

# Homework #4: Panoramic Image Stitching

**Assigned: Monday, October 17**

**Due: Wednesday, November 2**

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, projective warping, resampling, and blending the photos into a seamless output image. See the lecture slides for more description of these steps. Specifically, implement the following steps:

## 1. Take Images

Either use some of the images you took for HW #1 or another set of images. One possible set of images is available in [testimages.zip](#). Use at least **four** images to create your panorama. If you want, you can assume the images are ordered and overlap only occurs between images immediately before and immediately after each image in the sequence. You can also assume the images are named `1.jpg`, `2.jpg` and so on 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).

## 2. Compute Feature Points

In this step you need to align the images based on feature point correspondences between overlapping pairs of images. To do this automatically use the SIFT keypoint detector, which can be found at: <http://www.cs.ubc.ca/~lowe/keypoints/>. (An alternative implementation of SIFT in MATLAB is at <http://www.vlfeat.org>) Information on the SIFT detector and descriptor is given in the text in Sections 4.1.1 and 4.1.2. Here is a description of how to use Lowe's software:

- i. Download the '[SIFT demo program \(Version 4, July 2005\)](#)' zip file to your local machine from <http://www.cs.ubc.ca/~lowe/keypoints/siftDemoV4.zip> and then unpack it to a local directory. You do this on a Linux machine so that file permissions are okay.
- ii. Read the 'README' file to understand how to use the functions, including `sift` and `match`. For example, the following two lines of code show you how to use and display the results of the function `sift`:

```
[image, descripts, locs] = sift('foo.jpg');
showkeys(image, locs);
```

`showkeys` will display a figure showing the feature points detected in image `foo.jpg`. In the figure the tail of the arrow is the position of the feature point, the direction of the arrow indicates the dominant orientation, and the length its scale.

NOTE: `sift.m` as is only accepts grayscale images. Uncomment the lines 26-28 in `sift.m` to make it able to accept color images as well.

- iii. Modify the `match.m` file so that you can access the index of the matched feature points by changing the `match` function (line 11) to:

```
function [num, match, loc1, loc2] = match(image1, image2)
```

where the first output parameter is the number of matched feature points between `image1` and `image2`, the second output parameter is the index of matched feature points in `image1` and `image2`, and the third and fourth parameters are the feature

- point locations (row, column, scale, orientation) in `image1` and `image2` respectively.
- iv. Call this modified function using:
 

```
[num, matchIndex, loc1, loc2] = match(image1, image2);
```

 where, for example, `image1` is `foo.jpg` and `image2` is `bar.jpg`. If you want to change the default parameter that determines when two points match, modify the variable called `distRatio` in `match.m`
  - v. Extract the coordinates of the matched feature points in `image1` and `image2` using
 

```
im1_ftr_pts = loc1(find(matchIndex > 0), 1 : 2);
im2_ftr_pts = loc2(matchIndex(find(matchIndex > 0)), 1 : 2);
```

 In vector `matchIndex` elements that do *not* equal 0 are feature points in `image1` that have matches in `image2`, and the values represent the positions of the corresponding feature points in `image2`. The first column of `im1_ftr_pts` keeps the row index in `image1` of every matched feature point, and the second column of `im1_ftr_pts` keeps the column index of every matched feature point. Similarly for `im2_ftr_pts`. For example, if `match` returns the `matchIndex` value `[0 0 8 0 0 15 0]` it means there are two pairs of matched features. The first pair is the 3<sup>rd</sup> feature point in `image1` and the 8<sup>th</sup> feature point in `image2`. The second matched pair is the 6<sup>th</sup> feature point in `image1` and the 15<sup>th</sup> feature point in `image2`.
  - vi. The previous step computes two  $n \times 2$  matrices, `im1_ftr_pts` and `im2_ftr_pts`, which hold the (row, column) locations of all the matched feature points in `image1` and `image2` respectively, where  $n$  is the number of matched points.

### 3. Compute Homographies

Using these point correspondences, you need to recover the parameters of the transformation between each pair of overlapping images. In our case, the transformation is a homography:  $p' = H p$ , where  $H$  is a  $3 \times 3$  matrix with 8 degrees of freedom (lower-right corner is a scaling factor and can be set to 1). One way to recover the homography is via a set of  $(p', p)$  pairs of corresponding feature points taken from the two images, as computed in the previous step. Given these correspondences, write a function of the form:

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

where `im1_pts` and `im2_pts` are  $n \times 2$  matrices holding the  $(x, y)$  locations of  $n$  point correspondences from the two images, and  $H$  is the recovered  $3 \times 3$  homography matrix. In order to compute the entries in the matrix  $H$ , you will need to set up a linear system of  $n$  equations (i.e., a matrix equation of the form  $A h = b$  where  $h$  is a column vector holding the 8 unknown entries 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). Information on homographies is given in the textbook in Sections 2.1.2, 6.1.2, 6.1.3, and 9.1.1.

