Angel Nieto García 2211798052

Homework #3

1. Source code

Code was made based in the provided skeleton

```
# ------------------------------------------------------------------------
# Given K from intrinsics.txt
K = np.array([[3954.7557192818421754, -8.7442247413155467228, 1619.9232700857789951],
            [0., 3948.0083629740670403, 1151.4558912976590364],
            [0., 0., 1.]])

def run(self):

        self.load_images()

        # pair processing:
        idx = 0
        pair_images = (self.imgs[0], self.imgs[1])
        # pair_images = (self.imgs[1], self.imgs[2])
        # pair_images = (self.imgs[0], self.imgs[2])

        # step 1 and 2: detect and match feature
        p1, p2, matches_good, kp1, kp2 = self.detect_and_match_feature(
            pair_images[0], pair_images[1])

        self.visualize_matches(
            pair_images[0], pair_images[1], kp1, kp2, matches_good,
            save_path=join(self.output_dir, 'sift_match' + str(idx) + '.png'))

        # step 3: compute essential matrix
        E, mask = self.compute_essential(p1, p2)

        self.visualize_matches(
            pair_images[0], pair_images[1], kp1, kp2, matches_good, mask=mask,
            save_path=join(self.output_dir, 'inlier_match' + str(idx) + '.png'))

        self.visualize_epipolar_lines(
            pair_images[0], pair_images[1], p1, p2, E,
            save_path=join(self.output_dir, 'epipolar_lines' + str(idx) + '.png'))

        # step 4: recover pose
        R, trans = self.compute_pose(p1, p2, E)

        # step 5: triangulation
        point_3d = self.triangulate(p1, p2, R, trans, mask, idx)
        self.write_simple_obj(point_3d, None, filepath=join(
            self.output_dir, 'output' + str(idx) + '.obj'))

        # store points_3d in txt
```

```
        output_f = open("points_3d_case" + str(idx) + ".txt", 'w')
        for point in point_3d:
            output_f.write(str(point))
            output_f.write('\n')
        output_f.close()
```

```
def detect_and_match_feature(self, img1, img2):
        """
        img1, img2: are input images
        The following outputs are needed:
        kp1, kp2: keypoints (here sift keypoints) of the two images
        matches_good: matches which pass the ratio test
        p1, p2: only the 2d points in the respective images
        pass ratio test. These points should correspond to each other.

        Steps:
        1. Compute sift descriptors.
        2. Match sift across two images.
        3. Use ratio test to get good matches.
        4. Store points retrieved from the good matches.

        Hints: See SIFT_create
        For feature matching you could use
        - FLANN Matcher:
        (https://docs.opencv.org/3.4/dc/de2/classcv_1_1FlannBasedMatcher.html)
        """
        # TODO: step 1 and 2
        # ...
        sift = cv2.xfeatures2d.SIFT_create()

        kp1, des1 = sift.detectAndCompute(img1, None)
        kp2, des2 = sift.detectAndCompute(img2, None)

        # FLANN parameters
        FLANN_INDEX_KDTREE = 0
        index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
        search_params = dict(checks=50)   # or pass empty dictionary

        flann = cv2.FlannBasedMatcher(index_params, search_params)

        matches = flann.knnMatch(des1, des2, k=2)

        matches_good = []
        # ratio test as per Lowe's paper
        for _ ,(m,n) in enumerate(matches):
            if m.distance < 0.7 * n.distance:
                matches_good.append(m)

        # get matching points from view1 and view 2 by matching matches_good indices with
descriptors kp1 and kp2 based on:
        # https://opencv-python-
tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homogr
aphy.html
```

```
        p1 = np.asarray([kp1[m.queryIdx].pt for m in matches_good]).astype(np.float32).reshape(-
1,1,2)
        p2 = np.asarray([kp2[m.trainIdx].pt for m in matches_good]).astype(np.float32).reshape(-
1,1,2)
```

```
def compute_essential(self, p1, p2):
        """
        p1, p2: only the 2d points in the respective images
        pass ratio test. These points should correspond to each other.
        Outputs:
        Essential Matrix (E), and corresponding (mask)
        used in its computation. The mask contains the inlier_matches
        to compute E

        Hint: findEssentialMat
        """
        return cv2.findEssentialMat(p1, p2, K, method=cv2.RANSAC, prob=0.999, threshold=1.0)
```

```
def compute_pose(self, p1, p2, E):
        """
        p1, p2: only the 2d points in the respective images
        pass ratio test. These points should correspond to each other.
        E: Essential matrix
        Outputs:
        R, trans: Rotation, Translation vectors

        Hint: recoverPose
        """

        # TODO: step 4
        # ...
        _, R, trans, _ = cv2.recoverPose(E, p1, p2, K)
        return R, trans
```

```
    def triangulate(self, p1, p2, R, trans, mask, idx):
        """
        p1,p2: Points in the two images which correspond to each other
        R, trans: Rotation and translation matrix.
        mask: is obtained during computation of Essential matrix

        Outputs:
        point_3d: should be of shape (NumPoints, 3). The last dimension
        refers to (x,y,z) co-ordinates

        Hint: triangulatePoints
        """
        # TODO: step 5
        # ...
        matchesMask = mask.ravel().tolist()
```

```
        M_r = np.hstack((R, trans))
        M_l = np.hstack((np.eye(3, 3), np.zeros((3, 1))))

        # Discard outliers
        p1 = p1[np.asarray(matchesMask)==1,:,:]
        p2 = p2[np.asarray(matchesMask)==1,:,:]

        p1_un = cv2.undistortPoints(p1, K, None)
        p2_un = cv2.undistortPoints(p2, K, None)

        p1_un = np.squeeze(p1_un)
        p2_un = np.squeeze(p2_un)

        projection_matrix_r = np.dot(K, M_r)
        projection_matrix_l = np.dot(K, M_l)

        print('rotation case:' + str(idx))
        print(R)
        print('translation case:' + str(idx))
        print(trans)
        print('projection_matrix_l case:' + str(idx))
        print(projection_matrix_l)
        print('projection_matrix_r case:' + str(idx))
        print(projection_matrix_r)

        point_4d_hom = cv2.triangulatePoints(projection_matrix_l, projection_matrix_r, p1_un.T,
p2_un.T)
        point_4d = point_4d_hom / np.tile(point_4d_hom[-1, :], (4, 1))
        point_3d = point_4d[:3, :].T

        return point_3d
```

