Assignment 2:
Panorama Stitching
COMP 408

Ahmet Uysal

November 2, 2018

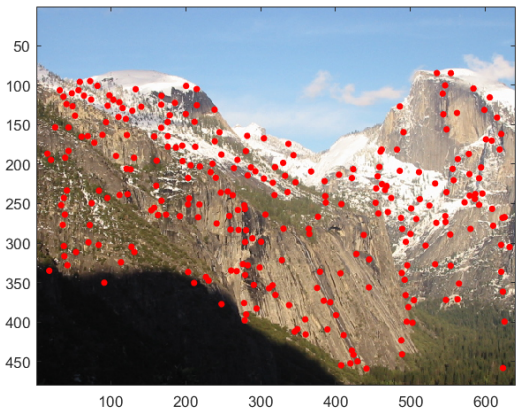
Instructor: Yücel Yemez

1 Feature Detection

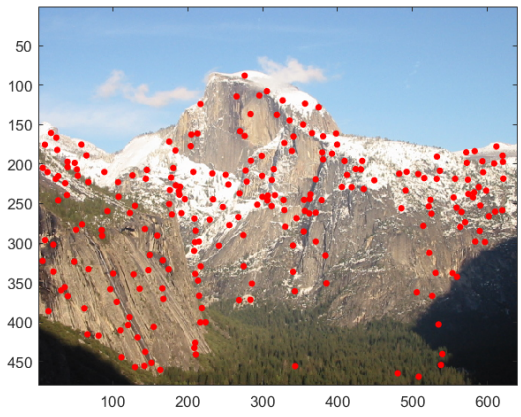
1.1 Algorithm

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

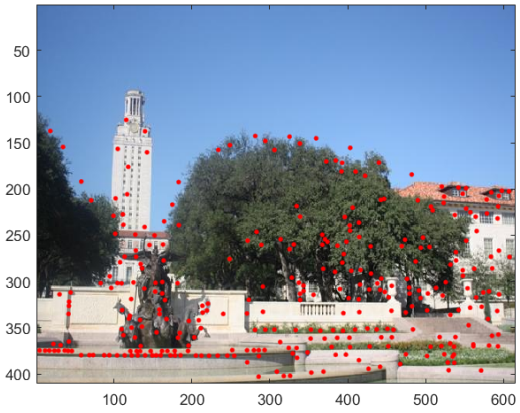


(a) Key points for given image yosemite1.

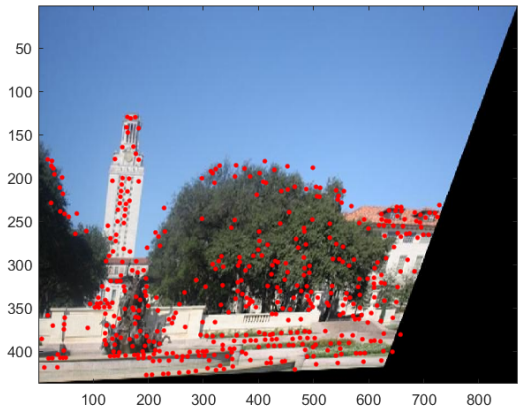


(b) Key points for given image yosemite2.

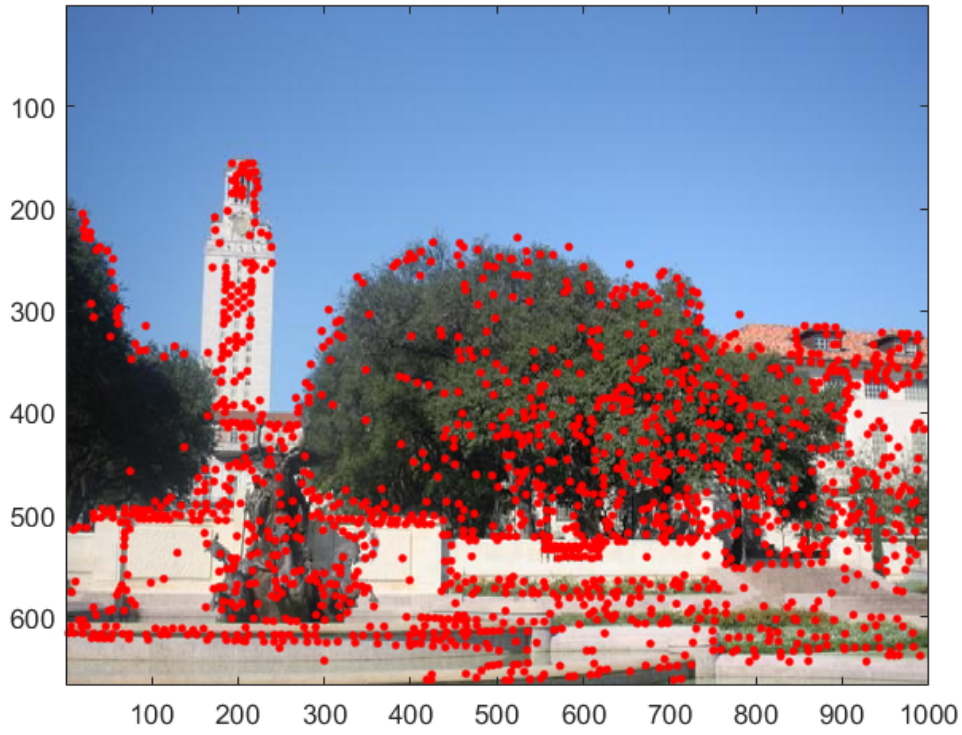
Figure 1: Visualization of key points extracted.



(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.

As seen from the extracted features, we got different features when the scale of the image has changed. Our algorithm solves this problem by calculating the features at different scales (Pyramids) and associating the corners with their best scale.

