Project 5
3D Reconstruction

Due date: 23:59 Tuesday 12/7th (2021)

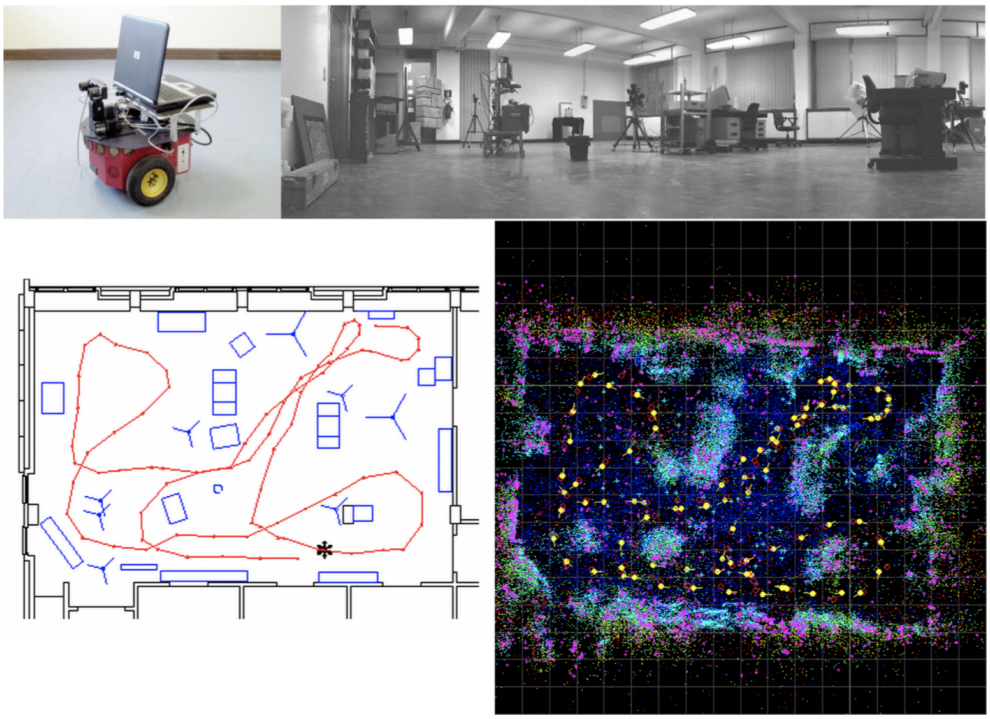
1. Instructions

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}
        - matlab
            - camera2.m
            - displayEpipolarF.m
            - eightpoint.m (Section 3.1.1)
            - epipolarCorrespondence.m (Section 3.1.2)
            - epipolarMatchGUI.m
            - essentialMatrix.m (Section 3.1.3)
            - get_depth.m (Section 3.2.3)
            - get_disparity.m (Section 3.2.2)
            - p2t.m
            - refineF.m
            - rectify_pair.m (Section 3.2.1)
            - testDepth.m
            - testKRt.m
            - testRectify.m
            - testTempleCoords.m (Section 3.1.5)
            - triangulate.m (Section 3.1.4)
            - warp_stereo.m
            - Any other helper functions you need
        - ec
            - estimate_params.m
            - estimate_pose.m
            - projectCAD.m
            - testPose.m
        - 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. File paths: Make sure that any file paths that you use are relative and not absolute so that we can easily run code on our end. For instance, you cannot write "imread('/some/absolute/path/data/abc.jpg')". Write "imread('../data/abc.jpg')" instead.
3. Project 3 has 22 pts.

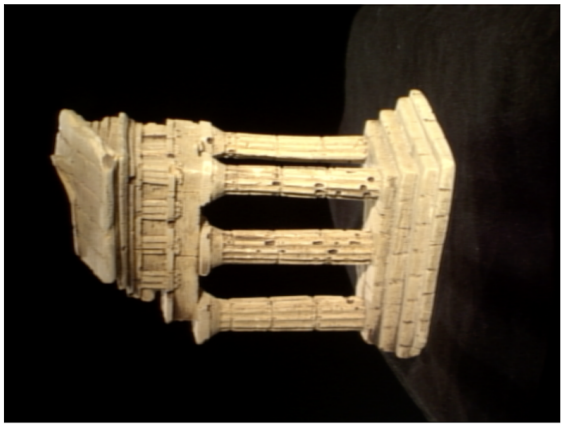
2. Overview

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 mat 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 mat files. someCorresps.mat contains good point correspondences. You will use this to compute the Fundamental matrix. Intrinsics.mat contains the intrinsic camera matrices, which you will need to compute the full camera projection matrices. Finally, templeCoords.mat 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.m (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.

