Shiva Ghose
Date: November 27, 2014

ECE661: Homework 8

Name: Shiva Ghose

Email: ghose0@purdue.edu

PUID: 00251 72564

Outline

The goal of this homework is to reconstruct a scene using two or more images from an uncalibrated camera. To this end, we use the normalized 8 point algorithm proposed by Richard Hartley [Hartley, 1997] in order to estimate the fundamental matrix between two images of a scene which were captured using uncalibrated cameras. The gist of the algorithm is as follows:

- Identify a set of correspondences between the two images (at least 7).
- Estimate the fundamental matrix for the two images. Estimate the epipoles in each image.
  - Compute a linear estimate of the fundamental matrix using a homogeneous least squares method.
  - Using the linear estimate of the fundamental matrix as a starting point, refine the result using a non-linear solver.
- Rectify the two images.
  - Send the epipole of each image to infinity along the x-axis.
  - Align the two images along the y-axis.
- Finally, we use the rectified images to limit our search space for further correspondences, and using the estimated camera projection matrices, we triangulate their position in 3D space.

Contents

1 Input images

I used the *arch of blocks* image set from Carnegie Mellon's stereo vision data set. It can be accessed at: *http://vasc.ri.cmu.edu//idb/html/stereo/arch/index.html*. I additionally adjusted contrast of each image to aid with correspondence detection. No other changes were made to the originals.

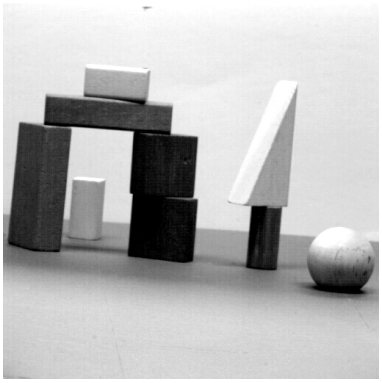

Figure 1: Input image 1.

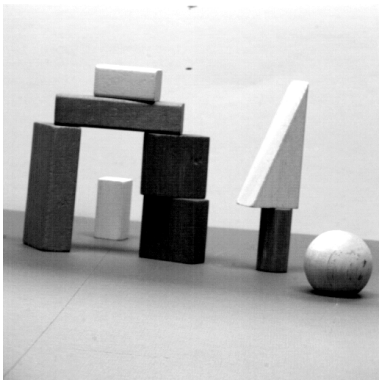

Figure 2: Input image 2.

2 Estimating correspondences

Finding good correspondences is a key aspect of estimating the fundamental matrix. To this end, I have implemented two methods to detect correspondences between the images:

2.1 Manual correspondences

This method requires the user to click on points of interest that are present in both images. The advantage with this method is that it uses human-level intelligence to accurately spot correspondences, hence we can use even the bare minimum number of points required to compute the fundamental matrix. However, a significant drawback is that while humans can identify correspondences easily, they cannot pinpoint the exact position of the correspondences. The accuracy of the correspondences greatly affects the outcome of the scene reconstruction process.

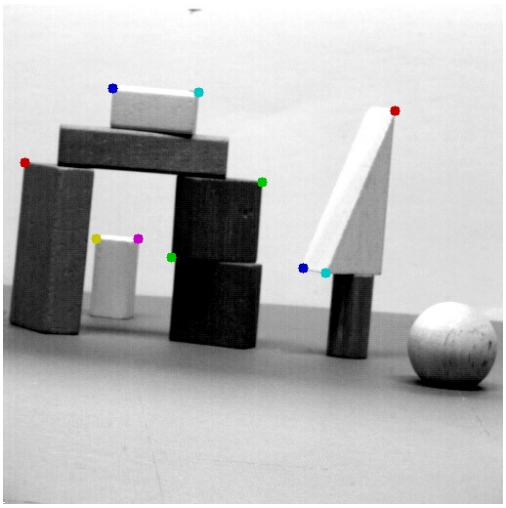

Figure 3: Points of interest manually marked on the first image

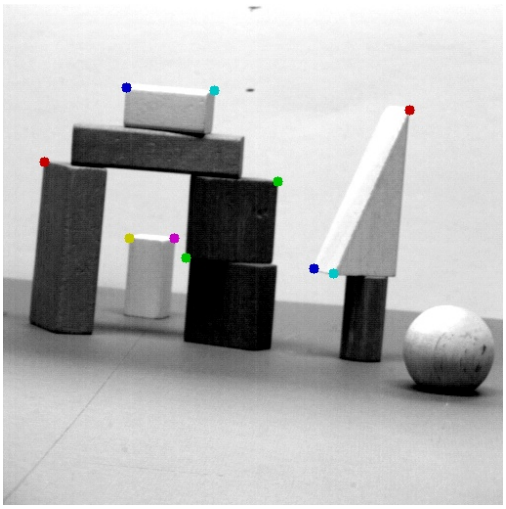

Figure 4: Points of interest manually marked on the second image

2.1.1 Improving accuracy using OpenCV's cornerSubPix

To overcome the accuracy issues of the above method, I used OpenCV's cornerSubPix method to pinpoint corners in the regions of correspondences selected by the user. Apart from the improvements in estimating the fundamental matrix, this method also sped up the correspondence marking process as the user did not have to focus too much on selecting the exact locations of corners.

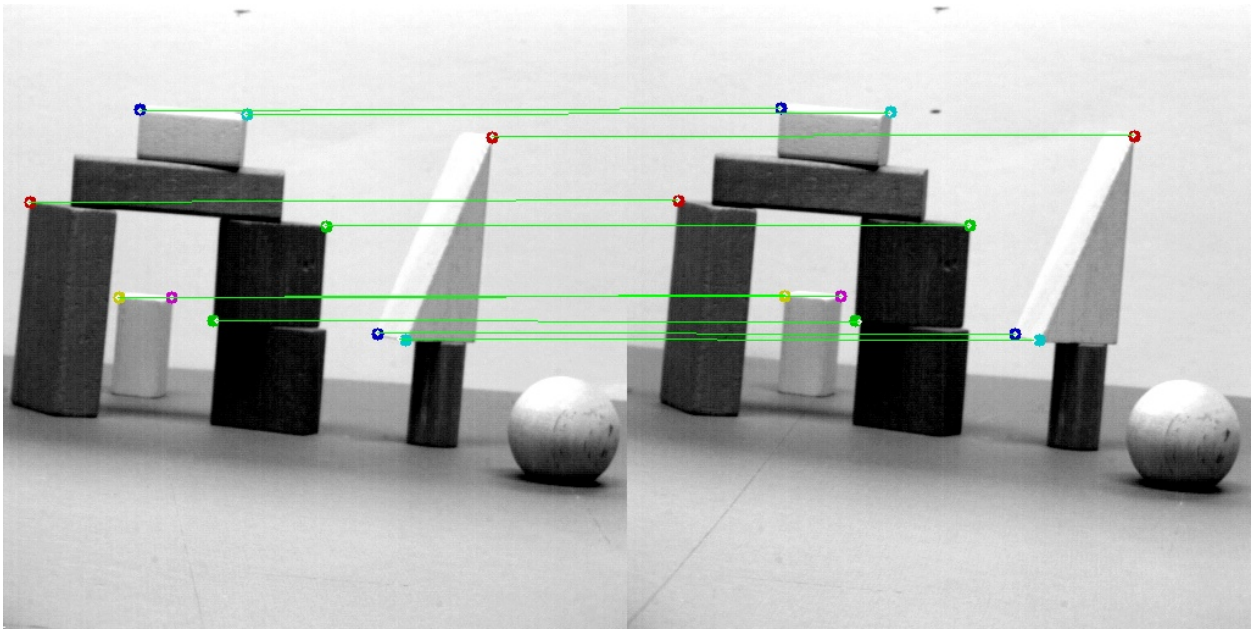

Figure 5: The white dots inside each colored circle is the location of sub-pixel estimate of the corner in the region associated with the circles. The correspondences between the two images are linked using the green lines.

2.2 Automatic correspondences

The manual point selection method was slow and often led to different estimates of the fundamental matrix. So, trading semantic corner accuracy for requiring more correspondences, I used the SIFT algorithm (addressed in the appendices) to quickly find sub-pixel estimates for correspondences between the two images.

The automatic correspondence detection method requires atleast 40 correspondences between the images. I used the Euclidean distance norm to compare the descriptors, and additionally used the ratio test to discard ambiguous matches. The figure below shows the result of the automatic correspondence detection routine. I started with a ratio of 0.2 and worked my way up to 0.9 in increments of 0.05 until I got atleast 40 correspondences.

Parameters used:

```
min_pts=40
starting_ratio=0.2,
max_threshold=0.9,
threshold_delta=0.05
```

This method has higher repeatability than the manual selection method, and hence this method is consistent across sessions. Also, the automatic correspondence detection method speeds up processing time as the program does not have to wait for user input.

A significant drawback of this method, however is that if bad correspondences are selected, it severely afftects the outcome of the fundamental matrix estimation.

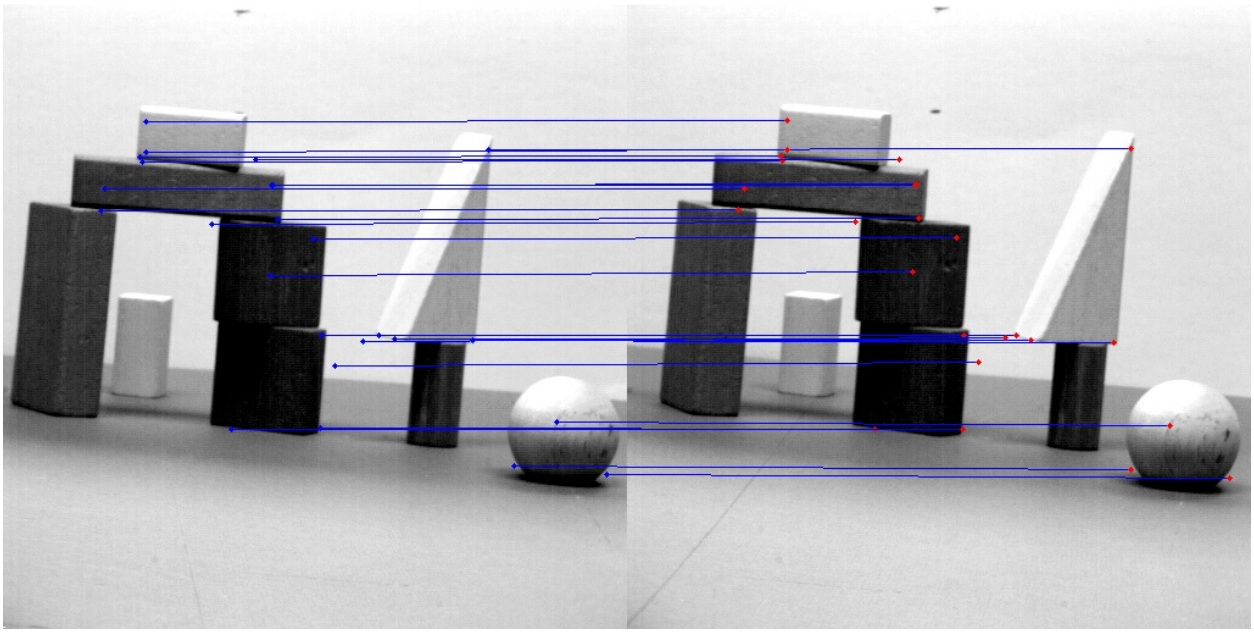

Figure 6: The colored dots indicate points of interest in each image. Corresponding points of interest between the two images are linked using the purple lines.

