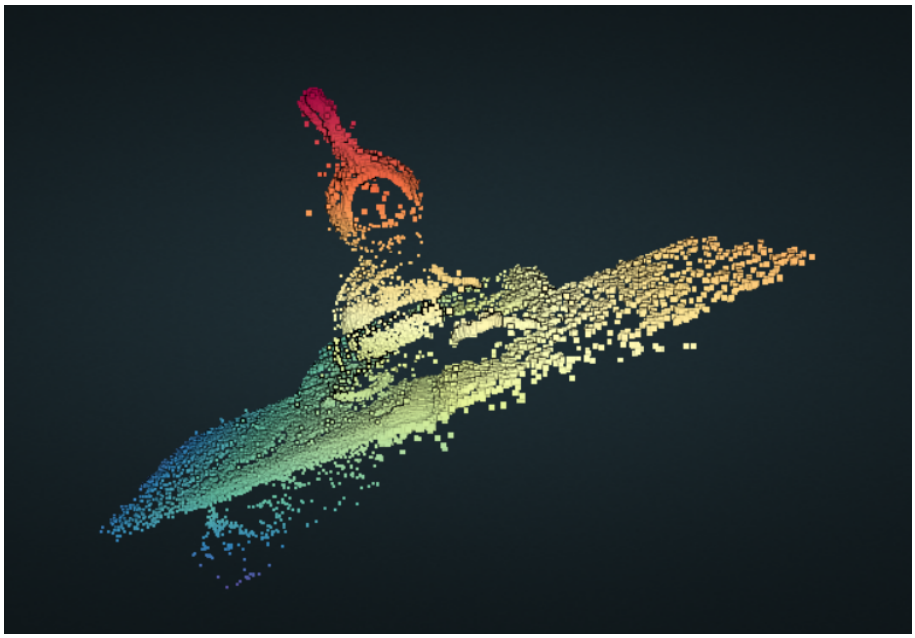
James Chung
A15852697

Final Project Writeup

Introduction

In this writeup, I will briefly explain the steps I took to achieve my final visualization. Much of the work used was from the tutorials mentioned in the notebook, so I will only really describe what changes I made from the tutorial for my implementation. For easy reference, here is what my final visualization looks like (will include at the end of this writeup as well):



