Computer Vision PA #1 Report

20195159 Wooseok Jeon

Abstract

This programming assignment #2 focuses on Structure-from-Motion (SFM) using multi-view images. SFM is a digital image processing technique for reconstructing 3D structures, which relies on images taken from various angles to recreate a 3D environment. The primary goal of this assignment is to accurately extract 3D structures from multiple images. To achieve this, the relative positions and orientations of the images are considered for matching feature points. Then, using these matched feature points, the camera pose for each image is estimated, and ultimately, this information is used to reconstruct 3D points. This process depends on calibrated camera settings and precise methods for extracting and matching feature points. In addition to the initial reconstruction, this assignment also involves the critical step of bundle adjustment in the SFM process. Bundle adjustment is an optimization technique used to refine the 3D structure and camera poses simultaneously. It adjusts the 3D point positions and camera parameters to minimize the re-projection error, which is the discrepancy between the observed image points and the projected points from the 3D model. This refinement process is essential for improving the accuracy of the final 3D model. It ensures that the reconstructed structure aligns closely with the actual scene geometry as captured in the multi-view images.

1. Method & Code Implementation

1.1. Input data

In PA2, the following data sets were provided for the Structure-from-Motion (SFM) process:

1. 3D Points Reconstructed:
   These are the 3D points that have been reconstructed from two images, namely 'sfm03.jpg' and 'sfm04.jpg'. These points form the foundation of the SFM process, representing the initial reconstruction of the 3D structure from multiple views.
2. 2D Keypoints Corresponding to 3D Points:
   For each image, a set of 2D keypoints corresponding to the reconstructed 3D points was provided. These keypoints play a crucial role in establishing the correspondence between the 2D images and the 3D world.
3. Camera Poses of Two Images:
   The camera poses for these two images are given. These poses include information about the position and orientation of the camera when each image was taken, which is essential for accurate 3D reconstruction.
4. Intrinsic Matrix:
   The intrinsic matrix of the camera is provided. This matrix contains information about the camera's internal characteristics, such as focal length and optical center, which are crucial for projecting 3D points onto 2D image planes.
5. Remaining Images:
   Besides two images, additional images are provided. These images can be used for further extending the 3D reconstruction to include more viewpoints, thereby enhancing the depth and accuracy of the 3D model.

1.2. Step 0: Preprocessing

Before diving into the main Structure-from-Motion (SFM) process, a preprocessing step is necessary. Below is the code for mentioned step.

```python
# Preprocess using indexing
sfm03_sorted_keypoints = []
sfm03_sorted_descriptors = []
for idx in sfm03_matched_idx:
    sfm03_sorted_keypoints.append(sfm03_keypoints[idx])
    sfm03_sorted_descriptors.append(sfm03_descriptors[idx])

sfm04_sorted_keypoints = []
sfm04_sorted_descriptors = []
for idx in sfm04_matched_idx:
    sfm04_sorted_keypoints.append(sfm04_keypoints[idx])
    sfm04_sorted_descriptors.append(sfm04_descriptors[idx])

sfm03_index_keypoints = []
sfm03_index_descriptors = []
sfm03_matched_idx_3D = []
sfm04_index_keypoints = []
sfm04_index_descriptors = []
sfm04_matched_idx_3D = []

for value in inlinear:
    sfm03_index_keypoints.append(sfm03_sorted_keypoints[value])
    sfm03_index_descriptors.append(sfm03_sorted_descriptors[value])
    sfm03_matched_idx_3D.append(sfm03_matched_idx[value])
    sfm04_index_keypoints.append(sfm04_sorted_keypoints[value])
    sfm04_index_descriptors.append(sfm04_sorted_descriptors[value])
    sfm04_matched_idx_3D.append(sfm04_matched_idx[value])

base_sfm03_keypoints = np.array(sfm03_index_keypoints)
base_sfm03_descriptors = np.array(sfm03_index_descriptors)
base_sfm03_matched_idx = np.array(sfm03_matched_idx_3D)
base_sfm04_keypoints = np.array(sfm04_index_keypoints)
base_sfm04_descriptors = np.array(sfm04_index_descriptors)
base_sfm04_matched_idx = np.array(sfm03_matched_idx_3D)

np.save("two_view_recon_info/base_sfm03_keypoints", base_sfm03_keypoints)
np.save("two_view_recon_info/base_sfm03_descriptors", base_sfm03_descriptors)
np.save("two_view_recon_info/base_sfm03_matched_idx", base_sfm03_matched_idx)
np.save("two_view_recon_info/base_sfm04_keypoints", base_sfm04_keypoints)
np.save("two_view_recon_info/base_sfm04_descriptors", base_sfm04_descriptors)
np.save("two_view_recon_info/base_sfm04_matched_idx", base_sfm04_matched_idx)
```

