Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Vision
and Geometry Lab

ETH Zurich
Computer Science Department

Computer Vision

Lab Assignment - Structure from Motion & Model Fitting

Lara Nonino - lnonino@student.ethz.ch

December 2022

Student ID:
22-937-833

Section 1

Structure form Motion

The first task was to produce a reconstruction of a small scene using *Structure from Motion* (SfM) method.
To initialize the scene, the relative pose between the two images needs to be estimated. Afterwards, it possible to triangulate the first points in the scene and then iteratively register more images and triangulate new points.

1.1 Essential matrix estimation

The first step in implementing the algorithm was completing the `EstimateEssentialMatrix` function in `geometry.py`.
This function receives the K matrix of the camera (which represents the camera intrinsic parameters), two images objects (which also contain the information about the keypoints of the relative image) and the matches between the keypoints of the two images. To estimate the Essential matrix the approach followed was:

1. **Normalizing the coordinates (to points on the normalized image plane)**
   After having found the keypoints on each image and converted them in homogeneous coordinates, the normalization was done for both images as follows ($\hat{x}$ is the normalized coordinate and x is the unnormalized one)

   $$\hat{x} = K^{-1}x$$

   $\hat{x}$ is then divided by its last element so that the third coordinate will be 1.

2. **Writing the contraint matrix**
   Since the E matrix is a 3×3 matrix and the system to solve is

   $$x^T E x = 0$$

   it is possible to use the 8-point algorithm. Indeed, assuming that the number of point correspondences is M, the system to solve for each correspondence $\{x_m, x'_m\}$ can be rewritten as

   $$\begin{bmatrix} x'_m & y'_m & 1 \end{bmatrix} \begin{bmatrix} e_1 & e_2 & e_3 \\ e_4 & e_5 & e_6 \\ e_7 & e_8 & e_9 \end{bmatrix} \begin{bmatrix} x_m \\ y_m \\ 1 \end{bmatrix} = 0$$

   $$x_m x'_m e_1 + x_m y'_m e_2 + x_m e_3 + y_m x'_m e_4 + y_m y'_m e_5 + y_m e_6 + x'_m e_7 + y'_m e_8 + e_9 = 0$$

Therefore, considering each correspondence the homogeneous linear system with 9 unknowns can be built as

$$\begin{bmatrix} x_1x'_1 & x_1y'_1 & x_1 & y_1x'_1 & y_1y'_1 & y_1 & x'_1 & y'_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_Mx'_M & x_My'_M & x_M & y_Mx'_M & y_My'_M & y_M & x'_M & y'_M & 1 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \\ e_9 \end{bmatrix} = 0$$

Where the $(M \times 9)$ matrix is the constraint matrix.

3. **Solve for the nullspace of the constraint matrix**
   The solution for the previous system can be found through singular value decomposition (SVD) which provides the E matrix flattened. Therefore it is necessary to reshape it to a 3×3 matrix.

4. **Fulfilling the internal constraints of E**
   As indicated in the code, the first two singular values need to be equal, the third one zero. Since E is up to scale, two equal singluar values can be chosen arbitrarily. Therefore it is possible to force the scalar matrix of the SVD decomposition of E to be diag([1,1,0]).

```
def EstimateEssentialMatrix(K, im1, im2, matches):
  # Normalize coordinates (to points on the normalized image plane)

  # homogeneous coordinates (with additional dimension)
  h_kps1 = np.append(im1.kps, np.ones((im1.kps.shape[0], 1)), 1)
  h_kps2 = np.append(im2.kps, np.ones((im2.kps.shape[0], 1)), 1)

  K_1 = np.linalg.inv(K)

  normalized_kps1 = np.matmul(K_1, h_kps1.T).T / np.matmul(K_1, h_kps1.T).T[:, -1, None]
  normalized_kps2 = np.matmul(K_1, h_kps2.T).T / np.matmul(K_1, h_kps2.T).T[:, -1, None]

  # Assemble constraint matrix as equation 2.1
  constraint_matrix = np.zeros((matches.shape[0], 9))
  for i in range(matches.shape[0]):
    # Add the constraints
    constraint_matrix[i] = np.reshape(np.outer(normalized_kps2[matches[i, 1]],\
                                               normalized_kps1[matches[i, 0]]), 9)

  # Solve for the nullspace of the constraint matrix
  _, _, vh = np.linalg.svd(constraint_matrix)
  vectorized_E_hat = vh[-1, :]

  # Reshape the vectorized matrix to it's proper shape again
  E_hat = np.reshape(vectorized_E_hat, (3, 3))

  # We need to fulfill the internal constraints of E
  # The first two singular values need to be equal, the third one zero.
  # Since E is up to scale, we can choose the two equal singluar values arbitrarily
  u, _, vh = np.linalg.svd(E_hat)
  s = np.diag([1, 1, 0])
  E = np.matmul(np.matmul(u, s), vh)

  # This is just a quick test that should tell you if your estimated matrix is not correct
  # It might fail if you estimated E in the other direction (i.e. kp2' * E * kp1)
  # You can adapt it to your assumptions.
```

```
for i in range(matches.shape[0]):
  kp1 = normalized_kps1[matches[i, 0], :]
  kp2 = normalized_kps2[matches[i, 1], :]

  #assert(abs(kp1.transpose() @ E @ kp2) < 0.01)
  assert (abs(kp2.transpose() @ E @ kp1) < 0.01)

return E
```

