Homework #4: Making Panoramas

Assigned: Thursday, October 13
Due: Thursday, October 27

Introduction

The goal of this assignment is to write a simple photo panorama stitcher. You will take four or more photographs and create a panoramic image by computing homographies, warping, resampling, and blending the photos into a seamless output image. See the lecture slides for more description of these steps. A simple panorama stitcher typically consists of the following steps, some of which have been implemented for you. Instructions that make your life easier are indicated in **bold green**. Instructions about code that you have to implement are indicated in **bold red**.

Prerequisite: Install VLFeat

In this assignment, we use the VLFeat open source library to detect feature points and find their correspondences in overlapping pairs of images.

i. Download the source code from the VLFeat homepage[1].
ii. Unzip the file to a local folder.
iii. Open the `README.md` file within the folder and follow the instructions to install VLFeat in your local machine.

---

1. Capture Images

You are to create *two* panorama images, one using a set of four provided images in `test.zip` For the second panorama use your own set of *at least four images*. You can assume the input images are all in a single folder and are named `1.jpg`, `2.jpg` etc. where the images are taken from left to right in increasing order (i.e., `1.jpg` overlaps with `2.jpg` and that `1.jpg` is to the left of `2.jpg` in the original scene). Be sure that consecutive images that you use for the second panorama overlap at least 30%.

Place four or more of your own images in the `input_images` folder provided.

Implementation tip: Test your work for correctness on a set of low resolution images before moving on to full resolution images. Images can be downsized in MATLAB with the `imresize` function and saved with `imwrite`.

2. Compute Feature Points

In this step you will detect feature points in each image. The output of this step is cell arrays `keypoints` and `descriptors`, such that `keypoints{i}` is a matrix of keypoints in image *i*, and `descriptors{i}` is a matrix of SIFT descriptors that describe the keypoints in image *i*. Information on the SIFT detector and descriptor is given in the Szeliski book in Sections 4.1.1 and 4.1.2. **This step has been implemented for you.**

[1] http://www.vlfeat.org/

3. Match Feature Points and Compute Homographies

In this step, you will find the spatial relationship between *each pair* of overlapping images. This spatial relationship is represented by a transformation known as a *homography*, **H**, where **H** is a 3 x 3 matrix. To apply homography *H* to a point *p*, simply compute *p'* = **H***p*, where *p* and *p'* are (3-dimensional) homogeneous coordinates. *p'* is then the transformed point. In this step however, we want to compute the homography given a set of (*p'*, *p*) pairs of corresponding feature points.

In the previous step, we computed a descriptor for every keypoint in every image. Each SIFT descriptor is a 128-dimesional vector, while each SIFT keypoint is an *x*-*y*-orientation-scale 4-tuple. Computing the homography between an image *i* and an image *i+1* can then be done in the following steps:

i. Find the correspondences between keypoints, using only descriptor information.
ii. Remove the correspondence outliers found in the previous step, using RANSAC and *x*-*y* positional information only.
iii. Calculate the homography, using the inliers found in the previous step.

Part (i) has been implemented for you. You must implement parts (ii) and (iii) by completing the `calcHWithRANSAC` function. The main script will call this function to define a cell array `H_list`. $H_{list} = \{H_{21}, H_{32}, H_{43}, \ldots, H_{(m)(m-1)}\}$ where H_{ij} is the homography that maps points in image *i* into points in image *j*.

Compute Homographies without RANSAC

Given point-to-point correspondences, the following function called `calcH` should compute **H**:

```
H = calcH(im1_ftr_pts, im2_ftr_pts)
```

where `im1_ftr_pts` and `im2_ftr_pts` are *n* x 2 matrices storing the (*x*, *y*) locations of *n* feature point correspondences between the two images. Implicitly, `im1_ftr_pts(i, :)` should correspond to `im2_ftr_pts(i, :)` for all $i = 1, 2, \ldots, n$. `calcH` returns **H**, the recovered 3 x 3 homography matrix.

