B657 Assignment 2: Warping, Matching, Stitching, Blending

Spring 2018
Due: Tuesday February 27, 11:59PM
Late Deadline: Thursday 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 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 Piazza 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 to get our skeleton code:

`git clone https://github.iu.edu/cs-b657-sp2018/`*your-repo-name*`-a2`

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

Part 1: Custom Billboards (Image Warping and Homogeneous Matrix)

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 1 shows an example of what "lincoln.png" (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. 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.

Figure 1: Sample projective transformation.

2. Now let's say you don't know the projective transformation, but you do know four pairs of corresponding points across two images. Create a function that solves for the homography (projective transformation) given these correspondences. To do this, you'll need to set up and solve a linear system of equations. Apply it to warp "book2.jpg" (in your repository) to be in the same coordinate system as "book1.jpg," as shown in Figure 2. Four sets of corresponding points are(Coordinates are listed in x,y format):

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

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

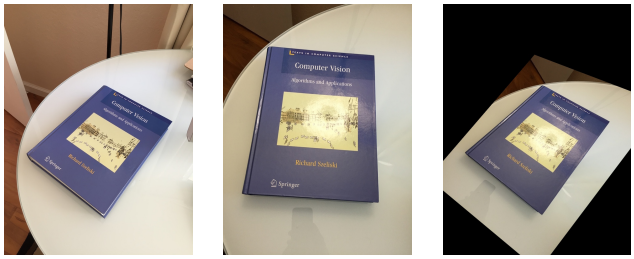
   (You can hard-code these values, but it's probably best to solve for the transformation in your code (as opposed to pre-computing the transformation in Matlab or another tool) since you'll need this code for later parts. CImg has a linear system solver!)

3. Finally, combine your homography estimation and your warping code to superimpose a given (rectangular) input image into a region (arbitrary quadrilateral) of another image. In particular, for each input image, your program should produce three output images, one for each of the the images billboard1.jpg, billboard2.png, and billboard3.jpg in your repo, that makes it appear as if the input image appeared on the while area of the billboard. Figure 3 shows an example. (You can hard-code your program to work only on these three particular billboard images; we won't test on others.)

Your program for Part 1 take command line arguments like this:

```
./a2 part1 poster_input.png
```

and produce several outputs, corresponding to the three items above: (1) warped image `lincoln_warped.png`, (2) `book_result.png` and print the homography matrix to the screen, and (3) `synthetic_billboard1.png`, `synthetic_billboard2.png`, and `synthetic_billboard3.png`.



(a) (b) (c)

Figure 2: The goal is to warp image (b) to appear as if it were taken from the same perspective as image (a), producing image (c).

(a) (b) (c)

Figure 3: Input image (a) is projected into the billboard of image (b), to produce synthetic image (c). (Image Source: bartush, nerdist website)

Part 2: Blending

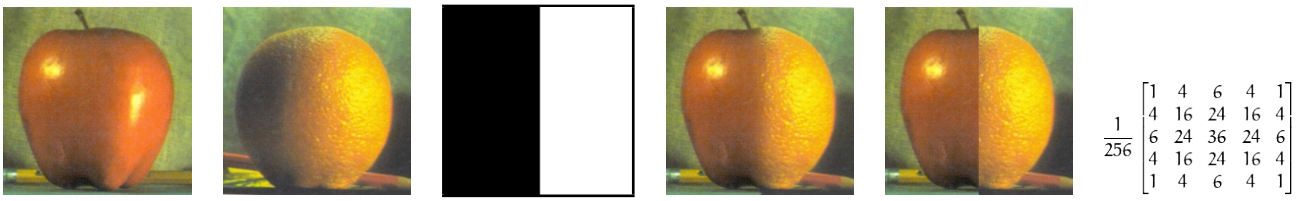
Blending is the process of combining two images together to make a smooth transition between them. For example, in Figure 4, the apple in (a) and orange in (b) have been combined to create the blended image in (d). The mask in (c) indicates where each pixel in the output image should come from, with black pixels indicating image (a) and white pixels indicating image (b). The trick is to do this in such a way that there is not a sharp discontinuity between the two images, by "blending" pixels values near the boundary in the mask of image (c). Figure 4(e) is direct blending of images side by side to help you identify the difference. Here is one way to create a smoothly blended image, given two images and a mask:

1. Compute the *Gaussian pyramid* and *Laplacian pyramid* for each image. Level 0 of the Gaussian Pyramid simply consists of the input image. Level i of the Gaussian Pyramid is the image at level $i-1$ smoothed with a Gaussian filter and subsampled at one-half the resolution (producing an image with one-quarter the number of pixels as the image at level $i-1$). Meanwhile, level i of the Laplacian pyramid is simply the image at level i of the Gaussian pyramid smoothed with a Laplacian-of-Gaussian. As we saw in class, you can calculate this by simply taking an image and subtracting out a version of the image that has been smoothed by a Gaussian. Compute several levels of each pyramid (e.g. 5). You can use the Gaussian function given in Figure 4(f) for convolution.

