Homework 4 Writeup

Instructions

- Describe any interesting decisions you made to write your algorithm.
- Show and discuss the results of your algorithm.
- Feel free to include code snippets, images, and equations.
- Use as many pages as you need, but err on the short side If you feel you only need to write a short amount to meet the brief, th
- **Please make this document anonymous.**

[NOTE]
I changed the function template of `get_interest_points()` and `match_features` . I added return value and parameters. For this reason, my code also has modification in `ProjLF.m` file as well, so please use the submitted `ProjLF.m`. The reason of change is described in the writeup document.

Matching

`match_features`

I used euclidean distance as a metric to define distance between two features. For each points in `features1`, I calculated distance between all points in `features2`, and find min distance and second min distance.
After the calculation end for all `features1`, sort in descending order of $1 - (min\ distance)/(second\ min\ distance)$, and using the corresponding matches at the front part of the sorted result automatically gives the points with large value of $1 - (min\ distance)/(second\ min\ distance)$ - i.e. small value of nearest neighbor distance ratio. So this gives the result of applying "nearest neighbor distance ratio test" to matches.
After implemeting matching, the accuracy was still near 0%.

Description

`get_descriptors`

[NOTE] I changed the function `match_features`'s template - I added `scale_indices` and `scales` parameter. The reason why I added them separately is described in extra credit section.

1. 16×16 intensity patches

Initially, I used 16×16 = 256 dimensional vector, which is simply normalization of image patch surrounding the point.

Using this simple descriptor resulted in accuracy of 40% on the Notre Dame, 25% on Mount Rushmore, and 9% on Episcopal Gaudi with `cheat_interest_points` set to true.

2. SIFT like descriptors

Next, I implemented SIFT like descriptors. After calculating the gradient of the image using `get_gradient()` that I implemented as an alternative of MATLAB `gradient()`, I quantized the direction into 8 steps to determine which bin of `features` to put magnitude weighted by a gaussian window.

```
for pt_idx = 1 : k
    pt_pos = [y(pt_idx), x(pt_idx)];

    window_top = pt_pos - descriptor_window_image_width / 2;
    window_bottom = window_top + descriptor_window_image_width - 1;
    window_image = image(...
        window_top(1):window_bottom(1), window_top(2):window_bottom(2));
    [dirs, mags] = get_gradient(window_image);
    dirs = min(max(ceil((dirs + 180) / 45), 1), 8);
    for cell_y = 1 : 4
        for cell_x = 1 : 4
            cell_top = cell_width * [cell_y - 1, cell_x - 1] + 1;
            cell_bottom = cell_top + cell_width - 1;

            dirs_cell = dirs( ...
                cell_top(1):cell_bottom(1), ...
                cell_top(2):cell_bottom(2));
            dirs_cell = reshape(dirs_cell, 1, numel(dirs_cell));

            mags_cell = mags( ...
                cell_top(1):cell_bottom(1), ...
                cell_top(2):cell_bottom(2));
            mags_cell = mags_cell .* gauss_window( ...
                cell_top(1):cell_bottom(1), ...
                cell_top(2):cell_bottom(2));
            mags_cell = reshape(mags_cell, 1, numel(mags_cell));

            cell_idx = 4 * (cell_y - 1) + cell_x;
```

```
            feature_cell_start = (cell_idx - 1) * 8;
            for i = 1 : numel(dirs_cell)
                features(pt_idx, feature_cell_start +
                    dirs_cell(i)) = ...
                    features(pt_idx, feature_cell_start +
                        dirs_cell(i)) ...
                    + mags_cell(i);
            end
        end
    end
end
```

Then, I normalized the features, clamped to threshold 0.2, and then normalized again. Using SIFT like descriptor resulted in accuracy of 61% on the Notre Dame, 32% on Mount Rushmore, and 10% on Episcopal Gaudi with `cheat_interest_points` set to true.

Detection

`get_interest_points`

[NOTE] I changed the function `get_interest_points`'s template - I added `scale_indices` to the return value of this function. The reason I added it additional to `scales` is described in extra credit section.

