2️⃣

HW2 3D Reconstruction from Epipolar Geometry

Due Date: May 22, 23:59

Instructions

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must submit their own work. If you work as a group, include the names of your collaborators in your write up. Code should **NOT** be shared or copied. Please **DO NOT** use external code unless permitted. Plagiarism is strongly prohibited and will lead to failure of this course.
2. **Start early!** Especially if you are not familiar with Python.
3. **Questions:** If you have any question, please look at blackboard first. Other students may have encountered the same problem, and is solved already. If not, post your question on the discussion board. TAs will respond as soon as possible.
4. **Write-up:** Items to be included in the write-up are mentioned in each question, and summarized in the write-up section. Please note that we **DO NOT** accept handwritten scans for your write-up in this assignment. Please type your answers to theory questions and discussions for experiments electronically.
5. **Handout:** ***PDF file*** is the assignment handout. In addition, the ***zip file*** contains 2 items: "*data*" contains 2 temple image files from the Middlebury MVS Temple Data set, and some npz files. "*python*" contains the scripts that you will make use of in this homework.
6. **Submission:** Your submission for this assignment should be a zip file, *<name_id>.zip,* composed of your write-up, your Python implementations (including helper functions), and your implementations, results for extra credit (optional). Your final upload should have the files arranged in this layout:

```
<name_id>.zip
    • <name_id>
        – <name_id>.pdf
        – python
            ∗ helper.py
            ∗ Q1eight_point.py
            ∗ Q2eppipolar_correspodences.py
            ∗ Q3essential_matrix.py
            ∗ Q4triangulation.py
            ∗ Q5results.py
            ∗ Q6rectify.py
            ∗ Q7disparity.py
            ∗ Q8depth.py
            ∗ Q9estimate_pose.py
            ∗ Q10estimate_params.py
```

```
          * Q11project_cad.py
      – data
          * im1.png
          * im2.png
          * intrinsics.npz
          * some_corresp.npz
          * temple_coords.npz
          * pnp.npz
          * extrinsics.npz (generated)
          * rectify.npz (generated)
```

Please make sure you do follow the submission rules mentioned above before uploading your zip file to Blackboard. Assignments that violate this submission rule will be **penalized by** up to 10% of the total score.

7. **File paths:** Please make sure that any file paths that you use are relative and not absolute.

   Not cv2.imread('/name/Documents/subdirectory/hw2/data/xyz.jpg') but cv2.imread('../data/xyz.jpg').

8. **External library**: Advanced functions from external libraries that are not mentioned in this PDF are not allowed for the core algorithm (e.g., *cv2.triangulate*).

1. Overview

One of the major areas of computer vision is 3D reconstruction. Given several 2D images of an environment, can we recover the 3D structure of the environment, as well as the position of the camera/robot? This has many uses in robotics and autonomous systems, as understanding the 3D structure of the environment is crucial to navigation. You don't want your robot constantly bumping into walls, or running over human beings!

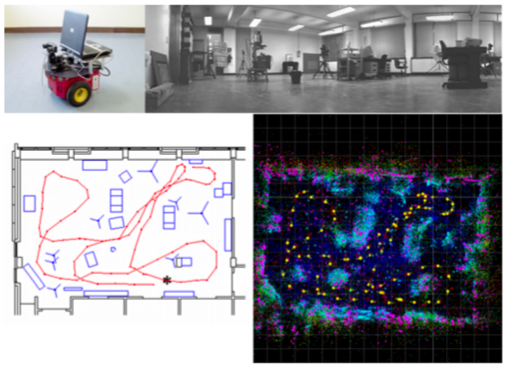

Figure 1: Example of a robot using SLAM, a 3D reconstruction and localization algorithm

In this assignment there are two programming parts: sparse reconstruction and dense reconstruction. **Sparse reconstructions generally contain a number of points, but still manage to describe the objects in question. Dense reconstructions are detailed and fine grained.** In fields like 3D modeling and graphics, extremely accurate dense reconstructions are invaluable when generating 3D models of real world objects and scenes.

In **Part 1**, you will be writing a set of functions to generate a sparse point cloud for some test images we have provided to you. The test images are 2 renderings of a temple from two different angles. We have also provided you with a *npz* file containing good point correspondences between the two images. You will first write a function that computes the **fundamental matrix** between the two images. Then write a function that uses the **epipolar constraint** to find more point matches between the two images. Finally, you will write a function that will triangulate the 3D points for each pair of 2D point correspondences.