1. Loading Initial Data:

The keypoints, descriptors, and camera poses for the initial two imaged are loaded. Additionally, matched indices and inlinear arrays are loaded, which are essential for aligning the keypoints with the 3D points.

2. Sorting Keypoints and Descriptors:
   For each image, the keypoints and descriptors are sorted according to the matched indices. This step aligns the keypoints and descriptors in the order of their corresponding indices in the 3D point array. The sorted keypoints and descriptors are stored in new lists.
3. Indexing Keypoints and Descriptors:
   The next step involves indexing these sorted keypoints and descriptors using the inlinear array. The inlinear array contains the indices of keypoints that are inlinear (aligned) with the 3D points. For each value in the inlinear array, corresponding keypoints and descriptors from the sorted lists are appended to new lists.
4. Creating Base Keypoints and Descriptors Arrays: The indexed keypoints and descriptors are then converted into numpy arrays for easier handling and storage. These arrays now represent the base keypoints and descriptors that correspond to the 3D points.
5. Saving Processed Data:
   Finally, these processed keypoints, descriptors, and matched indices are saved as numpy files for subsequent use in the SFM process.

This preprocessing step ensures that the keypoints and descriptors are correctly aligned with the 3D points, facilitating accurate feature matching and camera pose estimation in the subsequent stages of the SFM process.

1.3. Step 1: SFM from multi views

In the subsequent step, SFM process is vital to integrate the remaining images into the initially constructed 3D model. This step is referred to as the "Growing Step" and is necessary for enhancing the depth and accuracy of the 3D reconstruction. The process is iterative and expands the model by incorporating one image at a time. The following algorithms outline the detailed procedure:

1. Image Selection Based on Neighborhood:
   Unlike the typical approach of selecting an image based on the highest number of matched keypoints with the reconstructed keypoints, this process assumes that the images are neighbors by their index. Therefore, matching is performed with the immediate neighboring image rather than searching for the image with the maximum matches. This assumption is based on the general understanding that neighboring images tend to have a better match due to their proximity and similar viewpoints.
2. Camera Pose Estimation:
   For the selected image, estimate its camera pose using a 3-point Perspective-n-Point (PnP) algorithm within a RANSAC framework. This step is crucial for determining the spatial orientation and position of the camera when the image was captured, relative to the reconstructed 3D scene.
   P3P algorithm estimates the camera's pose using three 2D keypoints and their corresponding 3D world points. It's often used in conjunction with RANSAC to minimize the influence of outliers. The camera's projection matrix P is composed of the internal parameters K and the external parameters $[R|t]$:

   $$P = K[R|t]$$

   A 3D point X in the camera coordinate system is projected to a 2D image point x as follows:

   $$x = PX$$

   Using this equation, the P3P algorithm calculates P based on three pairs of 3D-2D points.
3. 3D Point Reconstruction:
   With the camera pose of the selected image now estimated, the next step is to reconstruct 3D points. This is achieved through triangulation, utilizing the keypoints from the pose of the selected image and the pose of an already reconstructed image. The triangulation process computes the 3D coordinates of points by finding the intersection of corresponding rays from two camera viewpoints. Given two camera projection matrices P_1 and P_2, for the same 3D point X observed in both cameras, the following relationships hold:

   $$x_1 = P_1 X$$

   $$x_2 = P_2 X$$

   To find an X that simultaneously satisfies both projection equations, we form a homogeneous linear system and solve it using Singular Value Decomposition (SVD). This yields an approximate 3D position X.

Now, I repeat the above steps (1-3) until every image from the set has been included in the reconstruction.

