Lab 6 – Structure from Motion

Computer Vision
263-5902-00L

Guntz Romain

21-828-348

Introduction:

This lab explores Structure from Motion, a method used to create 3D models from a series of 2D images. The process begins with estimating the relative pose between two images, which involves determining the rotation and translation between the two camera views. The second step is triangulation, where feature matches between the two images are used to compute the 3D positions of points in space. Finally, the iterative registration process incorporates new images into the scene by estimating the relative pose of each new image and triangulating additional 3D points based on the new image and existing points.

Implementation

Essential matrix estimation:

The first step is to normalize the matches from the image using the intrinsic matrix K. This is achieved with the following code:

```
  kps1 = im1[matches[:, 0]]
  kps2 = im2[matches[:, 1

  normalized_kps1 = (np.linalg.inv(K) @ np.hstack((kps1, np.ones((kps1.shape[0],
1)))).T).T[:, :2]
  normalized_kps2 = (np.linalg.inv(K) @ np.hstack((kps2, np.ones((kps2.shape[0],
1)))).T).T[:, :2]
```

Next, we construct the constrains of the equation $x_1^T E\ x_2$ with the following relations:

```
  constraint_matrix = np.zeros((matches.shape[0], 9))
  for i in range(matches.shape[0]):
    x1, y1 = normalized_kps1[i]
    x2, y2 = normalized_kps2[i]
    constraint_matrix[i] = [x1 * x2, x1 * y2, x1, y1 * x2, y1 * y2, y1, x2, y2, 1]
```

Next, we need to ensure that the matrix E satisfies the necessary constraints: the first two singular values must be equal to 1, and the third singular value must be equal to 0. This is achieved using the following code:

```
  U, S, Vt = np.linalg.svd(E_hat)
  S = np.array([1, 1, 0])
  E = U @ np.diag(S) @ Vt
```

Point Triangulation :

This method has already been implemented in the TriangulatePoints function within the geometry.py file. However, a filtering step was necessary to ensure that the points do not lie behind the camera, as this would be incorrect. This is achieved using the same code for both images:

```
  Camera_coords1 = (R1 @ points3D.T + t1.reshape(3, 1)).T
  valid_indices1 = camera_coords1[:, 2] > 0
  im1_corrs = im1_corrs[valid_indices1]
  im2_corrs = im2_corrs[valid_indices1]
  points3D = points3D[valid_indices1]
```

The first step is to obtain the camera coordinates from the 3D world coordinates of the point. Next, we check the third coordinate (z-coordinate), which represents the depth of the point relative to the camera. Only points with positive depth are retained.

Find the correct decomposition:

The decomposition of the matrix E into a relative pose is already implemented in the template. However, four different matrix decompositions are possible. The goal here is to determine the best decomposition.

```
  for R, t in possible_relative_poses:
      e_im1.SetPose(np.eye(3), np.zeros(3))
      e_im2.SetPose(R, t)

      points3D, im1_corrs, im2_corrs = TriangulatePoints(K, e_im1, e_im2,e_matches)
      num_points_in_front = np.sum(points3D[:, 2] > 0)

      if num_points_in_front > max_points:
          max_points = num_points_in_front
          best_pose = (R, t)
```

We first iterate through each possible pose. For each pose R and t, the 3D points are computed using the TriangulatePoints function. We then count how many points have a positive depth (are in front of the camera rather than behind it). The pose with the highest number of points with positive depth is selected.

Absolute pose estimation:

This task aims to determine the pose R and translation t of a camera relative to a 3D scene using 2D data points and their corresponding positions in 3D space.

