ECE 661 Fall 2020 - Homework 10

Brian Helfrecht

bhelfre@purdue.edu

1 Theory

In the present day, cameras are a powerful tool. Not only can they produce 2-D images of a scene, but given a set of images (as few as two!), a 3-D reconstruction of objects in that scene can be performed. Methods for performing this reconstruction were the focus of this assignment. First, projective stereo reconstruction of an object imaged from two slightly different views (stereo) was implemented with image rectification and interest point detection. Then, a second technique for finding interest points in a pair of rectified stereo images, called dense stereo matching, was performed. This method also enables the visualization of differences in the two images, especially occluded areas.

1.1 Projective stereo reconstruction

Projective stereo reconstruction describes the methods used to create a 3-D reconstruction of a scene from a pair of 2-D stereo images. The major steps in the process are 1) image rectification to align physical world points to corresponding image rows, 2) interest point detection to match interest points in each image row, and 3) 3-D projective reconstruction to obtain the 3-D world coordinates from the 2-D rectified correspondences. In this assignment, the 3-D reconstruction could only be accurate up to containing projective, affine, and similarity distortions because an uncalibrated camera was used to capture the stereo image pair.

1.1.1 Image rectification

Image rectification is used to transform a pair of stereo images such that the physical world points (or pixels corresponding to physical world points) in one image are mapped to the same row as the corresponding world points/pixels in a second image.

The rectification process begins with the manual selection of several corresponding interest points between the two stereo images. Although these points can be selected automatically, it was not required for this assignment. The goal is to use these correspondences to estimate the 3×3 fundamental matrix F, which mathematically describes the geometry of the scene, known as the *epipolar geometry*. Since F is a homogeneous matrix, we require at least 8 correspondences to calculate it. Before calculation, however, the pixel coordinates of the correspondences must be normalized per-image. We can do this as follows:

1. Compute the mean x-coordinate of the correspondences as $\overline{x}$
2. Compute the mean y-coordinate of the correspondences as $\overline{y}$
3. Compute the distance from each (x, y) coordinate to the mean coordinate $(\overline{x}, \overline{y})$ and store each distance value in a list D:
$$D_i = \sqrt{(x_i - \overline{x})^2 + (y_i - \overline{y})^2}$$
4. Compute the mean of the distances computed in the previous step as $\overline{D}$.
5. Set up the following normalization matrix where $c = \frac{\sqrt{2}}{\overline{D}}$:
$$T = \begin{pmatrix} c & 0 & -c\overline{x} \\ 0 & c & -c\overline{y} \\ 0 & 0 & 1 \end{pmatrix}$$

6. Compute the normalized pixel coordinates with $\hat{x} = Tx$, where x and $\hat{x}$ are homogeneous 3-vectors. Remember to convert back to normalized physical coordinates after the transformation.

With the normalized points, we can now compute an initial estimate for the 3×3 fundamental matrix (also called the *essential matrix* when normalized coordinates are used) describing the epipolar geometry of the scene. To do this, we use each normalized correspondence pair $(\hat{x}, \hat{y})$ in image 1 and $(\hat{x}', \hat{y}')$ in image 2 to construct the matrix A where each row i of A is:

$$A_i = \begin{pmatrix} \hat{x}\hat{x}' & \hat{x}'\hat{y} & \hat{x}' & \hat{y}'\hat{x} & \hat{y}'\hat{y} & \hat{y}' & \hat{x} & \hat{y} & 1 \end{pmatrix}$$

We then solve the linear least-squares problem $Af = 0$, where f is the eight unknowns in F written in vector form. The problem can be solved by taking the eigenvector corresponding to the smallest eigenvalue of the singular value decomposition (SVD) of A. This will give us an initial estimate of F when f is reshaped back into matrix form. However, we also require that F be of rank 2 to ensure all epipolar lines correspond exactly to each other between the two images. To enforce this rank constraint, we perform the following:

1. Take the SVD of F as $svd(F) = UDV^T$.
2. Set the smallest eigenvalue of D to zero. Let this new D be D'.
3. Recompute $F = UD'V^T$.

The last step in computing F is to "de-normalize" it to put it back into the image coordinate system by applying the normalization matrices T_1 and T_2 used to normalize the initial correspondences in images 1 and 2, respectively. That is,

$$F_{final} = T_2^T F T_1$$

With the information contained in F, we can now focus our attention on the projection matrices P and P' used in both image rectification and as the projection matrices to transform image pixel coordinates to 3-D world coordinates. The first step in computing these projection matrices is to compute the epipoles $\vec{e}$ and $\vec{e'}$ of both images as homogeneous vectors. We do this by solving the equations:

$$F\vec{e} = 0 \text{ and } \vec{e'}^T F = 0$$

We then compute P and P' using the canonical camera configuration approach, as:

$$P = [I|0] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$P' = [[\vec{e'}]_x F | \vec{e'}] \text{ where } [\vec{e'}]_x = \begin{pmatrix} 0 & -e'_z & e'_y \\ e'_z & 0 & -e'_x \\ -e'_y & e'_x & 0 \end{pmatrix} \text{ with } \vec{e'} = \begin{pmatrix} e'_x \\ e'_y \\ e'_z \end{pmatrix}$$

We now have all the information we need to rectify both input images and compute a 3-D projective reconstruction of the scene. However, especially when initial correspondences are selected manually, it is likely that the results calculated above may be imperfect. This may cause corresponding pixels between the two images to not fall in exactly the same row in each image after rectification. One way to combat this is to use nonlinear least-squares optimization to minimize this error. The Levenberg-Marquardt (LM) algorithm was used for this assignment to refine F (and subsequently recalculate P and P') to ensure proper image rectification. To do this, the cost function computes the 3-D world coordinates of each point and then reprojects them back into the image plane to compute an error metric. The calculation of the 3-D world coordinates from pixel correspondences is detailed in the "3-D projective reconstruction" section below. With the initial point correspondences as $(\vec{x_i}, \vec{x'_i})$ and the reprojected correspondences as $(\vec{\hat{x}_i}, \vec{\hat{x}'_i})$, the LM cost function is given by:

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

This will minimize the error in F and ultimately refine the projection matrices P and P'.

The final step is to rectify each input image through a homography that takes into account the epipoles and projections for each image. The homographies H and H' to rectify images 1 and 2, respectively, can be obtained through the following process, first by computing H', then H:

1. Compute H':
   (a) Compute the translation matrix T that sends the image center to the origin, given by:
   $$T = \begin{pmatrix} 1 & 0 & -w/2 \\ 0 & 1 & -h/2 \\ 0 & 0 & 1 \end{pmatrix}$$
   where w is the input image width, and h is the input image height.
   (b) Compute the angle θ necessary to rotate the epipole to be parallel with the x-axis:
   $$\theta = tan^{-1}(\frac{e'_y - h/2}{-e'_x - w/2})$$
   (c) Use the angle θ to compute the rotation matrix R to perform the rotation:
   $$R = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
   (d) Compute the scale factor f to send the epipole to infinity:
   $$f = (e'_x - w/2) * cos(\theta) - (e'_y - h/2) * sin(\theta)$$
   (e) Compute the transformation matrix G to send the epipole to infinity:
   $$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/f & 0 & 1 \end{pmatrix}$$
   (f) Compute an initial homography to rectify the image center as $H'_{center} = GRT$
   (g) Apply H'_{center} to the image center to rectify it. The rectified center is now $(\tilde{x}, \tilde{y})$.
   (h) Compute the translation matrix T_2 to move the rectified image center back to the true image center:
   $$T_2 = \begin{pmatrix} 1 & 0 & w/2 - \tilde{x} \\ 0 & 1 & h/2 - \tilde{y} \\ 0 & 0 & 1 \end{pmatrix}$$
   (i) Construct the final homography to rectify image 2 as $H' = T_2GRT$.
2. Compute H:
   (a) Repeat steps (a) through (f) above using image 1 and its epipoles to compute an initial $\tilde{H}$.
   (b) The homography is then given by the matrix that minimizes the distance between transformed points. This is equivalent to finding the homography that minimizes the cost:
   $$cost = \sum_i (dist(\tilde{H}x_i, H'x'_i))$$
   or equivalently minimizing:
   $$cost = \sum_i (ax_i + by_i + c - x_i^2)$$
   We solve the problem using linear least-squares to obtain values for a, b, and c.

(c) Compute the homography to rectify the center of image 1:

$$H_{center} = \begin{pmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tilde{H}$$

(d) Obtain the translation matrix T_1 to move the rectified image center back to the true image center as in steps (g) and (h) above.

