Computer Vision - Coursework

Constantin Eulenstein

November 2020

1 Exercise 1

I propose the SIFT Descriptor. Even though we do not have changes of intensity (lighting), clutter/mess nor occlusion of salient features (since we only look at two frames), the SIFT Descriptor is a good choice, because it is invariant to scale (estimates the scale using scale-space extrema detection and calculates the gradient smoothing with this scale), translation and rotation (rotates the gradient orientations using dominant orientation in neighborhood). Further it uses the Difference of Gaussian, which is more efficient to compute than the Laplacian of Gaussian. In general, it fulfills our requirements of Robustness, Distinctiveness, Compactness and Efficiency.

I will focus on detecting corners since they are the best features to detect. When analyzing our frame for salient features, we will use the intensity changes of a pre-specified window to detect specific features. Detecting corners, any movement of the window will result in significant changes in intensity.

2 Exercise 2

The choice depends on the video sequence. If we have small displacement between frames (when moving camera), then Lukas-Kanade tracker will work well. The two frames fulfil the necessary assumptions: brightness constancy, temporal persistence, spatial coherence. I'd utilise the Sobel filter to calculate I_x, I_y and compute I_t. Then, I'd define a pixel window of size n, to create the set of equations, which will be solved for our displacement vector by the LK algorithm using Least Square. This has to be performed for each salient feature. If we have large displacements, I'd use the Pyramid LK tracker: we can apply the LK tracker to low resolution images first and up-sample the found displacements to our original size to ensure tracking. Then, I'd evaluate my approach using error metrics (like ROC). For improving our tracker, we could also try incorporating temporal information to make predictions and use the Kalman filter.

3 Exercise 3

All following sections are evaluations of my implementation in code. Please refer to this code in the appendix.

3 (a) Using OpenCV in Jupyter Lab, I am applying the SIFT descriptor both on the first frame and on the second frame to find salient features. I chose the SIFT descriptor due to the reasons I mentioned before. SURF might also have been an option, but I believe it is not necessary in this case, because we only have 2 frames that are not very large. Therefore, we do not need the additional computational efficiency of SURF. I apply SIFT on the gray scale images in order to have one intensity value.