1.2 Point triangulation

The DLT for point triangulation was already implemented in the framework (`TriangulatePoints` in `geometry.py`).
Since some points found might lie behind one or both cameras, it was necessary to filter these points out. To do so, the 3D point coordinates were firstly converted in homogeneous coordinates. The points obtained are then projected to the two cameras. This was done by multiplying the coordinates of the 3D point with the projective matrix of each camera. If the last coordinate of the point obtained is grater than 0 for both camera, then the point is kept, otherwise it is filtered out.

```
# Filter points behind the cameras by transforming them into each camera space and
# checking the depth (Z)
# Make sure to also remove the corresponding rows in `im1_corrs` and `im2_corrs`

# Filter points behind the first camera
h_points3D = np.append(points3D, np.ones((points3D.shape[0], 1)), 1)
cam1 = np.matmul(h_points3D, P1.T)
im1_corrs = im1_corrs[cam1[:, -1] > 0]
im2_corrs = im2_corrs[cam1[:, -1] > 0]
points3D = points3D[cam1[:, -1] > 0]

# Filter points behind the second camera
h_points3D = np.append(points3D, np.ones((points3D.shape[0], 1)), 1)
cam2 = np.matmul(h_points3D, P2.T)
im1_corrs = im1_corrs[cam2[:, -1] > 0]
im2_corrs = im2_corrs[cam2[:, -1] > 0]
points3D = points3D[cam2[:, -1] > 0]

return points3D, im1_corrs, im2_corrs
```

1.3 Finding the correct decomposition

The decomposition of the essential matrix into a relative pose is already implemented (`DecomposeEssentialMatrix` in `geometry.py`). However, this gives four different poses that all fulfill the requirements of the essential matrix. To find the correct one, points were triangulated with each one and the one with the most points in front of both cameras was picked.
To do so, the algorithm implemented was:

1. The canonical coordinates were given to the first camera as absolute coordinates, while the second camera was set to have the rotation and translation of the four different poses.

$$P_1 = [I|0] \qquad\qquad P_2 = [R|t]$$

2. Once the triangulation is done for all the four different poses, the pose that produces the highest number of points (already filtered to be in front of both cameras) is picked as the correct pose for the second camera.

```
# ------------------Finding the correct decomposition-----------------------------------
# For each possible relative pose, try to triangulate points with function TriangulatePoints.
# We can assume that the correct solution is the one that gives the most points in front \
# of both cameras.
max_points = 0
best_pose = -1
# Be careful not to set the transformation in the wrong direction
# you can set the image poses in the images (image.SetPose(...))
# Note that this pose is assumed to be the transformation from global space to image space
e_im1.SetPose(np.identity(3), np.zeros(3))

for i_pose in possible_relative_poses:
  e_im2.SetPose(i_pose[0], i_pose[1])
  points3D, _, _ = TriangulatePoints(K, e_im1, e_im2, e_matches)
  if points3D.shape[0] > max_points:
    max_points = points3D.shape[0]
    best_pose = i_pose

# Set the image poses in the images (image.SetPose(...))
# Note that the pose is assumed to be the transformation from global space to image space
e_im1.SetPose(np.identity(3), np.zeros(3))
e_im2.SetPose(best_pose[0], best_pose[1])

# Triangulate initial points
points3D, im1_corrs, im2_corrs = TriangulatePoints(K, e_im1, e_im2, e_matches)

# Add the new 2D-3D correspondences to the images
e_im1.Add3DCorrs(im1_corrs, list(range(points3D.shape[0])))
e_im2.Add3DCorrs(im2_corrs, list(range(points3D.shape[0])))

# Keep track of all registered images
registered_images = [e_im1_name, e_im2_name]

for reg_im in registered_images:
  print(f'Image {reg_im} sees {images[reg_im].NumObserved()} 3D points')
```

