# Project 5
# 3D Reconstruction

Due date: 23:59 Tuesday 12/5th (2023)

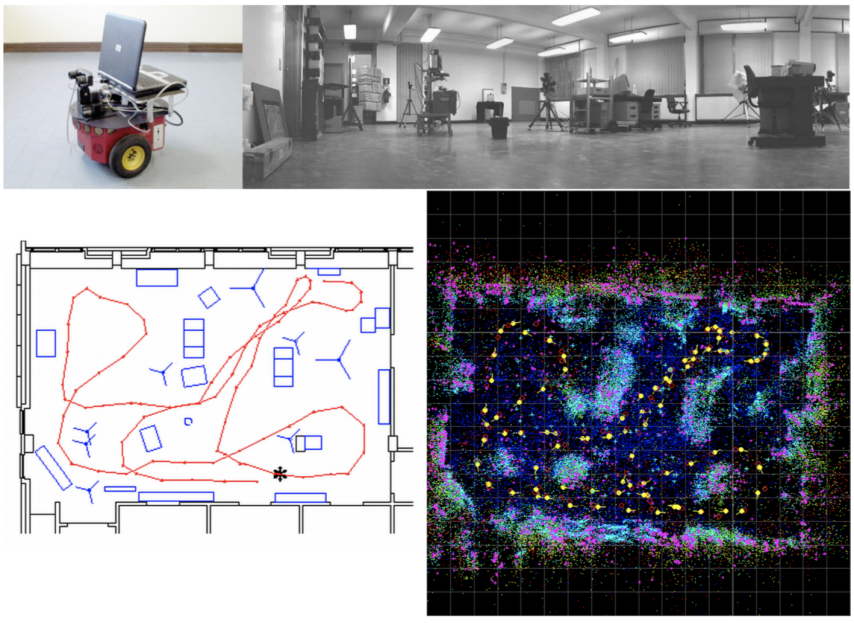
## 1. Instructions

Most instructions are the same as before. Here we only describe different points.

1. Generate a zip package, and upload to canvas. Also upload the pdf with the name {SFUID}.pdf. The package must contain the following in the following layout:
    - {SFUID}
        - python
        - result (You likely won't need this directory. If you have any results which you cannot include in the write-up but want to refer, please use.)
2. Project 3 has 22 pts.

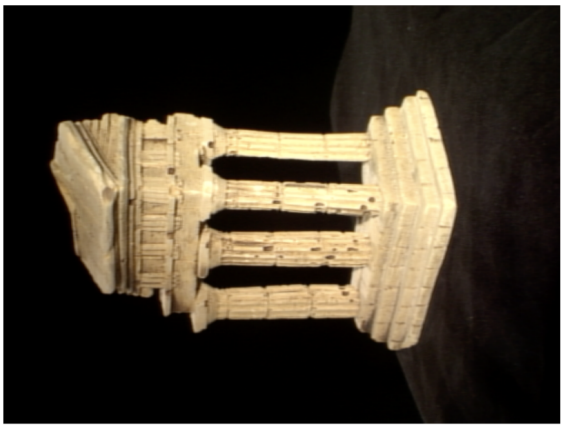
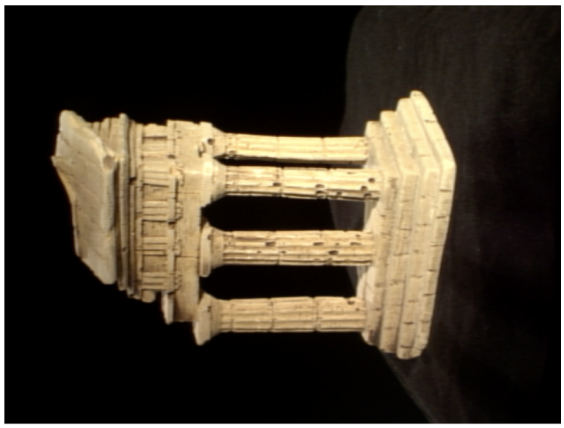
## 2. Overview

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!



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 modelling and graphics, extremely accurate dense reconstructions are invaluable when generating 3D models of real world objects and scenes.

In part 1, you will write 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 npy 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 you will 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.

We have provided a few helpful npy files. someCorresps.npy contains good point correspondences. You will use this to compute the Fundamental matrix. Intrinsics.npy contains the intrinsic camera matrices, which you will need to compute the full camera projection matrices. Finally, templeCoords.npt 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 testRectify.py (and some helper functions) that will use your rectification function to warp the stereo pair. You will then 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 positioned on the sides of their heads, 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 depth perception when navigating and interacting with their environment. Giving autonomous systems this information is very useful.