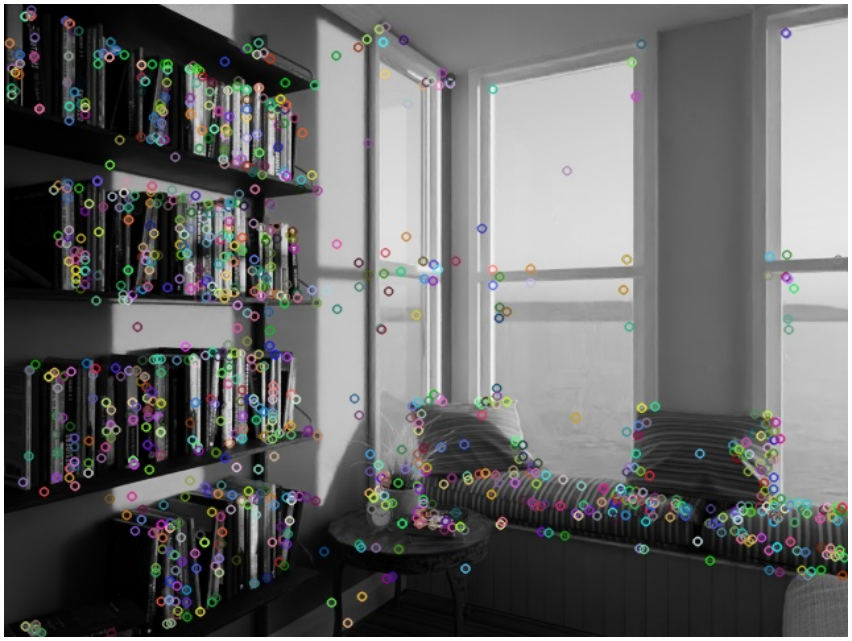


Figure 1: Detected Salient Features Using SIFT - Frame 1

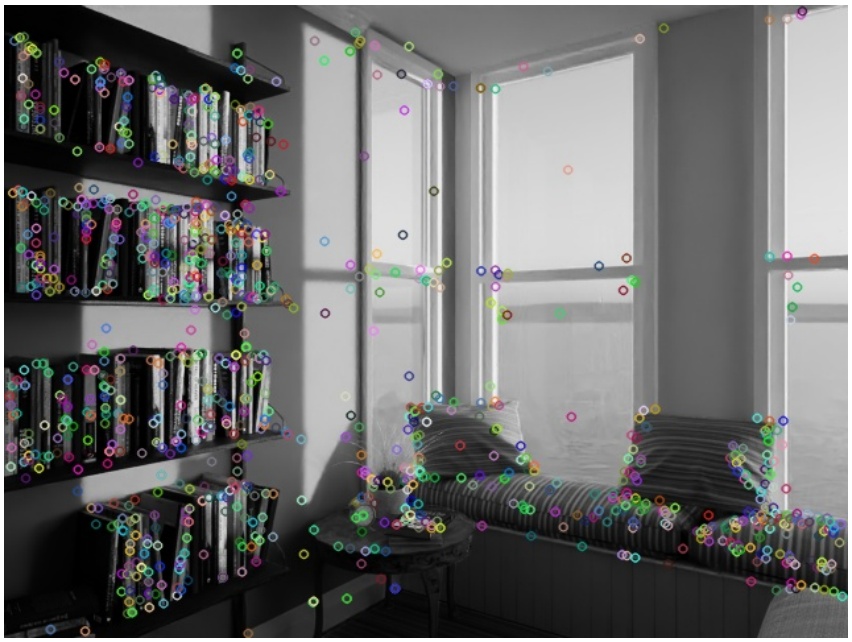


Figure 2: Detected Salient Features Using SIFT - Frame 2

3 (b) To find the matching features, I first apply SIFT Descriptors on both of the images and then use Brute-Force Matching. Brute-Force Matching takes the descriptor of one feature in the first frame and compares it to all other features in the second frame to find the ones with the smallest distance. To find the distance the L2 norm was used. I select the 2 features with the smallest distance and check whether the distance of one is less than 0.75 of the distance of the other. This Ratio Test with threshold of 0.75 is performed to obtain only features that are distinctive. I decided for 0.75, because it gave me a good amount of matches, even though one might reduce this threshold to reduce the amount of mismatched features. However, this might come at the cost of not matching other correctly matched features. This results in below image, where both frames are plotted side by side and matches features are connected through lines. One can see, that most of the time the algorithm performed very well and found the correct matches.

