# Homework 4

CMPSCI 370 Spring 2019, UMass Amherst

Due: April 10, 11:55 PM

Instructor: Subhransu Maji

TA: Tsung-Yu Lin

## Guidelines

**Submission.** Submit your answers via Gradescope that includes a pdf with your solutions and source code. You have a total of 5 late days for all your homework assignments which you can use in any way you like. However note that delay by even one minute counts as a full late day and submissions beyond the late days will not be given *any* credit. Please contact the instructor and obtain permission ahead of time if you seek exceptions to the policy.

**Plagiarism.** We might reuse problem set questions from previous years, covered by papers and webpages, we expect the students not to copy, refer to, or look at the solutions in preparing their answers. We expect students to want to learn and not google for answers. Any instance of cheating will lead to zero credit for the homework, and possibly a failure grade for the entire course.

**Collaboration.** The homework must be done individually, except where otherwise noted in the assignments. 'Individually' means each student must hand in their own answers, and each student must write their own code in the programming part of the assignment. It is acceptable, however, for students to collaborate in figuring out answers and helping each other solve the problems. We will be assuming that you will be taking the responsibility to make sure you personally understand the solution to any work arising from such a collaboration.

**Matlab requirements.** The code should work any Matlab version ($\geq$ 2011a) and relies on the Image Processing Toolbox for some functions.

**Python requirements.** We will be using Python 2.7. The Python starter code requires `scipy`, `matplotlib` and `pillow` for loading images and `numpy` for matrix operations (at least v1.12). If you are not familiar with installing those libraries through some package manager (like `pip`), the easiest way of using them is installing Anaconda. Contact the course staff if you are having trouble installing these.

**Using other programming languages.** We made the starter code in Python and Matlab. You are free to use other languages such as Octave or Julia with the caveat that these will not be supported by the course staff.

# 1 Scale Invariant Feature Transform

Answer these questions in one or two sentences.

- **(2 points)** What are the advantages of blob detection over the Harris corner detection?

- **(2 points)** How is rotation invariance achieved in SIFT features?

- **(2 points)** When can matching patches using sum-of-squared-differences between the vector of pixel values fail? List two scenarios when the pixel values can change significantly.

- **(2 points)** List two ways how the SIFT descriptor provides robustness during feature matching?

## 2 Panoramic Image Stitching

In this part you will implement a simple panoramic image stitching algorithm. The input to the algorithm are two images which are related by a unknown translation and scale `(tx, ty, s)`. Your goal is to estimate this transformation using feature matching and RANSAC to produce a combined image. Recall that the overall steps in the alignment algorithm are:

1. Detect corners in each image (code provided/previous homework).
2. Extract features at each corner.
3. Match features based on distance in feature space.
4. Use RANSAC to estimate transformation.
5. Transform the second image using the estimated transformation (code provided).

The codebase contains an implementation of Step 1 and Step 5. You are welcome to use your own implementation of the Harris corner detector, but in case you decide to use the provided `detectCorners()` function the details are described later.

The entry code for this homework is in the file `stitchImages`. The code loads two images as seen in Figure 1 provided in the latex folder. There are multiple variants of the right image, numbered 1 through 5, corresponding to different amount of translations and scaling. For this homework all the image pairs can be perfectly aligned with by scaling and translation. In reality panoramic stitching may require solving for a general affine transformation. You are welcome to implement this for extra credit (see the lecture slides for details).

Your goal is to implement the all the steps in the alignment algorithm, report the estimated transformation, as well present visualizations of matches and the final merged image. The steps are described next.

umass_building_left.jpg umass_building_right1.jpg

Figure 1: Input images for the image stitching.

### 2.a Detect corners [0 points]

The first step is to detect corners in each image independently. You already implemented this in the previous homework, but the codebase provides an implementation in case you want to use this. The provided function `c = detectCorners(I, isSimple, w, th)` returns a single matrix $c$ which has three rows `c = [cx; cy; score]` containing the x coordinates, y coordinates and the score. Note that you have to change your homework 3 code slightly to return all three outputs as a single array. This makes it convenient to pass these as input to the feature extraction and matching code. This code finds corners by setting `isSimple=false, w=1.5, th=0.0005`. It also keeps the top 200 corners for computational efficiency.