(e) The final homography is given by $H = T_1 \tilde{H}$

The two homographies H and H' calculated above can then be applied to the pixels in images 1 and 2, respectively, to rectify them. The rectification should produce images such that the pixel associated with a specific 3-D world point appears in the same row in both rectified images.

1.1.2 Interest point detection

To obtain a robust 3-D reconstruction, a large number of correspondences are needed between the two images. One way a large number of correspondences can be obtained is by performing edge detection on the rectified images to extract pixels of interest in the scene (perhaps the edges and corners of the object we wish to reconstruct). Since we know that corresponding world points in the scene must lie in the same row of two rectified images, for each edge pixel in one image, we search the same row in the other image. Depending on the nature of how the images were taken, there may be a rightward or leftward polarity in which the search can be conducted, which speeds up the algorithm. Finally, we match interest points by asserting that the order of interest points in the direction of polarity along an epipolar line (in each image row) must be the same in both images. The SSD or NCC metrics can also be used to filter correspondences or break ties if there are several potential correspondences for a single pixel.

1.1.3 3-D projective reconstruction

With many correspondences between two stereo rectified images, it is possible to reconstruct the object of interest in 3-D using projective reconstruction. The process used to extract world coordinates $\vec{X}$ from corresponding image points $\vec{x} = (x, y)$ and $\vec{x'} = (x', y')$ and the camera projection matrices P and P' is outlined below. Note the representation for P:

$$P = \begin{pmatrix} \vec{P_1}^T \\ \vec{P_2}^T \\ \vec{P_3}^T \end{pmatrix}$$

1. Compute the 4×4 matrix A, given by:

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

2. Solve the homogeneous system $A\vec{X} = 0$ using linear least-squares. The solution $\vec{X}$ is given by the smallest eigenvector of $A^T A$.

3. Homogenize the solution vector $\vec{X}$ by dividing by its last component.

The process above can be used to obtain several 3-D points on a sort of "mesh" that defines the object of interest. Interpolation between the points can be used to estimate the shape of the object. It is important to note, however, that the reconstruction will not be perfect, especially when uncalibrated cameras are used. Instead, the reconstruction can only be accurate up to a form with projective, affine, and similarity distortion. The calibrated camera parameters are required to obtain a reconstruction as it appears in the real world.

1.2 Dense stereo matching

Dense stereo matching refers to a relatively robust algorithm for finding correspondences in a pair of stereo images. It is also capable of identifying occluded regions between the images, which contain invalid pixels (pixels that cannot be matched because they only appear in one of the images). The benefit of this method over others like a Canny edge detector for finding correspondences is that it can match all pixels in a pair of rectified images, rather than just a subset.

Dense stereo matching is carried out through a process known as the Census transform. Starting with a pair of stereo rectified images and known polarity between the images, the process proceeds as follows:

1. For a pixel in the left image, note the intensity values of the pixels in an $M \times N$ neighborhood around it. This neighborhood does not need to be square, but it can be.
2. Create a second $M \times N$ neighborhood around the pixel in the right image that has the same coordinates as the pixel in the left image that is currently being analyzed.
3. Apply a binary threshold to each neighborhood such that elements strictly greater than the center element become 1, while those less than or equal to the center element become 0.
4. Perform a bitwise XOR of the two binary neighborhoods.
5. Count the number of 1s in the result of the previous step. This will be the cost value associated with the pixel in the right image.
6. Staying in the same row, consider the pixel one column over in the right image in the direction of polarity. In other words, now consider the pixel at a distance d from the original pixel along the direction of polarity, but still in the same row as the original. Repeat steps 2-5 for this pixel.
7. Repeat step 6 d_{max} times. This will produce a cost vector of length d_{max} associated with the pixel of interest in the left image.
8. Find the minimum value of the cost vector, and the index of its first occurrence in the vector. This index corresponds to the disparity value d associated with the pixel in the left image.
9. Repeat steps 1-8 for all pixels in the left image to find the disparity value associated with each. The image formed by all disparity values is known as the *disparity map*, which highlights the disparity, or pixel difference, between corresponding world points in a pair of stereo images.

One useful result of disparity maps is their ability to highlight separate objects at different distances from the camera. Parallax determines how much objects appear to move based on their distance from the imaging source: objects that are closer appear to move further between views, while objects that are further away do not move as much. These differences in motion are represented by the disparity map: objects that are closer move further, and thus have a higher disparity value and are represented by a brighter color. Those that are further away have a lower disparity value and therefore a darker color. Thus, disparity maps provide a good representation of the depth of a scene.

2 Implementation notes

2.1 Task 1: 3-D projective reconstruction

- Much of my implementation for image rectification was referenced from past years' reports, mainly report 2 from Fall 2018 and the report from Fall 2008. We were allowed to use open-source/past years solutions as reference for this, as mentioned in class.
- After image rectification, I found that my images had rightward polarity, so I only needed to search the second image in the rightward direction starting from the pixel column in the left image.
- A maximum search distance was applied to the interest point search to speed up calculations and to prevent some invalid correspondences from being selected. This value, set to 45 pixels, was selected empirically after analyzing the rectified images.
- The SSD metric was used to filter out poor correspondences. However, I left some outliers in my final results to show that it is difficult to determine correspondences perfectly.
- To ensure that only one correspondence in the right image was selected for each pixel in the left image, I removed the pixel in the right image when it was matched to a left-image pixel. This way it could not be selected again.

2.2 Task 2: Dense stereo matching

- The d_{max} value found from the ground truth disparity map was $d_{max} = 14$. This value was held constant while neighborhood sizes were adjusted.
- I used a square neighborhood ($M \times M$) for computing the disparity map.
- The polarity between the two images was found to be a leftward polarity from the left image to the right.
- To improve computational efficiency, to avoid overly complex code, and to reduce the chance of inconsistent results across the image, d_{max} was held fixed for each pixel. Pixels that were close enough to the image border such that not all disparity values or full neighborhoods could be checked were ignored.
  - This created a border around the resulting disparity maps where pixels were not checked. This was assumed to be acceptable, as the ground truth disparity map also had a similar border.
  - The border size was determined by: $borderSize = d_{max} + \lfloor \frac{M}{2} \rfloor$
- When computing the error between the ground truth and calculated disparity maps, the border region and any occluded regions were ignored. That is, the error was calculated and then the occlusion mask was applied to ensure those pixels were not counted.
- Instead of using a colormap to display the disparity map, I scaled the map such that the pixels would have an even distribution between 0-255. Since the initial map values only ranged from 0-14, the pixel value differences are easily distinguishable without a colormap.

3 Results

3.1 Task 1: 3-D projective reconstruction

Below are my results for Task 1, which involved creating a 3-D reconstruction of a box from a pair of images. I have not explicitly included any images of "before/after LM refinement" because the refinement produced negligible improvement which is not noticeable in the images. I found that before LM refinement, the maximum row deviation between corresponding points in the rectified images was 3 pixels, but afterwards it dropped to near zero (sub-pixel level).

Figure 1: Input image 1 (left image) with manually selected correspondences in green.

Figure 2: Input image 2 (right image) with manually selected correspondences in green.

Figure 3: Rectified images.

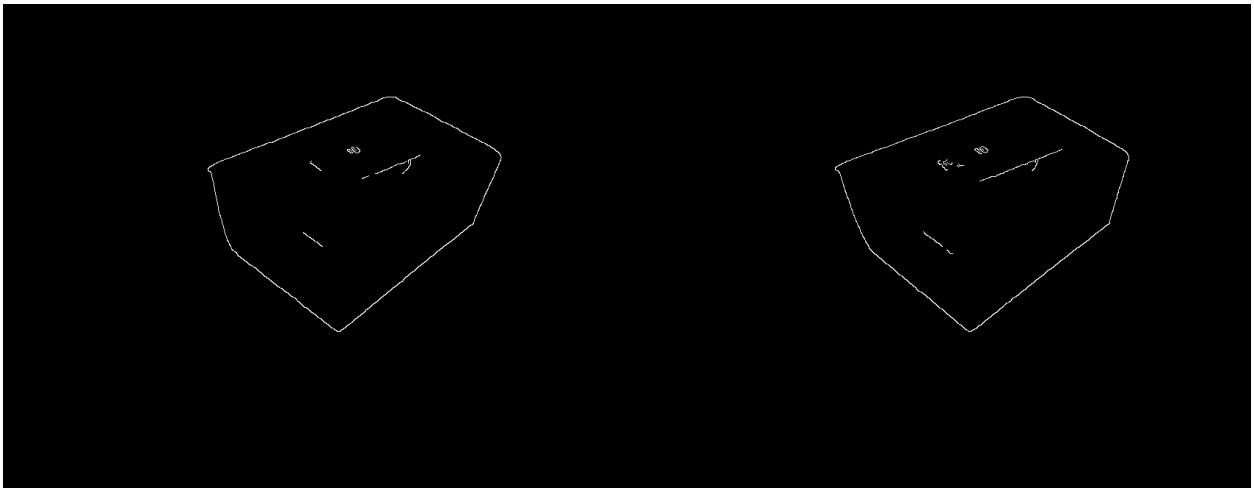

Figure 4: Rectified images after Canny edge detection to identify interest points.