In Part 3, we will implement the camera parameter estimation, first estimating the projection matrix given pairs of 2D image coordinates and 3D point coordinates, then decomposition it into intrinsics and extrinsics parameters.

Lastly in Part 4, we will see how more than 2 images will dramatically improve the quality of 3D reconstruction, which is called multi-view stereo. We will compute a depthmap for one reference image by using multiple images. We will have a different but very simple formulation.

# 3. Tasks

## 3.1 Sparse reconstruction

In this section, you will write a set of functions 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.

It may be helpful to read through Section 3.1.5 right now. In Section 3.1.5 we ask you to write a testing script that will run your whole pipeline. It will be easier to start that now and add to it as you complete each of the questions one after the other.

### 3.1.1 Implement the eight point algorithm (2 pts)

You will use the eight point algorithm to estimate the fundamental matrix. Please use the point correspondences provided in someCorresp.npy. Complete the function *eightpoint(pts1, pts2, M)* in eightpoint.py.

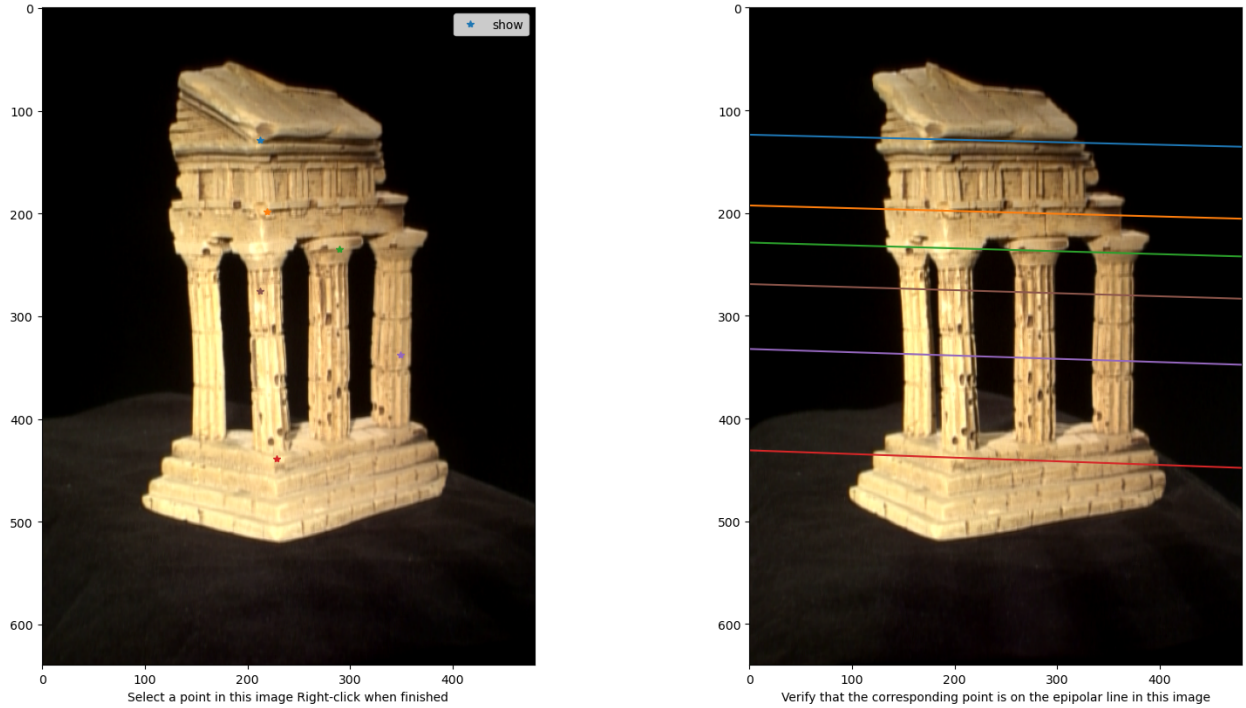
pts1 and pts2 are Nx2 matrices corresponding to the (x, y) coordinates of the N points in the first and second image respectively. M is a scale parameter.

- Normalize points and un-normalize F: You should scale the data by dividing each coordinate by M (the maximum of the image's width and height). After computing F, you will have to "unscale" the fundamental matrix. Note that you could subtract the mean coordinate then divide by the standard deviation for rescaling for better results. For this, you can do a simple scaling (w/o subtraction).
- You must enforce the rank 2 constraint on F before unscaling. Recall that a valid fundamental matrix F will have all epipolar lines intersect at a certain point, meaning that there exists a non-trivial null space for F. In general, with real points, the eightpoint solution for F will not come with this condition. To enforce the rank 2 condition, decompose F with SVD to get the three matrices U, Σ, V such that $F = U\Sigma V^T$ . Then

force the matrix to be rank 2 by setting the smallest singular value in Σ to zero, giving you a new Σ′. Now compute the proper fundamental matrix with F′ = UΣ′VT .

