Computer Vision
Mini-project 4
CS 670, Fall 2019

Name: Kunjal Panchal
Student ID: 32126469
Email: kpanchal@umass.edu

November 6, 2019

Contents

1 Problem 1: Scale-space blob detection **3**
- 1.1 Building a Laplacian Scale Space . . . 5
- 1.2 Perform Non-Maximum Suppression in Scale Space . . 6
- 1.3 Storing the Blobs . . . 8
- 1.4 Results and Conclusions . . . 9

2 Problem 2: Image Stitching **12**
- 2.1 Detect Keypoints in Each Image . . . 12
- 2.2 Extract SIFT Features at Each Detected Keypoints . . . 12
- 2.3 Match Features Based on Pairwise Distance . . . 13
- 2.4 Intermediate Results: Matching the SIFT features . . . 15
- 2.5 RANSAC to estimate the best affine transformation . . . 23
  - 2.5.1 Extracting all the matches between SIFT features 23
  - 2.5.2 Affine Transform . . . 24
  - 2.5.3 RANSAC Initial Structures . . . 25
  - 2.5.4 Running RANSAC . . . 26
- 2.6 Two Stitched images using the Estimated Transformation 28

3 Extensions for Extra Credit **38**
- 3.1 Downsample Image and Filter it with Fixed Kernel Size . 38
- 3.2 Homography Transformation in RANSAC . . . 40

A

Matlab code for Problem 1: Scale-space blob detection **52**
- A.1 Implementation of `detectBlobs.m` . . . 52

B

Matlab code for Problem 2: Image Stitching **57**
- B.1 Implementation of `computeMatches.m` . . . 57
- B.2 Implementation of `ransac.m` . . . 59

1 Problem 1: Scale-space blob detection

Let's look at some definitions which might be useful in understanding the blob detection:

- BLOB DETECTION methods are aimed at detecting regions in a digital image that differ in properties, such as brightness or color, compared to surrounding regions.[1]

- SCALE-SPACE BLOB DETECTION extract features with characteristic scale that matches the image transformation such as scaling and translation.

- THE LAPLACIAN OF GAUSSIAN is a circularly symmetric operator for blob detection in 2D. Given an input image $f(x, y)$, this image is convolved by a Gaussian kernel

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma}}$$

at a certain scale σ to give a scale space representation

$$L(x, y; \sigma) = g(x, y, \sigma) * f(x, y)$$

Then, the result of applying the LAPLACIAN OPERATOR

$$\nabla^2 L = L_{xx} + L_{yy}$$

is computed, which usually results in strong positive responses for dark blobs of radius $r = \sqrt{2\sigma}$ (for a two-dimensional image) and strong negative responses for bright blobs of similar size.

- SPATIAL SELECTION The magnitude of the Laplacian response will achieve a maximum at the center of the blob, provided the scale of the Laplacian is "matched" to the scale of the blob.

- HOWEVER, THE LAPLACIAN RESPONSE DECAYS AS SCALE INCREASES

- SCALE NORMALIZATION The response of a derivative of Gaussian filter to a perfect step edge decreases as σ increases. To keep response the same (scale-invariant), must multiply Gaussian derivative by σ. Laplacian is the second Gaussian derivative, so it must be multiplied by σ^2:

$$\nabla^2_{norm} L = \sigma^2 (L_{xx} + L_{yy})$$

- MAXIMUM RESPONSE The Laplacian achieves a maximum response to a binary circle of radius r at $\sigma = r/\sqrt{2}$
- CHARACTERISTIC SCALE The scale that produces peak of Laplacian response in the blob center.

We start with converting images to grayscale (to deal with only one channel as blob locations aren't affected by RGB color intensity) and then to double (as further processing with gaussian kernels is in double data type).

```
% Convert image to grayscale and convert it to double
   [0 1].
if size(im, 3) > 1
    im = rgb2gray(im);
end
if ~isfloat(im)
    im = im2double(im);
end
```

Then, we initialize some parameters which are needed for creating Laplacian Space and its response:

```
%% Compute the scale space representation
sigma = 1.6;  % Initial Scale
k = sqrt(2);  % Factor by which the scale is
   mulitplied each time
sigma_final = power(k,16); % Last Scale
% Dynamically decide the interation levels from the
   first and last scale values and multiplication
   factor
n = ceil((log(sigma_final) - log(sigma))/log(k)); %
   Iterations of Laplacian scale space
[h, w] = size(im);
scaleSpace = zeros(h, w, n); % [h,w] - Dimensions of
   Image, n - Number of Levels in Scale Space

% Generate the Laplacian of Gaussian for the First
   Scale Level
filt_size = 2 * ceil(3*sigma) + 1;
LoG = fspecial('log', filt_size, sigma);
```

Note that;

- We are setting σ from 0 to 3 and experimenting with the results.
- Scaling factor k is $\sqrt{2}$.
- We chose first and last σ values, each scales previous value by factor $\sqrt{2}$, and then calculated number of iterations/levels; because, the multiplication factor should depend on the largest scale at which we want regions to be detected.
- We use 3d array to represent the scale space. $scaleSpace(:,:,i)$ will give the i'th level of the scale space.

