

Assignment 2:
Panorama Stitching
COMP 408

Ahmet UYSAL

November 2, 2018

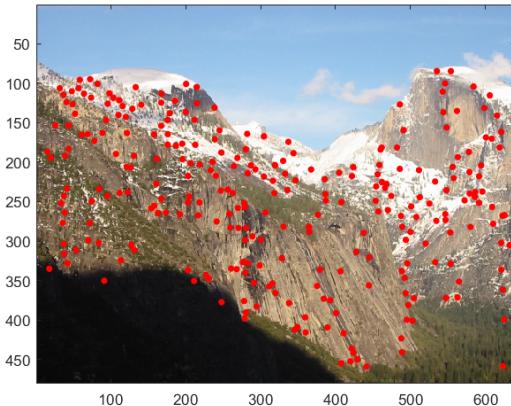
Instructor: Yücel YEMEZ

1 Feature Detection

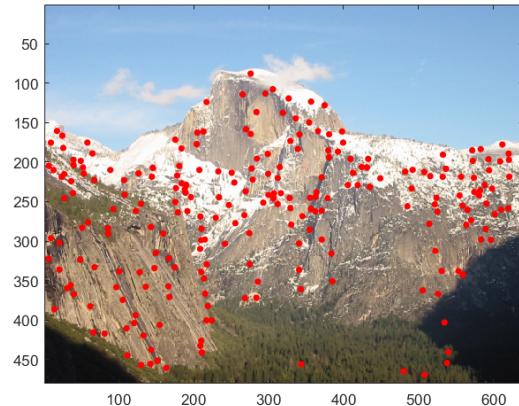
1.1 Algorithm

The code for this part was given in the assignment. Basically, differences of Gaussian filters are applied to image in different scales to detect set of key points, each associated with a best scale. Later, these key points will be used for creating SIFT descriptors.

1.2 Sample Outputs

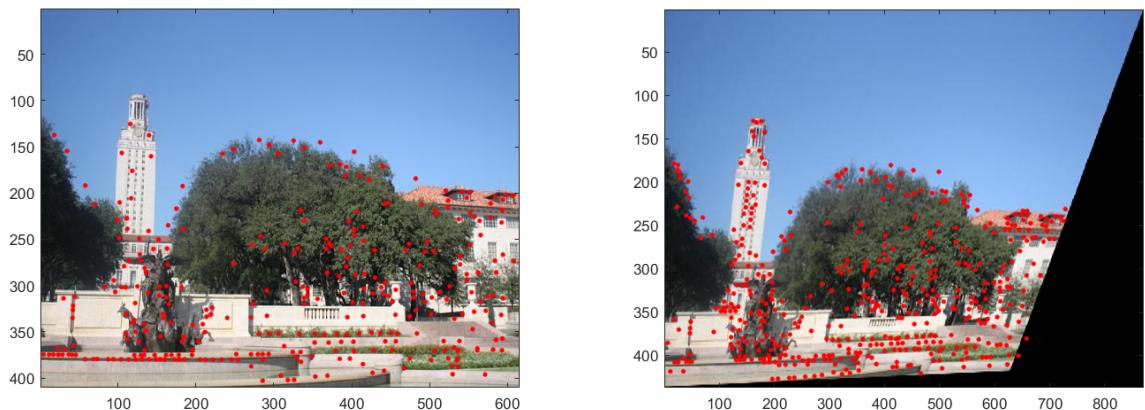


(a) Key points for given image yosemite1.



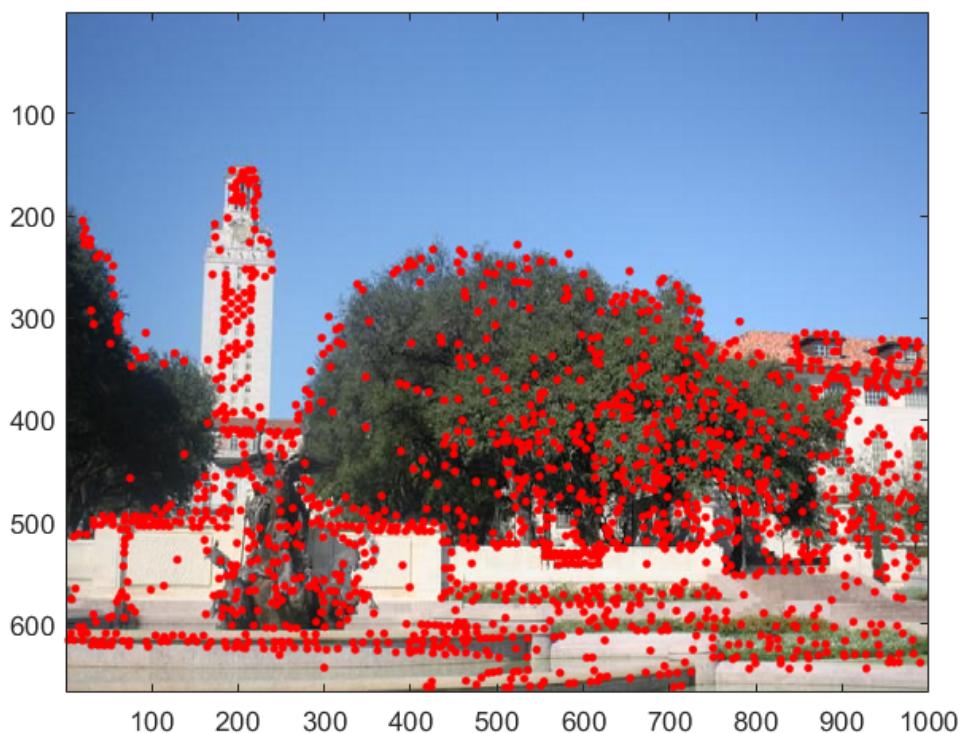
(b) Key points for given image yosemite2.

Figure 1: Visualization of key points extracted.



(a) Key points for given original image of UT Tower.

(b) Key points for disrupted image.



(c) Key points for scaled up image.

Figure 2: Visualization of key points extracted for different scales and orientations.