## 2.b Feature extraction [20 points]

The next step is to implement a function `f = extractFeatures(im, c, r)` that extracts a vector of raw-pixel values from `im` centered at each corner in `c`. The basic steps in the process are:

1. Convert color image to gray
2. For each center `cx` and `cy` in the corner array `c`, extract a patch of radius `r` centered at the `[cx,cy]` from the gray-scale image. Reshape the patch to a vector (using the (:) operator or reshape function) to obtain a feature of size $(2r+1)^2$. The scheme is illustrated in Figure 2. You can pad the image by r pixels to extract patches near the edges (using `padarray()`/`np.pad()` command), but be careful to take into account the shift in the positions of the corners due to padding.

The input is `c = 3×N`, where `N` is the number of corners, and the output is a matrix `f = d×N`, where `d` $= (2r+1)^2$ is the feature dimension. The code in `stitchImages.m` file calls this function to extract features from both the images using the detected corners in the previous step.

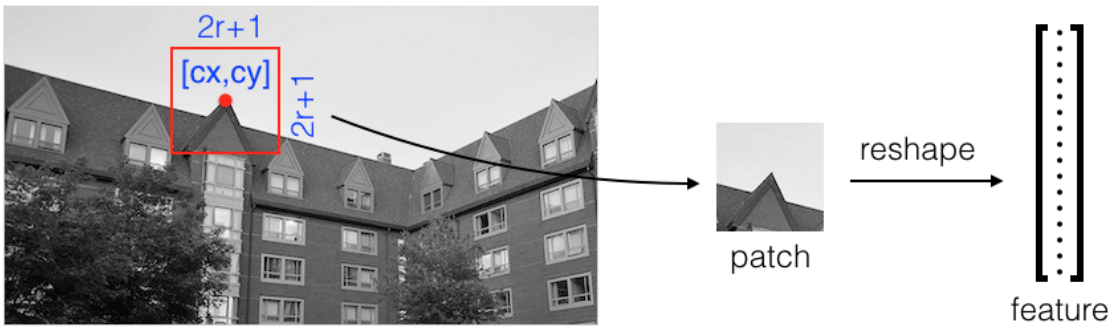


Figure 2: Patch feature of radius `r` extracted from a grayscale image at a point `[cx,cy]`.

## 2.c Computing matches [20 points]

The next step is to compute matches between the two sets of features. Implement the function `matches = computeMatches(f1,f2)` that returns the best match of each feature `f1` to `f2` using the smallest sum-of-squared-differences as the distance measure. The input are two matrices `f1=d×N` and `f2=d×M` and the output is a array of size `N×1` where each entry `matches(i)` $\in 1,\ldots,M$ is the closest feature in `f2` to the $i^{th}$ feature `f1(:,i)`. As an illustration say the first feature in `f1` matches to the $10^{th}$ feature in `f2`, then we have `matches(1) = 10`. Features that are not matched to any should be assigned a `matches(i) = 0`. Note that this can happen if you implement the ratio-test to remove unreliable matches.

Once you have implemented this function you can visualize the matches using the `showMatches(im1, im2, c1, c2, matches)` function provided in the codebase. Figure 3 shows the output of my implementation for reference.

## 2.d Estimating transformation using RANSAC [40 points]

The matching in the previous step is noisy and contains many outliers. Using RANSAC you can estimate the transformation that agrees with most matches. In particular the coordinates of the second image $(x', y')$ are related to the coordinates of the first image by an unknown scale $s$ and translation $(t_x, t_y)$:

$$x' = sx + t_x, \quad (1)$$
$$y' = sy + t_y. \quad (2)$$

You need at least two matches to estimate the scale and translation. For example, say you have two points $(x_1, y_1)$ and $(x_2, y_2)$ in the first image that get matched to $(x'_1, y'_1)$ and $(x'_2, y'_2)$ in the second image.