First, I implemented a function `get_harris()` to calculate Harris corner detector for each point in image.

```
function h = get_harris(image, sigma)

sigma_d = sigma * 0.7;
sigma_i = sigma;

Gd = fspecial('gaussian', 2*ceil(2*sigma_d)+1, sigma_d);
g_image = imfilter(image, Gd, 'replicate');
dy = fspecial('prewitt');
dx = dy';
Ix = imfilter(g_image, dy, 'conv', 'replicate');
Iy = imfilter(g_image, dx, 'conv', 'replicate');

IxIx = imgaussfilt(Ix .^ 2, sigma_i, 'Padding', '
    replicate');
IxIy = imgaussfilt(Ix .* Iy, sigma_i, 'Padding', '
    replicate');
IyIy = imgaussfilt(Iy .^ 2, sigma_i, 'Padding', '
    replicate');
```

```
16
17 alpha = 0.05;
18 h = (IxIx .* IyIy - IxIy .* IxIy) - alpha * (IxIx + IyIy)
       .^ 2;
19 h = (sigma_d ^ 2) * h;
20
21 end
```

Then, I selected the interest points using the code below - points whose Harris result is local maxima above a certain threshold.

```
1 threshold = 0.0000045;
2 Harris = get_harris(image, 1);
3 IsCorner = Harris > threshold & islocalmax(Harris) &
     islocalmax(Harris, 2);
4 [y_found, x_found] = find(IsCorner);
5 [~, sorted_i_found] = sort(Harris(IsCorner));
6
7 sorted_i = sorted_i_found(1:3000);
8 y = y_found(sorted_i);
9 x = x_found(sorted_i);
```

I eliminated the points that are close to image boundary regarding the size of `descriptor_window_image_w`. The eliminating part is omitted in the code snippet above.
Since now that the detection is implemented, I tested with `cheat_interest_points` set to false, and resulted in accuracy of 93% on the Notre Dame, 94% on Mount Rushmore, and 6% on Episcopal Gaudi.

Extra Credit - Adaptive non-maximum suppression

Below code shows my implementation of addaptive non-maximum suppression. It checks if all the points in the circle of radius 24 surrounding the point has value smaller than 0.9 * (center point's value). Instead of checking only the surrounding points identified as corner by Harris (locla maxima above threshold) using `bwconncomp`, my code redundantly checks all the surrounding points that lies in the radius. This could be optimized, but using vectorization, this part of the code didn't take long time, so I just did as below for simple implementation.

```
1 rad = 24;
2 circle_mask = get_circle_mask(rad);
3
4 for i = 1 : size(y_found, 1)
5     cy = y_found(i);
6     cx = x_found(i);
```

```
    if ~(cy-rad >= 1 && cx-rad >= 1 && cy+rad <= img_h &&
        cx+rad <= img_w && ...
            all(all( ...
            Harris(cy-rad : cy+rad, cx-rad : cx+rad) .*
               circle_mask) < ...
            Harris(cy, cx) * 0.9))
        IsCorner(cy, cx) = false;
    end
end

function h = get_circle_mask(radius)
hsize = radius * 2 + 1;
h = zeros(hsize);
for i = 1 : hsize
    for j = 1 : hsize
        if hypot(i - (radius+1), j - (radius+1)) <=
           radius
            h(i, j) = 1;
        end
    end
end
h(radius+1, radius+1) = 0;
end
```

Applying non-maximum suppression resulted in accuracy of 97% on the Notre Dame, 98% on Mount Rushmore, and 12% on Episcopal Gaudi.

Extra Credit - Scale selection to pick the best scale in detection

In order to pick the best scale, I first made a scale space and stored in `scale`.

```
scale = zeros(1, 5);
for s = 1 : size(scale, 2)
    scale(s) = 0.75 * (1.4 ^ (s - 1));
end
```

Then I calculated Harris detector and stored it in `[img_h, img_w, img_c]` size image, where `img_c` is the dimension of the scale space. For each scale, the Harris calculation differed by passing different argument for sigma like below.

```
for s = 1 : img_c
    Harris = get_harris(image, scale(s));
    % ... omitted ...
    HarrisAll(:, :, s) = Harris;
    HarrisCorner(:, :, s) = IsCorner;
```

```
end
```

Then, I picked the scale by examining which scale makes local maxima like below.

```
% ...
Logs = zeros([img_h, img_w, img_c]);
for s = 1 : img_c
    Logs(:, :, s) = get_log(image, scale(s));
end
HarrisCorner = HarrisCorner & islocalmax(Logs, 3);
% ...

function h = get_log(img, sigma)
log_filt = fspecial('log', 2*ceil(2*sigma)+1, sigma);
h = sigma * sigma * abs(imfilter(img, log_filt, 'conv', '
   replicate'));
end
```

