Filippo Ficarra: Lab 7 - Structure from Motion and Line Fitting

fficarra@student.ethz.ch, 22–938–062.

22/12/2023 - 17:28h

1 Structure from Motion

1.1 Implementation

The principal functions that had to be implemented were:

- Essential matrix estimation
- Point triangulation
- Map extension

1.1.1 Essential matrix estimation

Listing 1: homogeneous coordinate and normalizing

```
homogeneous_kps1 = np.array([MakeHomogeneous(kp) for kp in im1.kps[matches[:,0]]])
homogeneous_kps2 = np.array([MakeHomogeneous(kp) for kp in im2.kps[matches[:,1]]])

K_inv = np.linalg.inv(K)

normalized_kps1 = (K_inv @ homogeneous_kps1.T).T
normalized_kps2 = (K_inv @ homogeneous_kps2.T).T
```

In this code I went to homogeneous coordinates and then I normalized the points.

Listing 2: constraint matrix

```
constraint_matrix = np.zeros((matches.shape[0], 9))

for i in range(matches.shape[0]):
  for j in range(normalized_kps1[i].shape[0]):
    for k in range(normalized_kps2[i].shape[0]):
      constraint_matrix[i, j * normalized_kps2[i].shape[0] + k] \
                        = normalized_kps2[i,j] * normalized_kps1[i,k]
```

The previous code was used to find the constraint matrix from the formula $x_2^T E x_1 = 0$.

The we need to perform svd on the constraint matrix to find the essential matrix imposing the first 2 singular values to be 1 and the third to be 0.

1.1.2 Point triangulation

Listing 3: point triangulation

```

  homogeneous_points3D = np.array([MakeHomogeneous(p) for p in points3D])
  points3D_cam1 = (P1 @ homogeneous_points3D.T).T
  points3D_cam2 = (P2 @ homogeneous_points3D.T).T

  indices_cam_1 = np.where(points3D_cam1[:,2] > 0)[0]
  indices_cam_2 = np.where(points3D_cam2[:,2] > 0)[0]


  indices = np.intersect1d(indices_cam_1, indices_cam_2)


  return points3D[indices], im1_corrs[indices], im2_corrs[indices]
```

In this code we first compute the 3D points from the cam 1 and 2. We then filter the points for the one that have Z-coordinate < 0, so they are behind the camera.

1.1.3 Map extension

Listing 4: Trinagulate images

```

  offset = 0
  indices = []
  im1_corrs_array = np.array([])

  for registered_image in registered_images:

    m = GetPairMatches(image_name, registered_image, matches)

    points3D_, im1_corrs, im2_corrs = TriangulatePoints(K, image, images[registered_image], m)
    points3D = np.append(points3D, points3D_, 0)
    im1_corrs_array = np.append(im1_corrs_array, im1_corrs, 0)
    indices.extend([i + offset for i in range(points3D_.shape[0])])
    corrs[registered_image] = (im2_corrs, [i + offset for i in range(points3D_.shape[0])])
    offset += points3D_.shape[0]

  corrs[image_name] = (im1_corrs_array, indices)
```

In this code we iterate over all the registered images and we triangulate the points between the current image and the registered one. We then append the points to the points3D array and we save the correspondences between the current image and the registered one.

1.2 Results

The following imges show the result for the algorithm using all the images. Since the last 2 images contain some points far in the background, the cloud of points shows also them.