2 Results for t = 0.7

2.1 Rotation matrix

[[0.9894773 0.03736107 0.13978133] [-0.03990474 0.99908422 0.01543826] [-0.13907653 -0.02085374 0.99006204]]	[[0.99712814 0.0466656 0.05964729] [-0.0498411 0.99735482 0.0529077] [-0.05702055 -0.05572864 0.99681642]]	[[0.97689769 0.08316009 0.19686367] [-0.09676069 0.99346779 0.06049068] [-0.1905473 -0.07814187 0.97856301]]
R for rdimage000 and rdimage001	R for rdimage001 and rdimage002	R for rdimage000 and rdimage002

2.2 Translation vector/matrix

[[-9.99821590e-01] [2.90920108e-04] [1.88866196e-02]]	[[-0.53713824] [-0.34594363] [-0.76928897]]	[[-0.87862901] [-0.18336467] [-0.44089506]]
t for rdimage000 and rdimage001	t for rdimage001 and rdimage002	t for rdimage000 and rdimage002

2.3 Projection matrix

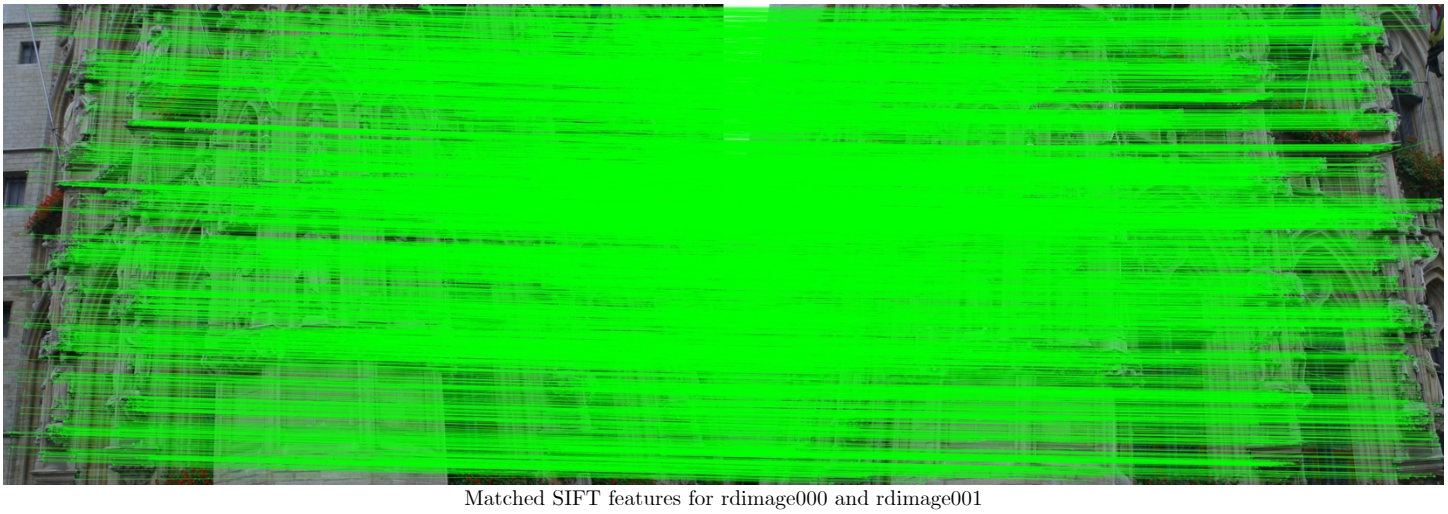
2.3.1 Left view = P = Id • K

[[3.95475572e+03 -8.74422474e+00 1.61992327e+03 0.00000000e+00] [0.00000000e+00 3.94800836e+03 1.15145589e+03 0.00000000e+00] [0.00000000e+00 0.00000000e+00 1.00000000e+00 0.00000000e+00]]	[[3.95475572e+03 -8.74422474e+00 1.61992327e+03 0.00000000e+00] [0.00000000e+00 3.94800836e+03 1.15145589e+03 0.00000000e+00] [0.00000000e+00 0.00000000e+00 1.00000000e+00 0.00000000e+00]]]]	[[3.95475572e+03 -8.74422474e+00 1.61992327e+03 0.00000000e+00] [0.00000000e+00 3.94800836e+03 1.15145589e+03 0.00000000e+00] [0.00000000e+00 0.00000000e+00 1.00000000e+00 0.00000000e+00]]
P_l for rdimage000 and rdimage001	P_l for rdimage001 and rdimage002	P_l for rdimage000 and rdimage002

