Assignment 7 - Structure from motion & model fitting

Federica Bruni (fbruni@student.ethz.ch) 18 December 2023

1 Overview

This computer vision lab report focuses on implementing Structure from Motion (SfM) methods for reconstructing a small scene. The employed approach combines relative pose estimation, absolute pose estimation, and point triangulation, forming a streamlined SfM pipeline.Keypoints and feature matches are pre-verified for geometric accuracy, eliminating outliers from the pipeline.The process begins by estimating relative poses between two images, followed by iterative image registration and triangulation of new points to refine the scene reconstruction.

2 Structure from Motion

The first task focuses on implementing Structure from Motion (SfM) methods for reconstructing a small scene. The employed approach combines relative pose estimation, absolute pose estimation, and point triangulation, forming a streamlined SfM pipeline.Keypoints and feature matches are pre-verified for geometric accuracy, eliminating outliers from the pipeline.The process begins by estimating relative poses between two images, followed by iterative image registration and triangulation of new points to refine the scene reconstruction.

2.1 Methodology

1. **Essential matrix estimation:** In this step, the intrinsics matrix K is used to calculate the relative pose between the initial images using the essential matrix E. The Direct Linear Transform (DLT) is applied to reformulate the constraint $\hat{x}_1^T E \hat{x}_2 = 0$. Keypoints are lifted to the normalized image plane $\hat{x} = K^{-1}x$ and it is important that the estimated essential matrix follows specific constraints, ensuring the equality and positivity of the first two singular values and a third singular value of 0.

```
def EstimateEssentialMatrix(K, im1, im2, matches):
  # TODO
  # Normalize coordinates (to points on the normalized image plane)
  # Normalize keypoints for image 1
  homogeneous_kps1 = MakeHomogeneous(im1.kps, 1)
  normalized_kps1 = np.dot(homogeneous_kps1, np.linalg.inv(K).T)

  # Normalize keypoints for image 2
  homogeneous_kps2 = MakeHomogeneous(im2.kps, 1)
  normalized_kps2 = np.dot(homogeneous_kps2, np.linalg.inv(K).T)


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

    # Get matched keypoints
    match_indices = matches[i]
    x1 = normalized_kps2[match_indices[1], :]
    x2 = normalized_kps1[match_indices[0], :]

    # Build constraint matrix
    constraint_matrix[i, :] = [x1[0] * x2[0], x1[0] * x2[1],x1[0], x1[1] * x2[0],x1
    [1] * x2[1],x1[1], x2[0],x2[1],1]


  # Solve for the nullspace of the constraint matrix
  _, _, vh = np.linalg.svd(constraint_matrix)
```

```
  vectorized_E_hat = vh[-1,:]

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

  # TODO
  # 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, _, Vt = np.linalg.svd(E_hat)
  E = U @ np.diag( [1, 1, 0]) @ Vt

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

  return E
```

