Practical session in 3D Vision

-

SfM : Structure from Motion

Objectives

Structure from motion is a technique to perceive depth based on movement. It is based on the human experience of trying to understand the structure of a scene. SfM is based on the principle that much of the 3D structure of a scene can be determined by 2D snapshots taken from different view point by a moving camera. Bear in mind that the motion can be with respect to an observer in a static scene, or with respect to an object with a static observer. Both cases provide an opportunity to reconstruct the 3D scene structure, but here, we consider the former.

This practical session summarizes the mathematical and programming concepts behind SfM. We will use here a sequence of 8 consecutive images acquired with the same calibrated camera (see figure 1).

Figure 1: Example of the images of the sequence.

The objective is then to compute the relative pose of the different views to reconstruct the sparse 3D model of the fountain (see figure 2).

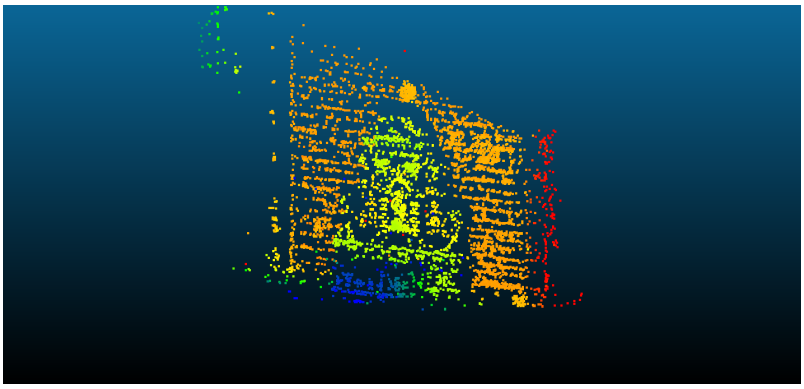

Figure 2: Sparse 3D reconstruction from incremental SfM.

1 Basics

1.1 Reminder

A camera is represented by a matrix M which transforms a point P in the real world to a pixel p in the image. This relationship is given by the following equation.

$$p = MP$$

In order to deduce P from its image coordinate p, we need to do the following:

$$P = M^{-1}p$$

To summarize, a world point is converted to camera coordinates using the extrinsic matrix, which are then converted to homogeneous image coordinates using the intrinsic matrix K. If we apply K^{-1} to a homogeneous image point p, we end up with something known as the normalized image coordinates (NICs), which are points transformed into the coordinate system of the camera. Applying the inverse of the extrinsic matrix on NICs gives us the 3D world coordinates.

$$\underbrace{s\begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix}}_{\text{2D image coordinates}} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Intrinsic matrix}} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}}_{\text{Extrinsic matrix}} \underbrace{\begin{bmatrix} P_X \\ P_Y \\ P_Z \\ 1 \end{bmatrix}}_{\text{3D world coordinate}}$$

So, finding the extrinsic matrix or the camera pose for each of the images (the 2D snapshots or views) is a part of the process of reconstructing 3D coordinates. SfM, thus, involves estimating the 3D points along with the camera pose from a sequence of images. The 3D points are recovered by a procedure known as triangulation that uses camera poses to accurately locate the points in space.

1.2 Two-View Geometry

In the figure 3, two image planes are shown, X/Y are the points in the 3D world, x/y and x'/y' are the corresponding image points in each of the views I and I', C and C' are the optical centres of the cameras, and XCC'/YCC' are the epipolar planes containing the 3D points and the camera centres. e and e' are the epipoles, or the images of the camera centers in the opposite views. $(e'x')/(e'y')$ are the images in the second view of the ray passing from C through x/y respectively and are called epipolar lines. They are also the lines of intersection of the each epipolar plane with the second view.