2 Feature Description

2.1 Algorithm

For the second part we need to create SIFT descriptors from our key points. For this purpose, we will implement the algorithm from David G. Lowe's paper "Distinctive Image Features from Scale-Invariant Keypoints". In summary, for each key point we detected from part one, we need to

1. Take a NxN window around the key point, and normalize it to 16x16, where N is proportional to scale of key point.
2. Compute $\nabla f(n_1, n_2)$ for each pixel in the window
3. Weight down the $|\nabla f(n_1, n_2)|$ of each pixel by a Gaussian function
4. For each pixel, compute edge orientation, i.e, $\angle \nabla f(n_1, n_2) - \frac{\pi}{2}$
5. Create histogram of edge orientations within the window with 8 bins
6. Calculate the dominant direction using the histogram
7. Divide 16x16 window to four 4x4 cells
8. Compute a normalized orientation histogram for each cell (normalization is done by subtracting the dominant angle)

to obtain a scale and orientation invariant descriptor. After these steps we will get a 128 dimensional ($\#cells \times \#bins$) descriptor for each key point. The implementation of this algorithm and helper functions used are given below.

2.2 MATLAB Implementation

```
1 function descriptors = SIFTDescriptor(pyramid, keyPt, keyPtScale)
2 % INPUT:
3 %   pyramid: Image pyramid. pyramid{i} is a rescaled version of the original
4 %   image, in grayscale double format
5 %   keyPt: N * 2 matrix, each row is a key point pixel location in pyramid{
6 %     round(scale)}. So pyramid{round(scale)}(y,x) is the center of the keypoint
7 %   scale: N * 1 matrix, each entry holds the index in the Gaussian pyramid
8 %   for the keypoint. Earlier code interpolates things, so this is a double,
9 %   but we'll just round it to an integer.
10 % OUTPUT:
11 %   descriptors: N * 128 matrix, each row is a feature descriptor
12
13 %% Initializations
14 % Number of keypoints that were found by the DoG blob detector
15 N = size(keyPt, 1);
16 % For each keypoint we will extract a region of size patch_size x patch_size
17 % centered at the keypoint.
18 patch_size = 16;
19 % The patch extracted around each keypoint will be divided into a grid of
20 % grid_size x grid_size.
21 grid_size = 4;
22 % Each histogram covers an area "pixelsPerCell" wide and tall
23 pixelsPerCell= patch_size / grid_size;
24 % The number of orientation bins per cell
25 num_bins = 8;
```

```

20 % Initialize descriptors to zero
21 descriptors = zeros(N, grid_size*grid_size*num_bins);
22
23 grad_mag = cell(length(pyramid),1);
24 grad_theta = cell(length(pyramid),1);
25 % Determine the gradient angles and magnitude of all images in the pyramid.
26 [grad_mag,grad_theta] = ComputeGradient(pyramid);
27
28 % Iterate over all keypoints
29 for i = 1 : N
30     scale = round(keyPtScale(i));
31     magnitudes = grad_mag{scale};
32     thetas = grad_theta{scale};
33     % Extract a patch of magnitudes and directions (16x16) around the keypoint
34     [patch_mag,patch_theta] = Extract_Patch(keyPt(i,:),patch_size,magnitudes,
35         thetas);
36     if( isempty(patch_mag))
37         continue;
38     end
39     % Normalize the gradient directions relative to the dominant gradient
40     % direction
41     patch_theta = Normalize_Orientation(patch_mag, patch_theta);
42     % Weight the gradient magnitudes using a Gaussian function
43     patch_mag = patch_mag .* fspecial('gaussian', patch_size, patch_size / 2);
44     % ComputeSIFTDescriptors for the patch, explanation is in the function
45     % implementation
46     feature = ComputeSIFTDescriptor(patch_mag, patch_theta, grid_size,
47         pixelsPerCell, num_bins);
48     % Add the feature vector we just computed to our matrix of SIFT
49     % descriptors.
50     descriptors(i, :) = feature;
51 end
52 % Normalize the descriptors.
53 descriptors = NormalizeDescriptors(descriptors);
54 end

```

Code Snippet 1: General structure of the code for creating SIFT Descriptors.

```

1 function [grad_mag,grad_theta] = ComputeGradient(pyramid)
2 % Input:
3 % pyramid: all the pyramid images in a cell array
4 % Output:
5 % grad_mag, grad_mag:
6 %     Two cell arrays of the same shape where grad_mag{i} and grad_theta{i}
7 %     give the magnitude and direction of the i-th pyramid image.
8 % Gradient_angles ranges from 0 to 2*pi.
9 grad_theta = cell(length(pyramid),1);
10 grad_mag = cell(length(pyramid),1);
11 %Do this for all pyramid images
12 for scale = 1:length(pyramid)
13     currentImage = pyramid{scale};
14     grad_mag{scale} = zeros(size(currentImage));
15     grad_theta{scale} = zeros(size(currentImage));
16     % Discrete approximation of gradients
17     img_dx = filter2([-1 0 1], currentImage);
18
19 end

```

```

16 img_dy = filter2([-1; 0; 1], currentImage);
17
18 grad_mag{scale} = sqrt(img_dx.*img_dx + img_dy.*img_dy);
19 grad_theta{scale} = atan2(img_dy, img_dx) - deg2rad(90);
20 % atan2 gives angles from -pi to pi. To make the histogram code easier, we
21 % will change that to 0 to 2*pi.
22 grad_theta{scale} = mod(grad_theta{scale}, 2*pi);
23 end
end

```

Code Snippet 2: My implementation of ComputeGradient function.

```

1 function descriptor = ComputeSIFTDescriptor(patch_mag, patch_theta, grid_size,
2     pixelsPerCell, num_bins)
% INPUT:
3 % patch_mag, patch_theta:
4 %     Two arrays of the same shape where patch_mag(i) and patch_theta(i)
5 %     give the magnitude and direction of the gradient for the ith point.
6 %     gradient_angles ranges from 0 to 2*pi
% OUTPUT:
7 % descriptor: A 128 length descriptor calculated from 8-bin histograms of
8 % the 16 cells in the grid
9
10 % initialize descriptor to size 0
11 descriptor = [];
12 % iterate over each pixels
13 for x = 1 : grid_size
14     % divide grid to cells
15     cell_x_start = 1 + (x-1)*pixelsPerCell;
16     cell_x_end = x * pixelsPerCell;
17     for y = 1:grid_size
18         cell_y_start = 1 + (y-1)*pixelsPerCell;
19         cell_y_end = y * pixelsPerCell;
20         % get cells from the grid
21         mag_cell = patch_mag(cell_x_start:cell_x_end, ...
22             cell_y_start:cell_y_end);
23         theta_cell = patch_theta(cell_x_start:cell_x_end, ...
24             cell_y_start:cell_y_end);
25         % get the orientation histogram by using ComputeGradientHistogram
26         cell_gradient_histogram = ComputeGradientHistogram(num_bins, mag_cell,
27             theta_cell);
28         % add descriptor of the cell to list of descriptors
29         descriptor = [descriptor, cell_gradient_histogram];
30     end
31 end
32 end

```

Code Snippet 3: My implementation for ComputeSIFTDescriptor function.

