B657 Assignment 2: Warping, Matching, Stitching, Blending

Spring 2020

This is an optional assignment, and completing it can only help (not hurt) your grade.

Due: Sunday April 26 11:59PM

In class recently we've discussed several important ingredients in computer vision: feature point (corner) detection, image descriptors, image matching, and transformations. This assignment will give you practice with these techniques.

You may work on this assignment with a group of up to four people. You may choose your own group members. If you plan to complete the assignment, please let us know your teammate(s) via a private Piazza post so that we can create a repo for your team. You should only submit one copy of the assignment for your team, and through GitHub, as described below. After the assignment, we will collect anonymous information from your teammates about everyone's contributions to the project. In general, however, all people on the team will receive the same grade on the assignment. Please read the instructions below carefully; we cannot accept any submissions that do follow the instructions given here. Most importantly: please start early, and ask questions on Piazza so that others can benefit from the answers.

You may choose to use any general-purpose programming language, with the restriction that you must implement the image processing and computer vision operations yourself, **and** your code must run on the SICE Linux servers (e.g. tank.soic.indiana.edu). For example, you may use Python, but you should implement your own computer vision operations (with some exceptions below). You do not have to re-implement image I/O routines (i.e. you may use Python libraries for reading and writing images). You may also use libraries for routines not related to computer vision (e.g. data structures, sorting algorithms, matrix operations, etc.). It is also acceptable to use multiple programming languages, as long as your code works as required below. All that said, we recommend using either Python with the Pillow library, or C/C++ with the CImg library. No matter what language and library you use, make sure that your program obeys the input and output requirements below (e.g., takes the right command line parameters in the right order, and creates the right output files) since we use testing scripts that automatically test your program. If you have any questions about any of this, please ask the course staff.

Academic integrity. You and your partners may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about C/C++ syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials in the documentation of your source code. However, the code that your team submits must be your own work, which you personally designed and wrote. You may not share written code with any other students except your own partner, nor may you possess code written by another student who is not your partner, either in whole or in part, regardless of format.

Part 0: Getting started

We have made skeleton code available via Canvas.

Part 1: Image matching and clustering

As cameras continue to pervade our lives, it is not an exaggeration to say that the world is drowning in visual data: consumers will take about 1.5 trillion photos this year alone, or roughly the same number that were taken in all of human history up to the year 2000. With so much visual data, we need tools to help people automatically organize their photos. Sometimes we might know ahead of time what people are looking for in photos, and can organize based on semantic content (e.g., place the photo was taken, who is in the photo,

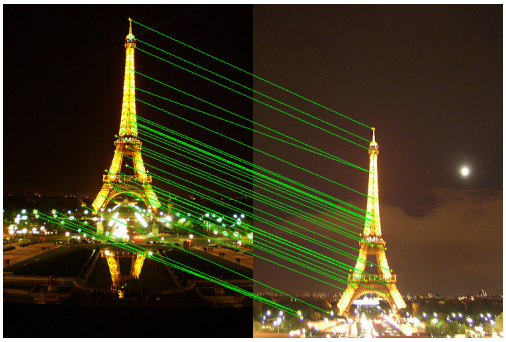


Figure 1: Sample visualization of SIFT matches.

etc). Other times we may simply want to help people organize their photos into coherent groups according to similar visual content, even if the algorithms do not attempt to recognize what that similar content is. A natural idea in this context is to cluster photos into a small number of groups according to visual similarity across photos.

In class, we discussed SIFT, a technique to detect corners and other "interest" or "feature" points. For each interest point, SIFT produces a 128-dimensional vector that is supposed to be invariant to image transformations like scaling, rotation, etc. These features can be useful for discovering similarities in visual content across images. SIFT is very popular but has several problems, including that it is protected by patents. Although we can use SIFT in academia for research and education purposes without a license, there are few high-quality open-source SIFT implementations available. We suggest using one of two approaches:

1. If you want to use Python, we suggest using a similar descriptor, ORB, instead of SIFT. The idea behind ORB is very similar: it takes an image and produces a set of keypoints (x,y coordinates) and descriptors (high-dimensional vectors). In your repository, we've provided some sample code for computing ORB features using the OpenCV library. The code also shows how to compute the distance between ORB features; they are binary features so a bitwise Hamming distance is typically recommended.
2. If you use C++, we have provided an implementation of SIFT that we prepared.

