ASSIGNMENT 3
(2018702012)

1) SIFT

My Sift Algorithm:

1. Set *num_octaves*, *num_scales*, *sigma*, *contrast_threshold*.
2. Preprocess the two images and assign to *images* cell.
3. For each *img* in *images.*
4. Create *pyramid* where number of images is equal to *num_octaves*. In each *octave* of the pyramid, first image is image of previous octave downsampled to half.
5. Create *blurred_images* equal to the number of *s = num_scales + 3* in each *octave* of the *pyramid*. Images are blurred using Gaussian Filter where standard deviation is *(2^(s/num_scales)) * sigma*.
6. Create *DoGs* for each *octave*. Number of images in *DoGs* will be *num_scales + 2*.
7. Find *keypoints* using *DoGs* which have maximum value in respective 26 neighbourhood and are greater than the *contrast_threshold*.
8. For each *keypoint* in *keypoints.*
9. Select suitable *blurred_image* of scale *s.*
10. Take 16 X 16 *window* around it and calculate *gradient* and *gradient_directions.*
11. Divide the *window into 4X4 subcells*.
12. For each *cell* in *subcells.*
13. Calculate 8 bin oriented histogram. Append it to *descriptor* of the corresponding *keypoint*.
14. End For.
15. Get actual location of the *keypoint* by multiplying current coordinates with *2^(keypoint_octave-1)*. Append this location to previous *keypoint_loc.*
16. End For.
17. End For.
18. Match features using the above obtained *descriptors*.

Challenge:
The dense sift algorithm took a lot of time to run (few hours) for actual image size. Due to this problem, the images were downsampled before the dense sift matching.

2) Intensity window-based correlation matching

Theory:
Initially, each pixel in left image as well as right image is considered to be a valid feature candidate. So, for each feature in the left image, descriptor is constructed by taking *win_size X win_size* window around it, keeping it at center. This descriptor is matched, using intensity based correlation, to all such descriptors of all the pixels lying on the corresponding row of the right image. Which ever gives maximum correlation value above a certain *match_thresh*, is considered a match. But this method didn't give a clear or any good picture of the matches. So I further used a detector to consider only a few feature points. For the finding suitable feature points, Harris corner detector is used because the images are of same size.

Algorithm:

1. Set *win_size*, *match_thresh.*
2. For each *r* in *rows(Left_image)*.
3. Generate *descriptors* considering a window of size *win_size X* win_size for all pixels lying on *r* in the *Right_image*.
4. For each *c* in *columns(Right_image)*.
5. Generate *descriptor* considering pixel *Left_image(r,c)* as center pixel.
6. Find *correlation* of *descriptor* with all the *descriptors*.
7. Set *max_corr = maximum(correlation).*
8. If *max_corr* > *match_thresh*, consider it as a suitable match.

Challenge:
If all points in left image are considered suitable feature points, then finding suitable match for keypoints of left image in right image using correlation gives a lot of matches. These matches are not unique – they are many to one and errorneous. That is why harris detector is used to atleast give a small set of candidate feature points which can be matched across the two images using intensity window-based correlation.

3) Comparison of Method 1 and Method 2:

1. The matches obtained using intensity window-based correlation without Harris Detector (Method 2) are poor when compared to those obtained using SIFT (Method 1).
2. Matches in Method 2 (without Harris Detector) are Many-To-Many whereas in DSIFT the matches are One-To-One.

4) Rectified images

Method:
In order to rectify the images, first fundamental matrices were found for all the three methods – my sift, vl-sift and window-based(using Harris Detector). Using the fundamental matrix, respective images were rectified.

Observation:
1) Rectification obtained using matches found by window-based matching (without Harris Detector) gives wrong fundamental matrix due to poor matches. Hence the rectification obtained is errorneous only.
2) Rectification obtained for lion image using window-based matching(Using Harris Detector) was very poor due to very less matched feature points which resulted in errorneous fundamental matrix, hence poor rectification.
3) Rectified images obtained using matches of My-SIFT and DSIFT look better.

5) Greedy Matching and DTW

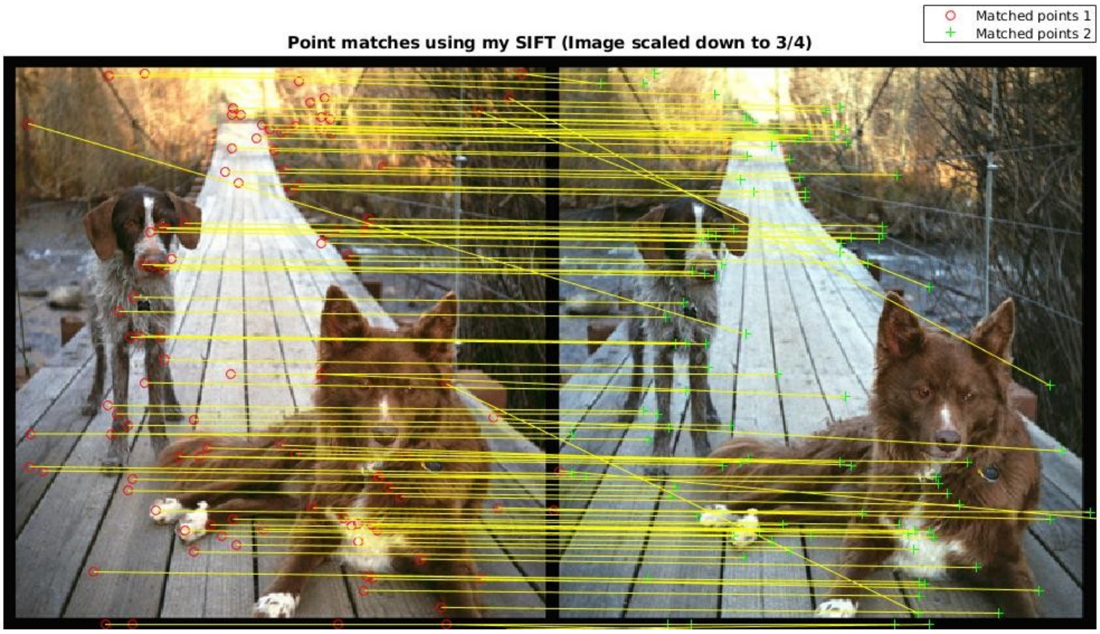
In greeding matching even after rectification, we can see a lot of errorneous matches between the two images.