2. **Point Triangulation:** In point triangulation, the framework's existing Direct Linear Transform (DLT) for this purpose is utilized. A crucial addition is the implementation of a filtering step within the point triangulation process to exclude points that may inaccurately lie behind one or both cameras.

```
def TriangulatePoints(K, im1, im2, matches):

  R1, t1 = im1.Pose()
  R2, t2 = im2.Pose()
  P1 = K @ np.append(R1, np.expand_dims(t1, 1), 1)
  P2 = K @ np.append(R2, np.expand_dims(t2, 1), 1)

  # Ignore matches that already have a triangulated point
  new_matches = np.zeros((0, 2), dtype=int)

  num_matches = matches.shape[0]
  for i in range(num_matches):
    p3d_idx1 = im1.GetPoint3DIdx(matches[i, 0])
    p3d_idx2 = im2.GetPoint3DIdx(matches[i, 1])
    if p3d_idx1 == -1 and p3d_idx2 == -1:
      new_matches = np.append(new_matches, matches[[i]], 0)


  num_new_matches = new_matches.shape[0]

  points3D = np.zeros((num_new_matches, 3))

  for i in range(num_new_matches):

    kp1 = im1.kps[new_matches[i, 0], :]
    kp2 = im2.kps[new_matches[i, 1], :]

    # H & Z Sec. 12.2
    A = np.array([
      kp1[0] * P1[2] - P1[0],
      kp1[1] * P1[2] - P1[1],
      kp2[0] * P2[2] - P2[0],
      kp2[1] * P2[2] - P2[1]
    ])

    _, _, vh = np.linalg.svd(A)
    homogeneous_point = vh[-1]
    points3D[i] = homogeneous_point[:-1] / homogeneous_point[-1]


  # We need to keep track of the correspondences between image points and 3D points
  im1_corrs = new_matches[:,0]
  im2_corrs = new_matches[:,1]
```

```
    # TODO
    # 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`
    # Transform points to homogeneous coordinates for the first camera
    homogeneous_points1 = MakeHomogeneous(points3D, 1)
    cam1_space = homogeneous_points1 @ P1.T

    # Filter points behind the first camera
    mask1 = cam1_space[:, -1] > 0
    im1_corrs = im1_corrs[mask1]
    im2_corrs = im2_corrs[mask1]
    points3D = points3D[mask1]

    # Transform points to homogeneous coordinates for the second camera
    homogeneous_points2 = MakeHomogeneous(points3D, 1)
    cam2_space = homogeneous_points2 @ P2.T

    # Filter points behind the second camera
    mask2 = cam2_space[:, -1] > 0
    im1_corrs = im1_corrs[mask2]
    im2_corrs = im2_corrs[mask2]
    points3D = points3D[mask2]

    return points3D, im1_corrs, im2_corrs
```

3. **Finding the Correct Decomposition:** Following essential matrix decomposition into a relative pose using the DecomposeEssentialMatrix function, a challenge arises due to the existence of multiple poses that fulfill essential matrix requirements. To identify the correct pose, a strategy involves triangulating points with each pose and selecting the one with the maximum points visible in front of both cameras. To facilitate this, one camera pose is set to identity, and care is taken in handling the transformation direction.

```
# ------------------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
    # TODO
    e_im1.SetPose(np.identity(3), np.zeros_like(possible_relative_poses[0][1]))

    # Initialize variables for the best pose
    best_pose = (e_im1.Pose(), None)
    max_points = 0

    # Loop through possible relative poses
    for relative_pose, translation in possible_relative_poses:
        # Set pose for e_im2
        e_im2.SetPose(relative_pose, translation)

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

        # Check if the current triangulation yields more points
        if points3D.shape[0] > max_points:
            max_points = points3D.shape[0]
            best_pose = (e_im1.Pose(), e_im2.Pose())
    # TODO
    # 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(best_pose[0][0], best_pose[0][1])
    e_im2.SetPose(best_pose[1][0], best_pose[1][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')
```

4. **Absolute Pose Estimation:** Absolute pose estimation involves determining the pose of a new image with respect to the scene, utilizing a given point cloud and correspondences to image keypoints.

```
def EstimateImagePose(points2D, points3D, K):
  # TODO
  # We use points in the normalized image plane.
  # This removes the 'K' factor from the projection matrix.
  # We don't normalize the 3D points here to keep the code simpler.
  normalized_points2D = np.dot(MakeHomogeneous(points2D, 1), np.linalg.inv(K).T)

  constraint_matrix = BuildProjectionConstraintMatrix(normalized_points2D, points3D)

  # We don't use optimization here since we would need to make sure to only optimize
     on the se(3) manifold
  # (the manifold of proper 3D poses). This is a bit too complicated right now.
  # Just DLT should give good enough results for this dataset.

  # Solve for the nullspace
  _, _, 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
```

5. **Map Extension:** Upon knowing the new image's pose, the map is extended by triangulating new points from all possible pairs of registered images. It is necessary to ensure the correct addition of new 2D-3D correspondences to each image.

```
def TriangulateImage(K, image_name, images, registered_images, matches):
  # TODO
  # 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_name: {'points2D': []}}
  start = 0
  for r in registered_images:
    img = images[r]
    if r < image_name:
      rpoints3D, r_corrs, img_corrs = TriangulatePoints(K, img, image, matches[(r,
    image_name)])
    else:
      rpoints3D, img_corrs, r_corrs = TriangulatePoints(K, image, img, matches[(
    image_name, r)])
```

```
    points3D = np.append(points3D, rpoints3D, 0)
    end = points3D.shape[0]
    corrs[r] = {'points2D': r_corrs, 'range': (start, end)}
    corrs[image_name]['points2D'].extend(img_corrs)
    start = end

  corrs[image_name]['range'] = 0, points3D.shape[0]

  return points3D, corrs

def UpdateReconstructionState(new_points3D, corrs, points3D, images):

  # TODO
  # 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.
  start_idx = points3D.shape[0]
  points3D = np.append(points3D, new_points3D, 0)

  for im_name in corrs:
    itvl_start, itvl_end = corrs[im_name]['range']
    images[im_name].Add3DCorrs(corrs[im_name]['points2D'], list(range(itvl_start +
    start_idx, itvl_end + start_idx)))

  return points3D, images
```

