16720-B F20 3D Reconstruction & Photometric Stereo

Instructor: Kris Kitani (Carnegie Mellon University)
TAs: Haoshuo H, Han D, Anand B, Alireza G, Nadine C, Rawal K, Zhengyi L

Total points: 170pts, EC: 45pts

Due November 16, 2020 11:59 PM

Instructions

1. **Integrity and collaboration**: Students are encouraged to work in groups but each student must submit their own work. Include the names of your collaborators in your write up. Code should **NOT** be shared or copied. Please properly give credits to others by **LISTING EVERY COLLABORATOR** in the writeup including any code segments that you discussed, Please **DO NOT** use external code unless permitted. Plagiarism is prohibited and may lead to failure of this course.

2. **Start early!** This homework will take a long time to complete.

3. **Questions:** If you have any question, please look at Piazza first and the FAQ page for this homework.

4. All questions marked with a **Q** require a submission, for theory part please answer the question in the writeup, for the practice part in section 3 and 4 please submit both code, `.npy` file if needed and short answer if question is asked.

5. **For the implementation part, please stick to the headers, variable names, and file conventions provided.**

6. **Attempt to verify your implementation as you proceed:** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.

7. **Do not import external functions/packages other than the ones already imported in the files:** The current imported functions and packages are enough for you to complete this assignment.

8. **Submission**: We have provided a script checkA4Submission.py which will check if you have all the files needed for submission. The submission is on Gradescope, **you will be submitting both your writeup and code zip file**. The zip file, andrew-id.zip, contains your code and any results files we ask you to save. **Note: You have to submit your writeup separately to Gradescope, and include results in the writeup**. Do **not** submit anything from the *data/* folder in your submission. Lastly, please remember to match the writeup pages to the appropriate questions in Gradescope.

9. Assignments that do not follow this submission rule will be **penalized up to 10% of the total score**.

10. Please make sure that the file paths that you use are relative and not absolute.

Contents

1 Theory: 3D Reconstruction (25 pts)

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 pts) Suppose two cameras fixated on a point x (see Figure 1) in space such that their principal axes intersect at that point. Show that if the image coordinates are normalized so that the coordinate origin $(0, 0)$ coincides with the principal point, the $\mathbf{F}_{33}$ element of the fundamental matrix is zero.

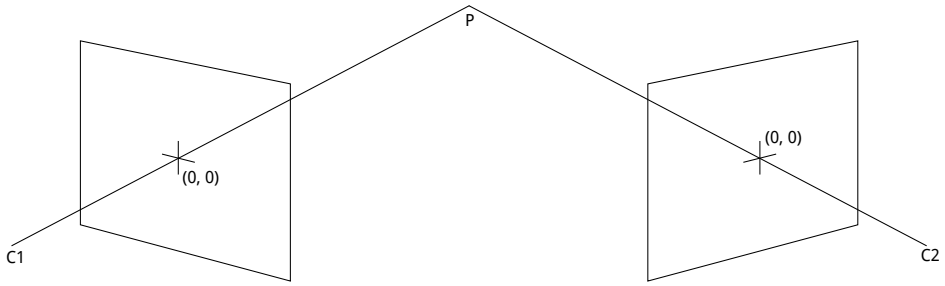


Figure 1: Figure for Q1.1. $C1$ and $C2$ are the optical centers. The principal axes intersect at point $\mathbf{w}$ (P in the figure).

Q1.2 (5 pts) Consider the case of two cameras viewing an object such that the second camera differs from the first by a *pure translation* that is parallel to the x-axis. Show that the epipolar lines in the two cameras are also parallel to the x-axis. Backup your argument with relevant equations.

Q1.3 (5 pts) Suppose we have an inertial sensor which gives us the accurate extrinsics $\mathbf{R}_i$ and $\mathbf{t}_i$ (Figure 2), the rotation matrix and translation vector of the robot at time i. What will be the effective rotation ($\mathbf{R}_{rel}$) and translation ($\mathbf{t}_{rel}$) between two frames at different time stamps? 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 $\mathbf{t}_{rel}$.

Q1.4 (10 pts) Suppose that a camera views an object and its reflection in a plane mirror. Show that this situation is equivalent to having two images of the object which are related by a skew-symmetric fundamental matrix. You may assume that the object is flat, meaning that all points on the object are of equal distance to the mirror (*Hint:* draw the relevant vectors to understand the relationship between the camera, the object, and its reflected image.)