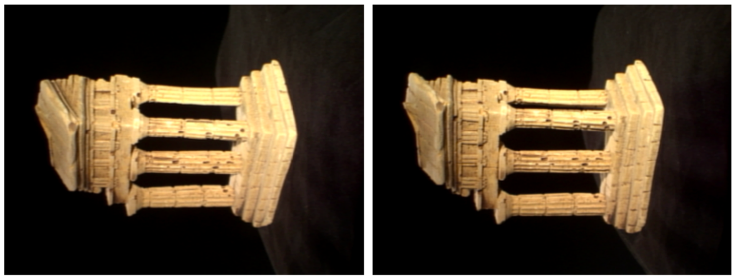
Figure 2: The two temple images we have provided to you

We have provided you with a few helpful *npz* files. The file *data/some_corresp.npz* contains good point correspondences. You will use this to compute the Fundamental matrix. The file *data/intrinsics.npz* contains intrinsic camera matrices, which you will need to compute the full camera projection matrices. Finally, the file *data/temple_coords.npz* contains some points on the first image that should be easy to localize in the second image.

In **Part 2**, we utilize the extrinsic parameters computed by Part 1 to further achieve dense 3D reconstruction of this temple. You will need to compute the rectification parameters. We have provided you with *Q6rectify.py* (and some helper functions) that will use your rectification function to warp the stereo pair. You will then *optionally* use the warped pair to compute a disparity map and finally a dense depth map.

In both cases, multiple images are required, because without two images with a large portion overlapping the problem is mathematically underspecified. It is for this same reason biologists suppose that humans, and other predatory animals such as eagles and dogs, have two front facing eyes. Hunters need to be able to discern depth when chasing their prey.
On the other hand herbivores, such as deer and squirrels, have their eyes position on the sides of their head, sacrificing most of their depth perception for a larger field of view. The whole problem of 3D reconstruction is inspired by the fact that humans and many other animals rely on dome degree of depth perception when navigating and interacting with their environment. Giving autonomous systems this information is very useful.

2. Sparse Reconstruction [85 pts]

In this section, you will be writing a set of function to compute the sparse reconstruction from two sample images of a temple. You will first estimate the Fundamental matrix, compute point correspondences, then plot the results in 3D.

Q1 Implement the eight point algorithm (15 points)

In this question, you're going to use the eight point algorithm which is covered in class to estimate the fundamental matrix. Please use the point correspondences provided in *data/some_corresp.npz*. Write a function in python/Q1eight_point.py with the following below step:

1. Load the two images and the point correspondences from *data/some corresp.npz*
2. Run *eight_point* to compute the fundamental matrix $\mathbf{F}$.

💡 *F = eight_point(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, and M is a scale parameter.

- Normalize points and un-normalize $\mathbf{F}$: You should scale the data by dividing each coordinate by *M* (the maximum of the image's width and height) using a transformation matrix $\mathbf{T}$. After computing $\mathbf{F}$, you will have to "unscale" the fundamental matrix. If $\mathbf{x_{norm}} = \mathbf{Tx}$, then $\mathbf{F_{unnorm}} = \mathbf{T}^\top \mathbf{FT}$.
- You must enforce the rank 2 constraint on $\mathbf{F}$ before unscaling. Recall that a valid fundamental matrix $\mathbf{F}$ will have all epipolar lines intersect at a certain point, meaning that there exists a non-trivial null space for $\mathbf{F}$. In general, with real points, the eight point solution for $\mathbf{F}$ will not come with this condition. To enforce the rank 2 constraint, decompose $\mathbf{F}$ with SVD to get the three matrices $\mathbf{U}, \Sigma, \mathbf{V}$such that $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^\top$. Then force the matrix to be rank 2 by setting the smallest singular value in Σ to zero, giving you a new $\Sigma^{'}$. Now compute the proper fundamental matrix with $\mathbf{F} = \mathbf{U}\Sigma^{'}\mathbf{V}^\top$.
- 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 *eight_point* before unscaling $\mathbf{F}$.
- 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).
- To visualize the correctness of your estimated $\mathbf{F}$, use the function *displayEpipolarF* in *python/helper.py*, which takes in $\mathbf{F}$, and the two images. This GUI lets you select a point in one of the images and visualize the corresponding epipolar line in the other image (Figure **3**).

In your write-up:

- Please include your recovered $\mathbf{F}$ and the visualization of some epipolar lines (similar to Figure **3**).