[NOTE] The reason I added `scale_indices` to return values of `get_interest_points()`:
Let's denote the number of detected interest point as k. As shown in the code below, Instead of storing scale in k-dimensional `scale`, I used `scale` as 5-dimensional vector to store the scale space I used, and for each k points, I stored index of `scale` to k-dimensional `scale_indices`. The reason I chose this way is described in the next section.

```
i_found = find(HarrisCorner);
[y_found, x_found, c_found] = ind2sub(size(HarrisCorner),
    i_found);

HarrisVal = HarrisAll(HarrisCorner);
[HarrisVal, sorted_i_found] = sort(HarrisVal, 'descend');

sorted_i = sorted_i_found(1:min(size(sorted_i_found, 1),
   9000));
y = y_found(sorted_i);
x = x_found(sorted_i);
scale_indices = c_found(sorted_i);
confidence = HarrisVal(sorted_i);
```

Extra Credit - Scale descriptor corresponding the the scale for the interest point

In order to reflect the scale selection to the descriptor as well, I used gaussian window corresponding to the point's scale, to weight the magnitude. In this way, using multi-

ple gaussian window with different variance in **detection** and also use corresponding gaussian window in **descriptor**, I could achieve effect of examining images in multiple scales by blurring with gaussian window, without really downsampling the image pixels. To apply gaussian window of corresponding variance of scale selected by detection, I first built 5 gaussian windows, each corresponding to 5 dimensional vector `scale` returned by `get_interest_points()`.

```
gauss_windows = zeros([descriptor_window_image_width,
   descriptor_window_image_width, size(scales, 2)]);
for i = 1 : size(scales, 2)
    gauss_windows(:, :, i) = ...
        fspecial('gaussian',
           descriptor_window_image_width, ...
        scales(i) * 8);
end
```

Then, when calculating the descriptor, I applied the corresponding guassian window using the `scale_indices` to see which guassian window should be applied.

```
mags_cell = mags_cell .* gauss_windows( ...
    cell_top(1):cell_bottom(1), ...
    cell_top(2):cell_bottom(2), ...
    scale_indices(pt_idx));
```

[NOTE] The reason I added `scale_indices` and `scales` to `match_features()`'s parameters is that I wanted to avoid creating new gausssian window of corresponding scale for each interest point. In this way, I can just pre-declare 5 gaussian windows and retrieve to use them by indices.
The return value `scale_indices` and `scale` from `get_interest_points()` are each used as parameter `scale_indices` and `scales` of `match_features()`.

Applying the scale for both (1) detection and (2) description resulted in accuracy of 97% on the Notre Dame, 95% on Mount Rushmore, and 13% on Episcopal Gaudi with `cheat_interest_points` set to true. I thought the reason why the accuracy decreased for Notre Dame and Mount Rushmore is because those two pairs', the pictures' scales are not very different, so applying the scale selection for detection and using the descriptor corresponding to it didn't effect much for the performance and just made more parameters. For Episcopal Gaudi, the scale of picture is quite different, and the accuracy did improve, but it wasn't a very dramatic improvement.

Below figure shows the matching result of my final implementaion.

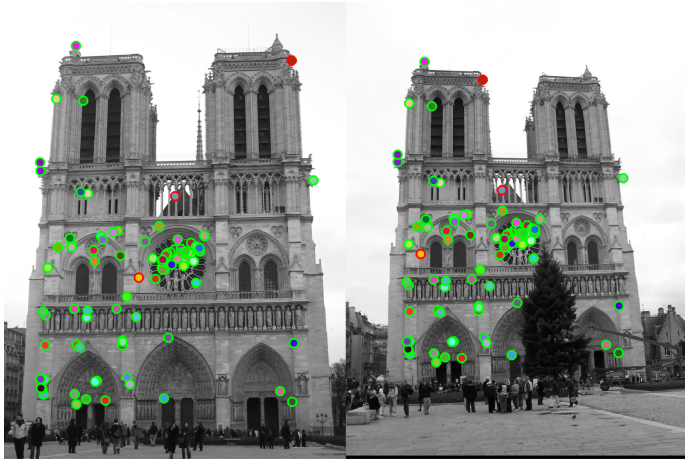

Figure 1: Result of match of Notre Dame pair. Accuracy 97% for highest confidence 100 matches.

Figure 2: Result of match of Mount Rushmore pair. Accuracy 95% for highest confidence 100 matches.

Figure 3: Result of match of Episcopal Gaudi pair. Accuracy 13% for highest confidence 100 matches.