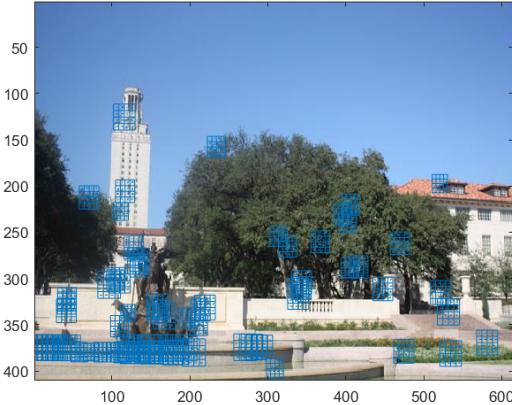
```

1 function [ histogram , angles ] = ComputeGradientHistogram( num_bins ,
    gradient_magnitudes , gradient_angles )
2 % INPUT
3 % num_bins: The number of bins to which points should be assigned .
4 % gradient_magnitudes , gradient_angles :
5 %           Two arrays of the same shape where gradient_magnitudes(i) and
6 %           gradient_angles(i) give the magnitude and direction of the gradient for
7 %           the ith point. gradient_angles ranges from 0 to 2*pi
8 %
9 % OUTPUT
10 % histogram: A 1 x num_bins array containing the gradient histogram . Entry 1
11 %           is the sum of entries in gradient_magnitudes whose corresponding
12 %           gradient_angles lie between 0 and angle_step . Similarly , entry 2 is for
13 %           angles between angle_step and 2*angle_step . Angle_step is calculated as 2*
14 %           pi/num_bins .
15 % angles: A 1 x num_bins array which holds the histogram bin lower bounds .
16 %           histogram(i) contains the sum of the gradient magnitudes of all points
17 %           whose gradient directions fall in the range [ angles(i) , angles(i + 1) )
18 %
19 % angle_step = 2 * pi / num_bins ;
20 % angles = 0 : angle_step : (2*pi-angle_step) ;
21 % histogram = zeros(1 , num_bins ) ;
22 % get sizes to use in loops
23 [ size_x , size_y ] = size(gradient_angles) ;
24 % iterate over all angles
25 for gradient_index_x = 1:size_x
26     for gradient_index_y = 1:size_y
27         grad_angle = gradient_angles(gradient_index_x , gradient_index_y );
28         % start checking from biggest bin
29         for angle_index = length(angles):-1:1
30             if( grad_angle >= angles(angle_index) )
31                 % update the bin value
32                 histogram(angle_index) = histogram(angle_index) ...
33                     + gradient_magnitudes(gradient_index_x , gradient_index_y );
34             % we are done if our angle is bigger than or equal to bin
35             % lower bound
36             break
37         end
38     end
39 end
40 end
41 end
42 end

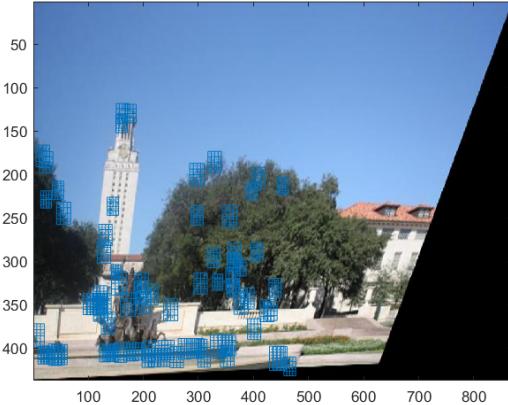
```

Code Snippet 4: My implementation of ComputeGradientHistogram function.

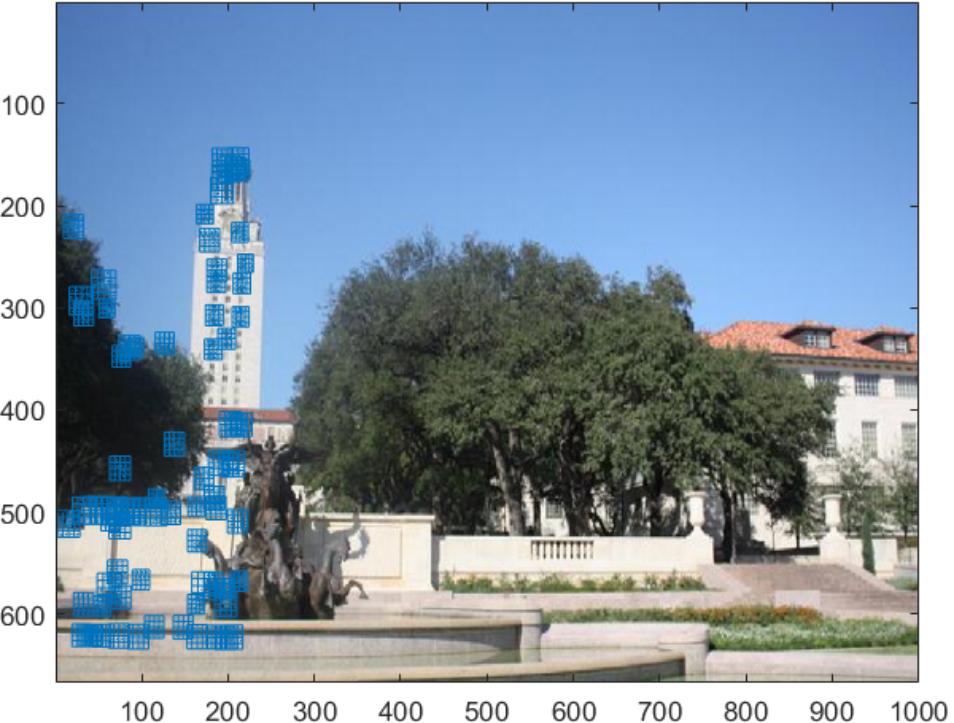
2.3 Sample Outputs



(a) SIFT descriptors for given original image of UT Tower.