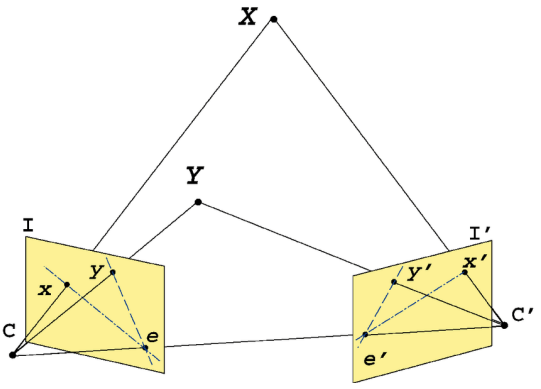

Figure 3: Two-view geometry.

1.3 The fundamental matrix F

If we try to find the coordinate in the second view that is looking at the same point as the coordinate in the first view, all we have to do is to look across the epipolar line. This geometry is basically encapsulated in a matrix known as the fundamental matrix, which relates a point in one image to a line in another image. The fundamental matrix F is defined by:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} .F. \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = 0$$

The camera matrices can then be deduced from the fundamental matrix given some pixels matchings between the two views I and I' ([1] - Chapter 9).

1.4 The essential matrix E

The essential matrix is a representation of the fundamental matrix in the case where we have calibrated cameras. The essential matrix E is defined by:

$$E = K^T.F.K$$

The NICs corresponding to a 3D point in a pair of views are related by the essential matrix, similar to how two image points are related by the fundamental matrix, so that if X_1 and X_2 are the NICs of the 3D point X respectively expressed in camera 1 and camera 2, X_1 and X_2 are related as such:

$$X_2^T.E.X_1 = 0$$

Finding the essential and fundamental matrices allows then to decompose them into the extrinsic matrix and ultimately obtain the camera pose or the camera matrix. As mentioned above, the camera pose is a component that is required for triangulation of the 3D points which is the objective of SfM.

1.5 Incremental and Global SfM

These are two of the prominent pipelines used to solve the SfM problem. Incremental SfM chooses two images as the baseline views, obtains an initial reconstruction, and incrementally adds new images. Global SfM considers all camera poses at once to obtain the reconstruction. In this practical session, we will implement a simplified version of incremental SfM. Typically, incremental SfM implements some form of view selection or filtering to choose the best image to add to the reconstruction. In this implementation, we consider that the camera has been calibrated, i.e. intrinsic matrix K is known, and we will use images which have been acquired in a sequence and are named in the order in which they have been taken. The baseline views are the first two views in the set, and the subsequent view to be added is the immediately next image in the sequence. This simplifies the implementation significantly, but introduces some errors in the reconstruction which is an acceptable trade-off given that the purpose of this session is to describe the main pipeline behind SfM.

2 Pixel matching between two views

Finding the fundamental matrix is done using a variety of techniques, but the 8-point algorithm is a pretty popular approach. The algorithm uses the epipolar constraint to build up a system of homogeneous equations from which the components of the fundamental matrix are computed. For two homogeneous image points p and p' in different images looking at the same 3D point, the epipolar constraint is mathematically defined as:

$$p'^T F p = 0$$

Since the fundamental matrix is defined up to scale, there are 8 unknowns in the matrix to be determined requiring a minimum of 8 point correspondences to solve the linear system of equations using least squares optimization. Usually, more correspondences are preferred to obtain low errors. In the below system of equations, (u_n, v_n) and (u'_n, v'_n) are the x- and y-coordinates of the n^{th} corresponding point in two different views and $F_{11} - F_{33}$ are the values of the fundamental matrix.

