ECE 66100 Computer Vision **Name:** *Nikita Ravi (ravi30)*
Homework 9
11/27/2022

1 Theory

1.1 Projective Stereo Reconstruction

The goal of projective stereo reconstruction is to construct a 3D model of a scene from its two views. There are many steps involved in making this possible which will be explained below.

1.1.1 Image Rectification

Image rectification involves transforming a pair of stereo images (so a left image and right image of the same scene) such that the pixels corresponding to the world coordinate in the left image is mapped to the same row as the corresponding world coordinates in the right image.

The process of image rectification involves"

1) We first normalize the eight coordinates from both stereo images. This is because the 8-point algorithm of constructing the fundamental matrix requires a translation and scaling of each image so that its centroid is at the origin and the RMS distance of the points from the origin is equal to $\sqrt{2}$ [1]. This is accomplished by first

   I. Compute the mean of the x-coordinates and y-coordinates

   II. Calculate the distance of each x-coordinate and y-coordinate to its respective mean coordinate

   III. Compute the mean of the distance vector calculated in step (II), denoted $\bar{D}$

   IV. Let $c = \frac{\sqrt{2}}{\bar{D}}$

   V. Create the normalization matrix such that the RMS is $\sqrt{2}$ as shown below

$$T = \begin{bmatrix} c & 0 & -cx \\ 0 & c & -cy \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

   VI. The normalized pixel is therefore a matrix multiplication of the homogeneous coordinates and T

2) Calculate the initial estimate of the fundamental matrix F. This is accomplished by the following process

   I. Using the normalized coordinates of the left image $(\hat{x}, \hat{y})$ and the normalized coordinates of the right image $(\hat{x}', \hat{y}')$ to construct the 8×9 matrix A where each row is as shown below

$$A[\text{idx}] = \begin{bmatrix} \hat{x}\hat{x}' & \hat{y}\hat{x}' & \hat{x}' & \hat{x}\hat{y}' & \hat{y}'\hat{y} & \hat{y}' & \hat{x} & \hat{y} & 1 \end{bmatrix} \tag{2}$$

II. We then solve the equation $Af = 0$ where f is a one-dimensional vector containing all the eight unknowns of F. This is done by doing a singular value decomposition (SVD) of A to get the initial estimate of F

III. The fundamental matrix must be conditioned to enforce the requirement: $\text{rank}(F) = 2$ to construct a linear least squares solution. This is done by taking the SVD of F, and then setting the smallest eigenvalue of D to zero, we denote this as D'. Once we get D', we do a matrix multiplication of the u from the SVD of F, with D', and with vh from the SVD of F: $F_{\text{conditioned}} = u \times D' \times vh$

IV. Finally, we de-normalize the conditioned initial estimate of F by doing a matrix multiplication of the normalization matrix T of both stereo images on F. In other words: $T_R^T \times F \times T_L$

3) We then optimize the denormalized F by using a non-linear least-squares optimization method called Levenberg-Marquadt algorithm to minimize the error to ensure proper rectification of the images. The cost function is as follows

$$\text{cost} = \sum_i (\|x_i - \hat{x}_i\|^2 + \|x_i' - \hat{x}_i'\|^2) \tag{3}$$

4) Using the optimized fundamental matrix, we then compute the epipole lines by solving these two following equations

$$F\vec{e} = 0 \tag{4}$$

$$\vec{e'^T} F = 0 \tag{5}$$

5) Using the canonical camera configuration method, we compute the projection matrices P and P' as follows

$$P = [I|0] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{6}$$

$$P' = [[\vec{e'}]_x F|\vec{e'}] \text{where} \quad [\vec{e'}]_x = \begin{bmatrix} 0 & -e_z' & e_y' \\ e_z' & 0 & -e_x' \\ -e_y' & e_x' & 0 \end{bmatrix} \tag{7}$$

6) Finally, we need to determine a homography $H_{\{L,R\}}$ that would rectify the stereo images. First, we need to calculate H_R using the following method

I. Create a translation matrix to translate the image such that the center is at the origin

$$T = \begin{bmatrix} 1 & 0 & \frac{-w}{2} \\ 0 & 1 & \frac{-h}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{8}$$

II. Create the rotation matrix with an angle $\theta = -\arctan \frac{\vec{e_y'} - h/2}{\vec{e_x'} - w/2}$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9}$$

III. Create a matrix G that would send the epipoles to infinity with a scale factor $f = (e_x' - \frac{w}{2})\cos\theta - (e_y' - \frac{h}{2})\sin\theta$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{f} & 0 & 1 \end{bmatrix} \tag{10}$$

IV . Rectify the right image center by first calculating the homography $H_{R-\text{center}} = GRT$ and then multiplying it with the centroid of the image. The rectified center has coordinates $(\tilde{x}, \tilde{y})$

V. Create a second translation matrix that would move the rectified image center back to the original image center

$$T_2 = \begin{bmatrix} 1 & 0 & w/2 - \tilde{x} \\ 0 & 1 & h/2 - \tilde{y} \\ 0 & 0 & 1 \end{bmatrix} \tag{11}$$

VI. The final homography to rectify the right image is given by $H_R = T_2 GRT$

7) To calculate H_L we use the following method

I. Repeat steps I-VI from the method used to compute the rectification homography for the right image

II. H_L is obtained by the matrix that minimizes the least square distance between the transformed points [1]

$$\text{cost} = \sum_i (ax_i + by_i + c - x_i^2) \tag{12}$$

III. The constants from the minimized cost function will be used to construct the homography H_a

$$H_a = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{13}$$

IV. Using H_a we repeat steps V-VI from the method to compute H_R

8) Using H_L and H_R rectify the left and right stereo images

1.1.2 Interest Point Detection

To make a 3D reconstruction of the stereo images, we need many correspondences between the two rectified images. We get these correspondences by

1) Perform edge detection on the rectified images to obtain key points
2) The corresponding points exist on the same row of the rectified images
3) SSD or NCC is used to filter our correspondences

