Carnegie Mellon University
16-720: Computer Vision
Homework 4: 3D Reconstruction

- **Due date.** Please refer to course schedule for the due date for **HW4**.
- **Gradescope submission.** You will need to submit both (1) a pdf of your python notebook submission and a zip file containing your notebook code, either as standalone python (`.py` or colab notebooks `.ipynb`). Remember you *must* use Gradescope's functionality to mark pages in your PDF that answer individual questions; if not, you will risk losing points!.
- **Suggestions for creating a PDF.** We suggest you create a PDF by `print`ing your colab notebook from your browser. However, you are responsible for making sure all your code is visible and not cut off. Long lines of code can print poorly; we suggest you add a backslash to break a single long line into multiple lines. You also may wish to look at this video for alternate pathways to convert notebooks to PDFs: https://youtu.be/-Ti9Mm21uVc?si=bo4kHfp2BoJvPpZI. In some cases, you may wish to download image or screengrabs and explicitly append them to your PDF using online tools such as https://combinepdf.com/. You may also find it useful to look at this post: https://askubuntu.com/questions/2799/how-to-merge-several-pdf-files.

The starter code can be found at the course gdrive folder (you will need your andrew account to access):
https://drive.google.com/drive/folders/1JZGnpUG6Ca1o0q47PCxsQVQASQwJTmUu

We recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead.

Please remember to list your collaborators in your report.

Overview

In this assignment you will be implementing an algorithm to reconstruct a 3D point cloud from a pair of images taken at different angles. In Part I you will answer theory questions about 3D reconstruction. In Part II you will implement the 8-point algorithm, triangulation, and other techniques to find and visualize 3D locations of corresponding image points.

Part I

Theory

Problem 1: Some Proofs

Before implementing our own 3d reconstruction, let's take a look at some simple theory questions that may arise. the answers to the below questions should be relatively short, consisting of a few lines of math and text (maybe a diagram if it helps your understanding).

Q1.1 [5 points] Suppose two cameras fixate on a 3d point $\mathbf{X}$ (see Figure 1) in space such that their principal axes intersect at that point. Show that if the image projection of the 3d point lies at pixel $(0, 0)$ for

both images, the $\mathbf{f}_{33}$ element of the fundamental matrix must be zero. See Figure 1 for a diagram visualizing the problem. (Hint: expand out $\mathbf{F}$, $\mathbf{x}$, $\mathbf{x}'$ into their individual elements)

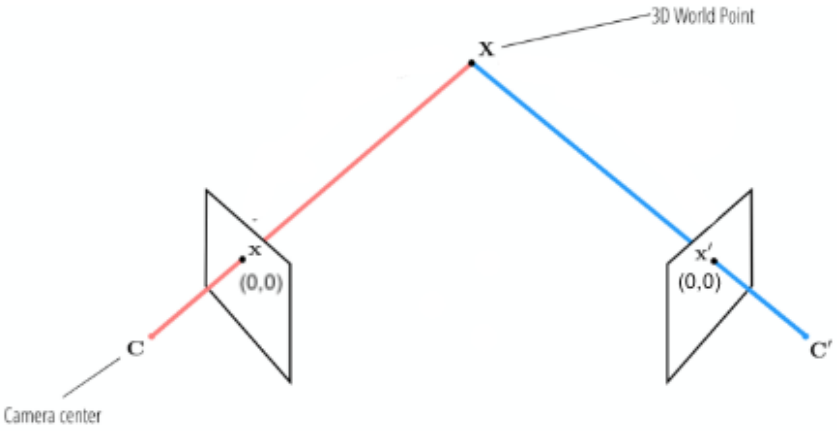

Figure 1: Figure for Q1.1. $\mathbf{C}$ and $\mathbf{C}'$ are the camera centers. $\mathbf{x}$ is the projection of the 3D point $\mathbf{X}$ onto the left image, and $\mathbf{x}'$ is the projection of $\mathbf{X}$ onto the right image, and both are (0,0) in their respective image frames. Thus the camera's principal axes intersect at 3D point $\mathbf{X}$ (which is in the global coordinate frame).

Q1.2 [5 points] Suppose a robot is moving with an inertial sensor that reports the absolute rotation $\mathbf{R}_i$ and translation $\mathbf{t}_i$ of the robot at time i. What is relative rotation ($\mathbf{R}_{rel}$) and relative translation ($\mathbf{t}_{rel}$) between time i and $i+1$? (Hint: find $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$ in terms of $\mathbf{R}_i$, $\mathbf{R}_{i+1}$, $\mathbf{t}_i$ and $\mathbf{t}_{i+1}$.)