Once **H** is computed, multiplying **H** times the homogeneous coordinates of a point in image 2 will return the homogeneous coordinates of that point projected into image 1. In order to compute matrix **H**, you need to set up a linear system of *n* equations (i.e., a matrix equation of the form **Ah** = **b** where **h** is a column vector holding the unknown values of **H**). If *n* = 4, the system can be solved using a standard technique. However, with only four points, homography recovery is very unstable and prone to noise. Therefore more than 4 correspondences should be used to produce an overdetermined system, which can be solved using least-squares. In MATLAB, this can be performed using the MATLAB "\" operator (see `mldivide` for details).

Compute Homographies with RANSAC

Unfortunately, not all the corresponding points found by SIFT will be correct matches. To eliminate "outliers," i.e., incorrect matches, **you need to modify `calcHWithRANSAC` so that it includes the RANSAC algorithm as part of the procedure for computing the best homography**. To do this, add a loop that runs `numIter` times, computing a homography at each iteration, and keeping the best homography found. In each iteration, select four pairs of points *randomly* from those computed by SIFT, compute **H** from these four pairs of points, and then count how many of the *other* pairs of points agree, i.e., a point projects very near its corresponding point in the pair. Use a constant threshold value of `maxDist` pixels to decide whether two points are close enough to be considered "inliers." For example, if $p_i' = (x_i', y_i')$ is a point in image 1 corresponding to point $p_i = (x_i, y_i)$ in image 2, first convert p_i to homogeneous coordinates, say $(x_i, y_i, 1)$; then multiply **H** by the 3 x 1 vector corresponding to the homogeneous coordinates of p_i to obtain q_i = **H**p_i ; then compute the Euclidean distance

between point p_i' and q_i after first converting q_i to its real-valued Cartesian coordinates; finally, check if the distance is less than `maxDist`, and, if it is, increment the count of the number of other pairs of points that agree with this **H**.

Repeat the above process of randomly selecting four pairs of points `numIter` times, computing their associated **H**, and counting the number of inliers for each. Keep the homography $\mathbf{H}^*$ with the most agreement, i.e., greatest number of inliers. Finally, compute the final homography using *all* the points that agree with $\mathbf{H}^*$.

Use `numIter = 100` and `maxDist = 3` in your implementation of `calcHWithRANSAC`.

*Recommendation: Implement RANSAC only after you have a working version of your code that does **not** use RANSAC.*
*Implementation tip: The RANSAC loop can be implemented in nine lines of MATLAB, including the 'for' and 'end' lines. You may find the following snippets helpful, though it's **not** necessary to use them to receive full credit for this homework.*

- *A = H * B; % A and B are* 3 x *n matrices and H is a* 3 x 3 *matrix*
- *dist = sqrt(sum((A - B).^2)); % A and B are n* x 3 *matrices and dist is a* 1 x *n matrix*
- *inds = randperm(n*, 4*); % inds is a vector of* 4 *random unique integers in* [1, *n*]

4. Warp Images

The next step is to warp all the input images into the output panorama image. For large field of view panoramas, it is common to use a cylindrical surface for this purpose. When only a few images are used covering a relatively small field of view, however, we can more simply warp all the images onto a plane defined by one of the input images, which we'll call the **reference image**. For simplicity, ***use image 1 as your reference image***.

This step can be broken down into the following parts:

i. Compute the homography of every image with respect to the reference image.
ii. Compute the size of the output panorama image, using forward warping.
iii. Warp every input image on to the panorama, using inverse warping.

You will implement part (i). That is, modify the `main` script to compute the cell array `H_map` of length *m*, such that $H_{map} = \{H_{11}, H_{21}, H_{31}, \ldots, H_{m1}\}$ for *m* input images. H_{11} is the identity matrix. **Parts (ii) and (iii) have been implemented for you.**

We describe each of these parts in detail below.

You must first compute a new set of homography matrices so as to warp each image *directly to the reference image*, image 1. As stated above, the homography H_{ij} is such that $p_j = H_{ij}p_i$ where p_i and p_j are points in images *i* and *j* respectively. Now we want to find H_{i1} for all images *i*, expressed in terms of the homographies that we computed in the previous step. Then we have:

$$H_{i1}p_i = p_1 = H_{21}p_2 = H_{21}(H_{32}p_3) = H_{21}(H_{32}(H_{43}p_4)) = H_{21}H_{32}\ldots H_{(i)(i-1)}p_i$$

Therefore, the homography H_{i1} that warps points in image *i* into points in image 1 is given by $H_{i1} = H_{21}H_{32}\ldots H_{(i)(i-1)}$. For example, $H_{31} = H_{21}H_{32}$ and $H_{41} = H_{21}H_{32}H_{43}$.