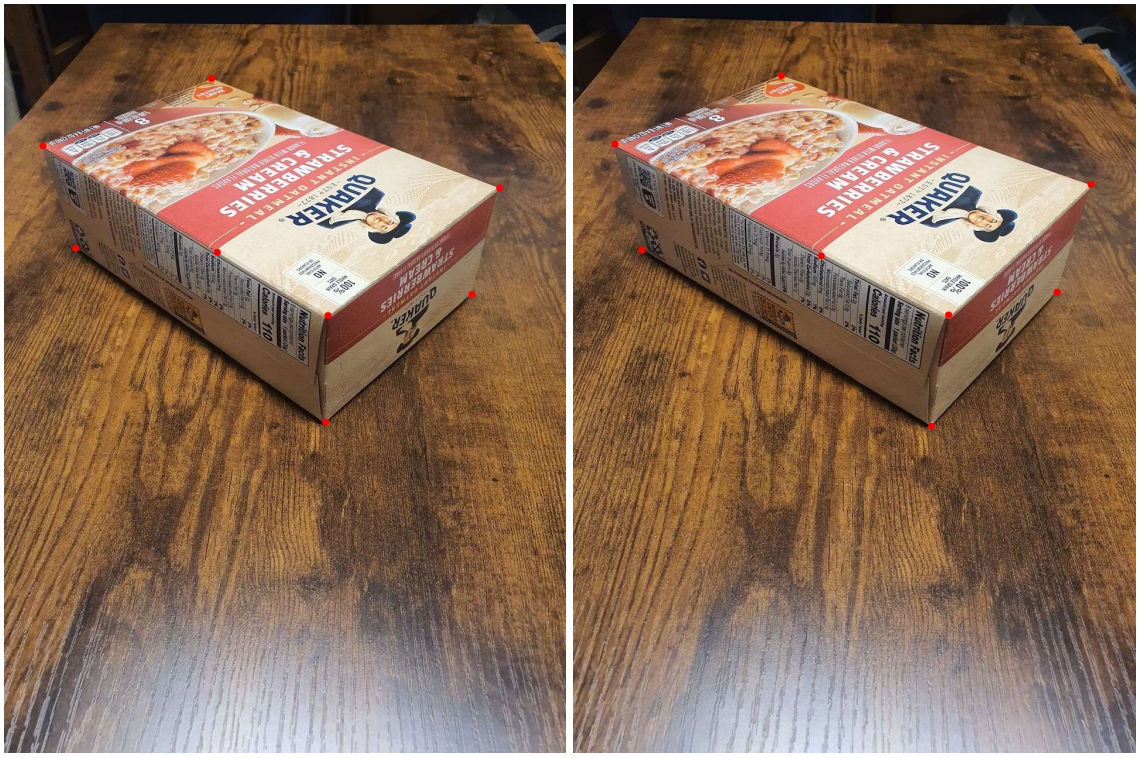
1.1.3 3D Projective Reconstruction

To compute the scene structure from a set of correspondences for a given camera pair (P_L, P_R) and stereo correspondences $\vec{X} = (x, y)$ and $\vec{X}' = (x', y')$ by constructing matrix A

$$A = \begin{bmatrix} x\vec{P_3}^T - \vec{P_1}^T \\ y\vec{P_3}^T - \vec{P_2}^T \\ x'\vec{P'_3}^T - \vec{P'_1}^T \\ y'\vec{P'_3}^T - \vec{P'_2}^T \end{bmatrix} \tag{14}$$

We then solve the equation $A\vec{X} = 0$ using singular value decomposition.