Suppose the camera intrinsics ($\mathbf{K}$) are known. Express the essential matrix ($\mathbf{E}$) and the fundamental matrix ($\mathbf{F}$) in terms of $\mathbf{K}$, $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$.

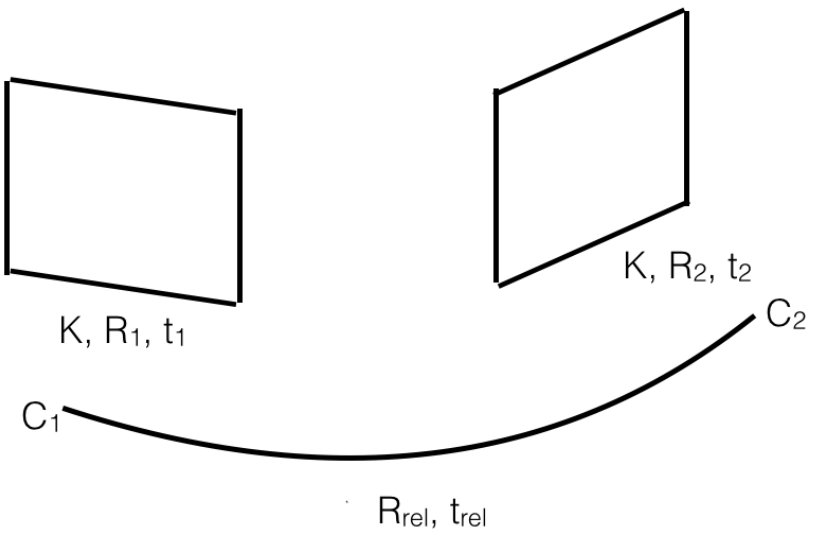

Figure 2: Figure for Q1.2. $C1$ and $C2$ are the camera centers. K is the camera intrinsic matrix. R_1 and t_1 are the rotation and translation of the robot in global coordinate frame at *e.g.* timestep 1, and R_2 and t_2 are the rotation and translation of the robot in global coordinate frame at the next timestep 2. R_{rel} and $\mathbf{t}_{rel}$ are the relative rotation and translation between the two frames.

Part II

Practice

Overview

In this part you will begin by implementing the 8-point algorithm seen in class to estimate the fundamental matrix from corresponding points in two images. Next, given the fundamental matrix and calibrated intrinsics (which will be provided) you will compute the essential matrix and use this to compute a 3D metric reconstruction from 2D correspondences using triangulation. Then, you will implement a method to automatically match points taking advantage of epipolar constraints and make a 3D visualization of the results. Finally, you will implement bundle adjustment to further improve your algorithm.

Problem 2: Estimating the Fundamental Matrix w/ Eight-Point Algorithm

In this section you will estimate the fundamental matrix given a pair of images using the Eight-Point Algorithm. In the `data/` directory, you will find two images (see Figure 3) from the Middlebury multi-view dataset[1], which is used to evaluate the performance of modern 3D reconstruction algorithms.

The 8-point algorithm (discussed in class, and outlined in Section 8.1 of [1]) is arguably the simplest method

[1] http://vision.middlebury.edu/mview/data/

for estimating the fundamental matrix. For this section, you can use provided correspondences you can find

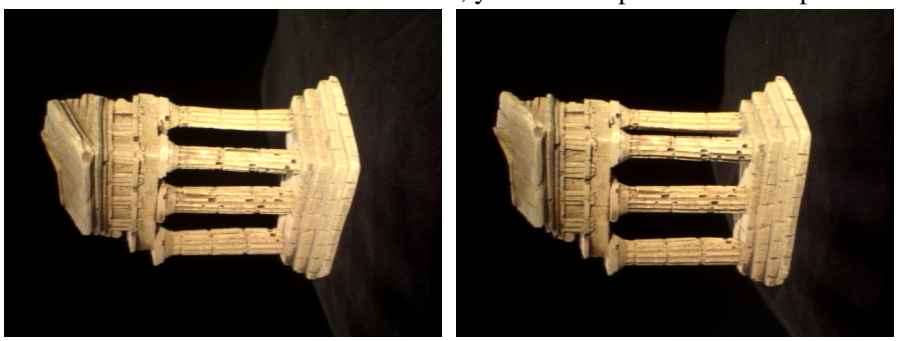

Figure 3: `Temple` images for this assignment

in `data/some_corresp.npz`.