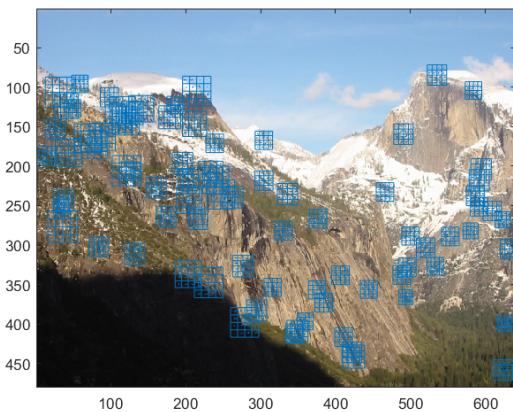


(b) SIFT descriptors for disrupted image.

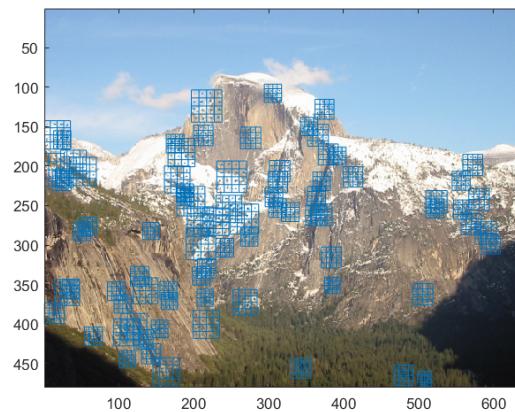


(c) SIFT descriptors for scaled up image.

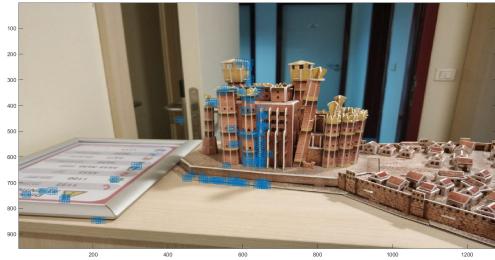
Figure 3: Visualization of SIFT descriptors extracted for different scales and orientations.



(a) SIFT Descriptors of given image yosemite1.



(b) SIFT Descriptors of given image yosemite2.



(c) SIFT Descriptors of left part of King's Landing puzzle.



(d) SIFT Descriptors of right part of the puzzle.

Figure 4: Visualization of SIFT Descriptors extracted.

3 Feature Matching

3.1 Algorithm

The simple solution for feature matching is defining a distance function, such as Euclidean Distance, and iterate over all SIFT descriptors of second image to find the match for a descriptor in the first image. However, this simple solution is not the best, it can give good scores to bad matches. To decrease the number of false matches, we can use the ratio distance, i.e, the ratio of Euclidean distance between f_1 and f_2 to Euclidean distance between f_1 and f'_2 , where f_1 is feature in first image, f_2 is the best scored match to f_1 according to distance function, and f'_2 is the second best match. We can filter false matches by eliminating the matches with ratios lower than some threshold value.

3.2 MATLAB Implementation

```
1 function match = SIFTSimpleMatcher(descriptor1, descriptor2, thresh)
2 % INPUT:
3 %   descriptor1: N1 * 128 matrix, each row is a SIFT descriptor.
4 %   descriptor2: N2 * 128 matrix, each row is a SIFT descriptor.
5 %   thresh: a given threshold of ratio. Typically 0.7
6 % OUTPUT:
7 %   Match: N * 2 matrix, each row is a match. For example, Match(k, :) = [i, j]
8 %         means i-th descriptor in descriptor1 is matched to j-th descriptor in
9 %         descriptor2.
10 if ~exist('thresh', 'var'),
11     thresh = 0.7;
12 end
13 match = [];
14 %Get the number of descriptors for iterating
15 [ld1, ~] = size(descriptor1);
16 [ld2, ~] = size(descriptor2);
17 for i = 1:ld1
18     sift_desc_1 = descriptor1(i, :);
19     %initialize distances with infinity
20     smallest = inf;
21     second_smallest = inf;
22     %initialize smallest index with zero
23     smallest_index = 0;
24     % this implementation assumes there are at least two descriptors in
25     % second array, smallest variables are wrong until end of first two
26     % iterations
27     for j = 1: ld2
28         sift_desc_2 = descriptor2(j, :);
29         distances = sift_desc_1 - sift_desc_2;
30         % dot product of row vector of distances with its transpose is
31         % equivalent with sum of squares of distances
32         euclidian_dist = sqrt(distances * distances');
33         % update values if necessary
34         if (euclidian_dist < second_smallest)
35             second_smallest = euclidian_dist;
36         end
37         if (euclidian_dist < smallest)
38             second_smallest = smallest;
39             smallest = euclidian_dist;
40             smallest_index = j;
```

```

36      end
37  end
38 % check ratio for threshold
39 if (smallest < second_smallest * thresh)
40     % add indexes of pair to match array if they pass the condition
41     match = [match; [i, smallest_index]];
42 end
43 end
44 end

```

Code Snippet 5: My implementation of SIFTSimpleMatcher function.

3.3 Sample Outputs

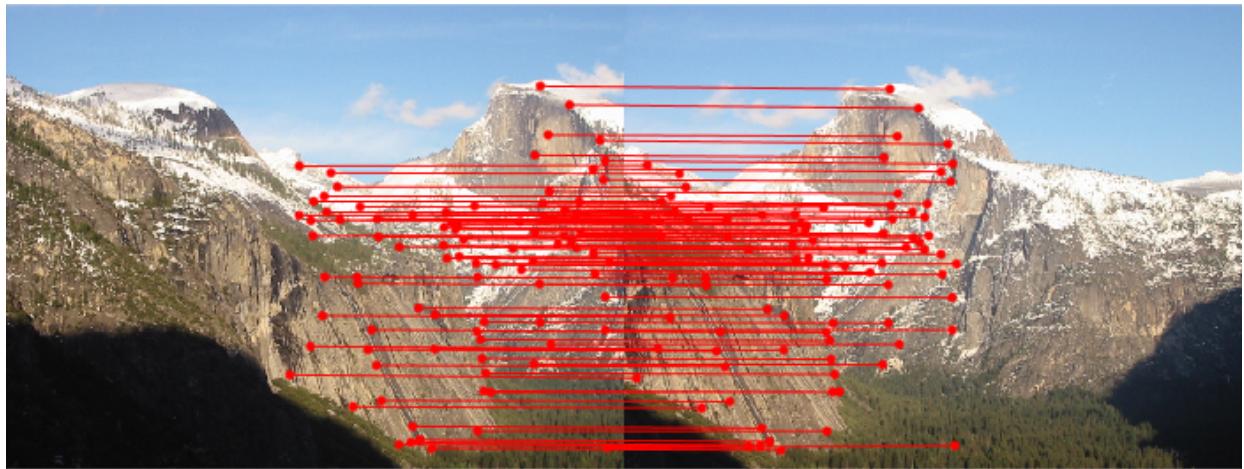


Figure 5: Visualization of matched key points between parts of Mount Yosemite.