1.1.4 Inputs



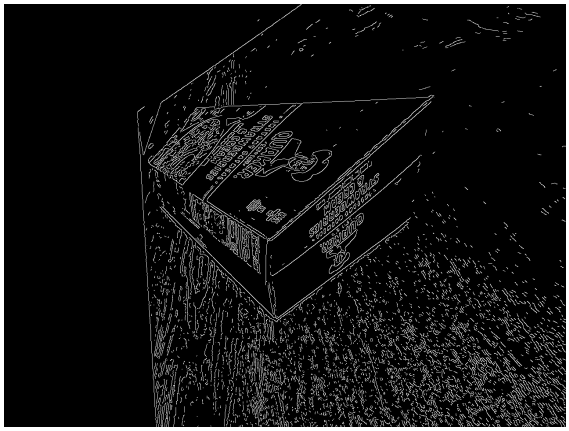
(a) Left Images (b) Right Images

Figure 1: Stereo Images

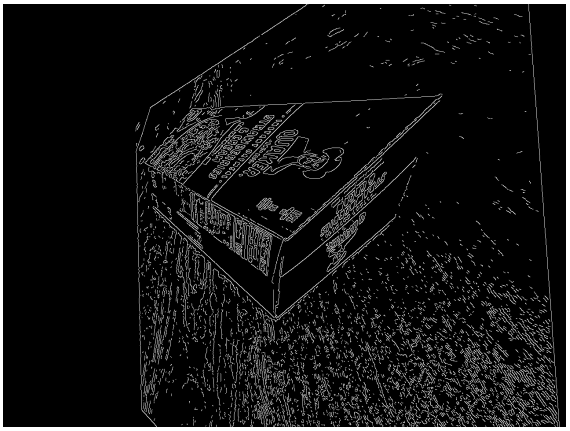
1.1.5 Outputs

(a) Rectified Left Image

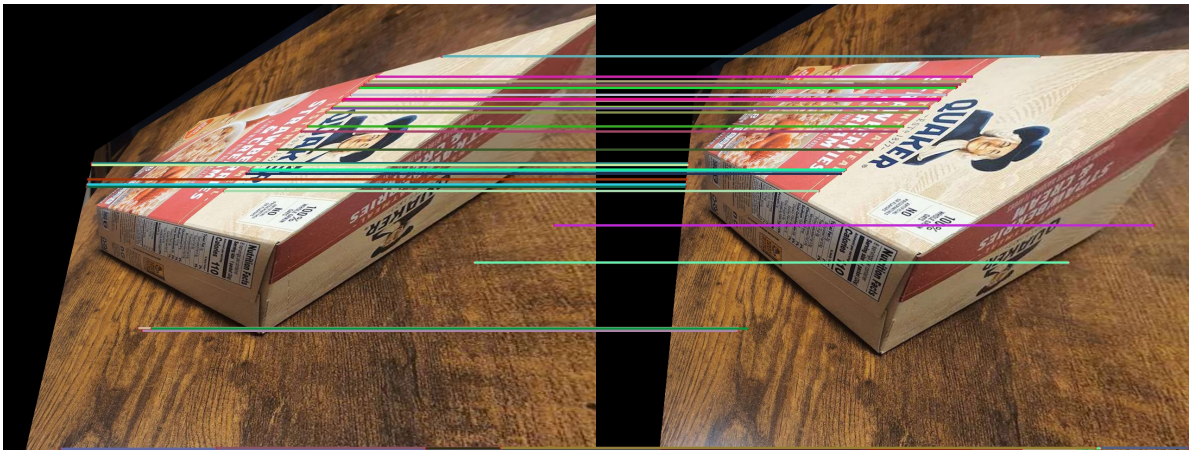
(b) Rectified Right Image



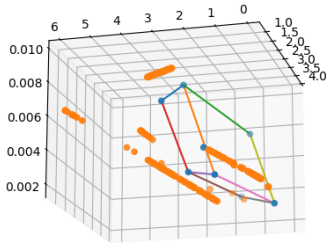
(c) Edges of the Rectified Left Image



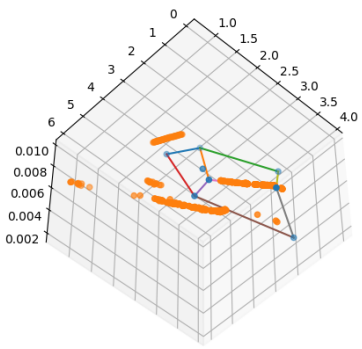
(d) Edges of the Rectified Right Image



(e) Correspondences



(f) 3D Reconstruction View 1



(g) 3D Reconstruction View 2

1.2 Loop and Zhang Algorithm

The Loop and Zhang algorithm decomposes the rectifying homographies:

$$H_L = H_{\text{sh}} H_{\text{sim}} H_{\text{p}} \tag{15}$$

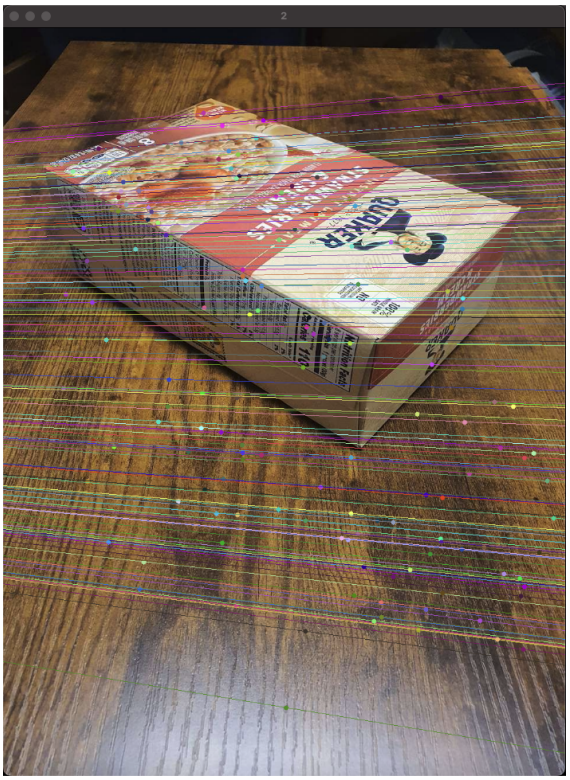
$$H_R = H'_{\text{sh}} H'_{\text{sim}} H'_{\text{p}} \tag{16}$$

where H_{p} and H'_{p} are homographies that will send the epipoles to infinity in their respective image planes. The direction in which the epipoles willl go to infinity is determined by the direction that causes the minimal amount of distortion. H_{sim} and H'_{sim} are similarity homographies which can rotate, translate, and uniformly scale an image. The purpose of the similarity homography is to rotate the epipoles onto the x-axis without causing any distortion. Finally, H_{sh} and H'_{sh} are shearing homographies. Since a non-linear distortion of the two images caused by projective homography cannot be undone by similarity transformations, the best we can do is reduce the distortion by introducing additional degrees of freedom into the overall rectification transformation which is where the shearing homographies come into the equation [2].

Despite the homographies being homogeneous, setting the last component of the matrices to one will prevent sending the origin in the image I to infinity. So typically, the corner of the sensor array is used as the image origin and the likelihood of the epipole coinciding with such an origin will be slim. The two cameras need to be angled sharply toward each other for this to happen [2].

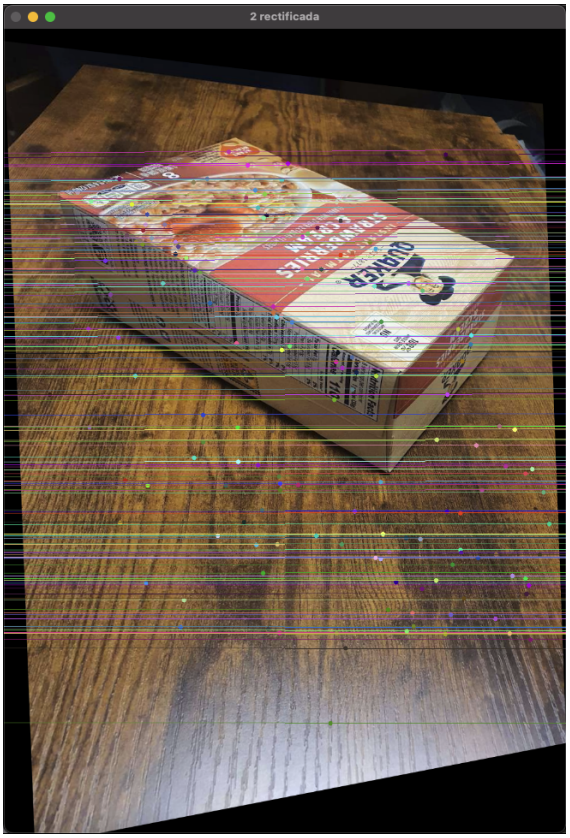
1.2.1 Outputs

(a) Left Image



(b) Right Image

(c) Rectified Left Image



(d) Rectified Right Image

Figure 3: Loop and Zhang Outputs

1.2.2 Observations

The rectified images obtained from the Loop and Zhang algorithm are different from the pipeline I developed in task 1. Just like the correspondence lines from task 1, the correspondence lines from the Loop and Zhang algorithms are completely horizontal suggesting that the images have been successfully rectified.

1.3 Dense Stereo Matching

Dense Stereo Matching is similar to the canny edge detector method to correspondences between stereo images. This method is better than the canny edge detector because it does not return a subset of correspondences but rather it can match all the pixels. The Dense Stereo Matching is conducted by doing a census transform which is done as follows

1) Calculate the d_{max} from the ground truth disparity map. The d_{max} obtained for this homework is 52.

2) For a pixel in the left image, save the intensity of the pixels around it in a $M \times M$ neighborhood.