Q2 [10 points] Finish the function `eightpoint`. In your submission, include an image that shows at least five different epipolar point-line correspondences using the provided debugging function.

```
F = eightpoint(pts1, pts2, M)
```

where `pts1` and `pts2` are $N \times 2$ matrices corresponding to the (x, y) coordinates of the N points in the first and second image respectively. `M` is a scale parameter.

- You should scale the data as was discussed in class, by dividing each coordinate by M (the maximum of the image's width and height). After computing $\mathbf{F}$, you will have to "unscale" the fundamental matrix.
  Hint: If $\mathbf{x}_{normalized} = \mathbf{T}\mathbf{x}$, then $\mathbf{F}_{unnormalized} = \mathbf{T}^T\mathbf{F}\mathbf{T}$.
  You must enforce the singularity condition of $\mathbf{F}$ before unscaling.
- You may find it helpful to refine the solution by using local minimization. This probably won't fix a completely broken solution, but may make a good solution better by locally minimizing a geometric cost function. For this we have provided a helper function `refineF` in `helper.py` taking in $\mathbf{F}$ and the two sets of points, which you can call from `eightpoint` before unscaling `F`.
- Remember that the x-coordinate of a point in the image is its column entry, and y-coordinate is the row entry. Also note that eight-point is just a figurative name, it just means that you need at least 8 points; your algorithm should use an over-determined system ($N > 8$ points).

Problem 3: Metric Reconstruction

In this problem, you will compute the projective camera matrices and triangulate the 2D point pairs from the image pair to obtain the 3D scene structure of the temple object. To obtain the 3D scene structure, first convert the fundamental matrix $\mathbf{F}$ to an essential matrix $\mathbf{E}$. Examine the lecture notes and the textbook to find out how to do this when the internal camera calibration matrices (aka intrinsic matrices) $\mathbf{K}_1$ and $\mathbf{K}_2$ are known. These are provided in `data/intrinsics.npz`.

Q3.1 [5 points] Complete the function `essentialMatrix` to compute the essential matrix $\mathbf{E}$ given $\mathbf{F}$, $\mathbf{K}_1$ and $\mathbf{K}_2$ with the signature:

```
E = essentialMatrix(F, K1, K2)
```

Given an essential matrix, it is possible to retrieve the camera projection matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ from it. Recall that the camera projection matrices represent the transformation from the world frame to image frame. It is defined as $\mathbf{C} = \mathbf{K}[\mathbf{R}|\mathbf{t}]$ where $\mathbf{K}$ is the camera intrinsic matrix, and $[\mathbf{R}|\mathbf{t}]$ is the camera extrinsic matrix.

Assuming $\mathbf{M}_1$ is fixed at $[\mathbf{I}, 0]$ (this signifies that the first camera's center is at the world origin, and the camera's principal axis is aligned with 0,0 in the image frame), $\mathbf{M}_2$ can be retrieved up to a scale and four-fold rotation ambiguity. For details on recovering $\mathbf{M}_2$, see section 11.3 in the Szeliski textbook [2]. We have provided you with the helper function `camera2` which recovers the four possible $\mathbf{M}_2$ matrices given $\mathbf{E}$.

Note: The matrices M1 and M2 here are of the form: $\mathbf{M}_1 = \big[\mathbf{I}|0\big]$ and $\mathbf{M}_2 = \big[\mathbf{R}|\mathbf{t}\big]$.

Q3.2 [10 points] Using the above, complete the function `triangulate` to triangulate a set of 2D coordinates in the image to a set of 3D points with the signature:

```
[w, err] = triangulate(C1, pts1, C2, pts2)
```

where `pts1` and `pts2` are the $N \times 2$ matrices with the 2D image coordinates and `w` is an $N \times 3$ matrix with the corresponding 3D points per row. `C1` and `C2` are the 3×4 camera matrices. Various methods exist for triangulation - probably the most familiar for you is based on least squares (see [2] Chapter 7 if you want to learn about other methods).

For each point i, we want to solve for 3D coordinates $\mathbf{w}_i = \big[x_i, y_i, z_i\big]^T$, such that when they are projected back to the two images, they are close to the original 2D points. To project the 3D coordinates back to 2D images, we first write $\mathbf{w}_i$ in homogeneous coordinates, and compute $\mathbf{C}_1\tilde{\mathbf{w}}_i$ and $\mathbf{C}_2\tilde{\mathbf{w}}_i$ to obtain the 2D homogeneous coordinates projected to camera 1 and camera 2, respectively.

