B657 Assignment 2: Image Warping, Matching and Stitching

Spring 2016
Due: Sunday February 28, 11:59PM
Late Deadline: Tuesday March 1, 11:59PM (with 10% grade penalty)

In class recently we've discussed several important ingredients in computer vision: feature point (corner) detection, image descriptors, image matching, and projections and transformations. This assignment will give you practice with these techniques, and show how they can be combined together to both recognize scenes and create panoramas.

We've once again assigned you to a group of other students. You should only submit one copy of the assignment for your team, 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 the OnCourse Forum so that others can benefit from the answers.

We once again recommend using C/C++ for this assignment, and we have prepared skeleton code that will help get you started. This time, we recommend using a different library called CImg, which is much more powerful than the image library we used last time. This library is open source and very well documented online at `http://cimg.eu/`. CImg is already included in your GitHub repo, and our skeleton code shows you an example of how to use it. For this project, you *may* use the image processing methods of the CImg class, instead of having to write everything from scratch. You may also use additional libraries for routines not related to image processing (e.g. data structures, sorting algorithms, etc.), or to what we expect for you to implement in the assignment below. Please ask if you have questions about this policy.

Academic integrity. You and your partner 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 you and your partner submit 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

Clone the GitHub repository with the test images:

`git clone https://github.iu.edu/cs-b657/`*your-repo-name*`-a2`

where *your-repo-name* is the one you found on the GitHub website.

Part 1: Image matching

In class, we discussed SIFT, a technique to detect corners and other "interest" or "feature" points. Our skeleton code includes a function that runs SIFT on an image, and returns a vector of all the detected SIFT point coordinates, and the 128-dimensional SIFT descriptor for each one.

1. Write a function that takes two images, applies SIFT on each one, finds matching SIFT descriptors across the two images, and creates an image like the one shown in Figure 1 to visualize the matches.

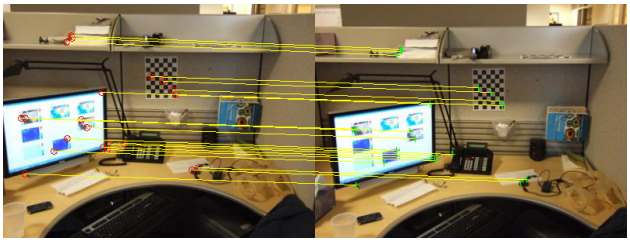

Figure 1: Sample visualization of SIFT matches. (Courtesy MathWorks.)

Remember that a 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.

2. Use your functions above to implement a simple image retrieval program, that searches an image library to find the best matches for any given query image. Your program should take command line arguments like this:

   ```
   ./a2 part1 query.png img_1.png img_2.png ... img_n.png
   ```

   and then compare query.png to each of the other images. The program should then output the list of remaining images, sorted in decreasing order of number of matched features.

3. We've provided a small test image collection for download on OnCourse, consisting of 10 images from each of 10 well-known tourist attractions, for a total of 100 images. To measure the performance of your retrieval algorithm on this task, conduct the following experiment. For each of the 10 attractions, choose 1 image of that attraction at random to use as the query image, and then pass all 100 images as the others. Look at the top 10 ranked images returned by your program, and calculate the percentage of images that are from the correct attraction. This is called the *precision* of your system. Repeat this process for each of the remaining attractions, and present the results in your report. Which attractions seem to be the easiest to recognize, and which are most difficult?

4. The SIFT matching step for the above parts can be quite slow, because finding nearest neighbors in a large set of high-dimensional points is very computationally demanding. There are various heuristics that can speed up the matching, but none of them is perfect; instead, each offers different trade-offs between accuracy and running time, and each works better in different circumstances.

   Here's one for you to try. For a vector $\mathbf{v}$, which in this case could be a 128D sift vector, we'll define a quantized projection function $f(\mathbf{v})$,

   $$f_i(\mathbf{v}) = \left\lfloor \frac{\mathbf{x}^i \cdot \mathbf{v}}{w} \right\rfloor$$

   where w is a constant that controls the degree of quantization, $i \in [0, k]$ is an integer index, and each $\mathbf{x}^i$ is a randomly-generated vector with the same dimensionality as $\mathbf{v}$ (e.g. each element of $\mathbf{x}^i$ is sampled from a uniform distribution in the range $[0, 1]$, or from a Gaussian distribution with $\mu = 0$, $\sigma = 1$). If $k << 128$, we can think of this function as "summarizing" a SIFT vector in just k numbers, by taking projections of $\mathbf{v}$ with multiple randomly-generated vectors. So to find the nearest neighbor of $\mathbf{v}$ in one image amongst a set of SIFT vectors in a second image, we can calculate $f(\cdot)$ on each of the SIFT vectors in the second image, and identify the summary vectors that are identical to $f(\mathbf{v})$, which gives a (hopefully small) set of candidate nearest neighbors. Then we calculate the Euclidean distance in 128D space between $\mathbf{v}$ and the set of candidates to find the closest one (and/or second closest one).

   The reason this works is that two similar SIFT vectors will usually have very similar "summary" vectors, and very different SIFT vectors will usually have very different summary vectors. But this is not guaranteed because some of the random projections might cause nearby SIFT vectors to end up

Figure 2: Sample projective transformation.

with different summaries and vice-versa. There are two ways to address this. One is by adjusting the parameter w, which controls how similar two projections have to be in order to be considered identical. The other is to run the above process multiple times with multiple sets of random vectors, and then choose the closest SIFT vector that is found across all of those trials.

Implement this algorithm for the image matching. Does the algorithm speed up image retrieval, and if so, by how much? Does the approximation seem to affect the quality of results? Give quantitative answers to these questions if possible. What is the best trade-off you can achieve between running time and accuracy, by adjusting w, k, and the number of trials?

Part 2: Image warping and homographies

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 a linear transformation 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 2 shows an example of what "lincoln.jpg" (which we've provided on OnCourse) 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. It's probably easiest to implement this using inverse warping (see the Szeliski book for details), and you'll get nicer results if you use some interpolation, like bilinear or bicubic (again, see the Szeliski book), although this is not required.

2. Now, write a function that automatically estimates a homography (projective transformation) between two images. To do this, you'll first need code that finds feature correspondences between the images; fortunately you already did this in Part 1! Then you'll need to write code that uses these noisy correspondences to estimate the homography itself. Implement this using RANSAC, as we discussed in class.

   Hint: CImg has a linear system solver!

3. Finally, combine your homography estimation and your warping code to create an image sequence warping application. Given a sequence of two or more images, the program should compute a homography between each image and the first image, and for each one produce a warped image that

appears to have been taken in the first camera's coordinate system. We will provide some test data via OnCourse, and please show some sample results on this data in your report. Your program should take command line arguments like this:

```
./a2 part2 img_1.png img_2.png ... img_n.png
```

and produce warped images called `img_2-warped.png`, `img_3-warped.png`, etc.

What to turn in

Make sure to prepare (1) your source code, and (2) a report that explains how your code works, including any problems you faced, and any assumptions, simplifications, and design decisions you made, and answers 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.