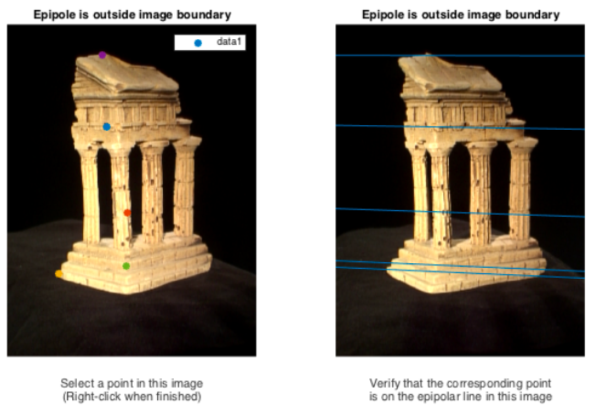
Epipole is outside image boundary
data1
Epipole is outside image boundary
Select a point in this image
(Right-click when finished)
Verify that the corresponding point
is on the epipolar line in this image

Figure 3: Epipolar lines visualization from *displayEpipolarF*

Q2 Find epipolar correspondences (25 points)

To reconstruct a 3D scene with a pair of stereo images, we need to find many point pairs. A point pair is two points in each image that correspond to the same 3D scene point. With enough of these pairs, when we plot the resulting 3D points, we will have a rough outline of the 3D object. You found point pairs in the previous homework using feature detectors and feature descriptors, and testing a point in one image with every single point in the other image. **But here we can use the fundamental matrix to greatly simplify this search.**

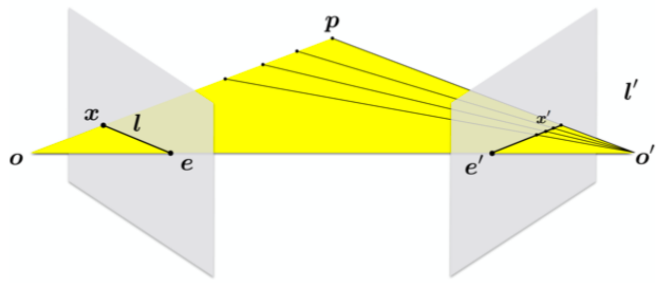

Figure 4: Epipolar Geometry (source Wikipedia)

Recall from class that given a point $\mathbf{x}$ in one image (the left view in Figure **4**). Its corresponding 3D scene point $\mathbf{p}$ could lie anywhere along the line from the camera center $\mathbf{o}$ to the point $\mathbf{x}$. This line, along with a second image's camera center $\mathbf{o}^{'}$ (the right view in Figure **4**) forms a plane. This plane intersects with the image plane of the second camera, resulting in a line l' in the second image which describes all the possible locations that $\mathbf{x}$ may be found in the second image. Line l' is the epipolar line, and we only need to search along this line to find a match for point $\mathbf{x}$ found in the first image. Write a function in python/Q2eppipolar_correspodences.py with the below step:

1. Load the points in image 1 contained in *data/temple_coords.npz* and run your

   epipolar_correspondences on them to get the corresponding points in image 2.

💡 *pts2 = epipolar_correspondences(im1, im2, F, pts1)*

Where *im1* and *im2* are the two images in the stereo pair, $\mathbf{F}$ is the fundamental matrix computed for the two images using your *eight_point* function, *pts1* is a $N \times 2$ matrix containing the (x, y) points in the first image, and the function should return *pts2*, a $N \times 2$ matrix, which contains the corresponding points in the second image.

- To match one point $\mathbf{x}$ in image 1, use fundamental matrix to estimate the corresponding epipolar line l' and generate a set of candidate points in the second image.
- For each candidate points $\mathbf{x'}$, a similarity score between $\mathbf{x}$ and $\mathbf{x'}$ is computed. The point among candidates with highest score is treated as epipolar correspondence.
- There are many ways to define the similarity between two points. Feel free to use whatever you want and **describe it in your write-up**. One possible solution is to select a small window of size w around the point $\mathbf{x}$. Then compare this target window to the window of the candidate point in the second image. For the images we gave you, simple Euclidean distance or Manhattan distance should suffice.
- Remember to take care of data type and index range.

You can use the function *epipolarMatchGUI* to visually test your function. Your function does not need to be perfect, but it should get most easy points correct, like corners, dots etc.

In your write-up:

- Please include a screenshot of *epipolarMatchGUI* running with your implementation of *epipolar_correspondences* (similar to Figure **5**). Mention the similarity metric you decided to use.
- Also comment on any cases where your matching algorithm consistently fails, and why you might think this is.

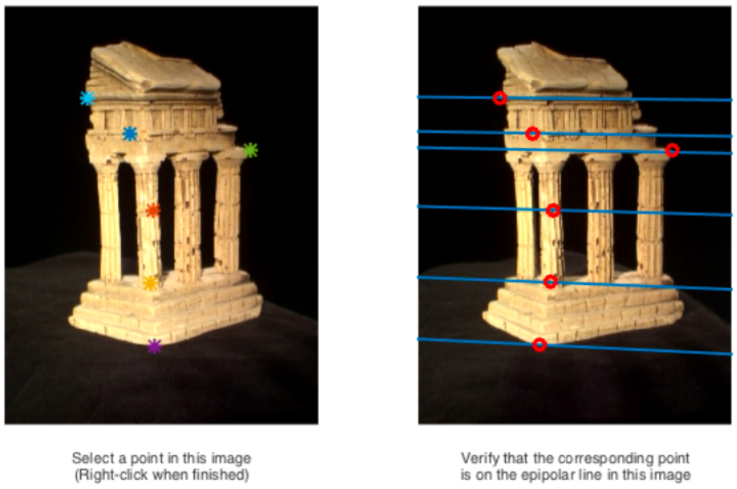
Select a point in this image
(Right-click when finished)
Verify that the corresponding point
is on the epipolar line in this image

Figure 5: Epipolar Match visualization. A few errors are alright, but it should get most easy points correct (corners, dots, etc.)

