Assignment 08

Please refer to MyCourses/Gradescope for assignment deadlines.

The goal of this assignment is to give you hands-on practice with the following concepts:

- Stereo vision and disparity (working out 3D scene depth given that we know where the cameras are)
- Camera pose estimation (working out where the cameras are from the images)

Part 0 of 2: Download Data

For the stereo and pose-estimation parts of this assignment, we need some image data. Your first task is to download the 2005 Middlebury Stereo Vision dataset from the following link: https://vision.middlebury.edu/stereo/data/scenes2005/. You should specifically download the "ALL-2views.zip (4.7MB)" zip file. There is some further crucial information on that website about the dataset format and camera calibration information.

For the second part of this assignment, the challenge is to infer where the cameras are in space given some images of a 3D object. For this, we'll also borrow from the folks at Middlebury and use their Multiview dataset, available here: https://vision.middlebury.edu/mview/data/. Specifically, we'll focus on inferring camera pose from nearby images of the Temple object. We recommend downloading the "TempleRing data set (11Mb)" for local development and testing.

Also be sure to `pip install` everything in the requirements file to run the 3D visualization tools.

Part 1 of 2: Dense Stereo

This part will use the Middlebury Stereo Vision dataset.

Each subfolder contains `view1.png` and `view5.png` – two rectified stereo images of the same scene – and a `disp1.png` and `disp2.png` file which contains ground-truth disparities at each pixel. Recall that disparity is inversely proportional to depth, so the disparity images will be *brighter* where objects are *closer* to the camera. You will notice that the disparity maps contain some black pixels around the edges of some objects. These are pixels where there is no true corresponding pixel in the other image, e.g. if an object is occluded by another object in one view but not another.

The Middlebury vision site linked above contains information about the camera setup such as baseline distance and focal length of the camera, as well as information on how to interpret the values in the disparity maps (hint: since disparities are stored in png files, they have been scaled to fit in the 0-255 range, but the 'true' disparities are on a different scale, and each image pair has a `dmin.txt` file that is required for converting disparities to actual depths, although we won't be doing that here.) Not all the images have a corresponding ground-truth disparity map.

Part 1.1: infer depth from disparity

The first task is to fill in the code for the `disparity_to_3d()` function. You should assume that the center of the image corresponds to x=0 and y=0. The other thing to be careful about is that the disparity formula

$$Z = \frac{fb}{d}$$

gives us the 3D z-coordinate, but some more work is still required to get the 3D x- and y-coordinates.

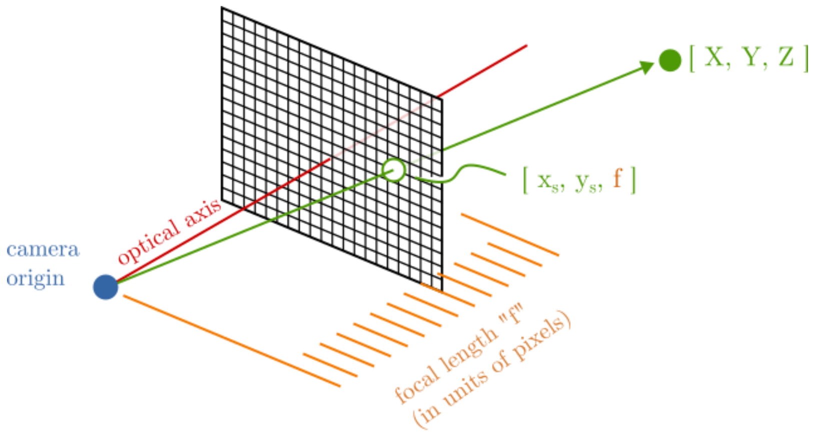

Figure 1: The picture you should have in mind when recovering 3D from 2D image coordinates and disparity

Part 1.2: implement a simple stereo-matching algorithm

Implement your own (simple) stereo-matching algorithm using numpy and "low-level" OpenCV functions. For grading and unit-testing purposes, you're required to implement a simple baseline algorithm using the sum of absolute differences (SAD) metric and simply set disparity to the best match for each pixel regardless of what its neighbors are doing.

The SAD metric at location `(i, j)`, disparity d, and window size w, is defined as

$$SAD(i,j,d) = \sum_{k=-w/2}^{w/2} \sum_{l=-w/2}^{w/2} |I_1(i+k, j+l) - I_2(i+k, j+l-d)|$$

Note that, since the images are rectified, we only need to apply the disparity d in the horizontal direction.