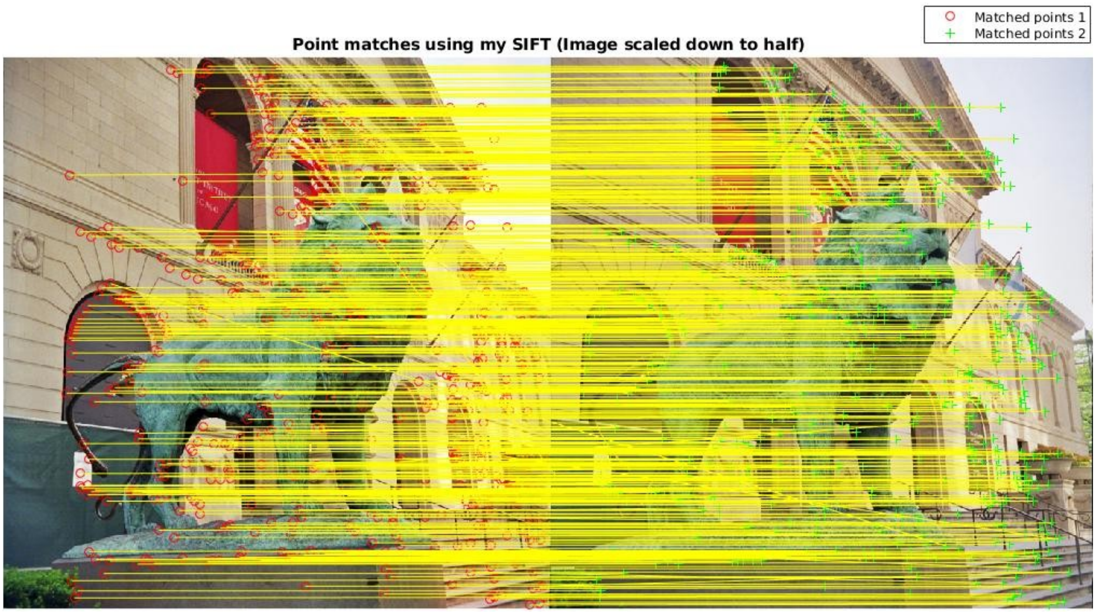
Image Name	DTW Error	Greedy Matching Error
Stereo_Pair1.jpg	7.1864e+05	4.5483e+07
Stereo_Pair2.jpg	2.6703e+05	6.0417e+06
Stereo_Pair3.jpg	2.7370e+05	5.3191e+06

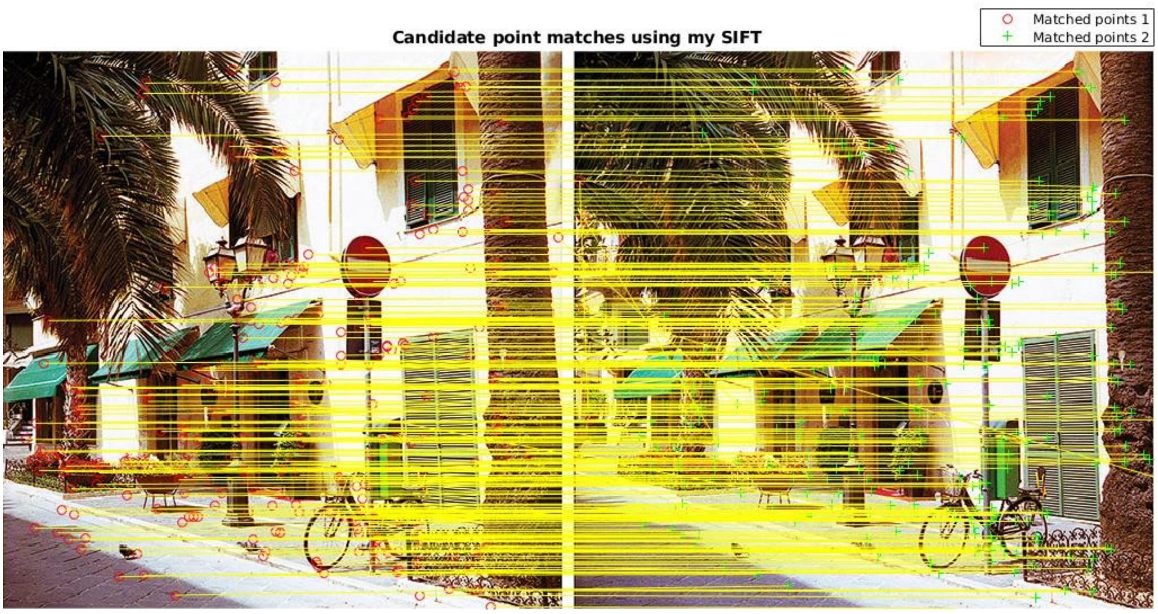
Greedy matching error is always greater than DTW error because it is greedily trying to measure pixel-by-pixel difference between the two images.

Results:

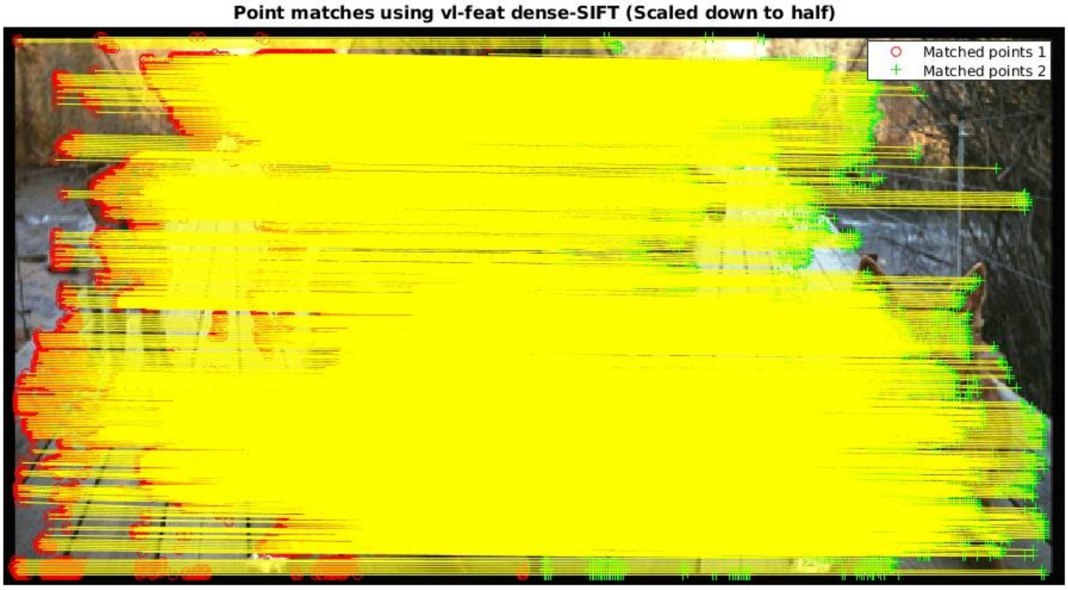
Matched points using my SIFT.