1.1 Building a Laplacian Scale Space

We start with `sigma` and go through n levels of iterations to `sigma_final`; and at level, we filter image with scale-normalized Laplacian with that level's σ value. We save the square of Laplace Response for each level in `scaleSpace(:,:,i)` and increase the σ for the next level by the factor k.

```
%% Compute the blob response
% Increase filter size, keep image the same

for i = 1:n
    sigma_next = sigma * k^(i-1); % Increament sigma for next level
    filt_size = 2*ceil(3*sigma_next)+1;  % Compute the filter kernel size in the same way as previously
    LoG = sigma_next^2 * fspecial('log', filt_size, sigma_next);   % Making a custom LoG filter with trial and error
    filter_next = imfilter(im, LoG, 'same', 'replicate');  % Create a replica of the previous filter
    filter_next = filter_next .^ 2;     % Multiply it by itself
    scaleSpace(:,:,i) = filter_next;    % Store filter of each level
end
```

As shown in code snippet, we created Laplacian of Gaussian filter with `fspecial('log', hsize, sigma)` function available in Matlab.

The first option, *'log'* stands for a rotationally symmetric Laplacian of Gaussian. The second option, is filter size which we compute with the formula

$$2\lceil 3\sigma \rceil + 1$$

$+1$ is to make the value odd, because x2 (for half above/left, other half below/right) will make it even and generally, kernel sizes are always odd to maintain symmetry around the center. $\lceil\ \rceil$ makes sure we are getting an integer for the filter size, multiplying it by almost x6 is proportional to increase in σ.

Then we apply the filter on our grayscale image with `imfilter`, square the result and save it in `scaleSpace` at appropriate level. We used *'same'* as a filter option because we want the output size to be the same and *'replicate'* as the padding option.

1.2 Perform Non-Maximum Suppression in Scale Space

NON-MAXIMUM SUPPRESSION If we think about the value of corner response R for all the points in a patch that contains a corner: it is likely that more than one cell value > threshold (intuition: moving the window by one pixel will not change the image much). Thus for each candidate point > threshold, we detect if it is the absolute maximum within its territory, if not then we discard it.

First, we evaluate the functions are available for NMS:

- For each pixel location in our image, `nlfilter` processes a neighborhood surrounding this pixel and a single output pixel is produced. For this reason, nlfilter cannot be used.
- `colfilt` operates the same way, but it takes pixel neighborhoods, reshapes them into single columns, and we filter each column separately. The output for each column is a single pixel, and `colfilt` then reshapes the output so that it is an image again and that's the output.

- `ordfilt2` replaces each element in filtered image by the $order^{th}$ element in the sorted set of neighbors specified by the nonzero elements in domain.

We used `ordfilt2` with 3x3 neighborhood domain (there are 8 neighbors at most for a pixel in 2D), and we choose order as $3 * 3 = 9^{th}$ order to make it a maximum filter (because we want a filter which suppresses all non-maximums).

```
% Perform Non-Maximum Suppression for each Scale-Space Slice
suppression_size = 3; % To make it Maximum Filter
max_space = zeros(h, w, n); % NMS output for each level
for i = 1:n
    max_space(:,:,i) = ordfilt2(scaleSpace(:,:,i), suppression_size^2, ones(suppression_size));
end
```

And save each level's result in `max_space(:, :, i)`.

Next, we set a threshold on the squared Laplacian response above which to report region detections. We played around with different threshold and σ values for eahc image, we report the values which produces the best result; with the output images in later sections.

Then, we zero out all positions that are not the Local Maxima of the score (if the value if not greater than all its neighbors).

```
% Non-Maximum Suppression between Scales and Threshold
% For scales, compare the current one with the previous and next ones, choose the maximum of the three
for i = 1:n
    max_space(:,:,i) = max(max_space(:,:,max(i-1,1):min(i+1,n)),[],3);
end
% Zero Out All Positions that are not the Local Maxima of the score (if the Value if not Greater than all its Neighbors)
max_space = max_space .* (max_space == scaleSpace);
```

1.3 Storing the Blobs

We need to store the x-y co-ordinates of the blob, the radius of the blob and the score associated with it, which is just the level of the scale space.