2.2.1 Effect of poor correspondence detection

The following images showcase the differences in the output generated by *bad* correspondences:

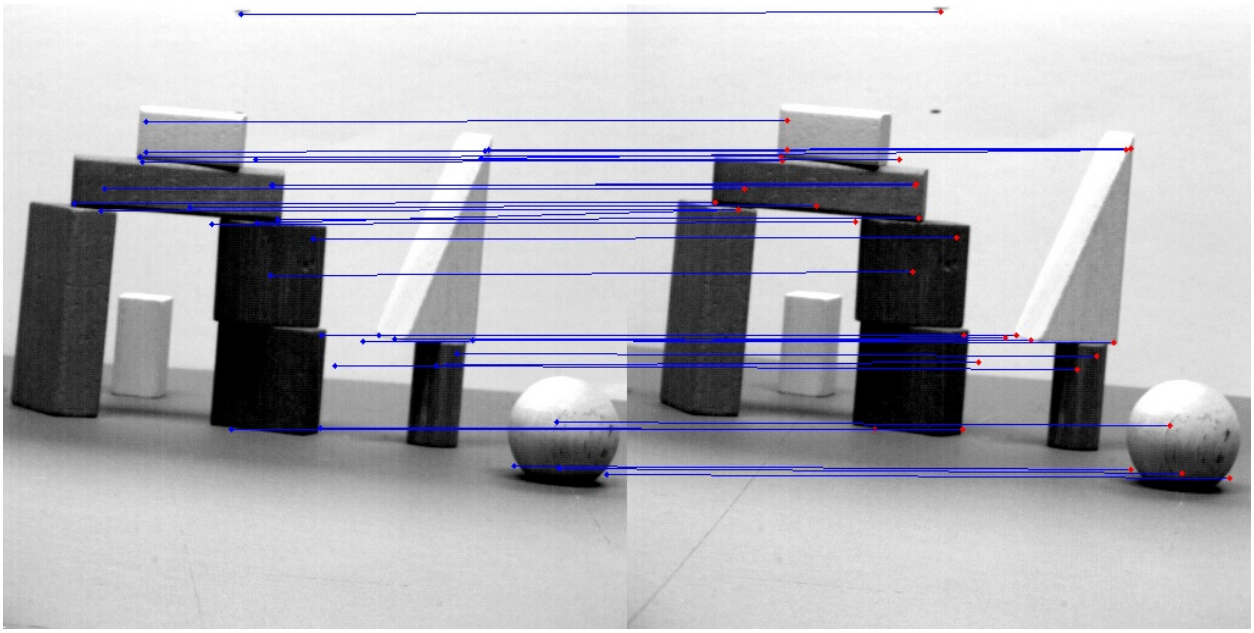

Figure 7: This image shows good correspondence matching between the two images.

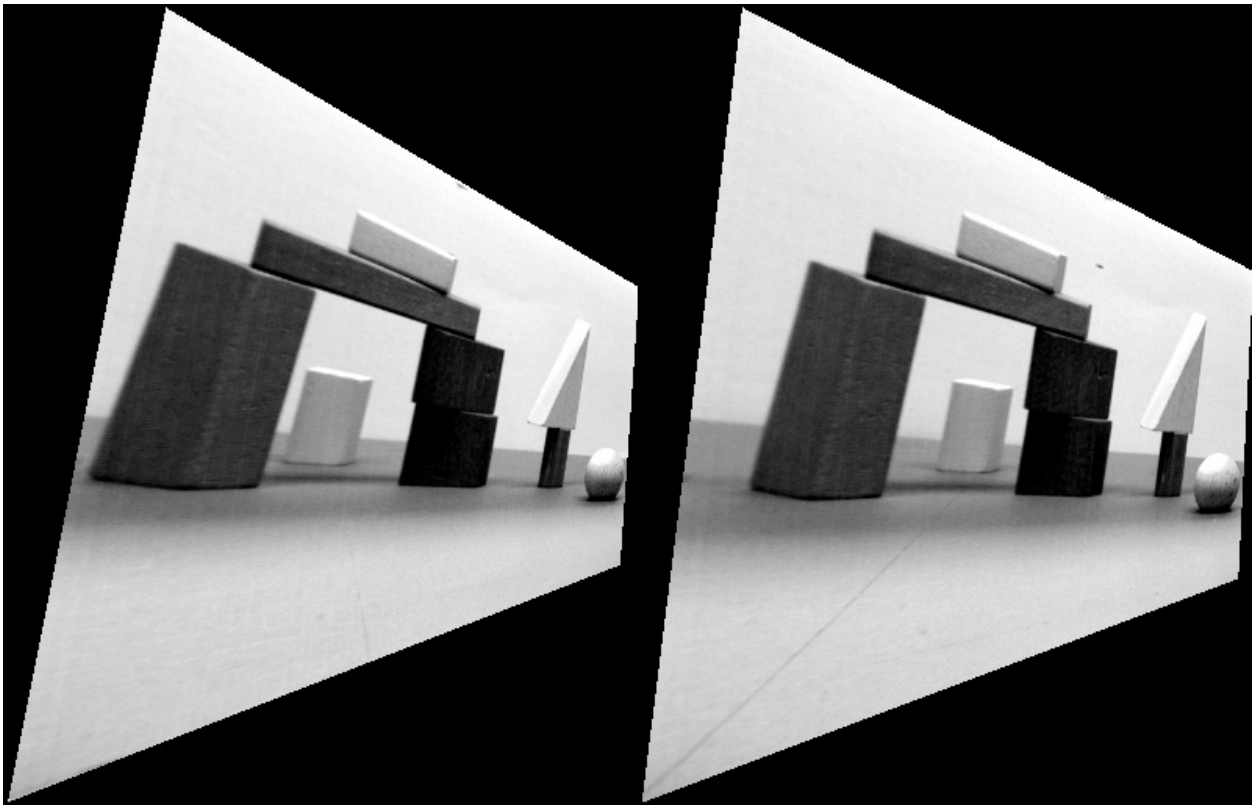

Figure 8: When the automatic algorithm works, it performs well enough- notice how the planes seem aligned and parallel.

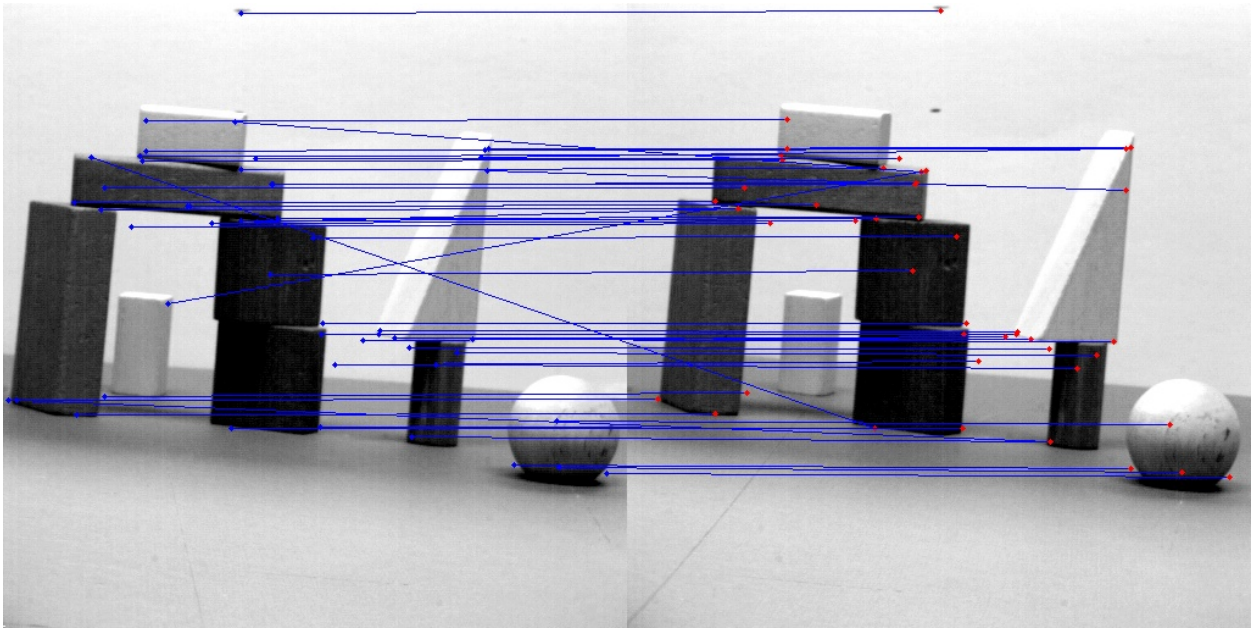

Figure 9: This image shows bad correspondence matching between the two images. Notice the criss-crossed lines.

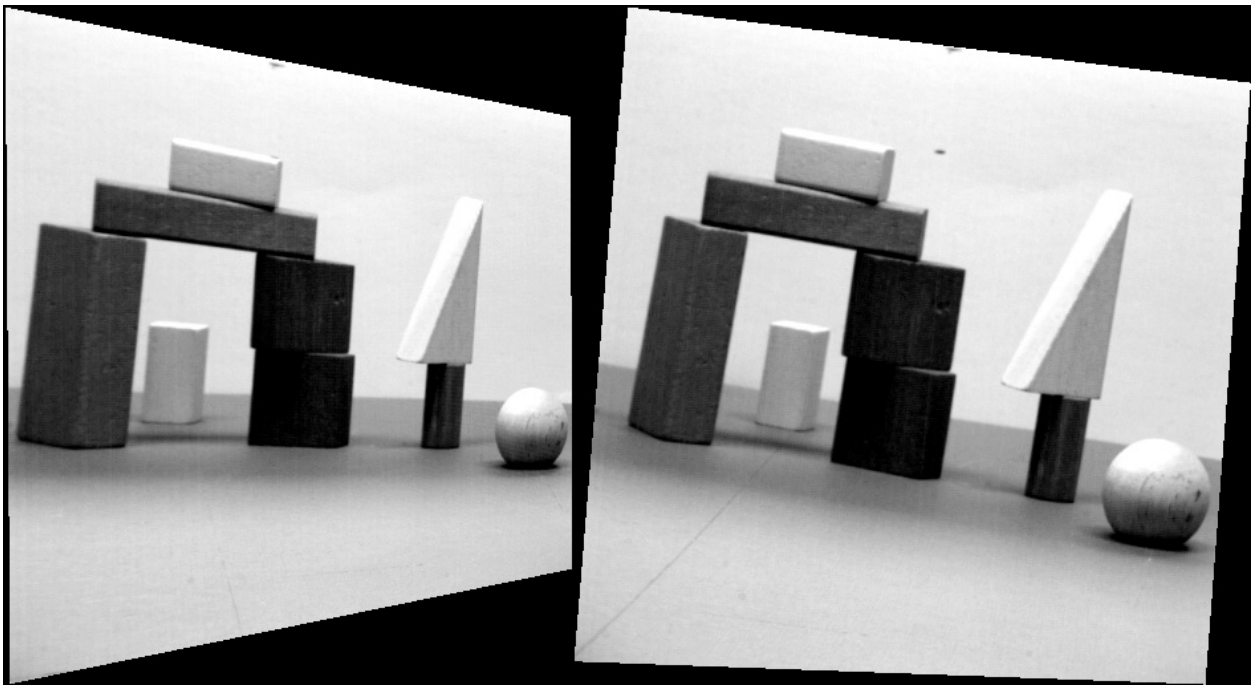

Figure 10: Poor correspondence detection leads to poor results. In this case, the alignment between the images is bad and the rectified images do not look aligned.