- 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 refineF.py (takes in Fundamental matrix and the two sets of points), which you can call from eightpoint before unscaling F. This function applies a non-linearly search for a better F that minimizes the cost function. For this to work, it needs an initial guess for F that is already close to the minimum.
- 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 overdetermined system (N > 8 points).
- To test your estimated F, use the provided function displayEpipolarF.py (takes in 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 like in the figure below.

**In your write-up, please include your recovered F and the visualization of some epipolar lines like the figure below.**




## 3.1.2 Find epipolar correspondences (2 pts)

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.

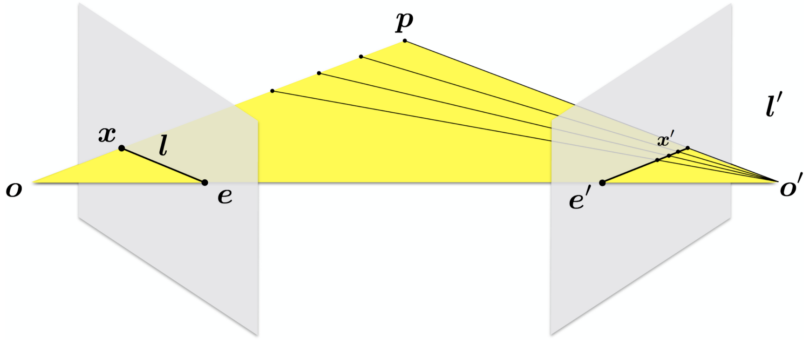



Figure 6: Epipolar Geometry (source Wikipedia)

Recall from class that given a point x in one image (the left view in Figure 6), its corresponding 3D scene point p could lie anywhere along the line from the camera center o to the point x. This line, along with a second image's camera center o′ (the right view in Figure 6) 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 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 x found in the first image.

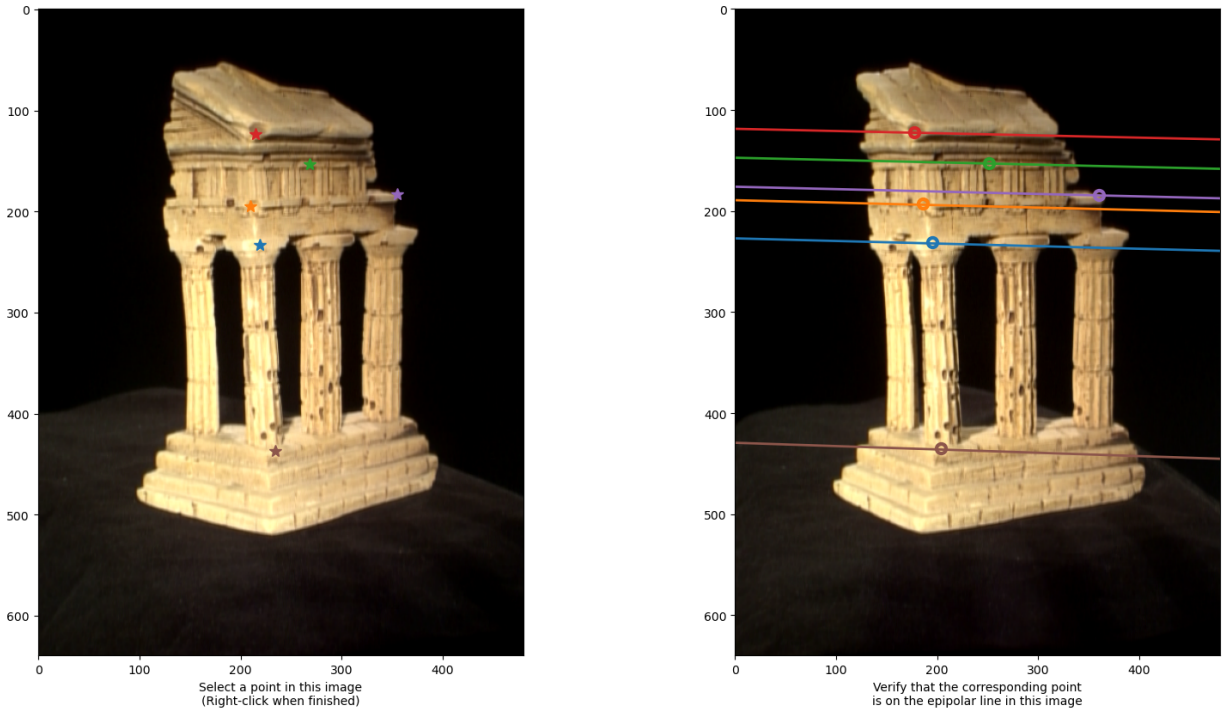
Complete the function *epipolarCorrespondence(im1, im2, F, pts1):*