We will look for all the maxima which are higher than some arbitrary threshold (we experimented between 0.005 to 0.03). The blobs having larger squared Laplacian than the threshold will be stored with their x-y co-ordinates and the level score.

We compute the radius from the level of the scale space where maxima is, the initial σ value and the scaling factor.

```
x = [];   % X - axis of blob's center
y = [];   % Y - axis of blob's center
radius = []; % Radius of the blob
score = [];

% For each level
for i=1:n
    % Set a Threshold on the Squared Laplacian
       Response above which to report Region
       Detections
    [rows, cols, value] = find(max_space(:,:,i).*(
       max_space(:,:,i) >= 0.01));
    numBlobs = length(rows);
    % Create Lists of each tuple fields separately
    if(numBlobs > 0)
        radii =  sigma * k^(i-1) * sqrt(2);
        radii = repmat(radii, numBlobs, 1);
        x = [x; rows];
        y = [y; cols];
        score = [score;value];
        radius = [radius; radii];
    end
```

```
20 end
21
22 blobs = [y, x, radius, score];
```

We now have all 4 attributes for every blob whose maxima passes the threshold.

1.4 Results and Conclusions

The FULL CODE for `detectBlob.m` is in Appendix A.1.

Now, we show the results we produced and the parameter values which produces the best results, see Figures 1, 2, 3 and 4:

Figure 1: butterfly.jpg: Initial Sigma - 2.0; Threshold - 0.02

Figure 2: einstein.jpg: Initial Sigma - 2.3; Threshold - 0.025

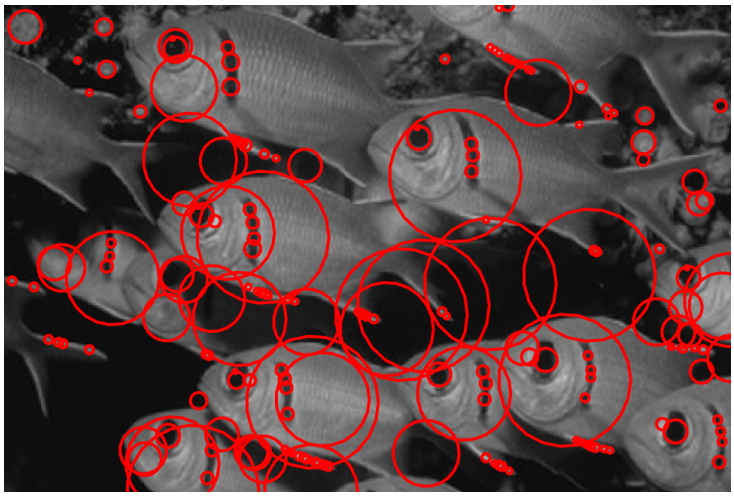


Figure 3: fishes.jpg: Initial Sigma - 1.0; Threshold - 0.019

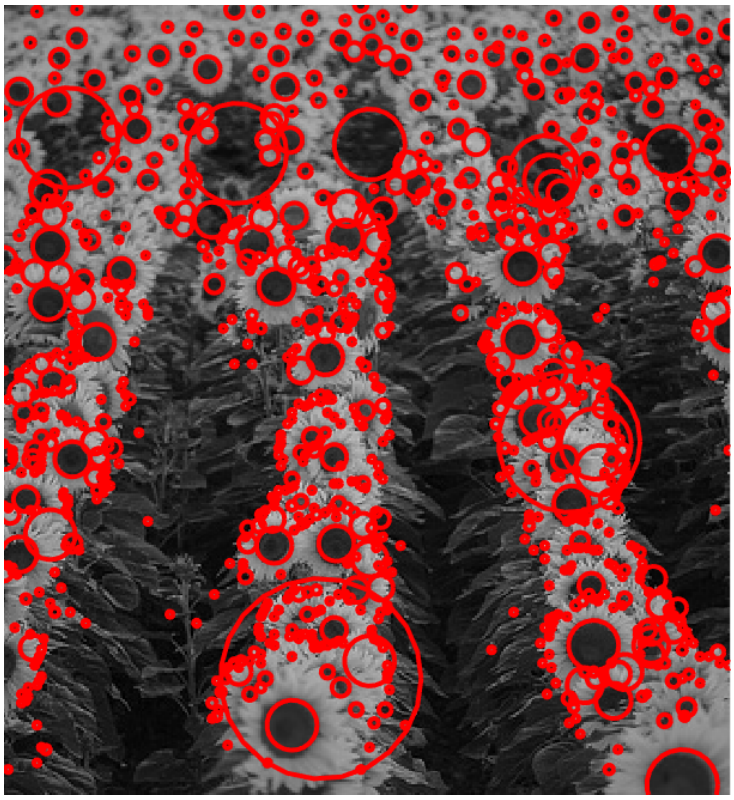


Figure 4: sunflowers.jpg: Initial Sigma - 1.0; Threshold - 0.019

2 Problem 2: Image Stitching

In this section, we will implement RANSAC to stitch two images. We will use our blob detector from Problem 1, to extract keypoints and extract feature descriptors on them. Our goal is to estimate an affine transformation using feature matching and RANSAC to produce a combined image.