3 Estimating the fundamental matrix

We estimate the fundamental matrix as a two step process:

- Linearly estimate of F.
- Use a nonlinear method to refine the initial, linear estimate of F.

Before diving into the estimation of F, however, there is some preprocessing required to improve the performance oft he algorithm.

3.1 Normalization

The first step in the normalized 8 point method is, as the name suggests, normalization. The least squares minimization is performed in homogeneous coordinates which also takes into account the scaling factor of the points. The magnitude of the scale factor (which we normally keep as 1.0) is significantly different from the magnitudes of the x and y coefficients. This will introduce unnecessary biases while trying to optimize the fundamental matrix. The solution is as follows:

- Move the origin to the centroid of the correspondences:

$$\vec{O} \rightarrow \vec{C} \implies \vec{x} \rightarrow \hat{\vec{x}}$$

  This can achieved easily in homogeneous coordinates using a purely translational rigid body transform, T.

- Finally scale the points so that their Euclidean distance in $\mathbb{R}^3$ is $\sqrt{2}$.

We then estimate the fundamental matrix, $\hat{F}$, using the schemes outlined below. Once we have the fundamental, we can de-normalize as follows:

$$F = T_2^T \hat{F} T_1$$

3.2 Forming the linear, homogeneous least squares problem

For the same point in 3D space, $\vec{X}$, observed by two cameras (thereby producing two images, x_1 and x_2), the fundamental matrix, F, provides the following relationship:

$$x_2^T F x_1 = 0$$

The fundamental matrix, F, is a 3×3 matrix which can be written in the form:

$$\begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix}$$

Hence, the initial equation can be rewritten to give us:

$$x_1 x_2 f_{11} + y_1 x_2 f_{12} + x_2 f_{13} + x_1 y_2 f_{21} + y_1 y_2 f_{22} + y_2 f_{23} + x_1 f_{31} + y_1 f_{32} + f_{11} = 0$$

This is an equation which is of the form $\mathbf{A}\vec{f} = vec0$. The size of $\mathbf{A}$ is $n \times 9$, while the sizes of $\vec{f}$ and $\vec{0}$ are 9×1, and $n \times 1$ respectively. Each point correspondence provides us with one constraint. Hence, for 8 unique elements of F (since we only need to compute it upto a scale), we require atleast 8 point correspondences.

We seek a non-trivial solution for $\vec{f}$, that minimizes $||\mathbf{A}\vec{f}||$ subject to $||\vec{f}|| = 1$ (this is to ensure we do get the trivial solution, $\vec{f} = \vec{0}$). The solution to the problem is given by the eigen vector of $\mathbf{A}^T\mathbf{A}$ which corresponds to the smallest eigen value. This can be computed by performing a signular value decomposition on $\mathbf{A}^T\mathbf{A}$ (the solution to $\vec{f}$ is the right most column of $\mathbf{V}$).

3.3 Enforcing a rank constraint on F

We require an F that is as close to the linearly estimated result as possible, while additionally imposing that its rank be 2. A rank 3 matrix could not satisfy $x_2^T F x_1 = 0$, and a rank 1 matrix does not have enough constraints to map a point to a line. We enforce the rank constraint as follows:

$$U\ S\ V^* = SVD(F)$$

$$S_0 = S[n][n] = 0$$

$$F_2 = U\ S_0\ V^*$$

3.4 Refining the fundamental matrix

In the linear estimation section, we minimized an algebraic distance to get in the ball park of what the fundamental matrix should be. However that will not provide accurate enough results, so we frame a geometric distance problem and use the Levenberg-Marquadt algorithm to get a good enough estimate.

3.4.1 A geometric cost function to minimize

We want to minimize the error in cross projecting the points from one image onto the other:

$$\arg\min ||\Sigma\ (x_1 - \hat{x_2}) + (x_2 - \hat{x_1})||^2$$

In order to reproject a point from one image onto another, we use our estimate of the fundamental matrix, F, we triangulate that point back into 3D space and then project it. This requires an estimate of the camera projection matrix as well as a way to triangulate the points back to 3D space.

3.4.2 Estimating the projection matrices from F

We use canonical configurations of the projection matrices. This gives us:

$$P_1 = [I_{3\times 3}|\vec{0}]$$

$$P_2 = [[e_2]_x F|\vec{e_2}]$$

$$[e_2]_x : \text{ Matrix cross product equivalent}$$

The epipoles of the principal image and the secondary image are the right and left null vectors of F respectively.

3.4.3 Reprojecting a point back to 3D

We can represent a projection matrix, P, as follows:

$$P = \begin{pmatrix} \vec{p_1}^T \\ \vec{p_2}^T \\ \vec{p_3}^T \end{pmatrix}$$

To triangulate an image point, x, back to its 3D coordinates, X, we build a homogeneous linear least squares problem as follows for the point using the canonical projection matrices:

$$A = \begin{pmatrix} \vec{x_0}\vec{p_3} - \vec{p_1} \\ \vec{x_1}\vec{p_3} - \vec{p_2} \end{pmatrix}$$

4 Image rectification

4.0.4 Rectifying the secondary image

We rectify an image by sending the epipole to infinity along the x-axis. This is done as follows:

- We rotate the image so that it is parallel to the epipolar line.

$$\theta = \tan^{-} 1\big(- \frac{\text{Img}_{ht}/2 - e_2[y]}{\text{Img}_{wd}/2 - e_2[x]}\big)$$

- We then translate the origin to the image center.
- The image center is now of the form $[f01]^T$, we can send it to infinity by multiplying by this matrix:

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1/f & 0 & 1 \end{pmatrix}$$

- Finally we shift the image center back to its original position with the opposite of translation used in the first step.

The above steps gives us a homography, H_2 which maps the original secondary image to the rectified image.

4.1 Rectifying the primary image

I used two methods for this section:

4.1.1 Textbook method

We attempt to find a H_1 that minimizes:

$$\Sigma_i ||H_1 x_1 - H_2 x_2||^2$$

This is found as follows:

$$M = P_2 P_1^{+}$$
$$H_0 = H_2 M$$

We then find a, b, and c, which minimizes:

$$\Sigma_i a\hat{x_i} + b\hat{y_i} + c\hat{x_i'}$$

We then build H_A as follows:

$$H_A = \begin{pmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$H_1 = H_A H_0$$

This method yielded unusable homographies (the images would not form, however the resultant errors were ¡ 3 pixels).

4.2 Using the H2 method

I used the H2 rectification method from image 1 as well and generated a corresponding homography. This method additionally requires an additional optimal translation that aligns the two images.

5 Reprojection to 3D

- We apply H_1 and H_2 to the their respective image.
- We find points of interest using the Canny algorithm.

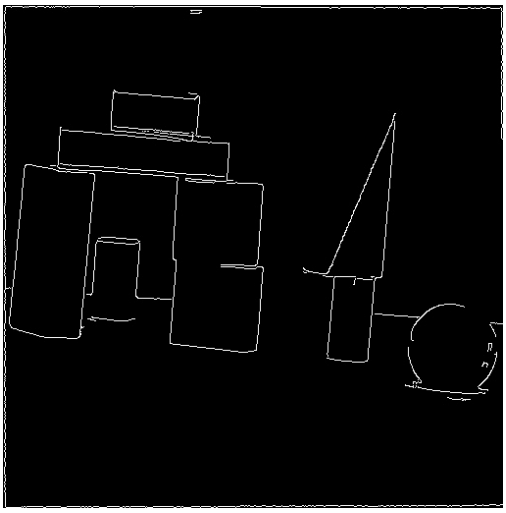

Figure 11: Canny's optimal edge detector applied to rectified image 1.

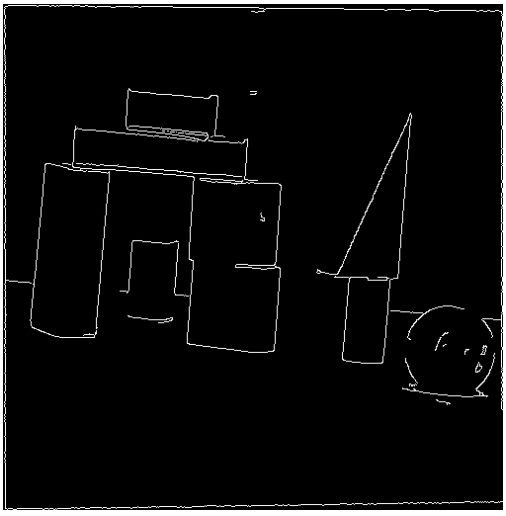

Figure 12: Canny's optimal edge detector applied to rectified image 2.

- For every foreground pixel in image 1, we look 9 rows above and below in image 2 for a correspondence using the Normalized Cross Correlation (NCC) metric.

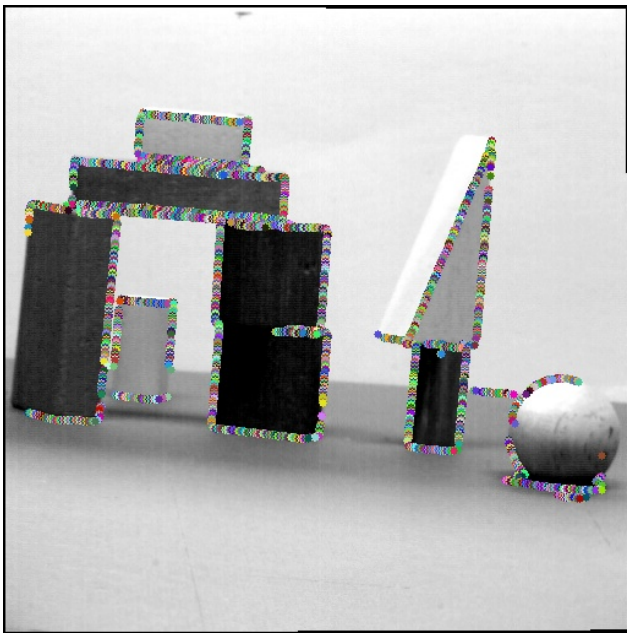

Figure 13: Points traversed in image 1.

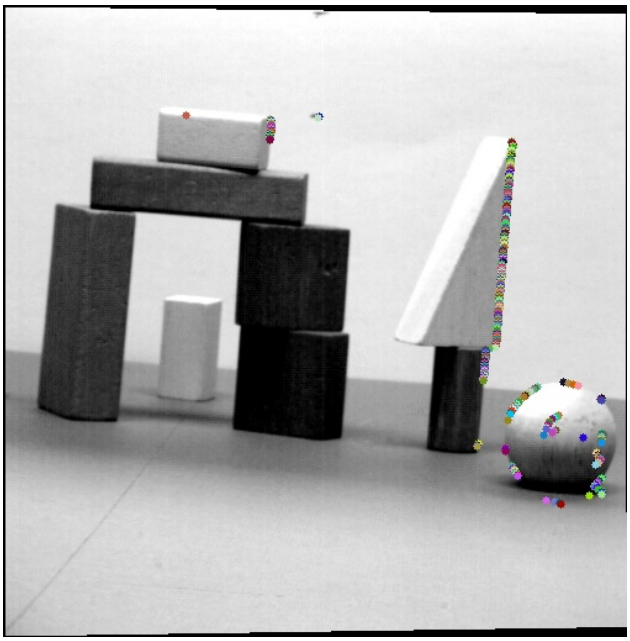

Figure 14: Correspondences found in image 2.