im1 and im2 are the two images in the stereo pair. F is the fundamental matrix computed for the two images using your eightpoint function. pts1 is an N × 2 matrix containing the (x,y) points in the first image. Your function should return pts2, an N × 2 matrix, which contains the corresponding points in the second image.

- To match one point 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 x′, a similarity score between x and 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 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. Manhattan distance was not covered in class. Consider Googling it.
- Remember to take care of data type and index range.

You can use epipolarMatchGui.py 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, include a screenshot of epipolarMatchGui running with your implementation of epipolarCorrespondence (like the figure below). 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.**




## 3.1.3 Write a function to compute the essential matrix (2 pts)

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.

Complete the function *essentialMatrix(F, K1, K2):*

F is the Fundamental matrix computed between two images, K1 and K2 are the intrinsic camera matrices for the first and second image respectively (contained in intrinsics.npy). The function returns the computed essential matrix E. The intrinsic camera parameters are typically acquired through camera calibration.

In your write-up, write your estimated E matrix for the temple image pair we provided.

### 3.1.4 Implement triangulation (2 pts)

Complete the function *triangulate(P1, pts1, P2, pts2)* which triangulates pairs of 2D points in the images to a set of 3D points.

pts1 and pts2 are the Nx2 matrices with the 2D image coordinates. P1 and P2 are the 3x4 camera projection matrices. For P1 you can assume no rotation or translation, so the extrinsic matrix is just [I|0]. For P2, pass the essential matrix to the provided camera2.py function to get four possible extrinsic matrices. The function returns pts3d which is an Nx3 matrix with the corresponding 3D points (in all cases, one point per row). Remember that you will need to multiply the given intrinsic matrices to obtain the final camera projection matrices. You will need to determine which of these is the correct one to use (see hint in Section 3.1.5).

Once you have it implemented, 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(2) and compute the mean Euclidean error between projected 2D points and pts1(2).

**In your write-up, describe how you determined which extrinsic matrices are correct. Report your re-projection error using pts1, pts2 from someCorresp.npy. If implemented correctly, the re-projection error should be less than 1 pixel.**

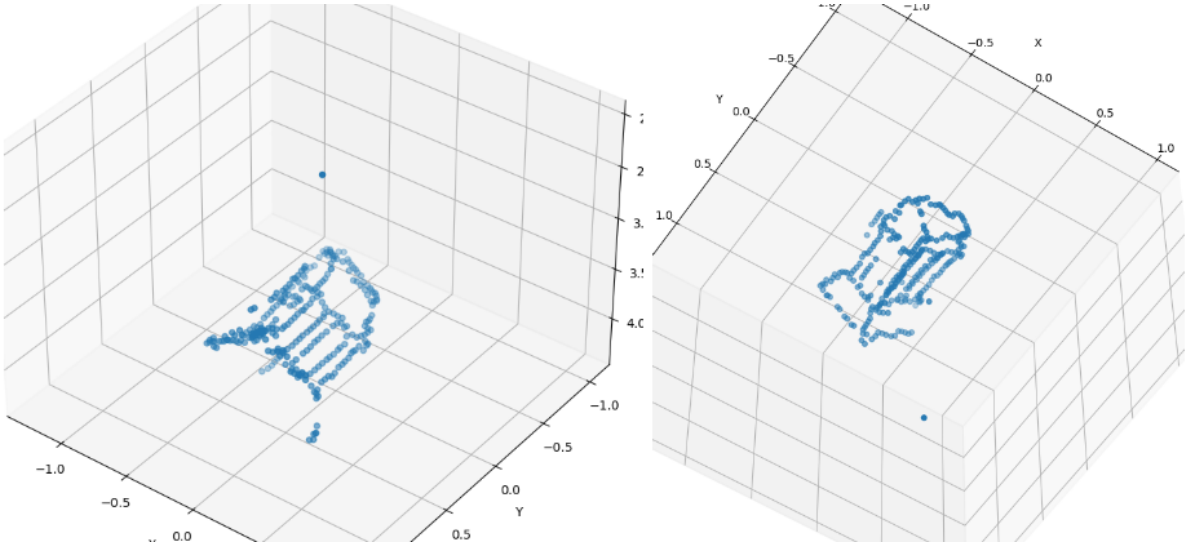
### 3.1.5 Write a test script that uses templeCoords (2 pts)