2 Theory: Photometric Stereo (10 pts, EC 15pts)

As we discussed in class, photometric stereo is a physics-based method to determine the shape of an object from its appearance under a set of lighting directions. In the below theory

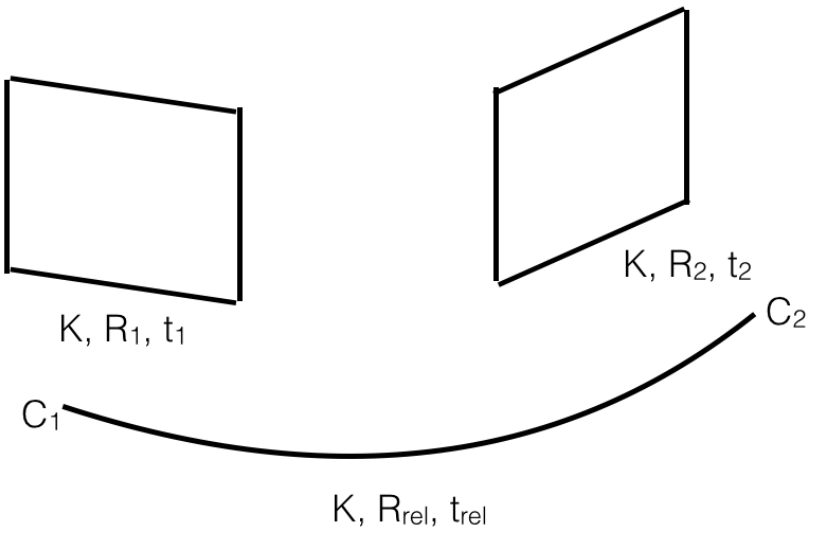


Figure 2: Figure for Q1.3. $C1$ and $C2$ are the optical centers. The rotation and the translation is obtained using inertial sensors. $\mathbf{R}_{rel}$ and $\mathbf{t}_{rel}$ are the relative rotation and translation between two frames.

questions, we make certain assumptions: the object is Lambertian and is imaged with an orthographic camera under a set of directional lights. We will look into some questions for calibrated/uncalibrated photometric stereo and we will solve a calibrated photometric stereo problem, in which the lighting directions are given.

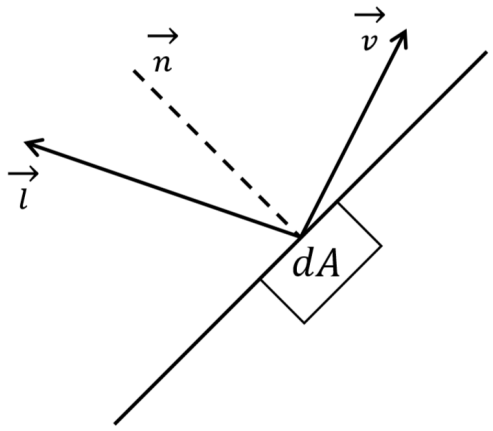


Figure 3: Geometry of photometric stereo

Q2.1 Understanding n-dot-l lighting (5 pts) Explain the geometry of the n-dot-l lighting model from Fig.3 Where does the dot product come from? Where does the projected area come into the equation? Why does the viewing direction not matter?

Q2.2 Normals and depth (5 pts) To estimate from the normals of the actual shape, we represent the shape of the surface as a 3D depth map given by a function $z = f(x, y)$. Let the normal at the point (x, y) be $\mathbf{n} = (n_1, n_2, n_3)$. Explain, in your write-up, why n is related to the **partial derivatives** of f at (x, y): $f_x = \partial f(x, y)/\partial x = -n_1/n_3$ and $f_y = \partial f(x, y)/\partial y = -n_2/n_3$. You may consider the 1D case where $z = f(x)$.

Q2.3 Extra Credit: Understanding integrability of gradients (5 pts) Consider the 2D discrete function g on the space given by the matrix below. Find its x and y gradients, given that the gradients are calculated as $g_x(x_i, y_i) = g(x_{i+1}, y_j) - g(x_i, y_j)$ for all i, j (and similar for y). Let us define $(0, 0)$ as the top left, with x going in the horizontal direction and y in the vertical.

$$g = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 8 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Note that we can reconstruct the entire of g given the values at its boundaries using g_x and g_y. Given that $g(0, 0) = 1$. perform these two procedures.