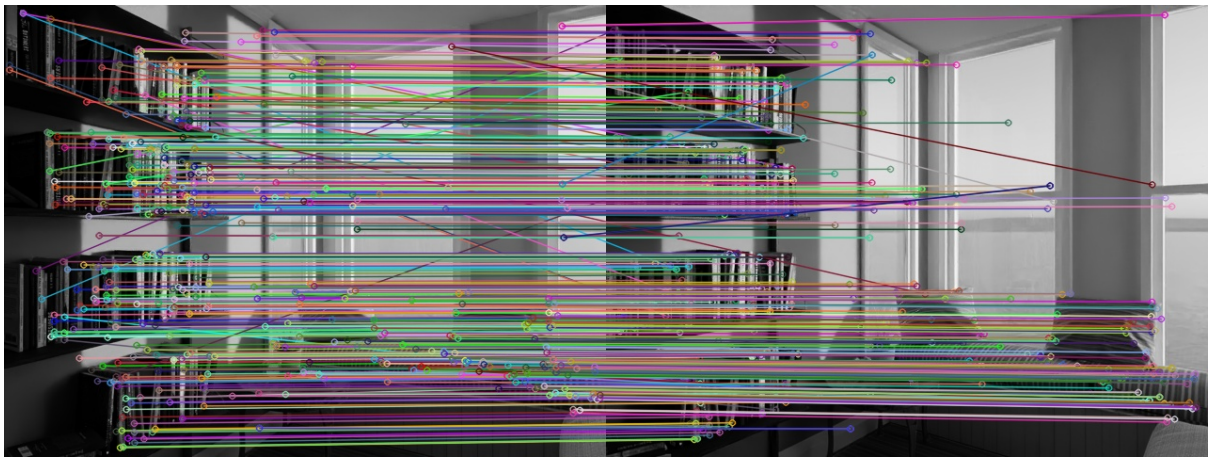
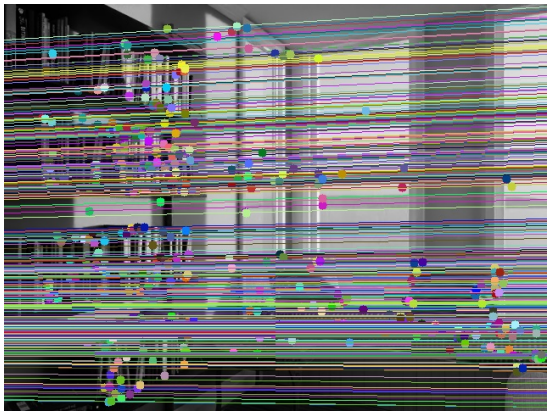


Figure 3: Corresponding features between frames 1 and 2

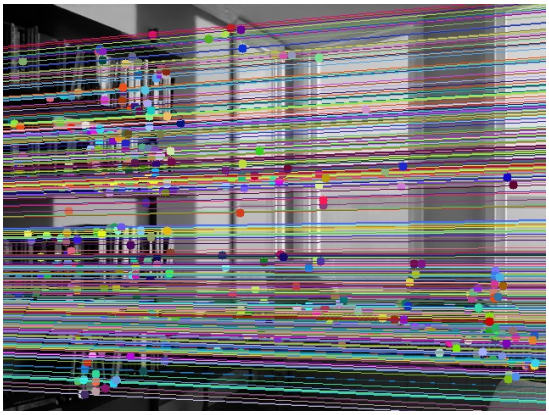
3 (c) After finding many matched features (we need a minimum of 8) we can estimate the fundamental matrix $\mathbf{F}$ by solving a system of equations (generated from the epipolar constraint) using least squares. The more points we found, the more correct our fundamental matrix will be, since we can remove potential outliers using a naive algorithm. With OpenCV, one quickly finds the fundamental matrix $\mathbf{F}$:

$$\mathbf{F} = \begin{bmatrix} 3.69842074e-07 & 2.80479875e-05 & -8.81313330e-03 \\ -1.44397504e-05 & 2.32175973e-07 & -8.18441166e-02 \\ 5.49995702e-03 & 7.75042240e-02 & 1.00000000e+00 \end{bmatrix}$$

3 (d) For finding the correctly matched points that meet the epipolar constraints, I first find the epipolar lines between all previously matched features. The fundamental matrix maps a point in one image to a line (the epipolar line) in the other image.