1.4 Absolute pose estimation

Given a point cloud and correspondences to the image keypoints `EstimateImagePose` in `geometry.py` estimates the absolute pose of a new image with respect to the scene.

```
def EstimateImagePose(points2D, points3D, K):

# We use points in the normalized image plane.
# This removes the 'K' factor from the proj

h_points2D = np.append(points2D, np.ones((points2D.shape[0], 1)), 1)
K_1 = np.linalg.inv(K)
normalized_points2D = (np.matmul(K_1, h_points2D.T).T / \
                       np.matmul(K_1, h_points2D.T).T[:, -1, None])[:, :-1]

constraint_matrix = BuildProjectionConstraintMatrix(normalized_points2D, points3D)

_, _, vh = np.linalg.svd(constraint_matrix)
P_vec = vh[-1,:]
P = np.reshape(P_vec, (3, 4), order='C')

# Make sure we have a proper rotation
u, s, vh = np.linalg.svd(P[:, :3])
R = u @ vh

if np.linalg.det(R) < 0:
  R *= -1
```

```
  _, _, vh = np.linalg.svd(P)
  C = np.copy(vh[-1,:])

  t = -R @ (C[:3] / C[3])

  return R, t
```

1.5 Map extension

`TriangulateImage` of `geometry.py` extends the map by triangulating new points from all possible pairs of registered images. The function receives a reference image set and a new image as inputs. In the for loop, for each registered image, the new keypoints obtained from the new image are combined with the registered keypoints. In this way new triagulated 3D points can be found. At the end, the 2D-3D correspondences for the registered images and the new image are saved in a dictionary.

```
def TriangulateImage(K, image_name, images, registered_images, matches):

  # Loop over all registered images and triangulate new points with the new image.
  # Make sure to keep track of all new 2D-3D correspondences, also for the registered images

  image = images[image_name]
  points3D = np.zeros((0, 3))

  # You can save the correspondences for each image in a dict and refer to the `local`\
  # new point indices here.
  # Afterwards you just add the index offset before adding the correspondences to\
  # the images.
  corrs_image = np.zeros(0)
  corrs = {}

  for reg_image_name in registered_images:
    e_matches = GetPairMatches(reg_image_name, image_name, matches)
    new_points3D, corrs_new_reg_image, corrs_new_image =\
                                  TriangulatePoints(K, images[reg_image_name],image, e_matches)
    corrs[reg_image_name] = [corrs_new_reg_image,\
                    np.arange(points3D.shape[0], points3D.shape[0]+new_points3D.shape[0])]
    points3D = np.append(points3D, new_points3D, axis=0)
    corrs_image = np.append(corrs_image, corrs_new_image, axis=0)

  corrs[image_name] = [corrs_image, np.arange(points3D.shape[0])]

  return points3D, corrs
```

`UpdateReconstructionState` of `corrs.py` updates the reconstruction adding the new points to the set of reconstruction points and adding the correspondences to the images.

```
# Update the reconstruction with the new information from a triangulated image
def UpdateReconstructionState(new_points3D, corrs, points3D, images):

  # Add the new points to the set of reconstruction points and add the correspondences to the images.
  # Be careful to update the point indices to the global indices in the `points3D` array.
  offset = points3D.shape[0]
  points3D = np.append(points3D, new_points3D, 0)

  for im_name in corrs:
    images[im_name].Add3DCorrs(corrs[im_name][0], offset + corrs[im_name][1])

  return points3D, images
```

1.6 Result

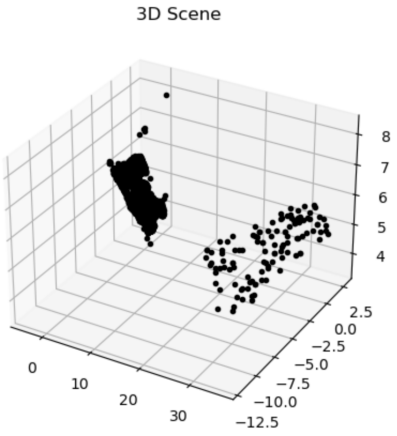


Figure 1.1: 3D points using all the images

The cloud of points on the right is determined by the 3D visible through camera 8 and 9, at the right end of both images (the plots follow).

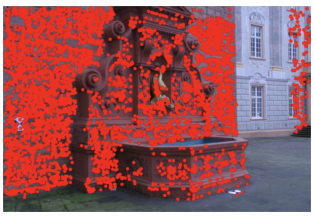


Figure 1.2: image 0008.png

Figure 1.3: image 0009.png

Excluding these images leads to a reconstruction without those points.