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

```
function descriptors = SIFTDescriptor(pyramid, keyPt, keyPtScale)
% INPUT:
%   pyramid: Image pyramid. pyramid{i} is a rescaled version of the original
    image, in grayscale double format
%   keyPt: N * 2 matrix, each row is a key point pixel location in pyramid{
    round(scale)}. So pyramid{round(scale)}(y,x) is the center of the keypoint
%   scale: N * 1 matrix, each entry holds the index in the Gaussian pyramid
    for the keypoint. Earlier code interpolates things, so this is a double,
    but we'll just round it to an integer.
% OUTPUT:
%   descriptors: N * 128 matrix, each row is a feature descriptor

%% Initializations
% Number of keypoints that were found by the DoG blob detector
N = size(keyPt, 1);
% For each keypoint we will extract a region of size patch_size x patch_size
    centered at the keypoint.
patch_size = 16;
% The patch extracted around each keypoint will be divided into a grid of
    grid_size x grid_size.
grid_size = 4;
% Each histogram covers an area "pixelsPerCell" wide and tall
pixelsPerCell= patch_size / grid_size;
% The number of orientation bins per cell
num_bins = 8;
```

```
% Initialize descriptors to zero
descriptors = zeros(N, grid_size*grid_size*num_bins);

grad_mag = cell(length(pyramid),1);
grad_theta = cell(length(pyramid),1);
% Determine the gradient angles and magnitude of all images in the pyramid.
[grad_mag,grad_theta] = ComputeGradient(pyramid);

% Iterate over all keypoints
for i = 1 : N
    scale = round(keyPtScale(i));
    magnitudes = grad_mag{scale};
    thetas = grad_theta{scale};
    % Extract a patch of magnitudes and directions (16x16) around the keypoint
    [patch_mag,patch_theta] = Extract_Patch(keyPt(i,:),patch_size,magnitudes,
        thetas);
    if( isempty(patch_mag))
        continue;
    end
    % Normalize the gradient directions relative to the dominant gradient
        direction
    patch_theta = Normalize_Orientation(patch_mag, patch_theta);
    % Weight the gradient magnitudes using a Gaussian function
    patch_mag = patch_mag .* fspecial('gaussian', patch_size, patch_size / 2);
    % ComputeSIFTDescriptors for the patch, explanation is in the function
        implementation
    feature = ComputeSIFTDescriptor(patch_mag,patch_theta,grid_size,
        pixelsPerCell,num_bins);
    % Add the feature vector we just computed to our matrix of SIFT
        descriptors.
    descriptors(i, :) = feature;
end
% Normalize the descriptors.
descriptors = NormalizeDescriptors(descriptors);
end
```

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

```
function [grad_mag,grad_theta] = ComputeGradient(pyramid)
%   Input:
%   pyramid: all the pyramid images in a cell array
%   Output:
%   grad_mag, grad_mag:
%       Two cell arrays of the same shape where grad_mag{i} and grad_theta{i}
    give the magnitude and direction of the i-th pyramid image.
    Gradient_angles ranges from 0 to 2*pi.
grad_theta = cell(length(pyramid),1);
grad_mag = cell(length(pyramid),1);
%Do this for all pyramid images
for scale = 1:length(pyramid)
    currentImage = pyramid{scale};
    grad_mag{scale} = zeros(size(currentImage));
    grad_theta{scale} = zeros(size(currentImage));
    % Discrete approximation of gradients
    img_dx = filter2([-1 0 1], currentImage);
```

```
    img_dy = filter2([-1; 0; 1], currentImage);

    grad_mag{scale} = sqrt(img_dx.*img_dx + img_dy.*img_dy);
    grad_theta{scale} = atan2(img_dy, img_dx) - deg2rad(90);
    % atan2 gives angles from -pi to pi. To make the histogram code easier, we'll change that to 0 to 2*pi.
    grad_theta{scale} = mod(grad_theta{scale}, 2*pi);
end
end
```

Code Snippet 2: My implementation of ComputeGradient function.

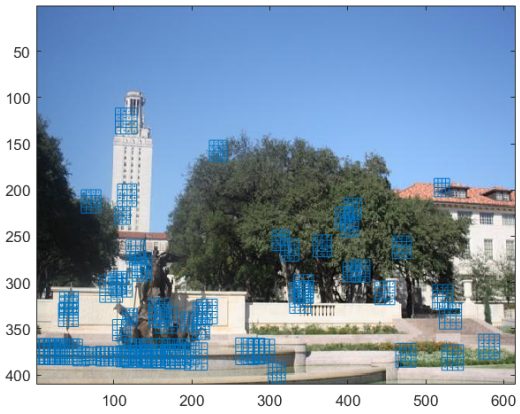
```
function descriptor = ComputeSIFTDescriptor(patch_mag,patch_theta,grid_size,pixelsPerCell,num_bins)
% INPUT:
%   patch_mag, patch_theta:
%       Two arrays of the same shape where patch_mag(i) and patch_theta(i) give the magnitude and direction of the gradient for the ith point. gradient_angles ranges from 0 to 2*pi
% OUTPUT:
%   descriptor: A 128 length descriptor calculated from 8-bin histograms of the 16 cells in the grid

% initialize descriptor to size 0
descriptor = [];
% iterate over each pixels
for x = 1 : grid_size
    % divide grid to cells
    cell_x_start = 1 + (x-1)*pixelsPerCell;
    cell_x_end = x * pixelsPerCell;
    for y = 1:grid_size
        cell_y_start = 1 + (y-1)*pixelsPerCell;
        cell_y_end = y * pixelsPerCell;
        % get cells from the grid
        mag_cell = patch_mag(cell_x_start:cell_x_end,...
            cell_y_start:cell_y_end);
        theta_cell = patch_theta(cell_x_start:cell_x_end,...
            cell_y_start:cell_y_end);
        % get the orientation histogram by using ComputeGradientHistogram
        cell_gradient_histogram = ComputeGradientHistogram(num_bins, mag_cell, theta_cell);
        % add descriptor of the cell to list of descriptors
        descriptor = [descriptor, cell_gradient_histogram];
    end
end
end
```

Code Snippet 3: My implementation for ComputeSIFTDescriptor function.

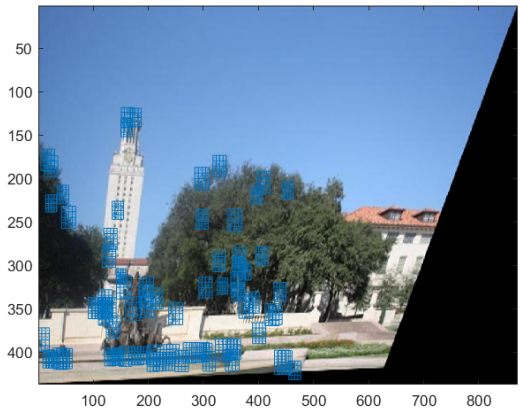
```
function [histogram, angles] = ComputeGradientHistogram(num_bins,
    gradient_magnitudes, gradient_angles)
% INPUT
% num_bins: The number of bins to which points should be assigned.
% gradient_magnitudes, gradient angles:
%       Two arrays of the same shape where gradient_magnitudes(i) and
    gradient_angles(i) give the magnitude and direction of the gradient for
    the ith point. gradient_angles ranges from 0 to 2*pi
%
% OUTPUT
% histogram: A 1 x num_bins array containing the gradient histogram. Entry 1
    is the sum of entries in gradient_magnitudes whose corresponding
    gradient_angles lie between 0 and angle_step. Similarly, entry 2 is for
    angles between angle_step and 2*angle_step. Angle_step is calculated as 2*
    pi/num_bins.
% angles: A 1 x num_bins array which holds the histogram bin lower bounds.
    histogram(i) contains the sum of the gradient magnitudes of all points
    whose gradient directions fall in the range [angles(i), angles(i + 1))

angle_step = 2 * pi / num_bins;
angles = 0 : angle_step : (2*pi-angle_step);
histogram = zeros(1, num_bins);
% get sizes to use in loops
[size_x, size_y] = size(gradient_angles);
% iterate over all angles
for gradient_index_x = 1:size_x
    for gradient_index_y = 1:size_y
        grad_angle = gradient_angles(gradient_index_x, gradient_index_y);
        % start checking from biggest bin
        for angle_index = length(angles):-1:1
            if(grad_angle >= angles(angle_index))
                % update the bin value
                histogram(angle_index) = histogram(angle_index)...
                    + gradient_magnitudes(gradient_index_x, gradient_index_y);
                % we are done if our angle is bigger than or equal to bin
                    lower bound
                break
            end
        end
    end
end
end
```

Code Snippet 4: My implementation of ComputeGradientHistogram function.

