CIS581: Computer Vision and Computational Photography
Project 3, Part A: Image Stitching
Due: Nov. 1, 2018 at 11:59 pm

Instructions

- Image Stitching is a **team** project. The maximum size of a team is **two** students.
  A team is not allowed to collaborate with another team. Only one individual from each team must submit the code for this part.
  Maximum late days allowed for this part of the project is the average of the late days of the two students in the team. The late days used will be subtracted from the individual tally of late days for each student.

- If you prefer, you can do the entire part on your own. However, completion of the project is MANDATORY and no extra credit will be offered if you do it individually.

- You **must make one submission** on Canvas. We recommend that you include a README.txt file in your submissions to help us execute your code correctly.

  – Place your **code** and **results** for part A into a folder named "Video_Mosaicing". Sub-mit this as a zip file named <Group_Number>_Project3A.zip

- Your submission folder should include the following:

  – your .m or .py scripts for the required functions

  – .m or .py demo scripts for generating the image stitch

  – any additional .m or .py files with helper functions for your code, e.g. Harris corner detectors or SIFT features

  – the input images you use

  – the resulting image stitching

  – a .pdf document containing the results of corner detection (in red dots), adaptive non-maximal suppression (in red dots) and post-RANSAC matching (outliers in blue dots) for at least five distinct frames, additional features of implementation and references to third-party code

- This handout provides instructions for two versions of the code: MATLAB and Python. You are free to select **either one of them** for this project.

- Feel free to create your own functions as and when needed to modularize the code. For MATLAB, ensure that each function is in a separate file and that all files are in the same directory. For Python, add all functions in a helper.py file and import the file in all the required scripts.

- **Start early!** If you get stuck, please post your questions on Piazza or come to office hours!

- **Follow the submission guidelines and the conventions strictly! The grading scripts will break if the guidelines aren't followed.**

1 Image Stitching

1.1 Feature Detection and Matching

In this section, you will detect features in an image frame and find the best matching features in other frames. These features should be reasonably invariant to translation and rotation.

- Feature Detection:
  In this section, you will identify features in an image using Harris corners or SIFT features. We recommend you to use detectHarrisFeatures or detectMinEigenFeatures in MATLAB and corner_harris in Python.
  Complete the following function:

  ```
  [cimg]=corner_detector(img)
  ```

  - (INPUT) `img`: $H \times W$ matrix representing the gray scale input frame
  - (OUTPUT) `cimg`: $H \times W$ matrix representing the corner-metric matrix for the image

- Adaptive Non-Maximal Suppression:
  Loop through all the feature points, and for each feature point, compare the corner strength to all the other feature points. Keep track of the minimum distance to a larger magnitude feature point (within 0.9 as large). After you have computed this minimum radius for each point, sort the list of interest points by descending radius and take the top N.
  Complete the following function:

  ```
  [x,y,rmax]=anms(cimg,max_pts)
  ```

  - (INPUT) `cimg`: $H \times W$ matrix representing the corner-metric matrix.
  - (INPUT) `max_pts`: The desired number of corners
  - (OUTPUT) `x`: $N \times 1$ matrix representing the column coordinates of the corners
  - (OUTPUT) `y`: $N \times 1$ matrix representing the row coordinates of the corners
  - (OUTPUT) `rmax`: Supression radius used to obtain `max_pts` corners

- Feature Descriptors:
  Now that you have identified points of interest, the next step is to come up with a descriptor for the feature centered at each interest point. This descriptor will be the representation you will use to compare features in different images to see if they match.

  Given an oriented interest point, you should sample a 8×8 patch of pixels around around the sub-pixel location of the interest point, using a spacing of $s = 5$ pixels between samples. After sampling, the descriptor vector should be normalized so that the mean is 0 and the standard deviation is 1. It is important to sample these patches from a larger 40×40 window to have a nice, big and blurred descriptor.
  Complete the following function:

```
[descs]=feat_desc(img,x,y)
```

- (INPUT) `img`: $H \times W$ matrix representing the gray scale input image frame
- (INPUT) `x`: $N \times 1$ matrix representing the column coordinates of the corners
- (INPUT) `y`: $N \times 1$ matrix representing the row coordinates of the corners
- (OUTPUT) `descs`: $64 \times N$ matrix, with column i being the 64-dimensional descriptor (8×8 linearized grid) computed at the location (x_i, y_i) in `img`

You may also choose to use the state-of-the-art SIFT features. The SIFT package has a README file which has clear instructions about generating SIFT features for a given image and visualizing them.