Q3 Write a function to compute the essential matrix (10 points)

In order to get the full camera projection matrices we need to compute the Essential matrix. So far, we have only been using the Fundamental matrix. Write a function in *python/Q3essential_matrix.py* with the following below step:

1. Load *data/intrinsics.npz* and compute the essential matrix $\mathbf{E}$.

> 💡 *E = essential_matrix(F, K1, K2)*

Where $\mathbf{F}$ is the Fundamental matrix computed between two images, $\mathbf{K_1}$ and $\mathbf{K_2}$ are the intrinsic camera matrices for the first and second image respectively (contained in *data/intrinsics.npz*), and $\mathbf{E}$ is the computed essential matrix. The intrinsic camera parameters are typically acquired through camera calibration. Refer to the class slides for the relationship between the Fundamental matrix and the Essential matrix.

In your write-up:

- Please include your estimated $\mathbf{E}$ matrix for the temple image pair.

Q4 Implement triangulation (15 points)

Write a function in *python/Q4triangulation.py* to triangulate pairs of 2D points in the images to a set of 3D points following below steps:

1. Compute the first camera projection matrix $\mathbf{P_1}$ and use camera2 to compute the four candidates for $\mathbf{P_2}$.
2. Run your *triangulate* function using the four sets of camera matrix candidates, the points from *data/temple_coords.npz* and their computed correspondences.
3. Figure out the correct $\mathbf{P_2}$ and the corresponding 3D points. Hint: You'll get 4 projection matrix candidates for camera2 from the essential matrix. The correct configuration is the one for which most of the 3D points are in front of both cameras (positive depth).
4. Compose 3D points with the correct $\mathbf{P_2}$.

> 💡 *pts3d = triangulate(P1, pts1, P2, pts2)*

Where *pts1* and *pts2* are the $N \times 2$ matrices with the 2D image coordinates, $\mathbf{P_1}$ and $\mathbf{P_2}$ are the 3×4 camera projection matrices and *pts3d* is an $N \times 3$ matrix with the corresponding 3D points (in all cases, one point per row). Remember that you will need to multiply the given intrinsic matrices with your solution for the extrinsic camera matrices to obtain the final camera projection matrices. For $\mathbf{P_1}$ you can assume no rotation or translation, so the extrinsic matrix is just $[\mathbf{I} \mid \mathbf{0}]$. **For $\mathbf{P_2}$, pass the essential matrix to the provided function in *python/helper.py* to get four possible extrinsic matrices.** You will need to determine which of these is the correct one to use (see hint). Refer to the class slides for one possible triangulation algorithm.

In your write-up:

- Describe which matrix of the index was selected for $\mathbf{P_2}$, how you determined which extrinsic matrix is correct. Note that simply rewording the hint is not enough.

Q5 Write a result script (20 points)

You now have all the pieces you need to generate a full 3D reconstruction! So let's see the results. Visualize the results first. You can use matplotlib's function to visualize this. If done well, you should get a picture similar to **Figure 6.** Next, check the performance by looking at the re-projection error. To compute the re-projection error, project the estimated 3D points back to the image 1 and compute the mean Euclidean error between projected 2D points and the given *pts1*. Please use the point correspondences provided in *data/some_corresp.npz* . Once you checked reprojection error, now you can save your parameters what you found.

Write a script *python/Q5results.py* that does the following:

1. Use matplotlib's *scatter* function to plot these point correspondences on screen.
2. Calculate the reprojection error. To compute the results, you should find 3D points from *data/some_corresp.npz with triangulate* function in Q4.
3. Save your computed extrinsic parameters $(\mathbf{R_1}, \mathbf{R_2}, \mathbf{t_1}, \mathbf{t_2})$ to *data/extrinsics.npz*. These extrinsic parameters will be used in the next section.

💡 reproj_err = *compute_reprojerr(P1, pts1, P2, pts2, pts3d)*

We will use your test script to run your code, so be sure it runs smoothly. In particular, use relative paths to load files, not absolute paths.

In your write-up:

- Report your re-projection error using the given *pts1* and *pts2* in *data/some_corresp.npz*. If implemented correctly, the re-projection error should be less than 3 pixel.
- Include 3 images of your final reconstruction of the points given in the file *data/temple_coords.npz*. See Figure **6** for an example .

