Feature Matching:
For visualizing the feature matching, since the matches were already given the points so all we had to do was plot them. For plotting them, we first got the w and h of the two images, made a new image, copied the images over, and then plotted it. For the lines we used plt.plot with the appropriate offsets.

Fundamental Matrix:
For estimating the fundamental matrix, we first convert the points to homogeneous points using np.hstack. Then we get the appropriate x and y values and then build the A matrix as:

```
A = np.column_stack([
    x1 * x2, y1 * x2, x2,
    x1 * y2, y1 * y2, y2,
    x1, y1, np.ones_like(x1)
])
```

At first, we were confused about building the A matrix and got it wrong, but after a few dozen trials we stacked it right. Next, we got the SVD using np.linalg.svd and then enforced rank 2 by setting the last singular value to 0 since if it had full rank then it wouldn't have had any epipoles. I also built a pytorch version for fun (not included) to use the GPU instead, seeing a time improvement of 10-15x with a batch size of 8192 on a RTX 4050.

RANSAC:
Being the third time that RANSAC was being used, we just went straight ahead and implemented it. We initialized all the necessary variables such as the best_inliers, best_F, etc., converted the points to the appropriate format, generated enough random iterations, then iterated through them, estimated the F matrix, counted error and inliers, and updated accordingly. We made several optimization improvements such as using vectorized operations, using masking instead of calculating in every iteration, and more.

Epipolar Lines:
This part was basically just plotting with the extra step of converting to homogeneous coordinates by adding 1s using np.hstack and computing the lines using the fundamental matrix by performing np.dot on F and coordinates.T and taking the T. Plotting took a little while but basically, for each line, it calculates x0 as starting from 0 and y0 using -c/b and x1 as ending of x axis and y1 using -(c + a * img2.shape[1]) / b which offsets accordingly. We then use cv2.line to plot the line.

Essential Matrix:
For estimating the essential matrix, our method took in the input of the two intrinsic matrixes for their respective cameras and the fundamental matrix. The essential matrix will allow us to find relative camera poses between the images. We calculated it as the intrinsic matrix 2 transposed multiplied by Fundamental multiplied by intrinsic matrix 1. Since there is going to be noise in our intrinsic matrix we can perform a singular value decomposition. Then S, V of the decomposition and multiply it by the identity matrix with an empty row. After this we can output the product to be our essential matrix, we also attempted to normalize the results of the essential matrix by np normalize. However our values were already so small it did not make a large difference in the output of our 3d points.

```
Essential Matrix
[[ 0.01782954 -0.02167986 -0.18114657]
 [ 0.13654521  0.04876278 -0.97280024]
 [ 0.22989948  0.96981408  0.07511446]]
```

Rotational & Translation Sets
Now by funneling our output of the essential matrix into get_RTSet we can extract the many different possible poses of the camera. This method will only take in our essential matrix and then output 4 3 by 3 rotation matrix as well as 4 3 by 1 translation matrices which have the x,y,z shifts. It is for 4 output sets of rotation and translation extracting from the essential matrix. Then we need to perform singular decomposition and then calculate the possible configurations from them. One key distinction here is when performing the matrix multiplication if the determinant of that output is -1 then you must inverse the camera pose by -1. For our input images we calculate the possible translation and rotation sets from the camera poses we set up to be captured in the essential matrix.

```
ROTATION SET
[array([[ 0.90132397, -0.41062197,  0.13785754],
       [-0.39950019, -0.91106962, -0.10174352],
       [ 0.16737594,  0.03662976, -0.98521244]]), array([[ 0.90132397, -0.41062197,  0.13785754],
       [-0.39950019, -0.91106962, -0.10174352],
       [ 0.16737594,  0.03662976, -0.98521244]]), array([[ 0.99287235, -0.05718361,  0.10456831],
       [ 0.05140014,  0.99703914,  0.05719246],
       [-0.10752917, -0.05140999,  0.99287184]]), array([[ 0.99287235, -0.05718361,  0.10456831],
       [ 0.05140014,  0.99703914,  0.05719246],
       [-0.10752917, -0.05140999,  0.99287184]])]
TRANSLATION SET
[array([[-0.98305545],
       [ 0.18065796],
       [-0.03105945]]), array([[ 0.98305545],
       [-0.18065796],
       [ 0.03105945]]), array([[-0.98305545],
       [ 0.18065796],
       [-0.03105945]]), array([[ 0.98305545],
       [-0.18065796],
       [ 0.03105945]])]
```

The last two but very important steps include triangulation of camera matrices and confirming the chirality condition. To approach triangulation, we had to construct the two camera matrices using the R set and T set. This was done using the supplemental slide deck provided on Piazza. After constructing these matrices, we use these to solve for x in the equation Ax = 0, where A is a 4x4 matrix. This provides a set of 3D points that we can then use for each (R, T) pair.

Chirality condition is a geometric principle that ensures that 3D points reconstructed from the stereo image pairs lie in front of the camera, and that was the next step now that we have our sets of 3D points. Essentially, what we need to find is the best combination of rotation and translation (hence pose) that satisfies the condition. You compute the camera center using the equation C = -R^T * T, and you use this along with the 3D points and the third row of the rotation matrix to compute chirality. Depth is computed using the equation (camera coordinate, which is)*(R_3)^T. We then select the pose with the maximum number of valid 3D points in front of the camera (i.e. where depth is positive/greater than zero).

Extra credit:

We started off by looking into COLMAP, the parent project of the pyCOLMAP bindings as provided. After several trials and looking at example.py and README.md and trying to figure out how it works, we finally got the python bindings working. We first started off by taking pictures of an xbox controller, however the images proved to be inadequate. So we then moved on to taking pictures of a pineapple but that didn't work well either. Finally, we used a bunch of bananas, set them in adequate lighting conditions, and took pictures in RAW format and JPEG format with manual mode of a Google Pixel phone with a focal length of 6.90mm and f/1.7. We set the ISO at minimum (44) and set the shutter speed to ¼ and used a stand to avoid shaking. The focus was set to auto. For maximum quality, the pictures were taken using the 50MP mode. Unfortunately, COLMAP does not accept RAW images so as a quick (and lazy) solution, we just used the JPEG images. This probably ruined half the efforts of getting the best possible quality but it worked and we ended up with a .ply file. To show the camera locations, we redid it with the GUI version instead. Finally, we used Meshlab to show the mesh generated. While I'm unhappy with the quality, this extra credit part has kindled an interest in me, and in my future efforts I'll be attempting this with my own code equipped with a nice mirrorless camera.