1. Use g_x to construct the first row of g, then use g_y to construct the rest of g.
2. Use g_y to construct the first column of g, then use g_x to construct the rest of g.

Are these the same?

Note that these are two ways to reconstruct g from its gradients. Given arbitrary g_x and g_y, these two procedures will not give the same answer, and therefore this pair of gradients does not correspond to a true surface. Integrability implies that the value of g estimated in both these ways (or any other way you can think of) is the same. How can we modify the gradients you calculated above to make g_x and g_y non-integrable? Why may the gradients estimated in this way be non-integrable? Note all this down in your write-up.

Q2.4 Extra Credit: Uncalibrated normal estimation (10 pts) Recall the relation $\mathbf{I} = L^T B$. Here, the L is $3 \times N$ matrix showing the position of light-source where N is number of light-sources, and B is is the pseudo-normal with shape $3 \times P$, where P is number of pixels in the image. We know neither L nor B. Therefore, this is a matrix factorization problem with the constraint that the estimated $\hat{L}$ and $\hat{B}$, the rank of $\hat{I} = \hat{L}^T B$ be, and the estimated $\hat{I}$ and $\hat{L}$ have appropriated dimensions.

It is well known that the best rank-k approximation to a $m \times n$ matrix $\mathbf{M}$, where $k \leq \min(m, n)$ is calculated as the following: perform a singular value decomposition SVD $\mathbf{M} = \mathbf{U}\Sigma\mathbf{V}^T$, set all singular values except the top k from Σ to 0 to get the matrix $\hat{\Sigma}$, and reconstitute $\hat{M} = \mathbf{U}\hat{\Sigma}\mathbf{V}^T$. Explain in your write-up how this can be used to construct a factorization of the form detailed above following the required constraints.

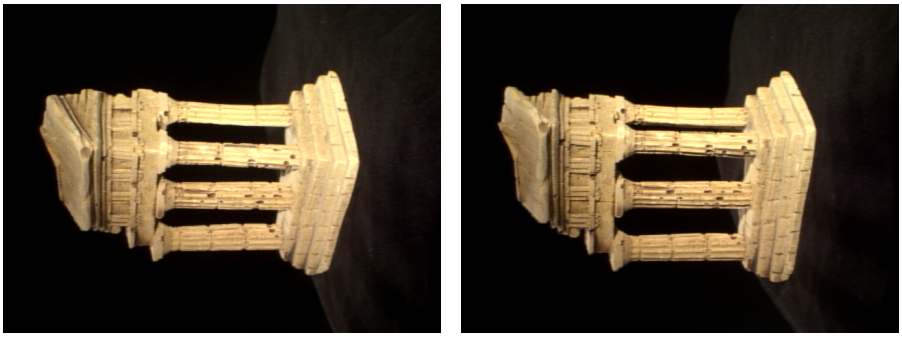


Figure 4: `Temple` images for this assignment

3 Practice: 3D Reconstruction (100 pts, EC 15 pts)

3.1 Overview

In this part, you will begin by implementing the two different methods seen in class to estimate the fundamental matrix from the corresponding points in two images (subsection 3.2). Next, given the fundamental matrix and the 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 (subsection 3.3). Then, you will implement a method to automatically match points taking advantage of epipolar constraints and make a 3D visualization of the results (subsection 3.4). Finally, you will implement RANSAC and bundle adjustment to further improve your algorithm (subsection 3.5).

3.2 Fundamental matrix estimation

In this section, you will explore different methods of estimating the fundamental matrix given a pair of images. In the `data/` directory, you will find two images (see Figure 4) from the Middlebury multiview dataset[1], which is used to evaluate the performance of modern 3D reconstruction algorithms.

The Eight Point Algorithm

The 8-point algorithm (discussed in class, and outlined in Section 10.1 of Forsyth & Ponce) is arguably the simplest method for estimating the fundamental matrix. For this section, you can use the provided correspondences you can find in `data/some_corresp.npz`.