Here is some sample code to compute the homography matrix  $H$ :

```
function H = calcH(p1, p2)
% Assume we have two images called '1' and '2'
% p1 is an n x 2 matrix containing n feature points, where each row
% holds the coordinates of a feature point in image '1'
```

```

% p2 is an n x 2 matrix where each row holds the coordinates of a
% corresponding point in image '2'
% H is the homography matrix, such that
% p1_homogeneous = H * [p2 ones(size(p2, 1), 1)]'
%
n = size(p1, 1);
if n < 3
    error('Not enough points');
end
A = zeros(n*3,9);
b = zeros(n*3,1);
for i=1:n
    A(3*(i-1)+1,1:3) = [p2(i,:),1];
    A(3*(i-1)+2,4:6) = [p2(i,:),1];
    A(3*(i-1)+3,7:9) = [p2(i,:),1];
    b(3*(i-1)+1:3*(i-1)+3) = [p1(i,:),1];
end
x = (A\b)';
H = [x(1:3); x(4:6); x(7:9)];

```

Unfortunately, not all the corresponding points found by SIFT will be correct matches. To eliminate outliers, you'll need to implement the **RANSAC algorithm** as part of the procedure for computing an homography. That is, select four pairs of points randomly from those returned by SIFT, compute **H** from these four pairs of points, and then see how many of the *other* pairs of points agree (i.e., a point projects very near its corresponding point in the pair). Repeat this many times (say 100) and choose the homography with the most agreement. Finally, compute the final homography using *all* the points that agree. You may want to implement RANSAC only at the end after you get a complete version working that does not use RANSAC. (For more information on RANSAC, see Section 6.1.4 in the textbook.)

One more thing: You should compute the homography matrices so as to warp each image *directly* to the reference image plane. For example, assume you have images 1, 2, 3, 4 and 5, and 3 is the reference image. You have homography matrices warping 1→2, 2→3, 4→3, and 5→4. You should create a SINGLE homography matrix for each non-reference image by composing one or more homography matrices so that you can warp directly to the reference plane rather than warping to one plane then warping to another. Here is how to do it: Assume you've got the homography matrix to warp 5→4 called **H\_54** which is a  $3 \times 3$  matrix to make  $p_4 = \mathbf{H}_{54} * p_5$ , and 4→3: **H\_43** which is also a  $3 \times 3$  matrix to make  $p_3 = \mathbf{H}_{43} * p_4$ , where  $p_5$ ,  $p_4$  and  $p_3$  are the homogeneous coordinate of feature points in image 5, 4 and 3, respectively. Also assume you have a point in image 5 that has homogeneous coordinates  $p_5$  (a  $3 \times 1$  column vector). Below are the two ways to map  $p_5$  directly into image 3:

#### First method

(a) Warp point  $p_5$  to image 4 using **H\_54** by computing  $p_4 = \mathbf{H}_{54} * p_5$ , where  $p_4$  is the corresponding point in image 4.

(b) Warp point  $p_4$  to image 3 using **H\_43** by computing  $p_3 = \mathbf{H}_{43} * p_4 = \mathbf{H}_{43} * \mathbf{H}_{54} * p_5$

(c) Finally, re-scale  $p3$  using  $p3 = p3 ./ p3(1, 3)$ . This results in the coordinates where point  $p5$  should be in image 3.

## Second method

You can see from the first method that you can alternatively first compute  $H_{53} = H_{54} * H_{43}$ . Then you compute  $p3 = H_{53} * p5$ , and re-scale using  $p3 = p3 ./ p3(1, 3)$  to get the coordinates where point  $p5$  should be in image 3.

The above two methods give the same result though the second method is better because it is easier to implement, introduces fewer artifacts, and is faster.

## 4. Warp Images

The next step is to warp all the input images onto an output compositing surface. For panoramas, it is usual to use a cylinder 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**. We call the result image a **flat panorama**.

- i. Before warping each of the images, first compute the size of the output mosaic image by computing the range of warped image coordinates for each input image. Do this by mapping the four corners of each source image using forward warping to determine its coordinates in the output image. Then compute the minimum  $x$ , minimum  $y$ , maximum  $x$ , and maximum  $y$  coordinates to determine the size of the output image. Finally, compute  $x$ -offset and  $y$ -offset values that specify the offset of the origin of the reference image relative to the origin of the output image. You may want to use the MATLAB `meshgrid` function to generate a 2D array of points for the output mosaic. Initialize all pixels to black.
- ii. Use **inverse warping** to map each pixel in the output image (that you formed in the previous step) into the planes defined by the corresponding source images. Use **bilinear interpolation** to compute the color values in the warped image. The MATLAB function `interp2` (with argument 'linear' to do bilinear interpolation) can be used for this purpose, though it may well be simpler to just implement this yourself. Here's sample code that inverse warps a pixel (at coordinates  $(r, c)$ ) in the output image onto the plane defined by the 1<sup>st</sup> image and then sets the color value appropriately:

```
% Inverse warping the coordinates. Here it is assumed that 3 is the
reference image and
% that the point (r, c) is mapped to the 1st image.
```

```
M = H_31*[r c 1]';
```

```
% Let (x,y) be the corresponding warped coordinates of (r,c) in the
coordinate system of the % 1st image
```

```
x = M(1);
```

```
y = M(2);
```

```
% Setting the color value of the output image at (r,c)
outputImage( r, c, : ) = image1( x, y, : );
```