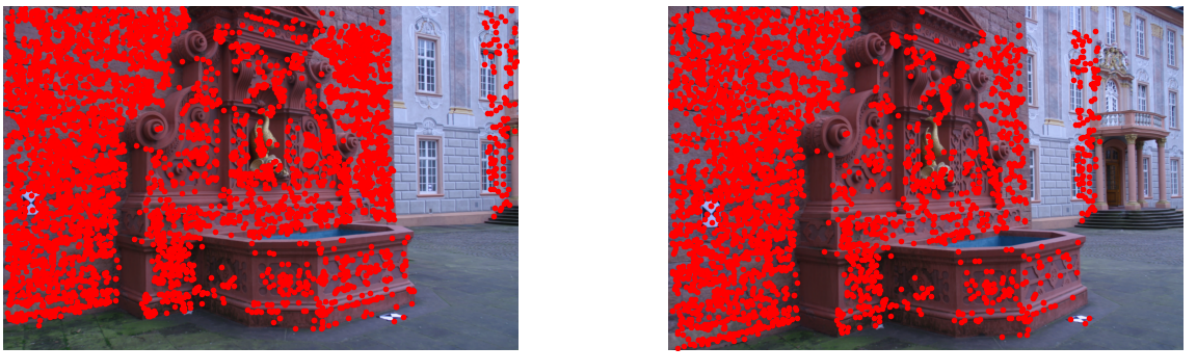

Figure 1: Images 8 and 9 with the points in the background.

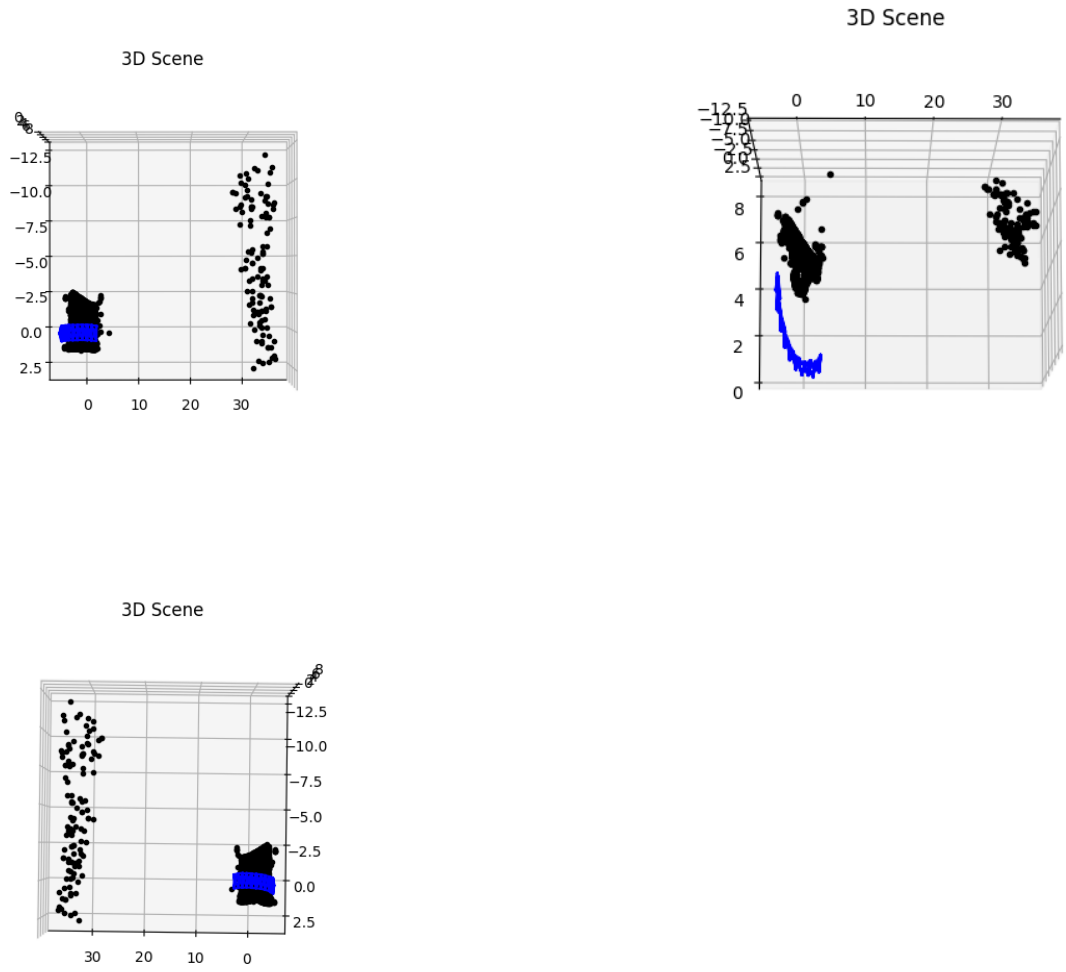

Figure 2: Results for all the images. From top left to bottom right: top-right frontal , top-frontal, back views.