In practice, we set a minimum and maximum disparity range and set the disparity value at `(i,j)` to argmin$_d SAD(i,j,d)$.

Implement this in the `my_sad_disparity_map()` function. It's reasonable and expected that you may need to write a loop over disparity values, but the rest of the function should be vectorized. (Vectorizing hint: compute a `(h, w, max_disparities)` array of SAD values then use `np.argmin(axis=-1)` for the final result). (Vectorizing hint number two: summing values over a $w \times w$ window is just a box filter! You can therefore compute "absolute differences" with numpy, then use opencv's filtering functions to compute the SAD metric quickly.)

(Optional) Part 1.3: improve on your algorithm and participate in the class leaderboard

(Optional) Try to improve your algorithm's performance, to be compared to your classmates on a Gradescope leaderboard. Winners will receive extra credit using the same scoring system as in Assignment 3 (1st = 5pts, 2nd = 4pts, 3rd = 3pts, 4th = 2pts, all who improved on the baseline SAD method = 1pt).

Important: once you have `my_sad_disparity_map()` working, don't modify it. Instead, your leaderboard submission should be given in `my_leaderboard_disparity_map()`.

We recommend finishing the entire assignment before returning to this optional part.

Some ideas to consider for improving your algorithm:

- Output `disparity=0` for pixels that have no match. We give a small penalty for zeros (so you cannot "win" by setting all to zero), but there's some quick improvements you can make by setting disparity to zero for pixels that have particularly bad matches.
- Use a metric besides SAD like SSD or NCC (watch this video for details).
- Adjust the window size and/or use a non-rectangular window.
- Implement a multi-scale approach where you first compute disparities at a low resolution and then refine them at a higher resolution (analogous to the pyramid approach to Lucas-Kanade optical flow)
- Use a sparse matching approach like SIFT-based feature matching for some of the pixels.
- Read up on what algorithms are used inside of OpenCV and try to replicate one of them.
- Implement some form of smoothing so that, when a pixel has a multiple "good" disparity matches, you bias the result towards the disparity value of its neighbors (this involves optimization and is usually done with graph cuts or belief propagation, which we haven't covered yet, so it's a bit of a stretch goal).
- Note that disparities are represented as floating point, so you are allowed to output fractional values, which may help in some cases.
- The code sets a maximum disparity value for each image to 1/8th the image width. Should you adjust this? Should you add a minimum disparity value? Can you find good defaults that work

for all images in this dataset?

Whatever approach you take, be sure to inspect the outputs and think carefully about what sort of errors your algorithm is making and how you might address them.

Some skeleton code is provided in `stereo.py`. It can be run using `python stereo.py path/` where `path/` is the path to a folder containing `view1.png`, `view5.png`, and `disp1.png` files. For debugging and visualization, we've provided some visualization code which will display the inferred 3D scene. But note that this scene might be wrong if your `disparity_to_3d()` function is wrong!

Part 2: Inferring Camera Pose

In this part you'll fill out the missing code in the `pose_estimation.py` file. Here, we'll use some carefully curated* examples from the middlebury multiview dataset to see how we can *solve for the Essential Matrix* after doing feature matching between the images, and how this then allows us to recover the camera extrinsics (the $\begin{bmatrix}\mathbf{R} & \mathbf{t}\end{bmatrix}$ matrix).

* "carefully curated" means that the authors of this dataset were very careful to precisely measure the calibration matrix for their camera, to control the lighting conditions, to remove any lens distortions from the images, and to ensure no background or clutter.

Part 2.1: Visualize what we're doing

Before we get into the nitty-gritty of the implementation, let's visualize what the goal is. Assuming you've extracted the middlebury templering dataset to `data/templeRing/`, run the file

$ python visualize_dataset.py data/templeRing/

You should get an interactive 3D scene like this:

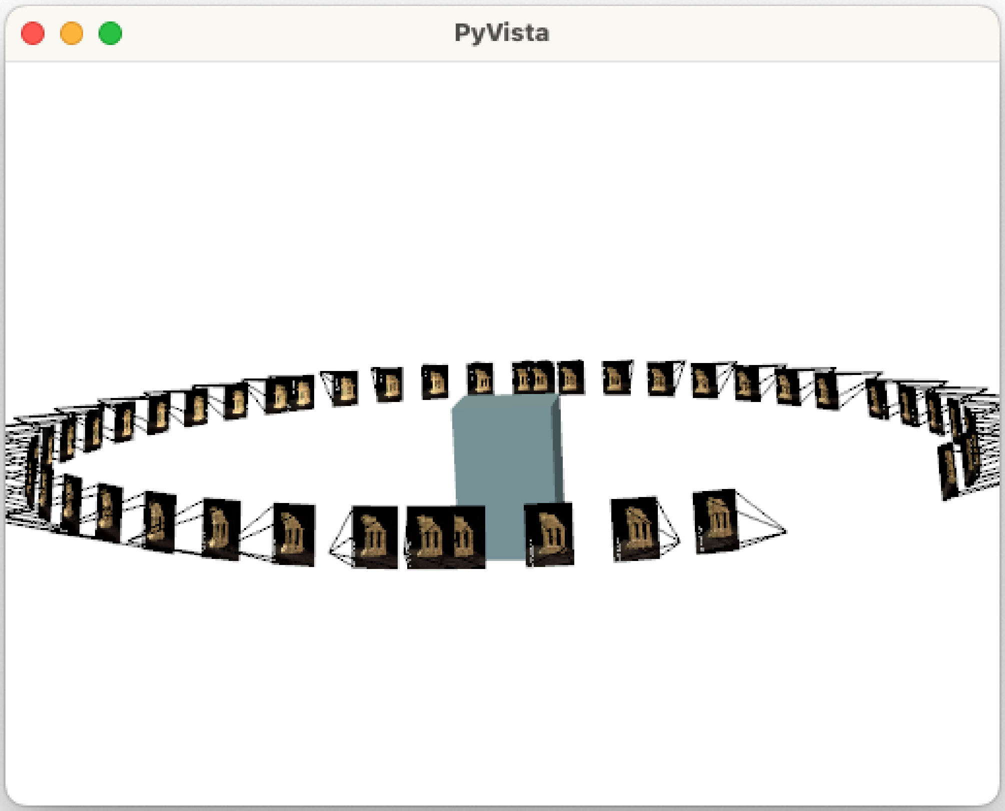

Figure 2: Visualization of the templeRing dataset

This visualization is based on ground-truth camera position information stored in the `templeR_par.txt` file. What we'll do below is pick two nearby images and *recover* the camera position information from the images themselves. Ultimately, the goal is to run

```
$ python pose_estimation.py data/templeRing/templeR0001.png data/templeRing/temp
-true-displacement=0.03
```

(where the `--true-displacement` flag gives the true distance between camera positions and can be worked out from the `templeR_par.txt` file). When this works, you'll get a visualization like this:

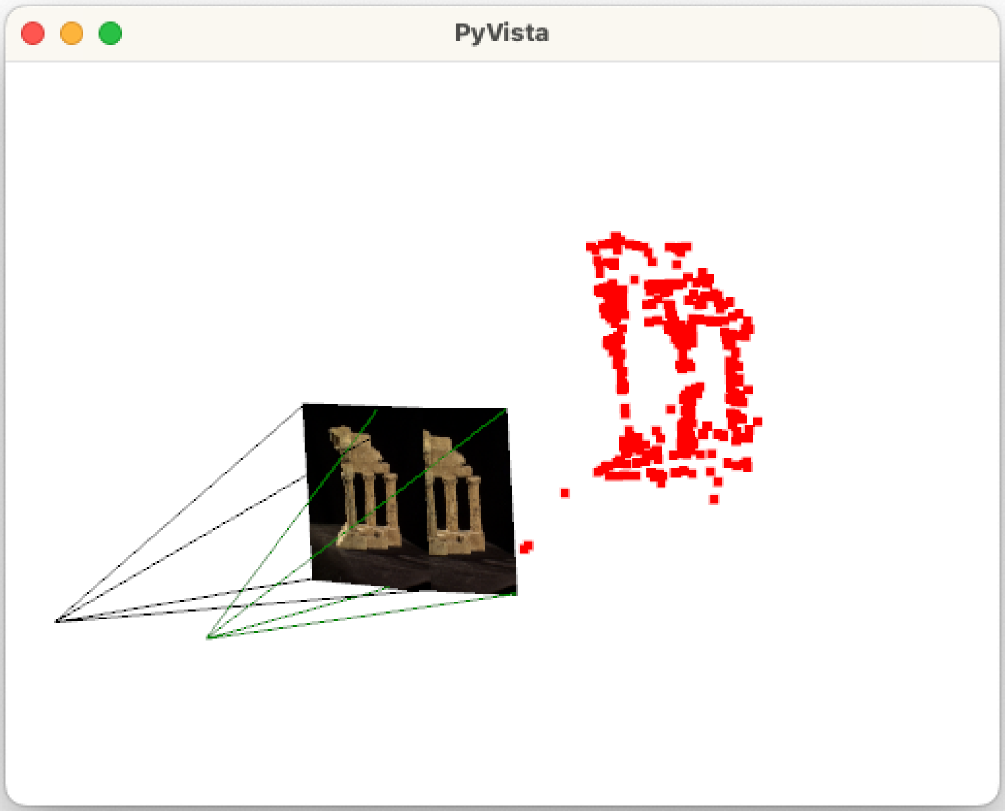

Figure 3: Visualization of the temple with estimated camera positions

In this image, the black lines are the camera for image1, the green lines are the *inferred* position of camera2, and the red dots are the *inferred* 3D points on the temple object itself.

Part 2.2: Implementation

You'll implement missing code in `pose_estimation.py`. The function `match_points` is provided. It provides matched SIFT features across the two images (like you already did in Assignment 7). Just like in A7, we must assume that *most* of these matches are correct, but some are not (outliers). In A7, we hypothesized that the inlier points were related by the equation

$$\tilde{\mathbf{x}}_0 = \mathbf{H}\bar{\mathbf{x}}_1 \ ,$$

i.e. that you can use a homography to get from one to the other. Now that we're dealing with 3D camera movement, we cannot assume a homography, but we *can* assume that the inlier points are related by the equation

$$0 = \hat{\mathbf{x}}_0^T \mathbf{E} \hat{\mathbf{x}}_1$$

where $\mathbf{E}$ is the essential matrix and the "hat" points indicate the 3D unit vectors pointing from the cameras' origins to the pixels. This equation is known as the "Epipolar Constraint".

The function `infer_camera_pose_and_triangulate` is also provided, which gives the high-level structure of the main algorithm. Your job will be to implement the individual steps with calls to OpenCV functions. The steps are:

- `estimate_essential_matrix()` - here, you should call `cv.findEssentialMat()` with appropriate flags so that it uses RANSAC under the hood. Be sure to return only the *subset* of points that are inliers to set up the next step. One tricky note is a difference in terminology between what's in the book/lectures and what's in OpenCV: OpenCV uses the term "camera matrix" for what we've been calling the "calibration matrix".
- `recover_pose_from_essential()` - here, you should call `cv.recoverPose()` to get the rotation and translation matrices from the essential matrix. Note that the `recoverPose()` function has many overloaded versions and can be quite confusing. We expect you to use the version with three input arguments (essential matrix and the two sets of points), and it will return four outputs. These outputs will contain among them the rotation and translation matrices, as well as another "inlier" mask that you should use to further filter down to points that CV has deemed well-fit.

  Note that the translation vector returned by `recoverPose()` will always have unit length. That is, you could `assert np.linalg.norm(t) == 1` and it should pass, regardless of the points you pass in. This is because the scale of the translation vector is not uniquely determined by the essential matrix, and so it is set to unit length by OpenCV. You must ensure that the translation vector you return has the length `scale`, which will allow us to visualize everything at the appropriate scale at the end.
- `triangulate_points()` - this function will now take points from each image, as well as the inferred pose (R and t), and triangulate them back to 3D coordinates. You should call the function `cv.triangulatePoints`. A few "gotchas" are worth noting here:
  - this function expects "projection matrices" as input. These are 3x4 matrices containing camera pose information for each camera. You should set the first camera to have no rotation and no translation (i.e. camera1 is the world coordinate system here), and set camera2 to have the rotation and translation parameters inferred earlier.
  - Mind your transposes. For some strange reason, `triangulatePoints` expects `2xN`

arrays of points and returns a `4xN` array of points, where all other functions above use the `Nx2` convention.

- The `4xn` array of points should be treated as *3D homogenous coordinates* and normalized appropriately. Your function should return a `Nx3` array of recovered 3D points.

Collaboration and Generative AI disclosure and Assignment Reflection

Did you collaborate with anyone? Did you use any Generative AI tools? How did the assignment go? Briefly explain what you did in the `collaboration-disclosure.txt` file. **Completing the collaboration disclosure for each assignment is required.** Completing the reflection part of the file is optional, but it is a good way to give feedback to the instructor and grader about how the course is going.

Submitting

As described above, use `make submission.zip` to generate a zip file that you can upload to Gradescope. You can upload as many times as you like before the deadline. To account for late penalties, contact the instructor or grader directly if you plan on uploading any further revisions after the deadline.