Programming Assignment 3

Macy Toppan

1 Information

Worked with Julian Wu and Emilie Hopkinson.

2

2.1

To understand the implementation of eight-point, reference the following step-by-step (inspired by the instructions for the assignment and the course slides).

- Divide each coordinate by M using transformation matrix T.
- Construct the M x 9 matrix A: xx' xy' x yx' yy' y x' y' 1
- Find the SVD of A.
- Get the column of V with the least singular value and reshape to F.
- Decompose F with SVD to get the three matrices, then set the smallest singular value in sigma to zero for sigma prime, then F' = u sigma prime × v.transpose.
- Call refineF...
- ...then unscale!

```
F: [[ 2.52874524e-09 -5.60294317e-08 -9.27849009e-06]
 [-1.33006796e-07  7.08991924e-10  1.12443633e-03]
 [ 2.81490965e-05 -1.08098447e-03 -4.51123569e-03]]
```

FIG. 1: A screenshot of the fundamental matrix from the terminal.

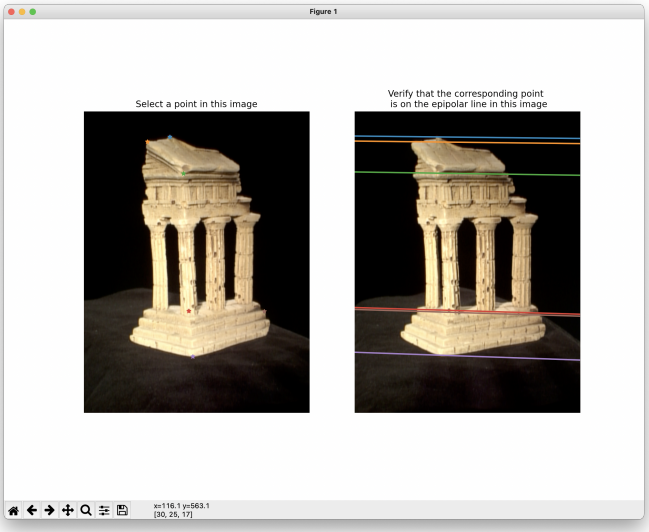

FIG. 2: A visualization of the epipolar lines resulting from the eight-point algorithm and generated using the displayEpipolarF helper function.

2.2

I used the recommended 'window-ing' metric for similarity combined with a finite search region for each point to locate the closest point within a given threshold distance from the original. To do this, I set a value for the larger window, then set another for the 'kernel'. This logic stems from the thought that, for the function to work in the first place, the objects in each image should sit within a fair region of each other without so much rotation to make corresponding points unrecognizable. It also is rooted in the observation, on early iterations, of incorrect points chosen significantly further along the epipolar line. By forcing the function to identify similar points within a given window, we decrease the likelihood of significantly different points being mistaken for the corresponding epipolar point. To evaluate the similarity of those points, we use the kernel to calculate the Euclidean distance of each point from the original, updating a value with the shortest distance as we go then storing that after applying the kernel to all values in the window.
This tactic is quite effective, especially on notable features and corners: however, it falls a bit flat for regions without significant color variation. Without a distinctive focal feature to work from, the algorithm struggles to precisely identify the correct solution. The window assists with this but without clear distinction on the selected object region, mistakes still occur.

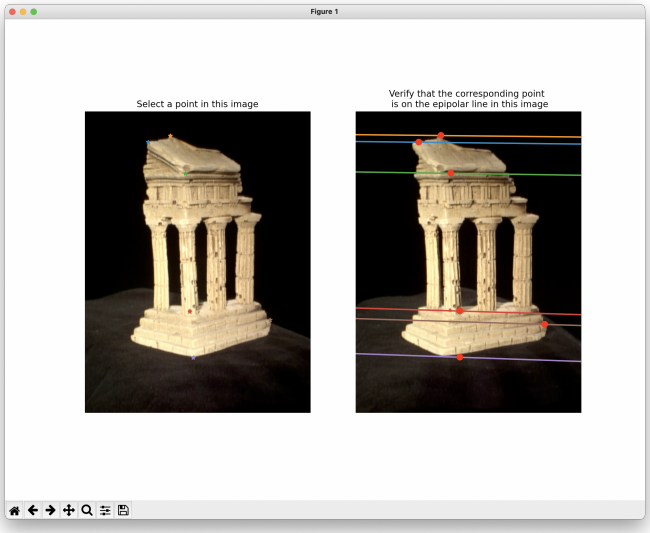

FIG. 3: A visualization of the epipolar lines and points resulting from the epipolar correspondences algorithm and generated using the epipolarMatchGUI helper function.