2.3.2 Right view = P = M • K

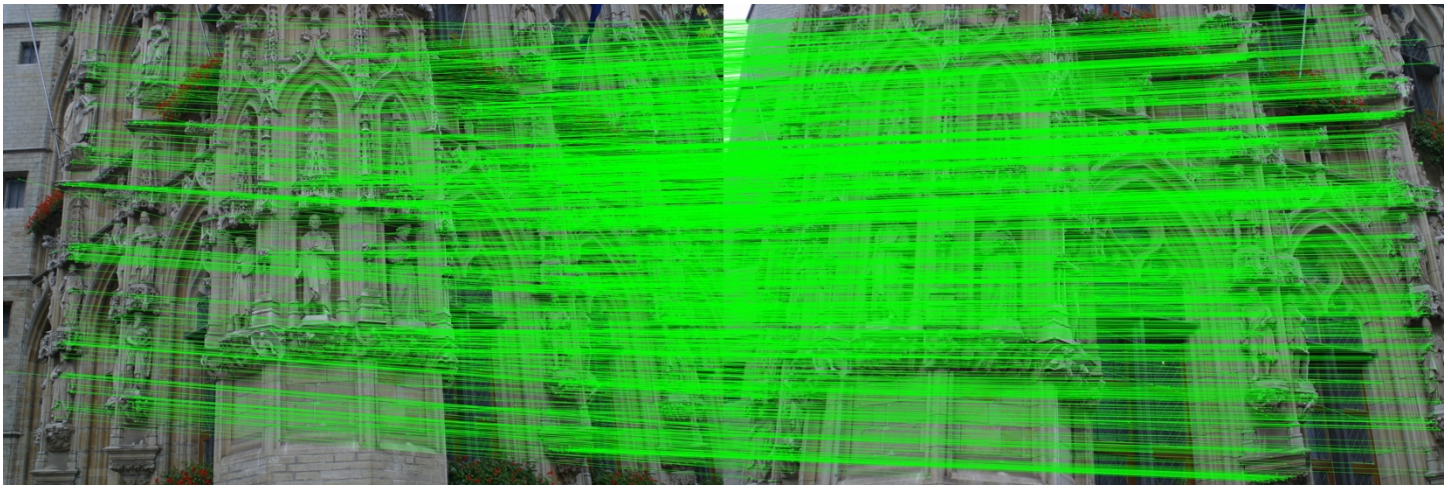
[[3.68819665e+03 1.05236235e+02 2.15649055e+03 -3.92345782e+03] [-3.17684748e+02 3.92038068e+03 1.20096313e+03 2.28956644e+01] [-1.39076531e-01 -2.08537412e-02 9.90062039e-01 1.88866196e-02]]	[[3.85146511e+03 8.55538359e+01 1.85019396e+03 -3.36741461e+03] [-2.62429725e+02 3.87339611e+03 1.35667017e+03 -2.25159067e+03] [-5.70205476e-02 -5.57286405e-02 9.96816420e-01 -7.69288967e-01]]	[[3.55556581e+03 1.93606922e+02 2.36321578e+03 -4.18737590e+03] [-6.01418814e+02 3.83224223e+03 1.36558985e+03 -1.23159647e+03] [-1.90547304e-01 -7.81418660e-02 9.78563015e-01 -4.40895061e-01]]
P_r for rdimage000 and rdimage001	P_r for rdimage001 and rdimage002	P_r for rdimage000 and rdimage002