Q3.2.1 (10 pts) Finish the function `eight-point` in `submission.py`. Make sure you follow the signature for this portion of the assignment:

```
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.

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

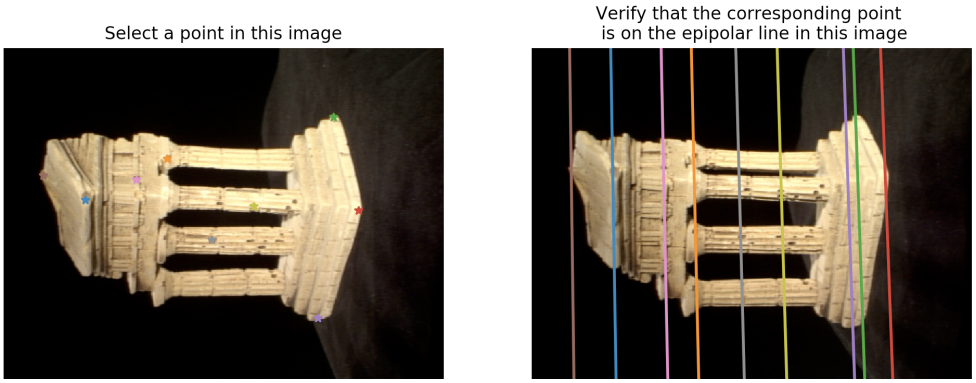


Figure 5: `displayEpipolarF in helper.py` creates a GUI for visualizing epipolar lines

- 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 the $\mathbf{F}$ 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 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).

- To visualize the correctness of your estimated $\mathbf{F}$, use the supplied function `displayEpipolarF` in `helper.py`, which 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 (Figure 5).

- **Output:** Save your matrix $\mathbf{F}$, scale `M` to the file `q3_2_1.npz`.
  In your write-up: Write your recovered $\mathbf{F}$ and include an image of some example outputs of `displayEpipolarF`.

The Seven Point Algorithm

Q3.2.2 (Extra Credits, 15 pts) Since the fundamental matrix only has seven degrees of freedom, it is possible to calculate $\mathbf{F}$ using only seven point correspondences. This requires solving a polynomial equation. In the section, you will implement the seven-point algorithm (described in class, and outlined in Section 15.6 of Forsyth and Ponce). Manually select 7 points from provided point in `data/some_corresp.npz`, and use these points to recover a fundamental matrix $\mathbf{F}$. The function should have the signature:

```
Farray = sevenpoint(pts1, pts2, M)
```

where `pts1` and `pts2` are 7×2 matrices containing the correspondences and `M` is the normalizer (use the maximum of the images' height and width), and `Farray` is a list array of length either 1 or 3 containing Fundamental matrix/matrices. Use `M` to normalize the point values between $[0, 1]$ and remember to "unnormalize" your computed $\mathbf{F}$ afterwards.

- Use `displayEpipolarF` to visualize $\mathbf{F}$ and pick the correct one.
- **Output:** Save your matrix $\mathbf{F}$, scale `M`, 2D points `pts1` and `pts2` to the file `q3_2_2.npz`. **In your write-up:** Write your recovered $\mathbf{F}$ and print an output of `displayEpipolarF`. Also, include an image of some example output of `displayEpipolarF` using the seven point algorithm.
- *Hints:* You can use Numpy's function `roots()`. The epipolar lines may not match exactly due to imperfectly selected correspondences, and the algorithm is sensitive to small changes in the point correspondences. You may want to try with different sets of matches.

3.3 Metric Reconstruction

You will compute the camera matrices and triangulate the 2D points to obtain the 3D scene structure. To obtain the Euclidean 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 $\mathbf{K}_1$ and $\mathbf{K}_2$ are known; these are provided in `data/intrinsics.npz`.

Q3.3.1 (5 pts) Write a function 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)
```

In your write-up: Write your estimated $\mathbf{E}$ using $\mathbf{F}$ from the eight-point algorithm.

Given an essential matrix, it is possible to retrieve the projective camera matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ from it. Assuming $\mathbf{M}_1$ is fixed at $[\mathbf{I}, 0]$, $\mathbf{M}_2$ can be retrieved up to a scale and four-fold rotation ambiguity. For details on recovering $\mathbf{M}_2$, see section 7.2 in Szeliski. We have provided you with the function `camera2` in `python/helper.py` to recover as the four possible $\mathbf{M}_2$ matrices given $\mathbf{E}$.

