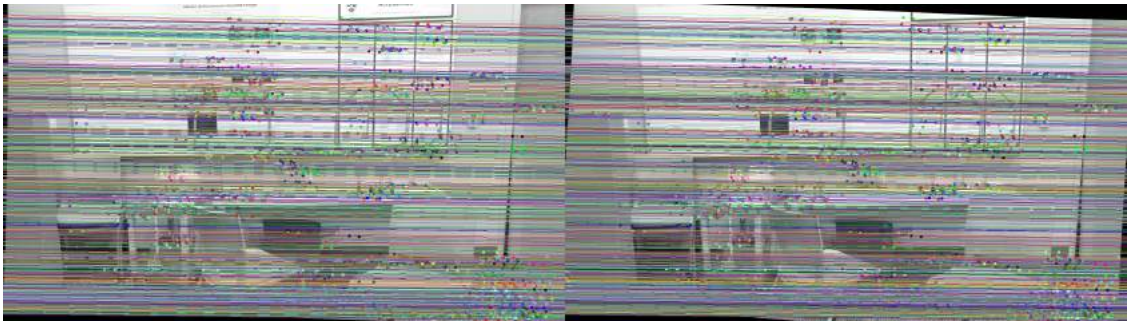
```
    r, c = img1src.shape

    for r, pt1, pt2 in zip(lines, pts1src, pts2src):

        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0, y0 = map(int, [0, pt1[1]])
        x1, y1 = map(int, [c, pt1[1]])
        img1color = cv.line(img1color, (x0, y0), (x1, y1), color, 3)
        img1color = cv.circle((img1color), (int(pt1[0]), int(pt1[1])), 7, (0,␣
 ↪0, 200), -1)

        x0, y0 = map(int, [0, pt2[1]])
        x1, y1 = map(int, [c, pt2[1]])
        img2color = cv.line((img2color), (x0, y0), (x1, y1), (color), 3)
        img2color = cv.circle(img2color, (int(pt2[0]), int(pt2[1])), 7, (0, 0,␣
 ↪200), -1)

    return img1color, img2color

def rectification(imag1, imag2):
    h1, w1 = imag1.shape[:2]
    h2, w2 = imag2.shape[:2]

    retval, H1, H2 = cv.stereoRectifyUncalibrated(np.
 ↪float32(best_points_calc1), np.float32(best_points_calc2), F_matrix,␣
 ↪imgSize=(w1, h1))

    print("\n")
    print("Homography matrix for left image  = \n", H1)
    print("\n")
    print("Homography matrix for right image  = \n", H1)
    print("\n")

    return H1, H2, (w1, h1), (w2, h2)
# Convert images to grayscale if not already done in 'features'
gray1 = cv.cvtColor(cv.imread(img_1), cv.COLOR_BGR2GRAY)
gray2 = cv.cvtColor(cv.imread(img_2), cv.COLOR_BGR2GRAY)

# Rectification of Images
H1, H2, image1_size, image2_size = rectification(gray1, gray2)
img1_rectified = cv.warpPerspective(gray1, H1, image1_size)
img2_rectified = cv.warpPerspective(gray2, H2, image2_size)

def rectified_epilines(imag1, imag2):

    img1_rectified = cv.warpPerspective(imag1, H1, image1_size)
```

```
    img2_rectified = cv.warpPerspective(imag2, H2, image2_size)

    points1_rectified = cv.perspectiveTransform(best_points_calc1.reshape(-1,␣
 ↪1, 2), H1).reshape(-1,2)
    points2_rectified = cv.perspectiveTransform(best_points_calc2.reshape(-1,␣
 ↪1, 2), H2).reshape(-1,2)

    lines1 = cv.computeCorrespondEpilines(
        points2_rectified.reshape(-1, 1, 2), 2, F_matrix)
    lines1 = lines1.reshape(-1, 3)
    img5, img6 = drawlines_rectified(img1_rectified, img2_rectified, lines1,␣
 ↪points1_rectified, points2_rectified)

    lines2 = cv.computeCorrespondEpilines(
        points1_rectified.reshape(-1, 1, 2), 1, F_matrix)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines_rectified(img2_rectified, img1_rectified, lines2,␣
 ↪points2_rectified, points1_rectified)

    result = np.concatenate((img3, img5), axis=1)
    horizontal_concat = cv.hconcat([img1_rectified, img2_rectified])

    return result, horizontal_concat, img1_rectified, img2_rectified

# Draw Epipolar Lines on Rectified Images
result, horizontal_concat, img1_rectified, img2_rectified =␣
 ↪rectified_epilines(img1_rectified, img2_rectified)
cv2_imshow(result)
cv.waitKey(0)
cv.destroyAllWindows()
def drawlines(img1, img2, lines, pts1, pts2):
    r, c = img1.shape

    color1 = cv.cvtColor(img1, cv.COLOR_GRAY2BGR)
    color2 = cv.cvtColor(img2, cv.COLOR_GRAY2BGR)

    np.random.seed(0)
    for r, pt1, pt2 in zip(lines, pts1, pts2):

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

        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [c, -(r[2]+r[0]*c)/r[1]])

        image_colour_1 = cv.line(color1, (x0, y0), (x1, y1), color, 1)
        imag1_with_lines = cv.circle(image_colour_1, (int(pt1[0]),␣
 ↪int(pt1[1])), 5, color, -1)
```

```
        imag2_with_lines = cv.circle(image_colour_1, (int(pt2[0]),␣
 ↪int(pt2[1])), 5, color, -1)

    return imag1_with_lines, imag2_with_lines

def un_rectified_epiplines(imag1, imag2, src_pts1, src_pts2):

    lines1 = cv.computeCorrespondEpilines(
        src_pts2.reshape(-1, 1, 2), 2, F_matrix)
    lines1 = lines1.reshape(-1, 3)
    img5, img6 = drawlines(imag1, imag2, lines1, src_pts1, src_pts2)

    lines2 = cv.computeCorrespondEpilines(
        src_pts1.reshape(-1, 1, 2), 1, F_matrix)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines(imag2, imag1, lines2, src_pts2, src_pts1)

    result = np.concatenate((img3, img5), axis=1)

    return result

# Optionally, draw epipolar lines on unrectified images
before_rectified_epiplines = un_rectified_epiplines(gray1, gray2,␣
 ↪inlier_points1, inlier_points2)
cv2_imshow(before_rectified_epiplines)
cv.waitKey(0)
cv.destroyAllWindows()
```