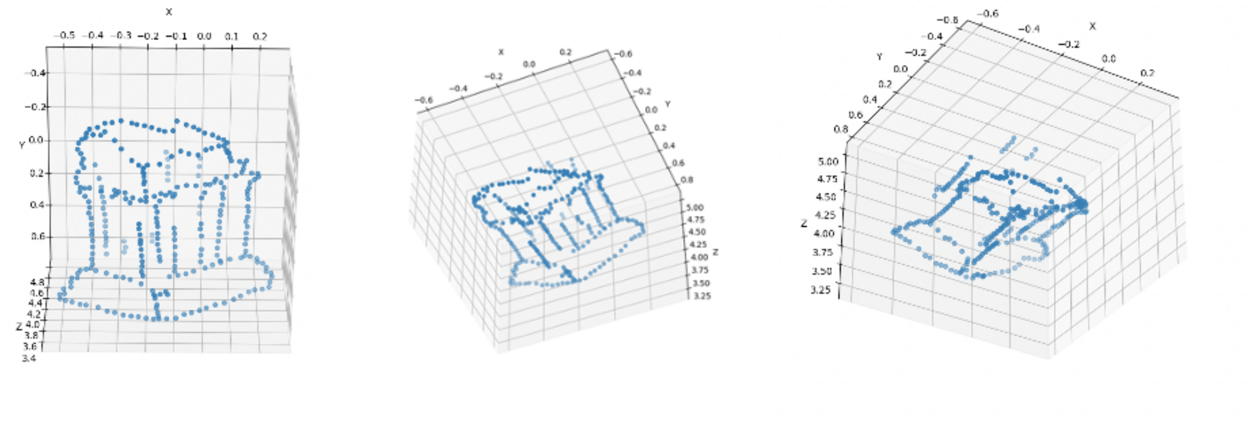

Figure 6: Sample reconstruction points. This example is not the exact solution. Your result can be different from this example

3. Dense Reconstruction [50 pts]

In applications such as 3D modelling, 3D printing, and AR/VR, a sparse model is not enough. When users are viewing the reconstruction, it is much more pleasing to deal with a dense reconstruction. To do this, it is helpful to rectify the images to make matching easier.

In this section, you will be writing a set of functions to perform a dense reconstruction on our temple examples. Given the provided intrinsic and computed extrinsic parameters, you will need to write a function to compute the rectification parameters of the two images. The rectified images are such that the epipolar lines are horizontal, so searching for correspondences becomes a simple linear. This will be done for every point. Finally, you can compute the disparity and depth map.

Q6 Image Rectification (20 points)

Write a program that computes rectification matrices.

M1, M2, K1p, K2p, R1p, R2p, t1p, t2p = rectify_pair(K1, K2, R1, R2, t1, t2)

This function takes the left $(\mathbf{K_1}, \mathbf{R_1}, \mathbf{t_1})$ and right camera parameters $(\mathbf{K_2}, \mathbf{R_2}, \mathbf{t_2})$, and returns the left $(\mathbf{M_1})$ and right $(\mathbf{M_2})$ rectification matrices and the updated camera parameters $(\mathbf{K_1p}, \mathbf{R_1p}, \mathbf{t_1p}, \mathbf{K_2p}, \mathbf{R_2p}, \mathbf{t_2p})$. You can test your function using the provided script *python/Q6rectify.py*. From what we learned in class, the *rectify_pair* function should consecutively run the following steps:

1. Compute the optical centers $\mathbf{c}_1$ and $\mathbf{c}_2$ of each camera by $\mathbf{c}_i = -\left(\mathbf{K}_i\mathbf{R}_i\right)^{-1}\left(\mathbf{K}_i\mathbf{t}_i\right)$
2. Compute the new rotation matrix $\widetilde{\mathbf{R}} = \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 \end{bmatrix}^\top$ where $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3 \in \mathbb{R}^{3\times1}$ are orthonormal vectors that represent x, y, and z axes of the camera reference frame, respectively.