2.3

The extrinsic matrix is a single line! It can be calculated using the formula given in the slides: $E = K_2^T F K_1$.

```
E: [[ 5.84548837e-03 -1.29987069e-01 -3.39748366e-02]
 [-3.08572889e-01  1.65079610e-03  1.65468710e+00]
 [-5.96270630e-03 -1.67505406e+00 -1.91346162e-03]]
```

FIG. 4: A screenshot of the essential matrix from the terminal.

2.4

To calculate the correct extrinsic matrix, we need to find the matrix with the most non-negative z-values. To do this, we start by setting up variables to keep track of the highest number of positive points produced by a P2 as well the matrix that produced them. We then loop through each P2 possibility returned by the helper function *camera2* (by the third index, or z function, to find the x- and y-values). We isolate each matrix from the others using that Z value and use it (multiplied by the intrinsic matrix K) in triangulation to generate a set of three dimensional points. Finally, we count up the points with z-value greater than zero and, if the results are than the existing best, we update the best count tracker and best P2 accordingly.

This works well: however, the resulting reprojection error leaves something to be desired. It's closer to 15 total. However, pixel-to-pixel, the reproduction error is closer to about 0.05. This is consistent across all P2 possibilities.

```
Option 1:  [[ 9.94148964e-01  1.66606129e-02  1.06725165e-01  1.00000000e+00]
 [ 1.64847593e-02 -9.99860917e-01  2.52976187e-03  2.04427696e-02]
 [ 1.06752469e-01 -7.55621481e-04 -9.94285341e-01 -7.75815690e-02]]
Option 2:  [[ 9.94148964e-01  1.66606129e-02  1.06725165e-01 -1.00000000e+00]
 [ 1.64847593e-02 -9.99860917e-01  2.52976187e-03 -2.04427696e-02]
 [ 1.06752469e-01 -7.55621481e-04 -9.94285341e-01  7.75815690e-02]]
Option 3:  [[ 0.9656441  -0.0240544   0.25875252  1.        ]
 [ 0.02357884  0.99970977  0.00494161  0.02044277]
 [-0.25879629  0.00132924  0.96593101 -0.07758157]]
Option 4:  [[ 0.9656441  -0.0240544   0.25875252 -1.        ]
 [ 0.02357884  0.99970977  0.00494161 -0.02044277]
 [-0.25879629  0.00132924  0.96593101  0.07758157]]
```

FIG. 5: A screenshot of the possible P2 matrices from the terminal.

2.5

Looks pretty good, right? It took a while to get to this point— for a while, the points visualized clung to a plane through the space rather than taking on semi-accurate dimensionality. The final result is still a bit messy, but (thanks to changes to epipolar_correspondences) it takes on the general shape of the object itself well.

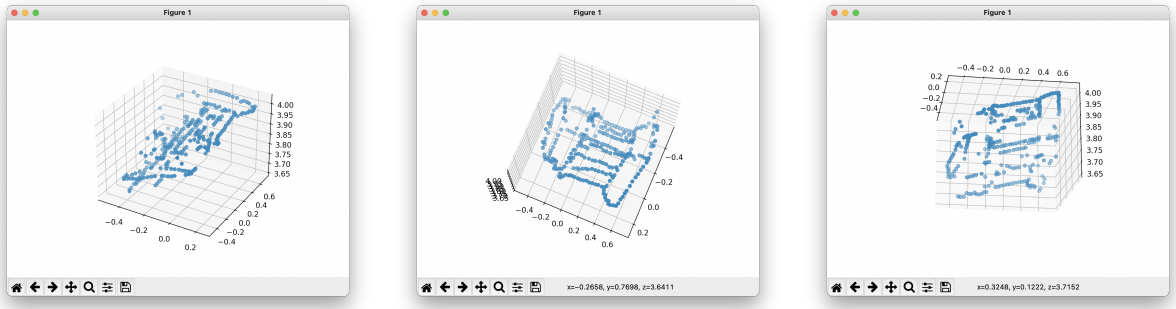

FIG. 6: Results of visualizations of the triangulate function, reconstructing the epipolar points into an approximation of the temple's form.

3

3.1

This draws heavily from the given instructions. It crops slightly more than desired, but as indicated in the Slack channel, this is not a significant issue since the points are correct and the lines are not angled.