We step-by-step go through alignment algorithm.

2.1 Detect Keypoints in Each Image

We already implemented this in the first section. We set initial scale $\sigma = 2$ and threshold at 0.02.

We will get blobs for both images by calling `detectBlob.m`.

2.2 Extract SIFT Features at Each Detected Keypoints

SIFT Feature Descriptors transforms an image into a large collection of feature vectors, each of which is invariant to image translation, scaling, and rotation, partially invariant to illumination changes and robust to local geometric distortion.

In this method[2]:

- We divide each blob in 4x4 grid.
- Inside each sub-patch, compute histogram of gradient orientations (8 reference angles).
- We normalize the vector to a unit length.
- Resulting descriptor will be 4x4x8 = 128 dimensions for each blob.

This step is already implemented; at the end of this step, we will get SIFT features for both of our images.

2.3 Match Features Based on Pairwise Distance

Here, we compute matches between the two sets of SIFT features. We find the best match of each feature f1 from image1 to f2 of image2 using the smallest sum-of-squared-differences.

If we find a match for f1(i) where i $\in$ [0,1,...N], from f2[0,1,...,M]; we will say `matches(i)` is the closest feature in f2 to the i_{th} feature f1(:,i). All other entries will be `matches(i)`=0, which indicates no matches.

We initialize `matches()` to all zeros and then loop through all SIFT features of image2, for each feature of image1, to find the pairs where SSD is the least and store that f2 for i^{th} feature in f1. Code snippet for this logic is presented here:

```
%% Computing Matches using SSD
matches = zeros(f1_size, 1);
% Start Stopwatch Timer
tic
if 0
    fprintf('Computing matching using SSD: \n');
    % For each feature from image 1
    for i = 1 : f1_size
        % Find its nearest match image 2
        bestMatch = inf;
        for j = 1 : f2_size
            % SSD
            match = sum(sum((f1(i,:)-f2(j,:)).^2));
            % Store the lowest SSD feature
            if(match<bestMatch)
                matches(i) = j;
                bestMatch = match;
            end
        end
    end
% Stop Stopwatch Timer
toc
end
```

There is a shortcoming of this method where we must deal with am-

biguous putative matches. We can compare distance of NEAREST NEIGHBOR to that of SECOND NEAREST NEIGHBOR.

The ratio of closest distance to second-closest distance will be HIGH FOR FEATURES THAT ARE NOT DISTINCTIVE. Threshold of 0.8 provides a good separation.

We set `matches(i)=0` if i^{th} feature f1(:,i) fails on the ratio test.

```
%% Computing Matches using ratio

% For each Descriptor in the First Image, Select its
   Match to Second Image
for i = 1 : f1_size
    % The most likely match
    bestMatch = inf;
    % The next most likely match after the best one
    secondMatch = inf;
    index = inf;
    for j = 1:f2_size
        % Compute SSD
        match = sum(sum((f1(i,:)-f2(j,:)).^2));
        % compare it with the best match
        if(match < bestMatch)
            secondMatch = bestMatch;
            bestMatch = match;
            index = j;
        % compare it with the second best match
        elseif(match < secondMatch && match ~=
           bestMatch)
                secondMatch = match;
        end
    end
    % Now we have determined best and second best
       match

    % Computer Ratio
    ratio = bestMatch/secondMatch;
    % Compare it with a threshold
    if(~isequal(index,inf) && ratio < 0.8)
        matches(i)= index;
```

```
30      else
31          matches(i) = 0;
32      end
33  end
```

Here, we have just fount out the best and second best matches by SSD and then if the ratio is above the threshold, we enter the index of best match into out `matches(i)`, otherwise it is 0.

2.4 Intermediate Results: Matching the SIFT features

Here we show the output of `showMatches(im1, im2, c1, c2, matches)`. We changed the initial σ to 1.0 and Region Detection threshold to 0.01 to get the best results. See Figures 5, 7, 9, 11, 13, 15, 17 and 19 for SIFT features. See Figures 6, 8, 10, 12, 14, 16, 18 and 20 for macthes between the SIFT features of two images.

