Edoardo Bastianello ID:2053077, Simone Mosco ID:2055298

Lab 2: Structure from Motion

Tasks

1. **Feature Extraction**: The first task was completed by using the `cv::detectAndCompute()` function, which filled the `features_` and `descriptors_` vectors. For each feature, we extracted the corresponding color value of the pixel by looping through the image. We tested both ORB and SIFT detector, finding out better features with ORB for all the datasets.

2. **Descriptors Matching**: We compute the matches using a Brute-force matcher with Hamming norm and `cv::knnMatch()` which finds the k best matches (in our case we set $k = 2$) for each descriptor from a query set. Then, we loop over all pairs of matches, using Lowe's ratio test with threshold equals to 0.9 in order to find the good matches and extract the points involved through the `features_` vector. Later, we extract the inlier masks for E and H using the `cv::findEssentialMat()` and `cv::findHomography()` functions, respectively and as threshold 1.0. We also tried different values for this threshold. In particular, we tried setting it to 3.0; however, with a value of 3.0, we obtained too many spurious matches. We consider as inliers matches all the matches found in at least one of the two inlier masks. Finally, as suggested, we set matches between two images if the amount of inliers matches is greater than 5.

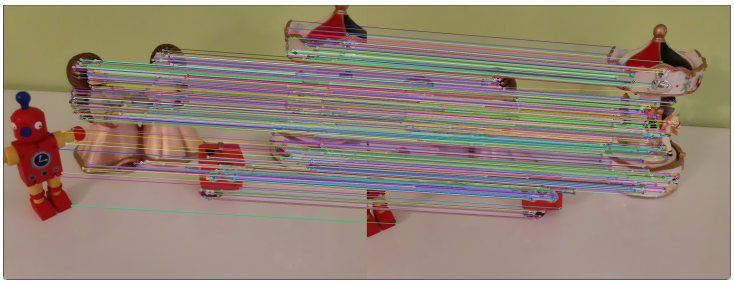

Figure 1: Images 1 matches

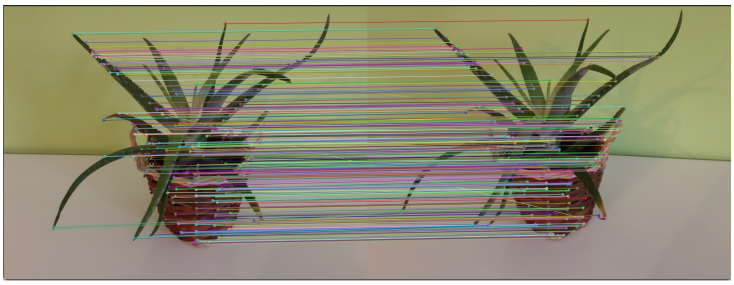

Figure 2: Images 2 matches

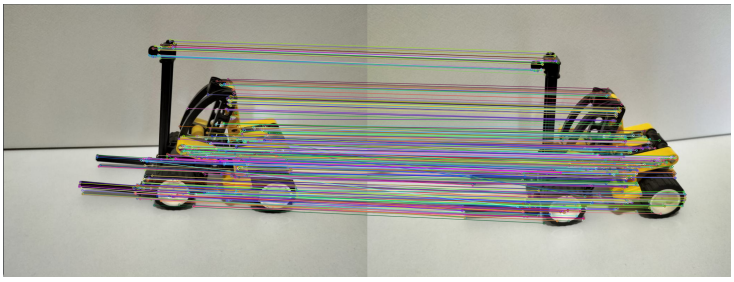

Figure 3: Custom dataset matches

3. **Seed Pair**: To select the seed pair, we first retrieve the essential matrix E and the homography matrix H using the same functions as in task 2. In particular, we tested multiple values for the threshold of the `cv::findEssentialMat()` and `cv::findHomographyMat()` functions. We found out that 2.5 was the value that was more consistent for all datasets. In fact, with other values only one of the datasets worked well. We then count the number of inliers for both models and check if the number of inliers for E is greater than the number for H. If this condition is true, we compute the rigid body transformation between the two camera poses using R and t through the `cv::recoverPose()` function. We then check if the motion is mainly sideward by examining the x coordinate of the translation vector t. If the absolute value of the x coordinate is greater than the absolute value of the z coordinate, we consider the motion to be mainly sideward and select this pair of camera poses as the seed pair. Regarding the check of the sideward motion, we also explored other options, such as adding a minimum thresold for the x coordinate, using a maximum threshold for the z coordinate, adding a check on the rotation; however, we didn't notice any improvements with respect to the first check ($x > z$), so we only kept this one. Finally, if the motion is sideward, we set the current pair as seed pair and we set `init_r_mat` and `init_t_vec` as the rigid body transformation between the two camera poses.

4. **Triangulation**: First of all, we retrieve the 2D points for both camera poses. To do this, we accessed the `observations_` vector. Then, for each camera, we build the `axis_angle` and `t` vectors by accessing the camera data. In particular, the first three elements of `cam0_data` and `cam1_data` contain the information to build the `axis_angle` vector, while the following three elements contain the information about the translation. Next, we compute the rotation matrix R using the `cv::Rodrigues()` function and we construct the projection matrices by concatenating the R matrix with the t vector. The triangulation of the 3D point is performed by OpenCV using the `cv::triangulatePoints()` function, which reconstructs the 3D point in homogeneous coordinates by taking in input the two projection matrices and the set of observations in the two cameras. Finally, we check the cheirality constraint by inspecting the z coordinate of the 3D point too see if it's greater than 0 and we add the code provided that increments the points counter and converts back the coordinates of the points from the homogeneous representation.