- Feature Matching:
Now that you've detected and described your features, the next step is to match them, i.e., given a feature in one image, find the best matching feature in one or more other images. Write a function to filter the correspondences using "the ratio test": (score of the best feature match)/(score of the second best feature match). You can set the threshold to 0.6. Complete the following function:

```
[match]=feat_desc(descs1,descs2)
```

  - (INPUT) `descs1`: $64 \times N_1$ matrix representing the corner descriptors of the first frame
  - (INPUT) `descs2`: $64 \times N_2$ matrix representing the corner descriptors of the second frame
  - (OUTPUT) `match`: $N_1 \times 1$ vector where $\texttt{match}_1$ points to the index of the descriptor in `descs2` that matches with the feature i in descriptor `descs1`. If the match is not found, $\texttt{match}_1 = -1$

You can use a fast-search algorithm to speed up the matching process. Some possibilities: k-d trees (knnsearch in MATLAB, annoy in Python), FLANN, wavelet indexing or Locality-Sensitive hashing.

- RAndom SAmpling Consensus (RANSAC):
We use RANSAC to pull out a minimal set of feature matches, estimate the homography and then count the number of inliers that agree with the current homography estimate. After repeated trials, the homography estimate with the largest number of inliers is used to compute a least squares estimate for the homography, which is then returned in the homography estimate H.

H is a 3×3 matrix with 8 degrees of freedom. You need to solve a system of at least 8 linear equations to solve for the 8 unknowns of H. These 8 linear equations are based on the 4 pairs of corresponding points.
Recall RANSAC:
1. Randomly select four feature pairs
2. Compute the homography relating the four selected matches with the function `est_homography.m`
3. Compute the number of inliers to count (SSD distance after applying the estimated homography below the threshold `thresh`) how many matches agree with this estimate. Don't forget to create `inlier_ind`
4. Repeat the above random selection `nRANSAC` times and keep the estimate with the largest number of inliers
5. Computes the least squares estimate for the homography using all of the matches previously estimated as inliers.
Complete the following function:

```
[H,inlier_ind]=ransac_est_homography(x1,y1,x2,y2,thresh)
```

  - (INPUT) `x1, y1, x2, y2`: $N \times 1$ vectors representing the corresponding feature coordinates in the first image and the second image. The point x_{1i}, y_{1i} in the first image are matched to x_{2i}, y_{2i} in the second image
  - (INPUT) `thresh`: The threshold on distance to determine if the transformed points agree
  - (OUTPUT) `H`: 3×3 matrix representing the homography computed at the end of RANSAC
  - (OUTPUT) `inlier_ind`: $N \times 1$ vector representing if the correspondence is an inlier or not. Denote inlier using 1 and 0 for outlier

We suggest 1000 iterations, a minimum consensus of 10 and an error of 0.5 This means trying 1000 times to find the homography based on 4 points in which at least 10 other transformed points have at most an error of 0.5 (half a pixel) to their actual correspondence points.
We strongly recommend you to play around with these values.

1.3 Frame Mosaicing

Once you have the homography, you will need to warp the images. Figure out how large the final stitched image will be and their absolute displacements in the panorama. You should warp the first and the third images to the second or the center image. You should map the pixels in the warped image to pixels in the input image so that you don't end up with holes in your image. You can use imwarp in MATLAB and scipy.ndimage.geometric_transform or scipy.ndimage.interpolation.map_coordinates in Python for the same.

You can stitch three (or more) frames to make a mosaic. You should composite the mosaic using Gradient Domain Blending. Complete the following function:

```
[img_mosaic]=mymosaic(img_input)
```

- (INPUT) `img_input`: $M \times N$ cell where M is the total number of frames in the video and N is three if the number of input videos is 3
- (OUTPUT) `img_mosaic`: $M \times 1$ cell vector representing the stitched image mosaic for every frame

2 Extra Credits:

The following tasks are for extra credit. Implementing any or all of them are optional.

2.1 Video Mosaicing

Once you have the mosaic for each of the frames in the video, combine them together to obtain a mosaiced video in .avi or .mp4 format. Complete the following function:

```
[video_mosaic]=myvideomosaic(img_mosaic)
```

- (INPUT) `img_mosaic`: $M \times 1$ cell vector representing the stitched image mosaic for every frame
- (OUTPUT) `video_mosaic`: Video file in either .avi or .mp4 format

2.2 Extra Features in Image Stitching

- Projecting your mosaic onto a cylinder or sphere
- Add multi-scale processing for corner and feature detection
- Add rotation invariance to the feature descriptors
- Create your own feature descriptor. You will need to compare it with the other descriptors
- Use SIFT features
- Implement a method that beats the "ratio test" for deciding if a feature is a valid match
- Incorporate Graphcut textures for blending image frames and compare with Poisson Blending