3) Create a $M \times M$ window for the right image around the pixel with the same coordinates as the one in the left image and save the intensity of the pixels in this window for d_{max} times

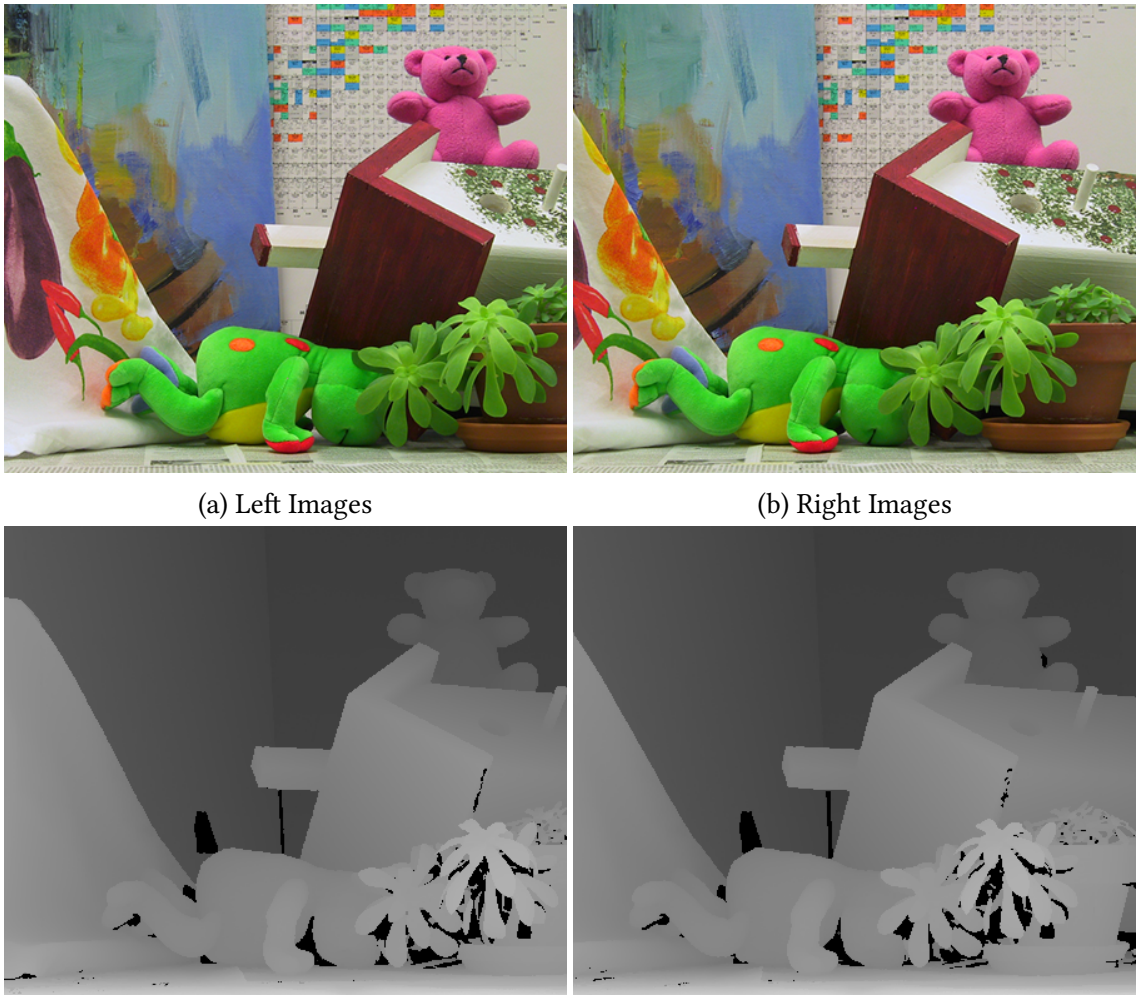
4) Pixels with intensity greater than the intensity of the center pixel are assigned 1, otherwise it is assigned a 0

5) Bitwise XOR the two binary neighborhoods obtained from step (3)

6) The cost associated to a pixel in the right image is the number of 1s obtained from step (4)

7) The pixel at a specific row and column in the left image will be assigned the lowest cost calculated in the previous step on the disparity map

1.3.1 Inputs

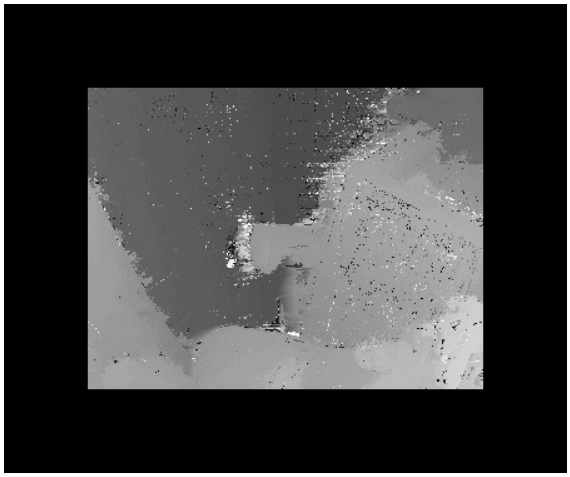


(a) Left Images (b) Right Images

(c) Ground Truth Disparity Map of Left Image (d) Ground Truth Disparity Map of Right Image

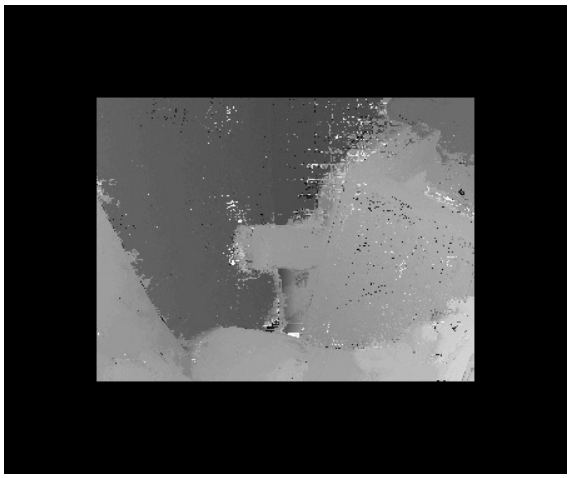
Figure 4: Dense Stereo Matching Inputs

1.3.2 Outputs



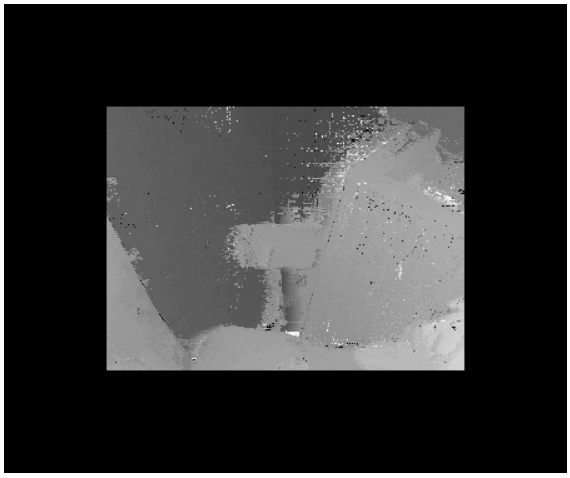
(a) Disparity Map $M = 30$

(b) Error Mask $M = 30$, Accuracy: 84.0%



(c) Disparity Map $M = 45$

(d) Error Mask $M = 45$, Accuracy: 82.5%



(e) Disparity Map $M = 60$

(f) Error Mask $M = 60$, Accuracy: 80.6%

Figure 5: Dense Stereo Matching Outputs

1.3.3 Observations

As the size of the window M increased, the disparity map became a lot more clearer but the quality of the error masks and accuracy kept decreasing.