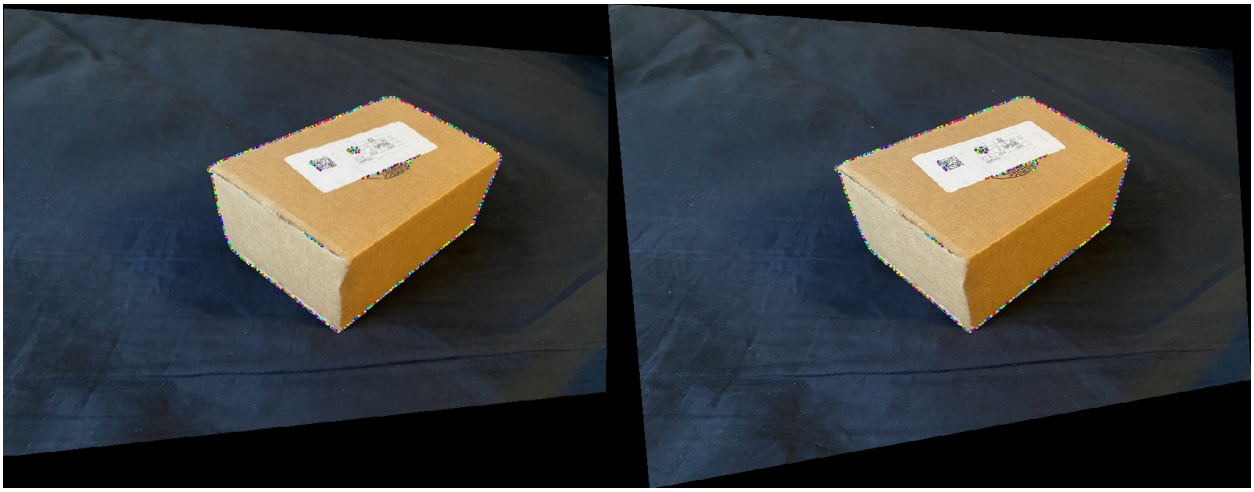

Figure 5: All detected correspondences between the two rectified images (color-coded).

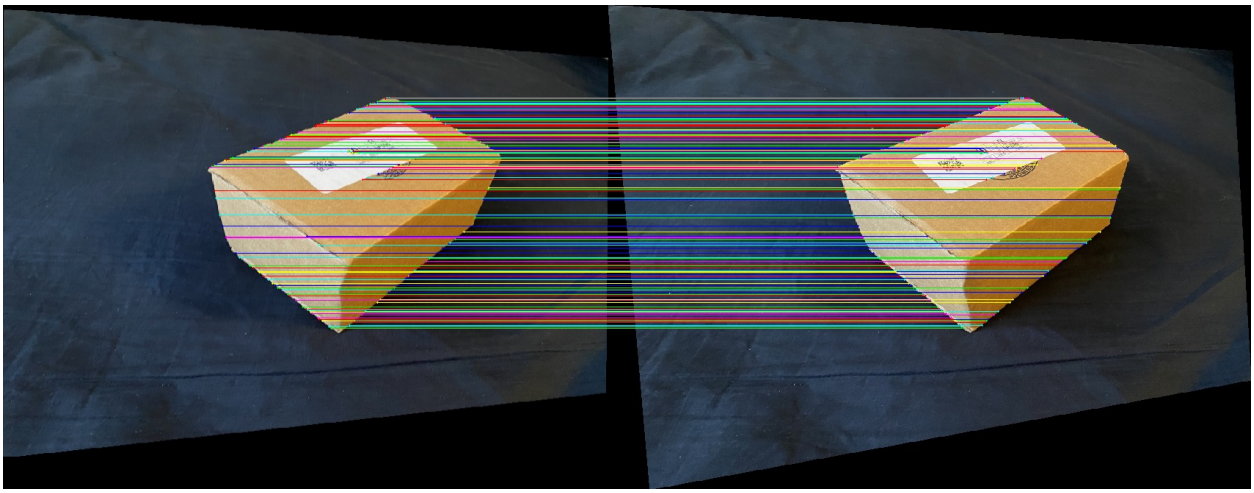

Figure 6: Selected correspondences between the two rectified images, with lines drawn for clarity. Not all correspondences are shown here, as the image would not be understandable.

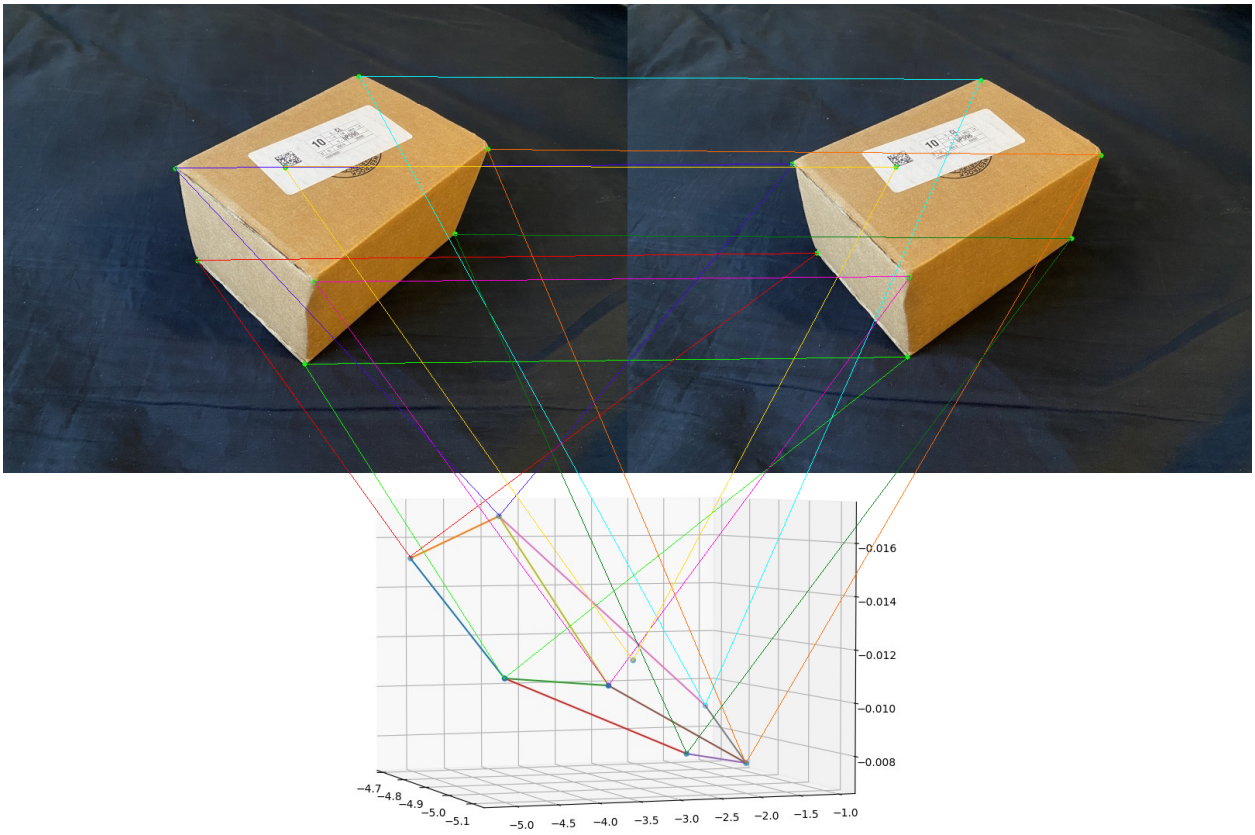

Figure 7: Initially selected correspondences and the corresponding 3-D reconstruction. I find this hard to interpret easily, so see below for an image with larger dots and no lines.