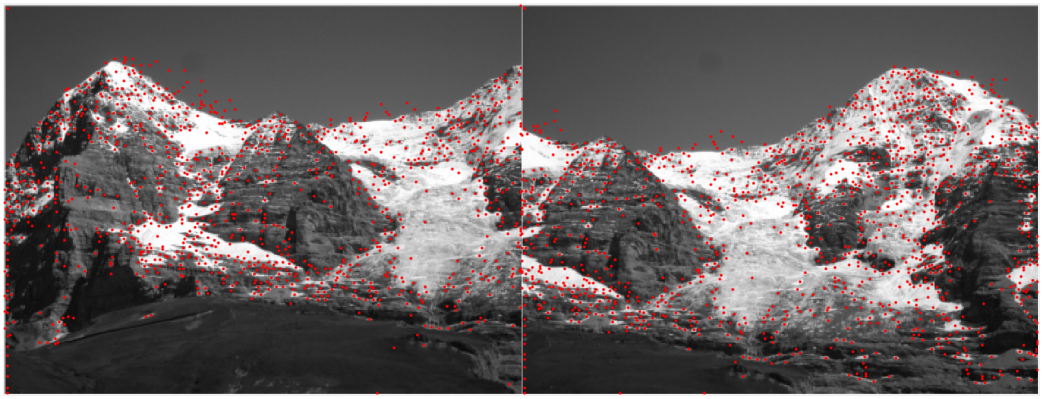


Figure 5: SIFT features of Hill images

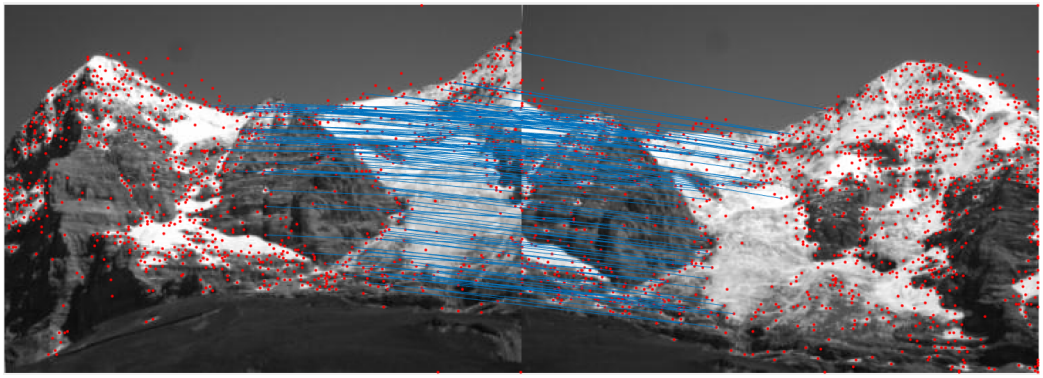


Figure 6: Matches between SIFT features of Hill images

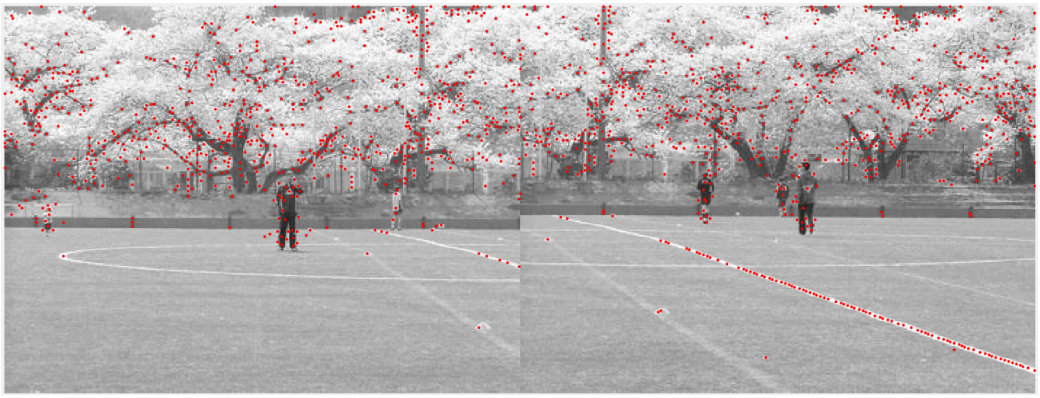


Figure 7: SIFT features of Field images

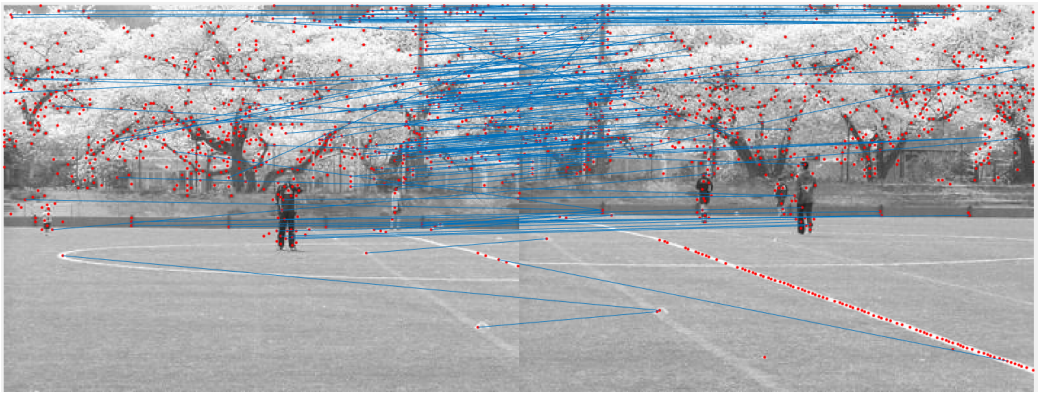