   (a) The new $x\text{-axis}$ $(\mathbf{r}_1)$ is parallel to the baseline: $\mathbf{r}_1 = (\mathbf{c}_2 - \mathbf{c}_1) / \|\mathbf{c}_2 - \mathbf{c}_1\|$

   (b) The new $y\text{-axis}$ $(\mathbf{r}_2)$ is orthogonal to $\mathbf{x}$ and to any arbitrary unit vector, which we set to be the $\mathbf{z}$ unit vector of the old left matrix: $\mathbf{r}_2$ is the cross product of $\mathbf{R}_1(3,:)^\top$ and $\mathbf{r}_1$

   (c) The new $z\text{-axis}$ $(\mathbf{r}_3)$ is orthogonal to $\mathbf{x}$ and $\mathbf{y}$: $\mathbf{r}_3$ is the cross product of $\mathbf{r}_2$ and $\mathbf{r}_1$

   Set the new rotation matrices as
   $\mathbf{R}_1' = \mathbf{R}_2' = \widetilde{\mathbf{R}}$

3. Compute the new intrinsic parameters as $\mathbf{K}_1' = \mathbf{K}_2' = \mathbf{K}_2$
4. Compute the new translation vectors as $\mathbf{t}_1 = -\widetilde{\mathbf{R}}\mathbf{c}_1$ and $\mathbf{t}_2 = -\widetilde{\mathbf{R}}\mathbf{c}_2$.
5. Finally, the rectification matrices of the cameras $(\mathbf{M}_1, \mathbf{M}_2)$ can be obtained by

$$\mathbf{M}_i = (\mathbf{K}_i'\mathbf{R}_i')(\mathbf{K}_i\mathbf{R}_i)^{-1}$$

Once you finished, run *python/Q6rectify.py* (Ensure *data/extrinsics.npz* is saved by your *python/Q5results.py*). This script will test your rectification code on the temple images using the provided intrinsic parameters and your computed extrinsic parameters. It will also save the estimated rectification matrices and updated camera parameters in *data/rectify.npz*, which will be used by the function *python/Q8depth.py*.

In your write-up:

- Include a screenshot of the result of *python/Q6rectify.py* on the temple images. The results should show some epipolar lines that are perfectly horizontal, with corresponding points in both images lying on the same line. See Figure **7** for an example.
- Why did it happen, and why should we compute the new matrix using that process (Step 2)? Please explain in your own words.

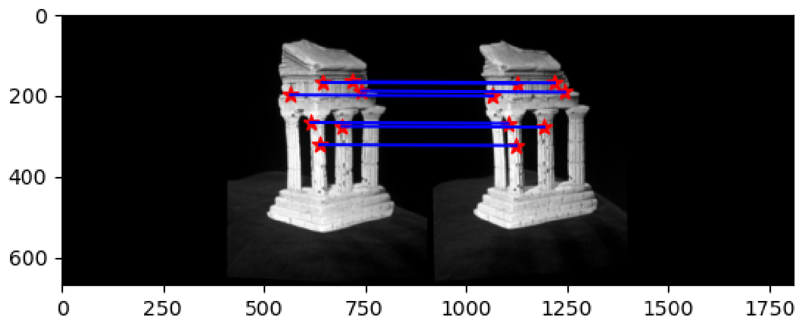

Figure 7 : We can see that the lines connecting the corresponding points are parallel to x-axis

Q7 Dense window matching to find per pixel disparity (10 points)

Write a program that creates a disparity map from a pair of rectified images.

💡 *dispM = get_disparity(im1, im2, min_disp, num_disp, win_size)*

where *min_disp* is the minimum disparity, *num_disp* is the number of disparity to search and *win_size* is the window size. The output *dispM* has the same dimension as *im1* and *im2*. Since *im1* and *im2* are rectified, computing correspondences is reduced to a 1D search problem. The value of *dispM(y, x)* is given by

$$\operatorname{dispM}(y,x) = \underset{\text{min\_disp} \le d \le \text{min\_disp}+\text{num\_disp}}{\operatorname{argmin}} \operatorname{dist}(\operatorname{im1}(y,x), \operatorname{im}2(y, x-d)) \quad (1)$$

where,

$$\operatorname{dist}(\operatorname{im}1(y,x), \operatorname{im}2(y,x-d))$$

$$= \sum_{i=-w}^{w} \sum_{j=-w}^{w} (\operatorname{im}1(y+i, x+j) - \operatorname{im}2(y+i, x+j-d))^2 \quad (2)$$

and w = (*win_size* - 1)/2. This summation on the window can be easily computed by using the *scipy.signal.convolve2d* function (i.e. convolution with a mask of ones). Note that this is not the only way to implement this.

In your write-up:

- Please include 2 images of the disparity maps and describe the effect of window size. Also, you must use the matplotlib's function *imshow* with option *cmap='inferno'* to visualize disparity map. See Figure **8, 9** for an example (Note that your result can be different from these figures).

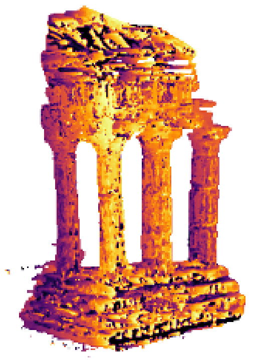
Figure 8: disparity map, window size = 3

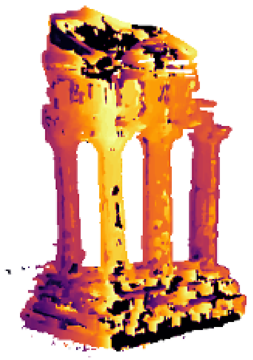
Figure 9: disparity map, window size = 11

Q8 Depth map (20 points)

Write a function that creates a depth map from a disparity map.