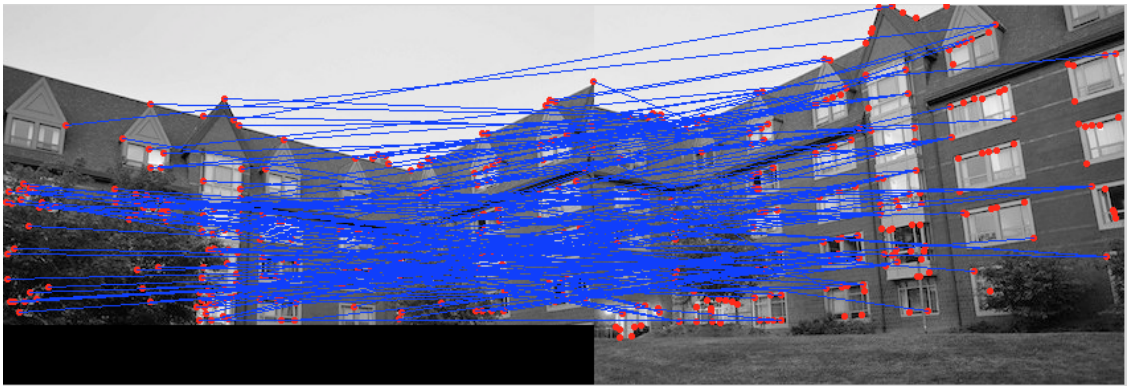

Figure 3: All matches between features extracted from the corners.

The coordinates of the points are related as:

$$x'_1 = sx_1 + t_x, \tag{3}$$
$$y'_1 = sy_1 + t_y, \tag{4}$$
$$x'_2 = sx_2 + t_x, \tag{5}$$
$$y'_2 = sy_2 + t_y. \tag{6}$$

From this one can obtain the scale $s$ as:

$$s = \frac{\sqrt{(x'_1 - x'_2)^2 + (y'_1 - y'_2)^2}}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}. \tag{7}$$

The translation parameters $(t_x, t_y)$ can be estimated by substituting the estimated $s$ in the above equations. You can also directly estimate all the three parameters by solving the above system of linear equations. See lecture slides for details.

Note that the estimated translation depends on the coordinate system of the images. The provided code for the corner detector assumes the top-left corner of the image is the origin, with x and y coordinates increasing from left to right and top to bottom respectively. Using this coordinate system the corresponding points in the right image are shifted to the left and up (the camera motion is to the right and down), so the estimated `tx` and `ty` should both be negative.

Given a transformation $T$, you can map a point in the second image to the first image using:

$$T(x'_1) = (x'_1 - t_x)/s, \tag{8}$$
$$T(y'_1) = (y'_1 - t_y)/s, \tag{9}$$

and check if it is close to its match in the first image (estimated in the earlier step) to decide if it is an inlier; For example if the squared Euclidean distance between the two is less than some threshold $\tau$:

$$(x_1 - T(x'_1))^2 + (y_1 - T(y'_1))^2 < \tau. \tag{10}$$

The RANSAC algorithm should return a transformation that has the highest number of inliers. Note that the threshold $\tau$ has to be set experimentally. You may also estimate the inliers by transforming the points from the first image to the second and checking if they are close to the destination. However since in the examples the first image is fixed, transforming the features from the destination to the source will allow you to keep the value of $\tau$ fixed across all image pairs.

Implement the function `[inliers, transf] = ransac(matches, c1, c2)`. Here `inliners` is the indices of the inliner matches, and `transf = [tx, ty, s]` is the estimated transformation from `c1` to `c2`, i.e., the estimated shift and scale of the right feature points with respect to the left feature points.