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

Q3.3.2 (10 pts) Using the above, write a function 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. Remember that you will need to multiply the given intrinsics matrices with your solution for the canonical camera matrices to obtain the final camera matrices. Various methods exist for triangulation - probably the most familiar for you is based on least squares (see Szeliski Chapter 7 if you want to learn about other methods):

For each point i, we want to solve for 3D coordinates $\mathbf{w}_i = \begin{bmatrix} x_i, y_i, z_i \end{bmatrix}^T$, 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:

$$\mathbf{A}_i w_i = 0 \tag{1}$$

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 $\mathbf{w}_i$.

In your write-up: Write down the expression for the matrix $\mathbf{A}_i$.

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 $\widehat{\mathbf{x}_{2i}} = Proj(\mathbf{C}_2, \mathbf{w}_i)$.

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

Q3.3.3 (10 pts) Write a script `findM2.py` to obtain the correct `M2` from `M2s` by testing the four solutions through triangulation. Use the correspondences from `data/some_corresp.npz`.
Output: Save the correct `M2`, the corresponding `C2`, and 3D points `P` to `q3_3_3.npz`.

3.4 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.

Q3.4.1 (15 pts) Implement a function 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 **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 Szeliski Chapter 11, on stereo matching, for a brief overview of these and other methods.

Implementation hints:

- Experiments 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 `epipolarMatchGUI` in `python/helper.py`, which takes in two images the fundamental matrix. This GUI allows you to click on a point in `im1`, and will use your function to display the corresponding point in `im2`. See Figure 6.

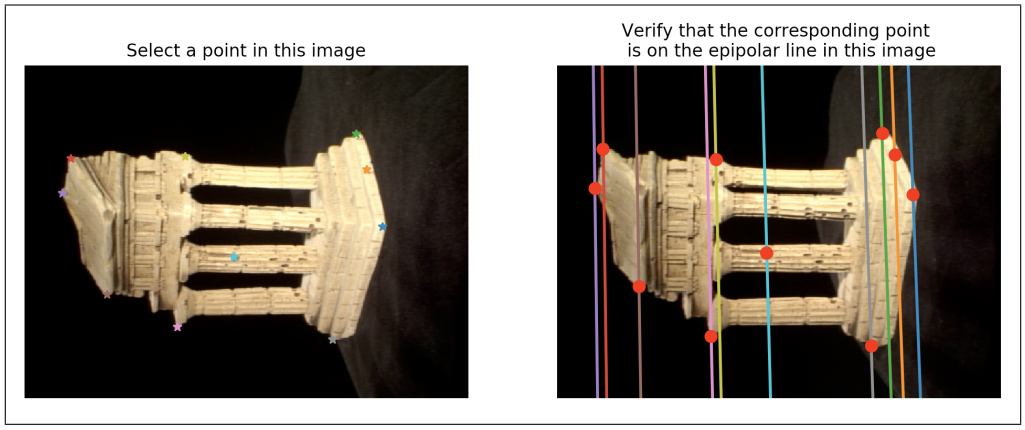


Figure 6: `epipolarMatchGUI` shows the corresponding point found by calling `epipolarCorrespondence`

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.

Output: Save the matrix **F**, points `pts1` and `pts2` which you used to generate the screenshot to the file `q3_4_1.npz`.

In your write-up: Include a screenshot of `epipolarMatchGUI` with some detected correspondences.