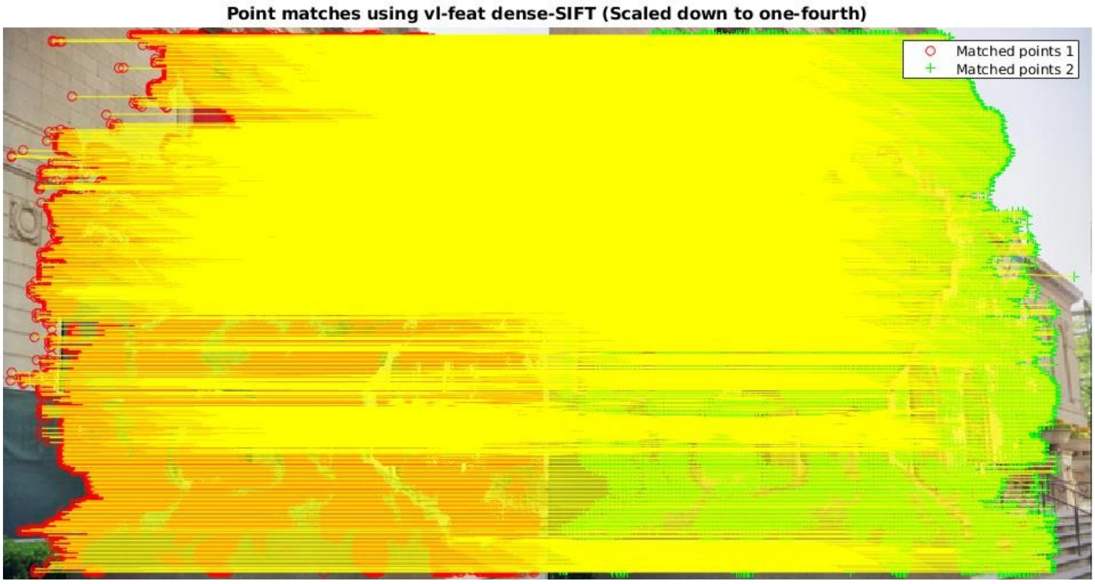


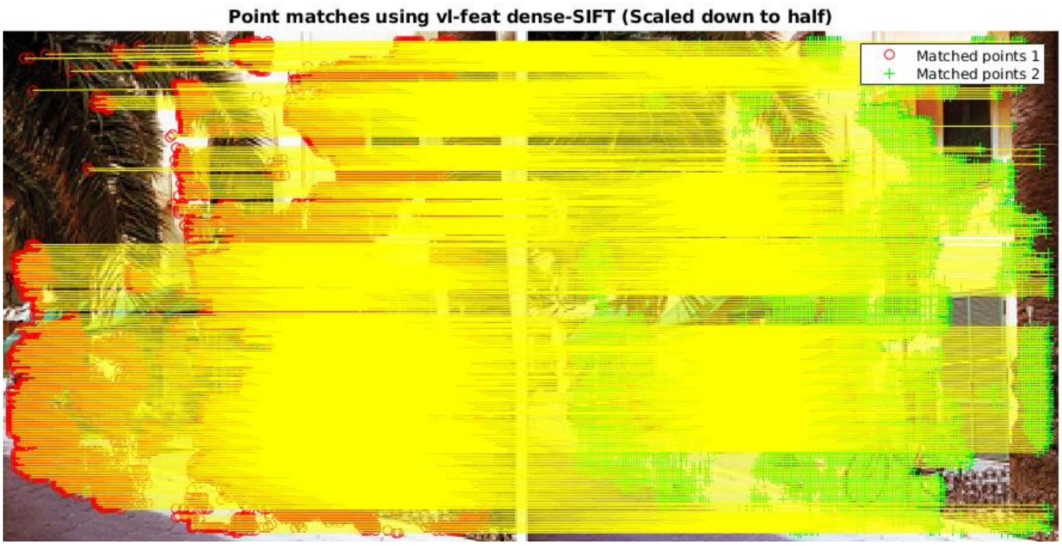




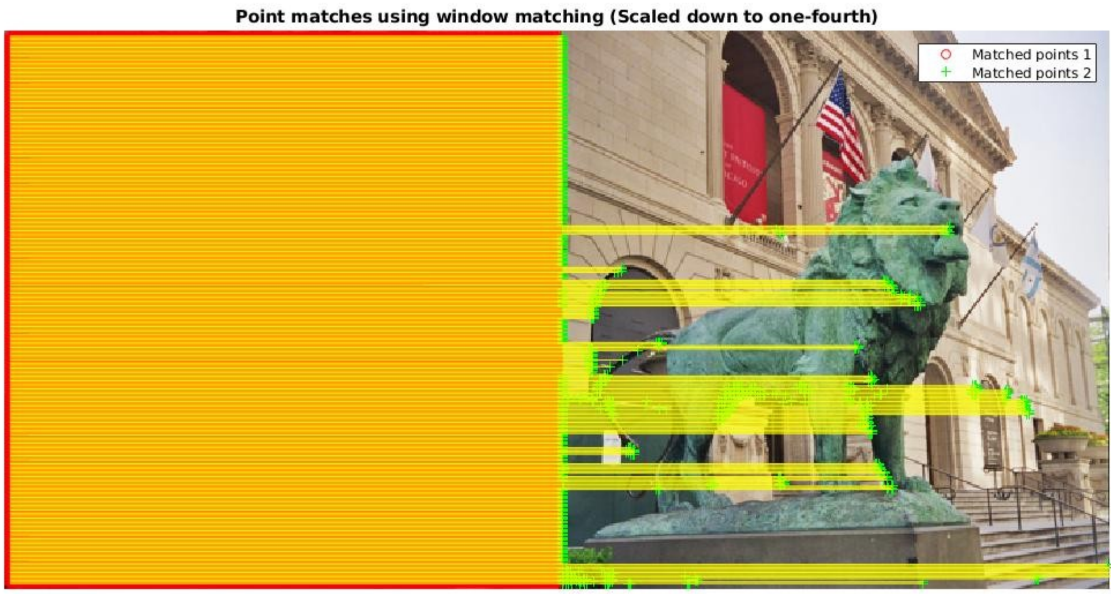
Matched points using vl-DSIFT

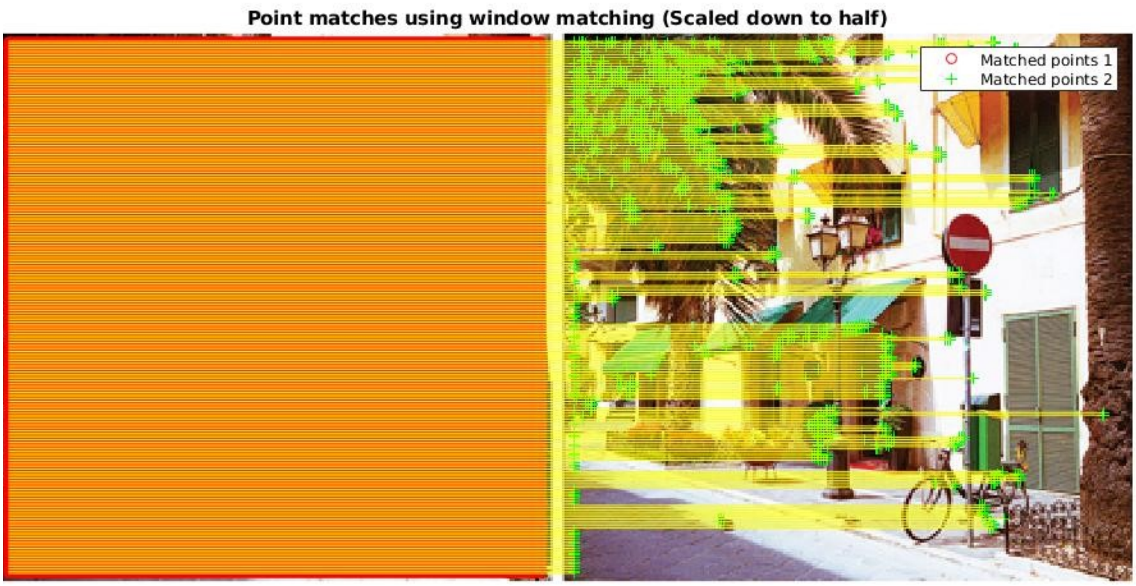




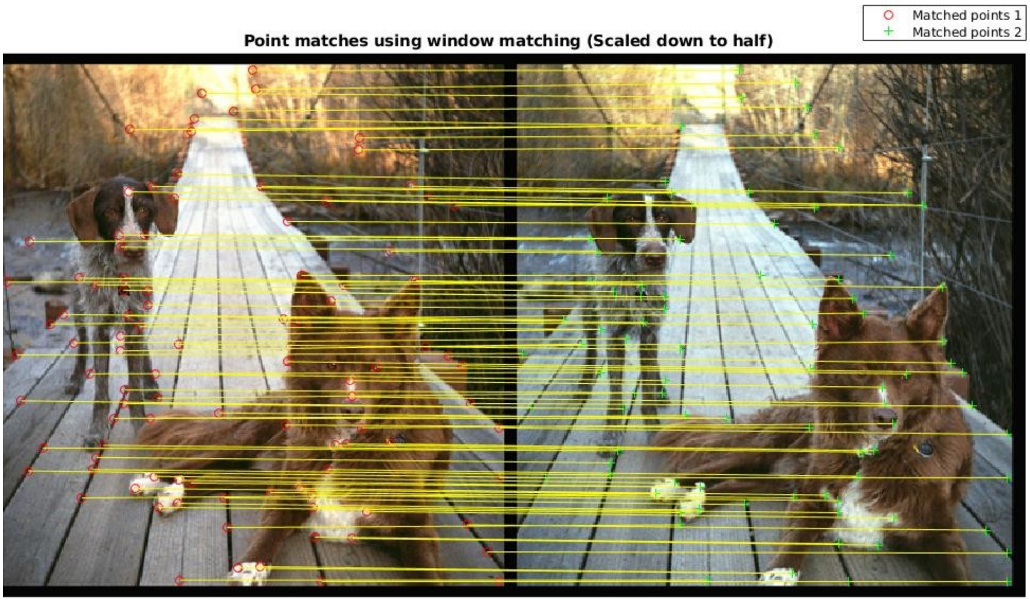


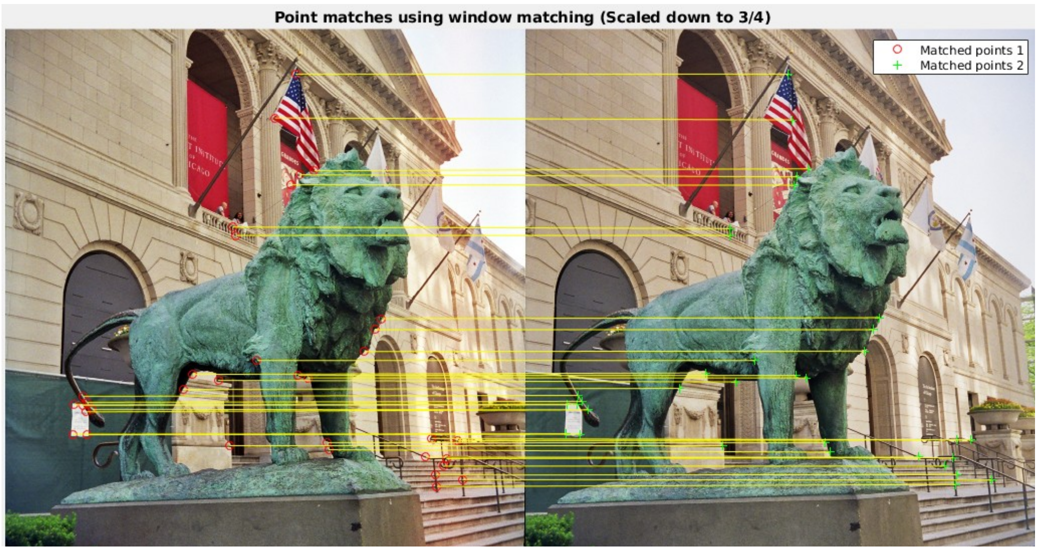