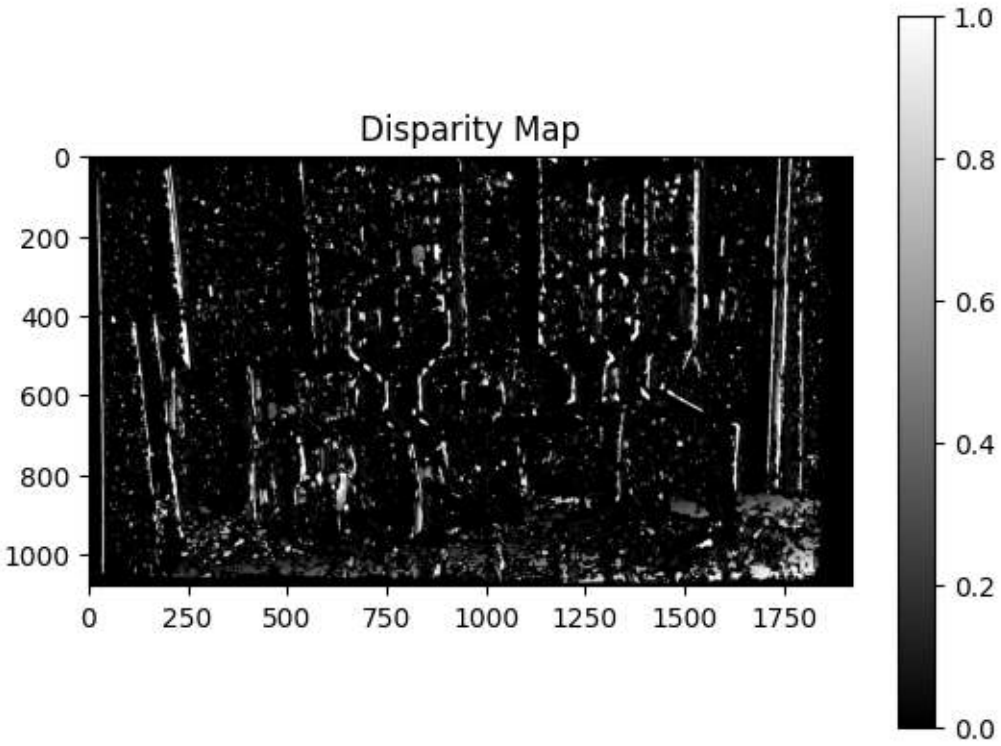
```
Homography matrix for left image  =
 [[ 4.56592537e-01  3.56267064e-03 -1.58165076e+01]
 [-8.85342326e-03  4.60951716e-01  7.29681291e+00]
 [-3.61286274e-06  3.46290415e-07  4.63355628e-01]]


Homography matrix for right image  =
 [[ 4.56592537e-01  3.56267064e-03 -1.58165076e+01]
 [-8.85342326e-03  4.60951716e-01  7.29681291e+00]
 [-3.61286274e-06  3.46290415e-07  4.63355628e-01]]
```