To facilitate the SFM process from multiple views, a series of functions were implemented in 'functions.py', and the SFM process was executed in 'main.py'. The functions in 'functions.py' encompassed various essential steps, each playing a critical role in the overall workflow:

1. Feature Extraction (extract features):
   This function utilizes the SIFT (Scale-Invariant Feature Transform) algorithm to extract keypoints and their descriptors from given images. It reads an image, converts it to RGB format, and applies SIFT to identify and describe significant features
2. Feature Matching (match features):
   Given the descriptors from a base image and a new image, this function matches features using the BFMatcher (Brute-Force Matcher) with a ratio test. It filters out good matches based on the distance criteria, ensuring that only the most reliable matches are retained.

3. Reprojection to Image Plane (reproject to image plane):

   This function reprojects a 3D point back to the 2D image plane using the rotation matrix, translation vector, and the intrinsic matrix. It is critical for evaluating the accuracy of the reconstructed 3D points against the original 2D keypoints.

4. Camera Pose Estimation (estimate camera pose):

   The function estimates the camera pose of a new image using matched keypoints and 3D points. It employs the P3P algorithm within a RANSAC framework to robustly estimate the pose despite potential outliers. The best pose is determined based on the inliers count.

5. 3D Point Reconstruction (reconstruct 3D points):

   This function triangulates 3D points from matched keypoints between two images, utilizing their respective camera poses. It forms and solves a homogeneous linear system using Singular Value Decomposition (SVD) for each pair of matched keypoints.

6. 3D Point Validation (is valid 3D):

   After reconstruction, this function validates the 3D points by checking their reprojection error. It compares reprojected 2D points with the original keypoints and retains those with errors below a threshold, ensuring the accuracy of the 3D reconstruction.

These functions collectively form the backbone of the SFM process, handling everything from feature extraction and matching to 3D point reconstruction and validation.