2.4 Matched features



Matched SIFT features for rdimage000 and rdimage001

Matched SIFT features for rdimage001 and rdimage002



Matched SIFT features for rdimage000 and rdimage002

2.5 Pruned features

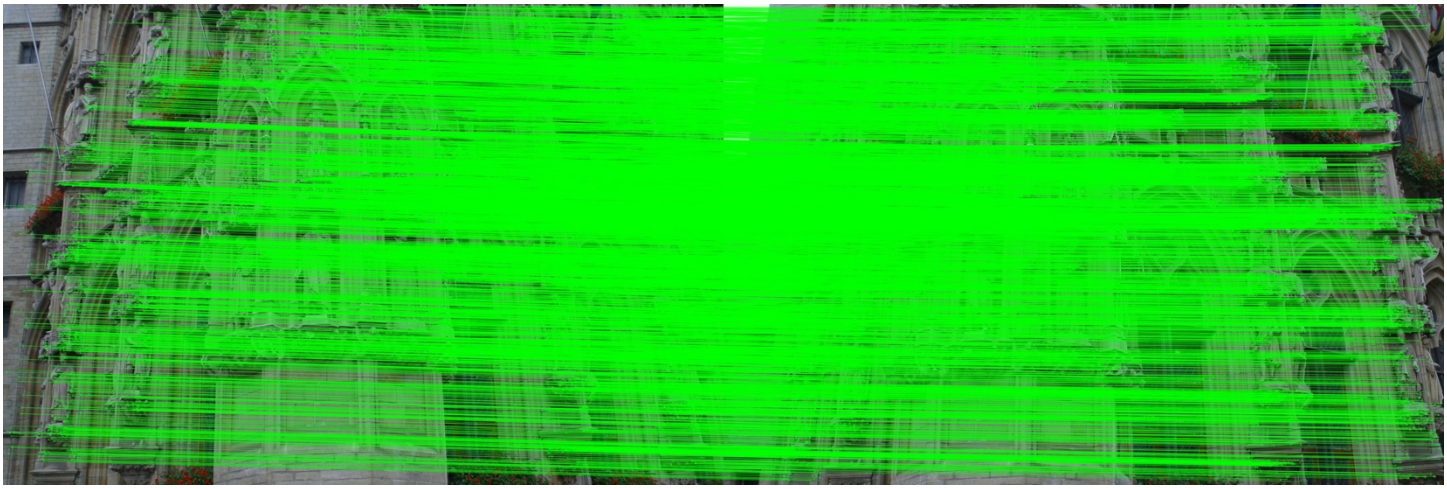
2.5.1 Epipolar lines

Epipolar lines for rdimage000 and rdimage001

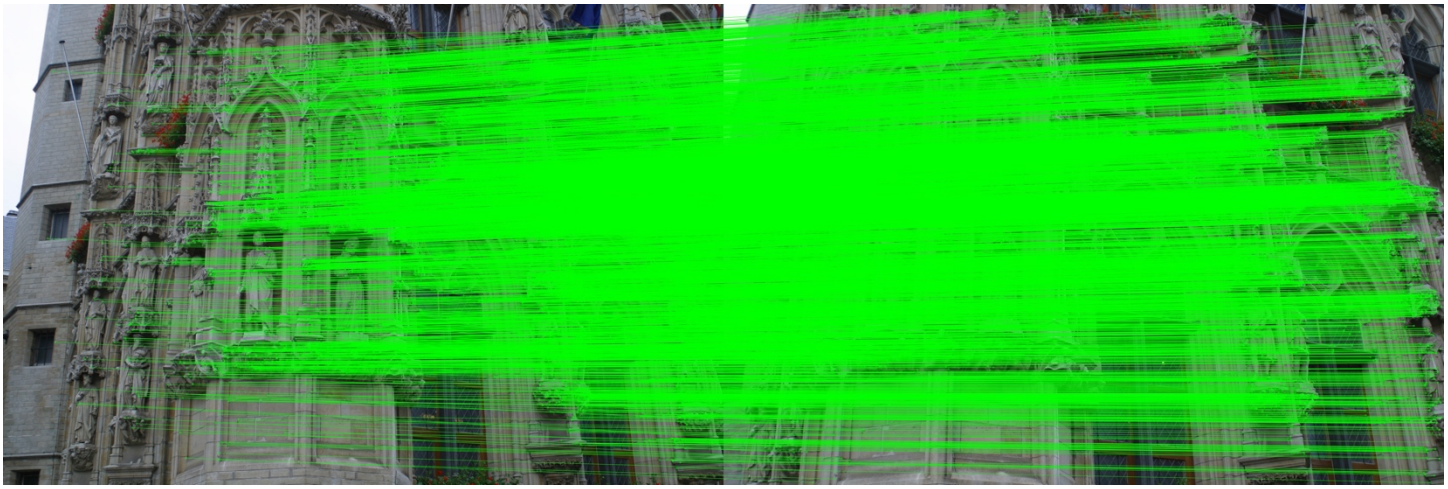
Epipolar lines for rdimage001 and rdimage002