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)

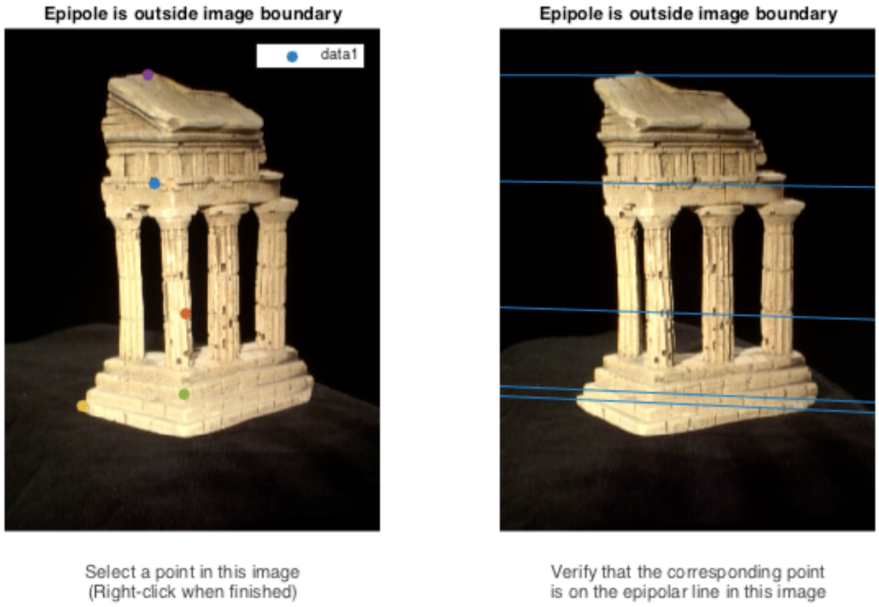
You will use the eight point algorithm to estimate the fundamental matrix. Please use the point correspondences provided in someCorresp.mat. Write a function with the following form:

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

X1 and x2 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 $F' = U\Sigma' V^T$.
- 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.m (takes in Fundamental matrix and the two sets of points), which you can call from eightpoint before unscaling F. This function uses matlab's fminsearch to 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.m (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.

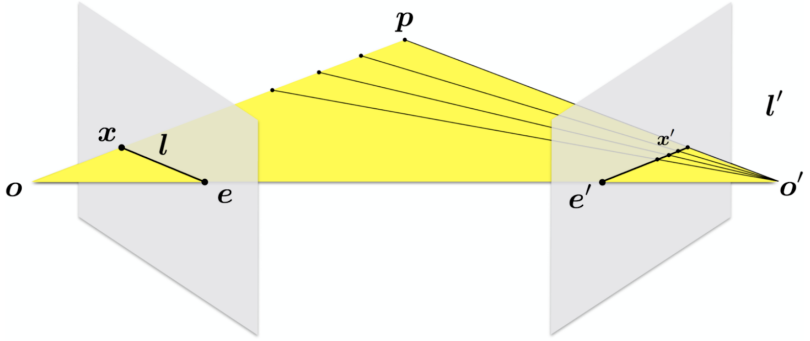


Figure 6: Epipolar Geometry (source Wikipedia)

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.

Write a function with the following form:

function pts2 = c(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.m 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. 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.

Write a function with the following form:

function E = 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.mat). E is the computed essential matrix. 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.