$$\begin{bmatrix} u_1u'_1 & v_1u'_1 & u'_1 & u_1v'_1 & v_1v'_1 & v'_1 & u_1 & v_1 & 1 \\ u_2u'_2 & v_2u'_2 & u'_2 & u_2v'_2 & v_2v'_2 & v'_2 & u_2 & v_2 & 1 \\ u_3u'_3 & v_3u'_3 & u'_3 & u_3v'_3 & v_3v'_3 & v'_3 & u_3 & v_3 & 1 \\ u_4u'_4 & v_4u'_4 & u'_4 & u_4v'_4 & v_4v'_4 & v'_4 & u_4 & v_4 & 1 \\ u_5u'_5 & v_5u'_5 & u'_5 & u_5v'_5 & v_5v'_5 & v'_5 & u_5 & v_5 & 1 \\ u_6u'_6 & v_6u'_6 & u'_6 & u_6v'_6 & v_6v'_6 & v'_6 & u_6 & v_6 & 1 \\ u_7u'_7 & v_7u'_7 & u'_7 & u_7v'_7 & v_7v'_7 & v'_7 & u_7 & v_7 & 1 \\ u_8u'_8 & v_8u'_8 & u'_8 & u_8v'_8 & v_8v'_8 & v'_8 & u_8 & v_8 & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = 0$$

So how do we get 8 or more corresponding points in two images? The solution is to use feature extraction algorithms such as SIFT, SURF and ORB. These feature points are coordinates in the image and are defined by feature descriptors - vectors that uniquely encapsulate certain properties of a point. The idea is that descriptors of two points can be matched using metrics like Euclidean distance to measure how similar they are. Essentially, for a point in one image, we could obtain a list of points in another image that represents the same point in the real world, i.e, we could obtain pixel matchings between two image pairs.

2.1 View

Each image in the `input/` folder is stored in Python as a *View* data structure. A *View* object contains essential information about the image such as the image array, image name, list of keypoints and descriptors (see figure 4), and the rotation and translation components of the extrinsic matrix. These last components are calculated gradually for every subsequent View and is used to triangulate new 3D points.

Figure 4: Example of extracted SIFT descriptors.

1. Compute the SIFT descriptors for each view and store them in the *keypoints* and *descriptors* attributes of the View object by completing the file `view.py`.
2. Display, for each image, the number of keypoints, draw them on the image and store it.

For such purpose, check OpenCV documentation of the functions `detectAndCompute`, `drawKeypoints` and `imwrite`.

2.2 Matchings

A *Match* object contains information about the feature matches between any two pairs of images. It contains the indices of the keypoints in each of the images that are similar (see figure 5). The matching is performed using the brute-force matcher. Update the file `match.py`.

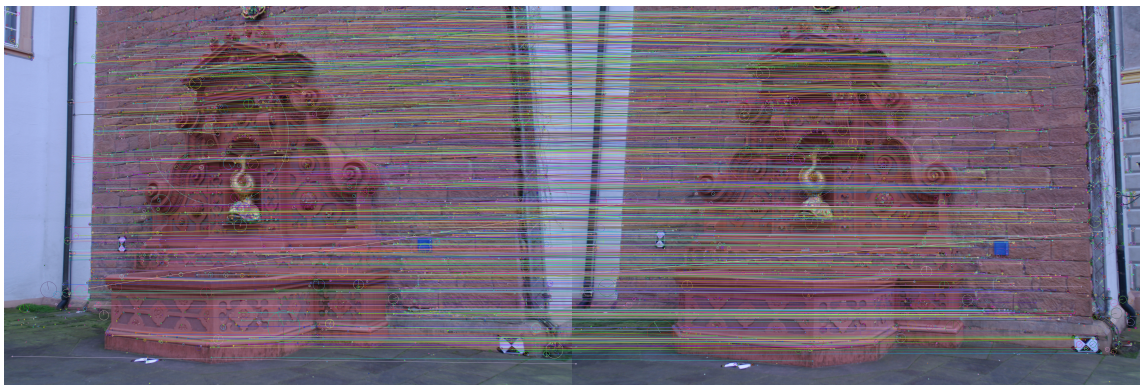

Figure 5: Example of matchings between two views.