💡 *depthM = get_depth(dispM, K1, K2, R1, R2, t1, t2)*

where we can use the fact that

$$\operatorname{depthM}(y, x) = b \times f / \operatorname{dispM}(y, x) \quad (3)$$

where b is the baseline and f is the focal length of camera. For simplicity, assume that $b = \|\mathbf{c_1} - \mathbf{c_2}\|$ (i.e., distance between optical centers) and $f = \mathbf{K_1}(1,1)$. Finally, let *depthM*$(y, x) = 0$ whenever *dispM*$(y, x) = 0$ to avoid division by 0. You can now test your disparity and depth map functions using *python/Q8depth.py*. Ensure that the rectification is saved (by running *python/Q6rectify.py*). This function will rectify the images, then compute the disparity map and the depth map.

In your write-up:

- Please include 2 images of the depth maps. Also, you must use the matplotlib's function *imshow* with option *cmap='inferno'* to visualize depth map. See Figure **10, 11** for an example (Note that your result can be different from these figures).
- Is there any relationship between the depth estimation accuracy and baseline or focal length, etc.? How can we improve the performance of depth estimation?

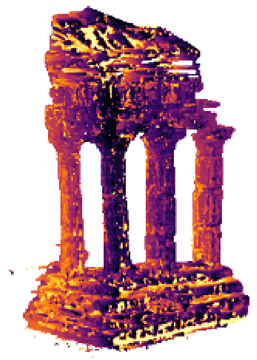

Figure 10: depth map, window size = 3

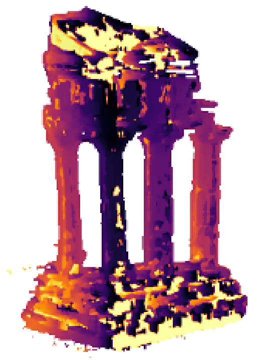

Figure 11: depth map, window size = 11

4. Pose Estimation [65 pts]

In this section, you will implement what you have learned in class to estimate the intrinsic and extrinsic parameters of camera given 2D points $\mathbf{x}$ on image and their corresponding 3D points $\mathbf{X}$. In other words, given a set of matched points $\{\mathbf{X}_i, \mathbf{x}_i\}$ and camera model,

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{1} \end{bmatrix} = f(\mathbf{X}; \mathbf{p}) = \mathbf{P} \begin{bmatrix} \mathbf{X} \\ \mathbf{1} \end{bmatrix} \tag{4}$$

we aim to estimate the camera matrix $\mathbf{P} \in \mathbb{R}^{3\times4}$, as well as intrinsic parameter matrix $\mathbf{K} \in \mathbb{R}^{3\times3}$, camera rotation $\mathbf{R} \in \mathbb{R}^{3\times3}$ and camera translation $\mathbf{t} \in \mathbb{R}^3$, such that

$$\mathbf{P} = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \tag{5}$$

Q9 Estimate camera matrix P (20 points)

Write a function that estimates the camera matrix $\mathbf{P}$ given 2D and 3D points $\mathbf{x}, \mathbf{X}$.

💡 *P = estimate_pose(x, X)*

where $\mathbf{x}$ is $2 \times N$ matrix denoting the (x, y) coordinates of the N points on the image plane and $\mathbf{X}$ is $3 \times N$ matrix denoting the (x, y, z) coordinates of the corresponding points in the 3D world. Recall that this camera matrix can be computed using the same strategy as homography estimation by Direct Linear Transform (DLT). Once you finish this function, you can run the provided script *python/Q9estimate_pose.py* to test your implementation.

In your write-up:

- Please include the output of the script *python/Q9estimate_pose.py*. If implemented correctly, the re-projection error and pose error with clean points should be less than 10^{-10}.
- Camera matrix $\mathbf{P}$ is not only the pose. Please explain the meaning of the matrix $\mathbf{P}$.

Q10 Estimate intrinsic/extrinsic parameters (25 points)

Write a function that estimates both intrinsic and extrinsic parameters from camera matrix.

💡 *K, R, t = estimate_params(P)*

From what we learned on class, the *estimate_params* function should consecutively run the following steps:

1. Compute the camera center $\mathbf{c}$ by using SVD. Hint: $\mathbf{c}$ is the eigenvector corresponding to the smallest eigenvalue.
2. Compute the intrinsic $\mathbf{K}$ and rotation $\mathbf{R}$ by using QR decomposition. $\mathbf{K}$ is a right upper triangle matrix while $\mathbf{R}$ is a orthonormal matrix.
3. Compute the translation by $\mathbf{t} = -\mathbf{R}\mathbf{c}$.