2. Build Gaussian pyramid for the mask image.

3. Now create a "blended" Laplacian pyramid. For level i of the pyramid, the blended Laplacian image should be a sum of the images in the corresponding level of the two input images, weighted by the mask image at level i of the Gaussian pyramid,

$$LB_i(x,y) = M_i(x,y) * L1_i(x,y) + (1 - M_i(x,y)) * L2_i(x,y),$$



(a) (b) (c) (d) (e)

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

(f)

Figure 4: Image Blending example. (Image Source: opencv)

where LB_i is level i of the blended Laplacian pyramid, $L1$ and $L2$ are the Laplacian pyramids of the two input images, and M_i is the image at level i of the Gaussian pyramid of the mask.

4. Finally you will form the blended image from the laplacian pyramids by joining them at each level. Reconstruction starts by up-sizing the LB_i image to twice the original in each dimension, with the new even rows and columns filled with zeros. Then perform a convolution with the same kernel shown above Figure 4(f) (multiplied by 4 to approximate the values of the missing pixels). This LB_i is then added to LB_{i-1} to give you an intermediate result. Keep on recursively doing the same step with the result until you get back the image with original shape, which eventually is your blended image.

Your program will be called as follows

```
./a2 part2 image_1.png image_2.png mask.png
```

and output the Laplacian and Gaussian pyramids at each level and the final blended image. Include them in your report as well.

Experiment with image blending by creating your own mask and doing Laplacian blending on different images to produce interesting results. Provide at least two examples of image blending in your report, including the apple and orange example.

Part 3: Image Matching and RANSAC

Recall that SIFT is a technique to detect "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 5 to visualize the matches. 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. Show an example figure in your report.

2. The image matching approach above is susceptible to false matches and is slow, because of the sheer number of "matching" descriptors. To solve the matching problem, let's introduce a check that the geometric arrangement of SIFT correspondences is consistent. Write a function that uses your previous homography (projective transformation) code between two images. You'll need to write additional code that uses noisy SIFT correspondences found above to estimate the homography using RANSAC. Visualize the inliers, similar to Figure 5, and show in your report. Do you notice cleaner correspondences?

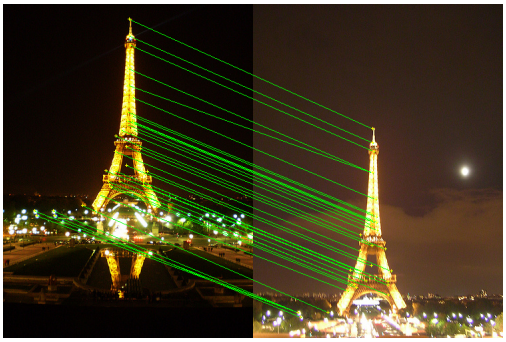


Figure 5: Sample image visualizing SIFT matches.

For this part, your program should take command line arguments like this:

```
./a2 part3 image_src.png image_dst.png
```

and then output two images: (1) `sift_matches.png`, and (2) `ransac_matches.png`, showing the two visualizations mentioned above.

Part 4: Creating Multi-image Panoramas

Our next goal is to create an application that stitches together multiple (at least three) images to create panoramas (Figure 6). Your program will be called as follows:

```
./a2 part4 image_1.png image_2.png image_3.png
```

and should output a file called `panorama.png`. In particular, your program should (1) extract SIFT features from each image, (2) figure relative homographies between the images, (3) transform the images into a common coordinate system, and (4) blend overlapping regions of the images together (e.g. by using the Laplacian technique above, with a suitably estimated mask). Your resulting panorama should be in color, although you may do all internal processing in grayscale.

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. We have provided two sample sets of images that you can use to create panoramas. However, these are just suggestions and it's more fun if you try your own images. Discuss any issues or artifacts that may be evident in your output. If your program did not succeed on some of the sequences, discuss why.

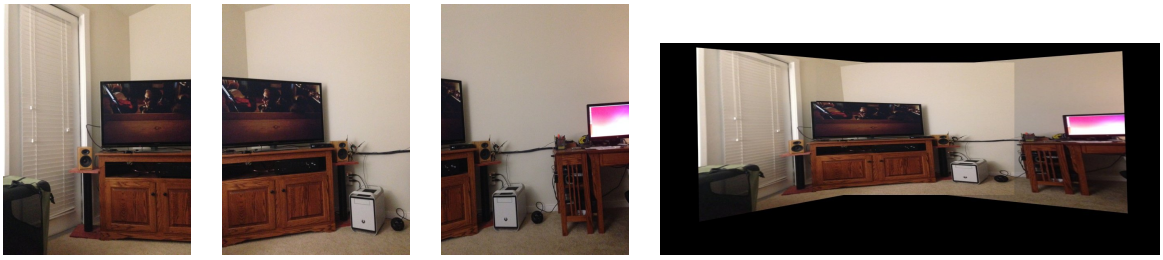


Figure 6: Sample panorama, based on three images. (Image Source: kushalvyas image-stiching application)

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 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.