5. **Cost Function**: in order to create an auto-differentiable cost function for Ceres solver we followed the Ceres Solver bundle adjustment tutorial provided. In particular, we created the struct `ReprojectionError` that takes as constructor parameters the x and the y coordinates of the considered point. This struct contains the operator() function that, given the camera, the 3D point and the residuals, projects the 3D point and computes the (2 dimensional) residual. With respect to the tutorial, we changed the number of parameters of the camera from 9 to 6, since we are considering a canonical camera, that has only three parameters for rotation as a Rodrigues' axis-angle vector and three parameters for translation.

6. **Residual Block**: we completed the last task using the Ceres function `AddResidualBlock()` with the `ReprojectionError` class defined in the previous task as cost function, the Cauchy Loss as loss function, the pointer to the camera parameters and the pointer to the 3D point parameters.

Issues

During the development of the Assignment we encountered the following problems:

- For the matcher application, the main problem we encountered was the large number of outliers in the Aloe dataset. For this reason, after several tests, we set the threshold to estimate the Essential and the Homography matrices to 1. With a higher value, we obtained too many outliers, while with a smaller value we didn't obtain enough matches.

- Due to the probabilistic nature of the Ceres solver, the point cloud given as output may differ from run to run. For this reason, in task 3, we set as threshold to estimate the Essential and Homography matrices 2.5, which allowed us to reproduce more or less the same output in consequential runs of the code for all the datasets.

Individual Contributions

We worked together on the entire project through meetings.

Results & Custom Dataset

Our camera calibration parameters are stored in the file `custom_cam.yml` inside the `datasets/` folder. For our dataset, we collected 22 images of a little model of a truck (`custom_dataset` folder contained in `datasets/`) from different points of view on a white textureless background to better extract the features. We rescaled the images from 4000x3000 to 1133x850 and we used 1.1 as focal length scale when computing the matches. The point cloud of our custom dataset can be found in the `outputs/` folder, saved as `cloud_truck.ply` as well as the other output files. Here are the obtained point clouds:

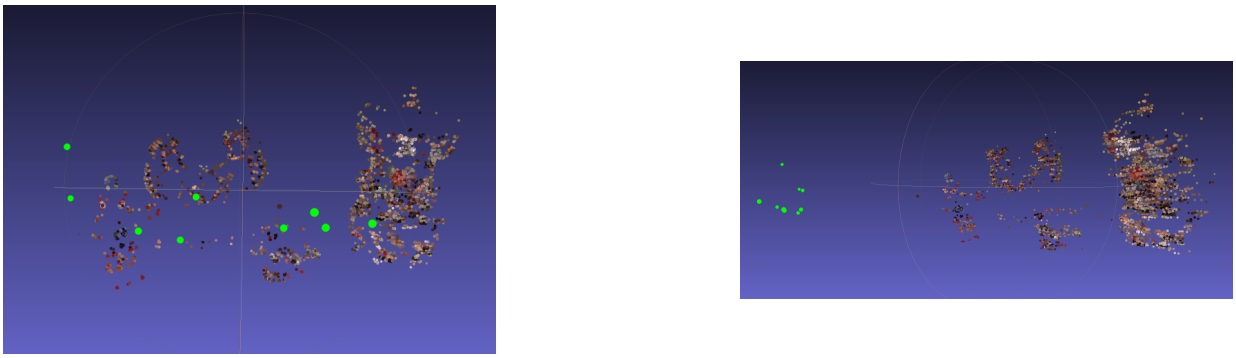

Figure 4: Image 1 point cloud

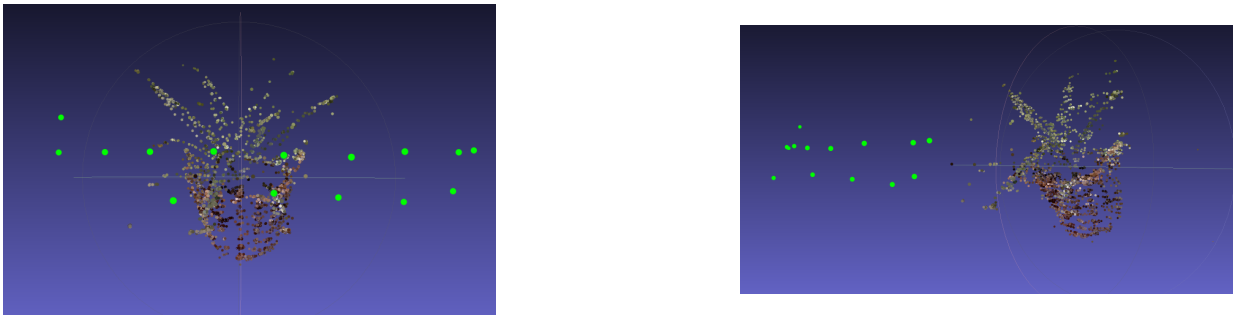

Figure 5: Image 2 point cloud

Figure 6: Custom dataset point cloud