ORB was designed to be much faster than SIFT while offering similar performance.

1. Write a function that takes two images, applies ORB/SIFT on each one, and counts the number of matching ORB/SIFT descriptors across the two images. Remember that a ORB/SIFT "match" is usually defined by looking at the ratio of the Euclidean distances between the closest match and the second-closest match, and comparing to a threshold. (Although not required, you may find it helpful to also create a function that visualizes these matches, as in Figure 1).
2. Use your function above to implement a simple image clustering program, that organizes a set of unordered images into k groups, where k is a parameter given by the user. Your program should take command line arguments like this:

   ```
   ./a2 part1 k img_1.png img_2.png ... img_n.png output_file.txt
   ```

   and then produce an output file with k lines, each of which lists the images assigned to that cluster. For example, for k=2 your output file might be:

```
img_2.png img_4.png
img_1.png img_3.png img_5.png
```

Hints: You can use any clustering technique you'd like, although a natural one might be to use k-means clustering. A key design decision is your choice of distance metric, i.e. how you compare two images. You might start with a count of matching ORB/SIFT descriptors as in #1 above, but feel free to refine this to get better results.

3. We will provide a small test image collection, in your GitHub repo, consisting of a few dozen images from well-known tourist attractions. There are 10 classes in this dataset, with the file names indicating the correct classes. (You can use these file names for evaluating the accuracy of your clustering, but of course you cannot use the names to help with the clustering — that would be cheating!) In your report, show some sample failures (assuming you have some failures) and try to explain what went wrong. Report your performance quantitatively using the following measure. For the N images in the dataset, consider each of the possible $N(N-1)$ pairs of images. Let TP be the number of pairs where your clustering correctly assigned them to the correct cluster, and TN be the number of pairs where your clustering correctly assigned them to different clusters. The Pairwise Clustering Accuracy is then simply $\frac{TP+TN}{N(N-1)}$, which ranges from 0 to 1 where higher accuracies indicate better performance. A small portion of your grade for this assignment will be based on how well your clustering performs on our (separate) test dataset on this task, compared with others in the class.

Part 2: Image transformations

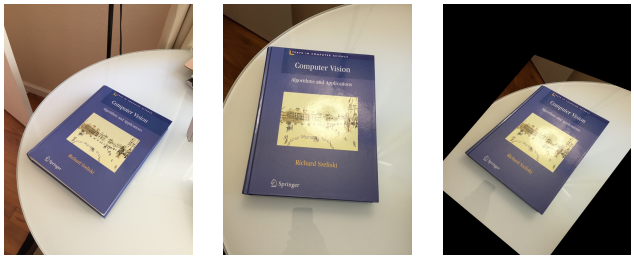
In class we discussed the idea of image transformations, which change the "spatial layout" of an image through operations like rotations, translations, scaling, skewing, etc. We saw that all of these operations can be written using linear transformations with homogeneous coordinates.

1. Write a function that takes an input image and applies a given 3x3 coordinate transformation matrix (using homogeneous coordinates) to produce a corresponding warped image. To help test your code, Figure 1 shows an example of what "lincoln.jpg" (which we've provided in your repository) should look like before and after the following projective transformation:

$$\begin{pmatrix} 0.907 & 0.258 & -182 \\ -0.153 & 1.44 & 58 \\ -0.000306 & 0.000731 & 1 \end{pmatrix}.$$

This matrix assumes a coordinate system in which the first dimension is a column coordinate, the second is a row coordinate, and the third is the extra dimension added by homogeneous coordinates. For best results, use inverse warping with bilinear interpolation.

Figure 2: Sample projective transformation.



(a) (b) (c)

Figure 3: The goal is to warp image (b) to appear as if it were taken from the same perspective as image (a), producing image (c).