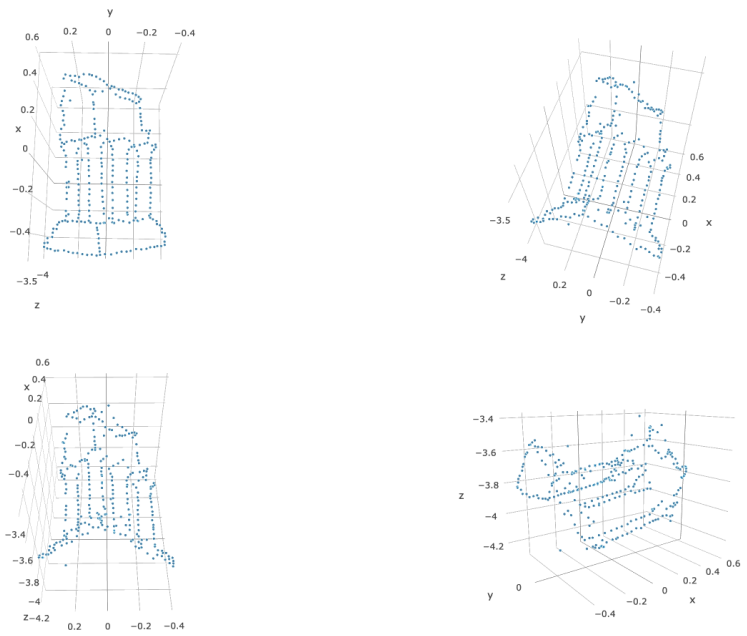


Figure 7: An example point cloud

Q3.4.2 (10 pts) 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. Write a script `visualize.py`, which loads the necessary files from `../data/` to generate the 3D reconstruction using `scatter`. An example is shown in Figure 7.

Output: Again, save the matrix $\mathbf{F}$, matrices $\mathbf{M}1, \mathbf{M}2, \mathbf{C}1, \mathbf{C}2$ which you used to generate the screenshots to the file `q3_4_2.npz`.

In your write-up: Take a few screenshots of the 3D visualization so that the outline of the temple is clearly visible, and include them with your homework submission.

3.5 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. Specifically,

- In Q3.5.1, you need to implement a RANSAC algorithm to estimate the fundamental matrix F and all the inliers.
- In Q3.5.2, you will need to write code to parameterize Rotation matrix $\mathbf{R}$ using Rodrigues formula [2], which will enable the joint optimization process for Bundle Adjustment.

[2]Please check this pdf for a detailed explanation.

- In Q3.5.3, you will need to first write down the objective function in rodriguesResidual, and do the bundleAdjustment.

Q3.5.1 (15 pts) In some real world applications, manually determining correspondences is infeasible and often there will be noisy correspondences. Fortunately, the RANSAC method seen in class can be applied to the problem of fundamental matrix estimation.

Implement the above algorithm with the signature:

```
[F, inliers] = ransacF(pts1, pts2, M)
```

where `M` is defined in the same way as in subsection 3.2 and inliers is a boolean vector of size equivalent to the number of points. Here inliers are set to true only for the points that satisfy the threshold defined for the given fundamental matrix F.

We have provided some noisy cooerespondances in `some_corresp_noisy.npz` in which around 75% of the points are inliers. Compare the result of RANSAC with the result of the eight-point algorithm when ran on the noisy correspondences. Briefly, explain the error metrics you used, how you decided which points were inliers, and any other optimizations you may have made.

Q3.5.2 (15 pts) So far we have independently solved for the camera matrix, $\mathbf{M}_j$ and 3D projections, $\mathbf{w}_i$. In bundle adjustment, we will jointly optimize the reprojection error with respect to the points $\mathbf{w}_i$ and the camera matrix $\mathbf{w}_j$.

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

where $\mathbf{w}_j = \mathbf{K}_j\mathbf{M}_j$.

For this homework, we are going to only look at optimizing the extrinsic matrix. The rotation matrix forms the Lie Group $\mathbf{SO}(3)$ that doesn't satisfy the addition operation so it cannot be directly optimized. Instead, we parameterize the rotation matrix to axis angle using Rodrigues formula to the Lie Algebra $\mathfrak{so}(3)$, which is defined in $\mathbb{R}^3$. through which the least squares optimization process can be done to optimize the axis angle. Try to implement function

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

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

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

Q3.5.3 (10 pts)

In this section, you need to implement the bundle adjustment algorithm. Using the parameterization you implemented in the last question, write an objective function for the extrinsic optimization:

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

where x is the flattened concatenation of $\mathbf{w}$, $\mathbf{r}_2$, 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 $\mathbf{M}_2$; $p1$ and $p2$ are 2D coordinates of points in image 1 and 2, respectively. The `residuals` are the difference between the original image projections and the estimated projections (the square of 2-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])])
```

Use this objective function and Scipy's nonlinear least squares optimizer `leastsq` write a function to optimize for the best extrinsic matrix and 3D points using the inlier correspondences from `some_corresp_noisy.npz` and the RANSAC estimate of the extrinsics and 3D points as an initialization.