You now have all the pieces you need to generate a full 3D reconstruction. Write a test script testTempleCoords.py that does the following:

1. Load the two images and the point correspondences from someCorresp.npy
2. Run eightpoint to compute the fundamental matrix F
3. Load the points in image 1 contained in templeCoords.npy and run your epipolarCorrespondences on them to get the corresponding points in image
4. Load intrinsics.npy and compute the essential matrix E.
5. Compute the first camera projection matrix P1 and use camera2.py to compute the four candidates for P2
6. Run your triangulate function using the four sets of camera matrix candidates, the points from templeCoords.npy and their computed correspondences.
7. Figure out the correct P2 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).
8. Plot these point correspondences on screen.
9. Save your computed rotation matrix (R1, R2) and translation (t1, t2) to the file ../result/extrinsics.npy. These extrinsic parameters will be used in the next section.

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, include 3 images of your final reconstruction of the templeCoords points, from different angles. Like the figure below.**




## 3.2 Dense reconstruction

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 will compute the depth map.

### 3.2.1 Image rectification (2 pts)

Write a function that computes rectification matrices: *rectify_pair (K1, K2, R1, R2, t1, t2)*

This function takes left and right camera parameters (K, R, t) and returns left and right rectification matrices (M1, M2) and updated camera parameters (K1n, K2n, R1n, R2n, t1n, t2n). You can test your function using the provided script testRectify.py.

From what we learned in class, the rectify_pair function should consecutively run the following steps:

1. Compute the optical center c1 and c2 of each camera by c = −(KR)^{−1}(Kt).
2. Compute the new rotation matrix $\widetilde{\mathbf{R}} = \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 \end{bmatrix}^T$ where r1 , r2 , r3 ∈ R3×1 are orthonormal vectors that represent x-, y-, and z-axes of the camera reference frame, respectively.
    a. The new x-axis (r1) is parallel to the baseline: r1 = (c1 − c2)/ // c1 − c2 // .
    b. The new y-axis (r2) is orthogonal to x and to any arbitrary unit vector, which we set to be the z unit vector of the old left matrix: r2 is the cross product of R1(3, :) and r1.
    c. The new z-axis (r3) is orthogonal to x and y: r3 is the cross product of r2 and r1.
3. Compute the new intrinsic parameter $\widetilde{\mathbf{K}}$. We can use an arbitrary one. In our test code, we just let $\widetilde{\mathbf{K}}$ = K2.
4. Compute the new translation: t1n=−$\widetilde{\mathbf{R}}$c1, t2n=−$\widetilde{\mathbf{R}}$c2.
5. Finally, the rectification matrix of the first camera can be obtained by
$$M_1 = (\tilde{K}\tilde{R})(K_1 R_1)^{-1}$$
M2 can be computed from the same formula.

Once you finished, run testRectify.py (Be sure to have the extrinsics saved by your testTempleCoords.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 matrix and updated camera parameters in rectify.npy, which will be used by the next test script testDepth.py.

**In your write-up, include a screenshot of the result of running testRectify.py on temple images. The results should show some epipolar lines that are perfectly horizontal, with corresponding points in both images lying on the same line.**

### 3.2.2 Dense window matching to find per pixel density (2 pts)

Write a program that creates a disparity map from a pair of rectified images (im1 and im2).

get_disparity(im1, im2, maxDisp, windowSize)

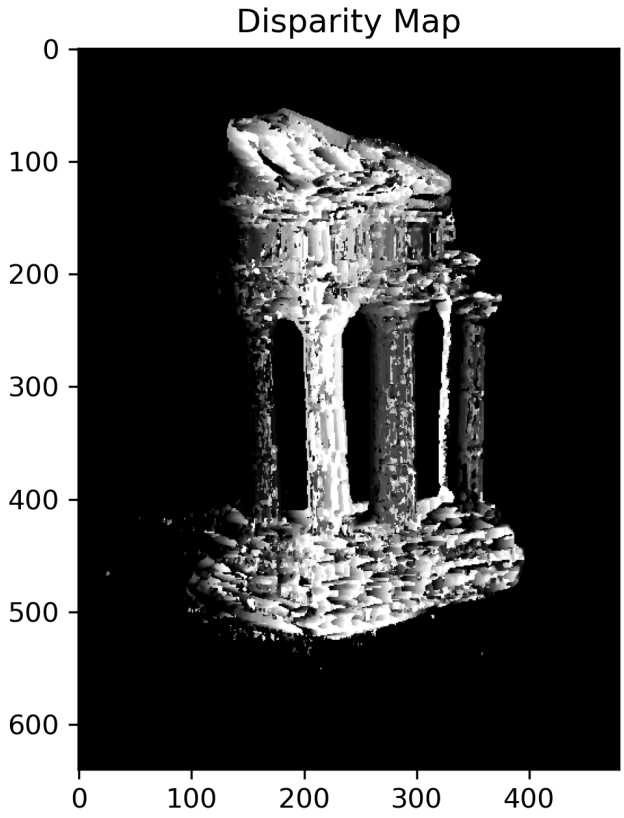
maxDisp is the maximum disparity and windowSize is the window size. The output dispM has the same dimension as im1 and im2. Since im1 and im2 are rectified, computing correspondences is reduced to a 1-D search problem.