For each point i, we can write this problem in the following form:

$$\mathbf{A}_i\mathbf{w}_i = 0,$$

where $\mathbf{A}_i$ is a 4×4 matrix, and $\tilde{\mathbf{w}}_i$ is a 4×1 vector of the 3D coordinates in the homogeneous form. Then, you can obtain the homogeneous least-squares solution (discussed in class) to solve for each $\mathbf{w}_i$.

Once you have implemented triangulation, check the performance by looking at the reprojection error:

$$\texttt{err} = \sum_i \|\mathbf{x}_{1i} - \widehat{\mathbf{x}_{1i}}\|^2 + \|\mathbf{x}_{2i} - \widehat{\mathbf{x}_{2i}}\|^2$$

where $\widehat{\mathbf{x}_{1i}} = Proj(\mathbf{C}_1, \mathbf{w}_i)$ and $\widehat{\mathbf{x}_{2i}} = Proj(\mathbf{C}_2, \mathbf{w}_i)$. You should see an error less than 500. Ours is around 350.

Note: `C1` and `C2` here are projection matrices of the form: $\mathbf{C}_1 = \mathbf{K}_1\mathbf{M}_1 = \mathbf{K}_1\left[\mathbf{I}|0\right]$ and $\mathbf{C}_2 = \mathbf{K}_2\mathbf{M}_2 = \mathbf{K}_2\left[\mathbf{R}|\mathbf{t}\right]$.

Q3.3 [10 points] Complete the function `findM2` to obtain the correct `M2` from the four `M2s` by testing the four solutions through triangulations.
Use the correspondences from `data/some_corresp.npz`.

Problem 4: 3D Visualization

You will now create a 3D visualization of the temple images. By treating our two images as a stereo-pair, we can triangulate corresponding points in each image, and render their 3D locations.

Q4.1 [15 points] Finish the function `epipolarCorrespondence` with the signature:

```
[x2, y2] = epipolarCorrespondence(im1, im2, F, x1, y1)
```

This function takes in the x and y coordinates of a pixel on `im1` and your fundamental matrix $\mathbf{F}$, and returns the coordinates of the pixel on `im2` which correspond to the input point. The match is obtained by computing the similarity of a small window around the (x_1, y_1) coordinates in `im1` to various windows around possible matches in the `im2` and returning the closest.

Instead of searching for the matching point at every possible location in `im2`, we can use $\mathbf{F}$ and simply search over the set of pixels that lie along the epipolar line (recall that the epipolar line passes through a single point in `im2` which corresponds to the point (x_1, y_1) in `im1`).

There are various possible ways to compute the window similarity. For this assignment, simple methods such as the Euclidean or Manhattan distances between the intensity of the pixels should suffice. See [2] chapter 11, on stereo matching, for a brief overview of these and other methods.

Implementation hints:

- Experiment with various window sizes.
- It may help to use a Gaussian weighting of the window, so that the center has greater influence than the periphery.
- Since the two images only differ by a small amount, it might be beneficial to consider matches for which the distance from (x_1, y_1) to (x_2, y_2) is small.

To help you test your `epipolarCorrespondence`, we have included a helper function that visualizes the two images and the point correspondences. This function takes in two images and the fundamental matrix, and a list of points which you specify, which you can use to test that your epipolar point and line correspondences are matching up. See Figure 4 for what it looks like. Make sure you show this image with least 5 pairs of point correspondencs in your submission.

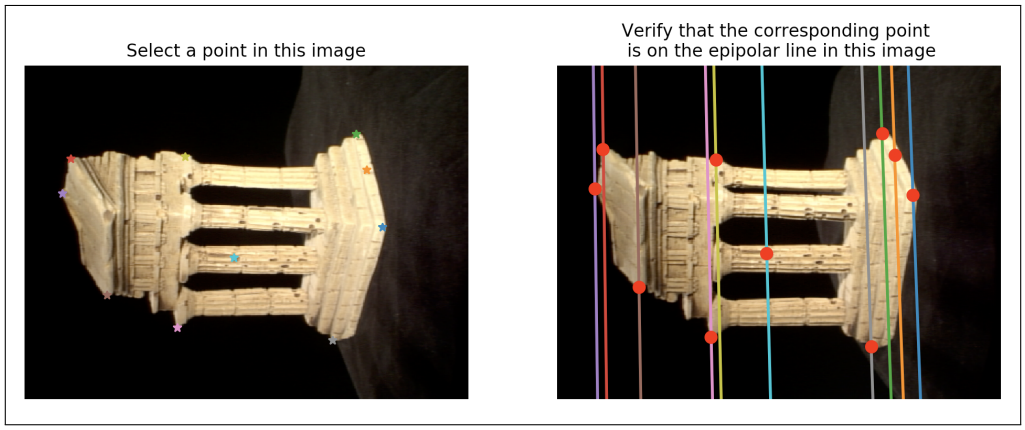

Figure 4: `epipolarMatchGUI` shows the corresponding point found by calling `epipolarCorrespondence`