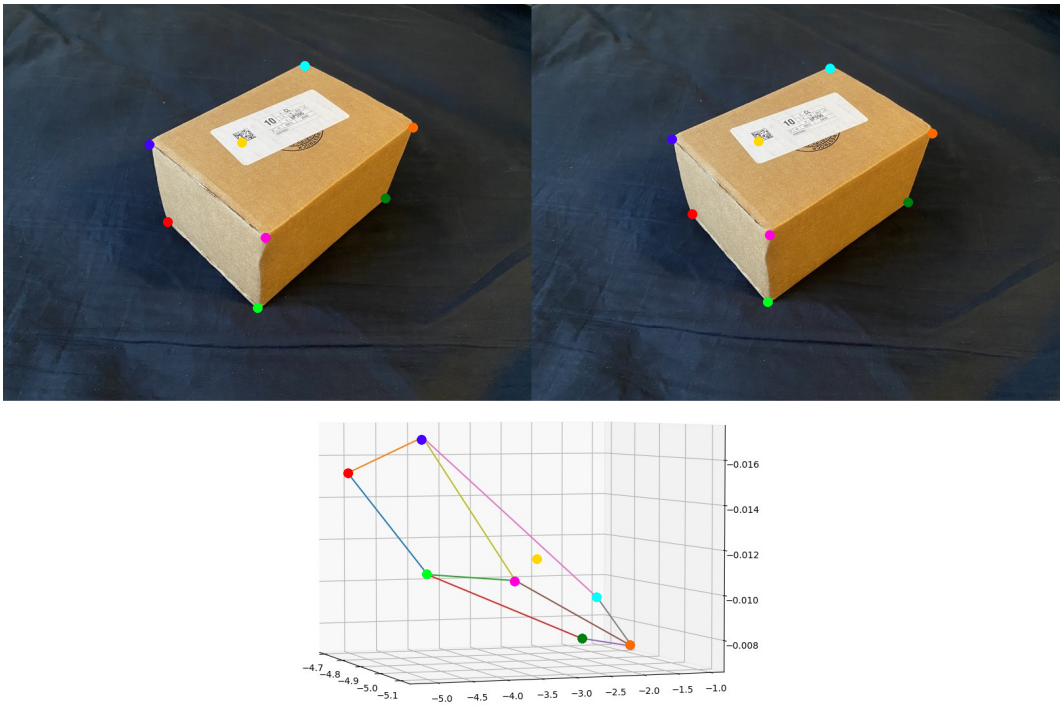

Figure 8: Initially selected correspondences and the corresponding 3-D reconstruction with no lines drawn between correspondences for clarity. The color indicates the correspondences.

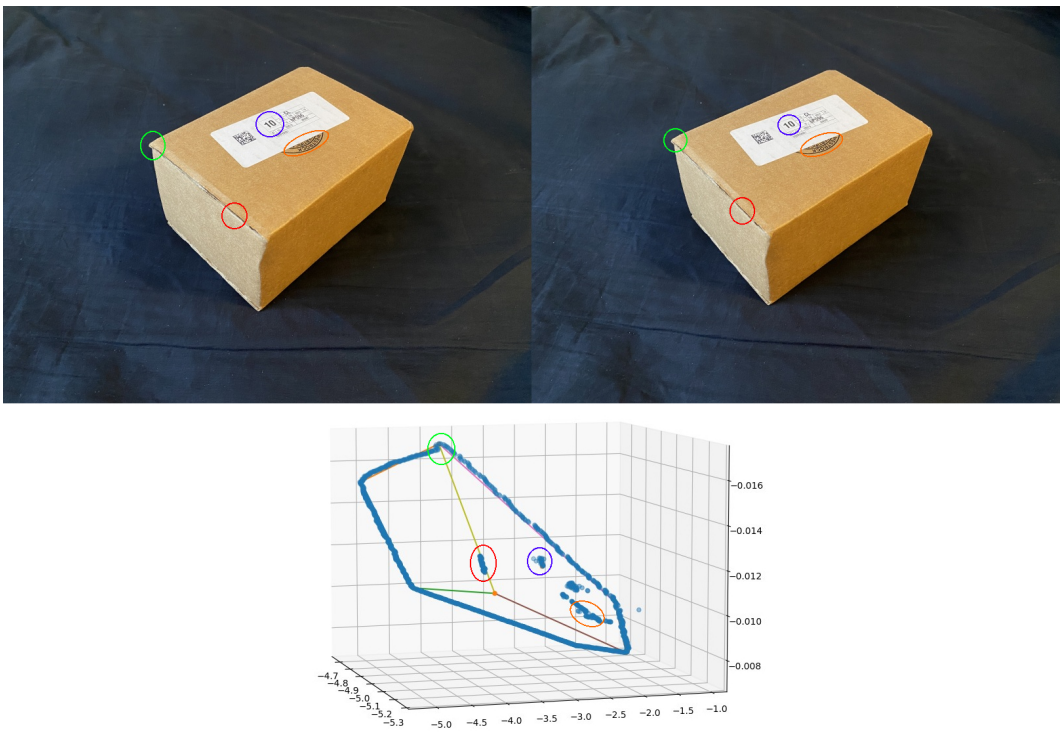

Figure 9: Notable areas in the initial images that can be seen in the 3-D reconstruction. Corresponding areas are circled in the same color. This 3-D reconstruction plot was formed using all correspondences found between the two Canny edge rectified images.

3.2 Task 2: Dense stereo matching

For this task, a pair of stereo rectified images was provided, and we were asked to compute the disparity map for multiple neighborhood sizes. A few notes on the results:

- The quality of the output disparity map seems to increase with M. As can be seen, there is a lot more noise when M is smaller, likely due to the fact that the neighborhood does not encompass a large enough region of the object in the window (i.e. it is doing a more local analysis, rather than global).
- We begin to see the finer details in depth (such as the contours on the face) once M surpasses d_{max}. I believe this is likely a coincidence, though.
- d_{max} was taken from the ground truth image, but it can also be measured by determining the maximum pixel distance a point on an object nearest the camera moves between the two images.

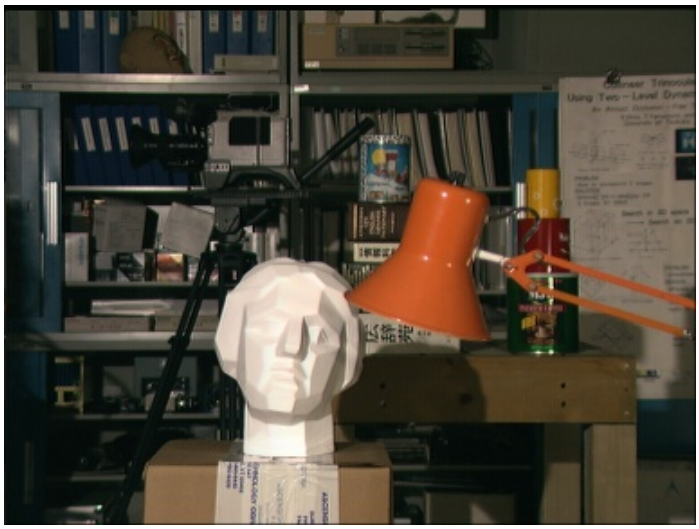

Figure 10: The input image from the left camera viewpoint.

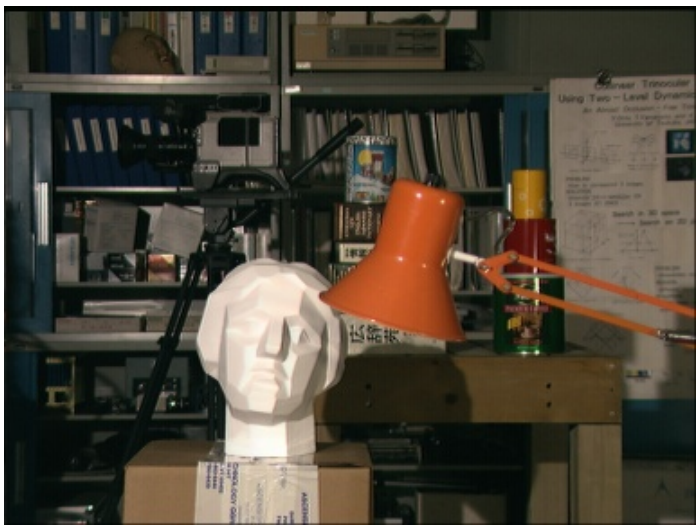

Figure 11: The input image from the right camera viewpoint.