The dispM(y, x) is

$$\texttt{dispM}(y,x) = \underset{0 \le d \le \texttt{maxDisp}}{\operatorname{argmin}} \operatorname{dist}(\texttt{im1}(y,x), \texttt{im2}(y, x-d)),$$

$$\text{dist}(\texttt{im1}(y,x),\texttt{im2}(y,x-d)) = \sum_{i=-w}^{w}\sum_{i=-w}^{w}(\texttt{im1}(y+i,x+j)-\texttt{im2}(y+i,x+j-d))^2$$

w is (windowSize-1)/2. This distance function is called the sum of squares difference. You can also try another metric, for example, normalized cross correlation, which is slower but more robust. The following is a sample output.




### 3.2.3 Depth map (2 pts)

Write a function that creates a depthmap from a disparity map (dispM).

get_depth(dispM,K1,K2,R1,R2,t1,t2)

Use the fact that depthM(y, x) = b × f /dispM(y, x) where b is the baseline and f is the focal length of camera. For simplicity, assume that b = // c1 − c2 // (i.e., distance between optical centers) and f = K1(1, 1). Finally, let depthM(y, x) = 0 whenever dispM(y, x) = 0 to avoid dividing by 0.

You can now test your disparity and depth map functions using testDepth.py. Be sure to have the rectification saved (by running testRectify.py). This function will rectify the images, then compute the disparity map and the depth map.

**In your write-up, include images of your disparity map and your depth map.**

## 3.3 Pose estimation

In this section, you will estimate both the intrinsic and extrinsic parameters of camera given 2D point x on image and their corresponding 3D points X. In other words, given a set of matched points {Xi,xi} and camera model (note that the following equation holds true up to scale)

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

we want to find the estimate of the camera matrix P ∈ R3×4, as well as intrinsic parameter matrix K ∈ R3×3, camera rotation R ∈ R3×3 and camera translation t ∈ R3, such that

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

### 3.3.1 Estimate camera matrix P (2 pts)

Write a function that estimates the camera matrix P given 2D and 3D points x, X.

estimate_pose(x, X),

where x is 2 × N matrix denoting the (x, y) coordinates of the N points on the image plane and X is 3 × 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 testPose.py to test your implementation.

**In your write-up, include the output of the script testPose.py**

### 3.3.2 Estimate intrinsic/extrinsic parameters (1 pts)

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

estimate_params(P)

The estimate_params function should consecutively run the following steps:

1. Compute the camera center c by using SVD. Hint: c is the eigenvector corresponding to the smallest eigenvalue.
2. Compute the intrinsic K and rotation R by using QR decomposition. K is a right upper triangle matrix while R is an orthonormal matrix. (See here for a reference https://math.stackexchange.com/questions/1640695/rq-decomposition)

3. Compute the translation by t = −Rc.

Once you finish your implementation, you can run the provided script testKRt.py.

**In your write-up, include the output of the script testKRt.py**

## 3.4 Multi-view stereo

Note that there are no template codes and you need to write all the codes from scratch. This part is a lot more challenging than the previous one.

We will implement a simpler version of an algorithm presented in “Multi-View Stereo Revisited” by Goesele, Curless, and Seitz. The paper can be found here. Our goal is to reconstruct a depthmap for one reference image. We will use 5 images in the data folder, named templeR00??.png. The corresponding camera parameters are found in templrR_par.txt. The following is the camera parameter format, given in the original dataset (multi-view stereo benchmark).

