Università degli studi di Padova
3D Data Processing
Lab2: Structure from motion

Federico Gelain, ID: 2076737
Anne Linda Antony Sahayam, ID: 2088365

Task 1: Features Extraction

To extract the features from the sample images ORB was chosen. As for its parameters, the only one that differs from the default values is **nFeatures**. This is because retaining at most 500 features from each image doesn't provide good results in the matching part. The final choice for its value is 12000, which sacrifices some execution speed in favor of more accuracy. After computing salient points (features) and descriptors, the color of each feature is extracted by simply locating each feature point (using the coordinates provided by the **pt** field of each feature) and retrieving from the image the corresponding color.

Task 2: Descriptors Matching

To perform the matching between the descriptors of each pair of images, a BruteForce matcher was employed (it compares each descriptor of the first image with all the ones of the second image and chooses the closest one). As measure of closeness between descriptors, **the normType** chosen was **NORM_HAMMING**, accordingly to the OpenCV documentation of the method create(), since ORB (with the default value of WTA_K) was chosen. As for the match of the descriptors, the description of crossCheck parameter of the same method gives some insight on what choices best suit the problem:

- if **crossCheck = false**, the best approach is to call knnMatch() to perform the descriptors matching and perform the **ratio test** to select only the matches that bring enough information. The simplest and most effective way to do that is to set k = 2 and check that the distance between the two chosen matches is greater than a tunable parameter, in which case you keep the nearest match;
- if **crossCheck = true**, call match() (which is essentially knnMatch() with k = 1) so that it automatically performs cross checking between the descriptors of the two images.

Ultimately the second approach was chosen, since it doesn't require to tune any parameter for the ratio test and the results obtained were very similar. Using the match points retrieved from the i-th image (query points) and the j-th image (train points), the Essential and Homograph matrices were computed for geometric verification. A key value returned by both of the functions used to compute them is **mask**, an array of N elements (same size of the matches found between the two images descriptors), where each element is equal to 0 if the corresponding match is an outlier, 1 otherwise. This easily allows to obtain all the inlier matches, and set them if at least 6 of them are found.

Task 3: Rigid body transformation between seed pairs

The Essential matrix E and the Homograph matrix H were computed using the same function of task 2. This again allows us to find the number of inliers of each model and check if the current seed pair is good or not (it is if the number of inliers of E is greater than the one of H). It's important to point out that the advised value of 0.001 for the threshold to estimate both models is too strict (most of the time the results of the reconstruction don't make sense and can vary a lot from different runs of the program). 0.1 was chosen instead, which guarantees way more consistent results (though

sometimes the point clouds are reconstructed with less points than expected, especially the Aloe). The rigid body transformation between the seed pair is recovered from E using its inlier mask, which returns the corresponding rotation matrix and translation vector separately. To then check if a "good" sideward motion (wrt the camera) occurs between the seed pair, it's enough to look at the first and last value of the translation vector. The first one provides the information of the sideward motion (x axis), while the last one provides the information of the forward motion (z axis), so it's just a matter of comparing the absolute value of both and checking that the first one is greater.

Task 4: Point Triangulation

To triangulate each 3D point obtained by using the observation of this point in the camera poses with indices **new_cam_pose_idx** and **cam_idx**, the function cv::triangulatePoints() requires the projection transformation matrix of both cameras, and the 2D projection of the observed 3D point. The pointers **cam0_data** and **cam1_data** provide the information needed for the computation of the matrices. In fact, they point to the axis-angle vector representation of the rotation performed (first 3 values) and the translation vector (last 3 values). The function cv::Rodrigues() allows to convert the axis-angle representation in the corresponding rotation matrix (3x3), and by concatenating it with the translation vector (3x1) the transformation matrix is obtained (3x4). After obtaining the triangulated point (4x1 vector, so the 3D point expressed in homogeneous coordinates $\begin{bmatrix} x_p & y_p & z_p & w \end{bmatrix}^T$), the cheirality constraint for both cameras is checked. This requires that the triangulated point obtained is in front of the camera, which can be easily checked as $\frac{z_p}{w} > 0$.

Task 5: Auto differentiable cost function definition

To compute the difference between observed and predicted image coordinates of a 3D point projected onto an image plane **ReprojectionError struct** is used. It holds the observed coordinates **(observed_x and observed_y)** and defines the operator () which computes the error given camera parameters and 3D point coordinates.Additionally,the constructor initializes the observed_x and observed_y coordinates of a point with the values provided when an instance of ReprojectionError is created.The template operator is where the actual computation of reprojection error happens. It takes in camera parameters and a 3D point as input. It first rotates the point according to the camera's rotation using **ceres::AngleAxisRotatePoint**. Then, it translates the rotated point by the camera's translation. After that, it computes the projected coordinates **(xp and yp)** by dividing the translated coordinates by the depth **(p[2])**. Finally, it computes the error by subtracting the observed coordinates from the projected ones and stores the result in the residuals array.
ceres::AutoDiffCostFunction creates a cost function that represents the reprojection error, which can be used by Ceres Solver during optimization. The **AutoDiffCostFunction** automatically computes derivatives using automatic differentiation, making it easier to use with custom cost functions like **ReprojectionError**. This comprehensive approach enables precise optimization in scenarios such as camera calibration or structure-from-motion tasks.