Matched points using Intensity window based correlation:

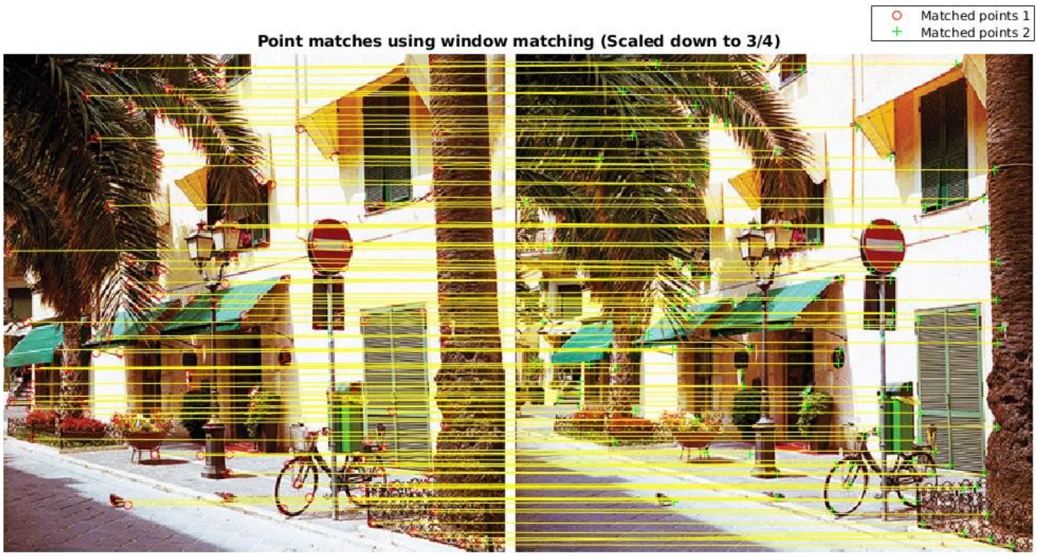




Matched points using Intensity window based correlation(with Harris Detector):







Rectified images: (With window matching using Harris Detector)

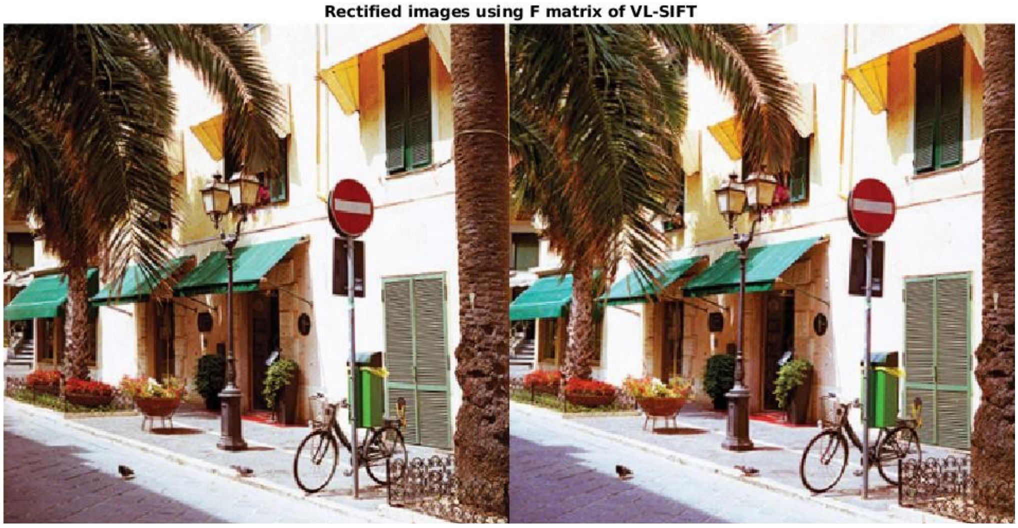
Rectified images using F matrix of MY SIFT