Epipolar lines for rdimage000 and rdimage002

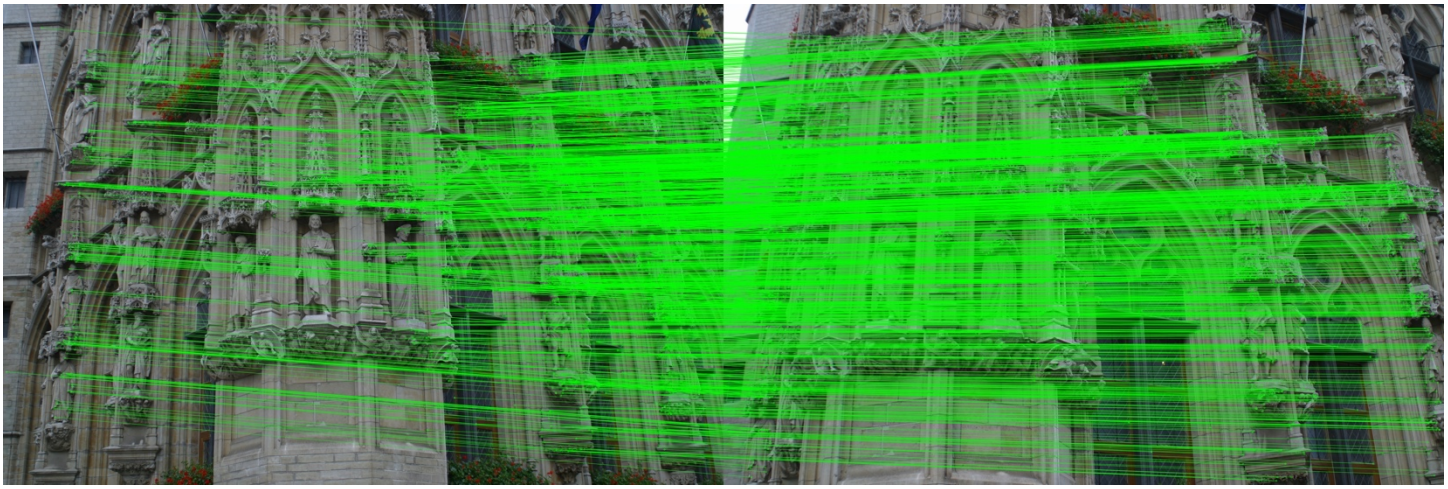
2.5.2 Inlier match



Inlier match for rdimage000 and rdimage001



Inlier match for rdimage001 and rdimage002



Inlier match for rdimage000 and rdimage002

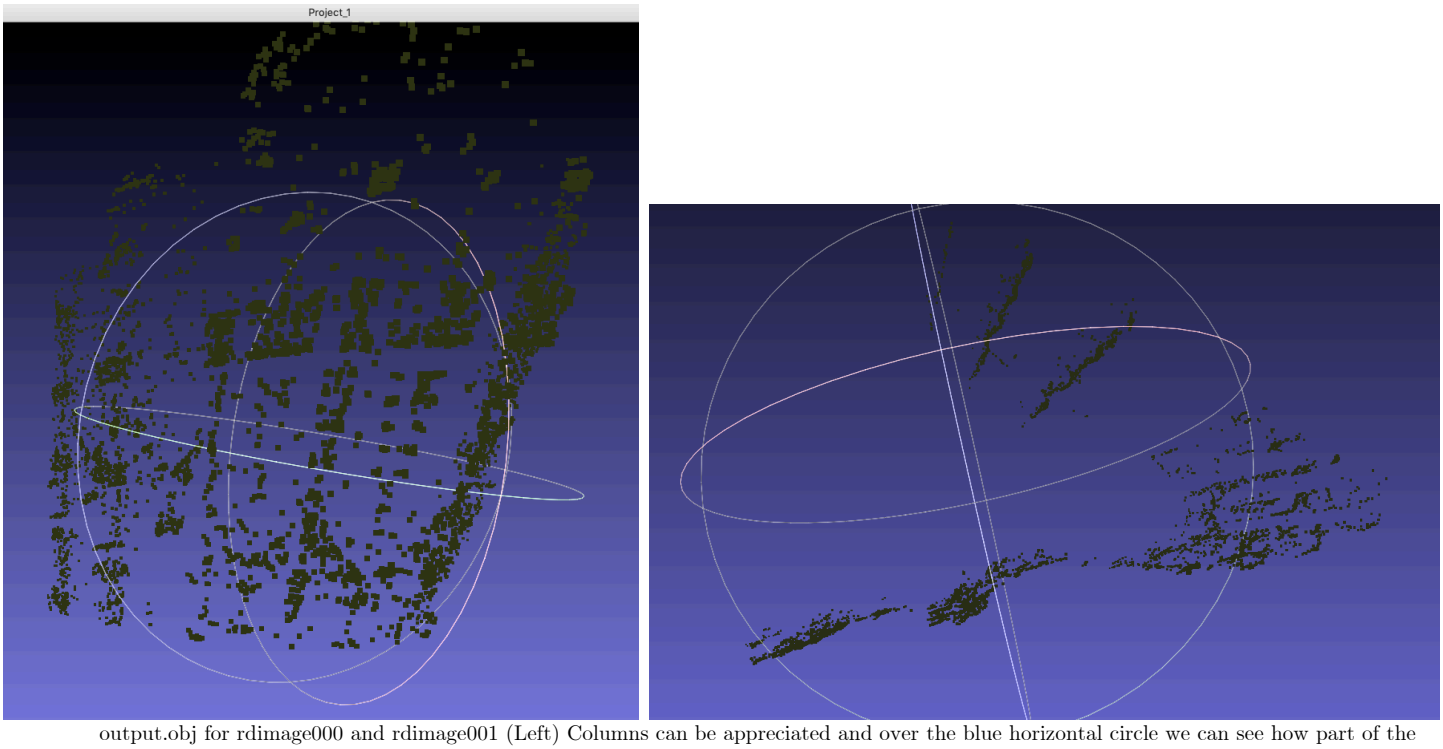
2.6 Reconstructed points

First and last 3 rows are shown here the rest can be found in the zip file

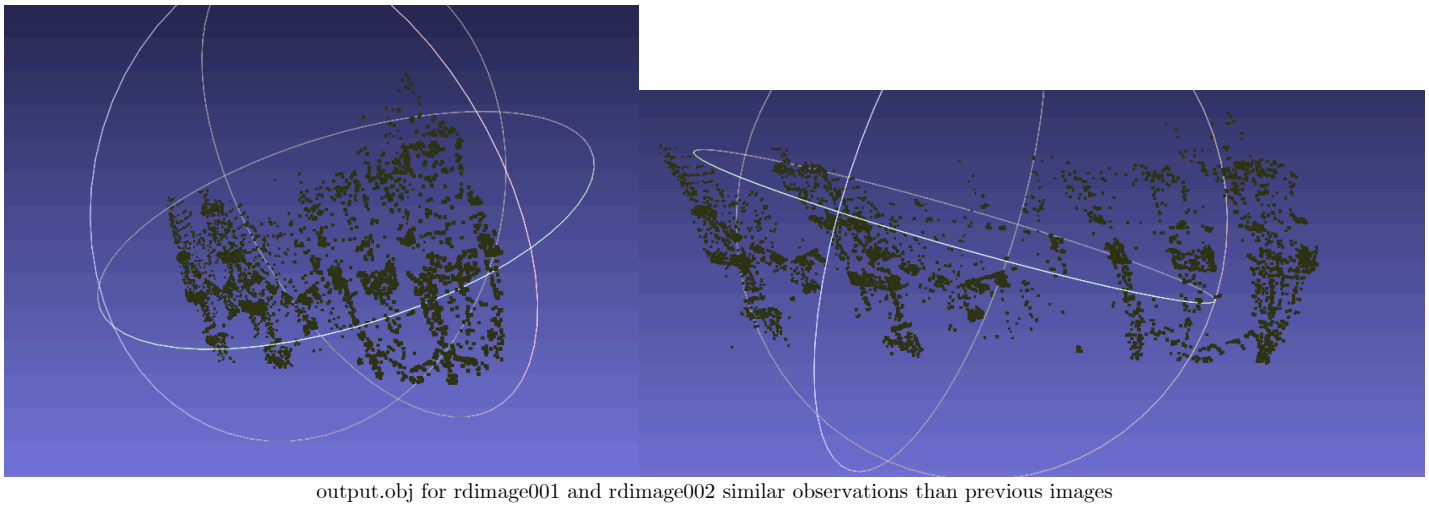
[[-2.6051643 -2.013611 6.3720336]	[[-4.1559567 -3.1827662 10.393883]	[[-1.820302 -1.45661 4.563359]
[-2.6051292 -2.0132935 6.371954]	[-4.1559277 -3.1832376 10.393866]	[-1.8202795 -1.4570673 4.563432]
[-2.6051195 -2.0131822 6.371932]	[-4.155953 -3.1824737 10.3939085]	[-1.8201872 -1.45651 4.563211]
...	...	...