2.2 Data and Final Results

In the following paragraph, I will present the data, followed by the results obtained from the above-implemented code and this information.

2.2.1 Data

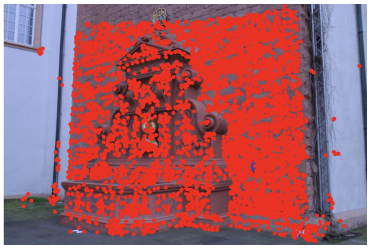
The provided data consists of 9 images of a fountain, captured from various angles. Below are some examples. From the images, it can be observed that the photos are taken by rotating around the fountain.

(a) Image 0

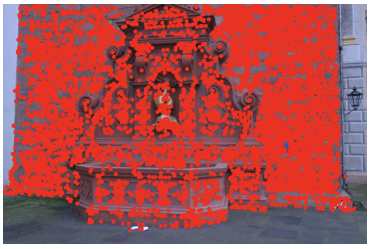
(b) Image 5

(c) Image 9

The preprocessing involves identifying keypoints.Below are several examples of images displaying these keypoints. As can be observed,many keypoints are identified, aided by the intricate details present in the fountain. So, I expect a detailed model to come out of this.



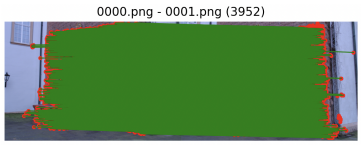
(a) Image 0



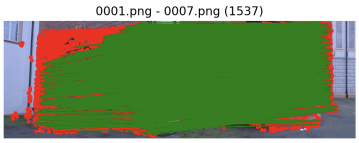
(b) Image 5

(c) Image 9

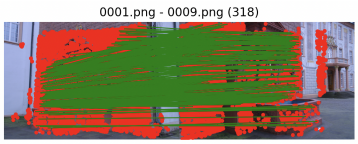
Finally, the last step involves matching keypoints between images. Hence, we observe that before applying the Structure from Motion method, keypoints are correctly identified and matched across the images.



(a) matches between Img1-Img2



(b) matches between Img1-Img7

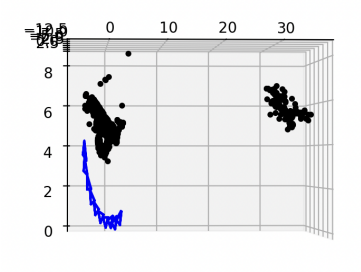


(c) matches between Img1-Img9

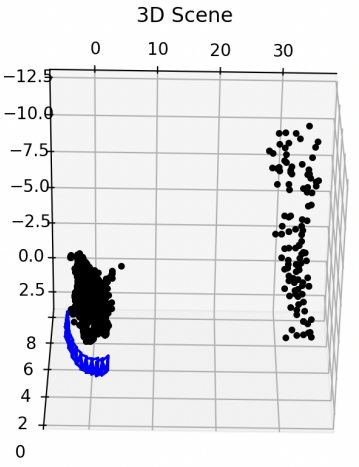
It is interesting to note that the farther apart the images are in terms of their number and consequently, their position when taken, the fewer matches are found between keypoints. This is because the angle of the photos varies significantly.

2.2.2 Results

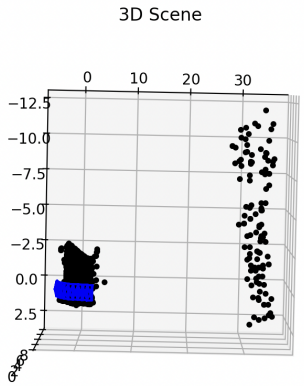
The obtained result is illustrated in the following images.