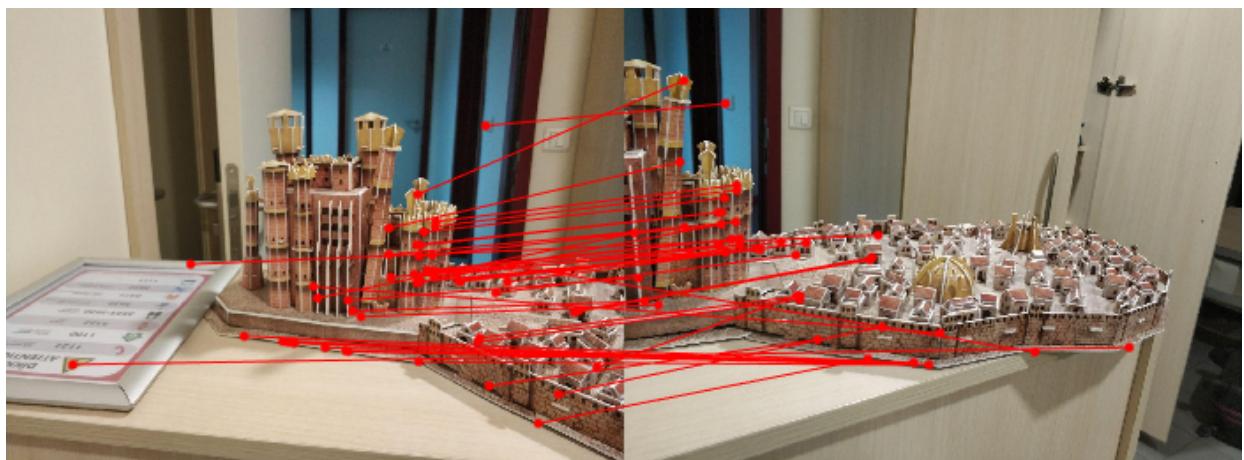


Figure 6: Visualization of matched key points between parts of King's Landing Puzzle.

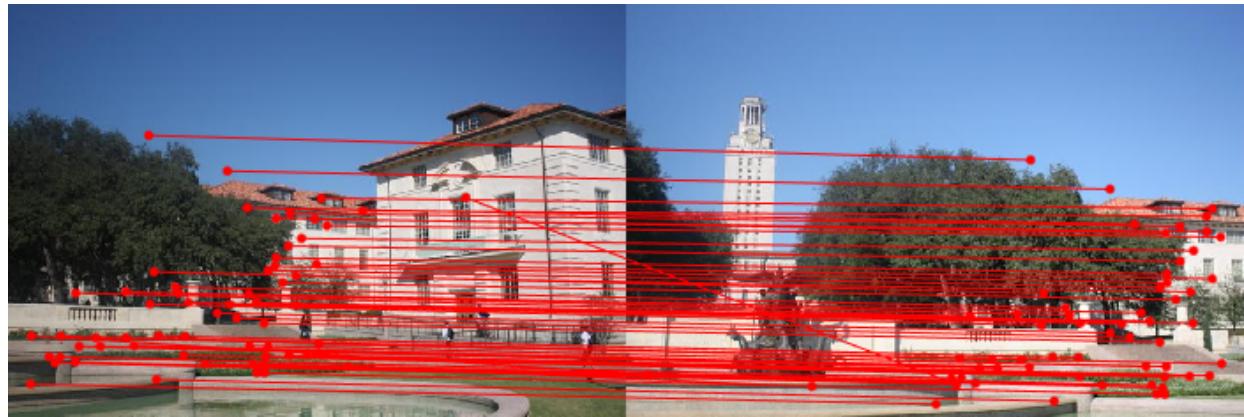


Figure 7: Visualization of matched key points between parts of UT Tower.

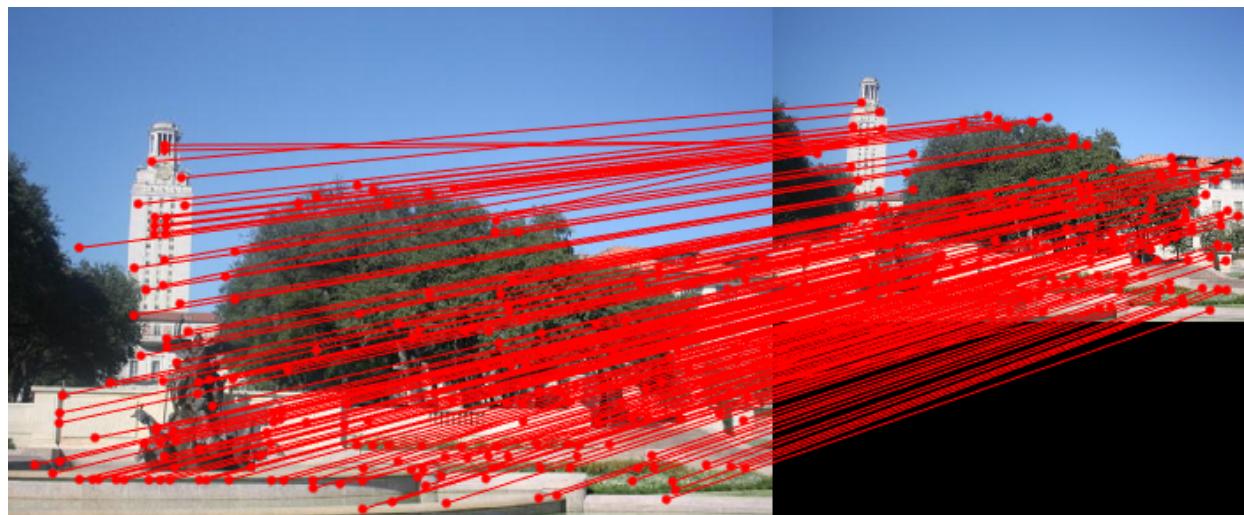


Figure 8: Visualization of matched key points between original and scaled UT Tower images.