1. Draw, for each pair of view, the matched keypoints and store the result.

For this, check OpenCV documentation of the functions `drawMatches` and `imwrite`.
Once the features and matchings are computed, the main reconstruction loop can start.

3 Baseline Pose Estimation

The first two images in the image set are taken as the baseline. This means that we consider the position of the first camera frame to be the origin in world coordinates and calculate the position of the remaining views relative to the first pose. The *sfm* class takes a list of *Views* and a dictionary of *Matches* and iterates through them to gradually reconstruct the points. The first two views are passed to a Baseline object which specifically estimates the pose for the baseline. The Baseline object Calls the `remove_outliers_using_F()` function to calculate the fundamental matrix between the two views. This is done using the OpenCV in-built function `findFundamentalMat()` that uses the 8-point algorithm. Since it is a least squares optimization, we can perform post-processing on the keypoints using the obtained fundamental matrix and remove the points that don't obey the epipolar constraint. These inliers are stored in the match object and are the final points that are reconstructed. This is easy as the OpenCV function returns a mask array to filter the points.

1. Calculate the essential matrix from the fundamental matrix F and the intrinsic matrix K. As mentioned earlier, the essential matrix is a specialization of the fundamental matrix. The proof for the above formula is given in definition 9.6 of Chapter 9 in [1].

2. To extract the rotation and translation elements of the extrinsic matrix from the essential matrix, update function `get_camera_from_E()` (file `utils.py`) to perform singular value decomposition on the essential matrix E using the function `np.linalg.svd()` so that:

   $$E = U.\Sigma.V^T$$

   where U and V are orthogonal 3×3 matrices and Σ is a 3×3 diagonal matrix with

   $$\Sigma = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{pmatrix}$$

   - The 2 solutions for the rotation matrix can then be computed so that:

     $$R_1 = U.W.V^T \text{ and } R_2 = U.W^T.V^T$$

     with

     $$W = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

   - The 2 solutions for the translation vector can then be computed so that:

     $$t_1 = U.\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{ and } t_2 = -t_1$$

   This procedure is quite lengthy and described in detail in section 9.6.2 of [1].

The 2 different values obtained for the rotation and the translation elements - R_1, R_2, t_1 and t_2 give us 4 different solutions for the extrinsic matrix based on the number of possible combinations. These four solutions are checked for their validity in the `check_pose()` function and the correct pair is chosen as the extrinsic matrix of the second view relative to the initial view.

4 Triangulation for 3D reconstruction

Triangulation is the process of using 2D image matchings between two image pairs as well as pose information to reconstruct 3D points. Applying the inverse of the camera matrix on an image point gives us a ray passing through the point. After acquiring the ray of the corresponding point in the second view, we can find the 3D point by finding the point of intersection of the two rays. This is the basic theory behind triangulation (see figure 6).

The function `triangulate()` loops through the inlier points from two views and calls the `get_3D_point()` function to obtain the 3D point. The triangulation method used in `get_3D_point()` is provided in Chapter 4 of [2], but alternately, we can also use the OpenCV function `triangulatePoints()` for the same purpose.

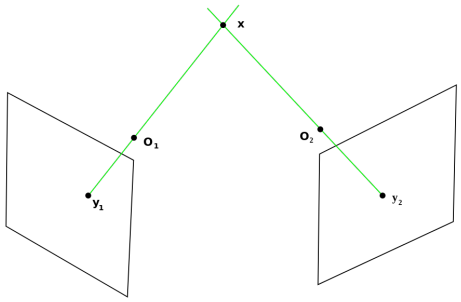

Figure 6: Triangulation between two views.

5 Reprojection error

The reprojection error is an accuracy metric for reconstruction problems. For each triangulated point, we project the 3D point back on the image plane using the camera matrix and compare the result with the 2D point that was used in the triangulation. Since triangulation is a least squares optimization problem, it has an error term involved. Calculating the mean reprojection error everytime we intergrate a new view gives us an idea about how off the triangulation is and allows us to apply a technique called bundle adjustment to minimize the error. Bundle adjustment is not implemented in the code, but it would be sufficient to know that the technique uses the reprojection error as an objective to minimize and makes corrections to the computed camera poses as well as the triangulated points.