> *_par.txt: camera parameters. There is one line for each image. The format for each line is:
> "imgname.png k11 k12 k13 k21 k22 k23 k31 k32 k33 r11 r12 r13 r21 r22 r23 r31 r32 r33 t1 t2 t3"
> The projection matrix for that image is given by K*[R t]
> The image origin is top-left, with x increasing horizontally, y vertically

Just in case, the projection matrix is defined as the multiplication of the intrinsics and the extrinsics matrices as below.

$$P = \begin{pmatrix} k11 & k12 & k13 \\ k21 & k22 & k23 \\ k31 & k32 & k33 \end{pmatrix} \begin{pmatrix} r11 & r12 & r13 & t1 \\ r21 & r22 & r23 & t2 \\ r31 & r32 & r33 & t3 \end{pmatrix}$$

Given a 3D point (X, Y, Z), the corresponding pixel coordinate (x, y) is computed as

$$d \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = P \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

d is the “depth” of the point, which we will compute at every pixel in a reference image. Instead of a disparity estimation in the previous part, we will directly estimate a depth value in this part. Without loss of generality, suppose we seek to compute a depthmap for a reference image I0, using 3 other images I1, I2, and I3. The following is the algorithm.

```
// Depthmap Algorithm for reference image I0
For (every pixel p in I0) {
  If p is in the background (e.g., close to a black color or use a manual mask), skip.
  Consider SxS pixels centered around p. (e.g., S = 5).
  Let P denote S^2 pixels in the square region.
  For (d = min_depth; d < max_depth; d += depth_step) {
    For (every pixel q in P) {
      Compute the 3D coordinate that projects to q and has depth d by Get3dCoord(q, I0, d).
      // See below for Get3dCoord function.
    }
    Let X denote S^2 3D coordinates computed above.
    Compute the image consistency scores between (I0 and I1), (I0 and I2), and (I0 and I3).
    score01 <- ComputeConsistency(I0, I1, X).
    score02 <- ComputeConsistency(I0, I2, X).
    score03 <- ComputeConsistency(I0, I3, X).
    average(score01, score02, score03) is the consistency score of depth d.
    Remember the depth with the best consistency score.
    // You may want to skip if the average consistency score
    // (e.g., Normalized Cross Correlation) is below a certain threshold (e.g., 0.7).
  }
  Set the depth with the best consistency score for p.
}
```

```
// Compute a 3D coordinate that projects to pixel q in image I0 and has a depth d.
Get3dCoord(q, I0, d) {
  Let (x, y) denote the xy coordinate of q.
  Let P denote the 3x4 projection matrix of I0.
```

$$d \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} & P & \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} dx \\ dy \\ d \end{pmatrix} = \begin{pmatrix} P_{11} & P_{12} & P_{13} & P_{14} \\ P_{21} & P_{22} & P_{23} & P_{24} \\ P_{31} & P_{32} & P_{33} & P_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} dx - P_{14} \\ dy - P_{24} \\ d - P_{34} \end{pmatrix} = \begin{pmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

$$\begin{pmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{pmatrix}^{-1} \begin{pmatrix} dx - P_{14} \\ dy - P_{24} \\ d - P_{34} \end{pmatrix} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

```
  Return X, Y, Z
}
```

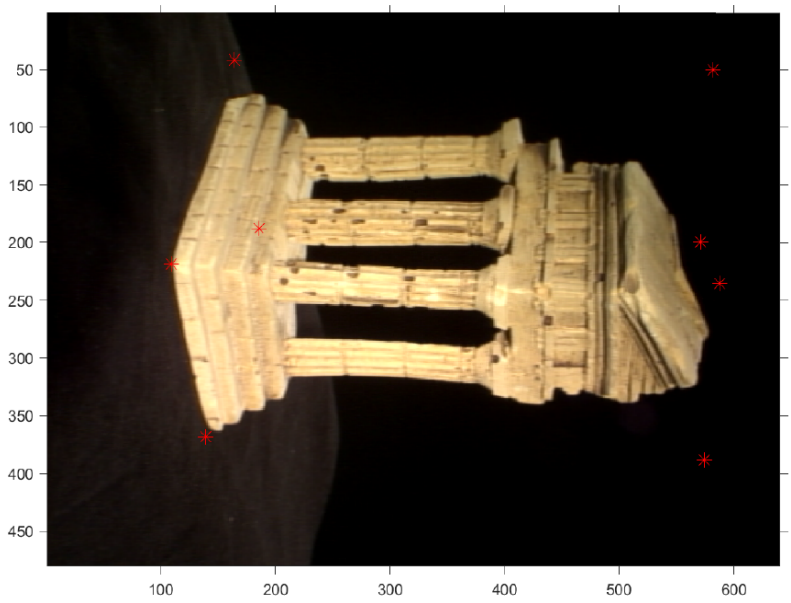
```
ComputeConsistency(I0, I1, X) {
  Project S^2 3d coordinates in X into image I0.
  Collect S^2 pixel colors C0.
  Project S^2 3d coordinates in X into image I1.
  Collect S^2 pixel colors C1.
  Return NormalizedCrossCorrelation(C0, C1).
}
```

```
NormalizedCrossCorrelation(C0, C1) {
  Compute average red, average green, and average blue of pixels in C0.
  Subtract average red, average green, and average blue from each pixel color in C0.
  Compute the L2 norm of all intensities (red, green, and blue together) in C0.
  Divide each pixel color by the L2 norm.

  Apply the above same normalization to C1.

  Consider C0 (and C1) to be a 1D vector (i.e., an array of normalized RGB intensities)
  Return a dot product between C0 and C1.
}
```