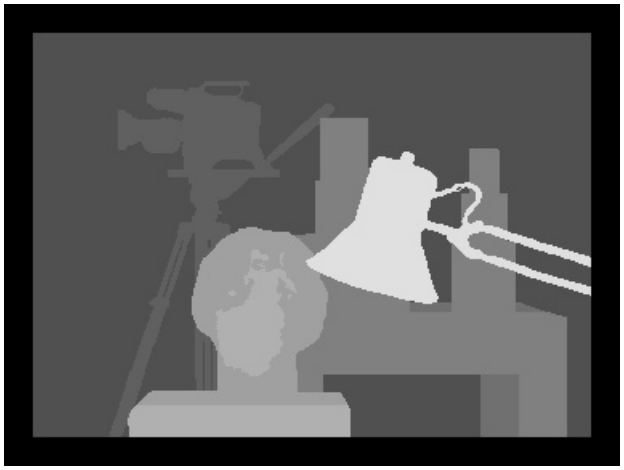

Figure 12: Ground truth disparity map.

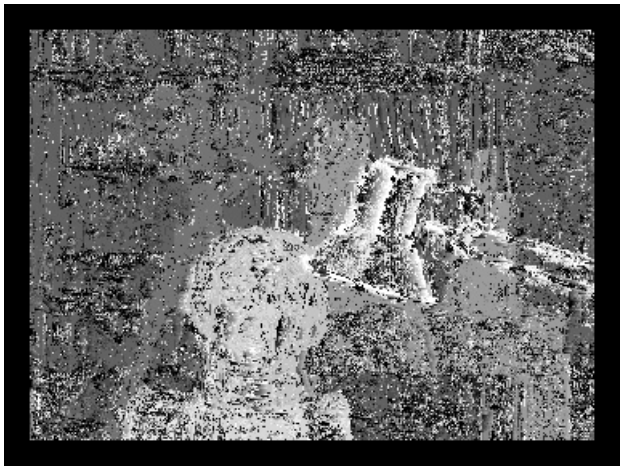

Figure 13: Disparity map for $d_{max} = 14, M = 5$.

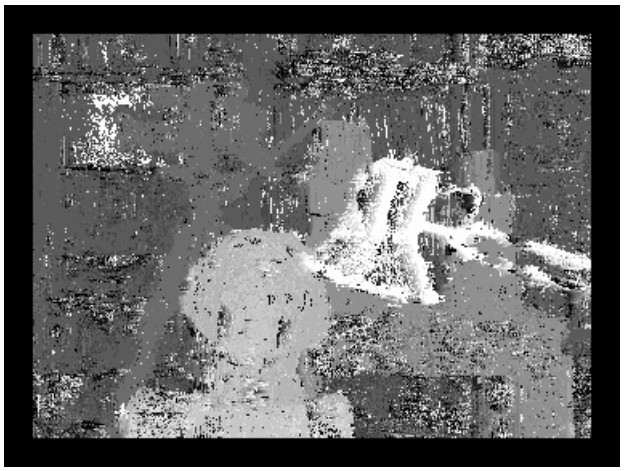

Figure 14: Disparity map for $d_{max} = 14, M = 9$.

Figure 15: Disparity map for $d_{max} = 14, M = 17$.

Figure 16: Disparity map for $d_{max} = 14, M = 31$.

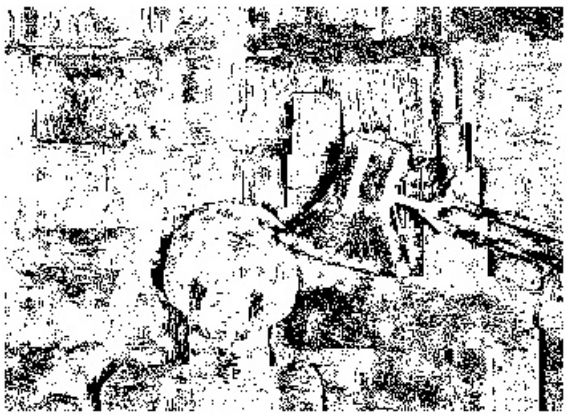

Figure 17: Error mask for $d_{max} = 14, M = 9$ showing white pixels where $error \leq 1$. The accuracy of this disparity map ignoring the occluded pixels is 75.356%.

Figure 18: Error mask for $d_{max} = 14, M = 9$ showing white pixels where $error \leq 2$. The accuracy of this disparity map ignoring the occluded pixels is 80.444%.

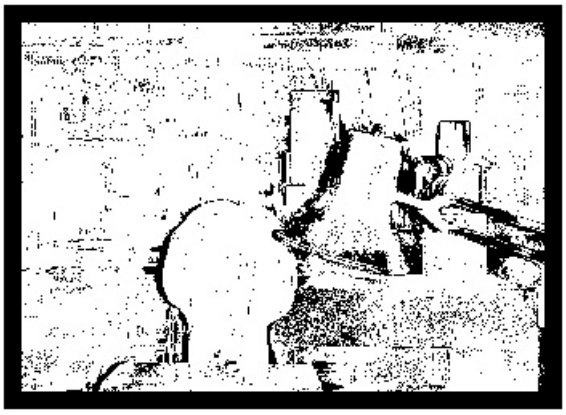

Figure 19: Error mask for $d_{max} = 14, M = 31$ showing white pixels where $error \leq 1$. Ignoring the border and occluded pixels, the accuracy of this disparity map is 93.127%.

Figure 20: Error mask for $d_{max} = 14, M = 31$ showing white pixels where $error \leq 2$. Ignoring the border and occluded pixels, the accuracy of this disparity map is 95.431%.