(a) View from above



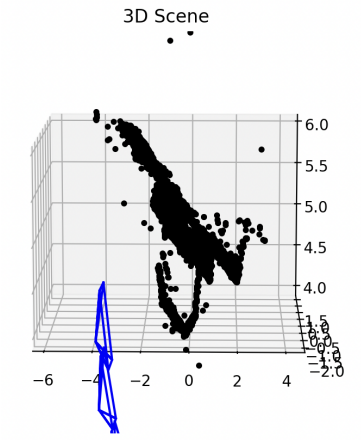
(b) Frontal view



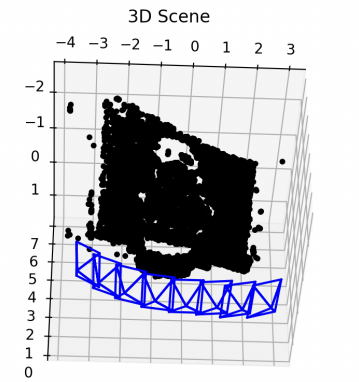
(c) View from the side

At first glance, it might not seem like a good outcome, but this is simply caused by some points from the building next to the fountain, making it challenging to understand the plot.

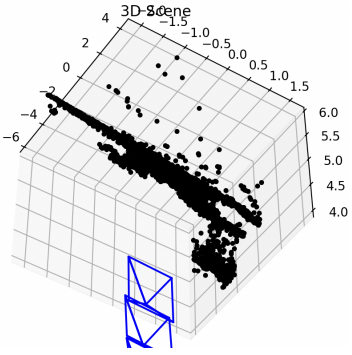
First, let's analyze the result by focusing on the points of the fountain, and then we'll understand the reason behind these other points.



(a) View from above



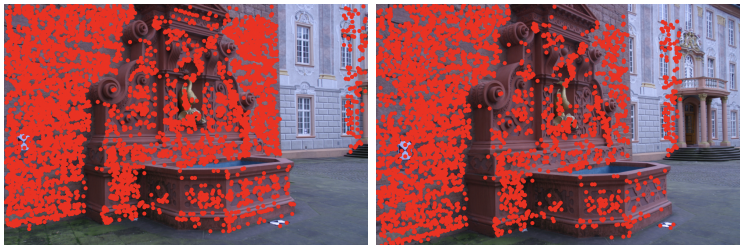
(b) Frontal view



(c) View from the side

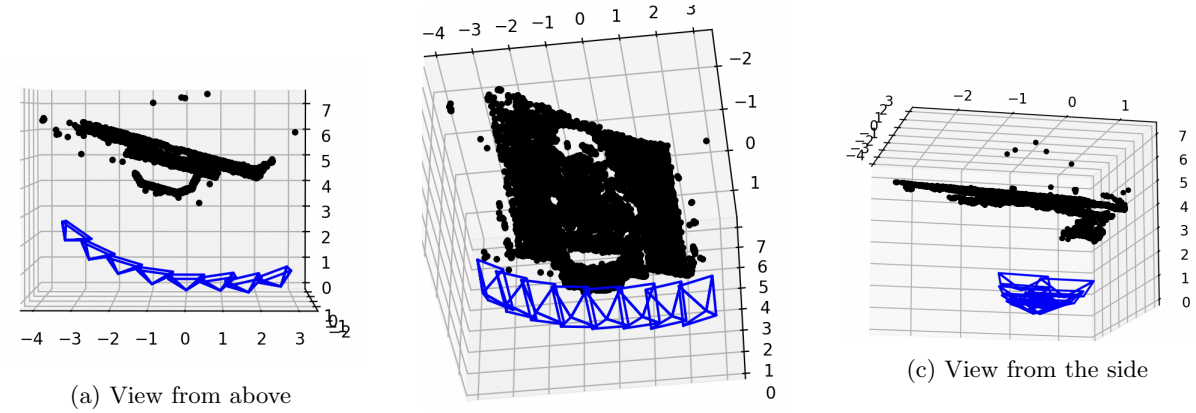
If we concentrate only on the fountain points, the result is good. Each view of the fountain yields a satisfactory model of the fountain itself. This is particularly impressive given the complexity of the fountain; we can clearly identify distinctive elements.

