Project 3: Image Alignment

Dr. Randy C. Hoover

October 31, 2022

Logistics

- Download the datasets from D2L that contain images of overlapping information from several locations.
- D2L: Submit your writeup (LaTeXfor graduate students) (single .pdf) answering questions, illustrating methods, and examples of image alignment (at least four from the data set provided and one from your own) and warping an image into a "framed region" in the second image, along with the processes you used to generate them, and all code to the dropbox in D2L.
- Due: Monday Nov. 14, 2022

Please read each step very carefully before proceeding.

You are allowed to use a limited set of built-in Python functions for your project (such as a good feature detector). If you have questions about the use of a function you find, please **ask first** so I can evaluate said function's capabilities.

As for simply finding a solution to these problems on-line and copying someone else's source code, this is **STRICTLY FORBIDDEN**.

Project Goals: Develop your own (*Python*) functions to stitch two or more images of similar views together. The program will have two parts, points chosen by the user and those automatically generated and matched by using the results obtained in project 2 (or some other suitable feature detector). An example of this process is illustrated in Figure 1.

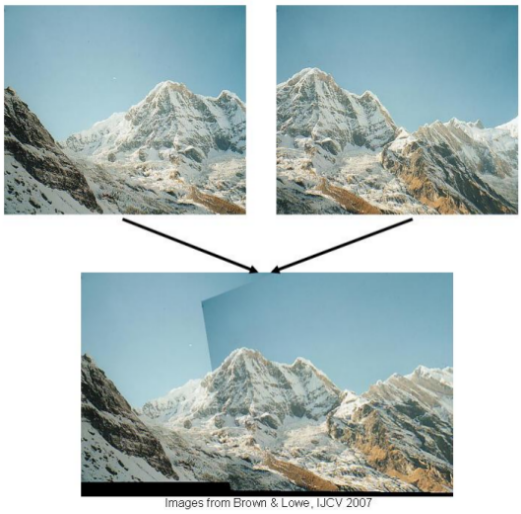

Figure 1: Example image mosaic showing two images being stitched together into a single image.

Part 1: User selected points and correspondences.

- **Getting correspondences:** write code to get manually identified corresponding points from two views. Look at *Matplotlib's* `ginput` function for an easy way to collect mouse click positions. The results will be sensitive to the accuracy of the corresponding points; when providing clicks, choose distinctive points in the image that appear in both views.
- **Computing the homography parameters:** Write a function that takes a set of corresponding image points and computed the associated 3×3 homography matrix using the discrete linear transform (DLT). The matrix transforms any point p_i in one view to its corresponding homogeneous coordinates in the second view, $\grave{p}_i$, such that $\lambda p_i = H\grave{p}_i$. Note that both p_i and $\grave{p}_i$ are in homogeneous coordinates (3-vectors). You function to compute the homography should take a list of $n \geq 4$ pairs of corresponding points from the two views, where each point is specified by its 2-d image coordinates.

  Hint: It would be a good idea to verify that the homography matrix your function computes is correct by mapping the clicked image points from one view to the other, and displaying them on top of each respective image. Be sure to handle homogenous and non-homogenous coordinates correctly.
- **Warping between image planes:** write a function that can take the recovered homography matrix and an image, and return a new image that is the warp of the input image

using H. Since the transformed coordinates will typically be sub-pixel values, you will need to sample the pixel values from nearby pixels. For color images, warp each RGB channel separately and then stack together to form the output.

To avoid holes in the output, use an inverse warp. Warp the points from the source image into the reference frame of the destination, and compute the bounding box in that new reference frame. Then sample all points in that destination bounding box from the proper coordinates in the source image. Note that transforming all the points will generate an image of a different shape/dimensions than the original input.

Useful Numpy functions: `round`, `interp2d`, `meshgrid`, `isnan`. You should look at the help documentation to understand how these function work and when you would use them.

- **Create and display the output mosaic:** once we have the source image warped into the destination image's frame of reference, we can create a merged image showing the mosaic. Create a new image large enough to hold both (registered) views; overlay one view onto the other, simply leaving it black wherever no data is available. Don't worry about artifacts that result at the boundaries.

Part 2: Autostitching.

Now that you have all the functionality in place, you can automate the process by using the code you wrote in Project 2 (**or I highly recommend you use SIFT**).