The above procedure has to be repeated for all the pixels in the output image. Note that part of the problem is to figure out to which of the source images the pixel  $(r, c)$  in the output image gets mapped to. In the above example it is assumed that the point  $(r, c)$  maps to (in other words, lies in) the 1<sup>st</sup> image. It is also assumed that the warped coordinates  $(x, y)$  are integers, which may not always be the case (in fact it rarely is the case). If the values for  $x$  and  $y$  are decimals, then use the 'interp2' command, as mentioned earlier, to interpolate the color value at  $(x, y)$  from its 4 surrounding pixels with integer coordinates.

## 5. Blend the Images

Given the warped images and their relative displacements, the final step is to blend overlapping pixel color values in such a way as to avoid 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 text). To do this use an alpha channel where the value of alpha for an image is 1 at its center pixel and decreases linearly to 0 at all the border pixels. 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 2 images,  $I_1, I_2$ , overlapping in the output image; each pixel  $(x, y)$  in image  $I_i$  is represented as  $I_i(x, y) = (\alpha_i R, \alpha_i G, \alpha_i B, \alpha_i)$  where  $(R, G, B)$  are the color values at the pixel. Compute the pixel value of  $(x, y)$  in the stitched output image as  $[(\alpha_1 R, \alpha_1 G, \alpha_1 B, \alpha_1) + (\alpha_2 R, \alpha_2 G, \alpha_2 B, \alpha_2)] / (\alpha_1 + \alpha_2)$ . Save your output image as a jpeg file.

Here is some sample code to do simple blending:

```
% input image name
appleImgName = 'apple.jpg';
orangeImgName = 'orange.jpg';
% read images
apple = imread(appleImgName);
orange = imread(orangeImgName);
% convert to double type
apple = double(apple);
orange = double(orange);
% check to see if the dimensions are the same
if (~((size(apple, 1) == size(orange, 1)) && (size(apple, 2) == size(orange, 2))))
    error('Size does not match!');
end
% get dimension information
height = size(apple, 1);
width = size(apple, 2);
channel = size(apple, 3);
% weight vector
weightApple = 1 : -1/(width-1) : 0;
% weight matrix: repeat of weight vector
weightAppleMatrix = repmat(weightApple, size(apple, 1), 1);
% weight vector
weightOrange = 1 - weightApple;
% weight matrix: repeat of weight vector
weightOrangeMatrix = repmat(weightOrange, size(orange, 1), 1);
% create the blending image
```

```

blendingImg = zeros(size(apple));
% compute pixel value for blending image
for i = 1 : channel
    blendingImg(:, :, i) = apple(:, :, i) .* weightAppleMatrix +
    orange(:, :, i) .* weightOrangeMatrix;
end
% convert output image into uint8 type
blendingImg = uint8(blendingImg);
% display the output image
figure;
imshow(blendingImg);
title('Image blending example: appange? oranple?');
% write the output image to disk
blendingImgName = 'blending.jpg';
imwrite(blendingImg, blendingImgName, 'jpg');

```

## Problems you might face while working with SIFT

- i. Error message: './sift: Permission Denied'  
You might get this error if you don't have permission to execute the 'sift' or 'siftWin32' programs. To resolve this issue, change the permissions of these files so that you (as the user) have executable permissions with them.
- ii. Error message: "Error using ==> ctranspose, Transpose on ND array is not defined."  
Uncomment the lines 26 - 28 in `sift.m`
- iii. 'SIFT' doesn't work on Mac OS. The provided executables run on Linux or Windows machines only.

If you still face problems with SIFT you can try using the SIFT keypoint detector by VLFeat.

## Program Instructions

Your main function should be called 'panorama' and should be written in a file named 'panorama.m'. It should be called as follows:

```
output = panorama('path/to/images');
```

where `output` is a 3D matrix of type `uint8` that contains the panoramic output image. Here's how an example run of your function should work:

```
output = panorama('images/');
```

Note that you can write any other supporting functions and files as needed.

## Extra Credit

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

## Hand-in Instructions:

Please submit electronically:

- The code you wrote (as one or more `.m` files)
- The input images you used if they are not any of the ones provided in `testimages.zip`, and the resulting panorama image(s) you created (in `jpg` format)
- A `README.txt` file containing comments on how to execute your program, any relevant implementation notes, and any extra work beyond the basic system if you have done any

Put all your files into a folder and copy it to the hand-in directory:

```
cp -r foldername /p/course/cs534-dyer/handin/your-cs-login/HW4/
```