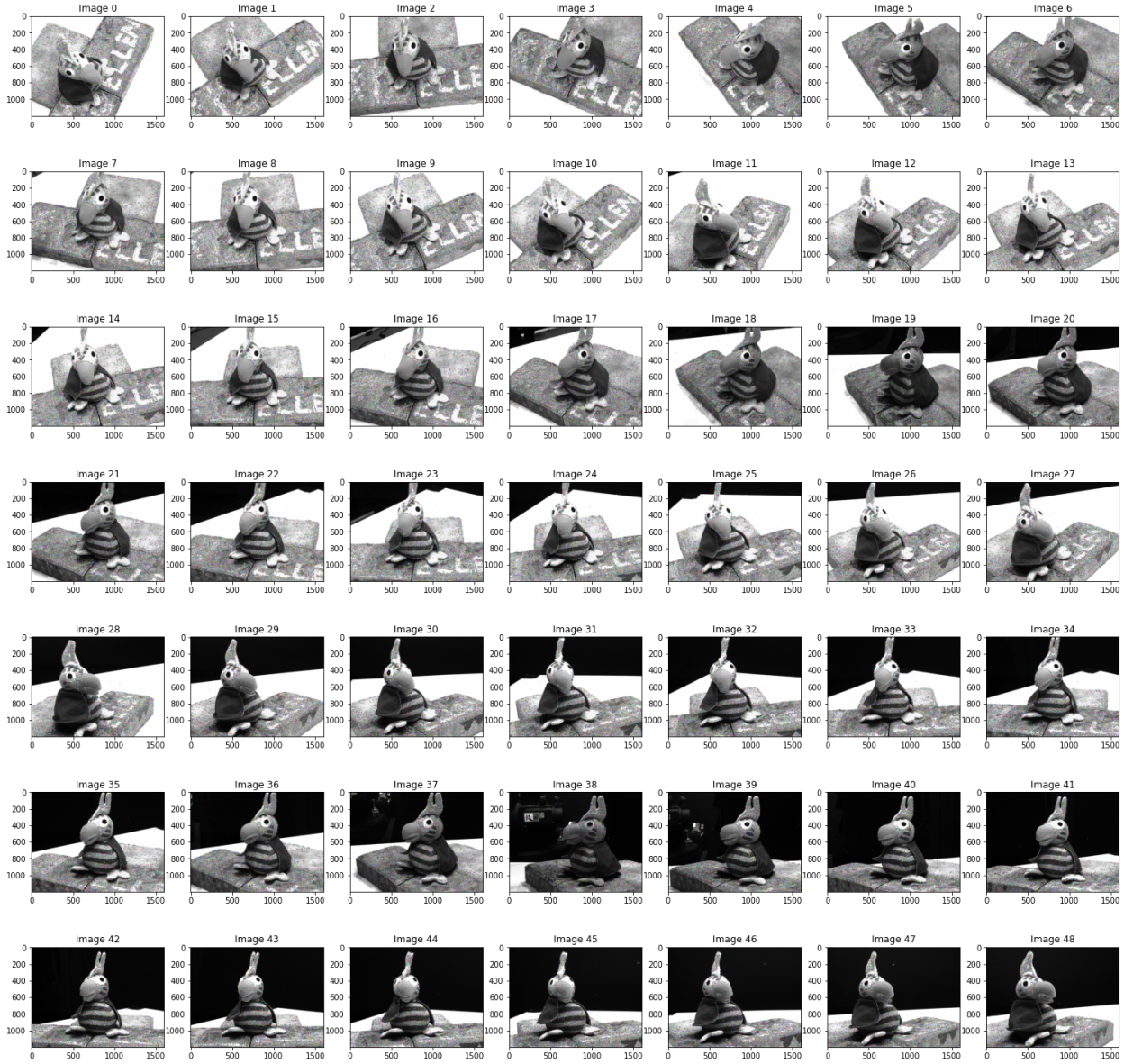
Part 1

Just preparation, nothing of note in this part.

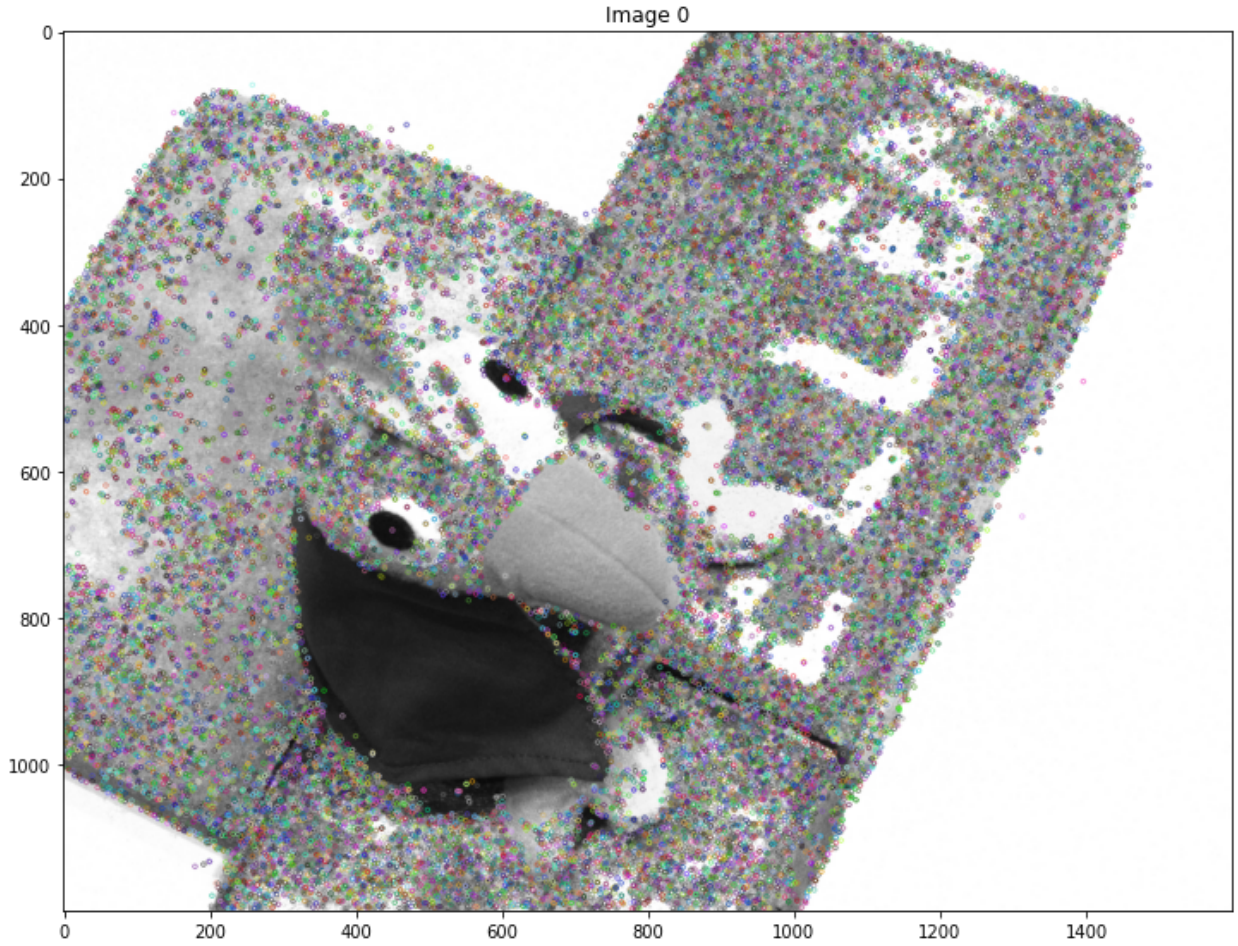
Part 2 (Points of Interest)