[-2.6041946 -2.0133476 6.372616]	[-4.154289 -3.1829834 10.39442]	[-1.8196266 -1.4569205 4.563783]
[-2.6041913 -2.0133266 6.372608]	[-4.1542873 -3.1829798 10.394419]	[-1.8196223 -1.4569384 4.563778]
[-2.6041927 -2.013338 6.372618]]	[-4.154296 -3.1829698 10.39444]]	[-1.8196231 -1.4569281 4.563783]]
3d points for rdimage000 and rdimage001	3d points for rdimage001 and rdimage002	3d points for rdimage000 and rdimage002

2.7 Final output and optimal parameter

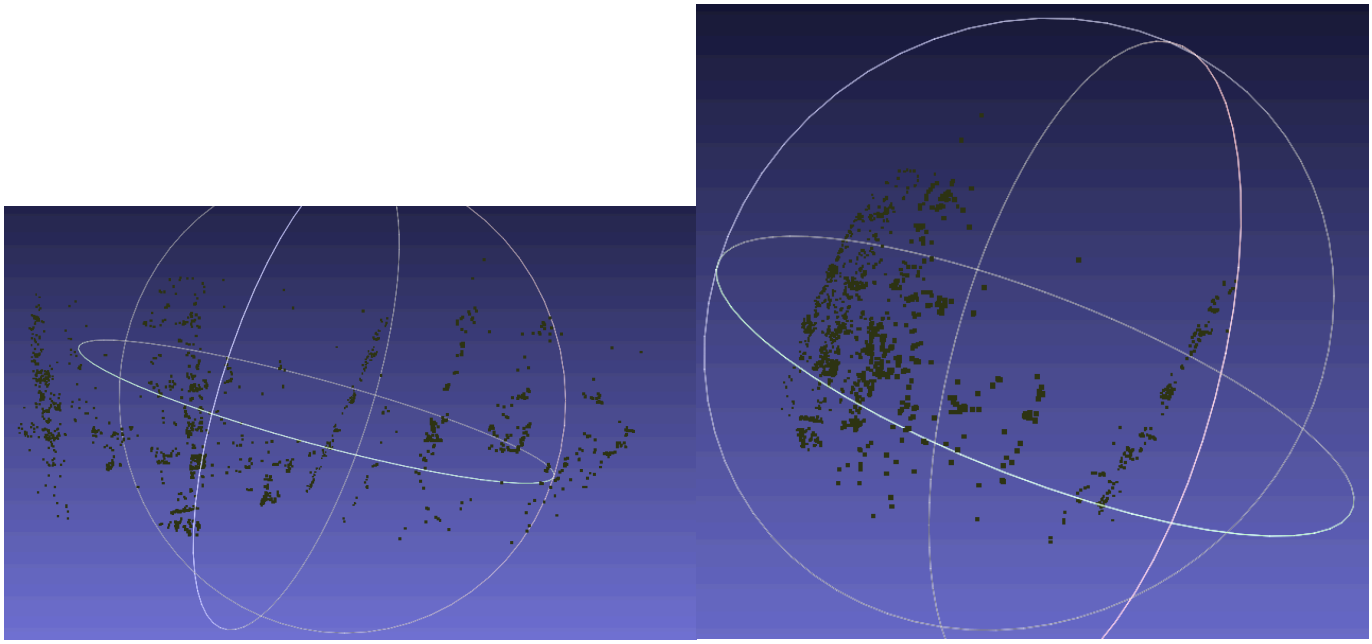
Selecting t = 0.7 in ratio test:



output.obj for rdimage000 and rdimage001 (Left) Columns can be appreciated and over the blue horizontal circle we can see how part of the statues are reconstructed (Right) Corner seen from above.