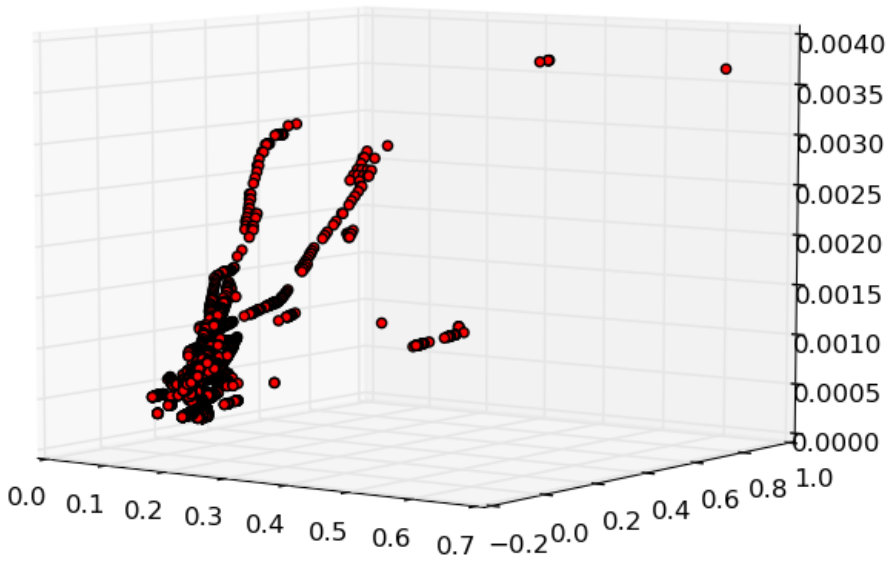

Figure 15: Output view 1.

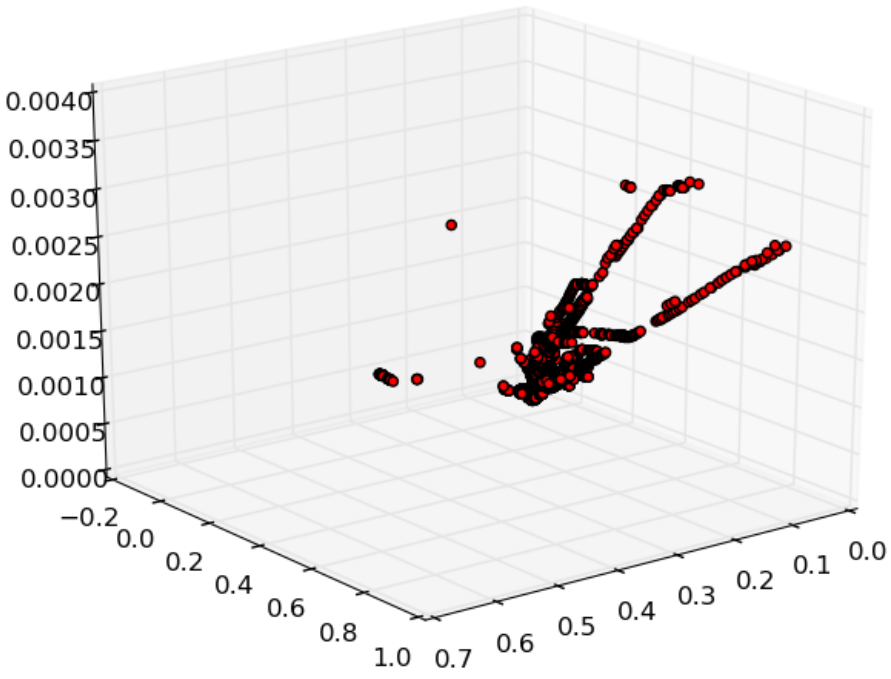

Figure 16: Output view 2.

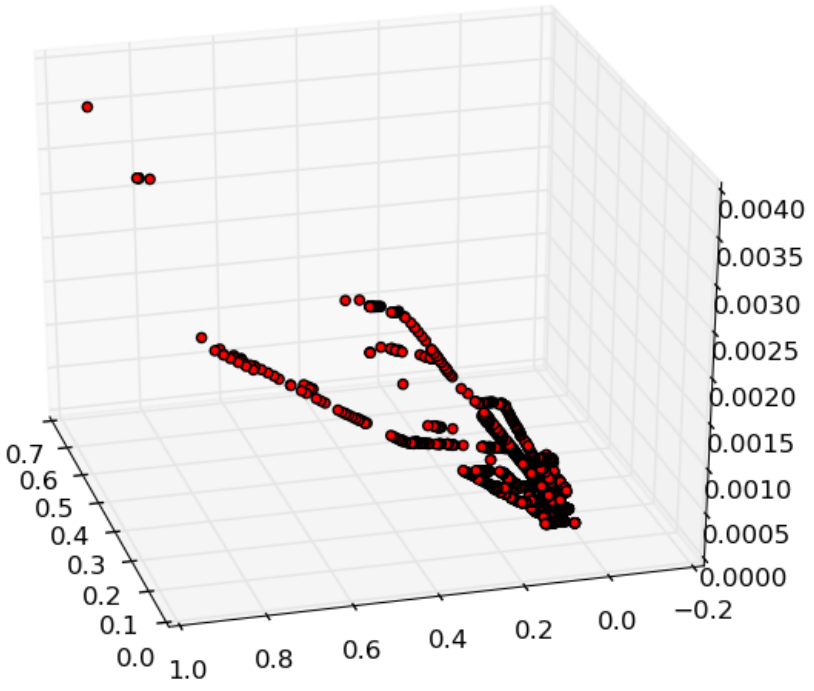

Figure 17: Output view 3.

6 Appendix

6.1 Keypoint detection using SIFT

The Scale Invariant Feature Transform (SIFT) attempts to find and characterize scale-space extrema in order to achieve invariance to scale. These scale-space points of interest are generated at the extremes of:

$$\nabla^2 ff(x, y, \sigma) = \frac{\delta^2 \ ff(x, y, \sigma)}{\delta x^2} + \frac{\delta^2 \ ff(x, y, \sigma)}{\delta y^2}$$

The scale space is generated through a difference of Gaussians pyramid as shown in figure 18. And extrema are located in a 3×3 neighborhood at the current scale, one scale above and one scale below as shown in figure 19.

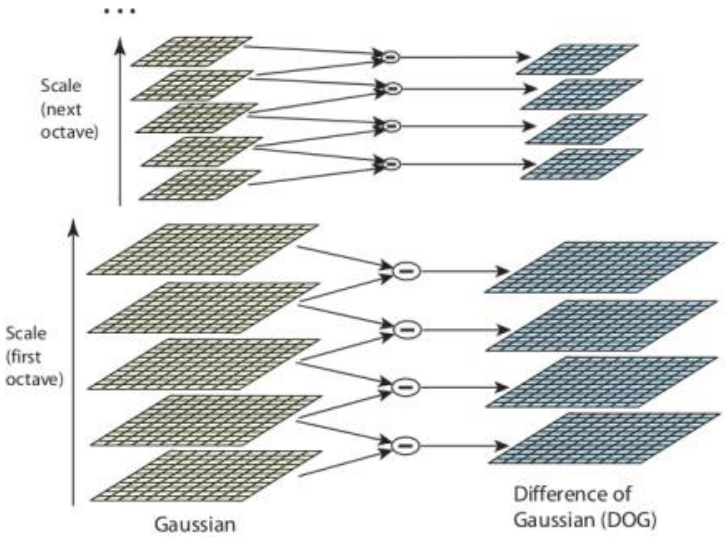

Figure 18: SIFT uses a difference of Gaussians (DoG) pyramid in order to approximate the Laplacian function. Source: http://goo.gl/q77FW6

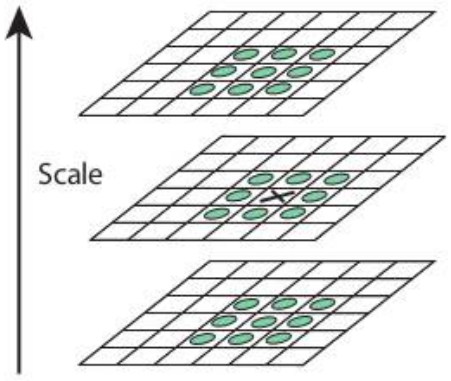

Figure 19: Neighbor hood for determining scale-space extrema. Source: http://goo.gl/q77FW6

The next step is to localize points of interest with sub-pixel accuracy. This done by linearizing the point about pixel location of the extremum via a Taylor series expansion:

$$P(\vec{x}) = P(\vec{x_0}) + J^T(\vec{x_0})(\vec{x}) + \frac{1}{2}\vec{x}^T H(\vec{x_0})\vec{x}$$

Where:

$$\vec{x} = \vec{x_0} + \delta\vec{x}: \text{ is the point of interest.}$$

$$J: \text{ is the Jacobian of } P \text{ at } \vec{x_0}$$

$$H: \text{ is the Hessian of } P \text{ at } \vec{x_0}$$

Hence, the point of interest is given by:

$$\vec{x} = -H^{-1}(\vec{x_0})J(\vec{x_0})$$

At this point we would like *strong* points to characterize our image, so we remove them by thresholding and checking their relative Eigen value magnitude to eliminate edges. The penultimate step is make our points of interest rotation invariant. This is done by finding a dominant vector for each scale-space extremum.

The last step is to characterize this point of interest along its dominant orientation. This is done through a 128 dimensional vector at the scale of the extremum.The descriptor is created by generating a 16×16 neighborhood around the POI, with each region having 4×4 cells, and each cell having 4×4 points. A histogram of gradients[1] is slowly built up. Finally, the 128 element SIFT descriptor is normalized to one in order to make the descriptor independent to the effects of illumination.

6.1.1 Euclidean distance metric

The SIFT descriptor gives us a 128 element characterization of the space around a key point (taken along the dominant vector of the region). As such, we do not require anything more complicated like an SSD or NCC metric to compare two descriptors.

$$\mathbf{ED} = \Sigma|d_1(i) - d_2(i)|^2$$

Thus we can conclude that the more similar two patches are, the lower the SSD will be.

Eliminating ambiguous correspondences If the following condition is met, we ignore the correspondence:

$$\frac{\text{ED}_{\text{best correspondence score}}}{\text{ED}_{\text{secon best correspondence score}}} > \tau_{\text{ED Ratio}}$$

6.2 Establishing inter-image correspondences

Given a set of two or more images for which we have detected points of interest, we would like to find the best correspondences between the images. This has been implemented via the following paradigm:

1. In each image, extract a window around each point of interest. This is referred to as a *patch*.
2. Use a brute-force method to compare each patch in the first image with each patch in the second image. Patches from two image are compared using a similarity metric such as the normalized cross correlation (NCC) or the sum of squared differences (SSD) of the two patches.
3. Correspondences are established between pairs of patches that are most similar.