Figure 8: Matches between SIFT features of Field images

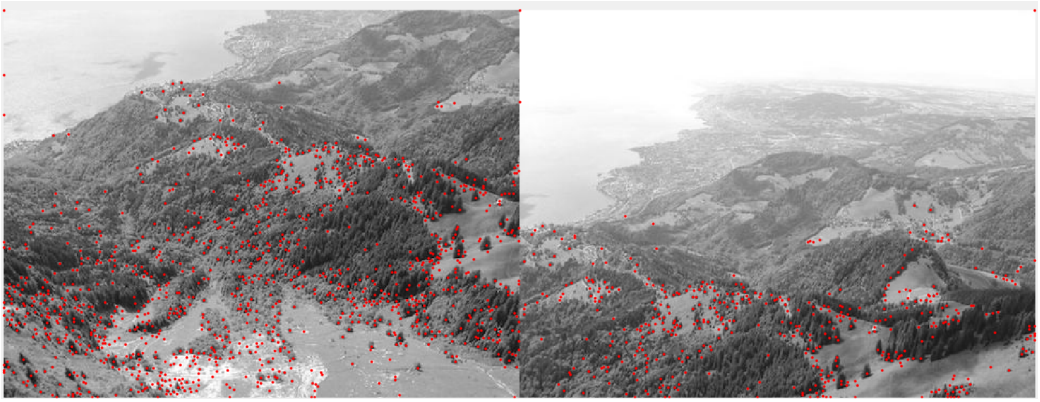


Figure 9: SIFT features of Ledge images

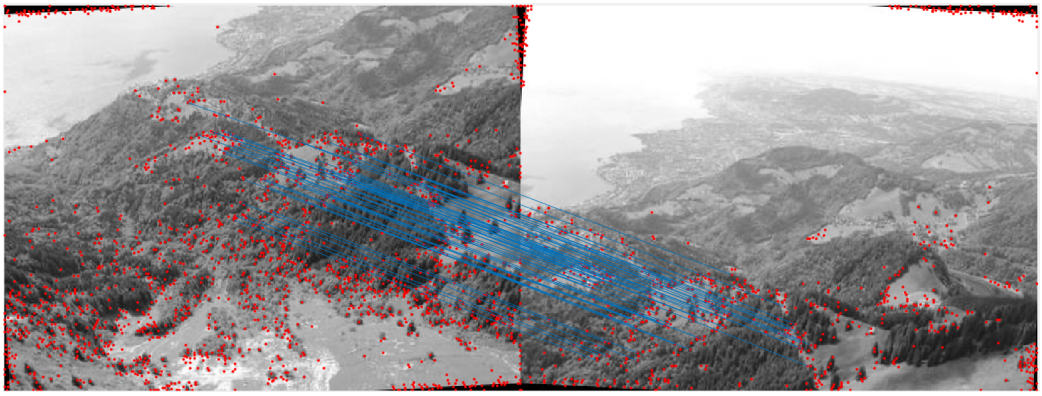


Figure 10: Matches between SIFT features of Ledge images

Figure 11: SIFT features of Pier images