The only step needed to complete the code was to normalize the points using the same formula applied for the Essential matrix estimation (2.1):

```
normalized_points2D = np.linalg.inv(K) @ np.hstack((points2D,
np.ones((points2D.shape[0], 1)))).T
normalized_points2D = normalized_points2D[:2, :].T
```

Map extension:

The next step is to extend the map by adding new points. This is done using the functions TriangulateImage and UpdateReconstructionState. The first step involves initializing an array of 3D points, points3D.

```
                                points3D = np.zeros((0,3)
```

The first step is to iterate through every image that has already been registered and find the matches:

```
for reg_image_name in registered_images:
      if reg_image_name not in matches[image_name]:
          continue
```

```
        match = matches[image_name][reg_image_name]
```

Furthermore, the rotation, translation, and projection matrices are computed in this code snippet:

```
        R1, t1 = image.GetPose()
        R2, t2 = images[reg_image_name].GetPose()
        P1 = K @ np.hstack((R1, t1.reshape(-1, 1)))
        P2 = K @ np.hstack((R2, t2.reshape(-1, 1)))
```

Then, the 3D point is created using the TriangulatePoints function.

Results

The results obtained are images with the corresponding matches between one image and another. This is shown in the following image:

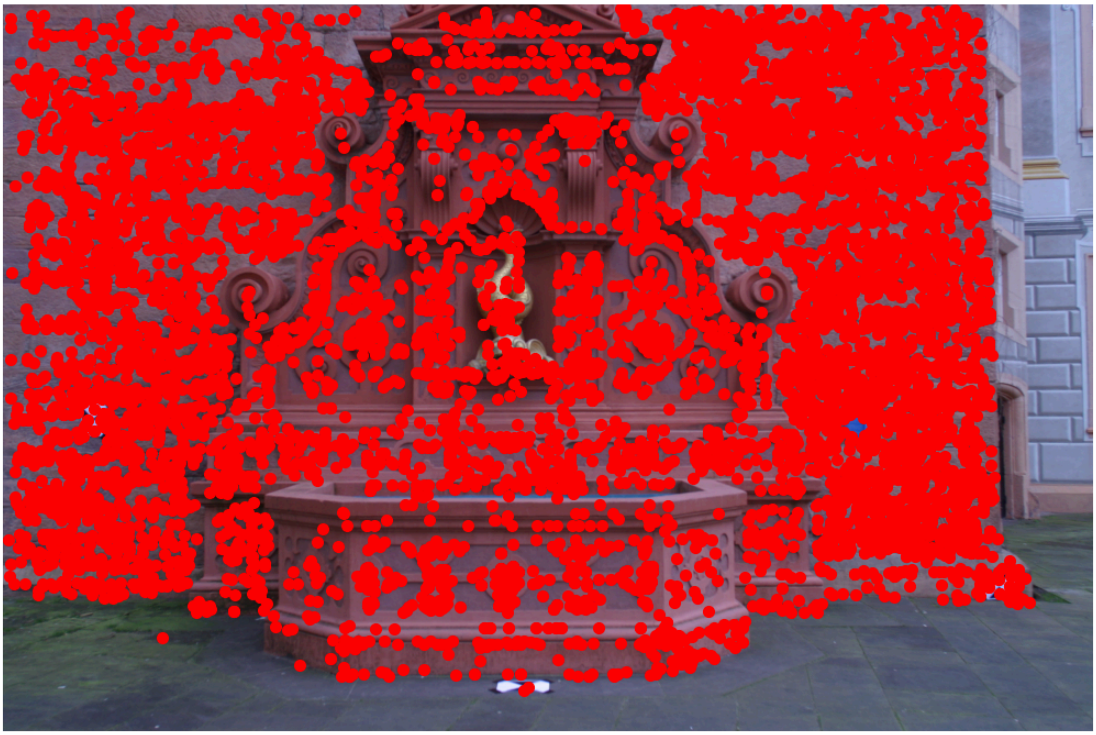

Figure 1: Matches Of One Image

Furthermore, a second result is available: the match between two images:

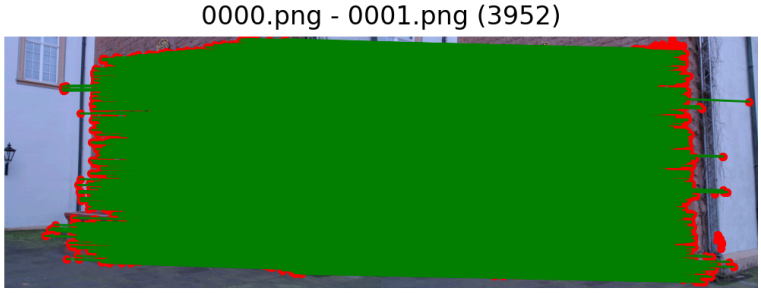

Figure 2: Matches between two images

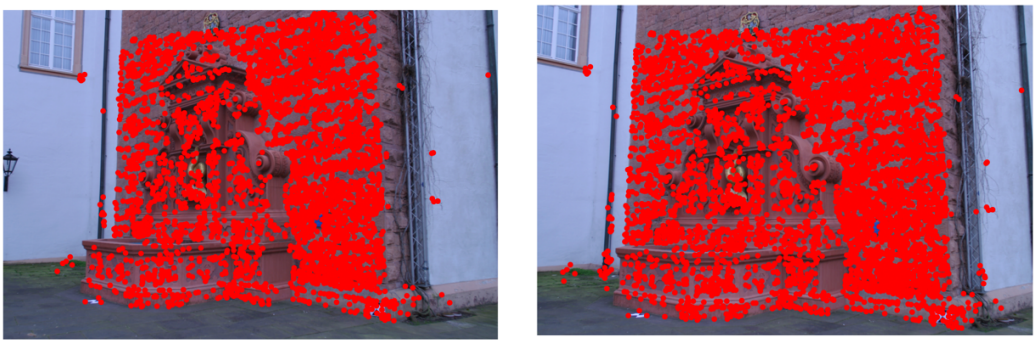

Figure 2: Set of images matches

The correspondence of key points from one image to another is quite accurate, as the same points in the 3D world appear to be well associated with their corresponding points in the two images. However, when the pose of the images is drastically different, as in the case of the following two images, some regions of the images lack matches. As the angle of the pose increases, it becomes more difficult to find matches between the first image and the second, which has a larger camera angle:

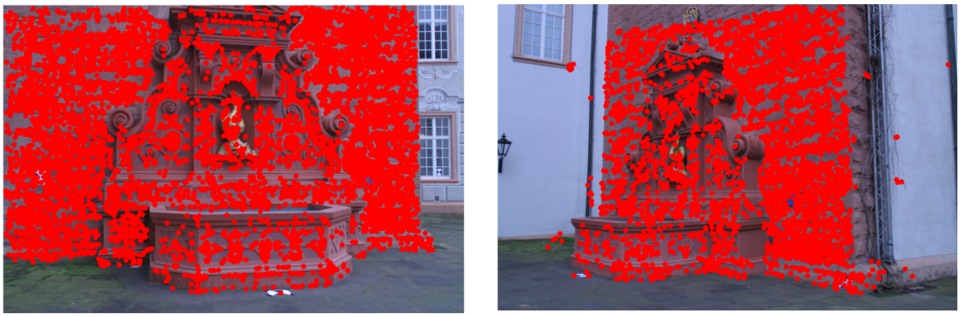

Figure 3: Different Image Poses

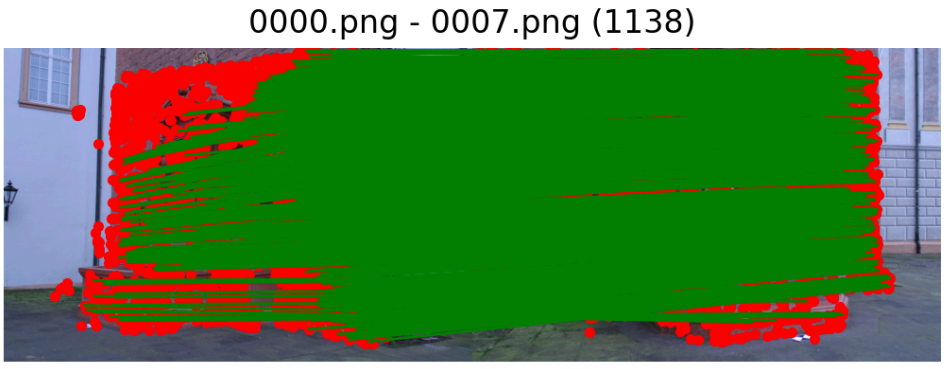

Figure 4: Matches for different camera poses

Line Fitting

The purpose of this exercise is to explore a model fitting technique known as RANSAC (RANdom Sample Consensus). The given line is modeled as y = kx + b with noise added to the points.

Least Square Solution:

The first step is to generate a function that computes the least squares solution for a set of points x and y. This line:

```
A = np.vstack([x, np.ones_like(x)]).T
```

It is used to obtain a matrix representation of the following form:

$$\begin{pmatrix} x_1 & x_2 & \dots & x_n \\ 1 & 1 & \dots & 1 \end{pmatrix}$$

This format is then passed to the solver linalg.lstsq:

```
solution, _, _, _ = np.linalg.lstsq(A, y, rcond=None)
```

Num_inlier:

This function is used to compute the number of inliers and the mask associated with a set of x and y values, along with their corresponding predictions using the modeled line. The following line of code calculates the distance between the correct and predicted y values.

```
distances = np.abs(y - y_pred)
```

If the distance to the prediction is smaller than a threshold, the position is selected as an inlier.

```
mask = distances < thres_dist
num = np.sum(mask)
```

Ransac function:

This function iterates through a number of iterations specified by the variable named “iter”. In each iteration, the least squares error function is applied to a small set of randomly selected x, y points. If the number of inliers is greater than that found in the previous iteration, the k and b variables of the modeled line are updated.

Results

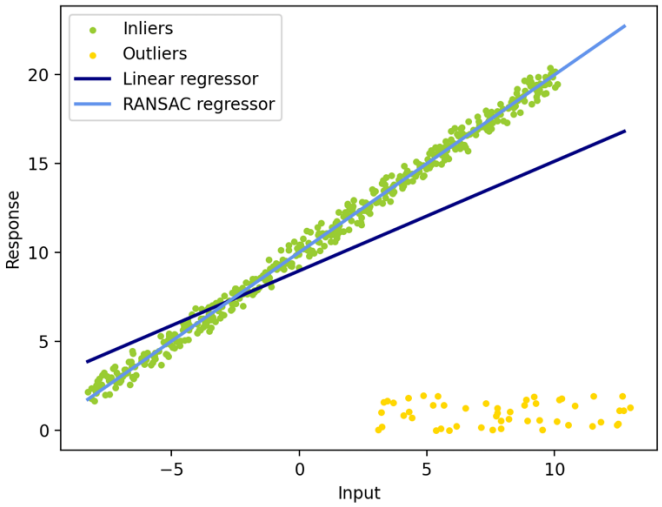

Figure 5: RANSAC regressor

Ground truth	1	10
Linear regression	0.615965657875546	8.961727141443642
RANSAC	0.9987449792570884	9.997009758791

The results for the slope and intercept demonstrate that the RANSAC regressor is more precise for both quantities. Specifically, the slope and intercept values for the RANSAC regressor are almost identical to the ground truth, whereas the linear regressor shows significant differences from the ground truth.

The results show that the RANSAC regressor is more accurate for predicting the line. This is due to the fact that the RANSAC regressor is much more robust to outliers than linear regression. Specifically, the noisy x observations were modified with some outlier values, which are plotted in yellow on the graph. These outlier values are greater than 2.5, and their weight significantly affects the linear regressor after 2.5. Before this threshold, both the linear and RANSAC regressors are close (with RANSAC being more precise). However, after 2.5, the linear regressor deviates significantly from the RANSAC regressor and becomes less accurate. In conclusion, it can be observed that the RANSAC regressor is more accurate than the linear regressor when dealing with very noisy observations, as it is more robust to large outlier values.