Problem Set 2

Introduction

Now that we have learned about some basic building blocks of computer vision, let's do something cool! The purpose of this project is to stitch images with image descriptors.

In this project, you are tasked to capture multiple sets of such images under different settings, detect and extract feature descriptors, stitch images, and finally blend the images. Basically, you'll go through all the contents learned in the past few lectures.

The following method of stitching images should work for most image sets, but you would need to be creative if you plan to work on harder image sets.

Submission: You will need to submit the code in a single zip file and a short report written (in English) in LaTex. Note: due to the nature of open problems and no provided templates, any students who submit similar codebases would undergo detailed checks for plagiarism. Again, you're allowed to discuss the problems in a group, but the code and the report must be done individually.

Q1. Implement Harris Corner Detector with non-maximal suppression. (20%)

Step 1. Calculate spatial derivative (5%)

You should choose appropriate filters that produce the gradient of the image along the X and Y axis. You can reuse the convolution function implemented in Problem Set 1. For simplification, the input image needs to be converted to grayscale. The code should be formatted as:

```
def gradient_x(img):
    # implementation
    return grad_x
```

[Input]

`img` = {ndarray: (W, H)} The image to process.

[Output]

`grad_x` = {ndarray: (W, H)} The gradient along X axis.

The function that calculates grad_y is similar in format.

Step 2. Calculate Harris response (5%)

Implement the function that calculates $R = det(M) - \alpha trace(M)^2 = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$ for each window (small image patches) when moved in both X and Y directions. Meanwhile, you should call the gradient_x and gradient_y functions in step 1.

```
def harris_response(img, alpha):
    # implementation
    return R
```

[Input]

img = {ndarray: (W, H)}   The image to process.

alpha = {float32}   Harris corner constant, usually between 0.04 - 0.06.

window_size = {int}   Size of the sliding window for the windowing function.

[Output]

R = {ndarray: (W, H)}   Harris response of each pixel.

Step 3. Select candidate corners and non-maximal suppression (10%)

Now it's time to select the pixels that are likely to be a corner. These candidates should go through a non-maximal suppression to avoid too much-duplicated information carried by nearby corner pixels. Only the candidate corner with maximal Harris response is kept within a given area.

```
def corner_selection(R, threshold, min_distance):
    # implementation
    return pixels
```

[Input]

`R` = {ndarray: (W, H)}   Harris response of each pixel.

`threshold` = {float32}   The threshold value of R which to consider as corner candidates.

`min_distance` = {int}   The minimum distance of two nearby corner.

[Output]

`pixels` = {list: N}   A list of tuples that contains N pixels selected as corners. e.g., pixels = [(5, 8), (3, 9)] means two pixels at (5, 8) and (3, 9).

Q2. Implement Histogram of Gradients. (15%)

In this part, you will implement a Histogram of Gradients (HOG) as a feature descriptor. Refer to the course slides for the details of implementation. You are not required to list the variables (e.g., window_size) as input of the function. Instead, hard-code them in the program. There's no template for coding, but you should take into consideration at least the modules below:

1. Horizontal and vertical gradients
2. Gradient direction calculation
3. Prominent gradient selection
4. Histogram for a given cell
5. Feature vector construction

Overall, the main function should be like this:

```
def histogram_of_gradients(img, pixels):
    # implementation
    return features
```

[Input]

`img` = {ndarray: (W, H)} The image to process.

`pixels` = {list: N} A list of tuples that contain N indices of pixels selected as corners in Q1.

[Output]

`features` = {ndarray: (N, L)} A list of L-dimensional feature vectors corresponding to the input `pixels`. Note that the order should be consistent with `pixels`.

Q3. Local feature matching (15%)

Given a pair of images, your task is to extract paired interest points. First, detect corners using `corner_selection` written in Q1. Next, generate corresponding features using `histogram_of_gradients` written in Q2. Finally, you need to match the two sets of features according to the euclidean distance and a certain threshold. The outputs should be two sets of pixel indices `pixels_1` and `pixels_2`. For example, `pixels_1 = [(1, 3), (2, 4)]` `pixels_2 = [(2, 5), (3, 7)]` means `(1, 3)` in img_1 matches `(2, 5)` in img_2, and `(2, 4)` in img_1 matches `(3, 7)` in img_2.

```
def feature_matching(img_1, img_2):
    # implementation
```

    return pixels_1, pixels_2

[Input]

`img_1`, `img_2` = {ndarray: (W, H)} The image to process.

[Output]

`pixels_1`, `pixels_2` = {list: N} A list of tuples `(x, y)` that contains N indices of pixels selected as corners in Q1.

Q4. Image stitching and blending (30%)

Warp a pair of images so that corresponding points align. Make full use of the functions you have written. We provide 5 pairs of images with parallel image planes. These images can also help you to validate the functions in Q.1~Q.3.

Step 1. Compute the alignment of image pairs (5%)

`compute_homography` takes two feature sets from `image_1` and `image_2`, and a list (lens>4) of feature matches and estimates a homography from image 1 to image 2.

```
def compute_homography(pixels_1, pixels_2):
    # implementation
    return homo_matrix
```

[Input]

`pixels_1`, `pixels_2` = {list: N'} A list of tuples `(x, y)` that contains N' indices of pixels.