Following the tutorial code from the OpenCV docs, all I really had to do was make sure I was capturing all of the output correctly. Each image has their own computed keypoints and descriptors (named `kps, des`), which are appended to the final arrays that hold keypoints and descriptors of all images, `keypoints, descriptors`. The corner detection visualizations from `cv.drawKeypoints` are also appended to `outputs` to draw them later.

Here is the output after finding points of interest on all images:



It is hard to tell if the keypoints are there, so here is one example image (image 0) with its drawn keypoints:

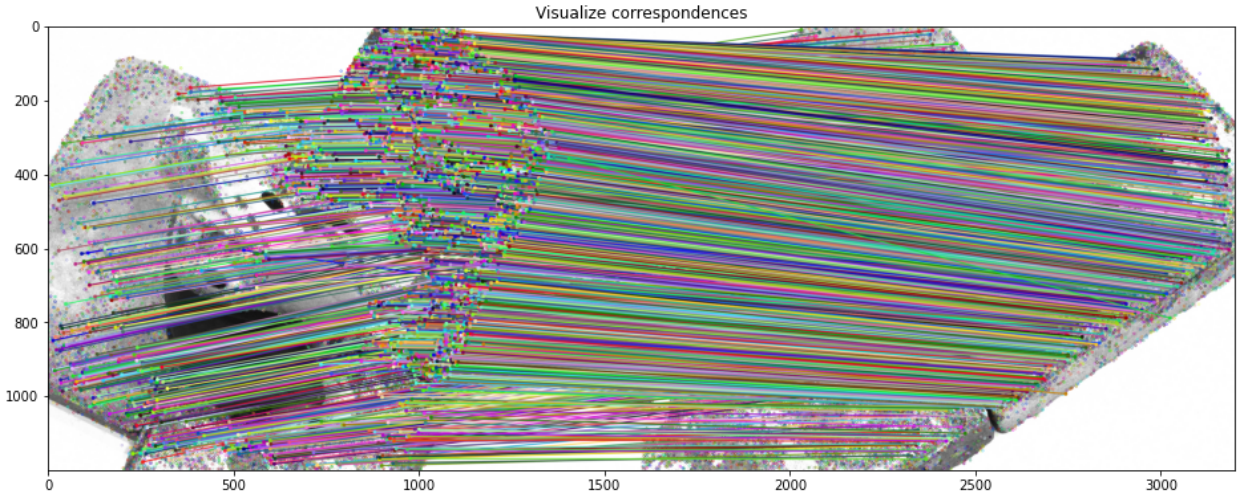


Part 3.1 (Point Correspondences)

Using the tutorial code for brute force matching for SIFT descriptors from the openCV docs, I just needed to get the appropriate output. The `good` array that holds all matches (not [matches] like in the tutorial) passing the ratio test consists of Matches `m` such that the index of the corresponding points in each image can be accessed by `m.queryIdx, m.trainIdx` (for `pts1, pts2` respectively). So for each match, I got the corresponding keypoints in each image and reshaped them to be 2x1 numpy arrays, appending them to pts1, pts2, which results in the desired 2xn arrays.