- **Automatically compute, features, descriptors, and correspondences:** Using your project 1 source code (or SIFT) and function calls, automate the image stitching process by computing a set of matching features (points) in both image frames to compute your homography.

  Note: For those not interested in installing OpenCV, you might investigate Python system calls and just use the SIFT code from Lowe's website.

- **RANSAC:** Because your feature matching is NOT perfect in project 1 (nor would it ever be in every situation), you may run into a situation where the homography being computed is *off* since it was computed with "bad" matching data. This can be remedied by using a technique called Random Sample and Consensus (RANSAC). We will discuss this algorithm in detail in class but the basic idea is: 1) randomly choose 4 matching correspondence points, 2) compute the homography matrix H using these points, 3) using thie H, transform all points in the correspondence list, and 4) count the number of inliers. Keep the H (and the set of inliers) with the largest inlier count. Using the inliers, you can compute a "best" estimate of H.

- **Warp and display:** Now that you have a good estimate of the homography computed, repeat Part 1 for warping and displaying the mosaic.

Deliverables:
All code, images, process, etc. embedded in a single document and submitted to D2L. This file should be sufficient for me to completely understand your approach, evaluate code you wrote, and visualize your solutions applied to many different image sets. As for the output, I will provide a set of "benchmark" images for you to test your algorithm on.

At a minimum you need to:

1. Apply your solution to the set of benchmark images and display the output mosaic.
2. Show at a minimum one additional example from your personal collection of images. This can be either a place where a wide view angle is required to see everything well or a mosaic of the same room with a person in the room in two different locations (of course you could do both if you want).

3. Warp one image into a "frame" region in the second image. To do this, let the points from the one view be the corners of the image you want to insert in the frame, and the let the corresponding points in the second view be the clicked points of the frame (rectangle) into which the first image should be warped. Use this idea to replace one surface in an image with an image of something else. For example – overwrite a billboard with a picture of your dog, or project a drawing from one image onto the street in another image, or replace a portrait on the wall with someone else's face, or paste a personal image onto a movie screen, ...

Rubric

- +50 pts: Working implementation of image alignment demonstrated on the benchmark images provided.
- +25 pts: Working implementation illustrating your algorithms warping one image into a frame of another image.
- +20 pts: Working demonstration of RANSAC illustrating the image alignment process is automated as opposed to hand selecting matching features..
- +05 pts: Writeup.
- -05*n pts: Where n is the number of times that you do not follow the instructions.

Extra Credit: (10 pts. / item - max 20 pts.)
Here are some possible extra credit extensions (feel free to come up with your own variants to make your program more robust and efficient):

(a) Do a mosaic of the entire Tech. campus for a full 360° view (need to bundle adjust to align the first and last image).
(b) Implement a cylindrical projection mosaic.
(c) Implement a spherical projection mosaic.
(d) Rectify an image with some known planar surface (say, a square floor tile, or the rectangular face of a building facade) and show the virtual fronto-parallel view. In this case there is only one input image. To solve for H, you define the correspondences by clicking on the four corners of the planar surface in the input image, and then associating them with hand-specified coordinates for the output image. For example, a square tile's corners from the non-frontal view could get mapped to $[0\ 0;\ 0\ N;\ N\ 0;\ N\ N]$ in the output.
(e) Make a short video in the style of the HP commercials. Building on #4 above, let the frame in the output video move to different positions over time, and warp the framed image into the correct position for every video frame in the sequence.

(f) Create a GUI to interactively create the mosaic.

Some suggestions:
This is a big project. Therefore, I suggest that you divide and conquer. You should write modular code so you can reuse it for future projects (i.e., functionalize everything if possible). I would also recommend you write some helper functions as needed.

Tips:

- You can use the `axis` command to adjust the size of your viewing window.
- You will need to compute the inverse of the homography to transform points backwards.
- Be aware that Matplotlib image (matrix) indices are specified in (row,col) order, i.e., (y,x) whereas the `plot` and `ginput` command use (col,row) order, i.e., (x,y) (you can see how these values are returned from ginput).
- You will need to click corresponding points in the same order for matching.
- Be careful when using casting (double vs. uint8). If using `interp2d` be sure to use doubles.
- When collecting your own images, be sure to either maintain the same center of projection (hold the camera at one location, but rotate between views), or else take shots of a scene with a large planar component. In either case, use a static scene. Textured images that have distinctive points you can click on are good. Also ensure that there is an adequate overlap between the two views.