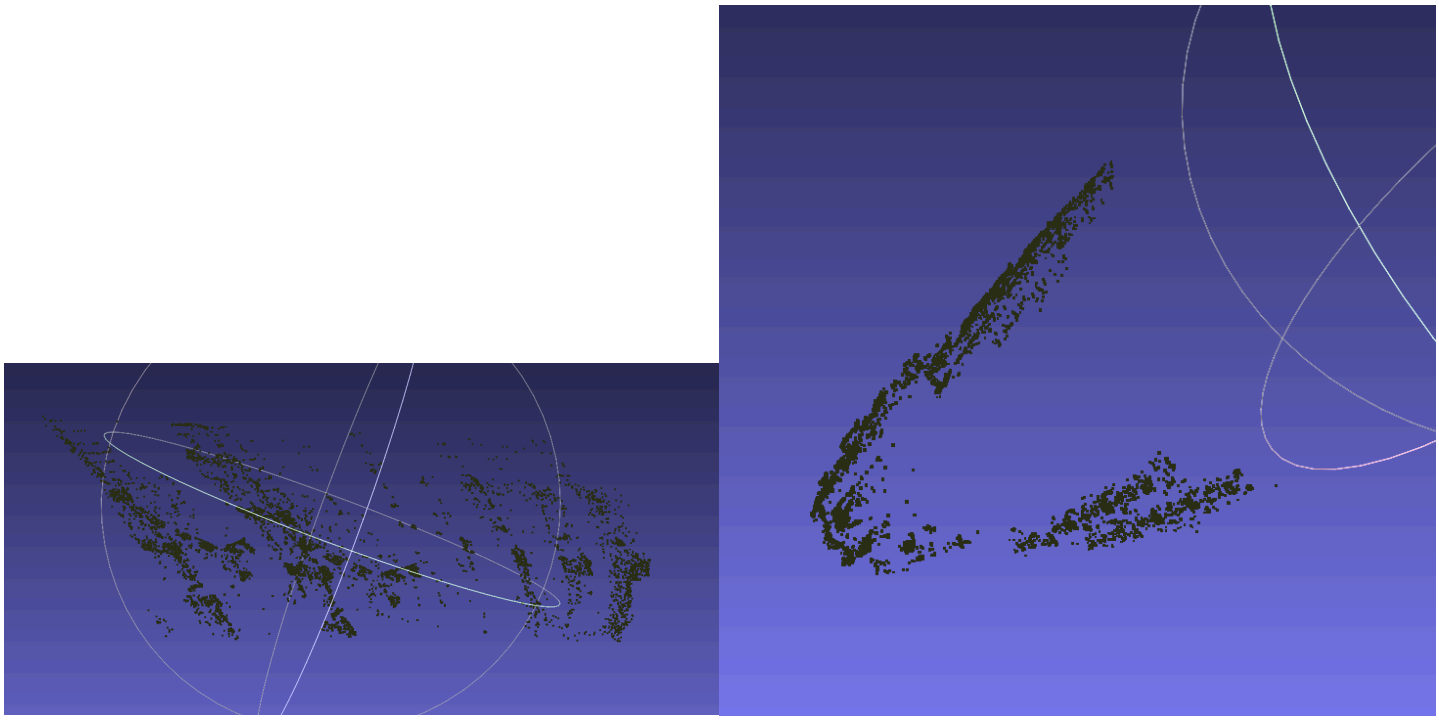


output.obj for rdimage001 and rdimage002 similar observations than previous images