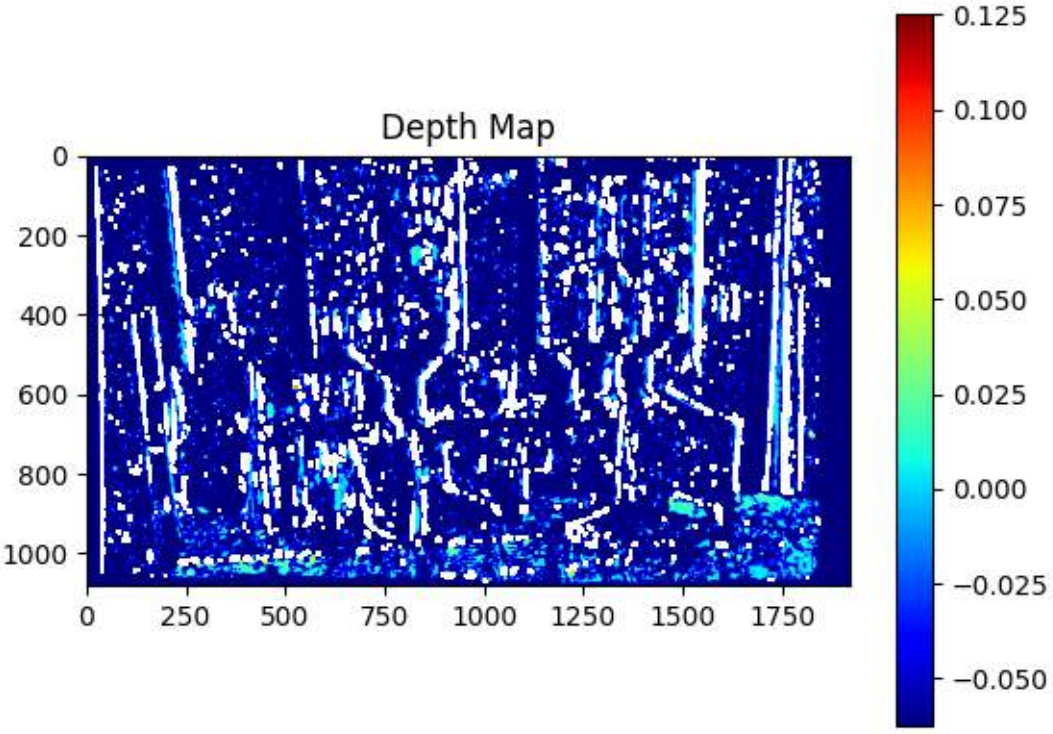


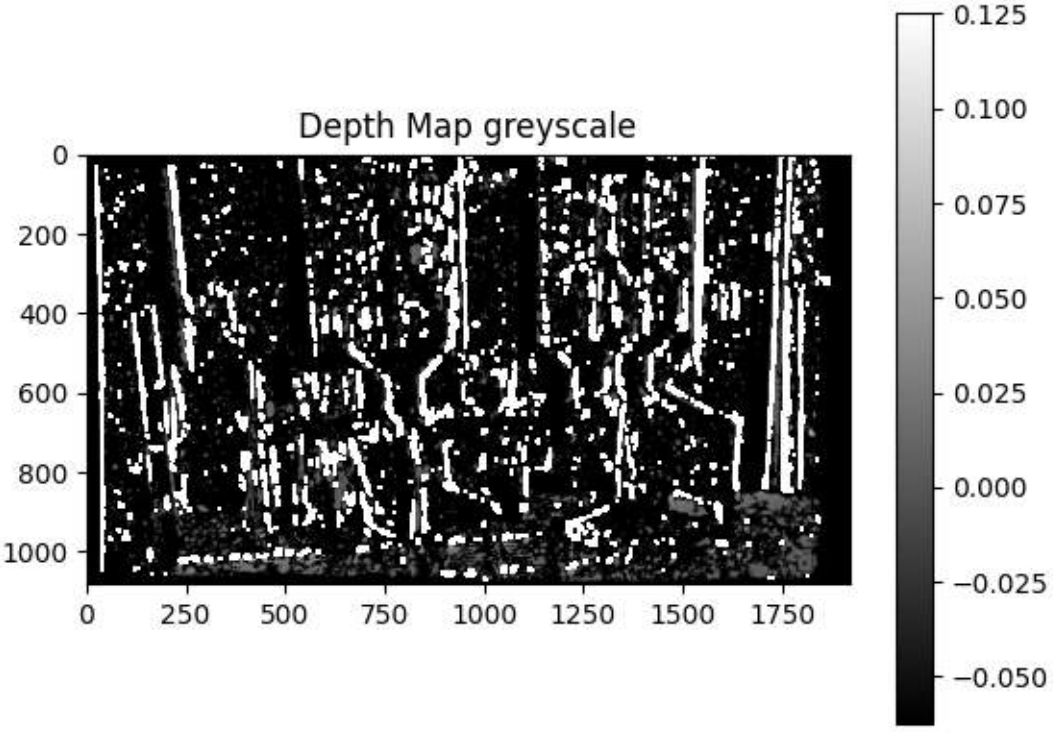
[16]:
```
# Assuming the camera has square pixels, so focal lengths are equal
focal_length = 1742.11

baseline = 221.76
# Create StereoBM object
stereo = cv.StereoBM_create(numDisparities=16, blockSize= 15)

# Compute disparity map
disparity = stereo.compute(img1_rectified, img2_rectified)

# Normalize disparity map for visualization
disparity_normalized = cv.normalize(disparity, None, alpha=0, beta=1,␣
 ↪norm_type=cv.NORM_MINMAX, dtype=cv.CV_32F)

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

def disparity_to_depth(baseline, f, img):
    """This is used to compute the depth values from the disparity map"""
```

```
    # Assumption image intensities are disparity values (x-x')
    depth_map = np.zeros((img.shape[0], img.shape[1]))
    depth_array = np.zeros((img.shape[0], img.shape[1]))

    for i in range(depth_map.shape[0]):
        for j in range(depth_map.shape[1]):
            depth_map[i][j] = 1/img[i][j]
            depth_array[i][j] = baseline*f/img[i][j]
            # if math.isinf(depth_map[i][j]):
            #     depth_map[i][j] = 1

    return depth_map, depth_array




depth_map,_ = disparity_to_depth(baseline ,focal_length, disparity)

# Show the depth map as heatmap
plt.imshow(depth_map, cmap='jet')
plt.title('Depth Map')
plt.colorbar()
plt.show()

plt.imshow(depth_map, cmap='gray')
plt.title('Depth Map greyscale')
plt.colorbar()
plt.show()
```




```
<ipython-input-16-65c750c23b23>:29: RuntimeWarning: divide by zero encountered
in divide
  depth_map[i][j] = 1/img[i][j]
<ipython-input-16-65c750c23b23>:30: RuntimeWarning: divide by zero encountered
in divide
  depth_array[i][j] = baseline*f/img[i][j]
```