2 Source Code

```
# Name: Nikita Ravi
# Class: ECE 66100
# Homework #9
# Deadline: 11/27/2022

# Import Modules
import cv2
import numpy as np
import math
import random
import matplotlib.pyplot as plt
from scipy.optimize import leastsquares

def displayimage(image, points=True):
  def clickevent(event, x, y, flags, params):
    # This function was inspired by https://www.geeksforgeeks.org/displaying-the-coordinates-of-the-points-clicked-on-the-image-using-python-opencv/
    if(event == cv2.EVENTLBUTTONDOWN):
      print(x, y)
      font = cv2.FONTHERSHEYSIMPLEX
      cv2.circle(image, (x, y), 1, (0,0,255), thickness=-1)
      cv2.putText(image, str(x) + ',' +
            str(y), (x,y), font,
            1, (0, 0, 255), 2)
      cv2.imshow('window', image)


  cv2.imshow("window", image)
  if(points):
    cv2.setMouseCallback('window', clickevent)
  cv2.waitKey(0)
  cv2.destroyAllWindows()
  quit()

def drawcircles(path, image, pts):
  for idx, pt in enumerate(pts):
    x, y = int(pt[0]), int(pt[1])
    cv2.circle(image, (x, y), radius=6, color=(0,0,255), thickness=-1)

  cv2.imwrite(path, image)

def normalizepoints(correspondences):
  """
  Normalization is necessary for the 8-point algorithm of constructing the
   fundamental matrix to be successful. The suggested normalization
```

```
  is a translation and scaling of each image so that the centroid of the
   reference points is at the origin of the coordinates and the RMS
  distance of the points from the origin is equal to sqrt(2) (Hartley and
   Zisserman, 2003).
  """
  xcoord = correspondences[:, 0]
  ycoord = correspondences[:, 1]

  # Compute the mean distance
  xmean, ymean = np.mean(xcoord), np.mean(ycoord)
  distances = np.sqrt((xcoord - xmean) ** 2 + (ycoord - ymean) ** 2)
  meandist = np.mean(distances)

  # Create normalizing transformation matrix consisting of translation and
   scaling
  c = np.sqrt(2) / meandist
  tMat = np.array([[c, 0, -c*xmean],
          [0, c, -c*ymean],
          [0, 0, 1]])

  # Apply the translation matrices
  normalizedpts = np.hstack((correspondences, np.ones((len(correspondences),
   1)))) # Make it homogeneous
  normalizedpts = np.dot(tMat, normalizedpts.T).T

  return normalizedpts, tMat

def conditionF(F):
  # Fundamental matrix F must be conditioned to enforce the requirement rank(F
   ) = 2 to construct a linear least squares solution
  u, d, vh = np.linalg.svd(F)
  dprime = np.array([[d[0], 0, 0],
            [0, d[1], 0],
            [0, 0, 0]])
  conditionedF = u @ dprime @ vh
  return conditionedF

def calculatefundamentalmatrix(normalizedptsL, normalizedptsR):
  A = np.zeros((8, 9))
  for corr in range(len(normalizedptsL)):
    x1, y1 = normalizedptsL[corr, 0], normalizedptsL[corr, 1]
    x2, y2 = normalizedptsR[corr, 0], normalizedptsR[corr, 1]

    A[corr] = np.array([x1 * x2, x2 * y1, x2, y2 * x1, y2 * y1, y2, x1, y1,
   1])

  u, s, vh = np.linalg.svd(A)
  Fflatten = vh[-1]
  F = np.reshape(Fflatten, (3, 3))
  return conditionF(F)

def computeEx(e):
  Ex = np.array([[0, -e[2], e[1]],
          [e[2], 0, -e[0]],
```

```
          [-e[1], e[0], 0]])

  return E x

def findepipoles(F):
  u,s,vh = np.linalg.svd(F)
  eL = vh[-1, :].T
  eL = eL / eL[2] # Make it homogeneous

  eR = u[:, -1]
  eR = eR / eR[2] # Make it homogeneous

  E x = computeEx(eR)
  return eL, eR, E x

def computeprojectionmatrices(F, eR, s):
  P1 = np.hstack((np.eye(3), np.zeros((3,1))))
  P2 = np.hstack((np.dot(s, F), np.transpose([eR])))

  return P1, P2

def costfunction(f, ptsL, ptsR):
  F = np.reshape(f, (3, 3))
  eL, eR, E x = findepipoles(F)
  P1, P2 = computeprojectionmatrices(F, eR, E x)

  ptsL = np.hstack((ptsL, np.ones((len(ptsL), 1)))) # Make it homogeneous
  ptsR = np.hstack((ptsR, np.ones((len(ptsR), 1)))) # Make it homogeneous

  dgeom = []
  for corr in range(len(ptsL)):
    A = np.zeros((4, 4)) # AX = 0
    A[0] = ptsL[corr][0] * P1[2, :] - P1[0, :]
    A[1] = ptsL[corr][1] * P1[2, :] - P1[1, :]
    A[2] = ptsR[corr][0] * P2[2, :] - P2[0, :]
    A[3] = ptsR[corr][1] * P2[2, :] - P2[1, :]

    # Compute X
    u, s, vh = np.linalg.svd(A)
    X = vh[-1, :].T
    Xhat = X / X[3] # Unit Vector of X
    xL = np.dot(P1, Xhat)
    xL = xL / xL[2] # Make it Homogeneous

    xR = np.dot(P2, Xhat)
    xR = xR / xR[2] # Make it Homogeneous

    # Append the re-projection errors
    dgeom.append(np.linalg.norm(xL - ptsL[corr]) ** 2)
    dgeom.append(np.linalg.norm(xR - ptsR[corr]) ** 2)

  return np.ravel(dgeom)

def computehomographyrectification(image, corrsL, corrsR, eL, eR):
```

```
height, width = image.shape[:2]

# Translation Matrix to translate the image such that the center is at the origin
T = np.array([[1, 0, -width / 2],
      [0, 1, -height / 2],
      [0, 0, 1]]) # Obtained Translation Matrix from https://web.stanford.edu/class/cs231a/coursenotes/03-epipolar-geometry.pdf

##### Calculate HR or H' #####
# Rotation Matrix to move the epipoles towards the x-axis
theta = -np.arctan((eR[1] - height/2)/(eR[0] - width/2))
R = np.array([[math.cos(theta), -math.sin(theta), 0],
      [math.sin(theta), math.cos(theta), 0],
      [0, 0, 1]]) # Inspired by 2020 hw#1 solution