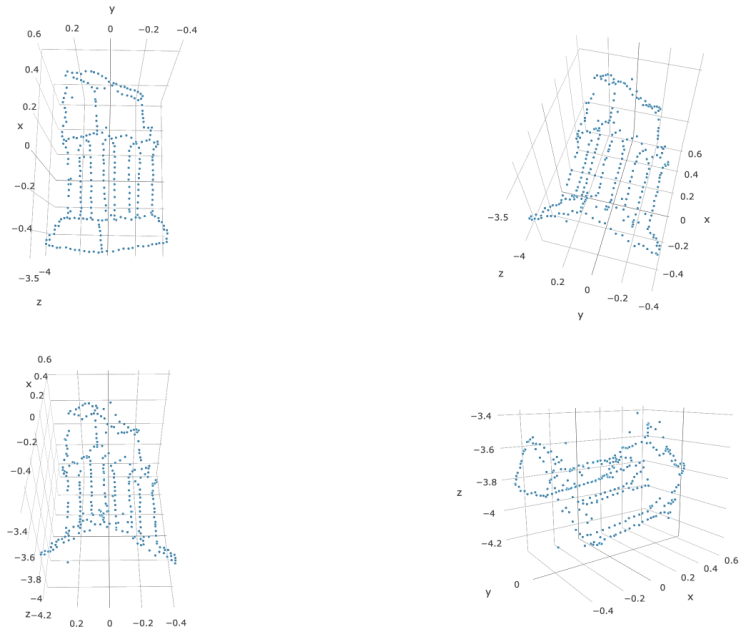

Figure 5: An example point cloud

It's not necessary for your matcher to get *every* possible point right, but it should get easy points (such as those with distinctive, corner-like windows). It should also be good enough to render an intelligible representation in the next question.

Q4.2 [5 points] Included in this homework is a file `data/templeCoords.npz` which contains 288 hand-selected points from `im1` saved in the variables `x1` and `y1`.

Now, we can determine the 3D location of these point correspondences using the `triangulate` function. These 3D point locations can then plotted using the Matplotlib or plotly package. Complete the `compute3D_pts` function, which loads the necessary files from `../data/` to generate the 3D reconstruction using `scatter` function matplotlib. An example is shown in Figure 5. Make sure you show the reconstructed 3d points in your submission.

Problem 5: Bundle Adjustment

Bundle Adjustment is commonly used as the last step of every feature-based 3D reconstruction algorithm. Given a set of images depicting a number of 3D points from different viewpoints, bundle adjustment is the process of simultaneously refining the 3D coordinates along with the camera parameters. It minimizes reprojection error, which is the squared sum of distances between image points and predicted points. In this section, you will implement bundle adjustment algorithm by yourself.

Preliminaries: RANSAC, Rodrigues, and Inverse Rodrigues In HW 4, you implemented RANSAC for more accurate estimation of the homography matrix using 4 pairs of point correspondences at a time. Provided for you is a RANSAC function which estimates a fundamental matrix using 8 pairs of point correspondences at a time (using the eightpoint algorithm you previously implemented). We will use the fundamental matrix estimation from RANSAC and our previously written `findM2` function to initialize our camera parameters before we do Bundle Adjustment.

So far we have independently solved for camera matrix, $\mathbf{M}_j$ and 3D points $\mathbf{w}_i$. In bundle adjustment, we will jointly optimize the reprojection error with respect to the points $\mathbf{w}_i$ and the camera matrix $\mathbf{C}_j$.

$$err = \sum_{ij} \|\mathbf{x}_{ij} - Proj(\mathbf{C}_j, \mathbf{w}_i)\|^2,$$

where $\mathbf{C}_j = \mathbf{K}_j\mathbf{M}_j$, same as in Q3.2.

For this homework we are going to only look at optimizing the extrinsic matrix. To do this we will be parameterizing the rotation matrix $\mathbf{R}$ using Rodrigues formula to produce vector $\mathbf{r} \in \mathbb{R}^3$. The Rodrigues formula converts a Rodrigues vector $\mathbf{r}$ to a rotation matrix $\mathbf{R}$

```
R = rodrigues(r)
```

and the inverse function that converts a rotation matrix $\mathbf{R}$ to a Rodrigues vector $\mathbf{r}$

```
r = invRodrigues(R)
```

The code for both are provided for you, but please view the pdf below for details, as you will need to understand how to use it.