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


Depth Map greyscale
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

DATASET 3

[ ]:
```
#traproom
import os
from google.colab import drive
from google.colab.patches import cv2_imshow
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
import glob
# Mount your Google Drive for file access
drive.mount('/content/drive/')
import random



path_to_folder = "ENPM673/traproom"
%cd /content/drive/My\ Drive/{path_to_folder}
```

```
Drive already mounted at /content/drive/; to attempt to forcibly remount, call
drive.mount("/content/drive/", force_remount=True).
/content/drive/My Drive/ENPM673/traproom
```

[ ]:
```
def features(image_1, image_2, x = 200):
    image_1 = cv.imread(image_1)
    image_2 = cv.imread(image_2)
    #x = 200
    gray1 = cv.cvtColor(image_1, cv.COLOR_BGR2GRAY)
    gray2 = cv.cvtColor(image_2, cv.COLOR_BGR2GRAY)

    sift = cv.SIFT_create()
    keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

    image_with_keypoints1 = cv.drawKeypoints(image_1, keypoints1, None)
    image_with_keypoints2 = cv.drawKeypoints(image_2, keypoints2, None)

    features_both_images = np.concatenate((image_with_keypoints1,␣
 ↪image_with_keypoints2), axis=1)

    #print(features_both_images, "detected features in both the images")

    cv2_imshow(features_both_images)

    index_params = dict(algorithm=1, trees=5)
    search_params = dict(checks=50)
    flann = cv.FlannBasedMatcher(index_params, search_params)
    matches = flann.knnMatch(descriptors1, descriptors2, k=2)
```

```
    # Need to draw only good matches, so create a mask
    matchesMask = [[0, 0] for i in range(len(matches))]
    good_matches = []
    for i, (m, n) in enumerate(matches):
        if m.distance < 0.7*n.distance:
           # matchesMask[i] = [1, 0]
            good_matches.append(m)

    points1 = np.float32([keypoints1[m.queryIdx].pt for m in good_matches])
    points2 = np.float32([keypoints2[m.trainIdx].pt for m in good_matches])

    # Compute Fundamental Matrix
    F, mask = cv.findFundamentalMat(points1, points2, cv.FM_RANSAC)

    # We select only inlier points
    inlier_points1 = points1[mask.ravel() == 1]
    inlier_points2 = points2[mask.ravel() == 1]



    matched_features = cv.drawMatches(image_1, keypoints1,image_2, keypoints2,␣
 ↪good_matches, None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    cv2_imshow(matched_features)
    cv.waitKey(0)
    cv.destroyAllWindows()
    return inlier_points1, inlier_points2, F



img_1 = ('im0.png')
img_2 = ('im1.png')
inlier_points1, inlier_points2, F_matrix = features(img_1, img_2, 250)

print("funda matrix", F_matrix)
def triangulation(points1, points2, projection_matrix_1, projection_matrix_r):

     points1_homo = cv.convertPointsToHomogeneous(points1)
     points2_homo = cv.convertPointsToHomogeneous(points2)

     points1_homo = points1_homo.reshape(-1, 3)[:, :2].T  # Take only x, y from␣
 ↪Nx3 and transpose to 2xN
     points2_homo = points2_homo.reshape(-1, 3)[:, :2].T  # Same for points2

    # Triangulate points
     points3D_homo = cv.triangulatePoints(projection_matrix_1,␣
 ↪projection_matrix_r, points1_homo, points2_homo)
```

```
    # Convert from homogeneous to 3D points
     points3D = cv.convertPointsFromHomogeneous(points3D_homo.T)

    # points3D is a n x 1 x 3 array, we need to reshape it to n x 3
     return points3D.reshape(-1, 3)


best_points_calc1 = inlier_points1
best_points_calc2 = inlier_points2

K_matrix = np.array([[1769.02, 0, 1271.89 ],
                  [0, 1769.02, 527.17],
                  [0, 0, 1]])




E = K_matrix.T @ F_matrix @ K_matrix

U, S, Vt = np.linalg.svd(E)

E = U @ np.diag([1, 1, 0]) @ Vt.T

print("\n")
print("Essential matrix = \n ", E)
print("\n")

U, S, Vt = np.linalg.svd(E)

W = np.array([[0, -1, 0 ],
              [1, 0, 0],
              [0, 0, 1]])


def decompose_essential_matrix(E):
    retval, R, t, mask = cv.recoverPose(E, best_points_calc1,␣
 ↪best_points_calc2, focal=1769.02, pp=(1271.89, 527.17))
    return R, t

Rot_mat, T_mat = decompose_essential_matrix(E)

print("Rotation matrix", Rot_mat)
print("Translation matrix", T_mat)
```

```
funda matrix [[ 2.88043980e-08  4.24251748e-05 -1.53486743e-02]
 [-4.16032020e-05  6.32098122e-06  3.94668621e-01]
 [ 1.47924927e-02 -4.00015675e-01  1.00000000e+00]]


Essential matrix =
  [[-0.12071017  0.09304527  0.15048157]
 [ 0.29974466 -0.60057338  0.7409916 ]
 [ 0.5808842  -0.48224552 -0.62010961]]


Rotation matrix [[ 0.65427739  0.75397795  0.05863746]
 [-0.60801694  0.478338    0.63364672]
 [ 0.44970713 -0.45023329  0.7713971 ]]
Translation matrix [[-0.97679421]
 [ 0.01990475]
 [-0.21325307]]
```