```
[M2, w] = bundleAdjustment(K1, M1, p1, K2, M2_init, p2, p_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.

In your write-up: include an image of the original 3D points and the optimized points as well as the reprojection error with your initial $\mathbf{M}_2$ and $\mathbf{w}$, and with the optimized matrices.

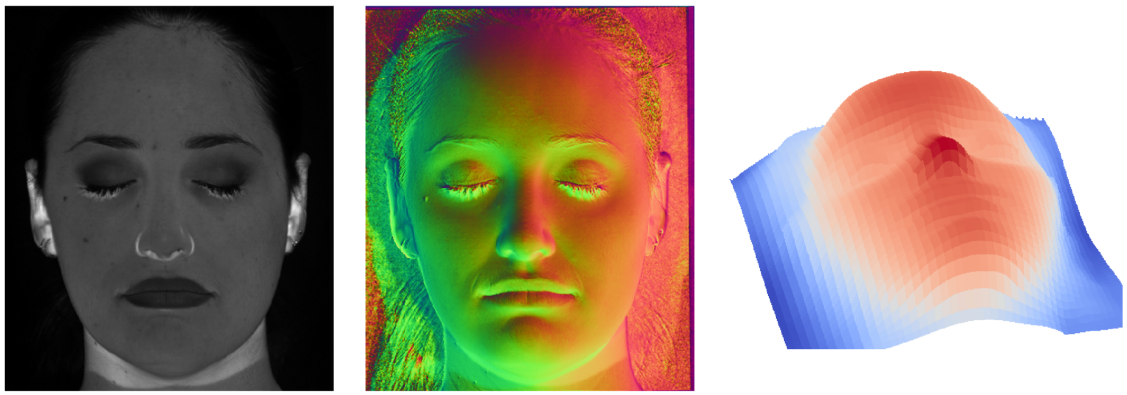
4 Practice: Calibrated Photometric Stereo (35 pt, EC 15 pts)

Overview

In this part you will first try to create an lambertian sphere to understand the n-dot-l lighting. Then, we will try to invert the image formation process you have done for the lambertian sphere. Seven images of a face lit from different directions are given to us, along with the ground-truth directions of light sources. The images live in the `data/` directory, named as `input_n.tif` for the n^{th} image. The source directions are given in the file `data/source.npy`.

4.1 Lambertian Sphere Rendering

Q4.1 Rendering n-dot-l lighting (10 pts) Consider a uniform **fully reflective Lambertian sphere** with its center at the origin and a radius of 5 cm. An orthographic camera located at (0, 0, 10) cm looks towards the negative z-axis with its sensor axes aligned with the x and y axes. The pixels on the camera are squares 5 μm in size, and the resolution of the camera is 3000×2500 pixels. Simulate the appearance of the sphere under the n-dot-l model with directional light sources with incoming lighting directions $(1, 1, 1)/\sqrt{3}$, $(1, -1, 1)/\sqrt{3}$ and $(-1, -1, 1)/\sqrt{3}$ in the function `renderNDotLSphere`.
Make sure that the length are in the same unit during implementation. To generate the



(a) (b) (c)

Figure 8: Results from calibrated photometric stereo: (a) albedos in the 'gray' colormap. (b) normals in the 'rainbow' colormap and (c) 3D reconstruction of the face.

normal for the sphere surface, use the fact that $\sqrt{x^2+y^2+z^2} = r$, here we can assume x and y are known. Then calculate the intensity of pixel on the image as $I = L^T \cdot B$ where L is the light direction and B is the pseudo normal.
Note that your rendering isn't required to be absolutely radiometrically accurate: we only need to evaluate the n-dot-l model. Include the rendering result in your writeup. (Hint: Recall that for the orthographic projection along the Z axis, consider homogeneous coordinates, the projection matrix is represented by the form

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This mapping takes a point $(X, Y, Z, 1)^T$ to the image point $(X, Y, 1)^T$, dropping the z coordinate.)

4.2 Estimating pseudonormals and albedos

Q4.2.1 Loading data (5 pts) In the function `loadData`, read the images into Python and do the steps as below.