Reference: Rodrigues formula and this pdf.

Q5 Bundle Adjustment [20 points]

Using this parameterization, write the objective function `rodriguesResidual`.

```
residuals = rodriguesResidual(K1, M1, p1, K2, p2, x)
```

where `x` is the flattened concatenation of $\mathbf{x}$, $\mathbf{r}_2$, and $\mathbf{t}_2$. $\mathbf{w}$ are the 3D points; $\mathbf{r}_2$ and $\mathbf{t}_2$ are the rotation (in the Rodrigues vector form) and translation vectors associated with the projection matrix $\mathbf{M}_2$. The `residuals` are the difference between original image projections and estimated projections (the square of $L2$-norm of this vector corresponds to the error we computed in Q3.2):

```
residuals = numpy.concatenate([(p1-p1_hat).reshape([-1]),
                               (p2-p2_hat).reshape([-1])])
```

Then, use this error function and Scipy's optimizer `minimize` to implement `bundleAdjustment`. This function will optimize for the best extrinsic matrix and 3D points using the inlier correspondences from

`some_corresp_noisy.npz`, using the RANSAC estimate of the extrinsics and 3D points as an initialization.

```
[M2, w, o1, o2] = bundleAdjustment(K1, M1, p1, K2, M2_init, p2, w_init)
```

Try to extract the rotation and translation from `M2_init`, then use `invRodrigues` you implemented previously to transform the rotation, concatenate it with translation and the 3D points, then the concatenate vector are variables to be optimized. After obtaining optimized vector, decompose it back to rotation using `rodrigues` you implemented previously, translation and 3D points coordinates.

Use `ransacF`, `findM2`, and your implemented `bundleAdjustment` to put it all together. Report the reprojection error with your initial $\mathbf{M}_2$ and $\mathbf{w}$, as well as with the optimized matrices. Show the before and after for BA in your writeup, a la Figure 6.

Hint: For reference, our solution achieves a reprojection error around 10 after optimization. Your exact error may differ slightly.

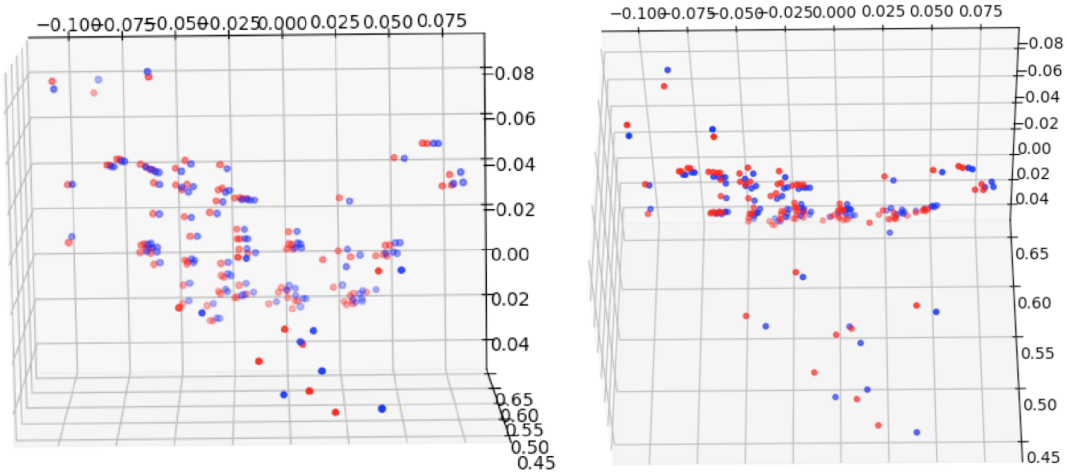

Figure 6: Visualization of 3D points for noisy correspondences before and after using bundle adjustment