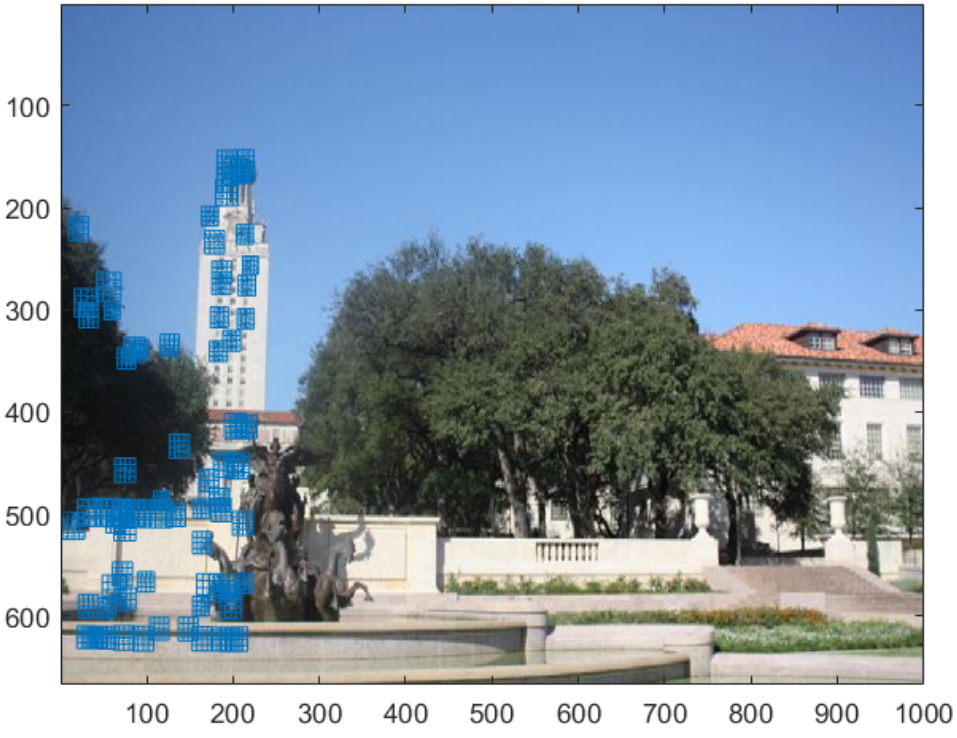
2.3 Sample Outputs



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

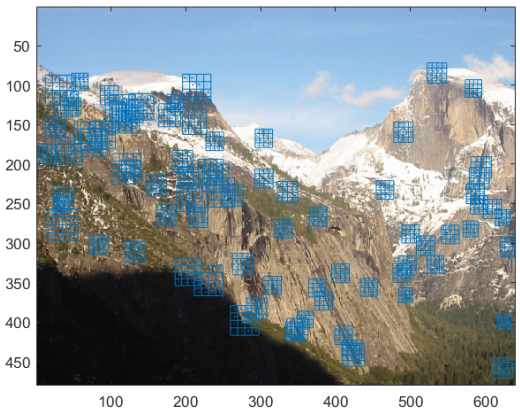


(b) SIFT descriptors for disrupted image.

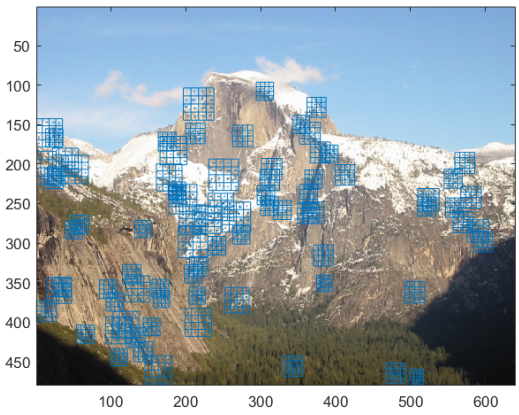


(c) SIFT descriptors for scaled up image.

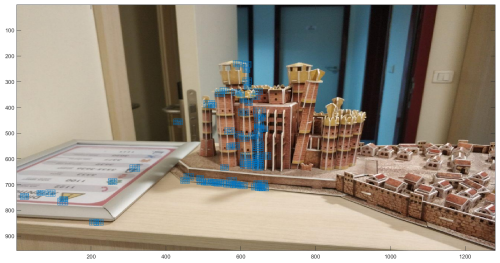
Figure 3: Visualization of SIFT descriptors extracted for different scales and orientations.



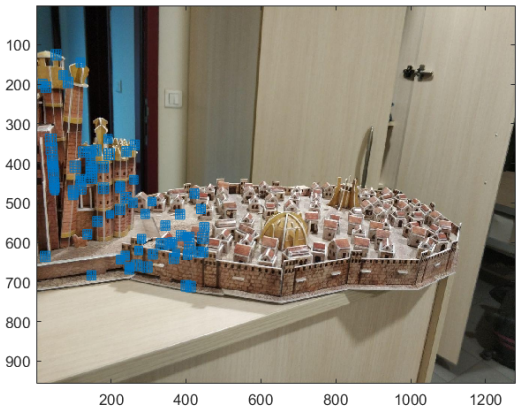
(a) SIFT Descriptors of given image yosemite1.