Rectified images using F matrix of VL-SIFT

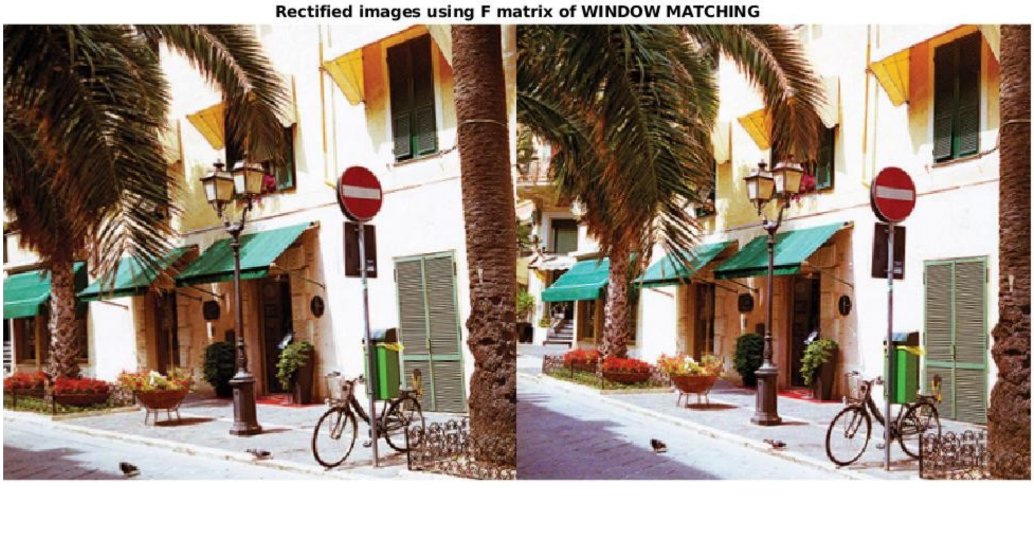
Rectified images using F matrix of WINDOW MATCHING

Rectified images using F matrix of MY SIFT

Rectified images using F matrix of VL-SIFT



Rectified images using F matrix of WINDOW MATCHING

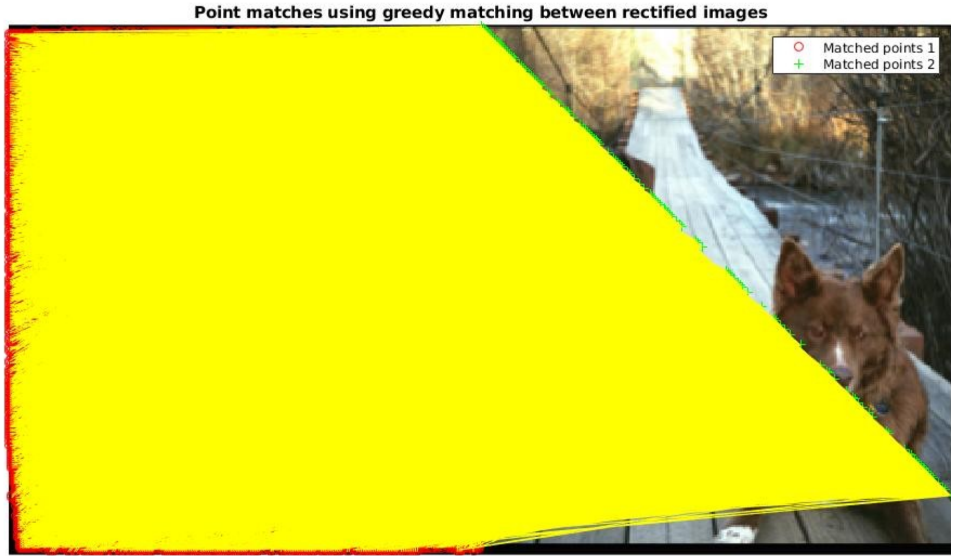


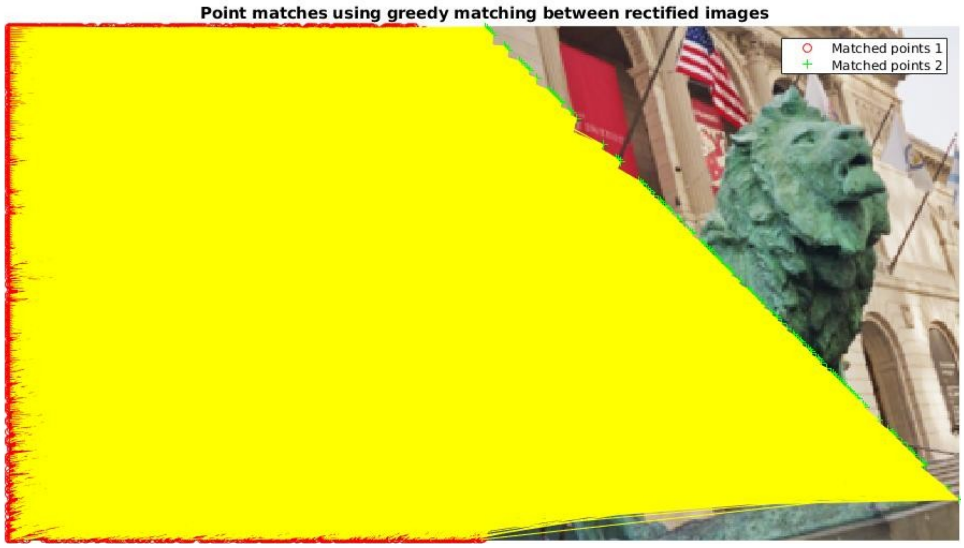
Rectified images using F matrix of MY SIFT

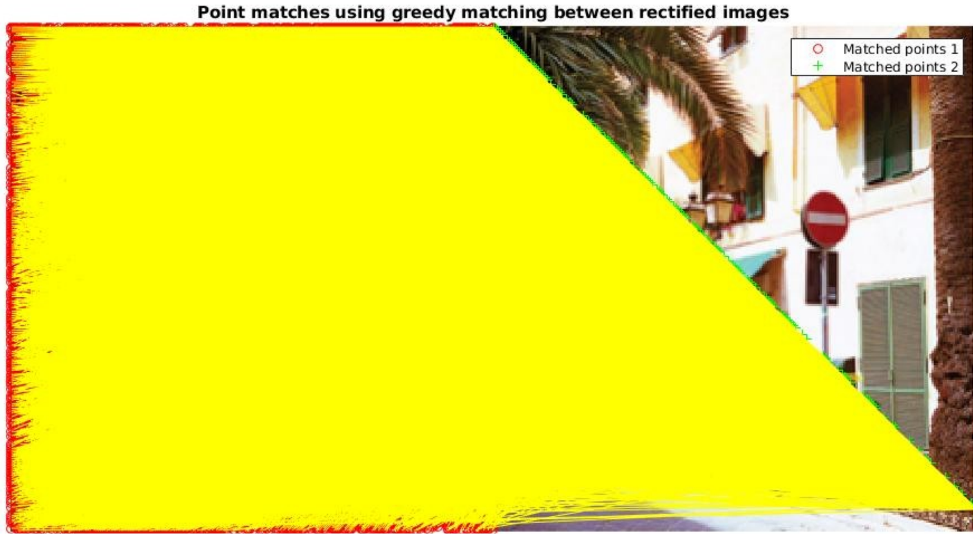
Rectified images using F matrix of VL-DSIFT