Next, before warping each of the images, the size of the output panorama image must be computed and initialized from the range of warped image coordinates for each input image.

This is done for you in the provided code by mapping the coordinates of the four corners (i.e., top-left, top-right, etc.) from *each* source image using *forward warping* to determine its coordinates in the output image. It computes the `min_row`, `min_col`, `max_row`, and `max_col` coordinates to determine the size of the output image, defined by `panorama_height` and `panorama_width`. Finally, `row_offset` and `col_offset` values are computed that specify the offset of the origin of the *reference* image relative to the upper-left corner coordinates of the output panorama image. The output image array, called `panorama_image`, is created and initialized to all black pixels (i.e., value 0).

Determining the size of the panorama used forward warping. However, if you use forward warping to map every pixel from each source image, there will be holes (i.e., some pixels in the output image will not be assigned an RGB value from any source image, and remain black) in the final output image. Therefore, we need to use ***inverse warping*** to map each pixel in the output image into the planes defined by the source images. **The provided code uses bilinear interpolation to compute the color values in the warped image** by calling the MATLAB function `interp2` (with argument '`linear`' to do bilinear interpolation). The provided code uses the MATLAB `meshgrid` function to do this efficiently. Type '`help interp2`' and '`help meshgrid`' to view the help documents for more details on these two functions.

Two issues must be dealt with when performing inverse warping: First, some pixels in the output image will *not* map to a pixel in a given source image because the output pixel's coordinates map outside the domain of the source image. In this case a value of zero will be returned for the given output image pixel by setting an extrapolation value of zero in the call to `interp2`. Second, some pixels in the output image will have *more than one* input image overlapping it; in this case, the provided code simply takes the sum of these source image's pixel values to the output image. While this will produce an output image for initial debugging, you must replace this with a better method of blending, as described in the next step below.

5. Blend Images

Now that the images are transformed to the same coordinate frame, the final step is to blend overlapping pixel color values in such a way as to hide seams. One simple way to do this, called ***feathering***, is to use weighted averaging of the color values to blend overlapping pixels (see Section 9.3.2 in the Szeliski book). To do this, use an `alpha channel' where the value of alpha for each input image is 1 at its center pixel and decreases linearly to 0 or a very small positive value at all the border pixels. You can use the MATLAB function `bwdist` with argument '`euclidean`' and applied to a binary image that has 1s beyond the edges of the warped image and 0s within the warped image, then normalize the distances to be in the interval (0, 1]. Compute an alpha channel for each input image. See below for an illustration of the above steps.