(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

```
function match = SIFTSimpleMatcher(descriptor1, descriptor2, thresh)
% INPUT:
%   descriptor1: N1 * 128 matrix, each row is a SIFT descriptor.
%   descriptor2: N2 * 128 matrix, each row is a SIFT descriptor.
%   thresh: a given threshold of ratio. Typically 0.7
% OUTPUT:
%   Match: N * 2 matrix, each row is a match. For example, Match(k, :) = [i, j
    ] means i-th descriptor in descriptor1 is matched to j-th descriptor in
    descriptor2.
    if ~exist('thresh', 'var'),
        thresh = 0.7;
    end
    match = [];
    %Get the number of descriptors for iterating
    [ld1, ~] = size(descriptor1);
    [ld2, ~] = size(descriptor2);
    for i = 1:ld1
        sift_desc_1 = descriptor1(i, :);
        %initilialize distances with infinity
        smallest = inf;
        second_smallest = inf;
        %initialize smallest index with zero
        smallest_index = 0;
        % this implementation assumes there are at least two descriptors in
            second array, smallest variables are wrong until end of first two
            iterations
        for j = 1: ld2
            sift_desc_2 = descriptor2(j, :);
            distances = sift_desc_1 - sift_desc_2;
            % dot product of row vector of distances with its transpose is
                equivalent with sum of squares of distances
            euclidian_dist = sqrt(distances * distances');
            % update values if necessary
            if (euclidian_dist < second_smallest)
                second_smallest = euclidian_dist;
            end
            if (euclidian_dist < smallest)
                second_smallest = smallest;
                smallest = euclidian_dist;
                smallest_index = j;
```

```
            end
        end
        % check ratio for threshold
        if (smallest < second_smallest * thresh)
            % add indexes of pair to match array if they pass the condition
            match = [match; [i, smallest_index]];
        end
    end
end
```

Code Snippet 5: My implementation of SIFTSimpleMatcher function.

3.3 Sample Outputs

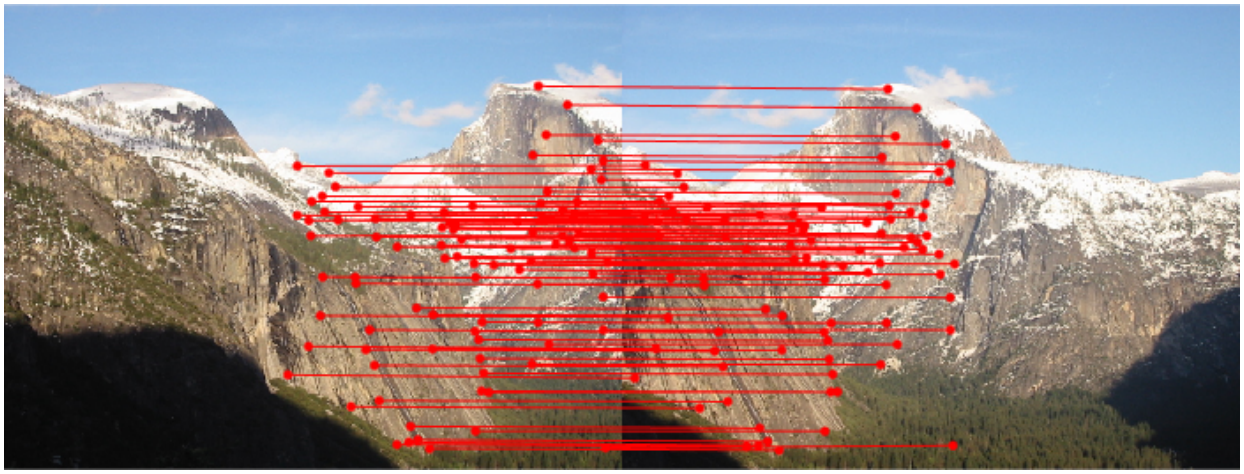


Figure 5: Visualization of matched key points between parts of Mount Yosemite.

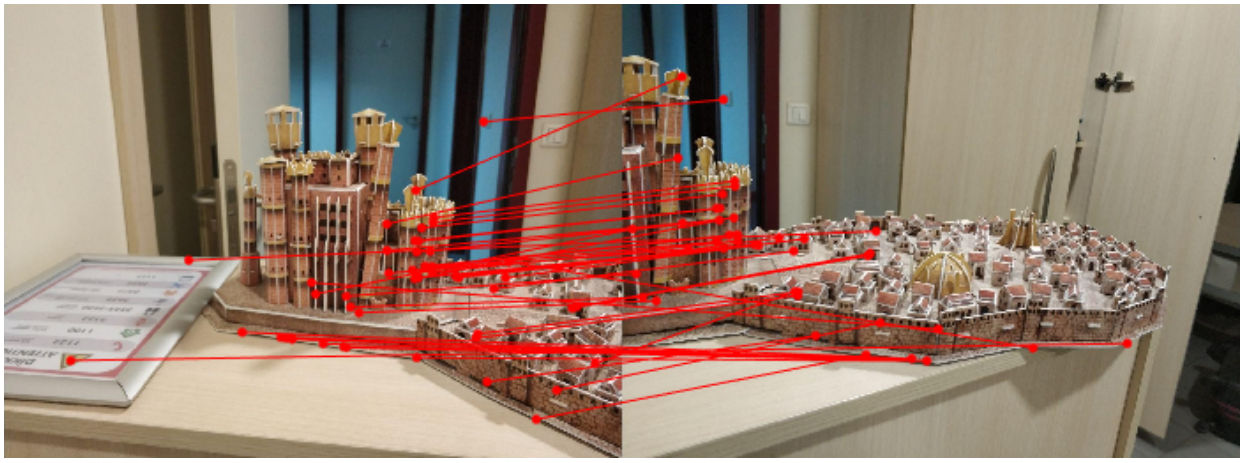


Figure 6: Visualization of matched key points between parts of King's Landing Puzzle.

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

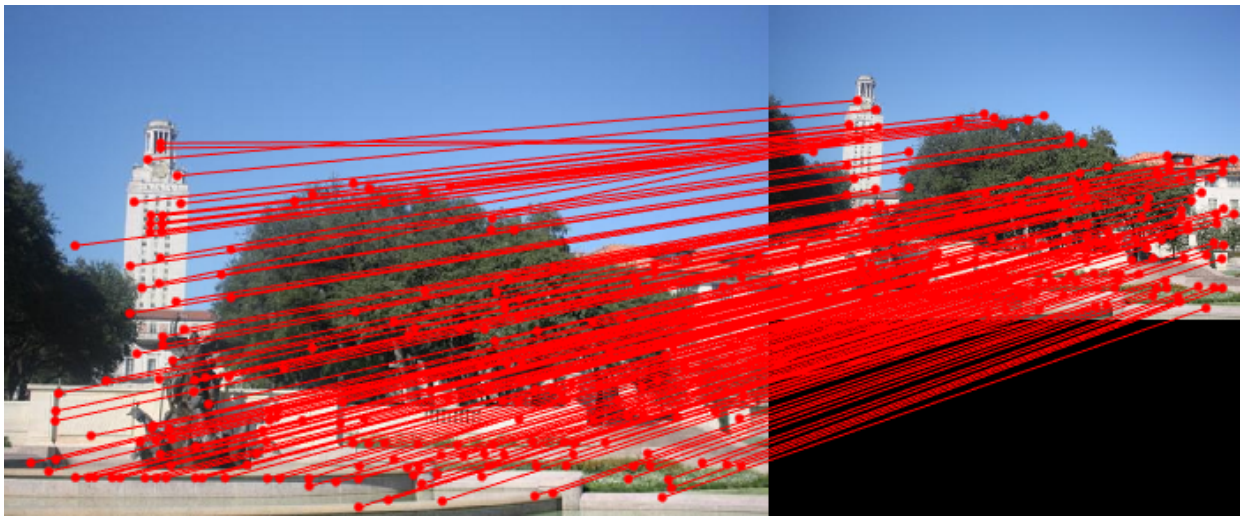


Figure 8: Visualization of matched key points between original and scaled UT Tower images.