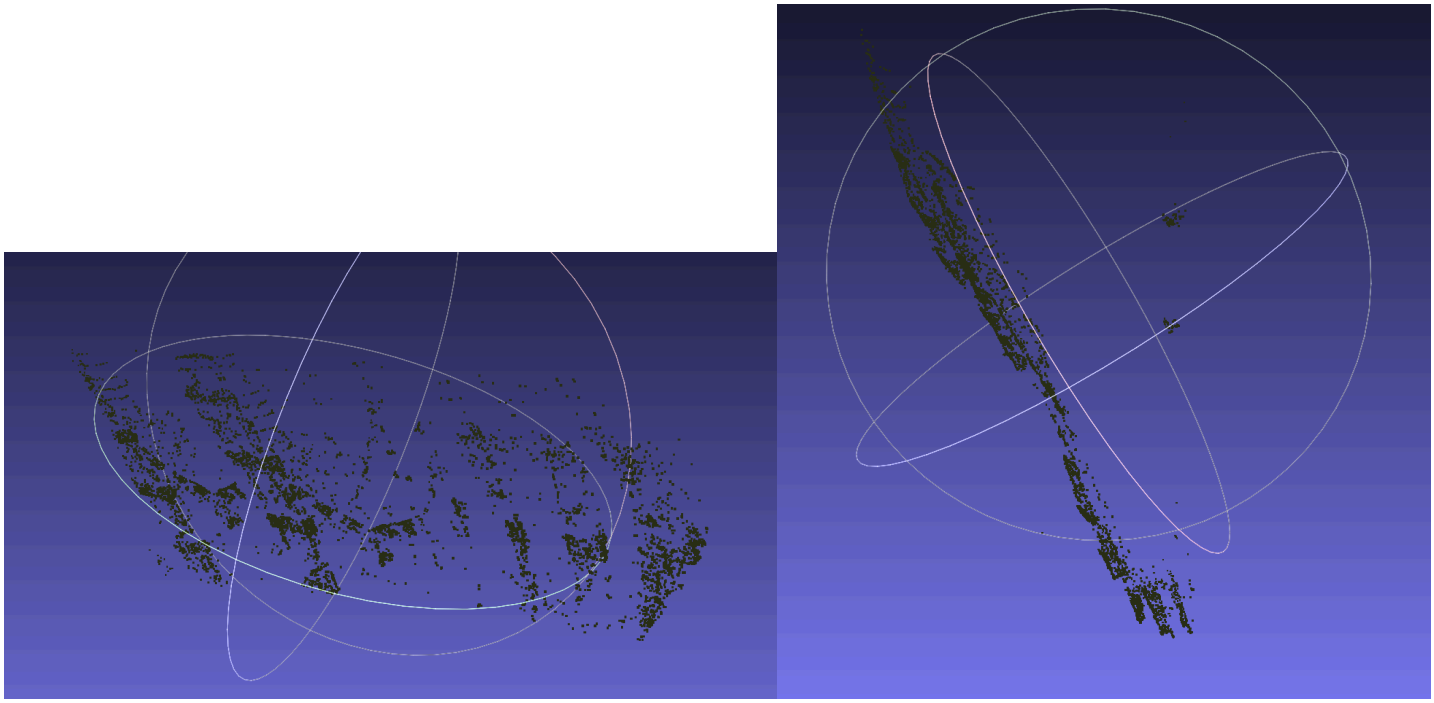


output.obj for rdimage000 and rdimage002 we can see that reconstruction points are more distant making more difficult to see the resemblance with original image

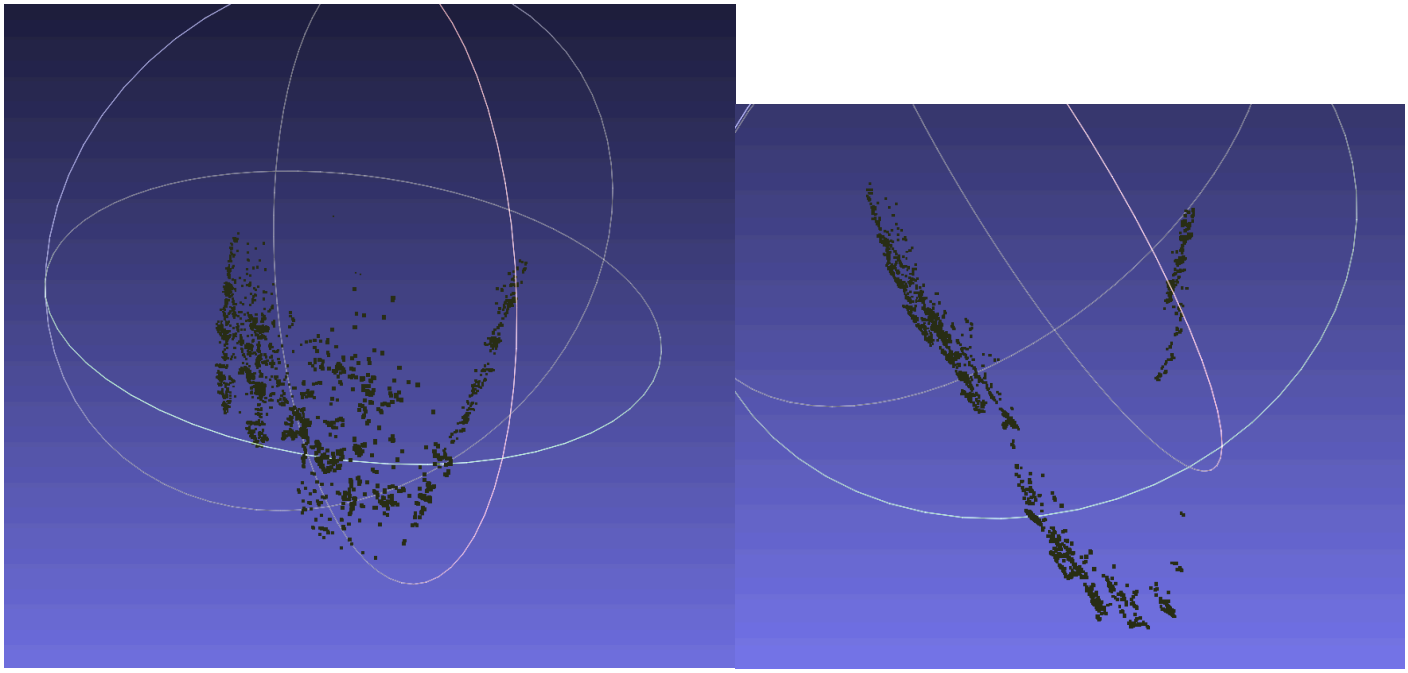
Selecting t = 0.8 in ratio test:



output.obj for rdimage000 and rdimage001 using t = 0.8



output.obj for rdimage001 and rdimage002 using t =0.8



output.obj for rdimage000 and rdimage002 using t =0.8

3. Analysis and conclusions

In general, the algorithm works very well when selecting t = 0.7 for the ratio test. As can be appreciated selecting t = 0.7 works satisfactory for the first combination of images (0-1 and 1-2) and t = 0.8 for the last two combinations of images (0-1 and 0-2), which makes sense for last combination since a greater t allows points matching at a longer distance. We can see the corner clearly with both parameters and any combination of photos. Nevertheless, seeing the front of the building output for t = 0.7 resembles more to the original images.

Additionally, we can notice that the first two combinations of images (0-1 and 1-2) have a similar or shorter angle of difference with respect to each other, which affects to the number of inliers matches directly with the combination (0-2) having 4x fewer inlier matches that the rest of the combinations. This conclusion confirms what we discuss in the lecture that images with small angles make reconstruction possible. On the other hand, a more considerable distance of photos from very different angles results in very bad or impossible reconstruction.

To improve output quality, I would try to combine more images into the output or use different features like HOG features.