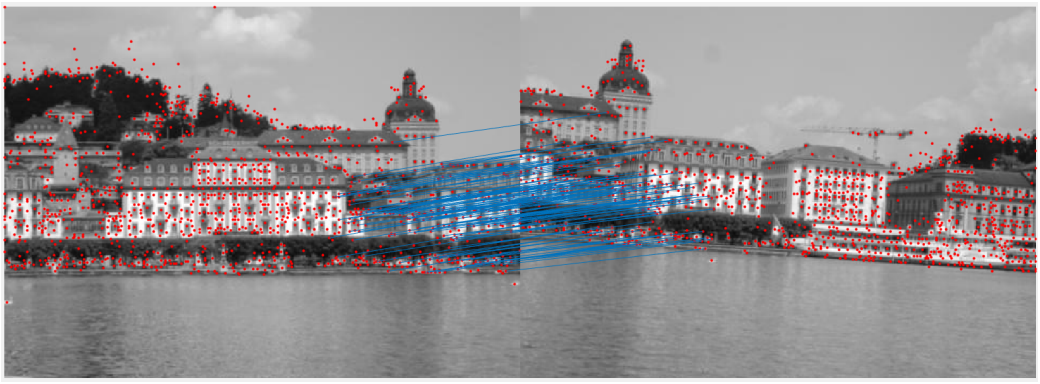


Figure 12: Matches between SIFT features of Pier images

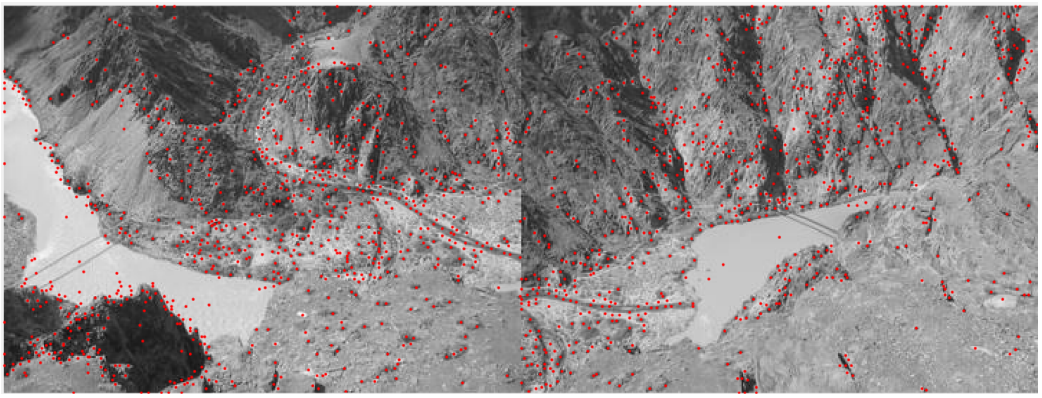


Figure 13: SIFT features of River images

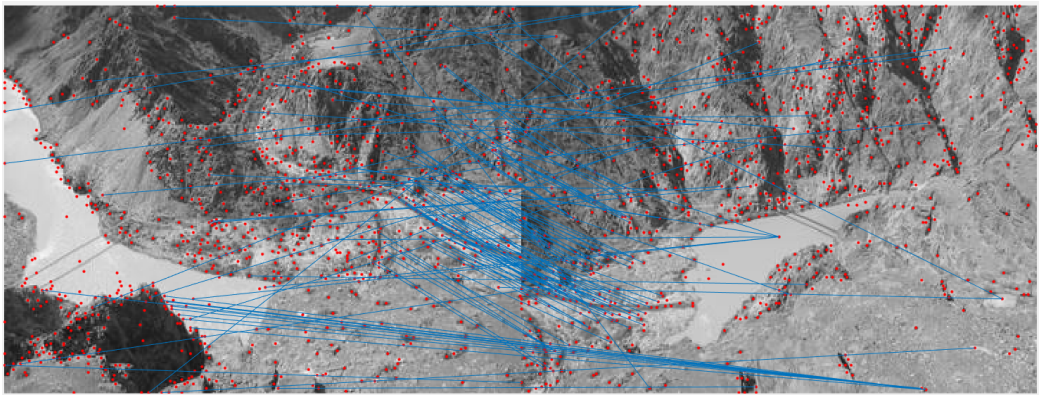


Figure 14: Matches between SIFT features of River images