Below is the code implementing this:

```python
def extract_features(img_path):
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    keypoints, descriptors = sift.detectAndCompute(img, None)
    return keypoints, descriptors

def match_features(base_descriptors, new_descriptors):
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)
    matches = bf.knnMatch(base_descriptors, new_descriptors, k=2)

    good_matches = []
    for m, n in matches:
        if m.distance < 0.8 * n.distance:
            good_matches.append(m)

    return good_matches

def reproject_to_image_plane(point_3d, R, t, K):
    t = t.reshape(-1)
    point_image_plane = K @ (R @ point_3d + t)
    point_image_plane /= point_image_plane[2]
    return point_image_plane[:2]
```

```python
def estimate_camera_pose(K, new_keypoints, new_descriptors, base_keypoints, base_descriptors, base_3D_points, matches):
    # base image의 matches
    query_idx = [match.queryIdx for match in matches]
    # new image의 matches
    train_idx = [match.trainIdx for match in matches]

    base_matched_keypoints = np.zeros([len(query_idx), 2])
    new_matched_keypoints = np.zeros([len(train_idx), 2])
    base_matched_descriptors = np.zeros([len(query_idx), 128])
    new_matched_descriptors = np.zeros([len(train_idx), 128])

    matched_3D_points = np.zeros([len(train_idx), 3])

    for i, idx in enumerate(query_idx):
        base_matched_keypoints[i] = base_keypoints[idx]
        base_matched_descriptors[i] = base_descriptors[idx]
        matched_3D_points[i] = base_3D_points[idx]

    for i, idx in enumerate(train_idx):
        new_matched_keypoints[i] = new_keypoints[idx].pt
        new_matched_descriptors[i] = new_descriptors[idx]

    best_pose = None
    best_inliers_count = 0

    for i in range(1000):
        indices = np.random.choice(new_matched_keypoints.shape[0], 3, replace=False)
        points_3D = matched_3D_points[indices]
        points_2D = new_matched_keypoints[indices]

        retval, rvecs, tvecs = cv2.solveP3P(points_3D, points_2D, K, None, flags=cv2.SOLVEPNP_AP3P)
        if retval:
            for rvec, tvec in zip(rvecs, tvecs):
                rvec, _ = cv2.Rodrigues(rvec)

                inliers_count = 0
                inliers = []

                for j in range(new_matched_keypoints.shape[0]):
                    point_2d = new_matched_keypoints[j]
                    point_3d = matched_3D_points[j]
                    reprojected_point = reproject_to_image_plane(point_3d, rvec, tvec, K)
                    error = np.sqrt(np.sum(np.square(point_2d - reprojected_point)))
                    if error < 2:
                        inliers_count += 1
                        inliers.append(j)
```

```python
def reconstruct_3D_points(K, base_camera_pose, new_camera_pose, base_keypoints, new_keypoints, matches):

    query_idx = [match.queryIdx for match in matches]
    train_idx = [match.trainIdx for match in matches]

    invk = np.linalg.inv(K)
    base_2D_points = np.float32([base_keypoints[m.queryIdx].pt for m in matches])
    new_2D_points = np.float32([new_keypoints[m.trainIdx].pt for m in matches])

    ones = np.ones((base_2D_points.shape[0], 1))
    base_points = np.hstack((base_2D_points, ones))
    new_points = np.hstack((new_2D_points, ones))

    normalized_base_points = (invk @ base_points.T).T
    normalized_new_points = (invk @ new_points.T).T

    P1 = base_camera_pose
    P2 = new_camera_pose

    points_3D = np.zeros((base_2D_points.shape[0], 3))

    for i, (base_point, new_point) in enumerate(zip(normalized_base_points, normalized_new_points)):
        A = np.zeros((4, 4))
        A[0] = base_point[0] * P1[2] - P1[0]
        A[1] = base_point[1] * P1[2] - P1[1]
        A[2] = new_point[0] * P2[2] - P2[0]
        A[3] = new_point[1] * P2[2] - P2[1]

        _, _, Vt = np.linalg.svd(A)
        points_3D[i] = Vt[-1, :3] / Vt[-1, 3]

    return query_idx, train_idx, base_2D_points, new_2D_points, points_3D
```

```python
def is_valid_3D(K, base_camera_pose, query_idx, base_matched_idx, base_3D_points, base_2D_points, points_3D):
    valid_3D_points = []
    inlinear = []
    R = base_camera_pose[:3, :3]
    t = base_camera_pose[:3, 3]

    for i, idx in enumerate(query_idx):
        if idx in base_matched_idx:
            j = np.where(base_matched_idx == idx)
            # idx가 base_matched_idx의 몇번째에 위치하는지 확인하고 j번째에 위치한면 base_3D_points[k]를 구함
            error = np.sqrt(np.sum(np.square(points_3D[i] - base_3D_points[j[0][0]].T)))
            if error < 0.5:
                valid_3D_points.append(base_3D_points[j[0][0]].T)
                inlinear.append(i)

        else:
            reprojected_to_2D = (R @ points_3D[i].T + t)
            reprojected_to_2D = K @ reprojected_to_2D
            reprojected_to_2D /= reprojected_to_2D[2]
            error = np.sqrt(np.sum(np.square(reprojected_to_2D[:2] - base_2D_points[i])))
            if error < 2.5:
                valid_3D_points.append(points_3D[i])
                inlinear.append(i)

    new_inlinear = np.array(inlinear)
    new_3D_points = np.float32(valid_3D_points)

    return new_inlinear, new_3D_points
```

Utilizing the functions defined in functions.py, the main.py script operates as follows to execute the Structure-from-Motion (SFM) process:

For each image in the sequence:

1. Determine the base image and related data depending on the current image index.
2. Extract features (keypoints and descriptors) from the new image.
3. Match features between the base image and the new image.

4. Estimate the camera pose of the new image using the matched features and pre-calculated 3D points.
5. Repeat the feature extraction and matching process for further refinement.
6. Reconstruct 3D points from the estimated camera pose and the base camera pose using triangulation.
7. Validate the reconstructed 3D points to ensure their accuracy.
8. Update and save the keypoints, descriptors, matched indices, and camera pose for the next iteration.
9. Append the new valid 3D points to the overall 3D point array.

Saving Final Data: Save the final aggregated 3D points for future use or analysis.