2. Now let's say you don't know the transformation matrix, but you do know some pairs of corresponding points across two images and you know the type of transformation that relates the two. Write a program that accepts command line parameters like this:

```
./a2 part2 n img_1.png img_2.png img_output.png img1_x1,img1_y1 img2_x1,img2_x1 ... img1_xn,img1_yn img2_xn,img2_yn
```

where n indicates the type of transformation and $n = 1$ means only a translation, $n = 2$ means a Euclidean (rigid) transformation, $n = 3$ means an affine transformation, and $n = 4$ means a projective transformation, and the remaining parameters indicate the feature correspondences across images (e.g. point (img1_x1, img1_y1) in img_1.png corresponds to (img2_x1, img2_y1) in img_2.png). Note that the parameter n is designed to indicate the number of point correspondences that are needed; e.g. only 1 pair of correspondences is needed for translations, while 4 are needed for projective transformations.

Your program should compute and display the transformation matrix it has found, and then apply that transformation matrix to img_1.png to produce a new image img_output.png. If all has worked correctly, your img_output.png should look very similar to img_2.png. For example, we've included images called "book1.jpg" and "book2.jpg" in your repository. For $n = 4$, four pairs of corresponding points are:

"book1.jpg": {(318, 256), (534, 372), (316, 670), (73, 473)}

"book2.jpg": {(141, 131), (480, 159), (493, 630), (64, 601)}

which you could run through your program like this:

```
./a2 part2 4 book1.jpg book2.jpg book_output.jpg 318,256 141,131 534,372 480,159 316,670 493,630 73,473 64,601
```

and get a result similar to that shown in Figure 3.

Hints: Start with $n = 1$ (and create a simpler test image involving just translation) and get that working first, then move on to $n = 2$, etc. Solving for the transformation matrices will involve solving a linear system of equations; feel free to use the solver built into NumPy.

Part 3: Automatic image matching and transformations

Finally, put the parts above together to automatically find the transformation between two images, to transform one into the coordinate system of the other, and then to produce a "hybrid" stitched image that combines the two together. One application of this is to find create panoramas (Figure 6). Another might be to align two images taken at different times (e.g., from satellites) to be able to quickly spot differences between them.

Your program will be called as follows:

```
./a2 part3 image_1.jpg image_2.jpg output.jpg
```

In particular, your program should (1) extract interest points from each image, (2) figure the relative transformation between the images by implementing RANSAC, (3) transform the images into a common coordinate system, and (4) blend the images together (e.g. by simply averaging them pixel-by-pixel) to produce output.jpg.

This is again an open-ended problem and you can take it in multiple directions, as long as you implement RANSAC with feature points to figure the transformation automatically. In your report, describe your solution, including outline of your algorithm, any important parameter values, any problems or issues you faced, design decisions you made, etc. Show some sample results on images of your choice, including failures and some intuition about why they failed.

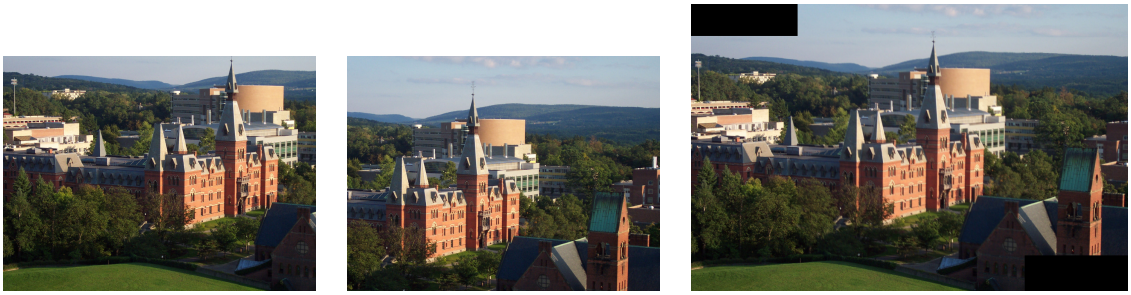


Figure 4: Sample panorama, based on two source images.

What to turn in

Make sure to prepare (1) your source code, and (2) a PDF report that explains how your code works, including any problems you faced, and any assumptions, simplifications, and design decisions you made, and answers to the questions posed above. To submit, simply put the finished version (of the code and the report) on GitHub (remember to `add`, `commit`, `push`) — we'll grade the version that's there at 11:59PM on the due date.