# Create matrix that would send the epipoles to infinity
f = (eR[0] - width / 2) * math.cos(theta) - (eR[1] - height / 2) * math.sin(theta) # Inspired by 2020 hw#1 solution
G = np.array([[1, 0, 0],
      [0, 1, 0],
      [-1 / f, 0, 1]])

# Rectify the right image center
HRcenter = np.dot(G, np.dot(R, T))
Xprime = np.dot(HRcenter, np.array([width / 2, height / 2, 1])) # Apply HRcenter to rectify the center of the right image (x', y')
T2 = np.array([[1, 0, width / 2 - Xprime[0]],
        [0, 1, height / 2 - Xprime[1]],
        [0, 0, 1]]) # Move the rectified center back to the true image center

# Rectify the entire right image
HR = np.dot(np.dot(T2, G), np.dot(R, T))
HR = HR / HR[2, 2] # Make it Homogeneous

##### Calculate HL or H #####
# Create a rotation matrix to move the epipole to the x-axis
theta = math.atan2(eL[1] - height / 2, (eL[0] - width / 2))
R = np.array([[math.cos(theta), -math.sin(theta), 0],
      [math.sin(theta), math.cos(theta), 0],
      [0, 0, 1]]) # Inspired by 2020 hw#1 solution

# Create matrix that would send the epipoles to infinity
f = math.cos(theta) * (eL[0] - width / 2) - math.sin(theta) * (eL[1] - height / 2) # Inspired by 2020 hw#1 solution
G = np.array([[1, 0, 0],
      [0, 1, 0],
      [-1 / f, 0, 1]])

# Rectify the left image center
H0 = G @ R @ T

# HL is obtained by the matrix that minimizes the least squares fistance
```

```
  # pg 307 (Hartley and Zisserman, 2003)
  ptsL = np.hstack((corrsL, np.ones((len(corrsL), 1)))) # Make it Homogeneous
  ptsR = np.hstack((corrsR, np.ones((len(corrsR), 1)))) # Make it Homogeneous

  # Transform the points by their respective homographies
  xL = np.dot(H0, ptsL.T).T
  xR = np.dot(HR, ptsR.T).T

  # Make them Homogeneous
  # xL = xL / xL[:, 2]
  xL[:, 0] = xL[:, 0] / xL[:, 2]
  xL[:, 1] = xL[:, 1] / xL[:, 2]
  xL[:, 2] = xL[:, 2] / xL[:, 2]

  xR[:, 0] = xR[:, 0] / xR[:, 2] # Only need the x-coordinate as shown on pg. 307 to minimize the dist

  # The least squares solution to an mxn system of equations Ax=b of rank n is given by x = (A+)b which minimizes ——x——. (pg. 590)
  a,b,c = np.dot(np.linalg.pinv(xL), xR[:, 0])
  Ha = np.array([[a, b, c], [0, 1, 0], [0, 0, 1]]) # Eq. 11.20 pg 306
  HLcenter = np.dot(Ha, H0)
  Xprime = np.dot(HLcenter, np.array([width / 2, height / 2, 1])) # Apply HRcenter to rectify the center of the right image (x', y')
  Xprime = Xprime / Xprime[2] # Make it Homogeneous

  T2 = np.array([[1, 0, width / 2 - Xprime[0]],
          [0, 1, height / 2 - Xprime[1]],
          [0, 0, 1]]) # Move the rectified center back to the true image center

  # Rectify the entire image
  HL = np.dot(T2, HLcenter)
  HL = HL / HL[2, 2]

  return HL, HR

def transform(domainimage, H):
  height, width = domainimage.shape[:2]
  return cv2.warpPerspective(domainimage, H, dsize=(height, width))

def detectedges(rectifiedimg, lowthreshold=60, highthreshold=70, edges=5):
  gray = cv2.cvtColor(rectifiedimg, cv2.COLORBGR2GRAY) if len(rectifiedimg.shape) == 3 else rectifiedimg
  gray = cv2.GaussianBlur(gray, ksize=(7, 7), sigmaX=1.3)

  return cv2.Canny(gray, threshold1=highthreshold, threshold2=lowthreshold, edges=edges)

def filtercorrespondences(imgL, imgR, corrs, kernelsize, maximumcorrs, mode="ncc"):
  halfkernelsize = kernelsize // 2
  if(maximumcorrs ¿ len(corrs)):
    print("Maximum correspondences exceeded the number of correspondences")
```

```
    maximumcorrs = len(corrs)

  dist = []
  for corr in corrs:
    corrL, corrR = corr
    f1 = imgL[corrL[1] - halfkernelsize:corrL[1] + halfkernelsize + 1,
          corrL[0] - halfkernelsize:corrL[0] + halfkernelsize + 1]

    f2 = imgR[corrR[1] - halfkernelsize:corrR[1] + halfkernelsize + 1,
          corrR[0] - halfkernelsize:corrR[0] + halfkernelsize + 1]

    if(f1.shape != f2.shape):
      continue

    if(mode == "ssd"):
      dist.append(np.sum((f1 - f2) ** 2))
    elif(mode == "ncc"):
      m1, m2 = np.mean(f1), np.mean(f2)
      numerator = np.sum((f1-m1)*(f2-m2))
      denominator = np.sqrt(np.sum((f1-m1)**2) * np.sum((f2-m2)**2))
      distance = numerator / denominator
      dist.append(1 - distance)

  # Sort the correspondences and choose the best maxcorrespondence
  sortedcorrespondences = [pt for d, pt in sorted(zip(dist, corrs), key=
    lambda x: x[0])]
  return sortedcorrespondences[0:maximumcorrs]

def getcolor():
  r = random.randint(0,255)
  g = random.randint(0,255)
  b = random.randint(0,255)
  return (r,g,b)

def drawcorrespondences(imgL, imgR, width1, correspondences, path):
  # From homework 4
  combined = np.hstack((imgL, imgR))
  for idx, corr in enumerate(correspondences):
    if(not idx % 10):
      pt1, pt2 = corr
      cv2.circle(combined, (pt1[0], pt1[1]), radius=3, color=(0,0,255))
      cv2.circle(combined, (pt2[0] + width1, pt2[1]), radius=3, color
    =(0,0,255))
      cv2.line(combined, pt1, (pt2[0] + width1, pt2[1]), color=getcolor(),
    thickness=3)

  cv2.imwrite(path, combined)

def detectcorrespondences(edgesL, edgesR, maxsearcharea):
  grayEdgesL = cv2.cvtColor(edgesL, cv2.COLORBGR2GRAY) if len(edgesL.shape)
    == 3 else edgesL
  grayEdgesR = cv2.cvtColor(edgesR, cv2.COLORBGR2GRAY) if len(edgesR.shape)
    == 3 else edgesR