Rectified images using F matrix of WINDOW MATCHING(Using Harris)

Greedy matching on rectified images:







Code:

```
clear all;
close all;
clc;

% Take input image
I = imread('Stereo_Pair3.jpg');
% Divide the image into two equal halves
L = I(1:size(I,1), 1:size(I,2)/2, 1:size(I,3));
R = I(1:size(I,1), 1+size(I,2)/2:size(I,2), 1:size(I,3));

scale = 0.5; % Downsamples the image to increase speed

scaled_L = imresize(L,scale);
scaled_R = imresize(R,scale);

run('/home/user/Downloads/softwares/vlfeat-0.9.21/toolbox/vl_setup');

%% SIFT implemented by me
[~, ~, matched_kpts] = sift(L, R, scale);
figure(1); ax = axes;
showMatchedFeatures(scaled_L, scaled_R,...
    matched_kpts{1}, matched_kpts{2}, 'montage','Parent',ax);
title(ax, 'Point matches using my SIFT (Image scaled down to one-fourth)');
legend(ax, 'Matched points 1','Matched points 2');

%% Using vl-feat SIFT
[f1, d1] = vl_dsift(single(rgb2gray(scaled_L)));
[f2, d2] = vl_dsift(single(rgb2gray(scaled_R)));
[matches, ~] = vl_ubcmatch(d1, d2);
pts1 = f1(1:2,matches(1,:));
pts2 = f2(1:2,matches(2,:));
figure(2); ax = axes;
showMatchedFeatures(scaled_L, scaled_R,...
    pts1', pts2', 'montage', 'Parent',ax);
title(ax, 'Point matches using vl-feat dense-SIFT (Scaled down to one-fourth)');
legend(ax, 'Matched points 1','Matched points 2');

%% Every pixel window matching
win_size = 3;
match_lambda = 0.85;

rows = size(scaled_L, 1); cols = size(scaled_R, 1);
buff = (win_size - 1) / 2;

matches = {};
for r = 1+win_size:rows-win_size

    R_vect = {};
    vect_idx = 0;
    for c = 1+win_size:cols-win_size
        vect_idx = vect_idx + 1;
        w = scaled_R(r-buff:r+buff, c-buff:c+buff);
        R_vect{vect_idx,1} = w(:)';
    end
    R_vect = cell2mat(R_vect);

    for c = 1+win_size:cols-win_size
        w = scaled_L(r-buff:r+buff, c-buff:c+buff);
        L_vect = w(:)';
```

```matlab
        correlation = sum(L_vect .* R_vect, 2) ./ ...
            (sqrt(sum(L_vect .^ 2, 2)) * sqrt(sum(R_vect .^ 2, 2)));
        [max_corr, corr_idx] = max(correlation);
        if ~isempty(max_corr)
            if (max_corr > match_lambda)
                matches = vertcat(matches,...
                    [r, c, r, corr_idx+win_size]);
            end
        end

    end

end

matches = cell2mat(matches);
figure(3); ax = axes;
showMatchedFeatures(scaled_L, scaled_R,...
    fliplr(matches(:,1:2)), fliplr(matches(:,3:4)),...
     'montage', 'Parent',ax);
title(ax, 'Point matches using window matching (Scaled down to half)');
legend(ax, 'Matched points 1','Matched points 2');
%% Using window matching
win_size = 3;
match_lambda = 0.85;
win_matches = corrMatchImages(L, R, scale, win_size, match_lambda);

figure(4); ax = axes;
showMatchedFeatures(scaled_L, scaled_R,...
    fliplr(win_matches(:,1:2)), fliplr(win_matches(:,3:4)),...
     'montage', 'Parent',ax);
title(ax, 'Point matches using window matching (Scaled down to one-fourth)');
legend(ax, 'Matched points 1','Matched points 2');

%% Rectification of the images after matching using above methods

% Estimate Fundamental Matrices for three methods
F_my_sift = estimateFundamentalMatrix(matched_kpts{1}, matched_kpts{2});
F_vl_sift = estimateFundamentalMatrix(pts1', pts2');
F_win = estimateFundamentalMatrix(fliplr(win_matches(:,1:2)),...
    fliplr(win_matches(:,3:4)));
F_gw = estimateFundamentalMatrix(fliplr(matches(:,1:2)),...
    fliplr(matches(:,3:4)));

% Rectified images
[t1_my_sift, t2_my_sift] = estimateUncalibratedRectification(F_my_sift,...
    matched_kpts{1}, matched_kpts{2}, size(R));
[rectL_mysift, rectR_mysift] = rectifyStereoImages(...
    scaled_L, scaled_R, t1_my_sift, t2_my_sift);
[t1_vl_sift, t2_vl_sift] = estimateUncalibratedRectification(F_vl_sift,...
    pts1', pts2', size(R));
[rectL_vlsift, rectR_vlsift] = rectifyStereoImages(...
    scaled_L, scaled_R, t1_vl_sift, t2_vl_sift);
[t1_win, t2_win] = estimateUncalibratedRectification(F_win,...
    fliplr(win_matches(:,1:2)), fliplr(win_matches(:,3:4)), size(R));
[rectL_win, rectR_win] = rectifyStereoImages(...
    scaled_L, scaled_R, t1_win, t2_win);
[t1_gw, t2_gw] = estimateUncalibratedRectification(F_gw,...
    fliplr(matches(:,1:2)), fliplr(matches(:,3:4)), size(R));
[rectL_gw, rectR_gw] = rectifyStereoImages(...
    scaled_L, scaled_R, t1_gw, t2_gw);

% Show rectified images
```

```
figure(5); imshow([rectL_mysift, rectR_mysift]);
title('Rectified images using F matrix of MY SIFT');
figure(6); imshow([rectL_vlsift, rectR_vlsift]);
title('Rectified images using F matrix of VL-DSIFT');
figure(7); imshow([rectL_win, rectR_win]);
title('Rectified images using F matrix of WINDOW MATCHING(Using Harris)');
figure(8); imshow([rectL_gw, rectR_gw]);
title('Rectified images using F matrix of WINDOW MATCHING');


%% Greedy matching
rows = size(rectL_vlsift,1); cols = size(rectL_vlsift,2);

gmatches = {};
midx = 0;
for rl = 1:rows
    for cl = 1:cols
        has_match = false;
        rr = 0; cr = 0;
        while(~has_match && rr<=rows && cr<=cols)
            rr = 1+rr; cr = 1+cr;
            if rectL_vlsift(rl,cl) == rectR_vlsift(rr,cr)
                midx = midx + 1;
                gmatches{midx,1} = [rl,cl,rr,cr];
                has_match = true;
            end
        end
    end
end
gmatches = cell2mat(gmatches);

figure(9); ax = axes;
showMatchedFeatures(rectL_vlsift, rectR_vlsift,...
    fliplr(gmatches(:,1:2)), fliplr(gmatches(:,3:4)),...
     'montage', 'Parent',ax);
title(ax, 'Point matches using greedy matching between rectified images');
legend(ax, 'Matched points 1','Matched points 2');


%% DTW error
[dtw_dist,~,~] = dtw(double(rgb2gray(rectL_vlsift)), ...
    double(rgb2gray(rectR_vlsift)));

%% Greedy error
greedy_dist = sum(sum(sum(abs(rectL_vlsift - rectR_vlsift))));


function [descriptors, kpt_loc, kpt_matched] = sift(img_1, img_2, scale)
%SIFT Finds SIFT detectors and descriptors between two image files
%   Detailed explanation goes here
% Inputs:
%   file_1 = Image 1 file name