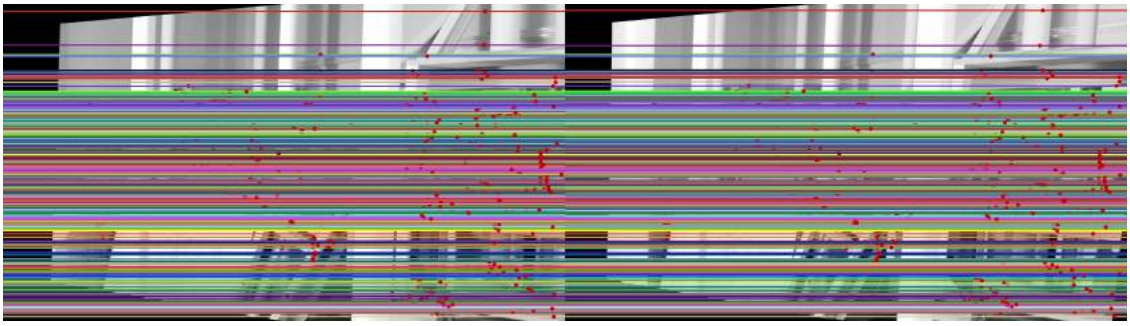
[ ]:
```python
def drawlines_rectified(img1src, img2src, lines, pts1src, pts2src):

    img1color = cv.cvtColor(img1src, cv.COLOR_GRAY2BGR)
    img2color = cv.cvtColor(img2src, cv.COLOR_GRAY2BGR)

    np.random.seed(0)
```