1 Insert the computation of the reprojection error in the function `calculate_reprojection_error()` of file `utils.py` by computing the euclidean distance between the 2D point and the projection of the corresponding 3D point in the image plane.

The triangulated points are stored in `self.points_3D` and are now ready to be plotted.

2 Update function `save_points()` in file `sfm.py` to store the 3D points in a .xyz file in order to open it with CloudCompare

6 Integration of New View

Ideally, the next view to be added to the reconstruction would be filtered and chosen using some criteria, such as number of keypoints and matches. But since we are processing all images sequentially, the next view to be added is simply the next image taken in order of the naming.

To compute the pose of the new view, we use a technique called Perspective n Point. The technique uses components of the new view such as 2D points, the intrinsic matrix, and the 3D points to compute it's pose. For that, we need to use the previous 3D points triangulated in the previous steps whose corresponding image points have matches in the new view. Given an old keypoint, we can extract, if any, the 3D point it contributed to. This is done immediately after triangulation of a point.

Thus, we extract the keypoints in the new view which have a match against a keypoint in an older view that was triangulated. This gives a list of image points in the new view and a list of 3D points that they possibly point to (through their matches). This is the input to the `solvePnPRansac()` function which estimates the pose of the new view. RANSAC is used to filter out matching outliers.

1. Update function `compute_pose_PNP()` in file `sfm.py` to compute the pose of the new View (R and t) by using `solvePnPRansac()`.

2. Uncomment the last lines of the function `reconstruct()` to automatically add the new Views (see figure 7)

Now that we have obtained the pose for the new view, we go through the matches of the new view with all of the old views to triangulate points that have so far not been triangulated. The matches information was computed in one of the initial step for every view with every other view, and now the reason for that is apparent.

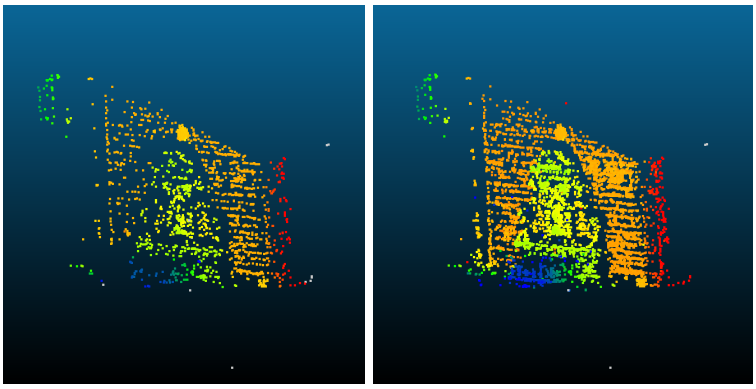

Figure 7: Initial and final sparse point cloud.

7 Discussion

Some of the images might not yield accurate reconstructions because of the absence of bundle adjustment. Incremental SfM is a gradual process of refining the reconstruction of a scene. Errors made in estimation in the previous steps can easily propagate to later steps and contribute to a reprojection mismatch. However, what is important here is the pipeline followed to obtain a reconstruction. To reiterate, we start with two baseline views, the second of which we compute the pose for, after which we triangulate 3D points between them. We then incrementally add new views by first computing their pose using Perspective n Point (PnP) and then triangulating new points in the scene. Usually, the next step is to densify the reconstructed point cloud. To do so, we can use a pair of image, rectify them using their relative pose to compute dense correlation.

References

[1] Hartley and Zisserman's Multiple View Geometry in Computer Vision

[2] Packt's Mastering OpenCV with Practical Computer Vision Projects