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

```
function H_best = RANSACFit(p1, p2, match, seedSampleSize, maxInlierError,
    goodFitThresh )
% Input:
%   p1: N1 * 2 matrix, each row is a point
%   p2: N2 * 2 matrix, each row is a point
%   match: M * 2 matrix, each row represents a match [index of p1, index of p2
    ]
%   seedSampleSize: The number of randomly-chosen seed points that we'll use
    to fit
%   maxInlierError: A match not in the seed set is considered an inlier if
%                   its error is less than maxInlierError. Error is
%                   measured as sum of Euclidean distance between transformed
%                   point1 and point2. You need to implement the
%                   ComputeCost function.
%   goodFitThresh: The threshold for deciding whether or not a model is
%                  good; for a model to be good, at least goodFitThresh
%                  non-seed points must be declared inliers.
% Output:
%   H: a robust estimation of affine transformation from p1 to p2

%% Initializations
N = size(match, 1);
if N<3
    error('not enough matches to produce a transformation matrix')
end
if ~exist('seedSetSize', 'var'),
    seedSampleSize = ceil(0.2 * N);
end
if ~exist('maxInlierError', 'var'),
    maxInlierError = 30;
end
if ~exist('goodFitThresh', 'var'),
    goodFitThresh = floor(0.7 * N);
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     % Use non_seed_group to compute error from eucledian distance
54     err = ComputeError(H, p1(non_seed_group(:, 1), :), p2(non_seed_group(:, 2)
       ,:));
55     % Select the points as inliers which have error less than maxInlierError
56     inliers = [];
57     for index = 1:length(non_seed_group)
58        if (err(index) < maxInlierError)
59            inliers = [inliers; non_seed_group(index, :)];
60        end
61     end
62     % recalculate H using only inliers if number of inliers is bigger than
       threshold
63     number_of_inliers = size(inliers,1) + size(seed_group,1);
64     if( number_of_inliers > goodFitThresh )
65         all_inliers = [seed_group ; inliers];
66         H = ComputeAffineMatrix( p1(all_inliers(:, 1), :), p2(all_inliers(:,
           2), :) );
67         err = ComputeError(H, p1(all_inliers(:, 1), :), p2(all_inliers(:, 2)
           ,:));
68         % I used sum of squares of errors to evaluate H
69         sum_err_squares = err' * err;
70         if(sum_err_squares < min_error)
71             min_error = sum_err_squares;
72             H_best = H;
73         end
74     end
75 end
76 if sum(sum((H_best - eye(3)).^2)) == 0,
77     disp('No RANSAC fit was found.')
78 end
79 end
```

Code Snippet 6: My implementation of RANSACFit function.

```
function H = ComputeAffineMatrix( Pt1, Pt2 )

N = size(Pt1,1);
if size(Pt1, 1) ~= size(Pt2, 1),
    error('Dimensions unmatched.');
elseif N<3
    error('At least 3 points are required.');
end
% Convert the input points to homogeneous coordintes.
P1 = [Pt1';ones(1,N)];
P2 = [Pt2';ones(1,N)];
% P1'*H' = P2' -> H' = P1'\P2'
H_transpose = P1'\P2';
H = H_transpose';
% Sometimes numerical issues cause least-squares to produce a bottom
% row which is not exactly [0 0 1], which confuses some of the later
% code. So we'll ensure the bottom row is exactly [0 0 1].
H(3,:) = [0 0 1];
end
```