4 Source Code

4.1 Task 1: 3-D projective reconstruction

```
## ============= FILE INFORMATION ============= ##
#
# Name:         Brian Helfrecht
# Email:        bhelfre@purdue.edu
# Course:       ECE 661
# Assignment:   Homework 10, Task 1
# Due date:     November 16, 2020
#
## ============= PACKAGE/FILE IMPORTS ============= ##
import numpy as np
import cv2 as cv
import math
import time
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy import optimize
## ============= FUNCTION DEFINITIONS ============= ##
def calcEx(e):
    return np.array([[0, -e[2], e[1]],
                     [e[2], 0, -e[0]],
                     [-e[1], e[0], 0]])

def conditionF(uncondF):
    #Condition the matrix F to make it rank 2
    u, d, vh = np.linalg.svd(uncondF)
    dMat = np.array([[d[0], 0, 0], [0, d[1], 0], [0, 0, 0]])
    condF = np.dot(np.dot(u, dMat), vh)
    return condF

def calcF(normPts1, normPts2):
    A = np.zeros((8, 9))

    #Populate the matrix to find F
    for i in range(8):
        x1 = normPts1[i][0] #Image 1
        y1 = normPts1[i][1]
        x2 = normPts2[i][0] #Image 2
        y2 = normPts2[i][1]
        A[i] = np.array([x1*x2, x2*y1, x2, y2*x1, y2*y1, y2, x1, y1, 1])

    #Now, compute F by solving the linear least-squares problem: Af=0
    _, _, vh = np.linalg.svd(A)
    uncondFVec = vh[-1]
    uncondF = np.reshape(uncondFVec, (3, 3))

    #Note that we also have to condition F by zeroing the smallest eigenvalue.
    return conditionF(uncondF)

def normalizePoints(pts):
    xCoords = pts[:, 0]
    yCoords = pts[:, 1]

    #Compute the mean distance between all points
    xMean = np.mean(xCoords)
    yMean = np.mean(yCoords)
    dists = np.sqrt((xCoords-xMean)**2 + (yCoords-yMean)**2)
    meanDist = np.mean(dists)

    #Compute the transform matrix
    c = np.sqrt(2) / meanDist
    tMat = np.array([[c, 0, -c*xMean], [0, c, -c*yMean], [0, 0, 1]])

    #Apply the transform matrix to the points to normalize them
    #Also convert to HC representation.
    normPts = np.hstack((pts, np.ones((len(pts), 1))))
    normPts = np.transpose(np.dot(tMat, np.transpose(normPts)))
    return normPts, tMat

def calcE(F):
    #Compute epipoles using the null space vectors of F
    u, _, vh = np.linalg.svd(F)
    e1 = np.transpose(vh[-1, :])
    e2 = u[:, -1]
    e1 = e1 / e1[2] #Homogenize
    e2 = e2 / e2[2]
    Ex = calcEx(e2)
    return e1, e2, Ex

def calcP(F, e2, Ex):
    #Compute camera projection matrices
    P1 = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]])
    P2 = np.hstack((np.dot(Ex, F), np.transpose([e2])))
    return P1, P2

def costFunc(fVec, corrs):
    #Reference: HW9 Soln 2 from Fall 2018
    #Compute the cost for LM optimization
    F = np.reshape(fVec, (3, 3))
    [_, e2, Ex] = calcE(F)
    [P1, P2] = calcP(F, e2, Ex)
    img1Pts = np.hstack((corrs[0], np.ones((len(corrs[0]), 1))))
    img2Pts = np.hstack((corrs[1], np.ones((len(corrs[1]), 1))))
    error = []

    #Compute the world coordinates
    for i in range(np.size(corrs, 1)):
        A = np.zeros((4, 4))
        A[0] = img1Pts[i][0]*P1[2, :] - P1[0, :]
        A[1] = img1Pts[i][1]*P1[2, :] - P1[1, :]
        A[2] = img2Pts[i][0]*P2[2, :] - P2[0, :]
```

```
            A[3] = img2Pts[i][1]*P2[2, :] - P2[1, :]

            #Compute X
            _, _, vh = np.linalg.svd(A)
            vec = np.transpose(vh[-1, :])
            normVec = vec / vec[3] #Convert to unit vector

            #Estimate x's
            x1 = np.dot(P1, normVec)
            x2 = np.dot(P2, normVec)
            x1 = x1 / x1[2] #Convert to HC
            x2 = x2 / x2[2] #Convert to HC

            #Add coordinate errors for both images to error vector
            error.append(np.linalg.norm(x1 - img1Pts[i]) ** 2)
            error.append(np.linalg.norm(x2 - img2Pts[i]) ** 2)
        return np.ravel(error)

    def calcRectHomography(img, img1Pts, img2Pts, F, e1, e2, P1, P2):
        #Reference: HW9 Soln 2 from Fall 2018
        height = img.shape[0]
        width  = img.shape[1]

        '''===== Computation of H2 ====='''
        #Compute the translation matrix
        T = np.array([[1, 0, -width  / 2], \
                      [0, 1, -height / 2], \
                      [0, 0,           1]])

        #Compute the rotation matrix to move the epipoles to the x-axis
        angle = math.atan2(e2[1] - height/2, -(e2[0] - width/2))
        R = np.array([[math.cos(angle), -math.sin(angle), 0], \
                      [math.sin(angle),  math.cos(angle), 0], \
                      [              0,                0, 1]])

        #Compute the matrix to send the epipole to infinity
        f = (e2[0] - width/2)*math.cos(angle) - (e2[1] - height/2)*math.sin(angle)
        G = np.array([[    1, 0, 0], \
                      [    0, 1, 0], \
                      [-1/f, 0, 1]])

        #Compute and apply the homography to rectify the image center
        H2Center = np.dot(G, np.dot(R, T))
        img2CenterRect = np.dot(H2Center, np.array([width / 2, height / 2, 1]))
        img2CenterRect = img2CenterRect / img2CenterRect[2]

        #Now compute the translation matrix to move the rectified center
        T2 = np.array([[1, 0,  width/2 - img2CenterRect[0]], \
                       [0, 1, height/2 - img2CenterRect[1]], \
                       [0, 0,                           1]])

        #Compute the overall homography to rectify image 2
        H2 = np.dot(np.dot(T2, G), np.dot(R, T))
        H2 = H2 / H2[2, 2]

        '''===== Computation of H1 ====='''
        #Compute the rotation matrix to move the epipoles to the x-axis
        angle = math.atan2(e1[1] - height/2, -(e1[0] - width/2))
        R = np.array([[math.cos(angle), -math.sin(angle), 0], \
                      [math.sin(angle),  math.cos(angle), 0], \
                      [              0,                0, 1]])

        #Compute the matrix to send the epipole to infinity
        f = math.cos(angle)*(e1[0] - width/2) - math.sin(angle)*(e1[1] - height/2)
        G = np.array([[    1, 0, 0], \
                      [    0, 1, 0], \
                      [-1/f, 0, 1]])

        #Compute and apply the homography to rectify the image center
        H0 = np.dot(G, np.dot(R, T))

        img1Pts = np.hstack((img1Pts, np.ones((len(img1Pts), 1))))
        img2Pts = np.hstack((img2Pts, np.ones((len(img2Pts), 1))))
        x1 = np.transpose(np.dot(H0, np.transpose(img1Pts)))
        x2 = np.transpose(np.dot(H2, np.transpose(img2Pts)))
        x1[:, 0] = x1[:, 0] / x1[:, 2] #Convert to HC
        x1[:, 1] = x1[:, 1] / x1[:, 2]
        x1[:, 2] = x1[:, 2] / x1[:, 2]
        x2[:, 0] = x2[:, 0] / x2[:, 2] #Only need x-coords for img 2

        #Minimize using linear least-squares
        abc = np.dot(np.linalg.pinv(x1), x2[:, 0])

        #Establish Ha and use it to find H1Center
        Ha = np.array([[abc[0], abc[1], abc[2]], [0, 1, 0], [0, 0, 1]])
        H1Center = np.dot(Ha, H0)
        img1CenterRect = np.dot(H1Center, np.array([width / 2, height / 2, 1]))
        img1CenterRect = img1CenterRect / img1CenterRect[2]

        #Now compute the translation matrix to move the rectified center
        T1 = np.array([[1, 0,  width/2 - img1CenterRect[0]], \
                       [0, 1, height/2 - img1CenterRect[1]], \
                       [0, 0,                           1]])

        #Compute the overall homography to rectify image 1
        H1 = np.dot(T1, H1Center)
        H1 = H1 / H1[2, 2]

        return H1, H2

    def applyHomography(domainImg, homographyMat):
        #Apply the homography using each pixel in the RANGE image corresponding to a pixel in the DOMAIN.
        #This method ensures all pixels in the destination image are filled and does not leave gaps.
        startTime = time.time() #Track elapsed time

        #Synthesize a blank range image
        [width, height] = [domainImg.shape[1], domainImg.shape[0]]
```

```
    vertices = ((0, 0), (0, height-1), (width-1, height-1), (width-1, 0))
    print(vertices)
    xVerts = []
    yVerts = []

    #Calculate transformed vertex coordinates
    for i in range(4):
        x = vertices[i][0]
        y = vertices[i][1]
        [newX, newY, newZ] = np.dot(homographyMat, [x, y, 1.0]) #World to Image
        xVerts.append(round(newX / newZ))
        yVerts.append(round(newY / newZ))

    print(xVerts)
    print(yVerts)
    #Determine image size and offset coordinates
    imgWidth = max(xVerts) - min(xVerts)
    imgHeight = max(yVerts) - min(yVerts)

    #Determine aspect ratio for scaling
    aspectRatio = height / width
    newHeight = aspectRatio * imgWidth
    scaleX = 1.0
    scaleY = newHeight / imgHeight

    #Determine offsets for shifting coordinates to (0, 0)
    offsetX = min(xVerts)
    offsetY = min(yVerts)
    rangeImg = np.zeros((round(newHeight), round(imgWidth), 3), np.uint8)
    #rangeImg = np.zeros((round(imgHeight), round(imgWidth), 3), np.uint8)
    print(rangeImg.shape)

    hInv = np.linalg.pinv(homographyMat) #Invert to transform Image to World

    #Use vectorized operations to speed up calculations
    #when applying the homography.
    finalWidth = rangeImg.shape[1]
    finalHeight = rangeImg.shape[0]
    xyIdxs = np.indices((finalWidth, finalHeight))
    xIdxs  = xyIdxs[0].reshape(finalWidth*finalHeight, 1)
    yIdxs  = xyIdxs[1].reshape(finalWidth*finalHeight, 1)
    zIdxs  = np.ones((finalWidth*finalHeight, 1))
    finalIdxs = np.ndarray.astype(np.concatenate((xIdxs, yIdxs, zIdxs), 1), int)
    scaleIdxs = finalIdxs.copy()
    scaleIdxs[:, 0] = scaleIdxs[:, 0] / scaleX + offsetX
    scaleIdxs[:, 1] = scaleIdxs[:, 1] / scaleY + offsetY
    newIdxs = np.transpose(np.dot(hInv, np.transpose(scaleIdxs)))
    newIdxs[:, 0] = newIdxs[:, 0] / newIdxs[:, 2]
    newIdxs[:, 1] = newIdxs[:, 1] / newIdxs[:, 2]
    newIdxs = np.ndarray.astype(np.round(newIdxs), int)

    #Trim rows that correspond to points outside the domain image