[1] Each vector is rounded to 1 of 8 directions, thus 8 bits can be used to characterize 8 constant directions.

6.2.1 Excluding ambiguous corners

Not all corners are unique in the context of the whole image- often a region that stands out in a window might be far from unique in the context of scene. Thus, we remove ambiguous corners by comparing the best correspondence score with the second best correspondence score. Similar magnitudes indicate ambiguity, hence it is better to ignore the corner in question. It is implemented slightly differently in NCC and SSD, however the idea behind it is the same in both cases- we observe the ratio of the best and second best response in order to ignore ambiguous points.

6.3 Sum of squared differences (SSD)

The sum of squared differences is a pseudo-euclidean distance measure in pixel-intensity space. As it's name suggests, it is the summation of the square of the differences between two pixel patches. Mathematically, it can be written as:

$$\textbf{SSD} = \Sigma\Sigma|f_1(i,j) - f_2(i,j)|^2$$

Thus we can conclude that the more similar two patches are, the lower the SSD will be.

Pros

- It is easy to implement.
- Runs quickly.

Cons

- Not robust to orientation changes.
- Sensitive to illumination differences as it not normalized w.r.t. the local overall intensity.
- It is an unbounded measure, so we can't tell much about a window unless we look at relative values.

6.3.1 Eliminating ambiguous correspondences

If the following condition is met, we ignore the correspondence:

$$\frac{\text{SSD}_{\text{best correspondence score}}}{\text{SSD}_{\text{secon best correspondence score}}} > \tau_{\text{SSD Ratio}}$$

6.4 Normalized Cross Correlation (NCC)

The NCC provides a normalized metric (between -1 and 1) of how similar two patches are, closer the value is to 1, the more similar the two patches are.

$$\textbf{NCC} = \frac{\Sigma\Sigma(f_1(i,j) - \mu_1)(f_2(i,j) - \mu_2)}{\sqrt{\Sigma\Sigma(f_1(i,j) - \mu_1)^2 \Sigma\Sigma(f_2(i,j) - \mu_2)^2}}$$

Pros

- More robust to lighting differences
- More robust than SSD to rotational and affine distortions.
- Bounded output (between -1 and 1), this allows us to get a better understanding of the characteristics of a point without looking at all the other pixels in an image.

Cons

- Much slower than SSD.

6.4.1 Eliminating ambiguous correspondences

If the following condition is met, we ignore the correspondence:

$$\frac{\text{NCC}_{\text{second best correspondence score}}}{\text{NCC}_{\text{best correspondence score}}} > \tau_{\text{NCC Ratio}}$$