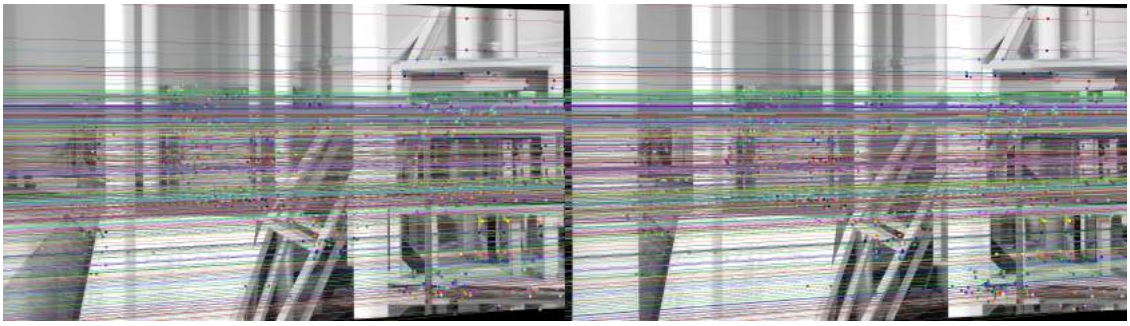
```
    r, c = img1src.shape

    for r, pt1, pt2 in zip(lines, pts1src, pts2src):

        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0, y0 = map(int, [0, pt1[1]])
        x1, y1 = map(int, [c, pt1[1]])
        img1color = cv.line(img1color, (x0, y0), (x1, y1), color, 3)
        img1color = cv.circle((img1color), (int(pt1[0]), int(pt1[1])), 7, (0,␣
 ↪0, 200), -1)

        x0, y0 = map(int, [0, pt2[1]])
        x1, y1 = map(int, [c, pt2[1]])
        img2color = cv.line((img2color), (x0, y0), (x1, y1), (color), 3)
        img2color = cv.circle(img2color, (int(pt2[0]), int(pt2[1])), 7, (0, 0,␣
 ↪200), -1)

    return img1color, img2color

def rectification(imag1, imag2):
    h1, w1 = imag1.shape[:2]
    h2, w2 = imag2.shape[:2]

    retval, H1, H2 = cv.stereoRectifyUncalibrated(np.
 ↪float32(best_points_calc1), np.float32(best_points_calc2), F_matrix,␣
 ↪imgSize=(w1, h1))

    print("\n")
    print("Homography matrix for left image  = \n", H1)
    print("\n")
    print("Homography matrix for right image  = \n", H1)
    print("\n")

    return H1, H2, (w1, h1), (w2, h2)
# Convert images to grayscale if not already done in 'features'
gray1 = cv.cvtColor(cv.imread(img_1), cv.COLOR_BGR2GRAY)
gray2 = cv.cvtColor(cv.imread(img_2), cv.COLOR_BGR2GRAY)

# Rectification of Images
H1, H2, image1_size, image2_size = rectification(gray1, gray2)
img1_rectified = cv.warpPerspective(gray1, H1, image1_size)
img2_rectified = cv.warpPerspective(gray2, H2, image2_size)

def rectified_epilines(imag1, imag2):

    img1_rectified = cv.warpPerspective(imag1, H1, image1_size)
```

```
    img2_rectified = cv.warpPerspective(imag2, H2, image2_size)

    points1_rectified = cv.perspectiveTransform(best_points_calc1.reshape(-1,␣
 ↪1, 2), H1).reshape(-1,2)
    points2_rectified = cv.perspectiveTransform(best_points_calc2.reshape(-1,␣
 ↪1, 2), H2).reshape(-1,2)

    lines1 = cv.computeCorrespondEpilines(
        points2_rectified.reshape(-1, 1, 2), 2, F_matrix)
    lines1 = lines1.reshape(-1, 3)
    img5, img6 = drawlines_rectified(img1_rectified, img2_rectified, lines1,␣
 ↪points1_rectified, points2_rectified)

    lines2 = cv.computeCorrespondEpilines(
        points1_rectified.reshape(-1, 1, 2), 1, F_matrix)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines_rectified(img2_rectified, img1_rectified, lines2,␣
 ↪points2_rectified, points1_rectified)

    result = np.concatenate((img3, img5), axis=1)
    horizontal_concat = cv.hconcat([img1_rectified, img2_rectified])

    return result, horizontal_concat, img1_rectified, img2_rectified

# Draw Epipolar Lines on Rectified Images
result, horizontal_concat, img1_rectified, img2_rectified =␣
 ↪rectified_epilines(img1_rectified, img2_rectified)
cv2_imshow(result)
cv.waitKey(0)
cv.destroyAllWindows()
def drawlines(img1, img2, lines, pts1, pts2):
    r, c = img1.shape

    color1 = cv.cvtColor(img1, cv.COLOR_GRAY2BGR)
    color2 = cv.cvtColor(img2, cv.COLOR_GRAY2BGR)

    np.random.seed(0)
    for r, pt1, pt2 in zip(lines, pts1, pts2):

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

        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [c, -(r[2]+r[0]*c)/r[1]])

        image_colour_1 = cv.line(color1, (x0, y0), (x1, y1), color, 1)
        imag1_with_lines = cv.circle(image_colour_1, (int(pt1[0]),␣
 ↪int(pt1[1])), 5, color, -1)
```

```
        imag2_with_lines = cv.circle(image_colour_1, (int(pt2[0]),␣
 ↪int(pt2[1])), 5, color, -1)

    return imag1_with_lines, imag2_with_lines

def un_rectified_epiplines(imag1, imag2, src_pts1, src_pts2):

    lines1 = cv.computeCorrespondEpilines(
        src_pts2.reshape(-1, 1, 2), 2, F_matrix)
    lines1 = lines1.reshape(-1, 3)
    img5, img6 = drawlines(imag1, imag2, lines1, src_pts1, src_pts2)

    lines2 = cv.computeCorrespondEpilines(
        src_pts1.reshape(-1, 1, 2), 1, F_matrix)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines(imag2, imag1, lines2, src_pts2, src_pts1)

    result = np.concatenate((img3, img5), axis=1)

    return result

# Optionally, draw epipolar lines on unrectified images
before_rectified_epiplines = un_rectified_epiplines(gray1, gray2,␣
 ↪inlier_points1, inlier_points2)
cv2_imshow(before_rectified_epiplines)
cv.waitKey(0)
cv.destroyAllWindows()
```