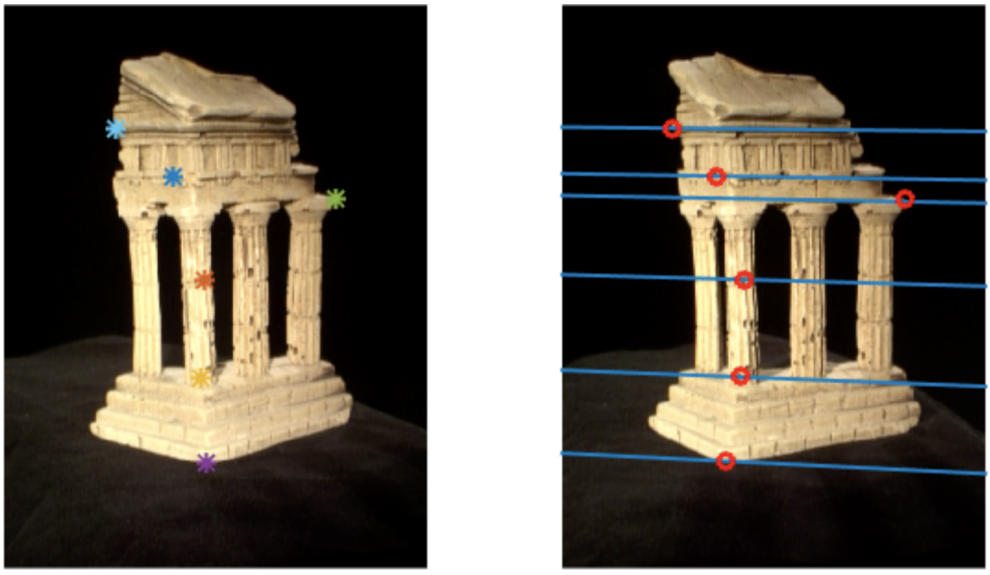


Figure 7: Epipolar Match visualization. A few errors are alright, but it should get most easy points correct (corners, dots, etc...)

3.1.4 Implement triangulation (2 pts)

Write a function to triangulate pairs of 2D points in the images to a set of 3D points with the form

function pts3d = triangulate(P1, pts1, P2, pts2)

pts1 and pts2 are the Nx2 matrices with the 2D image coordinates and pts3d is an Nx3 matrix with the corresponding 3D points (in all cases, one point per row). 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.m function to get four possible extrinsic matrices. 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.mat. 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.m that does the following:

1. Load the two images and the point correspondences from someCorresp.mat
2. Run eightpoint to compute the fundamental matrix F
3. Load the points in image 1 contained in templeCoords.mat and run your epipolarCorrespondences on them to get the corresponding points in image
4. Load intrinsics.mat and compute the essential matrix E.
5. Compute the first camera projection matrix P1 and use camera2.m to compute the four candidates for P2
6. Run your triangulate function using the four sets of camera matrix candidates, the points from templeCoords.mat 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. Use matlab's plot3 function to plot these point correspondences on screen. Please type "axis equal" after "plot3" to scale axes to the same unit.
9. Save your computed rotation matrix (R1, R2) and translation (t1, t2) to the file ../data/extrinsics.mat. 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.

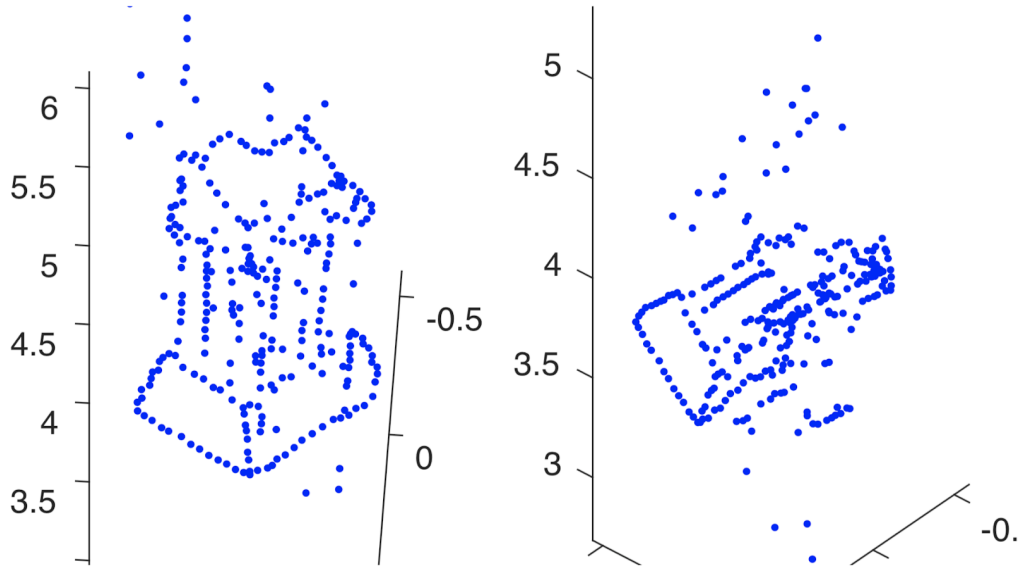


Figure 8: Sample Reconstructions

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 program that computes rectification matrices.

function [M1, M2, K1n, K2n, R1n, R2n, t1n, t2n] = 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. You can test your function using the provided script testRectify.m.

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.m (Be sure to have the extrinsics saved by your testTempleCoords.m). 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 temple.mat, which will be used by the next test script testDepth.m.