```

```
  listofcorrespondences = []
  for row in range(grayEdgesL.shape[0]):
    nonzeroidxL = np.where(grayEdgesL[row] ¿ 0)[0] # Get indices of all the
    edges that are not equal to 0
    if(not np.size(nonzeroidxL)):
      continue

    # For each pixel in the left image, find the left-most pixel in the right
    image
    for colL in nonzeroidxL:
      searchkernel = edgesR[row, colL:colL + maxsearcharea + 1]
      potentialnonzeroidxR = np.where(searchkernel ¿ 0)[0]

      if(not np.size(potentialnonzeroidxR)):
        continue

      colR = potentialnonzeroidxR[0] + colL
      edgesR[row, colR] = 0 # Don't count this pixel anymore
      listofcorrespondences.append([[colL, row], [colR, row]])

  return listofcorrespondences

def projectivereconstruction(corrs, PL, PR):
  worldcoords = []

  for idx, corr in enumerate(corrs):
    A = np.zeros((4, 4))
    corrL, corrR = corr

    A[0] = corrL[0] * PL[2, :] - PL[0, :]
    A[1] = corrL[1] * PL[2, :] - PL[1, :]
    A[2] = corrR[0] * PR[2, :] - PR[0, :]
    A[3] = corrR[1] * PR[2, :] - PR[1, :]

    u, s, vh = np.linalg.svd(A.T @ A)
    vh = vh[-1, :].T
    worldcoords.append(vh / vh[3])

  worldcoords = np.reshape(worldcoords, (len(corrs), 4))
  return worldcoords

def transformpts(pts, H):
  transformedpts = []
  for pt in pts:
    X = np.array([pt[0], pt[1], 1.])
    Xprime = np.dot(H, X)
    Xprime = Xprime / Xprime[-1]
    transformedpts.append((Xprime[0], Xprime[1]))
  return transformedpts

def projectivestereoreconstruction():
  imgL = cv2.imread(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/
    ECE 66100/hw09/Task3Images/imgL.JPG")
```

```
imgR = cv2.imread(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE 66100/hw09/Task3Images/imgR.JPG")

# Correspondence Points for the Left Image and Right Image
corrsL = np.int32(np.array([[66, 242],
          [122, 417],
          [549, 714],
          [553, 532],
          [846, 314],
          [799, 496],
          [354, 127],
          [364, 424]]))

corrsR = np.int32(np.array([[71, 239],
          [117, 420],
          [611, 721],
          [640, 531],
          [883, 308],
          [825, 492],
          [355, 122],
          [423, 430]]))

# Save the inputs with correspondence points
drawcircles(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE 66100/hw09/stereoInputs/imgL.jpg", imgL.copy(), corrsL)
drawcircles(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE 66100/hw09/stereoInputs/imgR.jpg", imgR.copy(), corrsR)

# Normalize the correspondences
normalizedptsL, tMatL = normalizepoints(corrsL)
normalizedptsR, tMatR = normalizepoints(corrsR)

# calculate the fundamental matrix
F = calculatefundamentalmatrix(normalizedptsL, normalizedptsR)
denormalizedF = tMatR.T @ F @ tMatL
denormalizedF = denormalizedF / denormalizedF[2, 2] # Make it homogeneous

# Find the epipoles and the camera projection matrices
eLnorm, eRnorm, Exnorm = findepipoles(denormalizedF)
PLnorm, PRnorm = computeprojectionmatrices(denormalizedF, eRnorm, Exnorm)

# Using nonlinear least squares minimization find optimum F
f = np.ravel(denormalizedF)
Foptimized = leastsquares(costfunction, f, args=[corrsL, corrsR], method="lm").x
Foptimized = conditionF(np.reshape(Foptimized, (3, 3))) # Make the fundamental matrix have a rank of 2
Foptimized = Foptimized / Foptimized[2, 2] # Make it homogeneous

# Get the optimized epipolar lines and camera projection matrices
eL, eR, Ex = findepipoles(Foptimized)
PL, PR = computeprojectionmatrices(Foptimized, eR, Ex)
```

```
388 # ================================IMAGE RECTIFICATION
     ======================================================
389 HL, HR = computehomographyrectification(imgL, corrsL, corrsR, eL, eR)
390
391 rectifiedleftimage = transform(imgL, HL)
392 cv2.imwrite(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE
     66100/hw09/rectified/imgL.jpg", rectifiedleftimage)
393
394 rectifiedrightimage = transform(imgR, HR)
395 cv2.imwrite(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE
     66100/hw09/rectified/imgR.jpg", rectifiedrightimage)
396
397 # ================================INTEREST POINT DETECTION
     ======================================================
398 # Detect Edges
399 edgesL = detectedges(rectifiedleftimage)
400 cv2.imwrite(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE
     66100/hw09/edges/imgL.jpg", edgesL)
401
402 edgesR = detectedges(rectifiedrightimage)
403 cv2.imwrite(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE
     66100/hw09/edges/imgR.jpg", edgesR)
404
405 # Detect Correspondences
406 corrsDetected = detectcorrespondences(edgesL, edgesR, maxsearcharea=15)
407 corrsFiltered = filtercorrespondences(rectifiedleftimage,
     rectifiedrightimage, corrsDetected, kernelsize=15, maximumcorrs=500,
     mode="ssd")
408 drawcorrespondences(rectifiedleftimage, rectifiedrightimage,
     rectifiedleftimage.shape[1], corrsFiltered, path=r"/Users/nikitaravi/
     Documents/Academics/Year 4/Semester 2/ECE 66100/hw09/correspondences/corr.
     jpg")
409
410 # ================================PROJECTIVE RECONSTRUCTION
     ======================================================
411
412 rectifiedCorrsL = transformpts(corrsL, HL)
413 rectifiedCorrsR = transformpts(corrsR, HR)
414 rectifiedcoordinates = [[[xL, yL], [xR, yR]] for (xL, yL), (xR, yR) in zip(
     rectifiedCorrsL, rectifiedCorrsR)]
415
416 drawcircles(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE