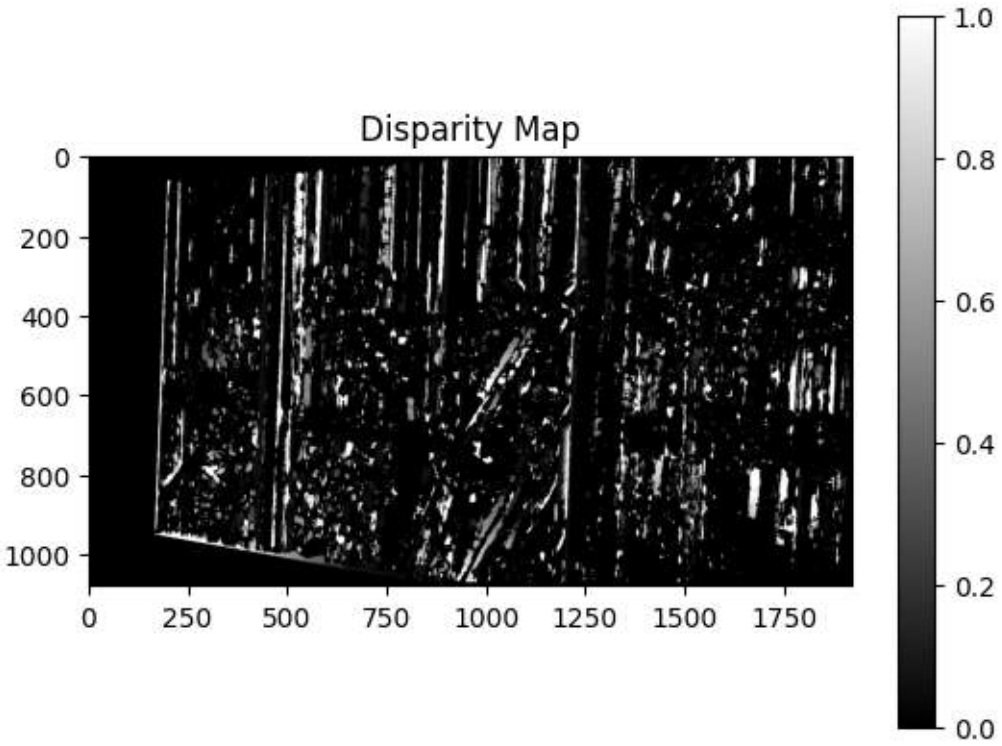
```
Homography matrix for left image  =
 [[ 3.37158324e-01 -5.23354195e-03  3.72340816e+01]
 [-1.65135496e-02  4.01024669e-01  1.50612086e+01]
 [-4.63626360e-05  8.04402652e-06  4.39463697e-01]]


Homography matrix for right image  =
 [[ 3.37158324e-01 -5.23354195e-03  3.72340816e+01]
 [-1.65135496e-02  4.01024669e-01  1.50612086e+01]
 [-4.63626360e-05  8.04402652e-06  4.39463697e-01]]
```