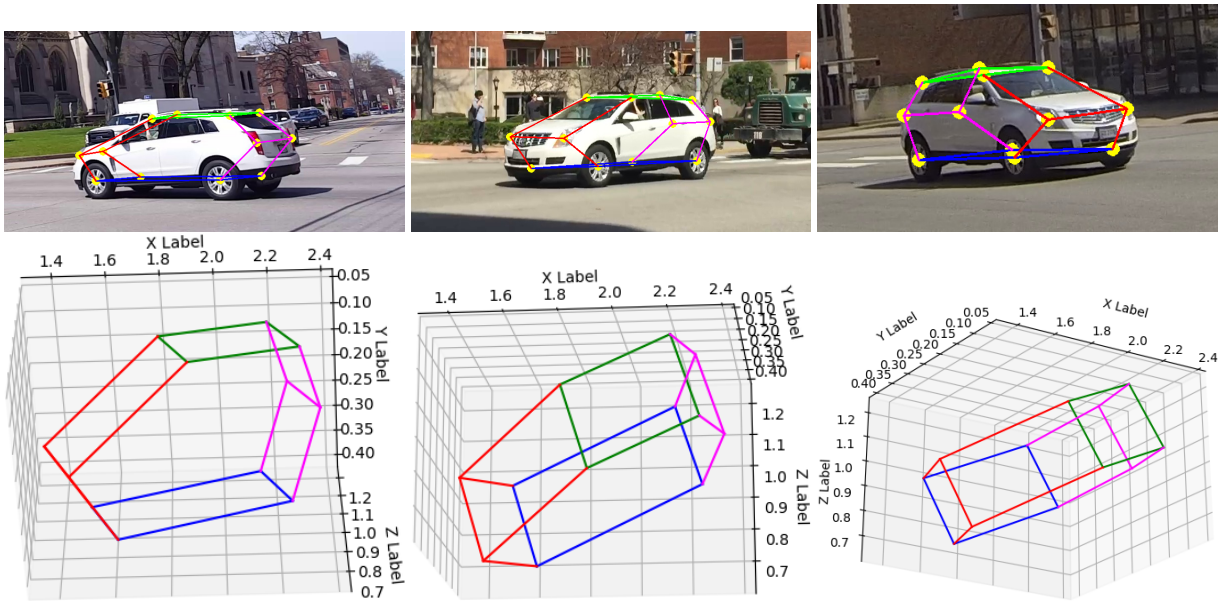

Figure 7: An example detections on the top and the reconstructions from multiple views

Problem 6: Multiview Keypoint Reconstruction (Extra Credit)

You will use multi-view capture of moving vehicles and reconstruct the motion of a car. The first part of the problem will be using a single time instance capture from three views (Figure 7 Top) and reconstruct vehicle keypoints and render from multiple views (Figure 7 Bottom). The `data/q6` folder contains the images.

Q6.1 [Extra Credit - 15 points] Write a function to compute the 3D keypoint locations `P` given the 2D part detections `pts1`, `pts2` and `pts3` and the camera projection matrices `C1`, `C2`, `C3`. The camera matrices are given in the numpy files.

```
[P, err] = MultiviewReconstruction(C1, pts1, C2, pts2, C3, pts3, Thres)
```

The 2D part detections (`pts`) are computed using a neural network[2] and correspond to different locations on a car like the wheels, headlights etc. The third column in `pts` is the confidence of localization of the keypoints. Higher confidence value represents more accurate localization of the keypoint in 2D. To visualize the 2D detections run `visualize_keypoints(image, pts, Thres)` helper function. `Thres` is defined as the confidence threshold of the 2D detected keypoints. The camera matrices (`C`) are computed by running an SFM from multiple views and are given in the numpy files with the 2D locations. By varying confidence threshold `Thres` (.i.e. considering only the points above the threshold), We get different reconstruction and accuracy. Try varying the thresholds and analyze its effects on the accuracy of the reconstruction.

Hint: You can modify the triangulation function to take three views as input. After you do the threshold lets say m points lie above the threshold and n points lie below the threshold. Now your task is to use these m good points to compute the reconstruction. For each 3D location use two view or three view triangulation for intialization based on visibility after thresholding. Describe the method you used to compute the 3D locations. Make sure to include an image of the reconstructed 3D points with the points connected using the helper function `plot_3d_keypoint(P)`. Report the reprojection error.