    finalIdxs = finalIdxs[newIdxs[:, 0] >= 0] #Trim any x < 0
    newIdxs = newIdxs[newIdxs[:, 0] >= 0] #Trim any x < 0
    finalIdxs = finalIdxs[newIdxs[:, 1] >= 0] #Trim any y < 0
    newIdxs = newIdxs[newIdxs[:, 1] >= 0] #Trim any y < 0
    finalIdxs = finalIdxs[newIdxs[:, 0] < domainImg.shape[1]] #Trim any x > width
    newIdxs = newIdxs[newIdxs[:, 0] < domainImg.shape[1]] #Trim any x > width
    finalIdxs = finalIdxs[newIdxs[:, 1] < domainImg.shape[0]] #Trim any y > width
    newIdxs = newIdxs[newIdxs[:, 1] < domainImg.shape[0]] #Trim any y > width

    #Iterate over the remaining pixels. I could not find a better method for this (yet)
    for row in range(np.size(newIdxs, 0)):
        rangeImg[finalIdxs[row, 1]][finalIdxs[row, 0]] = domainImg[newIdxs[row, 1]][newIdxs[row, 0]]

    endTime = time.time()
    print('Elapsed time:', endTime - startTime) #Print elapsed time
    return rangeImg

def extractEdges(img):
    #Ensure we use a grayscale image
    if (len(img.shape) > 2):
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    gray = cv.blur(gray, (3, 3))
    edges = cv.Canny(gray, 350, 200, 5) #Extract edges
    return edges

def findCorrs(edges1, edges2, windowSize, maxSearchDist):
    #Find corresponding interest points between the two Canny edge images.

    #Ensure we use grayscale images
    if (len(edges1.shape) > 2):
        edges1 = cv.cvtColor(edges1, cv.COLOR_BGR2GRAY)
    if (len(edges2.shape) > 2):
        edges2 = cv.cvtColor(edges2, cv.COLOR_BGR2GRAY)

    corrs = []

    #Loop over each row in the left image and find correspondences in the right image
    for row in range(edges1.shape[0]):
        colIdxsL = np.where(edges1[row] > 0)[0]
        if np.size(colIdxsL) == 0: #If list is empty, continue
                continue

        #For each valid pixel in the left image, find the left-most pixel
        #in the same row of the right image
        for colL in colIdxsL:
            searchVecR = edges2[row, colL:colL + maxSearchDist + 1]
            potentialColsR = np.where(searchVecR > 0)[0]
            if np.size(potentialColsR) == 0: #If list is empty, continue
                continue

            #Determine the global image coordinate and append to list
            colR = potentialColsR[0] + colL
            edges2[row, colR] = 0 #Don't consider this pixel anymore
            corrs.append([[colL, row], [colR, row]])
    return corrs
```

```
def filterCorrs(img1, img2, corrs, winSz, maxCorrs):
    #Compute the SSD for each correspondence pair, and eliminate those
    #that do not meet a desired metric
    print('Filtering correspondences...')
    halfWinSz = int(winSz / 2)
    if (maxCorrs > len(corrs)):
        print('Warning: Desired maxCorrs is greater than the number of ' + \
              'correspondences detected!')
        print('Desired maxCorrs:', maxCorrs, ' Actual corrs:', len(corrs))
        maxCorrs = len(corrs)

    ssdVec = []

    for corrPair in corrs:
        corr1 = corrPair[0]
        corr2 = corrPair[1]
        win1 = img1[corr1[1] - halfWinSz:corr1[1] + halfWinSz + 1, \
                    corr1[0] - halfWinSz:corr1[0] + halfWinSz + 1]
        win2 = img2[corr2[1] - halfWinSz:corr2[1] + halfWinSz + 1, \
                    corr2[0] - halfWinSz:corr2[0] + halfWinSz + 1]

        ssdVec.append(np.sum((win1 - win2) ** 2))

    #Sort the correspondences by lowest SSD and take the best maxCorrs
    sortedCorrs = [pt for _, pt in sorted(zip(ssdVec, corrs), key=lambda pair: pair[0])]
    return sortedCorrs[0:maxCorrs]

def showCorrespondences(img1, img2, correspondences, skipX, showLines):
    #Find the image with the shortest height
    img1Height = img1.shape[0]
    img2Height = img2.shape[0]
    if (img1Height < img2Height): #Image 1 shorter
        img1 = np.concatenate((img1, np.zeros((img2Height-img1Height, img1.shape[1], 3), np.uint8)), 0)
    elif (img2Height < img1Height): #Image 2 shorter
        img2 = np.concatenate((img2, np.zeros((img1Height-img2Height, img2.shape[1], 3)), np.uint8), 0)

    newImg = np.concatenate((img1, img2), 1) #Stack images horizontally
    img2OffsetX = img1.shape[1]

    #Establish possible line colors
    # RBGCMY (randomly assigned to differentiate)
    colors = ((255,   0,   0), #R
              (  0, 255,   0), #G
              (  0,   0, 255), #B
              (  0, 255, 255), #C
              (255,   0, 255), #M
              (255, 255,   0)) #Y

    #Draw correspondence points and lines between them
    j = 0
    for i in range(len(correspondences)):
        if (not i % skipX == 0):
            continue
        ptSet = correspondences[i]
        colorIdx = j % len(colors)
        pt1 = tuple(ptSet[0])
        pt2 = tuple(np.array(ptSet[1]) + [img2OffsetX, 0])
        if (showLines):
            cv.line(newImg, pt1, pt2, colors[colorIdx], 1)
        cv.circle(newImg, pt1, 1, colors[colorIdx], -1)
        cv.circle(newImg, pt2, 1, colors[colorIdx], -1)
        j += 1

    return newImg

def projReconstruct(corrs, P1, P2):
    worldCoords = []

    #Compute the world coordinates
    for i in range(len(corrs)):
        A = np.zeros((4, 4))
        corr1 = corrs[i][0]
        corr2 = corrs[i][1]
        A[0] = corr1[0]*P1[2, :] - P1[0, :]
        A[1] = corr1[1]*P1[2, :] - P1[1, :]
        A[2] = corr2[0]*P2[2, :] - P2[0, :]
        A[3] = corr2[1]*P2[2, :] - P2[1, :]

        #Compute X
        _, _, vh = np.linalg.svd(np.dot(np.transpose(A), A))
        vec = np.transpose(vh[-1, :])
        worldCoords.append(vec / vec[3]) #Convert to 3-D HC vector
    worldCoords = np.reshape(worldCoords, (len(corrs), 4))
    return worldCoords

## ============= MAIN CODE BEGINS BELOW ============= ##

INITIAL_CORRS = np.array([[[251,  331], #Image 1 points
                           [222,  212],
                           [389, 464],
                           [402, 358],
                           [583,  296 ],
                           [626, 187],
                           [459, 93], #[432, 160],
                           [364, 211]],
                          [[244,  321], #Image 2 corresponding points
                           [212,  206],
                           [361, 455],
                           [363, 352],
                           [573,  302],
                           [611, 195],
                           [456, 98], #[420, 163],
                           [346, 210]]])

img1 = cv.imread('./inputs/1.JPG')
img2 = cv.imread('./inputs/2.JPG')
```

```
for i in range(8):
    cv.circle(img1, tuple(INITIAL_CORRS[0][i]), 3, (0, 255, 0), -1)
    cv.circle(img2, tuple(INITIAL_CORRS[1][i]), 3 , (0, 255, 0), -1)
cv.imwrite('./outputs/img1_select.jpg', img1)
cv.imwrite('./outputs/img2_select.jpg', img2)
#cv.imshow('img1', img1)
#cv.imshow('img2', img2)
#cv.waitKey(0)


#Normalize the correspondences
normPts1, tMat1 = normalizePoints(INITIAL_CORRS[0])
normPts2, tMat2 = normalizePoints(INITIAL_CORRS[1])
initialF = calcF(normPts1, normPts2) #Compute the fundamental matrix
finalFNoLM = np.dot(np.dot(np.transpose(tMat2), initialF), tMat1)
finalFNoLM = finalFNoLM / finalFNoLM[2, 2] #Homogenize F

#Find epipoles and projection matrices
#[e1NoLM, e2NoLM, ExNoLM] = calcE(finalFNoLM)
#[P1NoLM, P2NoLM] = calcP(finalFNoLM, e2NoLM, ExNoLM)

#Compute nonlinear least-squares minimization
fVec = np.ravel(finalFNoLM)
finalF = optimize.least_squares(costFunc, fVec, \
        args = [INITIAL_CORRS], method = 'lm').x

#finalF = finalFNoLM

#Compute the refined parameters
finalF = np.reshape(finalF, (3, 3))
finalF = conditionF(finalF)
finalF = finalF / finalF[2, 2]
[e1, e2, Ex] = calcE(finalF)
[P1, P2] = calcP(finalF, e2, Ex)

''' ===== BEGIN IMAGE RECTIFICATION ===== '''
#Compute the homographies and use them to rectify the images
H1, H2 = calcRectHomography(img1, INITIAL_CORRS[0], INITIAL_CORRS[1], finalF, e1, e2, P1, P2)

img1Rect = applyHomography(img1, H1)
img2Rect = applyHomography(img2, H2)

#cv.imshow('img1Rect', img1Rect)
#cv.imshow('img2Rect', img2Rect)
#cv.imwrite('./outputs/img1Rect.jpg', img1Rect)
#cv.imwrite('./outputs/img2Rect.jpg', img2Rect)
#cv.waitKey(0)

''' ===== BEGIN INTEREST POINT DETECTION ===== '''
edges1 = extractEdges(img1Rect)
edges2 = extractEdges(img2Rect)

#cv.imshow('edges1', edges1)
#cv.imshow('edges2', edges2)
#cv.imwrite('./outputs/edges1.jpg', edges1)
#cv.imwrite('./outputs/edges2.jpg', edges2)
#cv.waitKey(0)

''' ===== BEGIN INTEREST POINT MATCHING ===== '''
corrs = findCorrs(edges1, edges2, 21, 45)
filteredCorrs = filterCorrs(img1Rect, img2Rect, corrs, 15, 1e6)
corrImg = showCorrespondences(img1Rect, img2Rect, filteredCorrs, 10, False)
cv.imwrite('./outputs/corrImg.jpg', corrImg)
#cv.imshow('corrImg', corrImg)
#cv.waitKey(0)

''' ===== BEGIN PROJECTIVE RECONSTRUCTION ===== '''
#Rectify the inital corner correspondences (written manually for ease of use)
rectCorrs = [[[311, 332], [355, 332]],
             [[279, 225], [309, 225]],
             [[457, 445], [493, 445]],
             [[472, 352], [489, 352]],
             [[641, 297], [683, 297]],
             [[679, 209], [709, 209]],
             [[526, 126], [566, 126]],
             [[433, 226], [462, 226]]]

trueCorners = projReconstruct(rectCorrs, P1, P2)
worldCoords = projReconstruct(filteredCorrs, P1, P2)

#Set up the plotting figure
#Reference: https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(worldCoords[:, 0], worldCoords[:, 1], worldCoords[:, 2])
ax.scatter(trueCorners[:, 0], trueCorners[:, 1], trueCorners[:, 2])

pairs = [[0, 2],
         [0, 1],
         [2, 3],
         [2, 4],
         [4, 5],
         [5, 3],
         [1, 6],
         [5, 6],
         [1, 3]]

for pair in pairs:
    ax.plot([trueCorners[pair[0]][0], trueCorners[pair[1]][0]], \
            [trueCorners[pair[0]][1], trueCorners[pair[1]][1]], \
            [trueCorners[pair[0]][2], trueCorners[pair[1]][2]])
plt.show()
```