Code Snippet 7: My implementation of ComputeAffineMatrix function.

```
function dists = ComputeError(H, pt1, pt2)
% Input:
%   H: 3 x 3 transformation matrix where H * [x; y; 1] transforms the point
%   (x, y) from the coordinate system of pt1 to the coordinate system of pt2.
%   pt1: N1 x 2 matrix where each ROW is a data point [x_i, y_i]
%   pt2: N2 x 2 matrix where each ROW is a data point [x_i, y_i]
%   match: M x 2 matrix, each row represents a match [index of pt1, index of
    pt2]
% Output:
%    dists: An M x 1 vector where dists(i) is the error of fitting the i-th
%           match to the given transformation matrix.
%           Error is measured as the Euclidean distance
dists = zeros(size(pt1,1),1);
% Convert the input points to homogeneous coordintes.
pt1_homogeneous = [pt1' ; ones(1, length(pt1))];
pt2_homogeneous = [pt2' ; ones(1, length(pt2))];
% transform p1 using H
pt1_transformed = (H * pt1_homogeneous);

distances = pt1_transformed - pt2_homogeneous;
distances_x = distances(1, :);
distances_y = distances(2, :);
% distance_z is always all zero
% Calculate the Euclidean distances
% transpose the vector to satify documentation (return M x 1 matrix).
dists = sqrt(distances_x .* distances_x + distances_y .* distances_y)';
if size(dists,1) ~= size(pt1,1) || size(dists,2) ~= 1
    error('wrong format');
end
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

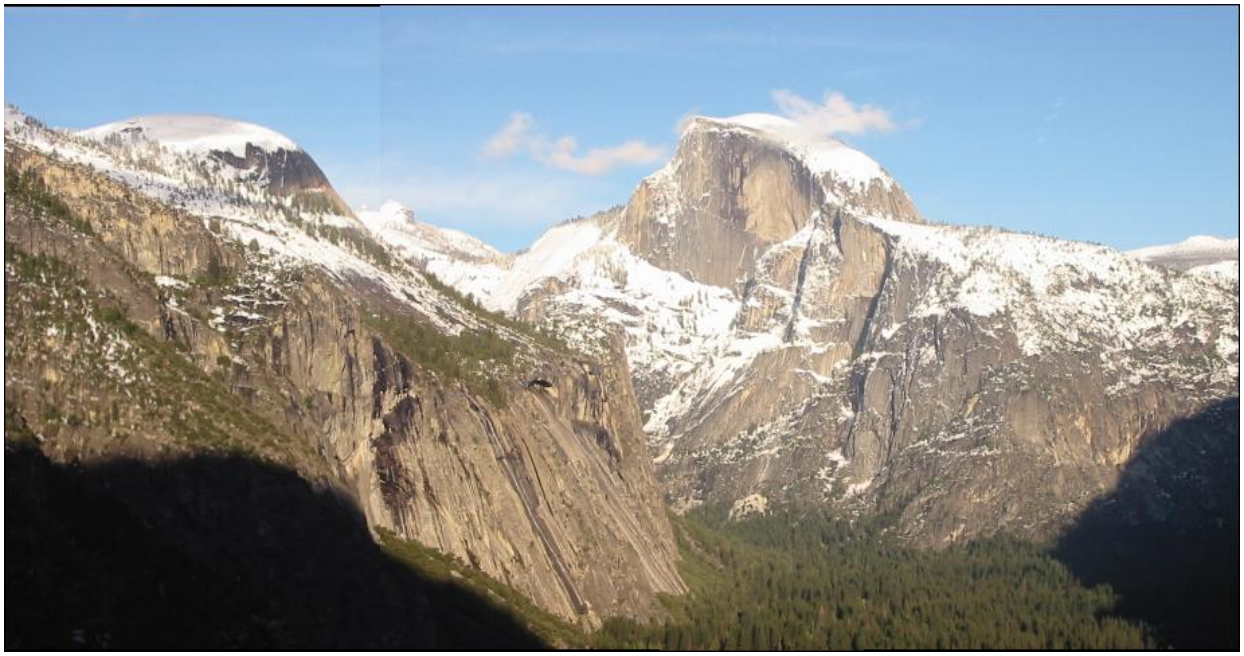


Figure 10: Stitched image of Mount Yosemite.

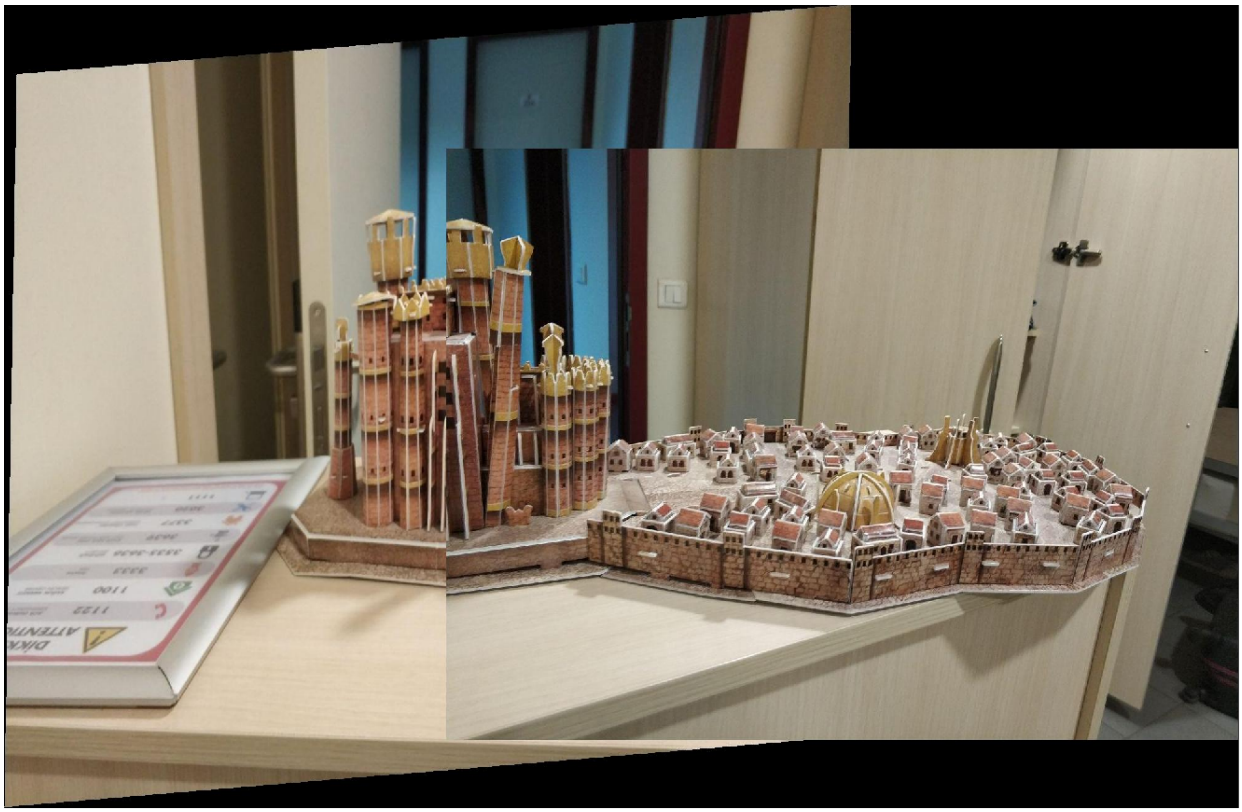


Figure 11: Stitched image of King's Landing Puzzle.

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

```
function T = makeTransformToReferenceFrame(i_To_iPlusOne_Transform, currentFrameIndex, refFrameIndex)
%makeTransformToReferenceFrame
% INPUT:
%   i_To_iPlusOne_Transform: this is a cell array where
%   i_To_iPlusOne_Transform{i} contains the 3x3 homogeneous transformation
%   matrix that transforms a point in frame i to the corresponding point in frame i+1
%   currentFrameIndex: index of the current coordinate frame in i_To_iPlusOne_Transform
%   refFrameIndex: index of the reference coordinate frame
% OUTPUT:
%   T: A 3x3 homogeneous transformation matrix that would convert a point
%   in the current frame into the corresponding point in the reference
%   frame. For example, if the current frame is 2 and the reference frame
%   is 3, then T = i_To_iPlusOne_Transform{2}. If the current frame and
%   reference frame are not adjacent, T will need to be calculated.