In your write-up, include a screenshot of the result of running testRectify.m 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).

function dispM = get_disparity(im1, im2, maxDisp, windowSize)

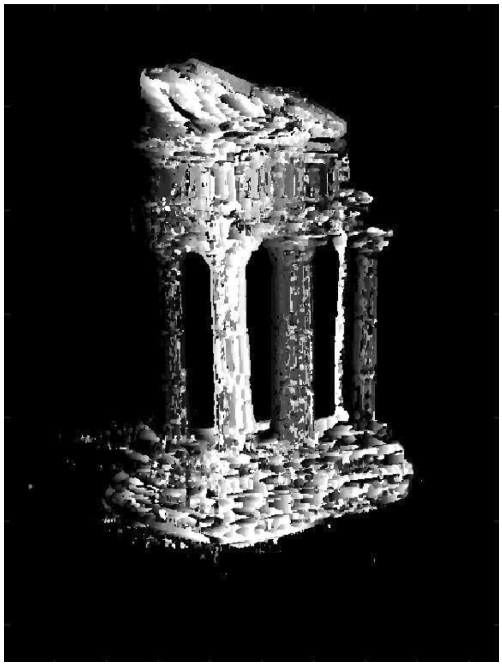
maxDisp is the maximum disparity and windowSize is the window size. The output dispM has the same dimension as im1 and im2. Since im1 and im2 are rectified, computing correspondences is reduced to a 1-D search problem.

The dispM(y, x) is

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

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

w is (windowSize-1)/2. This summation on the window can be easily computed by using the conv2 Matlab function (i.e. convolve with a mask of ones(windowSize,windowSize)). 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).

function depthM = 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.m. Be sure to have the rectification saved (by running testRectify.m). Also, you need to modify testDepth.m to apply rectification. 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.

function P = 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.m to test your implementation.

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

3.3.2 Estimate intrinsic/extrinsic parameters (2 pts)

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

function [K, R, t] = estimate_params(P)

The estimate_params should consecutively run the following steps:

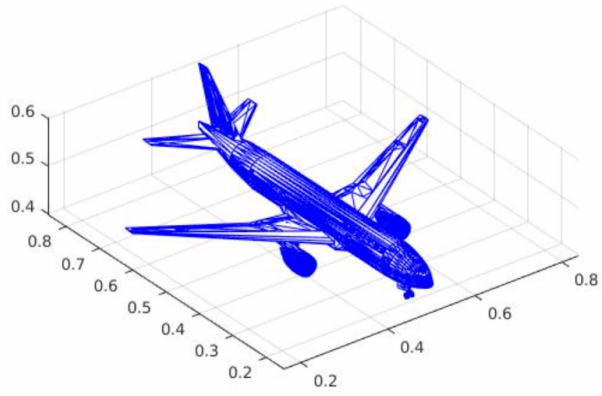
1. Compute the camera center c by using SVD. Hint: c is the eigenvector corresponding to the smallest eigenvalue.
2. Compute the intrinsic K and rotation R by using QR decomposition. K is a right upper triangle matrix while R is an orthonormal matrix. (See here for a reference https://math.stackexchange.com/questions/1640695/rq-decomposition)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

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

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

3.3.3 Project a CAD model to the image (2 pts)

Now you will utilize what you have implemented to estimate the camera matrix from a real image, shown at the left below, and project the 3D object (CAD model), shown at the right below, back on to the image plane.



Write a script projectCAD.m, which does the following:

1. Load an image, a CAD model cad, 2D points x and 3D points X from PnP.mat.
2. Run estimate_pose and estimate_params to estimate camera matrix P, intrinsic matrix K, rotation matrix R, and translation t.
3. Use your estimated camera matrix P to project the given 3D points X onto the image.
4. Plot the given 2D points x and the projected 3D points on screen. An example is shown at the left below. Hint: use plot.
5. Draw the CAD model rotated by your estimated rotation R on screen. An example is shown at the middle below. Hint: use trimesh.
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 at the right below. Hint: use patch.

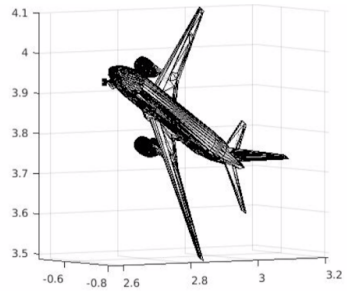


Figure: Project a CAD model back onto the image. Left: the image annotated with given 2D points (blue circle) and projected 3D points (red points). Middle: the CAD model rotated by estimated R. Right: the image overlapping with projected CAD model.

In your write-up, include the three images similar to the above figure. You have to use different colors from the figure. 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.