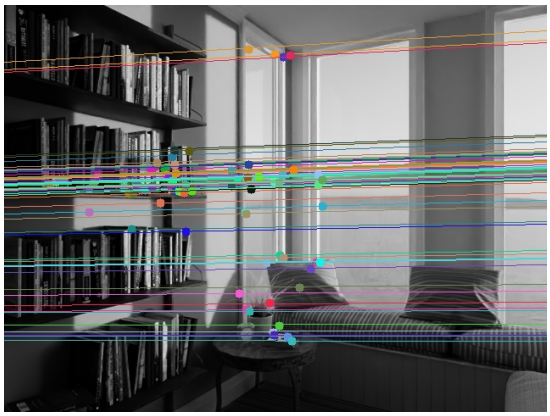
(a) Epipolar lines - Frame 1



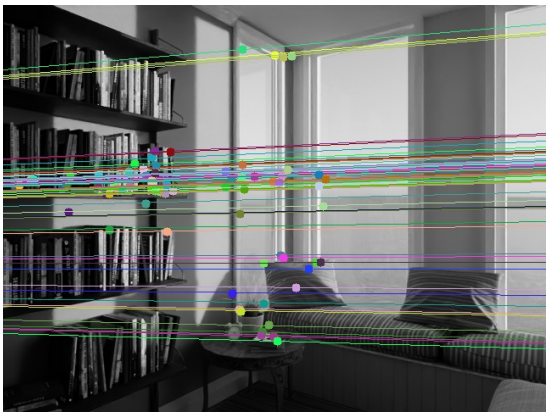
(b) Epipolar lines - Frame 2

Figure 4: Found Epipolar lines for Frames 1 and 2

Since these are still a lot of salient features and some of them are matched incorrectly, we can check for all found points $\mathbf{x}$ and $\mathbf{x}'$ that fulfil our epipolar constraint $\mathbf{x}'^{\top}\mathbf{F}\mathbf{x} = 0$. Instead of 0, I used a threshold of 0.1. I decided for 0.1 because a lower threshold, will miss important matches like the corners of the left window or the pillow corners. A higher threshold like 0.2 on the other hand, still resulted in some incorrect matches. The features that fulfil this constraint (with a threshold of 0.1) and their epipolar lines are shown in below figure.



(a) Epipolar lines after meeting the epipolar constraint - Frame 1



(b) Epipolar lines after meeting the epipolar constraint - Frame 2

Figure 5: Found Epipolar lines meeting epipolar constraint for Frames 1 and 2

3 (e) To find the distance between the individual points we first need to perform stereo image rectification on the two image frames. This is done by rotating both cameras. Using OpenCV's function stereoRectify() we compute the matrix $\mathbf{Q}$, which in turn gives us our focal length $f = Q[2,3] = 452.74$ and our baseline $t = -1/Q[3,2] = -0.25$. Once we rectified our two images (see code),we get a window as depicted in below figure 6 (I plotted horizontal lines every 25 pixels). In this window we can then manually read our x-coordinates (in pixel) x_{left} and x_{right} and calculate our disparity for each point from them ($disparity = x_{left} - (x_{right} - image\_width)$; our image width is 640).