For the `ratio` threshold in the ratio test, I tested low values (0.25) to check that the correspondences made sense, and then I found through trial and error that `ratio=0.6` seems close to reference outputs from discussion.

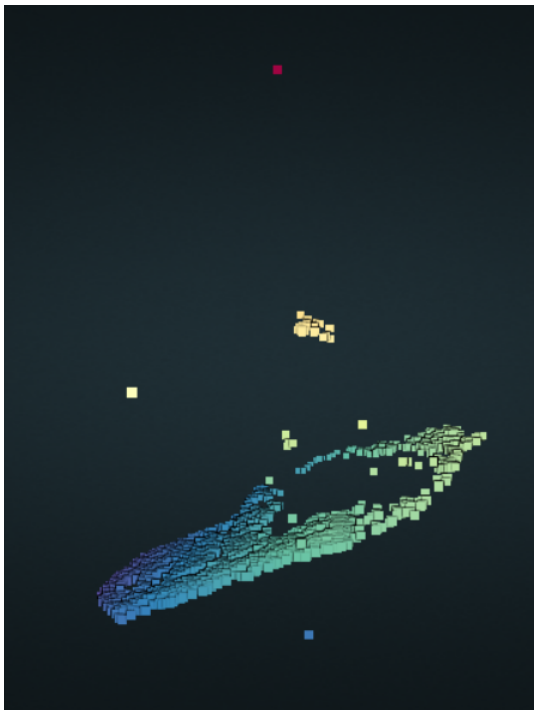
Here is the picture of correspondences:



`Number of correspondences derived: 2843`

Part 3.2 (Triangulation)

Triangulation is mainly done just using openCV's `triangulatePoints()`, but the projection matrices of the two cameras are necessary. So I simply computed the projection matrices using the cameras' intrinsic, extrinsic matrices, then called `triangulatePoints()` (all in homogeneous coords). The output will be used to draw on a Euclidean plane, so I just use `from_homog()` on the result to convert back to Euclidean coords.
Here is the resulting visualization (rotated and zoomed a bit):

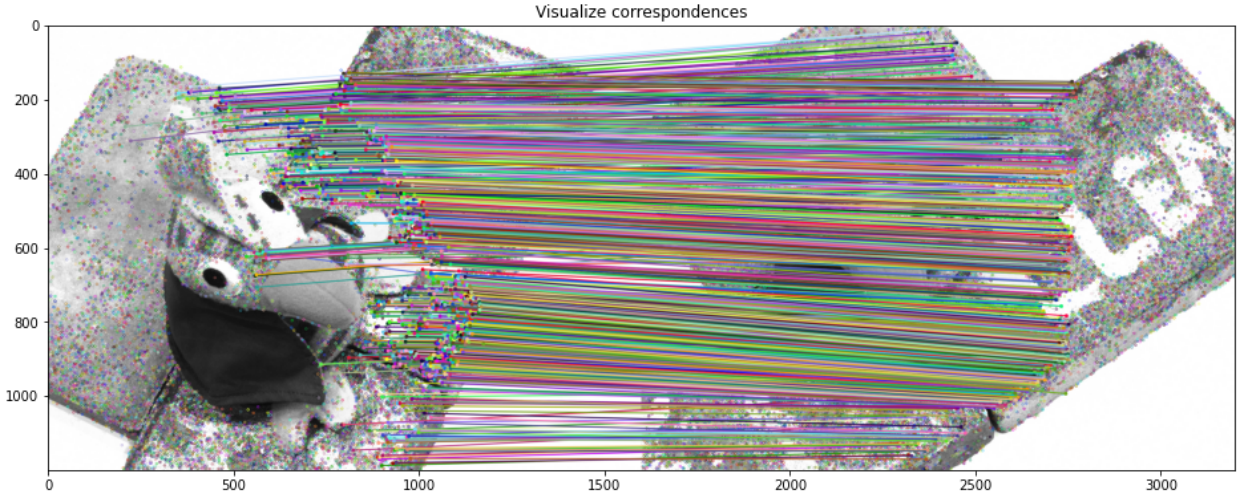


Part 4.1 (Validate Correspondences)

The general process is to compute the fundamental matrix and see if correspondences meet the epipolar constraint $||p_1^T F p_2||$ for some threshold. We need the essential matrix $([t]_\times R)$, so I need to compute the composite extrinsic matrix using the two camera's extrinsic matrices, which I use as `ext_mat`. Then the translation and rotation components are extracted to build the essential matrix and because we are given the intrinsic matrices, the fundamental matrix can directly be computed. For removing outliers, I just took corresponding points, evaluated their epipolar constraint, and marked the points as inliers if the constraint is less than a `percent` of the image's height/width. The results are the remaining inliers.