T = eye(3);
% if target index is bigger than current index, iteratively multiply transformation matrix with T from left until new matrix until goal is reached
if (currentFrameIndex < refFrameIndex)
   for i = currentFrameIndex:refFrameIndex - 1
      T =  i_To_iPlusOne_Transform{i} * T;
   end
 % if target index is lower than current index, first find the inverse of transformation matrix and multiply it with T from left until goal is reached
elseif (currentFrameIndex > refFrameIndex)
    for i = currentFrameIndex - 1: -1:refFrameIndex
      T =  pinv(i_To_iPlusOne_Transform{i}) * T;
    end
end
end
```

Code Snippet 9: My implementation of makeTransformToReferenceFrame function.

6.3 Sample Outputs

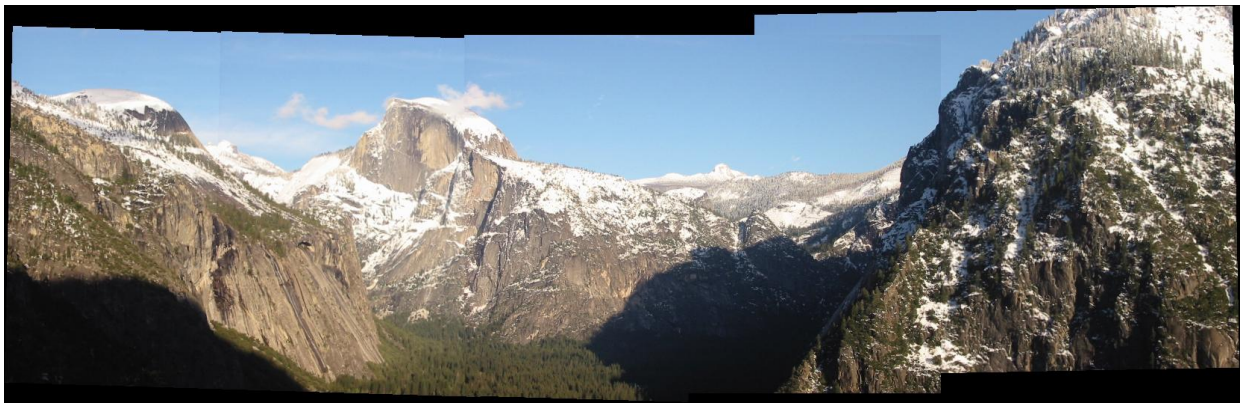


Figure 13: Stitched panoramic image of Mount Yosemite from multiple images.

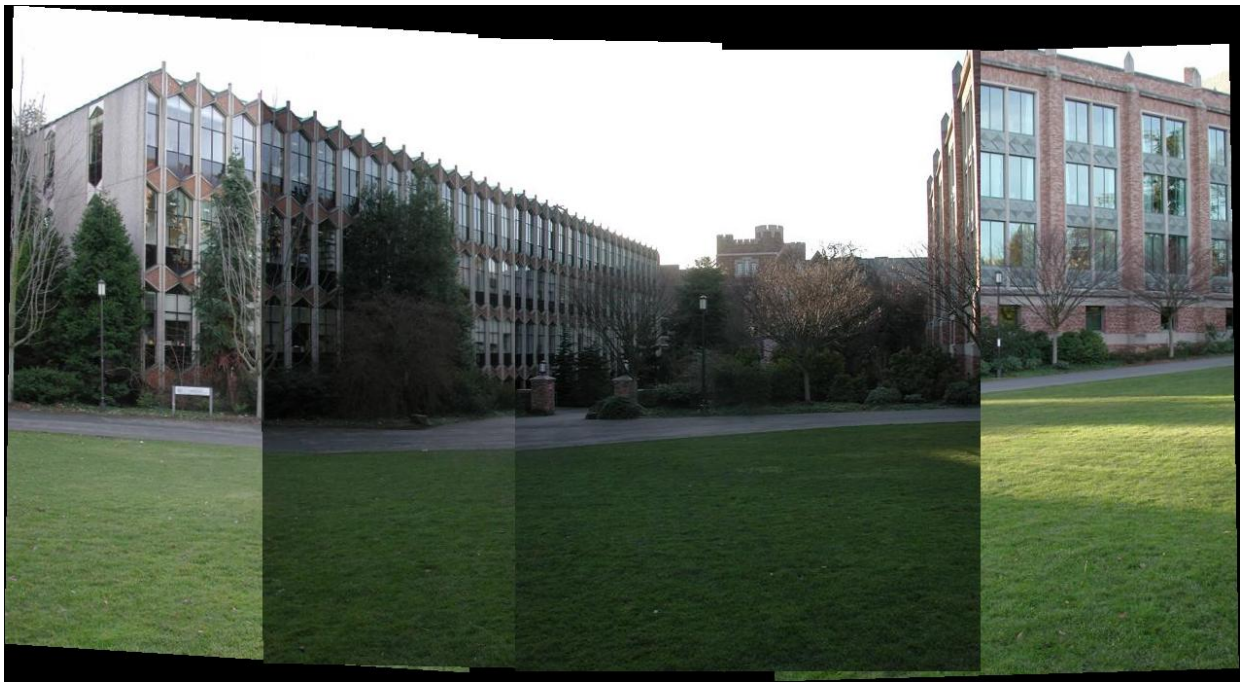


Figure 14: Stitched panoramic image of campus from multiple images..

Figure 15: Stitched panoramic image of given yard from multiple images.

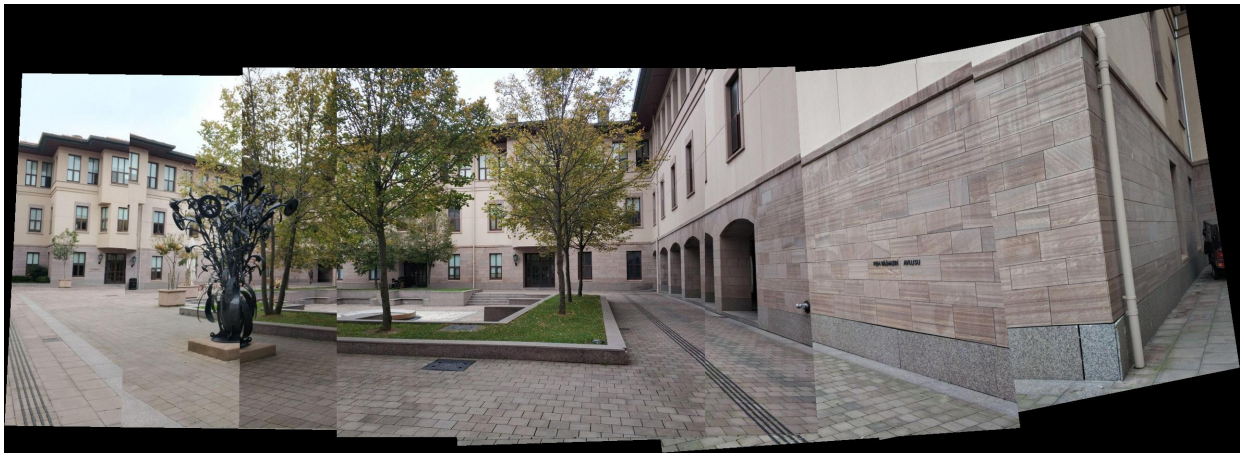


Figure 16: Stitched panoramic image of Koç University Science Courtyard from multiple images.

Figure 17: Stitched panoramic image from Koç University Campus from multiple images.

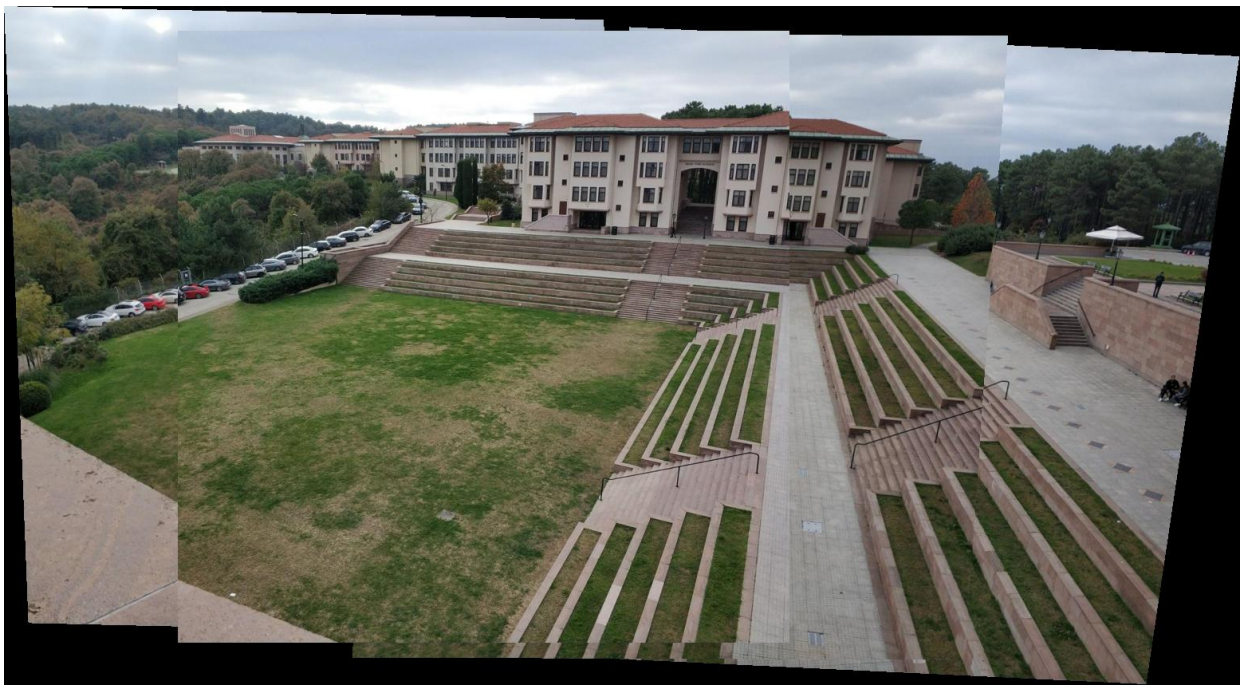


Figure 18: Stitched panoramic image of Koç University Henry Ford Building from multiple images.

7 Principal Component Analysis

We have K=3 observations of a two-dimensional random data vector f are given as $f_1 = (1, 0)$, $f_2 = (0, 1)$ and $f_3 = (0, -1)$.

7.1 Question 1

Find the direction v_1 that described the data, i.e, satisfies given condition using Lagrange Multipliers method.

$$v_1 = \arg\max_{||v||=1} \sum_{n=1}^{K} (f_k v^T)^2$$

Solution:

$$Let \quad v = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \quad \text{then} \quad \sum_{n=1}^{K} (f_k v^T)^2 = (\begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} v_x & v_y \end{bmatrix})^2 + (\begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} v_x & v_y \end{bmatrix})^2 + (\begin{bmatrix} 0 \\ -1 \end{bmatrix} \begin{bmatrix} v_x & v_y \end{bmatrix})^2 = {v_x}^2 + 2{v_y}^2$$

So, we can maximize this value using Lagrange Multipliers method with the given constraint.

$$f(v_x, v_y) = {v_x}^2 + 2{v_y}^2 \quad \text{is subject to constraint} \quad {v_x}^2 + {v_y}^2 = 1$$
$$F(v_x, v_y, \lambda) = {v_x}^2 + 2{v_y}^2 - \lambda({v_x}^2 + {v_y}^2 - 1) \quad \text{we need to solve for} \quad F_{v_x} = 0, \quad F_{v_y} = 0, \quad F_\lambda = 0$$