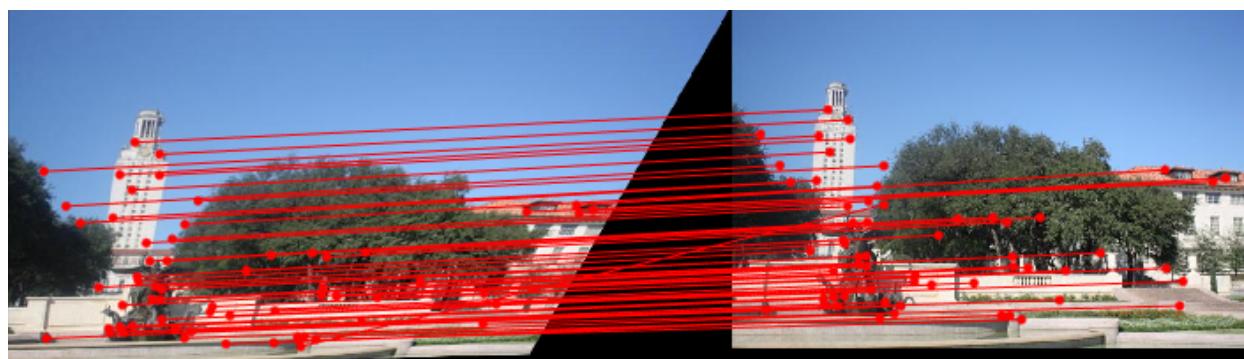


Figure 9: Visualization of matched key points between original and disrupted UT Tower images.

As seen in the samples, SIFT Features are scale and rotation invariant.

4 Homography Estimation

4.1 Algorithm

Homography is a mapping between two projection planes with the same center of projection, it can be represented as 3x3 matrix in homogeneous coordinates. A pixel in projection plane can be mapped to another projection plane with formula

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

So, we can estimate H by solving equations of the form: $wp_i = Hp_i$ using the matching pairs that we detected from previous parts. We will use RANSAC (Random Sample Consensus) algorithm to calculate H probabilistically. In RANSAC, we choose random n samples that it required to estimate H, and we will classify all the matches as inliers and outliers by thresholding them according to their errors. If the number of inliers exceed the threshold value, we will recalculate H using only the inliers. We repeat this procedure k times and keep the transformation with the minimum error.

4.2 MATLAB Implementation

```
1 function H_best = RANSACFit(p1, p2, match, seedSampleSize, maxInlierError,
2     goodFitThresh)
3 % Input:
4 %   p1: N1 * 2 matrix, each row is a point
5 %   p2: N2 * 2 matrix, each row is a point
6 %   match: M * 2 matrix, each row represents a match [index of p1, index of p2]
7 %   seedSampleSize: The number of randomly-chosen seed points that we'll use
8 %                   to fit
9 %   maxInlierError: A match not in the seed set is considered an inlier if
10 %                   its error is less than maxInlierError. Error is
11 %                   measured as sum of Euclidean distance between transformed
12 %                   point1 and point2. You need to implement the
13 %                   ComputeCost function.
14 %   goodFitThresh: The threshold for deciding whether or not a model is
15 %                   good; for a model to be good, at least goodFitThresh
16 %                   non-seed points must be declared inliers.
17 %
18 % Output:
19 %   H: a robust estimation of affine transformation from p1 to p2
20
21 %% Initializations
22 N = size(match, 1);
23 if N<3
24     error('not enough matches to produce a transformation matrix')
25 end
26 if ~exist('seedSampleSize', 'var'),
27     seedSampleSize = ceil(0.2 * N);
28 end
29 if ~exist('maxInlierError', 'var'),
30     maxInlierError = 30;
31 end
32 if ~exist('goodFitThresh', 'var'),
33     goodFitThresh = floor(0.7 * N);
34 end
```

```

32
33 %probability that all samples fail
34 p_fail = 0.001;
35 %fraction of inliers
36 w = 0.7;
37 %seed sample size
38 n = seedSampleSize;
39 % Determine the number of iterations required to ensure a good fit
40 maxIter = ceil(log(p_fail)/log(1 - w^n));
41 % initialize H with identity and error with infinity
42 H_best = eye(3);
43 min_error = inf;
44 % iterate maxIter times
45 for i = 1 : maxIter
46     % Randomly select a seed group
47     idx = randperm(size(match, 1));
48     seed_group = match(idx(1:seedSampleSize), :);
49     % Select remaining as the non-seed group
50     non_seed_group = match(idx(seedSampleSize+1:end), :);
51     % Use seed_group to compute the transformation matrix.
52     H = ComputeAffineMatrix( p1(seed_group(:, 1), :) , p2(seed_group(:, 2), :)
53         );
54     % Use non_seed_group to compute error from euclidian distance
55     err = ComputeError(H, p1(non_seed_group(:, 1), :) , p2(non_seed_group(:, 2)
56         ,:));
57     % Select the points as inliers which have error less than maxInlierError
58     inliers = [];
59     for index = 1:length(non_seed_group)
60         if (err(index) < maxInlierError)
61             inliers = [inliers; non_seed_group(index, :)];
62         end
63     end
64     % recalculate H using only inliers if number of inliers is bigger than
65     % threshold
66     number_of_inliers = size(inliers,1) + size(seed_group,1);
67     if( number_of_inliers > goodFitThresh )
68         all_inliers = [seed_group ; inliers];
69         H = ComputeAffineMatrix( p1(all_inliers(:, 1), :) , p2(all_inliers(:, 2),
70             ,:) );
71         err = ComputeError(H, p1(all_inliers(:, 1), :) , p2(all_inliers(:, 2)
72             ,:));
73         % I used sum of squares of errors to evaluate H
74         sum_err_squares = err' * err;
75         if(sum_err_squares < min_error)
76             min_error = sum_err_squares;
77             H_best = H;
78         end
79     end
80 end
81 if sum(sum((H_best - eye(3)).^2)) == 0,
82     disp('No RANSAC fit was found.')
83 end
84 end