I found through experimenting with the base 1% that changing it slightly does not make much of a perceivable difference in the final visualization, so I just used .95% arbitrarily.

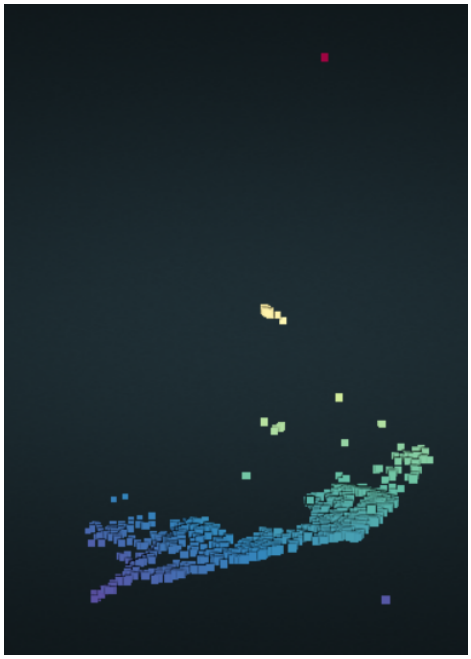
Here is the visualization of correspondences:



`Number of correspondences after validation: 1071`

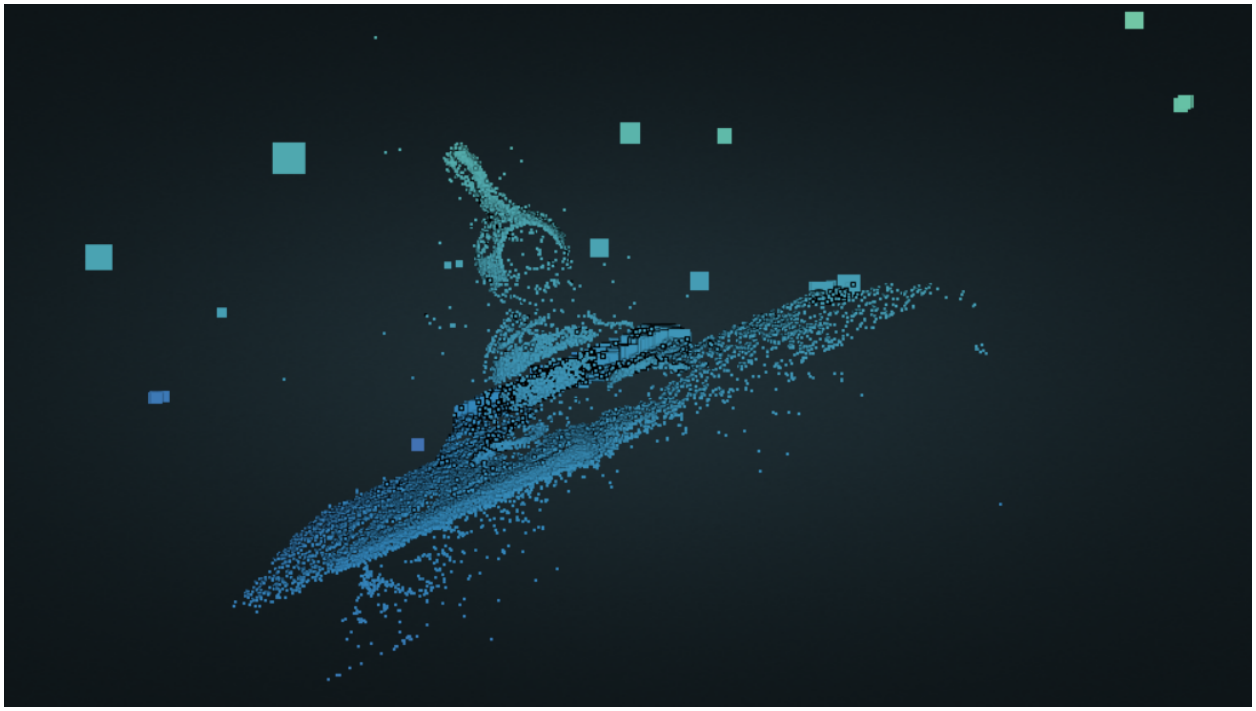
(# of correspondences has gone down as expected)