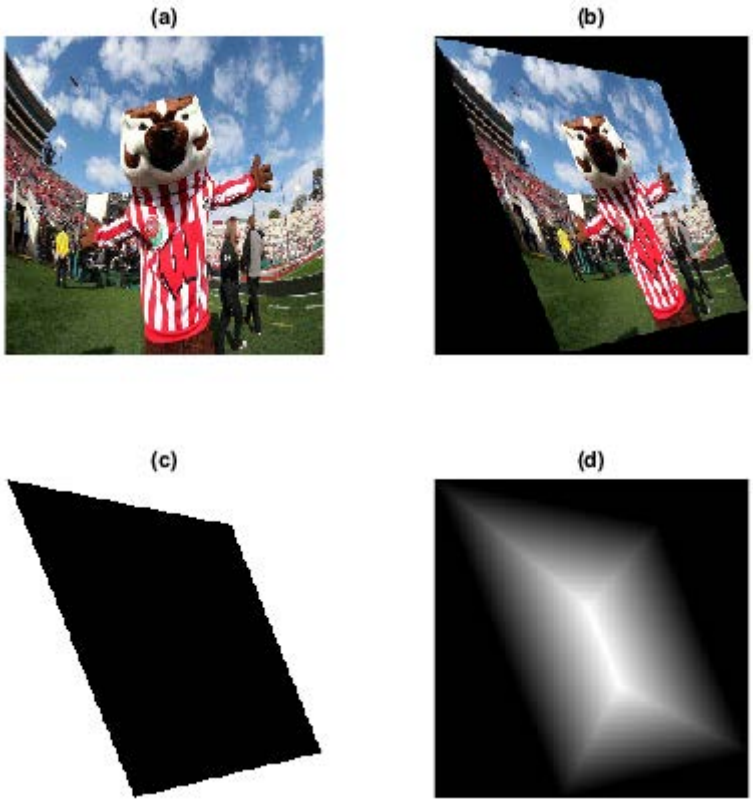

Illustration of computing the alpha channel of an input image. (a) Input image before warping. (b) Input image after warping. (c) Binarization of input - 1s outside the boundary of the warped image and 0s inside. (d) Distance transform of (c). The alpha channel should be the distance transform normalized to the interval (0, 1].

Use these alpha values as follows to compute the color at a pixel where at least two images overlap in the output image. For example, suppose there are two warped images I_1 and I_2 that overlap in the output image. Let their red channels be I_{R1} and I_{R2} and their alpha channels α_1 and α_2 respectively. Then the output red channel I_{Rout} is given by:

$$I_{Rout} = \frac{\alpha_1 I_{R1} + \alpha_2 I_{R2}}{\alpha_1 + \alpha_2}$$

Do the same for the output blue and green channels, I_{Bout} and I_{Gout}. Note that both alpha channels have to have non-zero values, otherwise you'll encounter a division-by-zero error in the above equation. The final panorama image is then simply all your warped images blended together sequentially. **You will implement blending by completing the `blend` function. You will blend the warped images into a panorama by calling `blend` in the `main` script.**

Program Instructions

In addition to code provided for computing and matching feature points using the SIFT algorithm, skeleton code is also provided in three files: `main.m` which is a script file that contains the basic parts for creating your panorama, `calcHWithRANSAC.m` which is a function that calculates a homography matrix from a given pair of images' feature points, and `blend.m` which is a function that blends two color images. Read the comments at the top of these files to

learn more about what they do. Both files have sections marked "YOUR CODE STARTS HERE" indicating places where code needs to be added. Be sure to save your output panorama as a `.jpg` image file (see below for naming information). You can also write any other supporting functions and scripts as needed.

Impementation tip: The `main` script is divided into several code sections delimited by the double percentage sign '%%'. Click 'Editor > Run Section' in MATLAB to run your current code section. This saves you from running the entire script when you only made changes to a single code section.

Hand-In Instructions

Hand in the following files and folders:

- `main.m` – the script file that computes the panorama
- `calcHWithRANSAC.m` – the function that computes the projective transformation between two sets of points and includes the RANSAC algorithm
- `blend.m` – the function file that blends two RGB images.
- `input_images` – a folder containing your own input images used to create your second panorama. There must be at least four images, named `1.jpg`, `2.jpg`, `3.jpg`, etc.
- `output_images` – a folder containing the two output panorama images, one named `test.jpg` (created from the four given test images) and the second panorama named `<NetID username>.jpg` (created from your own set of images)
- `README.txt` file that contains comments on any relevant implementation notes including parameter values used, and any extra work beyond the basic requirements, if you did any.
- Any additional scripts or functions that you wrote.

Zip all the above into a single zip file called `<NetID username>-HW4.zip` and submit this *one* file to Moodle.

Extra Credit

Instead of mapping the images onto a plane, map them onto a cylinder by using cylindrical projection as described in the lecture slides. See Section 9.1.6 in the Szeliski book for more information. To interactively view your resulting panorama you can use one of the existing viewers listed on the homework page.

Further Reading

- To understand how the VLFeat functions work, read the document on the VLFeat homepage, especially this SIFT tutorial[2]. We use `vl_sift` and `vl_ubcmatch` in this assignment.
- Information on homographies is given in the Szeliski book in Sections 2.1.2, 6.1.2, 6.1.3, and 9.1.1.
- For more information on RANSAC, see Section 6.1.4 in the Szeliski book.
- To see what a more full-fledge panorama stitcher looks like, look at the OpenCV documentation[3].

[2] http://www.vlfeat.org/overview/sift.html
[3] http://docs.opencv.org/3.0-beta/modules/stitching/doc/introduction.html