     66100/hw09/rectified/imgLcircles.jpg", rectifiedleftimage.copy(),
     rectifiedCorrsL)
417 drawcircles(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE
     66100/hw09/rectified/imgRcircles.jpg", rectifiedrightimage.copy(),
     rectifiedCorrsR)
418
419 unfilteredworldcoords = projectivereconstruction(rectifiedcoordinates,
     PL, PR)
420 filteredworldcoords = projectivereconstruction(corrsFiltered, PL, PR)
421
422 # ================================3D RECONSTRUCTION
     ======================================================
```

```
  fig = plt.figure()
  ax = fig.addsubplot(111, projection="3d")
  ax.scatter(unfilteredworldcoords[:, 0], unfilteredworldcoords[:, 1],
   unfilteredworldcoords[:, 2])
  ax.scatter(filteredworldcoords[:, 0], filteredworldcoords[:, 1],
   filteredworldcoords[:, 2])

  pairoflines = [[0, 1],
      [0, 3],
      [0, 6],
      [1, 2],
      [2, 3],
      [2, 5],
      [3, 4],
      [4, 5],
      [4, 6]] # Points that should form a line

  for pair in pairoflines:
    ax.plot([unfilteredworldcoords[pair[0]][0], unfilteredworldcoords[pair
   [1]][0]], "
            [unfilteredworldcoords[pair[0]][1], unfilteredworldcoords[pair
   [1]][1]], "
            [unfilteredworldcoords[pair[0]][2], unfilteredworldcoords[pair
   [1]][2]])

  plt.show()

def getdmax(gtDisp, mul):
  gtDisp = cv2.cvtColor(gtDisp, cv2.COLORBGR2GRAY)
  gtDisp = gtDisp.astype(np.float32) / (16 * mul) # divide by mul because
   disparity scale is 4
  gtDisp = gtDisp.astype(np.uint8)

  dmax = np.max(gtDisp)
  return gtDisp, dmax

def censustransform(imgL, imgR, M, dmax):
  height, width = imgL.shape[:2]

  grayL = cv2.cvtColor(imgL, cv2.COLORBGR2GRAY) if len(imgL.shape) == 3 else
   imgL
  grayR = cv2.cvtColor(imgR, cv2.COLORBGR2GRAY) if len(imgR.shape) == 3 else
   imgR

  halfM = M // 2
  bordersize = dmax + halfM
  disparitymap = np.zeros((height, width), dtype=np.uint8)

  for rowL in range(bordersize, height - bordersize):
    print(f"Row: -rowL + 1  / -height - bordersize ")
    for colL in range(width - bordersize - 1, bordersize - 1, - 1):
      cost = []
      leftWindow = grayL[rowL - halfM: rowL + halfM + 1,
                colL - halfM: colL + halfM + 1]
```

```
469
470         binaryleftwindow = np.ravel(np.where(leftWindow ¿ grayL[rowL, colL],
        1, 0))
471         for d in range(dmax + 1):
472           rowR, colR = rowL, colL - d
473           rightWindow = grayR[rowR - halfM: rowR + halfM + 1,
474                   colR - halfM: colR + halfM + 1]
475
476           binaryrightwindow = np.ravel(np.where(rightWindow ¿ grayR[rowL, colR
        ], 1, 0))
477           cost.append(np.sum(np.bitwisexor(binaryleftwindow,
        binaryrightwindow)))
478
479         disparitymap[rowL, colL] = np.argmin(cost)
480
481     return disparitymap.astype(np.uint8)
482
483 def computedisparityaccuracy(gtDisp, dispMap, dmax, M, delta=2):
484     halfM = M // 2
485     bordersize = halfM + dmax
486     dispMap = dispMap[bordersize:dispMap.shape[0]-bordersize, bordersize:
        dispMap.shape[1]-bordersize]
487     gtDisp = gtDisp[bordersize:gtDisp.shape[0]-bordersize, bordersize:gtDisp.
        shape[1]-bordersize]
488
489     error = np.abs(dispMap.astype(np.uint16) - gtDisp.astype(np.uint16)).astype(
        np.uint8)
490     validpixels = cv2.countNonZero(gtDisp) # Non-black pixels are valid pixels
        # 74952
491     print("The number of valid pixels: ", validpixels)
492
493     accuracy = np.sum(error ¡= delta)
494     percentageofaccuracy = accuracy / validpixels
495
496     errormask = np.where(error ¡= delta, 255, 0)
497     return percentageofaccuracy, errormask
498
499 def densestereomatching(M=10):
500     imgL = cv2.imread(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/
        ECE 66100/hw09/Task3Images/im2.ppm")
501     dispL = cv2.imread(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester
        2/ECE 66100/hw09/Task3Images/disp2.pgm")
502
503     imgR = cv2.imread(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/
        ECE 66100/hw09/Task3Images/im6.ppm")
504     dispR = cv2.imread(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester
        2/ECE 66100/hw09/Task3Images/disp6.pgm")
505
506     gtDisp, dmax = getdmax(dispL, mul=0.25) # dmax = 13 * mul
507     print("dmax: ", dmax)
508
509     disparitymap = censustransform(imgL, imgR, M, dmax)
510
```

```
511   savedisparitymap = cv2.normalize(disparitymap, dst=None, alpha=0, beta
        =255, normtype = cv2.NORMMINMAX).astype(np.uint8)
512   cv2.imwrite(r"/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE
        66100/hw09/disparityMap/" + str(M) + "size.jpg", savedisparitymap)
513
514   percentageofaccuracy, errormask = computedisparityaccuracy(gtDisp,
        disparitymap, dmax, M)
515   print("Percentage of Accuracy: ", percentageofaccuracy)
516   cv2.imwrite("/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE
        66100/hw09/errormask/errormask" + str(M) + ".jpg", errormask)
517
518
519 if __name__ == "__main__":
520   # TASK 1
521   projectivestereoreconstruction()
522
523   # TASK 2
524   densestereomatching(M=30)
525   densestereomatching(M=45)
526   densestereomatching(M=60)
```

Listing 1: The Source Code

References

[1] R. Hartley, A. Zisserman, *Multiple View Geometry in Computer Vision*, Cambridge, 2003.

[2] A. Kak, "A Loop and Zhang Reader for Stereo Rectification", `https://engineering.purdue.edu/kak/Tutorials/StereoRectification.pdf` 2022.