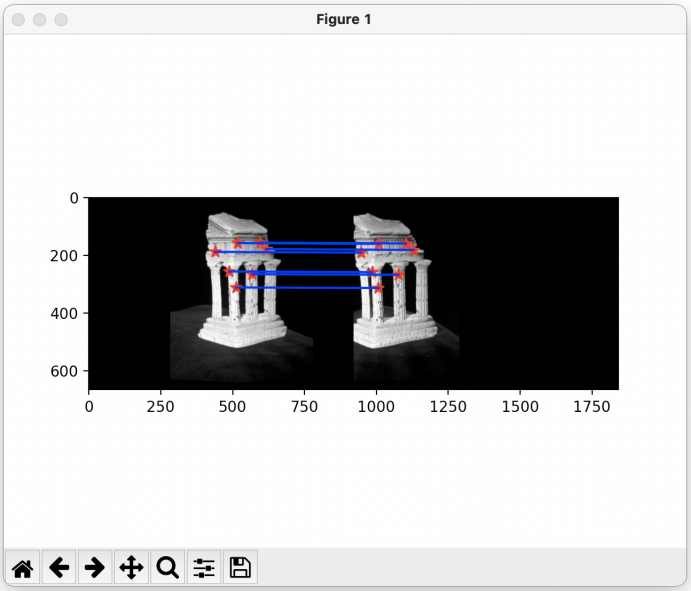

FIG. 7: The results of test_rectify— note those lines, parallel to the x-axis!

3.2

Although I attempted to use the recommended function to vectorize (scipy.signal.convolve2d), I found that my use of it produced incorrect results. On the other hand, the mess of nested loops needed to manually calculate the desired sums and compare them is more precise, although it takes a slightly massive amount of time to calculate. Using a try-except method here ensures the validity of the points used in the ultimate calculation of shortest distance.

3.3

These images are quite similar to those sent in the slack to use as reference for 'correct' results. Though these are slightly noisier, that difference is likely the result of a different implementation of epipolar correspondences, resulting in a different triangulation that led to this point. This noise doesn't obstruct the overall success of the disparity and depth images in their representations.

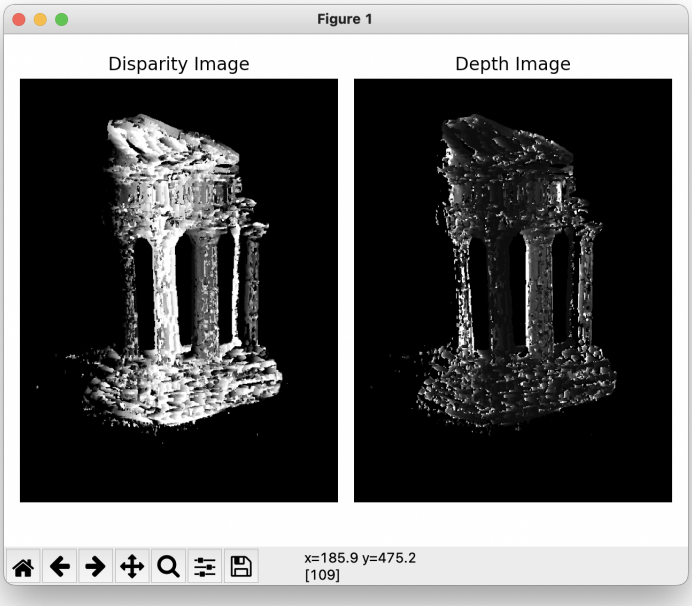

FIG. 8: Disparity and depth maps calculated using dispM, depthM, and test_depth.

4

4.1

The output of test_pose is below. Though the reprojection error is a bit high with noisy points, it's generally extremely small— see the clean point results for reference.

```
Reprojection Error with clean 2D points: 5.728186255716239e-10
Pose Error with clean 2D points: 5.006878155542038e-12
Reprojection Error with noisy 2D points: 8.873574167746868
Pose Error with noisy 2D points: 0.9932684569293677
```

FIG. 9: The results of test_pose quantifying the accuracy of matrix P calculations.

4.2

As recommended, the reprojection error is in the sweet spot, hovering just below 2.14!

```
Intrinsic Error with clean 2D points: 2.000000000001494
Rotation Error with clean 2D points: 2.82842712474619
Translation Error with clean 2D points: 5.913131730322363
Intrinsic Error with noisy 2D points: 2.090160542592896
Rotation Error with noisy 2D points: 2.828426371324767
Translation Error with noisy 2D points: 5.861988810201148
```

FIG. 10: The results of test_params quantifying the accuracy of extrinsic and intrinsic matrix calculations (K, R, t).