[2] Code Used For Detection and Reconstruction

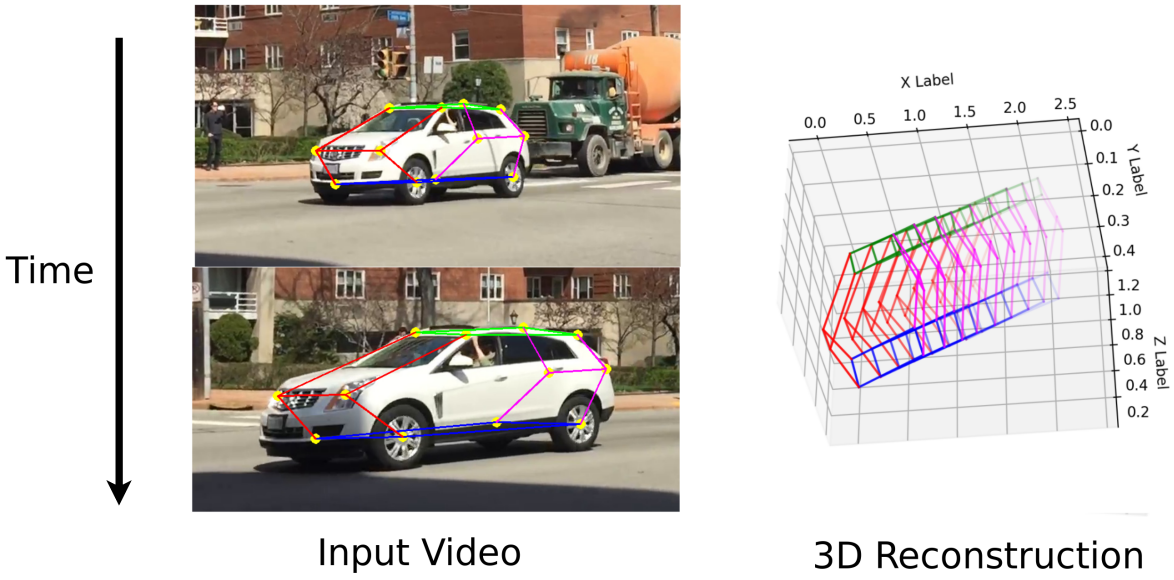

Figure 8: Spatiotemporal reconstruction of the car (right) with the projections at two different time instances in a single view (left)

Now, iteratively repeat the process over time and compute a spatio temporal reconstruction of the car. Show the visualizations of the keypoints plotted over time using the `plot_3d_keypoint_video` and the `visualize_keypoint` and the `plot_3d_keypoint` helper functions a la Figure 8 in your writeup. The images in the `data/q6` folder shows the motion of the car at an intersection captured from multiple views. The images are given as `(cam1_time0.jpg, ..., cam1_time9.jpg)` for camera 1 and `(cam2_time0.jpg, ..., cam2_time9.jpg)` for camera2 and `(cam3_time0.jpg, ..., cam3_time9.jpg)` for camera3. The corresponding detections and camera matrices are given in `(time0.npz, ..., time9.npz)`. Use the above details and compute the spatio temporal reconstruction of the car for all 10 time instances and plot them by completing the `plot_3d_keypoint_video` function. A sample plot with the first and last time instance reconstuction of the car with the reprojections shown in the Figure 8.

1 FAQs

Credits: Paul Nadan

Q2.1: Does it matter if we unscale $\mathbf{F}$ before or after calling refineF?

The relationship between $\mathbf{F}$ and $\mathbf{F}_{normalized}$ is fixed and defined by a set of transformations, so we can convert at any stage before or after refinement. The nonlinear optimization in refineF may work slightly better with normalized $\mathbf{F}$, but it should be fine either way.

Q3.2: How can I get started formulating the triangulation equations?

One possible method: from the first camera, $x_{1i} = P_1\omega_1 \implies x_{1i} \times P_1\omega_1 = 0 \implies A_{1i}\omega_i = 0$. This is a linear system of 3 equations, one of which is redundant (a linear combination of the other two), and 4 variables. We get a similar equation from the second camera, for a total of 4 (non-redundant) equations and 4 variables, i.e. $A_i\omega_i = 0$.

Q3.2: What is the expected value of the reprojection error?

The reprojection error for the data in `some_corresp.npz` should be around 352 (or 89 without using refineF). If you get a reprojection error of around 94 (or 1927 without using refineF) then you have somehow ended up with a transposed $\mathbf{F}$ matrix in your eightpoint function.

Q3.2: If you are getting high reprojection error but can't find any errors in your triangulate function?

one useful trick is to temporarily comment out the call to refineF in your 8-point algorithm and make sure that the epipolar lines still match up. The refineF function can sometimes find a pretty good solution even starting from a totally incorrect matrix, which results in the F matrix passing the sanity checks even if there's an error in the 8-point function. However, having a slightly incorrect F matrix can still cause the reprojection error to be really high later on even if your triangulate code is correct.

Q4.2 Note: Don't worry if your solution is different from the example as long as the 3D structure of the temple is evident.

Q5.1: How many inliers should I be getting from RANSAC?

The correct number of inliers should be around 10 0. This provides a good sanity check for whether the chosen tolerance value is appropriate.

References

[1] David A Forsyth and Jean Ponce. *Computer vision: a modern approach*. prentice hall professional technical reference, 2002.

[2] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Nature, 2022.