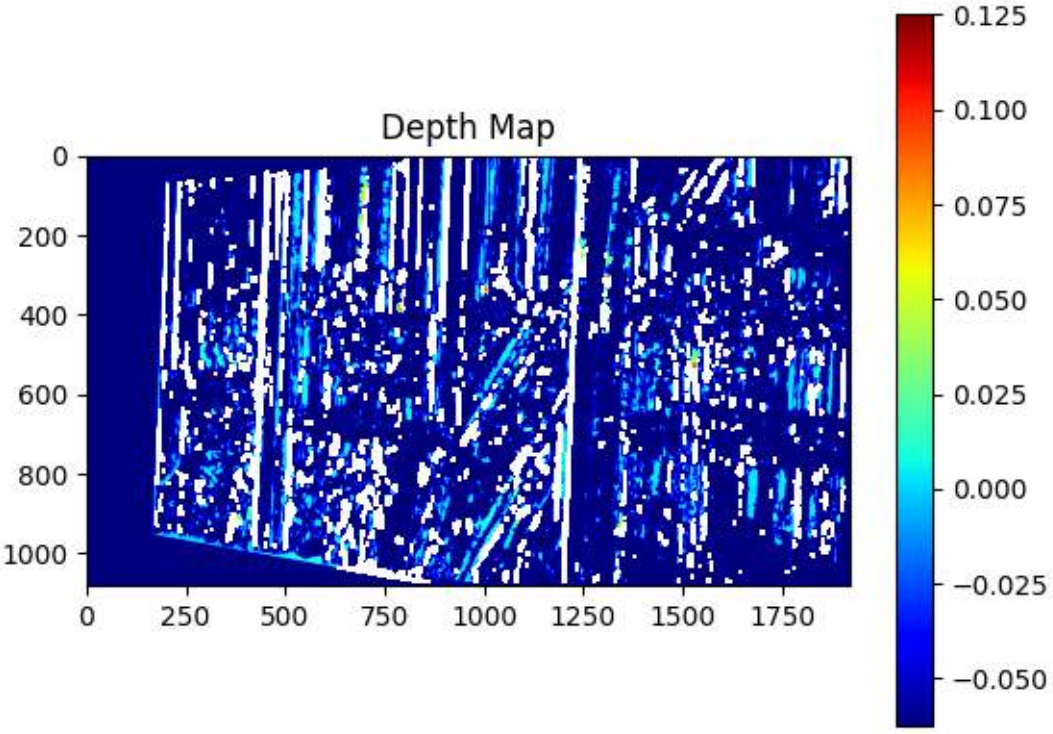


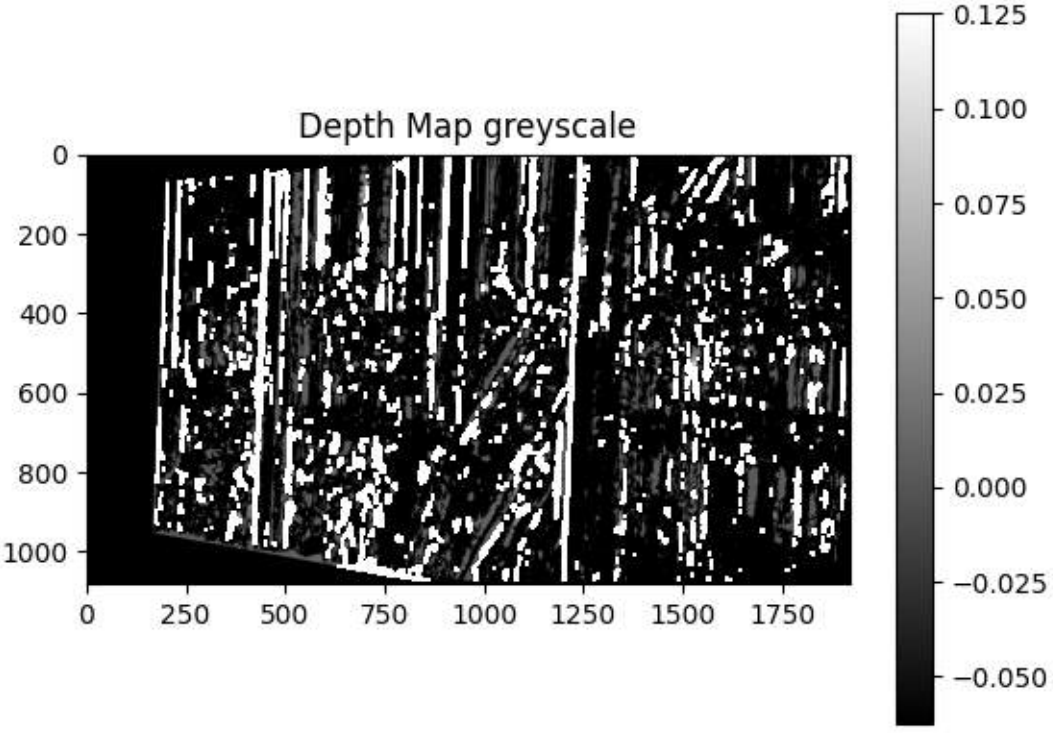
[ ]:
```
# Assuming the camera has square pixels, so focal lengths are equal
focal_length = 1769.02

baseline = 295.44
# Create StereoBM object
stereo = cv.StereoBM_create(numDisparities=16, blockSize= 17)

# Compute disparity map
disparity = stereo.compute(img1_rectified, img2_rectified)

# Normalize disparity map for visualization
disparity_normalized = cv.normalize(disparity, None, alpha=0, beta=1,␣
 ↪norm_type=cv.NORM_MINMAX, dtype=cv.CV_32F)

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

def disparity_to_depth(baseline, f, img):
    """This is used to compute the depth values from the disparity map"""
```

```
    # Assumption image intensities are disparity values (x-x')
    depth_map = np.zeros((img.shape[0], img.shape[1]))
    depth_array = np.zeros((img.shape[0], img.shape[1]))

    for i in range(depth_map.shape[0]):
        for j in range(depth_map.shape[1]):
            depth_map[i][j] = 1/img[i][j]
            depth_array[i][j] = baseline*f/img[i][j]
            # if math.isinf(depth_map[i][j]):
            #     depth_map[i][j] = 1

    return depth_map, depth_array




depth_map,_ = disparity_to_depth(baseline ,focal_length, disparity)

# Show the depth map as heatmap
plt.imshow(depth_map, cmap='jet')
plt.title('Depth Map')
plt.colorbar()
plt.show()

plt.imshow(depth_map, cmap='gray')
plt.title('Depth Map greyscale')
plt.colorbar()
plt.show()
```




```
<ipython-input-60-0d105d70cb4b>:29: RuntimeWarning: divide by zero encountered
in divide
  depth_map[i][j] = 1/img[i][j]
<ipython-input-60-0d105d70cb4b>:30: RuntimeWarning: divide by zero encountered
in divide
  depth_array[i][j] = baseline*f/img[i][j]
```


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


Depth Map greyscale
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