And the new reconstruction:



Part 4.2 (All Cameras)

Combining all the camera correspondences to recreate the image, I get:



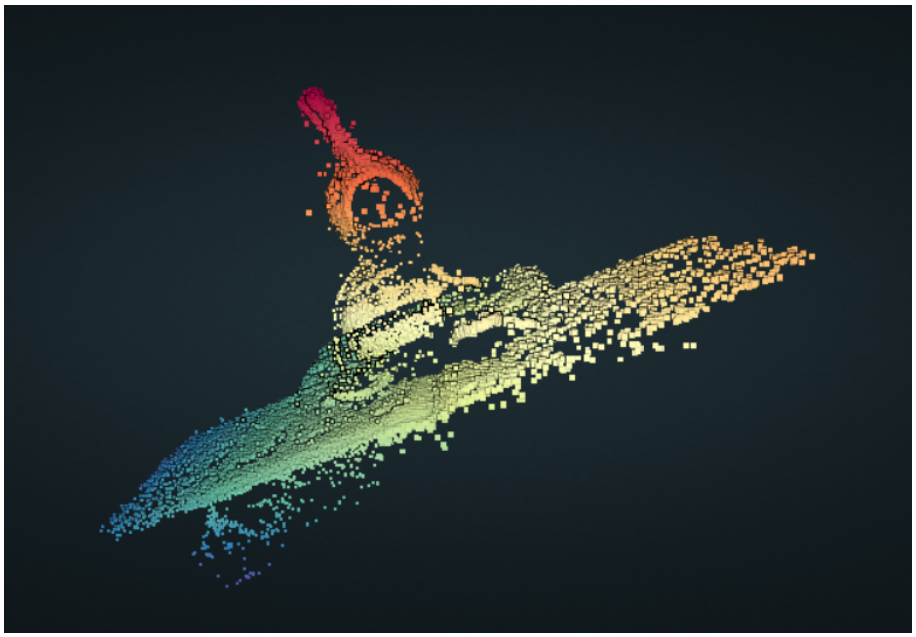
Obviously, this is very noisy and required a lot of zooming in because of extreme outliers!

Part 5 (Optimization)

To clean up the noise/outliers, I use the statistical outlier removal function provided in Open3D. In order to pass in the reconstruction into Open3D to call `remove_statistical_outlier()`, I need to give it the set of points in the reconstruction `recon`. So I just created an Open3D PointCloud `pcd` and assigned its points to the *transpose* of `recon`. I use the transpose because PointCloud expects an array of shape (n, 3), not (3, n). I call the outlier removal function with appropriate parameters and then convert back the PointCloud `res` to a Numpy array `recon_cleaned`.

I mentioned "appropriate parameters" `nb_neighbors, std_ratio`. I found the example parameters `nb_neighbors=20, std_ratio=2.0` was not appropriate for my reconstruction as there were still many outliers. In general, I found that increasing `nb_neighbors` helped remove the "far"/extreme outliers and decreasing `std_ratio` was better for cleaning up closer/minor outliers. However, decreasing the standard deviation ratio too much would be too aggressive, removing many points for the flat surface part of the image. So I settled on `nb_neighbors=40` and `std_ratio=0.4`.

Here is the visualization of `recon_cleaned` (rotated appropriately to match the ideal reference:



This looks a lot better, and in fact, is almost identical to the reference result!

References

https://docs.opencv.org/4.5.4/da/df5/tutorial_py_sift_intro.html
https://docs.opencv.org/4.5.4/dc/dc3/tutorial_py_matcher.html
https://docs.opencv.org/3.4/d0/dbd/group__triangulation.html
http://www.open3d.org/docs/latest/tutorial/Advanced/pointcloud_outlier_removal.html
http://www.open3d.org/docs/0.9.0/tutorial/Basic/working_with_numpy.html