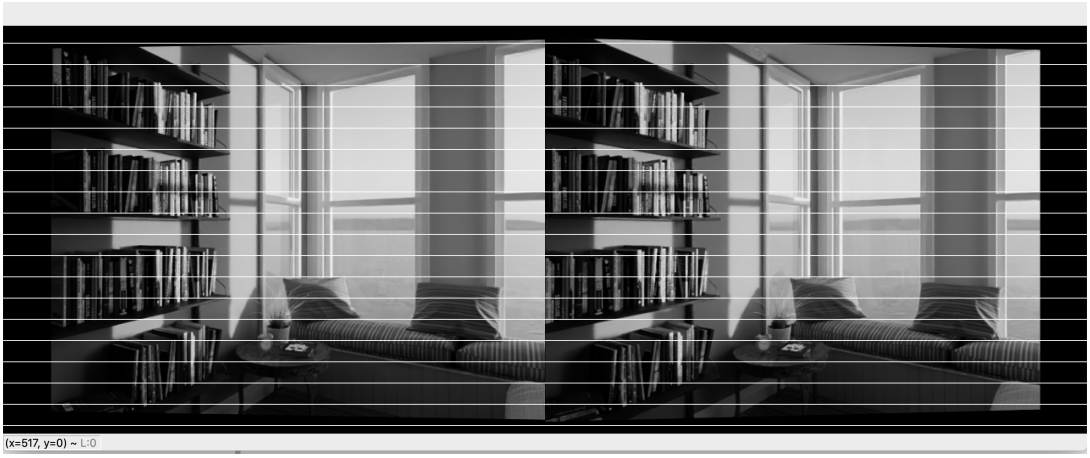


Figure 6: Rectified images

Once we have calculated the disparity for each point (top shelf, window corner, pillow corner etc.) we can use the formula for our depth $d = \frac{tf}{disparity}$ (remember that the focal length is in pixels and the baseline in meter, which yields meters for the depth unit) to calculate the image depth per point and the difference between the depth of two points. This yields a distance of **1.12 meters** between the points connected through the purple line (disparity of top shelf corner: 61, disparity of window corner: 38). A distance of **0.47 meters** between the points connected through the green line (disparity of book corner: 60, disparity of window corner: 48). And a distance of **0.43 meters** between the points connected through the red line (disparity of alarm clock: 50, disparity of pillow corner: 42).

4 Exercise 4 - Optional

In this optional exercise I created a disparity map, where I used number of disparities 128 and a block size of 15. One can see how the objects become darker, the further they are away from the camera.