The sensitive hyperparameters are min_depth, max_depth, and depth_step. The algorithm searches for a depth in the range of [min_depth, max_depth] with an increment of depth_step. We can use the following information to compute these hyperparameters. The object is in the following axis-aligned bounding box [minx,miny,minz]=[-0.023121 -0.038009 -0.091940] & [maxx,maxy,maxz]=[0.078626 0.121636 -0.017395]. Given a camera I0, we compute 8 corners of the bounding box, project to I0, and compute their depth values. min_depth and max_depth can be set to the minimum and the maximum of the 8 depth values. depth_step should be set so that a projected pixel coordinate in the other images change around 0.5 pixel at every iteration.

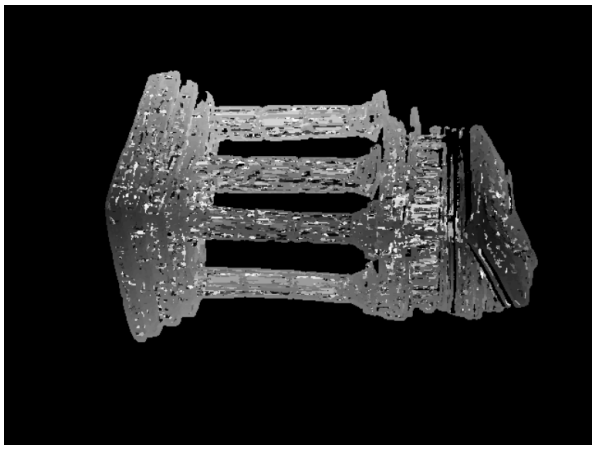
### 3.4.1 (1 pts)

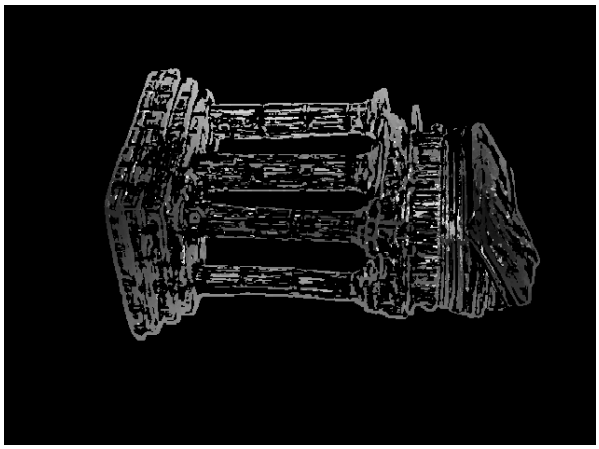
For debugging, compute the 8 corners of the axis-aligned bounding box [minx,miny,minz]=[-0.023121 -0.038009 -0.091940] & [maxx,maxy,maxz]=[0.078626 0.121636 -0.017395].
Project the corners to all the images and visualize the corners. **Include all the images with projected corners in the report.** The following is a sample image, showing what projected corners might look like.




## 3.4.2 (1 pts)

Compute a depthmap of I0 using the above algorithm, and visualize the depthmap in the same way as the previous part. **Include the depthmap visualization in the final report.** The depthmap quality should be higher compared to the 2-view case, and your visualization should demonstrate it. A depthmap image should look like the following examples. Despite some noisy depth values (white pixels), the overall quality must be much higher than the binocular case.





## 3.4.3 (1 pts)

Lastly, save a depthmap as a point cloud in the OBJ format, load it into software such as MeshLab, and visualize the point cloud in 3D. Concretely, for each pixel with a depth value, compute its 3D coordinate, and save that 3D coordinate as a 3D point. The code template has already lines to save a point cloud in the OBJ format. **Include the screenshots of the point**

**cloud with the visualization software in at least 3 viewing angles that clearly show the structure of the reconstruction.** The following is an example set.