1. Convert the images into the XYZ color space and extract the luminance channel use provided function `lRGB2XYZ` in `utils.py`.Noted that the Y channel is the luminance channel.

2. Vectorize these luminance images and stack them in a $7 \times P$ matrix, where P is the number of pixels in each image. This is the matrix I, which is given to us by the camera.

3. Load the sources file and covert it to a 3×7 matrix L.

Q4.2.2 Estimating pseudonormals (10 pts) Since we have more measurements (7 pixel) than variables (3 per pixel), we will estimate the pseudonormals using least-squares.

Note that there is a linear relation between $\mathbf{I}$ and $\mathbf{B}$ through $\mathbf{L}$: therefore, we can write a linear system of the form $\mathbf{Ax} = \mathbf{y}$ out of the relation $\mathbf{I} = \mathbf{L}^T\mathbf{B}$ and solve it to get $\mathbf{B}$. Solve this linear system in the function `estimatePseudonormalsCalibrated`. Estimate per-pixel albedos and normals from this matrix in `estimateAlbedosNormals`. In your write-up, mention how you construct the matrix A and the vector y.

Note that here matrices you end up creating might be huge and might not fit in your computer's memory. In that case, to prevent your computer from freezing or crashing completely, make sure to use the `sparse` module form `scipy`. You might also want to use the sparse linear system solver in the same module.

Q4.2.3 Albedos (5 pts) Note that the albedos are the magnitudes of the pseudonormals because of the definition of the pseudonormals. Recall the definition of pseudonormals $\mathbf{B}$ defined in : $\mathbf{I} = \mathbf{L}^T\mathbf{B}$

Calculate the albedos, reshape them into the original size of the images and display the resulting image in the function `displayAlbedosNormals`. Include the image in your write-up and comment on any unusual or unnatural features you may find in the albedo image, and on why this might be happening. Make sure to display in the `gray` colormap.

Q4.2.4 Normals (5 pts) The per-pixel, normalized normals can be viewed as an RGB image. Reshape the estimated normals into an image with 3 channels and display it in the function `displayAlbedosNormals`. Note that the components of these normals will have values in the range [-1, 1]. You will need to rescale them so that they lie in [0, 1] to display them properly as RGB images. Include this image in the write-up. Do the normals match your expectation of the curvature of the face? Make sure to display in the `rainbow` colormap.

4.3 Depth recovering from normals

Q4.3.1 Extra Credit: Normal Integration using Frankot-Chellappa algorithm (10 points) Read the implementation of `integrateFrankot` in `utils.py`, briefly list the steps (in words & math equations) to show how the integrability is enforced in the algorithm. Use the algorithm provided to get an estimate of the depth map $f(x, y)$.

Write a function `estimateShape` to apply the Frankot-Chellappa algorithm to your estimated normals. Once you have the function $f(x, y)$, plot it as a surface in the function `plotSurface` and include some significant viewpoints in your write-up.

The 3D projection from `mpl_toolkits.mplot3d.Axes3D` along with the function `plot_surface` might be of help. Make sure to plot this in the `coolwarm` colormap.

Q4.3.2 Extra Credit: Undefined quantities (5 pts) The value of $F(0, 0)$ was left undefined by the algorithm above. Here the $F(0, 0)$ is the 2D Fourier transform values at zero frequency along x and y axis. Note in your write-up what this tells us in terms of $f(x, y)$. (Hint: the value of the Fourier transform at zero frequency is the average of the image-domain function).

5 Deliverables

If your andrew id is `bovik`, your submission should be the writeup `bovik.pdf` and a zip file `bovik.zip`. **Please submit the zip file to Canvas and the pdf file to Gradescope.**

The zip file should include the following directory structure:

- ☐ `submission.py`: your implementation of algorithms for section 3 and section 4
- ☐ `findM2.py`: script to compute the correct camera matrix.
- ☐ `visualize.py`: script to visualize the 3d points.
- ☐ `q3_2_1.npz`: file with output of Q3.2.1.
- ☐ `q3_2_2.npz`: file with output of Q3.2.2.
- ☐ `q3_3_3.npz`: file with output of Q3.3.3.
- ☐ `q3_4_1.npz`: file with output of Q3.4.1.
- ☐ `q3_4_2.npz`: file with output of Q3.4.2.
- ☐ Answer all questions in the writeup.

References