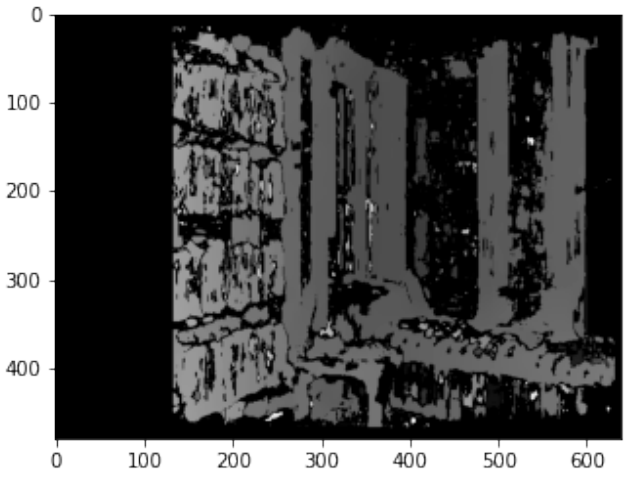


Figure 7: Disparity map

5 Appendix - Code

```
from matplotlib import pyplot as plt
import cv2 as cv
import numpy as np


###implement SIFT on image 1
img1 = cv.imread('Data/Frame1.tif')
gray1= cv.cvtColor(img1,cv.COLOR_BGR2GRAY)
sift1 = cv.SIFT_create()
kp1 = sift1.detect(gray1,None)
img1=cv.drawKeypoints(gray1,kp1,img1)
cv.imwrite('sift_keypoints_img1.jpg',img1)


###implement SIFT on image 2
img2 = cv.imread('Data/Frame2.tif')
gray2= cv.cvtColor(img2,cv.COLOR_BGR2GRAY)
sift2 = cv.SIFT_create()
kp2 = sift2.detect(gray2,None)
img2=cv.drawKeypoints(gray2,kp2,img2)
cv.imwrite('sift_keypoints_img2.jpg',img2)


###Draw matches of salient features
img1 = cv.imread('Data/Frame1.tif',cv.IMREAD_GRAYSCALE)
img2 = cv.imread('Data/Frame2.tif',cv.IMREAD_GRAYSCALE)
# Initiate SIFT detector
sift = cv.SIFT_create()
# find the keypoints and descriptors with SIFT
```

```
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
# BFMatcher with default params
bf = cv.BFMatcher()
matches = bf.knnMatch(des1,des2,k=2)
# Apply ratio test
good = []
for m,n in matches:
    if m.distance < 0.75*n.distance: #default is 0.75
        good.append([m])
#draw matches
img3 =
    cv.drawMatchesKnn(img1,kp1,img2,kp2,good,None,flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
plt.imshow(img3),plt.show()
cv.imwrite('two_compared.jpg',img3)


###Find and print fundamental matrix
img1 = cv.imread('Data/Frame1.tif',cv.IMREAD_GRAYSCALE)
img2 = cv.imread('Data/Frame2.tif',cv.IMREAD_GRAYSCALE)
# Initiate SIFT detector
sift = cv.SIFT_create()
# find the keypoints and descriptors with SIFT
kp1, des1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
# BFMatcher with default params
bf = cv.BFMatcher()
matches = bf.knnMatch(des1,des2,k=2)
# Apply ratio test
good = []
pts1 = []
pts2 = []
for m,n in matches:
    if m.distance < 0.75*n.distance: #default is 0.75
        good.append([m])
        #store salient features coordinates
        pts2.append(kp2[m.trainIdx].pt)
        pts1.append(kp1[m.queryIdx].pt)
pts1 = np.int32(pts1)
pts2 = np.int32(pts2)
F, mask = cv.findFundamentalMat(pts1,pts2,cv.FM_LMEDS)
print(F)
# We select only inlier points
pts1 = pts1[mask.ravel()==1]
pts2 = pts2[mask.ravel()==1]


### function to draw epipolar lines
def drawlines(img1,img2,lines,pts1,pts2):
    ''' img1 - image on which we draw the epilines for the points in img2
        lines - corresponding epilines '''
    r,c = img1.shape
    img1 = cv.cvtColor(img1,cv.COLOR_GRAY2BGR)
    img2 = cv.cvtColor(img2,cv.COLOR_GRAY2BGR)
    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] ])
        img1 = cv.line(img1, (x0,y0), (x1,y1), color,1)
        img1 = cv.circle(img1,tuple(pt1),5,color,-1)
        img2 = cv.circle(img2,tuple(pt2),5,color,-1)
    return img1,img2
```

```
# Find epilines corresponding to points in right image (second image) and drawing its lines on
    left image
lines1 = cv.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)
lines1 = lines1.reshape(-1,3)
img5,img6 = drawlines(img1,img2,lines1,pts1,pts2)
# Find epilines corresponding to points in left image (first image) and
# drawing its lines on right image
lines2 = cv.computeCorrespondEpilines(pts1.reshape(-1,1,2), 1,F)
lines2 = lines2.reshape(-1,3)
img3,img4 = drawlines(img2,img1,lines2,pts2,pts1)
plt.subplot(121),plt.imshow(img5)
plt.subplot(122),plt.imshow(img3)
plt.show()
cv.imwrite('epipolar_1.jpg',img5)
cv.imwrite('epipolar_2.jpg',img3)


### keep only epilines that fulfill the epipolar constraint with a threshold of below 0.1
temp_pts1 = pts1
temp_pts2 = pts2
good = []

for i in range(len(pts1)):
    if np.abs(np.transpose(np.append(pts1[i],1))@ F @ np.append(pts2[i],1)) < 0.1:
        good.append(i)

pts1 = pts1[good]
pts2 = pts2[good]