%   file_2 = Image 2 file name
%   scale = Downsamples the image
% Outputs:
%   descriptors = 128 dimension vector for each keypoint
%   kpt_locations = Keypoint locations

    num_scales = 3; % Scales per octave.
    num_octaves = 5; % Number of octaves.
    sigma = 1.6; % Gaussian smoothening factor.
    contrast_threshold = 0.02; % Threshold to invalidate noisy keypoints.
```

```
    images = {processImage(img_1, scale), processImage(img_2, scale)};

    kpt_loc = cell(1, 2); % Locations of the keypoints.
    descriptors = cell(1, 2); % 128 dimension descriptors of the keypoints.
    kpt_matched = cell(1,2); % Matching keypoints.

    % Create Pyramid of octaves containing blurred images and DoGs. Finding
    % keypoints and descriptors.
    for img_idx = 1:2
        pyramid = createPyramid(num_octaves, cell2mat(images(img_idx)));
        blur = createBlurOctaves(pyramid, num_scales, sigma);
        dog = createDogOctaves(blur);
        keypts = createKeypoints(dog, contrast_threshold);
        [descriptors(img_idx), kpt_loc(img_idx)] = ...
            genDescriptors(keypts, blur);
    end

    % Show matched keypoints

    indexPairs = matchFeatures(descriptors{1}, descriptors{2},...
        'MatchThreshold', 100, 'MaxRatio', 0.45, 'Unique', true);

    % Flip row and column to change to image coordinate system.
    kpt_match_1 = fliplr(kpt_loc{1}(indexPairs(:,1), :));
    kpt_match_2 = fliplr(kpt_loc{2}(indexPairs(:,2), :));

    kpt_matched{1} = kpt_match_1;
    kpt_matched{2} = kpt_match_2;

end


function image = processImage(I, rescale_factor)
% PROCESSIMAGE Processes the image
%   Converts RGB to gray, downsamples the image and converts data type to
%   double.
% Inputs:
%   I = Image
%   rescale_factor = Scale for downsampling the image
% Output:
%   image = Processed image

    if size(I,3) == 3
        I = rgb2gray(I);
    end

    image = im2double(imresize(I, rescale_factor));
end


function [pyramid] = createPyramid(num_octaves, I)
%CREATEPYRAMID Creates pyramid for the SIFT
% Inputs:
%   num_octaves = Number of octaves in the pyramid
%   I = Image for which octaves must be created
% Output:
%   pyramid = Pyramid of original image and sampled version of the image

    pyramid = cell(1, num_octaves);
    for oct_idx = 1:num_octaves
        % Downsample the images by 2^o where o = [0,..., num_octaves-1]
```

```
        J = imresize(I, 0.5^(oct_idx-1));
        pyramid(oct_idx) = {J};
    end

end


function [blur] = createBlurOctaves(pyramid, num_scales, sigma)
%CREATEBLUROCTAVES Creates blurred images within each octave
% Inputs:
%   pyramid = Pyramid of original image and down-sampled images
%   num_scales = Number of scales
%   sigma = Gaussian smoothening parameter
% Output:
%   blur = Blurred images in the SIFT pyramid in each octave

    num_octaves = numel(pyramid);
    blur = cell(1, num_octaves);
    for oct_idx = 1:num_octaves
        I = cell2mat(pyramid(oct_idx));
        [r,c] = size(I);
        blurs_octave = zeros(r, c, num_scales+3);

        for blur_idx = 1:num_scales+3
            s = blur_idx - 2;
            blurs_octave(:,:,blur_idx) = imgaussfilt(I, (2 ^ (s/num_scales)) *
sigma);
        end

        blur{1, oct_idx} = blurs_octave;
    end

end


function [dog] = createDogOctaves(blur)
%CREATEDOGOCTAVES Summary of this function goes here
%   Detailed explanation goes here
% Inputs:
%   blur = Octaves containing blur images at different scales (r,c,scale)
    num_octaves = numel(blur);
    dog = cell(1, num_octaves);
    for oct_idx = 1:num_octaves
        blurs_octave = cell2mat(blur(1, oct_idx));

        [r,c,s] = size(blurs_octave);
        dog_octave = zeros(r, c, s-1);

        for blur_idx = 2:s
            dog_octave(:, :, blur_idx-1) = ...
                abs(blurs_octave(:, :, blur_idx - 1)...
                - blurs_octave(:, :, blur_idx));
        end

        dog{1, oct_idx} = dog_octave;
    end

end


function [keypts] = createKeypoints(dog, contrast_threshold)
%CREATEKEYPOINTS Summary of this function goes here
```

```
%   Detailed explanation goes here
% Inputs:
%   dog = Octaves containing DoGs {1,octaves -> (r,c,scale)}
%   contrast_threshold = Keypoints below this threshold are rejected
% Output:
%   keypts = Keypoints detected (r,c,scale,oct_idx)

    keypts = {};
    for oct_idx = 1:length(dog)
        oct_dog = dog{oct_idx};

        max_mat = imdilate(oct_dog,ones(3,3,3));
        max_mat(max_mat ~= oct_dog) = 0;
        max_mat(max_mat == oct_dog & max_mat >= contrast_threshold) = 1;
        max_mat(:,:,1) = 0;
        max_mat(:,:,end) = 0;

        [r,c,scale] = ind2sub(size(max_mat),find(max_mat==1));
        keypts = vertcat(keypts,[r,c,scale,oct_idx*ones(length(r),1)]);
    end

    keypts = cell2mat(keypts);

end


function [descriptors, loc] = genDescriptors(keypts, blur)
%GENDESCRIPTORS Summary of this function goes here
%   Detailed explanation goes here
% Inputs:
%   keypts = Keypoints in the format (row, col, scale, octave)
%   blur   = Blur images in octaves (1, octaves -> (row, col, scale))

    num_keypts = size(keypts,1);
    keypt_descriptors = zeros(num_keypts, 128);
    keypt_loc = zeros(num_keypts, 2);

    for kp_idx = 1:num_keypts
        blur_octave = cell2mat(blur(1, keypts(kp_idx,4)));
        % Blurred image to be used to compute gradients
        blur_img = blur_octave(:, :, keypts(kp_idx,3));


        [norm, orientation] = imgradient(blur_img);
        norm_pad = padarray(norm, [7,7], 'pre');
        norm_pad = padarray(norm_pad, [8,8], 'post');
        orientation_pad = padarray(orientation, [7,7], 'pre');
        orientation_pad = padarray(orientation_pad, [8,8], 'post');

        row = keypts(kp_idx,1) + 7; col = keypts(kp_idx,2) + 7;
        norm_w = norm_pad(row - 7 : row + 8, col - 7 : col + 8);
        orientation_w = orientation_pad(row-7:row+8, col-7:col+8);

        sigma_w = 1.5 * 16;
        norm_w = imgaussfilt(norm_w, sigma_w);