```

Code Snippet 6: My implementation of RANSACFit function.

```

1 function H = ComputeAffineMatrix( Pt1 , Pt2 )
2
3 N = size(Pt1,1);
4 if size(Pt1, 1) ~= size(Pt2, 1),
5     error('Dimensions unmatched.');
6 elseif N<3
7     error('At least 3 points are required.');
8 end
9 % Convert the input points to homogeneous coordinates.
10 P1 = [Pt1'; ones(1,N)];
11 P2 = [Pt2'; ones(1,N)];
12 %  $P1' * H' = P2' \rightarrow H' = P1' \setminus P2'$ 
13 H_transpose = P1' \ P2';
14 H = H_transpose';
15 % Sometimes numerical issues cause least-squares to produce a bottom
16 % row which is not exactly [0 0 1], which confuses some of the later
17 % code. So we'll ensure the bottom row is exactly [0 0 1].
18 H(3,:) = [0 0 1];
19 end

```

Code Snippet 7: My implementation of ComputeAffineMatrix function.

```

1 function dists = ComputeError(H, pt1, pt2)
% Input:
3 % H: 3 x 3 transformation matrix where H * [x; y; 1] transforms the point
4 % (x, y) from the coordinate system of pt1 to the coordinate system of pt2.
5 % pt1: N1 x 2 matrix where each ROW is a data point [x_i, y_i]
6 % pt2: N2 x 2 matrix where each ROW is a data point [x_i, y_i]
7 % match: M x 2 matrix, each row represents a match [index of pt1, index of
8 % pt2]
% Output:
9 % dists: An M x 1 vector where dists(i) is the error of fitting the i-th
10 % match to the given transformation matrix.
11 % Error is measured as the Euclidean distance
12 dists = zeros(size(pt1,1),1);
13 % Convert the input points to homogeneous coordinates.
14 pt1_homogeneous = [pt1' ; ones(1, length(pt1))];
15 pt2_homogeneous = [pt2' ; ones(1, length(pt2))];
16 % transform p1 using H
17 pt1_transformed = (H * pt1_homogeneous);
18
19 distances = pt1_transformed - pt2_homogeneous;
20 distances_x = distances(1, :);
21 distances_y = distances(2, :);
22 % distance_z is always all zero
23 % Calculate the Euclidean distances
24 % transpose the vector to satisfy documentation (return M x 1 matrix).
25 dists = sqrt(distances_x .* distances_x + distances_y .* distances_y)';
26 if size(dists,1) ~= size(pt1,1) || size(dists,2) ~= 1
27     error('wrong format');
28 end
29 end

```

Code Snippet 8: My implementation of ComputeError function.

5 Image Stitching

We implemented all the necessary algorithms. We can stitch images by creating SIFT descriptors, performing feature matching, determining the transformation matrix H and finally using H to transform first image on top of second image. Code for this part is already given.

5.1 Sample Outputs



Figure 10: Stitched image of Mount Yosemite.

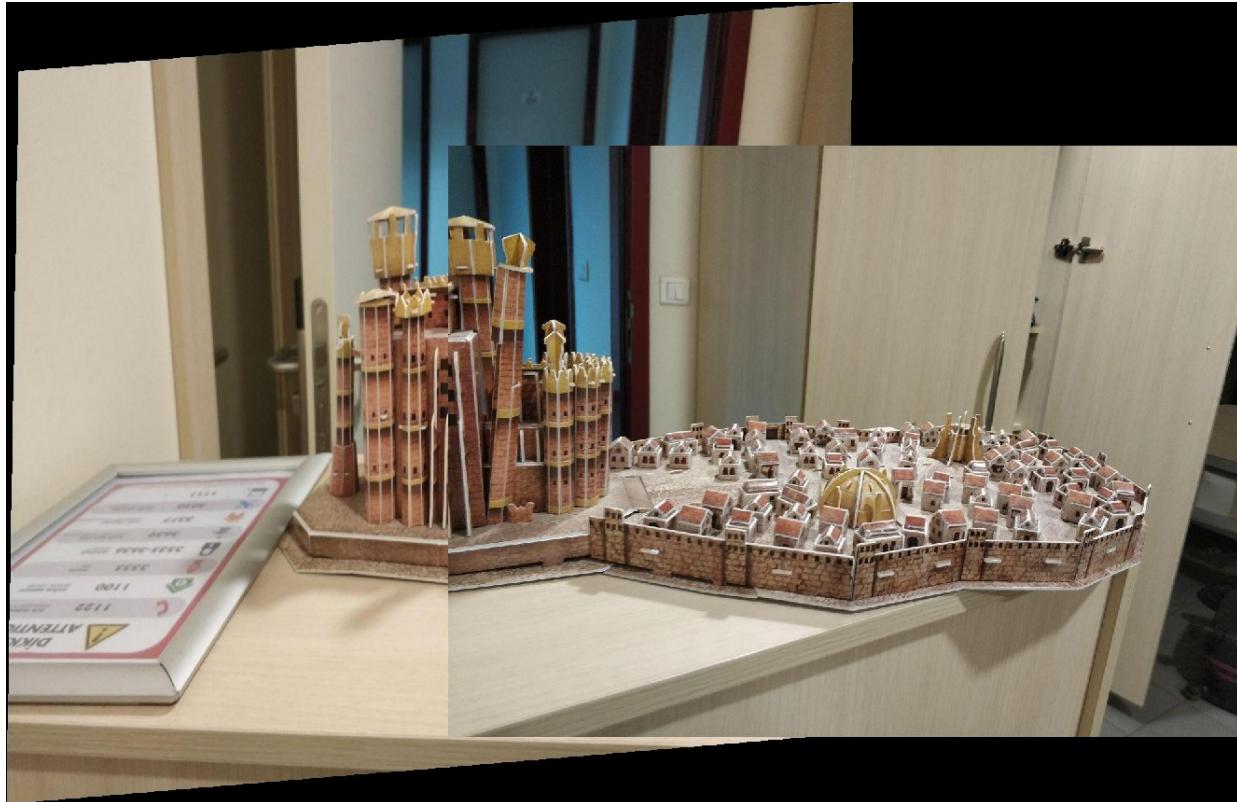


Figure 11: Stitched image of King's Landing Puzzle.



Figure 12: Stitched image of UT Tower.

6 Stitching Multiple Images

6.1 Algorithm

We can simply stitch more than two images by first stitching two consequent images and continue until we are done. We can calculate homography matrices one for transformation in consequent images and use these matrices to transform all of the images to the same project plane.