Regarding the other points, these can be found in images 8 and 9, and they are keypoints located in the building behind the fountain, as can be seen in the following images.



(a) Image 8 (b) Image 9

In fact, by excluding these two from the algorithm, the result obtained is much better. In particular the fountain's geometry is is preserved, as evident from the shape of the basin. Some images are provided below for reference.



(a) View from above

(b) Frontal view

(c) View from the side

3 Model Fitting

For this task, the goal is to implement an application of RANSAC (Random Sample Consensus) for robust model fitting. Utilizing a ground truth linear model represented by $y = kx + b$, we create a set of points (x noisy, y noisy) based on the linear model and introduce noise to the data. Additionally, outliers are intentionally incorporated into the point set.

3.1 Methodology

1. **Least squares:** The initial step in this process involved finding a solution for the least-squares problem.The objective was to find the best-fitting linear model $y = kx + b$ for a given set of input data points (x, y).This was achieved by formulating a system of equations, represented as a matrix, to obtain the coefficients k and b that minimize the squared differences between the predicted and actual y values.

```
def least_square(x,y):
  # TODO
  # 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. **RANSAC:** The next task involves implementing RANSAC. The num_inlier function computes the number of inliers and generates a mask indicating the indices of inliers. It calculates the distances between the observed y values and the corresponding values predicted by the current model (defined by k and b).

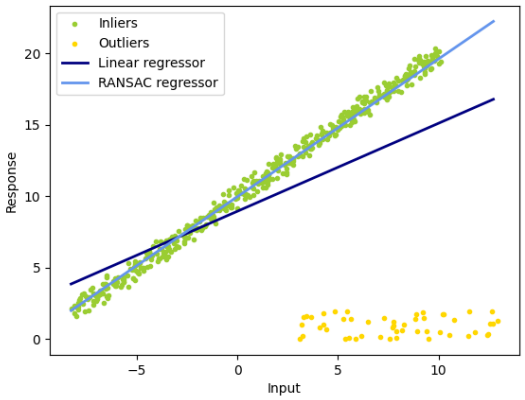
   The ransac function iteratively applies the RANSAC algorithm for robust model fitting. In each iteration, a random subset is chosen from the input data, and the least-squares solution is computed. The number of inliers and the corresponding mask are then determined using the num_inlier function. If the number of inliers exceeds the current best result, the model parameters and inlier mask are updated. The final result consists of the parameters k_ransac and b_ransac, as well as the inlier mask.

```
def num_inlier(x,y,k,b,n_samples,thres_dist):
  # TODO
  # compute the number of inliers and a mask that denotes the indices of inliers
  distances = np.abs(y - (k * x + b)) / np.sqrt(k**2 + 1)
  mask = distances < thres_dist
  num = np.sum(mask)
  return num, mask

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

  for _ in range(iter):
    subset_indices = random.sample(range(len(x)), num_subset)
    subset_x = x[subset_indices]
    subset_y = y[subset_indices]
    k_subset, b_subset = least_square(subset_x, subset_y)
    num_inliers_subset, inlier_mask_subset = num_inlier(x, y, k_subset, b_subset,
    n_samples, thres_dist)
    if num_inliers_subset > best_inliers:
      best_inliers = num_inliers_subset
      k_ransac, b_ransac, inlier_mask = k_subset, b_subset, inlier_mask_subset
  return k_ransac, b_ransac, inlier_mask
```

.

3.2 Result

A visual representation of the reported RANSAC implementation is displayed in the following plot. Unlike the standard linear regressor, which is sensitive to outliers and tends to produce a line that deviates from the true relationship within the dataset, the RANSAC regressor tackles the challenges posed by outliers. By automatically categorizing the data into inliers and outliers, the resulting fitted line is exclusively determined by the identified inliers. This method ensures a more precise representation of the authentic data distribution, highlighting RANSAC's capability to construct a robust linear model even in the presence of disruptive data points. Finally as requested the ground truth, estimation from least-squares



(a) Robust linear model fitting using RANSAC

and estimation from RANSAC:

```
'Estimated coefficients:'
# ground truth
1 10
# least-squares
0.615965657875546 8.961727141443642
# RANSAC
0.9643894946568982 9.982921294966832
```