Once you finish your implementation, you can run the provided script *python/Q10estimate_params.py*.

In your write-up:

- Please include the output of the script *python/Q10estimate_params.py*.

Q11 Project a CAD model to the image (20 points)

Now you will utilize what you have implemented to estimate the camera matrix from a real image, shown in **Figure 12** (left), and project the 3D object (CAD model), shown in **Figure 12** (right), back on to the image plane. Write a script *python/Q11project_cad.py* that does the following:

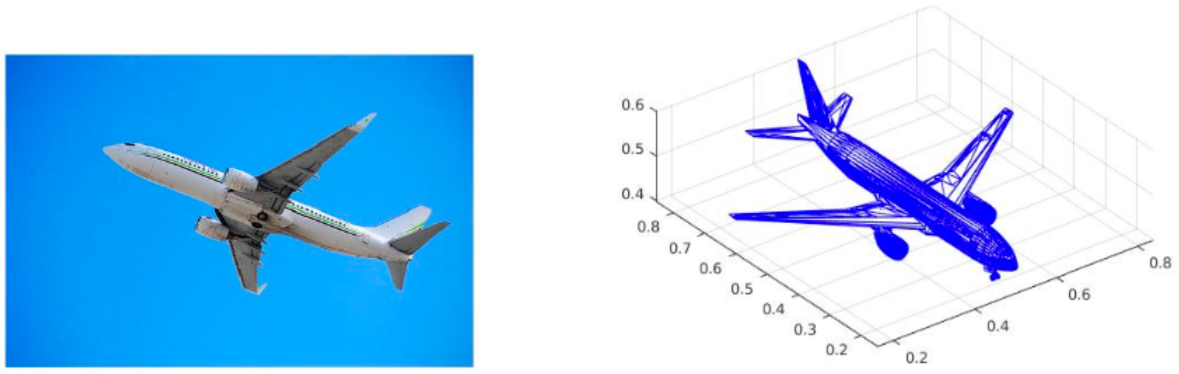

Figure 12: The provided image and 3D CAD model

1. Load an image *image*, a CAD model *cad*, 2D points $\mathbf{x}$ and 3D points $\mathbf{X}$ from the data file *data/pnp.npz*.
2. Run *estimate_pose* and *estimate_params* to estimate camera matrix $\mathbf{P}$, intrinsic matrix $\mathbf{K}$, rotation matrix $\mathbf{R}$, and translation $\mathbf{t}$.
3. Use your estimated camera matrix $\mathbf{P}$ to project the given 3D points $\mathbf{X}$ onto the image.
4. Plot the given 2D points $\mathbf{x}$ and the projected 3D points on screen. An example is shown in Figure **8** (left).
5. Draw the CAD model rotated by your estimated rotation $\mathbf{R}$ on screen. An example is shown in Figure **8** (middle).
6. Project the CAD's all vertices onto the image and draw the projected CAD model overlapping with the 2D image. An example is shown in **Figure 13** (right).

In your write-up:

- Please include the three images similar to Figure 13. **You must use different colors from Figure 13**. For example, green circle for given 2D points, black points for projected 3D points, blue CAD model, and red projected CAD model overlapping on the image. You will get **NO** credit if you use the same color. Library *mpl_toolkits.mplot3d* will be helpful but not the only way to implement this.

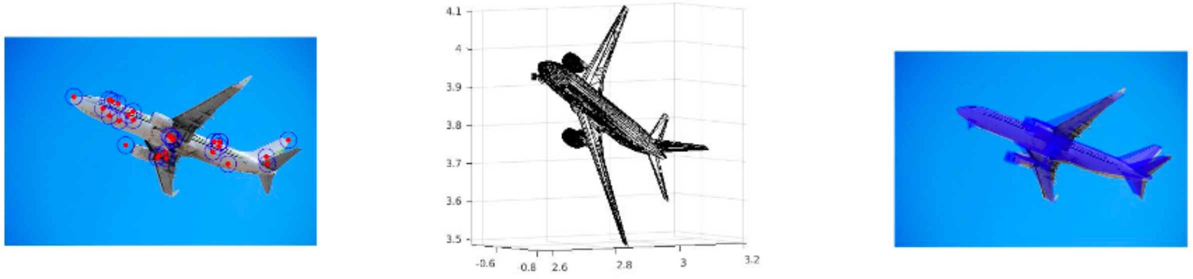

Figure 13: Project a CAD model back onto the image. Left: the image annotated with given 2D points (blue circle) and projected 3D points (red points); Middle: the CAD model rotated by estimated R; Right: the image overlapping with projected CAD model.