4.2 Task 2: Dense stereo matching

```
## ============= FILE INFORMATION ============= ##
#
# Name:         Brian Helfrecht
# Email:        bhelfre@purdue.edu
# Course:       ECE 661
# Assignment:   Homework 10, Task 2
# Due date:     November 16, 2020
#
## ============= PACKAGE/FILE IMPORTS ============= ##
import numpy as np
import cv2 as cv
## ============= FUNCTION DEFINITIONS ============= ##
def applyCensus(imgL, imgR, M, dMax):
    #Ensure we use grayscale images
    if (len(imgL.shape) > 2):
        imgL = cv.cvtColor(imgL, cv.COLOR_BGR2GRAY)
    if (len(imgR.shape) > 2):
        imgR = cv.cvtColor(imgR, cv.COLOR_BGR2GRAY)

    halfWinSz = int(M / 2)
    borderSz = dMax + halfWinSz

    width, height = imgL.shape[1], imgL.shape[0]
    dMap = np.zeros((height, width))

    #Iterate over all rows in the left image
    for rowL in range(borderSz, height - borderSz):
        print('Analyzing row %d/%d...' % (rowL + 1, height))

        #Iterate over all pixels (cols) in the row
        for colL in range(width - borderSz - 1, borderSz - 1, -1):
            costVec = []
            windowL = imgL[rowL - halfWinSz:rowL + halfWinSz + 1, \
                           colL - halfWinSz:colL + halfWinSz + 1]
            binWinL = np.ravel((windowL > imgL[rowL, colL]) * 1)

            #Iterate over disparity values to create right image windows
            for d in range(dMax + 1):
                rowR = rowL
                colR = colL - d
                windowR = imgR[rowR - halfWinSz:rowR + halfWinSz + 1, \
                               colR - halfWinSz:colR + halfWinSz + 1]
                binWinR = np.ravel((windowR > imgR[rowR, colR]) * 1)
                costVec.append(sum(binWinL ^ binWinR))
            dMap[rowL, colL] = np.argmin(costVec)

    dMap = dMap.astype(np.uint8)
    return dMap


## ============= MAIN CODE BEGINS BELOW ============= ##

M = 31 #9, 31

#Accuracy:
# - M = 9
#   d <= 1: 75.35639879210656
#   d <= 2: 80.44429878976568
# - M = 31
#   d <= 1: 75.35639879210656 -> 93.12662916723831
#   d <= 2: 80.44429878976568 -> 95.43147208121827

#Read in the ground truth disparity map to determine dMax
gtDMap8 = cv.imread('./inputs/Task2_Images/left_truedisp.pgm')
gtDMap8 = cv.cvtColor(gtDMap8, cv.COLOR_BGR2GRAY)
gtDMap16 = gtDMap8.astype(np.float32) / 16.0
gtDMap16 = gtDMap16.astype(np.uint8) #np.int16 produces same results but
                                     #leads to data type issues in OpenCV
dMax = np.max(gtDMap16)
print('dMax:', dMax)
#cv.imwrite('./outputs/gtDMap.jpg', gtDMap)

#Read in the input images
imgL = cv.imread('./inputs/Task2_Images/Left.ppm')
imgR = cv.imread('./inputs/Task2_Images/Right.ppm')
#cv.imwrite('./outputs/Left.jpg', imgL)
#cv.imwrite('./outputs/Right.jpg', imgR)

#Apply census transform to left image to estimate the disparity map.
dMap = applyCensus(imgL, imgR, M, dMax)
print(np.min(dMap), np.max(dMap))
dMapView = (dMap / np.max(dMap) * 255).astype(np.uint8) #Convert to view the mask
cv.imwrite('./outputs/dMap' + str(dMax) + '_' + str(M) + '.jpg', dMapView)
cv.imshow('dmap', dMapView)
#cv.waitKey(0)

#Compute the error
error = abs(dMap.astype(np.int16) - gtDMap16.astype(np.int16)).astype(np.uint8)
print('Error min/max:', np.min(error), np.max(error))
challenge1 = (error <= 1) * 255
challenge1 = challenge1.astype(np.uint8) #Ensure data type
challenge2 = (error <= 2) * 255
challenge2 = challenge2.astype(np.uint8) #Ensure data type

#validMask = cv.imread('./inputs/Task2_Images/mask0nocc.png')
validMask = cv.cvtColor(validMask, cv.COLOR_BGR2GRAY)
N = cv.countNonZero(validMask) #Compute number of valid pixels

#Apply masks to eliminate unconsidered error pixels
validErr1 = cv.bitwise_and(validMask, challenge1)
validErr2 = cv.bitwise_and(validMask, challenge2)

#Compute error in the results
d1Error = cv.countNonZero(validErr1) / N
```

```
d2Error = cv.countNonZero(validErr2) / N
print('d = 1 error:', d1Error)
print('d = 2 error:', d2Error)

#Show and save the resulting images
cv.imshow('challenge1', validErr1)
cv.imshow('challenge2', validErr2)
cv.imwrite('./outputs/error' + str(dMax) + '_' + str(M) + '.jpg', error)
cv.imwrite('./outputs/challenge1_' + str(dMax) + '_' + str(M) + '.jpg', challenge1)
cv.imwrite('./outputs/challenge2_' + str(dMax) + '_' + str(M) + '.jpg', challenge2)
cv.waitKey(0)
```