7 Source code

7.1 hw9_lib.py

```
"""
Homework 9 custom library.

More information is provided in the doc-string of each method.

Course: ECE661: Computer Vision, Fall 2014
Homework: 9
Name: Shiva Ghose
Email: ghose0@purdue.edu
Date: 11.25.2014
"""

import sys
import logging

import cv2
import numpy as np
from scipy.optimize import leastsq

def normalize_vector_array(mat_x):
    """
    Normalizes the first two columns of a nx3 vector by the third column.

    Given an nx3 vector-array mat_x = (vec_x_1, vec_x_2, vec_x_3), we
    normalize the vector to get (vec_x_1/vec_x_3, vec_x_2/vec_x_3, 1).
    Note that while the elements of x_1 and x_2 can take on most floating
    point values, the elements of x_3 cannot be zero.

    Args:
        mat_x:  nx3 NumPy array that represents a collection of 2D positional
                vectors in homogeneous coordinates.

    Returns:
        mat_nrm: nx3 NumPy array that represents normalized mat_x.


    Raises:
        ValueError: If vex_x.shape[1] != 3.
    """
```

```
    if mat_x.shape[1] != 3:
        raise ValueError("""Input vector shape is wrong, vec_x.shape = {0},
                            which is != (n,3)""".format(mat_x.shape))

    vec_nrm = (mat_x.T/mat_x[:,2]).T

    return vec_nrm

def get_image_side_by_side(image1, image2):
    """
    Places 2 images side by side and returns the new image as well as offsets
    of each images origin (if displaced). For the time being, the system
    requires both images to be of the same size.

    ----------------------------------------------------------------------------
    TODO
    * Modify to allow for the use of images that are not of the same size.
    ----------------------------------------------------------------------------

    Args:
        image1: An nxm NumPy array of the first image.

        image2: An nxm NumPy array of the second image.


    Returns:
        image_side_by_side: An nx2m numpy array that represents the 2 images
                            stacked side by side.

        vec_image1_offset: An offset vector for image1 made up of:
                               [0] - first index's offset
                               [1] - second index's offset
                               [2] - 0.0 (so that you can perform homogeneous
                                     vector addition)

        vec_image2_offset: An offset vector for image2 made up of:
                               [0] - first index's offset
                               [1] - second index's offset
                               [2] - 0.0 (so that you can perform homogeneous
                                     vector addition)

    Raises:
        ValueError: If the images are of different sizes.
    """
    image_side_by_side = np.hstack((image1, image2))
    vec_image1_offset = np.array([0.,0.,0.]).reshape(3,1)
    vec_image2_offset = np.array([0.,image1.shape[1],0.]).reshape(3,1)

    return image_side_by_side, vec_image1_offset, vec_image2_offset

def get_corners_subpix(img_gray, points):
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
```

```
    corners = np.zeros((len(points),2),dtype=np.float32)
    corners[:,0] = points[:,1].copy()
    corners[:,1] = points[:,0].copy()
    cv2.cornerSubPix(img_gray,corners,(5,5),(-1,-1),criteria)
    points_subpix_img = np.zeros((len(points),3),dtype=np.float32)
    points_subpix_img[:,0] = corners[:,1]
    points_subpix_img[:,1] = corners[:,0]
    points_subpix_img[:,2] = 1
    return points_subpix_img

def linear_least_squares_hom(mat_A):
    """

    Computes the linear least squares solution to the problem:
                Ax = 0
    Returns x, the vector of unknowns.
    """
    mat_AtA = np.dot(mat_A.T,mat_A)
    eigenValues,eigenVectors = np.linalg.eig(mat_AtA)
    #Sort the Eigen Values to find the smallest one [credits:http://stackoverflow.com/a/8093043]
    idx = eigenValues.argsort()
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:,idx]
    #Return the smallest Eigen Value's Eigen Vector
    return eigenVectors[:,0]

def compute_mat_F_linear(points_1, points_2):
    mat_A = np.zeros((len(points_1),9))
    vec_x = points_1[:,1]
    vec_x_ = points_2[:,1]
    vec_y = points_1[:,0]
    vec_y_ = points_2[:,0]
    mat_A[:,0] = np.multiply(vec_x_, vec_x)
    mat_A[:,1] = np.multiply(vec_x_, vec_y)
    mat_A[:,2] = vec_x_
    mat_A[:,3] = np.multiply(vec_y_, vec_x)
    mat_A[:,4] = np.multiply(vec_y_, vec_y)
    mat_A[:,5] = vec_y_
    mat_A[:,6] = vec_x
    mat_A[:,7] = vec_y
    mat_A[:,8] = 1.0
    vec_f = linear_least_squares_hom(mat_A)
    mat_F_unconditioned = vec_f.reshape(3,3)
    #Condition mat_F
    mat_F = make_mat_F_rank_2(mat_F_unconditioned)
    return mat_F

def make_mat_F_rank_2(mat_F_unconditioned):
    U, s, V = np.linalg.svd(mat_F_unconditioned)
    #Set the last Eigen value to zero
    s[-1] = 0
    #Recompose mat_F:
```

```
#     S = np.zeros((len(s), len(s)))
    S = np.diag(s)
    return np.dot(U, np.dot(S, V))


def get_cross_product_matrix(vec_x):
    mat_X = np.zeros((len(vec_x),len(vec_x)))
    mat_X[0,1] = -vec_x[2]
    mat_X[0,2] = vec_x[1]
    mat_X[1,2] = -vec_x[0]
    mat_X[1,0] = vec_x[2]
    mat_X[2,0] = -vec_x[1]
    mat_X[2,1] = vec_x[0]
    return mat_X

def get_null_space(mat_A, rtol=1e-5):
    """
    Based on code found at: http://stackoverflow.com/a/19821304
    """
    _mat_U, vec_s, mat_V_T = np.linalg.svd(mat_A)
    rank = (vec_s > rtol*vec_s[0]).sum()
    return mat_V_T[rank:].T.copy(), rank

def triangulate_to_3D(vec_x_1, vec_x_2, mat_P_1, mat_P_2):
    list_vec_X = []
    for i in range(len(vec_x_1)):
        mat_A1 = _get_rows_of_triangulation_A(mat_P_1, vec_x_1[i])
        mat_A2 = _get_rows_of_triangulation_A(mat_P_2, vec_x_2[i])
        mat_A = np.vstack((mat_A1,mat_A2))
        vec_X = linear_least_squares_hom(mat_A)
        #Normalize vec_X
        vec_X = vec_X/vec_X[3]
        list_vec_X.append(vec_X)
    return np.array(list_vec_X)

def _get_rows_of_triangulation_A(mat_P, vec_x):
    vec_p_1 = mat_P[0,:]
    vec_p_2 = mat_P[1,:]
    vec_p_3 = mat_P[2,:]
    vec_a1 = ((vec_x[0]*vec_p_3) - vec_p_1).reshape(1,4)
    vec_a2 = ((vec_x[1]*vec_p_3) - vec_p_2).reshape(1,4)
    mat_A = np.vstack((vec_a1,vec_a2))
    return mat_A

def get_projection_matrices(mat_F):
    #Projection matrix 1:
    mat_P_1 = np.zeros((3,4))
    mat_P_1[:,0:3] = np.eye(3)
    #Projection matrix 2:
    vec_e_2,_rank = get_null_space(mat_F.T)
    if _rank<2:
        print "ERROR: Rank of mat_F < 2!"
```

```
    mat_E_2 = get_cross_product_matrix(vec_e_2)
    mat_P_2 = np.dot(mat_E_2, mat_F)
    mat_P_2 = np.hstack((mat_P_2,vec_e_2))
    #Return values:
    return mat_P_1, mat_P_2

def unpack_params(params):
    params= np.append(params,[1.0])
    mat_F_unconditioned = params.reshape(3,3)
    mat_F = make_mat_F_rank_2(mat_F_unconditioned)
    mat_P1, mat_P2  = get_projection_matrices(mat_F)
    return mat_P1, mat_P2, mat_F

def pack_params(mat_F):
    mat_F = mat_F/mat_F[2,2]
    vec_f = mat_F.flatten()
    params = vec_f[:-1]
    return params.reshape(8,1)

def residuals(params, points_1, points_2):
    mat_P_1, mat_P_2, _mat_F = unpack_params(params)
    #project the points to 3D:
    mat_points_3D = triangulate_to_3D(points_1, points_2, mat_P_1, mat_P_2)
    #project the points to image 1's FOV:
    mat_points_3D_in_1 = (np.dot(mat_P_1, mat_points_3D.T)).T
    #Normalize
    mat_points_3D_in_1 = normalize_vector_array(mat_points_3D_in_1)
    mat_points_3D_in_2 = (np.dot(mat_P_2, mat_points_3D.T)).T
    #Normalize
    mat_points_3D_in_2 = normalize_vector_array(mat_points_3D_in_2)
    error_1 = reprojection_error(points_1, mat_points_3D_in_1)
    error_1 = error_1.reshape(len(error_1),1)
    error_2 = reprojection_error(points_2, mat_points_3D_in_2)
    error_2 = error_2.reshape(len(error_2),1)
    total_error = np.vstack((error_1,error_2))
    return total_error.flatten()

def reprojection_error(observed_points, reprojected_points):
    error = observed_points - reprojected_points
    error = error[:,0:2]
    return error.flatten()

def get_normalization_matrix(points):
    #Find the centroid of the points:
    y_mean = points[:,0].mean()
    x_mean = points[:,1].mean()
    #Make the centroid the origin of the points:
    shifted_points = np.zeros_like(points)
    shifted_points[:,0] = points[:,0] - y_mean
    shifted_points[:,1] = points[:,1] - x_mean
    #Leave shifted_points[:,2] as zero, so we get the distance from the origin
    mean_distance = np.linalg.norm(shifted_points,axis=1).mean()
```

```
    #Scale the mean distance to sqrt(2)
    scale = np.sqrt(2)/mean_distance
    mat_T = np.array([[scale,   0.0,      -scale*y_mean],
                      [0.0,       scale,  -scale*y_mean],
                      [0.0,       0.0,      1.0]])
    return mat_T

def compute_mat_F_nonlinear(points_subpix_img1, points_subpix_img2, mat_F_linear):
    points_nrm_1 = points_subpix_img1
    points_nrm_2 = points_subpix_img2
    #Nonlinear estimate of mat_F:
    # mat_P_1_linear, mat_P_2_linear = hw9_lib.get_projection_matrices(mat_F_linear)
    params0 = pack_params(mat_F_linear)
    params_nonlin, _int_flag = leastsq(residuals, params0, args=(points_nrm_1, points_nrm_2))
    _mat_P_1_temp, _mat_P_2_temp, mat_F_nonlinear_temp = unpack_params(params_nonlin)
    return mat_F_nonlinear_temp

def compute_mat_F(points_subpix_img1, points_subpix_img2,
                  use_nonlinear=True, use_normalization=True):
    if use_normalization:
        #Normalize the points:
        mat_T_1 = get_normalization_matrix(points_subpix_img1)
        mat_T_2 = get_normalization_matrix(points_subpix_img2)
        points_nrm_1 = (np.dot(mat_T_1, points_subpix_img1.T)).T
        points_nrm_2 = (np.dot(mat_T_2, points_subpix_img2.T)).T
    else:
        print "WARNING: Not using the normalization step!"
        points_nrm_1 = points_subpix_img1
        points_nrm_2 = points_subpix_img2
    #Linearly estimate the Fundamental matrix:
    mat_F_linear = compute_mat_F_linear(points_nrm_1, points_nrm_2)
    mat_F_linear = mat_F_linear/mat_F_linear[2,2]
    print "Linear estimation of the fundamental matrix:"
    if not use_nonlinear:
        mat_F_prime = mat_F_linear
    else:
        print mat_F_linear
        #Nonlinear estimate of mat_F:
        mat_F_prime = compute_mat_F_nonlinear(points_nrm_1, points_nrm_2, mat_F_linear)
    if use_normalization:
        #Compensate for the normalization step:
        mat_F = np.dot(mat_T_2.T, np.dot(mat_F_prime,mat_T_1))
        print "Non-linear estimation of the fundamental matrix:"
    else:
        mat_F = mat_F_prime
    #Normalize
    mat_F = mat_F/mat_F[2,2]
    mat_P_1, mat_P_2 = get_projection_matrices(mat_F)
    print mat_F
    # Print the maximum reprojection error:
    params = pack_params(mat_F)
    reprojection_residuals = residuals(params, points_subpix_img1, points_subpix_img2)
```

```
    print "Maximum reprojection error:", reprojection_residuals.max()
    return mat_F, mat_P_1, mat_P_2

def get_2DRBT_rotation(theta):
    """
    Provides a counter-clockwise positive 3x3 rotation matrix for 2D points.

    Theta is assumed to be in radians, and -Pi<Theta<=Pi
    """
    mat_R = np.zeros((3,3))
    mat_R[0,0] = np.cos(theta)
    mat_R[0,1] = np.sin(theta)
    mat_R[1,0] = -np.sin(theta)
    mat_R[1,1] = np.cos(theta)
    mat_R[2,2] = 1.0
    return mat_R

def get_2DRBT_translation(ty, tx):
    mat_T = np.eye(3)
    mat_T[0,2] = ty
    mat_T[1,2] = tx
    return mat_T

def get_matrix_G(f):
    mat_G = np.eye(3)
    mat_G[2,1] = -1.0/f
    return mat_G

def compute_H_A(x_1_hat, x_2_hat):
    mat_A = np.ones((len(x_1_hat), 3))
    i = -1
    for point in x_1_hat:
        i += 1
        mat_A[i,0] = point[0]
        mat_A[i,1] = point[1]
    vec_b = x_2_hat[:,1].copy()
    _ret = np.linalg.lstsq(mat_A, vec_b)
    vec_h_a = _ret[0]
    mat_H_a = np.eye(3)
    mat_H_a[1,0] = vec_h_a[0]
    mat_H_a[1,1] = vec_h_a[1]
    mat_H_a[1,2] = vec_h_a[2]
    return mat_H_a


def normalize_vector(vec_x):
    """
    Normalizes the first two elements of a 3x1 vector by the third element.

    Given a 3x1 vector vec_x = (x_1, x_2, x_3)^T, we normalize the vector to
    get (x_1/x_3, x_2/x_3, 1)^T. Not that while x_1 and x_2 can take on most
    floating point values, x_3 cannot be zero.
```

```
    Args:
        vec_x:  3x1 NumPy array that represents a 2D positional vector in
                homogeneous coordinates.

    Returns:
        vec_nrm: 3x1 NumPy array that represents normalized vec_x.


    Raises:
        ValueError: If vex_x.shape != (3,1).

        ValueError: If x_3 = 0.
    """
    if vec_x.shape[0] != 3:
        raise ValueError("""Input vector shape is wrong, vec_x.shape = {0},
                            which is != (3,1)""".format(vec_x.shape))
    if vec_x[2] == 0:
        raise ValueError("""Input vector value is illegal, Require: x_3 == 0.
                            Found: Vec_x = {0}""".format(vec_x))

    vec_nrm = np.zeros((3,1))
    for i in range(vec_x.shape[0]):
        vec_nrm[i] = vec_x[i]/vec_x[2]

    return vec_nrm


def project_image_using_homogrpahy(image_src, mat_H_src2dest):
    """
    Returns a new image_1 by applying the specified homogrpahy to it.

    Note: This method is designed to works best for scaling up operations.

    Args:
        image_src: an nxmx3 NumPy array representing an image_1.

        mat_H_src2dst: a 3x3 NumPy array that represents a homography from
                       the source frame to the destination frame.

    Returns:
        image_dest: an axbx3 NumPy array that represents the transformed
                    image_1.

        vec_offsets: offsets of the src image_1's origins in the destination
                     image_1's coordinate system.

    Raises:
        ValueError: if mat_H_src2dst.shape != 3x3

        LinAlgError: if mat_H_src2dst is not square or if the inversion fails.
    """
```

```
    if mat_H_src2dest.shape != (3,3):
        logging.error("""mat_H_src2dst.shape != (3,3). mat_H_src2dst.shape
                         = {0}""".format(mat_H_src2dest.shape))
        raise ValueError("""mat_H_src2dst.shape = {0}""".format(mat_H_src2dest.shape))
    #Invert mat_H_src2dest to get the homography from the destination to the
    #source
    mat_H_dest2src = np.linalg.inv(mat_H_src2dest)
    #Find the dimensions of the final image_1:
    max_x1 = max_x2 = sys.float_info.min
    min_x1 = min_x2 = sys.float_info.max
    print "image_src.shape = ", image_src.shape
    for y_coord in [0, image_src.shape[0]]:
        for x_coord in [0, image_src.shape[1]]:
            point_src = np.array([y_coord, x_coord, 1.0]).reshape(3,1)
#             print "point_src = ",point_src.T
            point_dest = mat_H_src2dest.dot(point_src)
#             print "point_dest = ",point_dest.T
            point_dst_nrm = normalize_vector(point_dest)
#             print "point_dest_nrm = ",point_dst_nrm.T
            if point_dst_nrm[0] > max_x1:
                max_x1 = point_dst_nrm[0]
            if point_dst_nrm[0] < min_x1:
                min_x1 = point_dst_nrm[0]
            if point_dst_nrm[1] > max_x2:
                max_x2 = point_dst_nrm[1]
            if point_dst_nrm[1] < min_x2:
                min_x2 = point_dst_nrm[1]
    #Convert back to pixel coordinates from floats.
    scale = float(image_src.shape[0])/float(max_x1-min_x1)
    img_final_shape_1 = int(float(max_x2-min_x2)*scale)

    vec_offsets = np.array([max_x1, min_x1, max_x2, min_x2])
    vec_off = np.array([float(min_x1), float(min_x2), 0.0]).reshape((3,1))
    print "vec_offsets = ", vec_offsets.T

    #Create a destination image_1 based on the values of the maximas and minimas.
    image_dest = np.zeros((image_src.shape[0], img_final_shape_1, 3), np.uint8)
    print "image_dest.shape = ", image_dest.shape

    total_iterations = image_dest.shape[0]*image_dest.shape[1]
    progress_image_scale = 300.0/(max_x1-min_x1)

    #Go through the final image_1 and compute each pixel's corresponding
    #sub-pixel equivalent in the source image_1. Then use an interpolation method
    #to estimate the value of the final-image_1 pixel.
    for y_coord in range(0, image_dest.shape[0]):
        for x_coord in range(0, image_dest.shape[1]):
            #Remember to use the offset!
            point_dest = np.array([(y_coord), (x_coord), scale]).reshape((3,1))
            point_dest = normalize_vector(point_dest)
            point_dest = np.add(point_dest,vec_off)
            #Convert the point from final-image_1 coordinates back to
```

```
            #source-image_1 coordinates.
            point_src = mat_H_dest2src.dot(point_dest)
            point_src_nrm = normalize_vector(point_src)
            #Ignore the edge pixels as they cannot be interpolated properly
            if int(point_src_nrm[0]) > (image_src.shape[0]-2) or int(point_src_nrm[0]) < 1:
                continue
            if int(point_src_nrm[1]) > (image_src.shape[1]-2) or int(point_src_nrm[1]) < 1:
                continue
            try:
                #Use OpenCV's bilinear interpolation tool to estimate the sub-pixel value
                image_dest[y_coord][x_coord] = cv2.getRectSubPix(image_src, (1,1), (point_src_nrm[1], p
            except IndexError, e:
                logging.error("Error: %s"%(e))
                pass

        #Show progress:
        #From http://stackoverflow.com/a/18767569 :
        if y_coord%20 == 0:
            progress = y_coord*100.0/total_iterations
#             image_final_small = cv2.resize(image_dest, (0,0), fx=progress_image_scale, fy=progress_ima
            image_final_small = image_dest
            cv2.imshow("Progress".format(progress), image_final_small)
            cv2.waitKey(10)
    cv2.destroyWindow("Progress")
#     image_final_small = cv2.resize(image_dest, (0,0), fx=progress_image_scale, fy=progress_image_scal
    image_final_small = image_dest
    cv2.imshow("Done!".format(progress), image_final_small)
    return image_dest, vec_offsets

def get_euclidean_distance(vec_1, vec_2):
    """
    Returns the Euclidean distance between two vectors

    Args:
        vec_1:  nxm NumPy array.

        vec_2:  nxm NumPy array.

    Returns:
        euclidean_dist:    The Euclidean distance between the 2 inputs.

    Requires:
        vec_1 and vec_2 to be the same shape.
    """
    return np.linalg.norm(vec_1-vec_2)

def get_correspondences_sift(kp1, des1, kp2, des2, threshold_ratio=0.70):
    """
    Finds the best correspondences b/w the SIFT descriptors using the
    Euclidean distance as a measure.
```

```
    Args:
        kp1:    Keypoint list of the first image generated by OpenCV's SIFT

        des1:   Descriptor list of the first image generated by OpenCV's SIFT
                class.

        kp2:    Keypoint list of the second image generated by OpenCV's SIFT

        des2:   Descriptor list of the second image generated by OpenCV's SIFT
                class.

    Returns:
        sift_correspondences: A list made up of the following tupels:
                                    (des1, des2, euclidean_score)

    Raises:
        ValueError: If the window size is not odd
    """
    sift_correspondences = []
    sift_scores = []
    des1_index = -1
    for patch1 in des1:
        des1_index += 1
        sift_best_score = sys.float_info.max #lower is better
        sift_second_best_score = sys.float_info.max
        sift_best_index = -2
        des2_index = -1
        for patch2 in des2:
            des2_index += 1
            sift_score = get_euclidean_distance(patch1, patch2)
            if sift_score < sift_best_score:
                sift_second_best_score = sift_best_score
                sift_best_score = sift_score
                sift_best_index = des2_index
        sift_score_ratio = float(sift_best_score)/float(sift_second_best_score)
        if sift_score_ratio > threshold_ratio:
            continue
        point_temp = kp1[des1_index].pt
        point1 = np.array([point_temp[1], point_temp[0], 1.0])
        point_temp = kp2[sift_best_index].pt
        point2 = np.array([point_temp[1], point_temp[0], 1.0])
        sift_correspondences.append([point1, point2])
        sift_scores.append(sift_best_score)
    sift_correspondences = np.array(sift_correspondences)
    sift_scores = np.array(sift_scores)
    return sift_correspondences, sift_scores

def automatic_keypoint_detection(img1, img2, min_pts=15,
                                 starting_ratio=0.2,
                                 max_threshold=0.9,
                                 threshold_delta=0.05):
    # Initiate SIFT detector
```

```
    sift = cv2.SIFT()
    # find the keypoints and descriptors with SIFT
    kp1, des1 = sift.detectAndCompute(img1,None)
    kp2, des2 = sift.detectAndCompute(img2,None)
    # BFMatcher with default params
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des1,des2, k=2)
    # Apply ratio test
    threshold = starting_ratio
    while(True):
        threshold += threshold_delta
        if threshold >= max_threshold:
            threshold = max_threshold
            break
        sift_correspondences, sift_scores = \
                    get_correspondences_sift(kp1, des1, kp2, des2, threshold)

        if len(sift_correspondences) >= min_pts:
            threshold += threshold_delta
            break
    sift_correspondences, sift_scores = \
                    get_correspondences_sift(kp1, des1, kp2, des2, threshold)
    print "Number of correspondences detected: ", len(sift_correspondences)
    print "Sift threshold used:", threshold
    return sift_correspondences, sift_scores


def get_patch(image, y_coord, x_coord, window_size):
    patch_radius = int(window_size/2)

    y_min = y_coord-patch_radius
    y_max = y_coord+patch_radius
    x_min = x_coord-patch_radius
    x_max = x_coord+patch_radius

    if y_min < 0:
        y_min = 0
    if y_max > image.shape[0]:
        y_max = image.shape[0]
    if x_min < 0:
        x_min = 0
    if x_max > image.shape[1]:
        x_max = image.shape[1]

    return image[y_min:y_max, x_min:x_max]

def _make_same_sized_patches(patch1, patch2):
    if patch1.shape[0] < patch2.shape[0]:
        max_size_0 = patch2.shape[0]
    else:
        max_size_0 = patch1.shape[0]
```

```
    if patch1.shape[1] < patch2.shape[1]:
        max_size_1 = patch2.shape[1]
    else:
        max_size_1 = patch1.shape[1]

    patch1_new = np.zeros((max_size_0, max_size_1))
    patch2_new = np.zeros((max_size_0, max_size_1))

    patch1_new[0:patch1.shape[0], 0:patch1.shape[1]] = patch1
    patch2_new[0:patch2.shape[0], 0:patch2.shape[1]] = patch2

    return patch1_new, patch2_new

def get_ncc_score(patch1, patch2):
    """
    Returns the normalized cross correlation between the two patches.

    NCC varies between -1 and 1, an NCC value that tends to 1 indicates a
    strong match.

    Args:
        patch1: An nxn NumPy array that represents a square patch from
                image 1.

        patch2: An nxn NumPy array that represents a square patch from
                image 2.

    Returns:
        ncc: The normalized cross correlation between the two patches.
    """
    if patch1.shape != patch2.shape:
        patch1, patch2 = _make_same_sized_patches(patch1, patch2)

    mean_patch1 = np.mean(patch1)
    mean_patch2 = np.mean(patch2)
    #Calculate the deviation from the mean:
    mean_dev1 = patch1 - mean_patch1
    mean_dev2 = patch2 - mean_patch2
    sq_mean_dev1 = mean_dev1**2
    sq_mean_dev2 = mean_dev2**2
    sum_sq_mean_dev1 = sq_mean_dev1.sum()
    sum_sq_mean_dev2 = sq_mean_dev2.sum()
    #denominator of ncc
    denominator = np.sqrt(sum_sq_mean_dev1 * sum_sq_mean_dev2)
    #Sum of the element-wise product of the deviation from the mean
    numerator = np.multiply(mean_dev1,mean_dev2).sum()
    return numerator/denominator

def reconstruct_3D(img1_rec, img2_rec, mat_P_1, mat_P_2,
                   y_delta=9, threshold_ncc=0.85,
                   threshold_ncc_ratio=0.95, border=20):
    ##Init:
```

```
    img1_out = img1_rec.copy()
    img2_out = img2_rec.copy()
    #Convert to gray scale
    img1_gray = cv2.cvtColor(img1_rec, cv2.COLOR_BGR2GRAY)
    img2_gray = cv2.cvtColor(img2_rec, cv2.COLOR_BGR2GRAY)
    #Detect edges
    img1_canny = cv2.Canny(img1_gray, 100, 200)
    img2_canny = cv2.Canny(img2_gray, 100, 200)
    cv2.imwrite("canny_1.jpg", img1_canny)
    cv2.imwrite("canny_2.jpg", img2_canny)
    ##Detect correspondences:
    list_correspondences = []
    list_points_1 = []
    list_points_2 = []
    for i in range(border, img1_canny.shape[0]-border):
        for j in range(border, img1_canny.shape[1]-border):
            if img1_canny[i][j] < 255:
                continue
            patch1 = get_patch(img1_gray, i, j, 15)
            ncc_best_score = sys.float_info.min #higher is better
            ncc_seoncd_best_score = sys.float_info.min
            ncc_best_point = None
            for y in range(i-y_delta, i+y_delta,1):
                for x in range(border, img2_canny.shape[1]-border):
                    if img2_canny[y][x] < 255:
                        continue
                    patch2 = get_patch(img2_gray, y, x, 15)
                    p2 = np.array([y, x, 1.0]).reshape(3,1)
                    ncc_score = get_ncc_score(patch1, patch2)
                    if ncc_score > ncc_best_score:
                        ncc_seoncd_best_score = ncc_best_score
                        ncc_best_score = ncc_score
                        ncc_best_point = p2.copy()
            if ncc_best_point is None:
                continue
            if ncc_seoncd_best_score < 0.02:
                continue
            ncc_score_ratio = float(ncc_seoncd_best_score)/float(ncc_best_score)
            if ncc_best_score < threshold_ncc:
                continue
            if ncc_score_ratio < threshold_ncc_ratio:
                continue
            p1 = np.array([i, j, 1.0]).reshape(3,1)
            list_correspondences.append((p1,ncc_best_point))
            list_points_1.append(p1.T)
            list_points_2.append(ncc_best_point.T)
            color = ((np.random.randint(0,255,(1,3))).tolist())[0]
            cv2.circle(img1_out,(int(p1[1]),int(p1[0])),3,color,-1)
            cv2.circle(img2_out,(int(p2[1]),int(p2[0])),3,color,-1)
#         if len(list_correspondences) > 300:
#             break
    ##Triangulate the 3D position of the points:
```

```
    cv2.imwrite("canny_matches_1.jpg", img1_out)
    cv2.imwrite("canny_matches_2.jpg", img2_out)
    list_points_1 = np.array(list_points_1).reshape(len(list_points_1),3)
    list_points_2 = np.array(list_points_2).reshape(len(list_points_2),3)
    mat_points_3D = triangulate_to_3D(list_points_1, list_points_2, mat_P_1, mat_P_2)
    return mat_points_3D, img1_out, img2_out
```