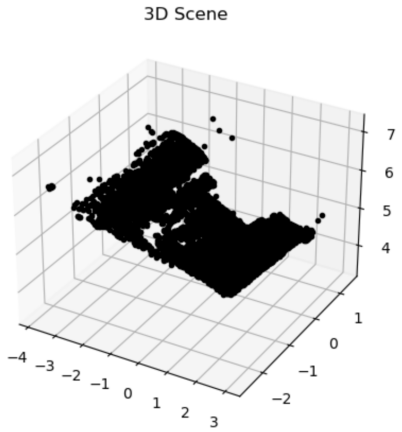


Figure 1.4: 3D points using images from 0 to 7

Section 2

Model Fitting

In this assignment, we were asked to implement RANSAC (RANdom SAmple Consensus) for robust model fitting on the simple case of 2D line estimation.

2.1 Least-squares Solution

The first task was to implement the function `least_square` in line_fitting.py. This should compute the coefficients k, b of the least-square solution line which has the form

$$y = kx + b$$

Using `np.linalg.lstsq`, the implementation is

```
def least_square(x,y):
        # return the least-squares solution
        # you can use np.linalg.lstsq
        A = np.vstack([x, np.ones(len(x))]).T
        k, b = np.linalg.lstsq(A, y, rcond=None)[0]
        return k, b
```

2.2 Least-squares Solution

The function `ransac` in `line_fitting.py` (loops for *iter* iterations) works as follows:

1. randomly choose a small subset, with *num_subset* elements, from the noisy point set
2. compute the least-squares solution for this subset
3. compute the number of inliers and the mask denotes the indices of inliers
4. if the number of inliers is larger than the current best result, update the parameters *k_ransac*, *b_ransac* and also the *inlier_mask*

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

        for _ in range(iter):
                rng = np.random.default_rng()
```

```
                random_index = (n_samples * rng.random(num_subset)).astype(int)
                random_x = x[random_index]
                random_y = y[random_index]

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

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

        return k_ransac, b_ransac, inlier_mask
```

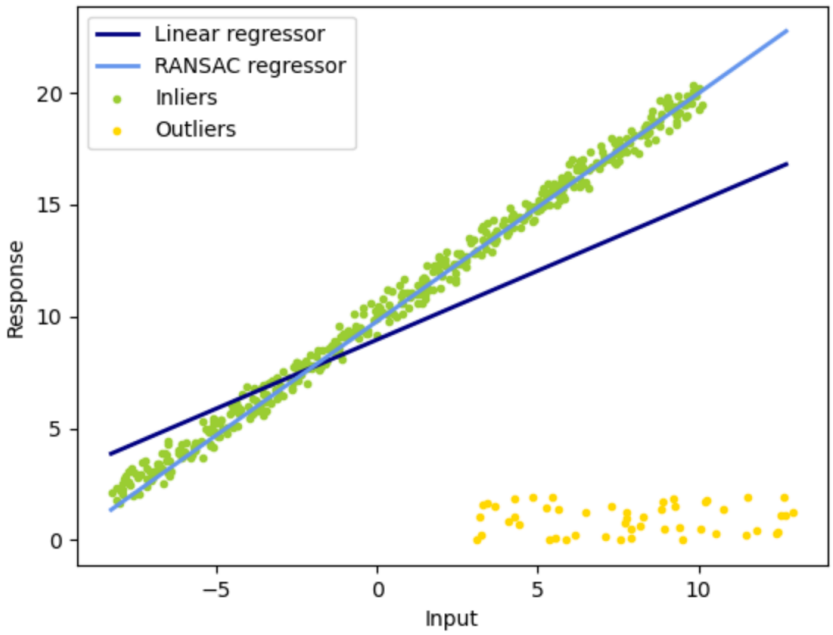
The function `num_inlier` in `line_fitting.py` computes the distance between each point received as input and the line whose parameters are k and b. Afterwards it filters the points whose distance is lower than *thresh_dist* through a mask and counts them.

```
def num_inlier(x,y,k,b,n_samples,thres_dist):
        # compute the number of inliers and a mask that denotes the indices of inliers
        num = 0
        mask = np.zeros(x.shape, dtype=bool)

        if k == 0:
                dist = y - b
        else:
                x_1 = (y + (x/k) - b) / (k + 1/k)
                y_1 = k * x_1 + b
                dist = np.sqrt((x_1 - x)**2 + (y_1 - y)**2)

        mask = dist < thres_dist
        num = np.count_nonzero(mask)

        return num, mask
```

2.3 Results

Estimated coefficients					
True		Linear Regression		RANSAC	
k	b	k	b	k	b
1	10	0.6159656578755456	8.96172714144364	1.0186999322757846	9.786667808497462

As the plots shows, using the RANSAC algorithm, the regressor in not disturbed by the outliers, therefore the estimation of the coefficients is more accurate.


Linear regressor
RANSAC regressor
Inliers
Outliers
Response
Input
20
15
10
5
0
−5
0
5
10