Task 6: Residual block definition for Ceres problem

A residual block is added to the Ceres solver problem, contributing to the bundle adjustment process in Structure from Motion tasks. **c_index** and **p_index** are used to store the indices of the camera pose and 3D point associated with the current observation **i_obs**. A cost function representing the reprojection error for the current observation is created. This cost function quantifies the discrepancy between the observed image coordinates and the coordinates predicted by the current camera pose and 3D point position. Finally, a residual block is added to the optimization problem using the created cost function. This residual block encapsulates the error between observed and predicted image coordinates. Additionally, a Cauchy loss function is applied to handle outliers in the data, ensuring robust optimization. The pointers to camera parameters and 3D point coordinates are included in the residual block, indicating which parameters need to be adjusted during optimization.

Task 7: Reconstruction divergence handling

This task is crucial for detecting divergence in the reconstruction process within a Structure from Motion framework. It assesses whether the iterative refinement of camera poses and 3D point positions has deviated substantially from the correct solution, which can occur due to various factors such as incorrect triangulations or convergence to erroneous local minima during optimization. To do so, the parameters are retrieved **before** the current iteration refinement (so essentially before the call of the function **bundleAdjustmentIter()**) in a vector of 3D points called **parameters_prev**, and compared with the updated ones, stored in **parameters_**
The challenge was to find a suitable way to check whether a divergence had occurred or not, and several methods were tested.
For the camera poses, it was decided to compare the values of each pose component and if, for at least one of them, the ratio between the one before and after the update is below a certain threshold (the one chosen was 0.09, meaning that after the update it has increased by more than 10 times), then the pose is counted as "diverged". If more than half of the camera poses have "diverged", then the reconstruction must be reset and restarted from scratch with a new pair of seeds.
For the 3D points, after some not satisfactory attempts (similar approach of the camera poses computing the Euclidean distance, computation of the bounding box that encloses all the 3D points), it was ultimately decided to perform PCA (Principal Component Analysis). This procedure allows to extract from a given set of points (the 3D points contained in parameters in this case) the most important features, enclosed in the resulting eigenvectors, whose directions indicates where the data varies the most (the direction that yields more information about the dataset), and eigenvalues (the size of each eigenvector). Thus the idea was to compare the eigenvalues and check if after the refinement their value was too different, meaning a divergence in the reconstruction.
From both parameters vectors, all 3D points whose coordinates are not infinite are stored as rows in the matrices **data_pts_prev** and **data_pts_curr** and the PCA analysis is performed by calling PCA(). After that, the eigenvalues of both set of points are retrieved and compared with each other. If the ratio between the eigenvalue before and after the refinement is lower than a certain threshold (e.g. 0.5, which means that after the refinement it has more than doubled), then a divergence in the reconstruction process has occurred. In that case, the function returns false to reset the reconstruction and restart it with a new pair of seeds.

Results

Below are shown the resulting point clouds for the two provided datasets:

Figure 1: Sampled image from dataset1

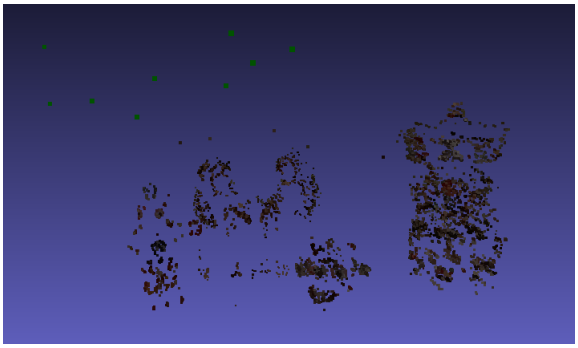

Figure 2: Point cloud obtained using dataset1

Figure 3: Sampled image from dataset2

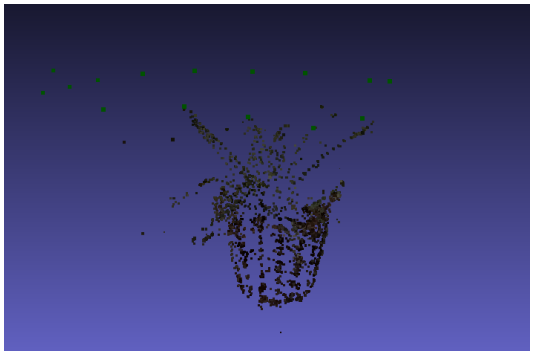

Figure 4: Point cloud obtained using dataset2

Here is shown the result of the point cloud reconstruction of the dataset created using the calibrated camera, comprising of some pottery figures:

Figure 5: Sampled image from custom dataset

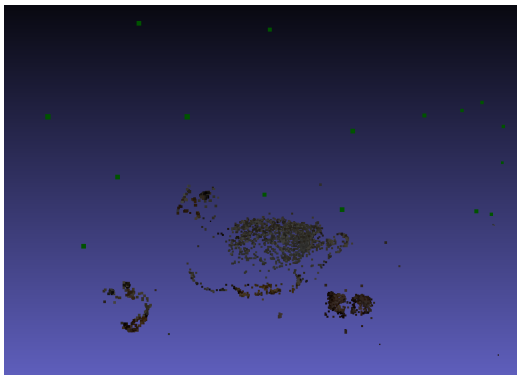

Figure 6: Custom dataset point cloud

Contributions

The work was divided between each member in the following way:

- Tasks 1,2,3,4,7 and custom dataset definition (Federico Gelain);
- Tasks 5,6 (Anne Linda Antony Sahayam);

It's important to point out that even if each task was written (code-wise and in the report) by the corresponding person listed above, we didn't hesitate to help each other in case of any problem/doubt. This is relevant in particular for task 7, since there wasn't a clear and definitive idea on how to solve it.