If we remove the last two images we can clearly see the reconstruction of the fountain as shown below. The construction is much more visible, there are less points to be considered when computing and plotting.

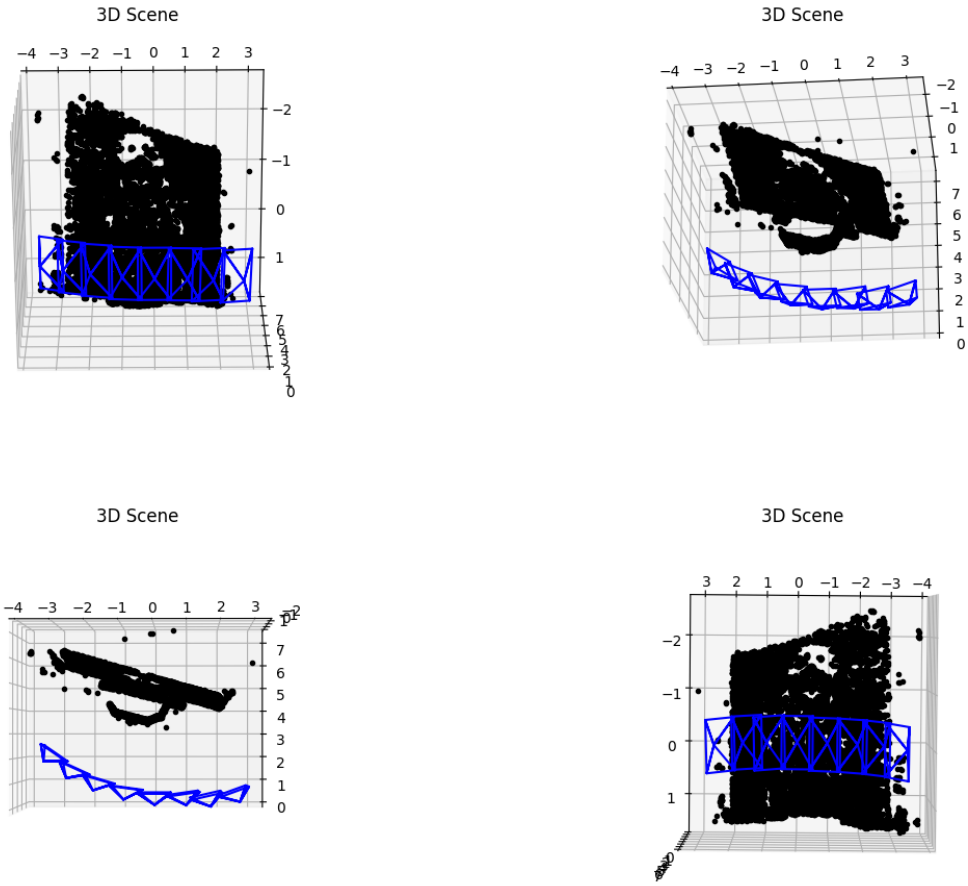

Figure 3: Results for images 0 to 7. From top left to bottom right: frontal, top-frontal, top, back views.

2 Model Fitting

2.1 Implementation

The main function of the algorithm is shown below. The algorithm basically iterate all over the number of iterations, samples a random sample of points and compute the least squares. The least squares coefficient are then passed to a function to compute the number of outliers and their indices.

Listing 5: RANSAC

```
def ransac(x,y,iter,n_samples,thres_dist,num_subset):
  k_ransac = None
  b_ransac = None
  inlier_mask = None
  best_inliers = 0

  for _ in range(iter):
    indices = random.sample(range(n_samples), num_subset)

    k, b = least_square(x[indices], y[indices])

    num, mask = num_inlier(x, y, k, b, n_samples, thres_dist)

    if num > best_inliers:
      best_inliers = num
      k_ransac = k
      b_ransac = b
      inlier_mask = mask


  return k_ransac, b_ransac, inlier_mask
```

The inliers are computed with the following function:

Listing 6: num_inlier

```
def num_inlier(x,y,k,b,n_samples,thres_dist):
  num = 0
  mask = np.zeros(n_samples, dtype=bool)

  # distance point line
  line = k*x + b
  dist = np.abs(line-y) / np.sqrt(k**2 + 1)

  mask = dist < thres_dist

  num = len(dist[mask])

  return num, mask
```

A point is considered to be inlier if the its distance from the line is less then a threshold.