        % Calculate 8 bin orientation histogram
        k = 1;
        for y = 1:4:16
            for x = 1:4:16
                wh = weightedhistc(reshape(norm_w(y:y+3,x:x+3),...
                    [1, 16]), reshape(orientation_w(y:y+3,x:x+3),...
```

```
                    [1, 16]), -180:45:180);
                keypt_descriptors(kp_idx, k:k+7) = wh(1,1:8);
                k = k+8;
            end
        end

        keypt_loc(kp_idx,:) = [row,col] .* 2^(keypts(kp_idx,4)-1);
    end
    descriptors = {normalize(keypt_descriptors, 2, 'norm', 2)};
    loc = {keypt_loc};
end


function h = weightedhistc(vals, weights, edges)
% WEIGHTEDHISTC Creates histogram

    if ~isvector(vals) || ~isvector(weights) || length(vals)~=length(weights)
        error('vals and weights must be vectors of the same size');
    end

    Nedge = length(edges);
    h = zeros(size(edges));

    for n = 1:Nedge-1
        ind = find(vals >= edges(n) & vals < edges(n+1));
        if ~isempty(ind)
            h(n) = sum(weights(ind));
        end
    end

    ind = find(vals == edges(end));
    if ~isempty(ind)
        h(Nedge) = sum(weights(ind));
    end

end


function [matches] = corrMatchImages(L, R, scale, win_size, match_lambda)
%CORRMATCHIMAGES Finds matches using intensity based correlation
% Inputs:
%   L = Image 1
%   R = Image 2
%   scale = Scale to downsample images
%   win_size = Window size
%   match_lambda = Correlation Threshold
% Outputs:
%   matches = (rL,cL,rR,cR)

    % Both the images must be of same size.
    if isequal(size(L),size(R))

        harris_patch_size = 9;
        harris_kappa = 0.08;
        nonmaximum_supression_radius = 8;

        imgL = processImage(L,scale); imgR = processImage(R,scale);

        % Get keypoints using Harris detector.
        harris_scores_L = harris(imgL, harris_patch_size, harris_kappa);
        keypoints_L = selectKeypoints(...
            harris_scores_L, nonmaximum_supression_radius)';
```

```
        harris_scores_R = harris(imgR, harris_patch_size, harris_kappa);
        keypoints_R = selectKeypoints(...
            harris_scores_R, nonmaximum_supression_radius)';

        % Generate descriptors for each keypoint.
        descriptors_L = describeKeypoints(imgL, keypoints_L, win_size);
        descriptors_R = describeKeypoints(imgR, keypoints_R, win_size);

        % Match keypoints based on correlation between the descriptors.
        matches = corrMatchDescriptors(descriptors_L, descriptors_R,...
            keypoints_L, keypoints_R, match_lambda);

    else

        disp('Images must of same size');
        matches = [];

    end

end


function scores = harris(img, patch_size, kappa)

    % Gradients according to the sobel filter
    [Ix,Iy] = imgradientxy(img);
    Ixx = Ix.^2;    Iyy = Iy.^2;    Ixy = Ix.*Iy;

    % Sum of gradients in a given patch
    sIxx = imfilter(Ixx, ones(patch_size));
    sIyy = imfilter(Iyy, ones(patch_size));
    sIxy = imfilter(Ixy, ones(patch_size));

    scores = sIxx.*sIyy - sIxy*2 - kappa * ((sIxx + sIyy).^2);

    scores(scores<0) = 0;
end


function keypoints = selectKeypoints(scores, r)
% Selects the best scores as keypoints and performs non-maximum
% supression of a (2r + 1)*(2r + 1) box around the current maximum.

    [i,j] = find(scores > 0);
    indices = [i,j];

    scorePadded = zeros(size(scores,1) + 2*r, size(scores,2) + 2*r);
    scorePadded(r+1:size(scores,1) + r, r+1:size(scores,2) + r) = scores;

    % Suppress neighbouring maximum values
    for i = 1:length(indices)
        idx = indices(i,:); u = idx(1) + r; v = idx(2) + r;
        s = scores(idx(1),idx(2));

        w = scorePadded(u-r:u+r, v-r:v+r);
        m = max(max(w));
        f = zeros(size(w)); f(r+1,r+1) = 1;

        if s == m
            scorePadded(u-r:u+r, v-r:v+r) = w.*f;
        elseif s <= m
            scorePadded(u,v) = 0;
```

```
        end
    end

    maxScores = scorePadded(r+1:size(scores,1) + r, r+1:size(scores,2) + r);
    [i,j] = find(maxScores > 0);
    A = [i,j,maxScores(maxScores > 0)];
    A = sortrows(A, 3);
    A = fliplr(A')';

    if length(A) > 200
        A = A(1:200,:);
    end

    keypoints = A(:,1:2)';
end




function descriptors = describeKeypoints(img, keypoints, win_size)
% keypoints = (N,2)
    descriptors = zeros(length(keypoints),win_size^2);
    buff = (win_size - 1) / 2;

    img_pad = padarray(img,[buff,buff],'both');

    for i = 1:length(keypoints)
        idx = keypoints(i,:) + [buff,buff];
        w = img_pad(idx(1)-buff:idx(1)+buff,...
            idx(2)-buff:idx(2)+buff);
        descriptors(i,:) = w(:)';
    end

end


function [matches] = corrMatchDescriptors(...
    descriptors_L, descriptors_R, kpt_L, kpt_R, match_lambda)
%CORRMATCHDESCRIPTORS Summary of this function goes here
%   Detailed explanation goes here
% Inputs:
%   descriptors_L = Descriptors in left image(num_kpts,dimensions)
%   descriptors_R = Descriptors in right image(num_kpts,dimensions)
%   kpt_L = Keypoints in left image
%   kpt_R = Keypoints in right image
%   match_lambda = Correlation threshold for matching
% Output:
%   matches = (rL,cL,rR,cR)

    matches = {};

    for desc_idx = 1:size(descriptors_L,1)
        L_vect = descriptors_L(desc_idx,:);

        r = kpt_L(desc_idx,1);
        rows = kpt_R(:,1);
        [idx_1,~]= find(rows <= r + 5);
        [idx_2,~]= find(rows >= r - 5);
        idx = intersect(idx_1,idx_2);

        R_vect = descriptors_R(idx,:);
```

```matlab
        correlation = sum(L_vect .* R_vect, 2) ./ ...
            (sqrt(sum(L_vect .^ 2, 2)) * sqrt(sum(R_vect .^ 2, 2)));

        [max_corr, corr_idx] = max(correlation);
        if ~isempty(max_corr)
            if ((max_corr > match_lambda) && (pdist2(kpt_L(desc_idx,:),...
                    kpt_R(idx(corr_idx),:)) < 50))
                matches = vertcat(matches,...
                    [kpt_L(desc_idx,:), kpt_R(idx(corr_idx),:)]);
            end
        end

    end

    matches = cell2mat(matches);

end
```