[Output]

`homo_matrix` = {ndarray: (3, 3)} The estimated homography matrix.

Note: In `compute_homography`, you will compute the best-fit homography using the Singular Value Decomposition (SVD).

Step 2. Align the image with RANSAC (10%)

`align_pair` takes two-pixel sets, the list of pixel matches obtained from the feature detecting and matching component, a *motion model*, m (described below) as parameters. Then it estimates and returns the inter-image transform matrix `homo_matrix` using the RANSAC to compute the optimal alignment. You will need RANSAC to find the homography with the most inliners.

```
def align_pair(pixels_1, pixels_2):
    # implementation
    return est_homo
```

[Input]

`pixels_1`, `pixels_2` = {list: N'} A list of tuples `(x, y)` that contains N' indices of pixels

[Output]

`est_homo` = {ndarray: (3, 3)} The optimal homography matrix.

Step 3. Stitch and blend the image (10%)

Given an image and a homography, figure out the box bounding the image after applying the homography, warp the image into target bounding box with inverse warping and blend the pixels with their neighbors.

```
def stitch_blend(img_1, img_2, est_homo):
    # implementation
    return est_img
```

[Input]

`img_1`, `img_2` = {ndarray: (W, H)} The image to process.

`est_homo` = {ndarray: (3, 3)} The optimal homography matrix

[Output]

`est_img` = {ndarray: (W', H')} The blended image.

Note:

1. When working with homogeneous coordinates, don't forget to normalize when converting them back to Cartesian coordinates.
2. Watch out for **black pixels** in the source image when inverse warping. You don't want to include them in the accumulation.
3. When doing inverse warping, use linear or other interpolation for the source image pixels.
4. First try to work out the code by looping over each pixel. Later you can optimize your code using array instructions and numpy tricks (numpy.meshgrid, cv2.remap). You are not required to do this optimization.
5. Save the output blended image as *blend_id.png*.

Step 4. Generate a panorama (5%)

Your end goal is to be able to stitch any number of given images - maybe 2 or 3 or 4 or 100; your algorithm should work, at least not report any errors. If random paired images with no matches are given, your algorithm must report an error.

```
Def generate_panorama (ordered_img_seq)
    # implementation
    return est_pano
```

[Input]

`ordered_img_seq` = {List} The list of images to process.

[Output]

`est_pano` = {ndarray: (W', H')} The panorama image.

Note:

1. The input is an ordered sequence of images, which means the `i` th image is supposed to match with `i-1` th and `i+1` th image.
2. Save the output panorama image as *panorama_id.png*.

Q5. Analyze (20%)

With the final `stitch_blend` function, you should be able to stitch two or multiple images into a panorama. However, the quality of the stitched image is determined by lots of factors, including how you move your camera during shooting, what kind of descriptors you use, how good the descriptors are, and how you blend the images. In this problem, you need to

1. Try various settings for shooting the image sequences, e.g., (i) rotate the camera only, (ii) translate the camera only, and (iii) simultaneously rotate and translate the camera. (5%)
2. Try small and large translational/rotational distances while moving the cameras, and analyze how it affects the panorama generation. (5%)
3. Try shooting a sequence with some objects moving. What can you do to remove "ghosted" versions of the objects? (5%)
4. Try a sequence in which the same person appears multiple times. (5%)

Report the aforementioned images and results. Summarize how to get a better panorama in terms of shooting the sequences and designing the algorithms.

Extra Credits (20%)

You can choose at most two bonus problems.

BQ1 (10%)

Explore how to combine filtering and advanced blending techniques (e.g., pyramid blending, poisson blending) to get a better panorama. Make comparisons to show the improvements and analyze. (10%)

Note: third-party code for blending is allowed.

BQ2 (10%)

Explore how to use more powerful and robust feature detectors and descriptors to improve the panorama, make comparisons to show the improvements and analyze. (10%)

Note: third-party code for feature extractor is allowed.

BQ3 (10%)

Explore how to get a 360 panorama with spherical projection. You might additionally need the focal lengths of images. Make comparisons with getting 360 panorama from homography and analyze. (10%)

Note: third-party code for spherical projection is allowed.

Coding Rules

You may use NumPy, SciPy, and OpenCV2 functions to implement mathematical, filtering, and transformation operations. Do not use functions that implement keypoint detection or feature matching.

When using the Sobel operator or gaussian filter, you should use the 'reflect' mode, which gives a zero gradient at the edges.

Here is a list of potentially useful functions; you are not required to use them:

- `scipy.ndimage.sobel`
- `scipy.ndimage.gaussian_filter`
- `scipy.ndimage.filters.convolve`
- `scipy.ndimage.filters.maximum_filter`
- `scipy.spatial.distance.cdist`
- `np.max, np.min, np.std, np.mean, np.argmin, np.argpartition`
- `np.degrees, np.radians, np.arctan2`