7.2 hw9_main.py

```
import cv2
import numpy as np
from scipy.optimize import leastsq
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

import hw9_lib

np.set_printoptions(precision=3)

# path_to_image1 = "/media/shiva/MoarSpace/Coursework/Fall 2014/ECE 661/bitbucket/homework09/Pics/set2/
# path_to_image2 = "/media/shiva/MoarSpace/Coursework/Fall 2014/ECE 661/bitbucket/homework09/Pics/set2/
path_to_image1 = "/media/shiva/MoarSpace/Coursework/Fall 2014/ECE 661/bitbucket/homework09/Pics/set3/3.
path_to_image2 = "/media/shiva/MoarSpace/Coursework/Fall 2014/ECE 661/bitbucket/homework09/Pics/set3/4.

img1 = cv2.imread(path_to_image1)
img2 = cv2.imread(path_to_image2)

## Find correspondences:
sift_correspondences, sift_scores = hw9_lib.automatic_keypoint_detection(img1, img2)

image_side_by_side, offset_image1, offset_image2 = \
                                    hw9_lib.get_image_side_by_side(img1, img2)
cv2.imshow("side by side", image_side_by_side)
cv2.waitKey(10)
points_subpix_img1 = None
points_subpix_img2 = None
for correspondence in sift_correspondences:
    p1 = correspondence[0].reshape(3,1)
    if points_subpix_img1 is not None:
        points_subpix_img1 = np.vstack((points_subpix_img1,p1.T))
    else:
        points_subpix_img1 = p1.T.copy()
    p1 = p1+offset_image1
    p2 = correspondence[1].reshape(3,1)
    if points_subpix_img2 is not None:
        points_subpix_img2 = np.vstack((points_subpix_img2,p2.T))
    else:
        points_subpix_img2 = p2.T.copy()
    p2 = p2+offset_image2
    cv2.line(image_side_by_side, (p1[1],p1[0]), (p2[1],p2[0]), (255,0,0), 1)
    cv2.circle(image_side_by_side,(int(p1[1]),int(p1[0])),2,(255,0,0),-1)
    cv2.circle(image_side_by_side,(int(p2[1]),int(p2[0])),2,(0,0,255),-1)
```

```
cv2.imshow("Side by side", image_side_by_side)
cv2.imwrite("correspondences_sift.jpg", image_side_by_side)


mat_F, mat_P_1, mat_P_2 = hw9_lib.compute_mat_F(points_subpix_img1,
                                                points_subpix_img2,
                                                use_nonlinear=True,
                                                use_normalization=True)
np.save("mat_P_1", mat_P_1)
np.save("mat_P_2", mat_P_2)
vec_e_1,rank_1 = hw9_lib.get_null_space(mat_F)
vec_e_2,rank_2 = hw9_lib.get_null_space(mat_F.T)
vec_e_1 = vec_e_1/vec_e_1[2]
vec_e_2 = vec_e_2/vec_e_2[2]
print "Epipole 1:", vec_e_1.T
print "Epipole 2:", vec_e_2.T

##Image rectification:
#mat_H2 computation:
img_ht = img2.shape[0]/2.0
img_wd = img2.shape[1]/2.0
theta = np.arctan(-(img_ht - vec_e_2[0])/(img_wd-vec_e_2[1]))
# print "Theta: ", (theta*180.0/np.pi)
mat_R = hw9_lib.get_2DRBT_rotation(theta)
mat_T = hw9_lib.get_2DRBT_translation(-img_ht, -img_wd)
vec_e2_origin = np.dot(np.dot(mat_R, mat_T), vec_e_2)
# print "vec_e2_origin", vec_e2_origin.T
scalar_f = vec_e2_origin[1]
# print "f: ", f
mat_G = hw9_lib.get_matrix_G(scalar_f)
mat_H2_temp = np.dot(mat_G, np.dot(mat_R, mat_T))
vec_e2_rectified = np.dot(mat_H2_temp, vec_e_2)
print "Rectified epipole 2: ", vec_e2_rectified.T
vec_center_2_original = np.array([img_ht, img_wd, 1.0]).reshape(3,1)
vec_center_2_new = np.dot(mat_H2_temp, vec_center_2_original)
vec_center_2_new = vec_center_2_new/vec_center_2_new[2]
print "vec_center_2_new",vec_center_2_new.T
mat_T2 = hw9_lib.get_2DRBT_translation((img_ht-vec_center_2_new[0]),
                                       (img_wd-vec_center_2_new[1]))
mat_H2 = np.dot(mat_T2, mat_H2_temp)

#mat_H1 computation:
img_ht = img1.shape[0]/2.0
img_wd = img1.shape[1]/2.0
theta = np.arctan(-(img_ht - vec_e_1[0])/(img_wd-vec_e_1[1]))
# print "Theta: ", (theta*180.0/np.pi)
mat_R = hw9_lib.get_2DRBT_rotation(theta)
mat_T = hw9_lib.get_2DRBT_translation(-img_ht, -img_wd)
vec_e1_origin = np.dot(np.dot(mat_R, mat_T), vec_e_1)
# print "vec_e2_origin", vec_e2_origin.T
scalar_f = vec_e1_origin[1]
```

```
# print "f: ", f
mat_G = hw9_lib.get_matrix_G(scalar_f)
mat_H1_temp = np.dot(mat_G, np.dot(mat_R, mat_T))
vec_e1_rectified = np.dot(mat_H1_temp, vec_e_1)
print "Rectified epipole 1: ", vec_e1_rectified.T
vec_center_1_original = np.array([img_ht, img_wd, 1.0]).reshape(3,1)
vec_center_1_new = np.dot(mat_H1_temp, vec_center_1_original)
vec_center_1_new = vec_center_1_new/vec_center_1_new[2]
print "vec_center_1_new",vec_center_1_new.T
mat_T2 = hw9_lib.get_2DRBT_translation((img_ht-vec_center_1_new[0]),
                                       (img_wd-vec_center_1_new[1]))
mat_H1 = np.dot(mat_T2, mat_H1_temp)

#Check correspondences
points_1_rec = (np.dot(mat_H1, points_subpix_img1.T)).T
points_1_rec = hw9_lib.normalize_vector_array(points_1_rec)
points_2_rec = (np.dot(mat_H2, points_subpix_img2.T)).T
points_2_rec = hw9_lib.normalize_vector_array(points_2_rec)
# print "Correspondences:"
y_diff = []
for i in range(len(points_1_rec)):
#     print "{0} :: {1}".format(points_1_rec[i], points_2_rec[i])
    y_diff.append(points_1_rec[i,0]-points_2_rec[i,0])

y_diff = np.array(y_diff)
print "y_diff.max(): ", y_diff.max()
print "y_diff.min(): ", y_diff.min()
print "y_diff.mean(): ", y_diff.mean()

img1_rec, image1_vec_offsets = hw9_lib.project_image_using_homogrpahy(img1, mat_H1)
img2_rec, image2_vec_offsets = hw9_lib.project_image_using_homogrpahy(img2, mat_H2)

image_rec_side_by_side, offset_image1, offset_image2 = \
                                    hw9_lib.get_image_side_by_side(img1_rec, img2_rec)
for i in range(len(points_1_rec)):
    p1 = (points_1_rec[i].reshape(3,1))+offset_image1
    p2 = (points_2_rec[i].reshape(3,1))+offset_image2
#     cv2.circle(image_rec_side_by_side,(int(p1[1]),int(p1[0])),2,(255,0,0),-1)
#     cv2.circle(image_rec_side_by_side,(int(p2[1]),int(p2[0])),2,(0,0,255),-1)
#     cv2.line(image_rec_side_by_side, (p1[1],p1[0]), (p2[1],p2[0]), (0,255,0), 1)
cv2.imshow("Rectified images", image_rec_side_by_side)
cv2.waitKey(10)
cv2.imwrite("img2_rec.jpg", img2_rec)
cv2.imwrite("img1_rec.jpg", img1_rec)
cv2.imwrite("rectified_images.jpg",image_rec_side_by_side)

y_delta = 3*int(np.ceil(np.abs(y_diff.mean()))) #3*mean difference
mat_points_3D, img1_out, img2_out = hw9_lib.reconstruct_3D(img1_rec, img2_rec, mat_P_1, mat_P_2, y_delta

print "mat_points_3D.shape", mat_points_3D.shape
print mat_points_3D
```

```
mat_points_3D_sparse = mat_points_3D[0:5:,:]

fig = plt.figure()
color='r'
marker='o'
ax = fig.add_subplot(111, projection='3d')
for point_3D in mat_points_3D_sparse:
    ax.scatter(point_3D[1], point_3D[0], point_3D[2], c=color, marker=marker)

plt.show()


cv2.waitKey(0)
```