$$\frac{\partial F}{\partial v_x} = 2v_x - \lambda 2v_x = 0 \implies \lambda_1 = 1$$
$$\frac{\partial F}{\partial v_x} = 4v_y - \lambda 2v_y = 0 \implies \lambda_2 = 2$$
$$\frac{\partial F}{\partial \lambda} = {v_x}^2 + {v_y}^2 - 1 = 0 \quad \text{initial constraint.}$$

When we put λ_2 in to F we get,

$$f(v_x, v_y, 2) = -{v_x}^2 + 2$$

So, we need to select $v_x = 0$ to maximize the value. Combined with our constraint this gives us our eigenvector corresponding to eigenvalue 2 as

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

7.2 Question 2

Find the principal eigenvector of the correlation matrix of f, that is the one associated with the largest eigenvalue. Show that it is equal to the solution of the above optimization problem.
Solution:
We can find correlation matrix as D^TD, where D consist of our f_i vectors.

$$D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \implies C = D^T D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

We will find all eigenvalues λ such that the matrix $A - \lambda I$ is not invertible. To do so, we perform row and column operations on the matrix $C - \lambda I$ as follows:

$$C - \lambda I = \begin{bmatrix} 1 - \lambda & 0 \\ 0 & 2 - \lambda \end{bmatrix}$$

It is easy now to see that $C - \lambda I$ is singular if and only if $(1 - \lambda)(2 - \lambda) = 0$, and this means that the eigenvalues of C are 1 and 2.

To find the eigenvector associated with the largest eigenvalue, we must find vectors x which satisfy $(C - \lambda I)x = 0$ for largest eigenvalue. So, we must first form the matrix C - 2I. and then construct the augmented matrix (C - 2I : 0) and convert it to row reduced echelon form.

$$C - 2I = \begin{bmatrix} -1 & 0 \\ 0 & 0 \end{bmatrix} \implies C = D^T D = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \implies \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \implies x_1 = 0$$

Since we know the length of the v must be 1, we conclude $v_1 =$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

As seen from calculations, result is equal to result of optimization problem.
Note: For both part 1 and 2 choosing v_y = -1 also satisfies the given conditions. I preferred 1 for simplicity.

7.3 Question 3

Compute also the average (mean) of the three data vectors provided. Compare the resulting mean vector with v_1 . Are they equal?
Solution:
The average of the given data vectors is

$$\frac{f_1 + f_2 + f_3}{3} = \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} \quad where \quad v_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

So, they are not equal.