### Plot these new epilines
lines1 = cv.computeCorrespondEpilines(pts2.reshape(-1,1,2), 2,F)
lines1 = lines1.reshape(-1,3)
img5,img6 = drawlines(img1,img2,lines1,pts1,pts2)
# Find epilines corresponding to points in left image (first image) and
# drawing its lines on right image
lines2 = cv.computeCorrespondEpilines(pts1.reshape(-1,1,2), 1,F)
lines2 = lines2.reshape(-1,3)
img3,img4 = drawlines(img2,img1,lines2,pts2,pts1)
plt.subplot(121),plt.imshow(img5)
plt.subplot(122),plt.imshow(img3)
plt.show()
cv.imwrite('epipolar_1_new.jpg',img5)
cv.imwrite('epipolar_2_new.jpg',img3)


### Camera parameters
k_1 = np.array([[497.77777778, 0., 319.5 ],
    [ 0., 497.77777778, 239.5 ],
    [0., 0., 1. ]], dtype=np.float64)
k_2 = k_1.copy()
distortion = np.array([0.0, 0.0, 0.0, 0.0, 0.0], dtype=np.float64)
r_vec = np.array([[9.96194713e-01, -5.41550256e-08, 8.71558063e-02],
                  [-1.60560472e-08, 1.00000012e+00, -5.68126794e-08],
                  [-8.71558795e-02, 1.94016203e-08, 9.96194765e-01]],dtype=np.float64)
t_vec = np.array([-2.48746482e-01, 1.28814475e-08, 1.27242718e-02],dtype=np.float64)
im_size = gray1.shape


###start rectifying images / get Q matrix to grab f and t
rotation1, rotation2, pose1, pose2, Q, roi1, roi2 = \
       cv.stereoRectify(cameraMatrix1=k_1,
```

```
                        distCoeffs1=distortion,
                        cameraMatrix2=k_2,
                        distCoeffs2=distortion,
                        imageSize=(im_size[1], im_size[0]),
                        R=r_vec,
                        T=t_vec,
                        flags=cv.CALIB_ZERO_DISPARITY,
                        alpha = 1.0
                        )
f = Q[2,3]
t = 1/Q[3,2]*(-1)
f_t = f*t
print('f={}'.format(f))
print('t={}'.format(t))
print('Q={}'.format(Q))


### Rectify images and plot them side. by side
map1x, map1y = cv.initUndistortRectifyMap(k_1,distortion,rotation1, pose1, (im_size[1],
    im_size[0]), cv.CV_32FC1)
map2x, map2y = cv.initUndistortRectifyMap(k_2,distortion,rotation2, pose2, (im_size[1],
    im_size[0]), cv.CV_32FC1)
img_rect1 = cv.remap(gray1, map1x, map1y, cv.INTER_LINEAR)
img_rect2 = cv.remap(gray2, map2x, map2y, cv.INTER_LINEAR)

    # draw the images side by side
total_size = (max(img_rect1.shape[0], img_rect2.shape[0]), img_rect1.shape[1] +
    img_rect2.shape[1])
img = np.zeros(total_size, dtype=np.uint8)
img[:img_rect1.shape[0], :img_rect1.shape[1]] = img_rect1
img[:img_rect2.shape[0], img_rect1.shape[1]:] = img_rect2

# draw horizontal lines every 25 px accross the side by side image
for i in range(20, img.shape[0], 25):
    cv.line(img, (0, i), (img.shape[1], i), (255, 0, 0))

cv.imshow('imgRectified', img)
cv.waitKey()


print('Purple Difference')
disparity_1 = 61
disparity_2 = 38
depth_1 = f_t/disparity_1
depth_2 = f_t/disparity_2
difference = np.abs(depth_1 - depth_2)
print(difference)


print('Green Difference')
disparity_1 = 60
disparity_2 = 48
depth_1 = f_t/disparity_1
depth_2 = f_t/disparity_2
difference = np.abs(depth_1 - depth_2)
print(difference)


print('Red Difference')
disparity_1 = 50
disparity_2 = 42
depth_1 = f_t/disparity_1
depth_2 = f_t/disparity_2
```

```
difference = np.abs(depth_1 - depth_2)
print(difference)


###Get disparity matrix
map1x, map1y = cv.initUndistortRectifyMap(k_1,distortion,rotation1, pose1, (im_size[1],
    im_size[0]), cv.CV_32FC1)
map2x, map2y = cv.initUndistortRectifyMap(k_2,distortion,rotation2, pose2, (im_size[1],
    im_size[0]), cv.CV_32FC1)
img_rect1 = cv.remap(gray1, map1x, map1y, cv.INTER_LINEAR)
img_rect2 = cv.remap(gray2, map2x, map2y, cv.INTER_LINEAR)
stereo = cv.StereoBM_create(numDisparities=128, blockSize=15)

disparity = stereo.compute(img_rect1, img_rect2)
plt.imshow(disparity,'gray')
plt.savefig('disparity_map')
plt.show()
```