7.3 hw9_manual_poi_selector.py

```
"""
Provides a GUI to interact with the images.

This module provides a front end GUI which can be used to interact with
images. The code in this module is based on code found at:
http://docs.opencv.org/trunk/doc/py_tutorials/py_gui/py_mouse_handling/py_mouse_handling.html

Course: ECE661: Computer Vision, Fall 2014
Homework: 9
Name: Shiva Ghose
Email: ghose0@purdue.edu
Date: 11.19.2014
"""
import sys
import logging

import cv2
import numpy as np

#Global variables
global_points_img1 = []
global_points_img2 = []
global_current_correspondence_img1 = -1
global_current_correspondence_img2 = -1
global_poi_color = (0,200,0)
global_color_counter = 0
global_colors = [(0,200,0), (0,0,200), (200,0,0),
                 (200, 200,0), (0,200,200), (200,0,200)]
global_img = None
global_image_selector = True


def _handle_mouse_event(event,x,y,flags,param):
    """
    Stores the left-click location on an image.

    This method handles left-click operations and sets the coordinates of the
```

```
    click location to the global variable that represents the point of
    interest.
    """
    if event == cv2.EVENT_LBUTTONDOWN:
        #Get global variables
        global global_poi_color, global_img
        global global_image_selector
        global global_current_correspondence_img1
        global global_current_correspondence_img2
        global global_image_selector
        #Store the POI
        point_of_interest = np.array([y, x,1.0])
        if global_image_selector:
            global_current_correspondence_img1 = point_of_interest
        else:
            global_current_correspondence_img2 = point_of_interest
        #Mark the POI
        cv2.circle(global_img,(x,y),5,global_poi_color,-1)
        logging.info("Registered point: {0}".format(point_of_interest))
        global_image_selector = not global_image_selector


def _reset():
    global global_current_correspondence_img1
    global global_current_correspondence_img2
    global global_image_selector
    global_current_correspondence_img1 = -1
    global_current_correspondence_img2 = -1
    global_image_selector = True

def _next_poi():
    """
    Prepares for next correspondence selection.
    """
    global global_image_selector
    global global_points_img1, global_points_img2
    global global_current_correspondence_img1
    global global_current_correspondence_img2
    global_points_img1.append(global_current_correspondence_img1)
    global_points_img2.append(global_current_correspondence_img2)
    global_image_selector = True
    _next_color()

def _next_color():
    """
    Sets global_color from a set of colors.
    """
    global global_color_counter, global_poi_color, global_colors
    global_color_counter += 1
    i = global_color_counter%(len(global_colors))
    global_poi_color = global_colors[i]
```

```
def main(image1, image2):
    ##Init:
    global global_img, global_image_selector
    global global_points_img1, global_points_img2

    cv2.namedWindow('Select POI')
    #Handle mouse click operations:
    cv2.setMouseCallback('Select POI',_handle_mouse_event)

    print "Press the escape key, 'ESC', to exit."
    print "Press 'r' to reset current selection."
    print "Press 'n' to select the next set of correspondences."
    print "Press 'p' to move on to processing."
    ##Display the image:
    while(1):
        if global_image_selector:
            global_img = image1
        else:
            global_img = image2
        cv2.imshow('Select POI',global_img)
        k = cv2.waitKey(10) & 0xFF
        #Check to see if the escape key has been pressed:
        if k == 27:
            break
        elif k == ord('n'):
            _next_poi()
            print "Ready for new point selection."
        elif k == ord('p'):
            _next_poi()
            print "Points selected in image 1:"
            for point in global_points_img1:
                print point
            print "Points selected in image 2:"
            for point in global_points_img2:
                print point
            break
    cv2.destroyWindow('Select POI')
    print "Saving images as img_1_manual_poi.jpg and img_2_manual_poi.jpg"
    cv2.imwrite("img_1_manual_poi.jpg",image1)
    cv2.imwrite("img_2_manual_poi.jpg",image2)
    print "Returning selected points."
    return np.array(global_points_img1), np.array(global_points_img2), image1, image2


if __name__ == "__main__":
    args = sys.argv[1:]
    if len(args) < 1:
        path_to_image1 = "/media/shiva/MoarSpace/Coursework/Fall 2014/ECE 661/bitbucket/homework09/Pics/
        path_to_image2 = "/media/shiva/MoarSpace/Coursework/Fall 2014/ECE 661/bitbucket/homework09/Pics/
    else:
        path_to_image1 = args[0]
```

```
    path_to_image2 = args[1]
image1 = cv2.imread(path_to_image1)
image2 = cv2.imread(path_to_image2)
main(image1, image2)
```