6.2 MATLAB Implementation

```
1 function T = makeTransformToReferenceFrame(i_To_iPlusOne_Transform ,  
2   currentFrameIndex , refFrameIndex)  
3 %makeTransformToReferenceFrame  
4 % INPUT:  
5 %   i_To_iPlusOne_Transform: this is a cell array where  
6 %   i_To_iPlusOne_Transform{i} contains the 3x3 homogeneous transformation  
7 %   matrix that transforms a point in frame i to the corresponding point in  
8 %   frame i+1  
9 %   currentFrameIndex: index of the current coordinate frame in  
10 %   i_To_iPlusOne_Transform  
11 %   refFrameIndex: index of the reference coordinate frame  
12 % OUTPUT:  
13 %   T: A 3x3 homogeneous transformation matrix that would convert a point  
14 %   in the current frame into the corresponding point in the reference  
15 %   frame. For example, if the current frame is 2 and the reference frame  
16 %   is 3, then T = i_To_iPlusOne_Transform{2}. If the current frame and  
17 %   reference frame are not adjacent , T will need to be calculated.  
18  
19 T = eye(3);  
20 % if target index is bigger than current index , iteratively multiply  
21 % transformation matrix with T from left until new matrix until goal is  
22 % reached  
23 if (currentFrameIndex < refFrameIndex)  
24   for i = currentFrameIndex:refFrameIndex - 1  
25     T = i_To_iPlusOne_Transform{i} * T;  
26   end  
27 % if target index is lower than current index , first find the inverse of  
28 % transformation matrix and multiply it with T from left until goal is  
29 % reached  
30 elseif (currentFrameIndex > refFrameIndex)  
31   for i = currentFrameIndex - 1: -1:refFrameIndex  
32     T = pinv(i_To_iPlusOne_Transform{i}) * T;  
33   end  
34 end  
35 end
```

Code Snippet 9: My implementation of makeTransformToReferenceFrame function.

6.3 Sample Outputs

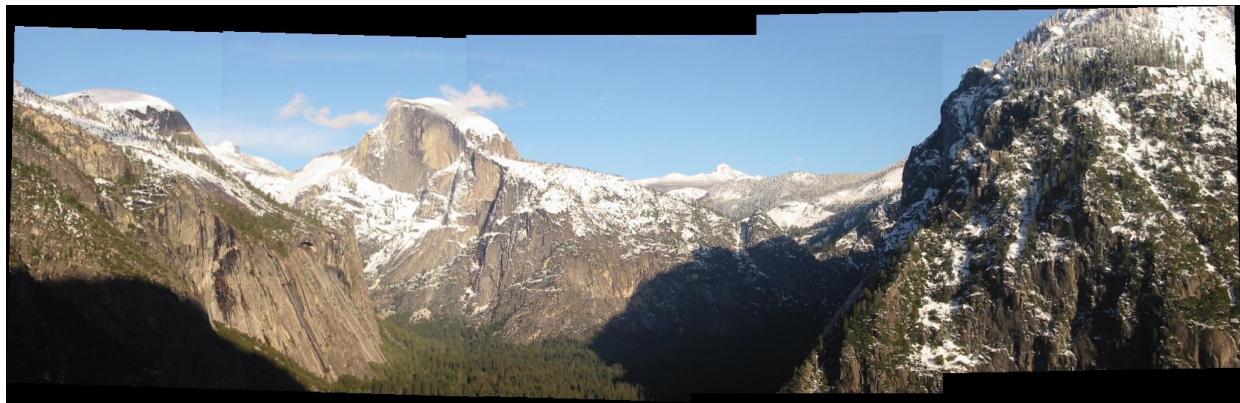


Figure 13: Stitched image of Mount Yosemite from multiple images.

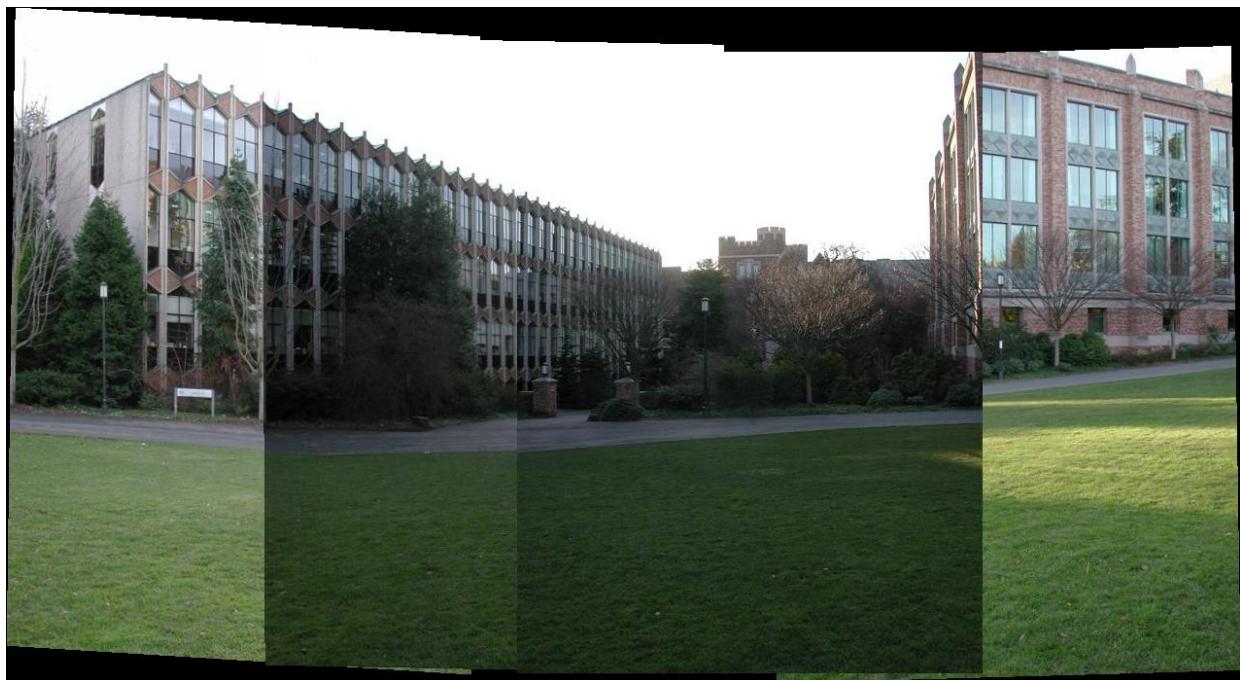


Figure 14: Stitched image of campus from multiple images..



Figure 15: Stitched image of yard from multiple images.

7 Principal Component Analysis