After implementation of this step you can visualize the inlier matches using the `showMatches` function (see `stitchImages` for details). Figure 4 shows the output of my implementation for reference.

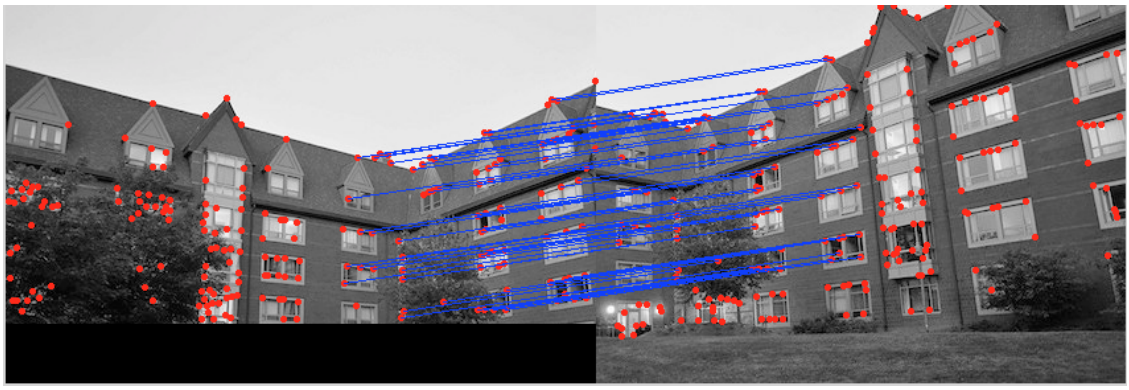

Figure 4: Inlier matches between features computed using RANSAC.

## 2.e Visualizing the stitched image [0 points]

Merge the images using `mergeImages(im1,im2,transf)` function provided in the codebase. The images should align well in the overlapping region as shown in Figure 5.

Figure 5: Result of stitching the two images

## 2.f What to submit?

Keeping the `im1` same run your algorithm for various `im2` corresponding to each of the five right images. These images are included in the homework folder. For this part of the homework your submission should include:

- The estimated transformation `[tx,ty,s]` and the number of inliers for each of the provided images in a tabular form (see below).
- The outputs of each step, i.e., matches (Figure 3), inliers (Figure 4), and the stitched image (Figure 5), for each of the five image pairs.
- The code for the files `extractFeatures.m`, `computeMatches.m` and `ransac.m` (or the python variants)

| im2 | tx | ty | s | #inliers |
|---|---|---|---|---|
| umass_building_right1.jpg | | | | |
| umass_building_right2.jpg | | | | |
| umass_building_right3.jpg | | | | |
| umass_building_right4.jpg | | | | |
| umass_building_right5.jpg | | | | |

Table 1: Estimated transformation.

### 2.g Tips

Here are some tips:

- Start with im2=umass_building_right1.jpg which will be easiest to align since the two images are roughly at the same scales, i.e., $s \sim 1$, and the transformation is a pure translation.
- The images 1 through 5 are of progressively smaller scales. Since your feature extractor is not scale invariant, matching features across scales will not be as robust leading to fewer inliers. Accordingly you may have to run RANSAC for more iterations for some of these images. SIFT features are better suited for this since the features are scale invariant.
- RANSAC algorithm is randomized so your results will be slightly different each time you run it. To make debugging easier you may fix the random seed to make the algorithm deterministic.
- Feel free to explore the parameters of the corner detector, the number of features used in matching, and the patch radius used for feature extraction. The default values work well for all the provided images, but depending on your implementation other values may give you more robust results.
- You will know when your results are correct as the images will line up as shown in Figure 5.
- Compare this approach to the Prokudin-Gorskii alignment procedure in Homework 1. Try running your code on those images to see if your RANSAC produces the same alignments as the brute-force search.

## 3 Submission and rubric

- Follow the outline on Gradescope for your submission.
- A fraction of the points for each coding assignment will be for readability and efficiency of your code. Thus it is important to properly document parts of the code you implemented.