Below is the code implementing this:

```
# Extract features for the new image
new_keypoints, new_descriptors = extract_features(new_image_path)

# Match features between the base image and the new image
good_matches = match_features(base_descriptors, new_descriptors)

# Estimate camera pose
new_camera_pose = estimate_camera_pose(K, new_keypoints, new_descriptors, base_keypoints, base_descriptors, base_3D_points, good_matches)

# 3D Recon
new_keypoints, new_descriptors = extract_features(new_image_path)
base_keypoints, base_descriptors = extract_features(base_image_path)
good_matches = match_features(base_descriptors, new_descriptors)
query_idx, train_idx, base_2D_points, new_2D_points, recon_3D_points = reconstruct_3D_points(K, base_camera_pose, new_camera_pose, base_keypoints, new_keypoints, good_matches)

new_inlinear, new_3D_points = is_valid_3D(K, base_camera_pose, query_idx, base_matched_idx, base_3D_points, base_2D_points, recon_3D_points)

new_index_keypoints = []
new_index_descriptors = []
new_matched_idx_3D = []

for value in new_inlinear:
    new_index_keypoints.append(new_keypoints[value].pt)
    new_index_descriptors.append(new_descriptors[value])
    new_matched_idx_3D.append(train_idx[value])

next_base_keypoints = np.array(new_index_keypoints)
next_base_descriptors = np.array(new_index_descriptors)
next_base_matched_idx = np.array(new_matched_idx_3D)

prev_3D_points = new_3D_points

all_3D_points = np.vstack((all_3D_points, new_3D_points))

prev_keypoints = next_base_keypoints
prev_descriptors = next_base_descriptors
prev_matched_idx = next_base_matched_idx
prev_camera_pose = new_camera_pose

np.save("prev_keypoints", prev_keypoints)
np.save("prev_descriptors", prev_descriptors)
np.save("prev_matched_idx", prev_matched_idx)
np.save("prev_camera_pose", prev_camera_pose)
```

1.4. Step 2: Bundle Adjustment

Bundle Adjustment is a sophisticated optimization technique in the Structure-from-Motion (SFM) process, focusing on simultaneously refining both the 3D structure and camera poses. Unlike warping, which involves transforming an image based on geometric alterations, Bundle Adjustment optimizes the 3D points and camera parameters to minimize the reprojection error. This error is the discrepancy between the observed image points (2D keypoints) and the projected points from the 3D model onto the image plane.

In theory, Bundle Adjustment formulates the SFM problem as a non-linear least squares problem, where the objective is to find the 3D point positions and camera parameters (including position, orientation, and intrinsic parameters) that best fit the observed data. The process iteratively adjusts the 3D points and camera poses to minimize the sum of the squared differences (errors) between the observed keypoints in the images and the projected points from the 3D model.

However, in the context of this project, the implementation of Bundle Adjustment was not accomplished. The theoretical understanding of its importance in enhancing the quality of the 3D reconstruction is acknowledged, but due to constraints of technical complexity, its practical application was not realized in the current scope of the project. The future aim would be to incorporate this critical step to further refine and improve the accuracy of the 3D models generated from the SFM process.

2. Result & Discussion

While working on this Structure-from-Motion (SFM) project, there was a lot of learning and effort put into trying to make it work. Despite this, the results weren't entirely as expected:

1. Issues with Later Images:
   In the beginning, the SFM process did a good job of creating 3D points from the first few images. This part of the project was quite promising and showed that the approach could work. However, as the project went on, it became clear that the later images weren't getting turned into 3D points as well. A lot of these points either didn't get made correctly or were thought to be errors or 'outliers'. One big reason for these issues could be that the camera positions weren't figured out accurately. Knowing exactly where the camera is and how it's tilted is really important for making a good 3D model. If there are mistakes here, it can mess up the whole process. Another reason might be related to not setting the right limits or 'thresholds' for deciding which features of the images to use and which ones to ignore. Getting these limits right is really important for good results.
2. Hopes for the Future:
   Despite these problems, there's still a strong wish to get the SFM process to work fully. Also, once the main part of SFM is working well, it would be great to try out the Bundle Adjustment phase. This advanced step is expected to make the 3D models even better.

The image file (multiview.ply) shows the 3D points that were made from 14 images. Although it doesn't show everything perfectly, it gives a good idea of where things are at right now, showing both what went well and what needs more work.

This whole experience really shows how tricky it can be to make 3D models from several images, and it highlights how important it is to get the camera position just right and to set the right limits in the process.