2.2 Results

In this section we are going to show the result for the RANSAC algorithm. The value for the k and be for the 3 methods are the following:

- $k_{gt} = 1,\ b_{gt} = 10$
- $k_{ls} = 0.615965657875546,\ b_{ls} = 8.961727141443642$
- $k_{RANSAC} = 0.9643894946568982,\ b_{RANSAC} = 9.982921294966832$

As shown above, the coefficient for the RANSAC are a very good estimation of the ground truth line, while the least squares is far away from the real line, due to the outliers present in the dataset.

These are the resulting lines that you can see in the figure below:

$$y_{gt} = x + 10$$
$$y_{ls} = 0.615965657875546 * x + 8.961727141443642$$
$$y_{RANSAC} = 0.9643894946568982 * x + 9.982921294966832$$

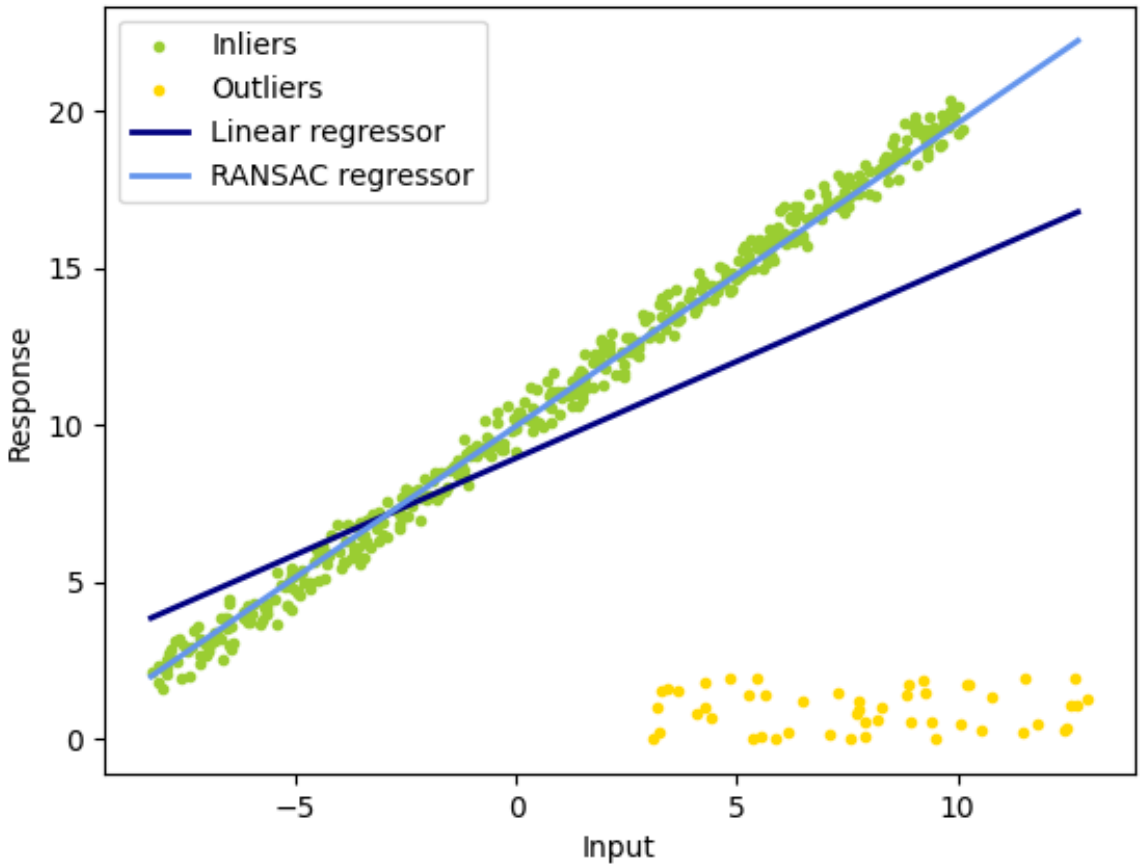

Figure 4: Model fitting results.