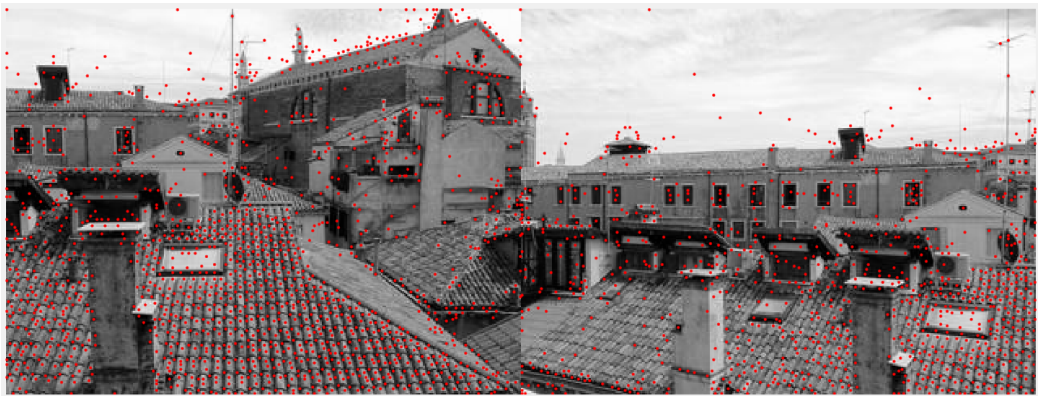


Figure 15: SIFT features of Roofs images

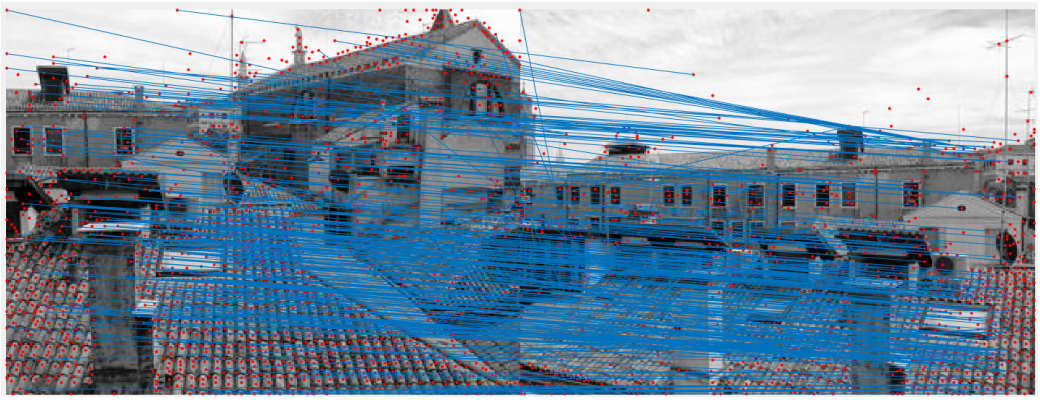


Figure 16: Matches between SIFT features of Roofs images

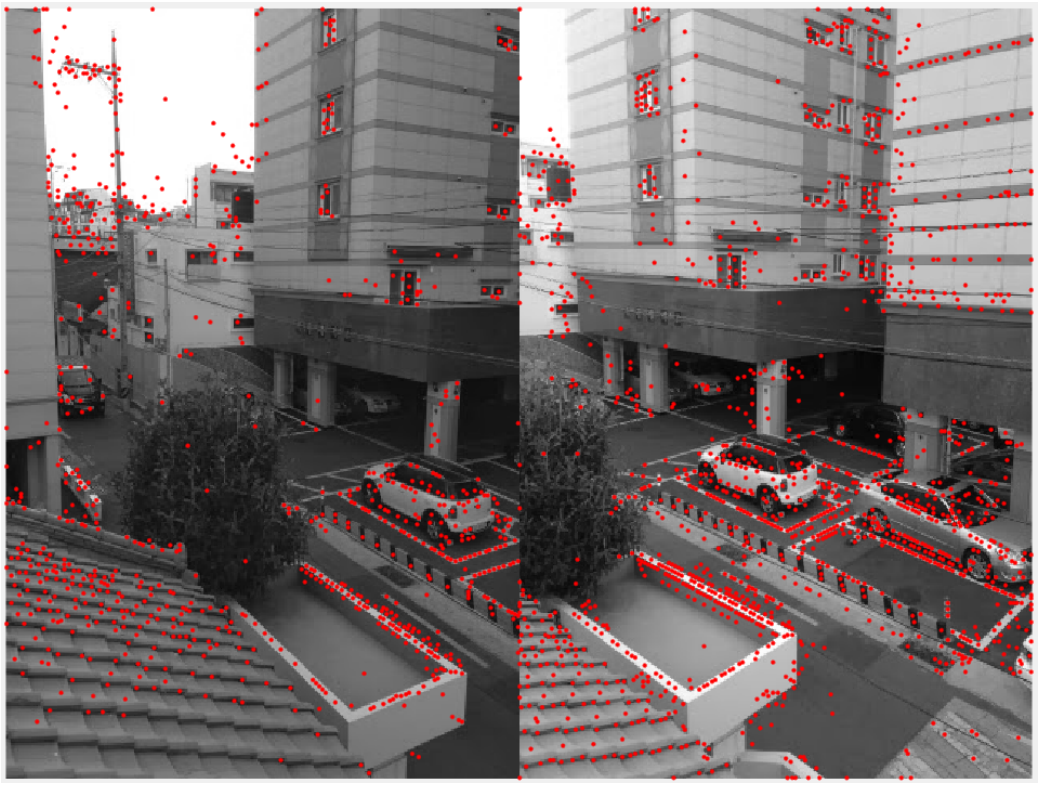


Figure 17: SIFT features of Building images

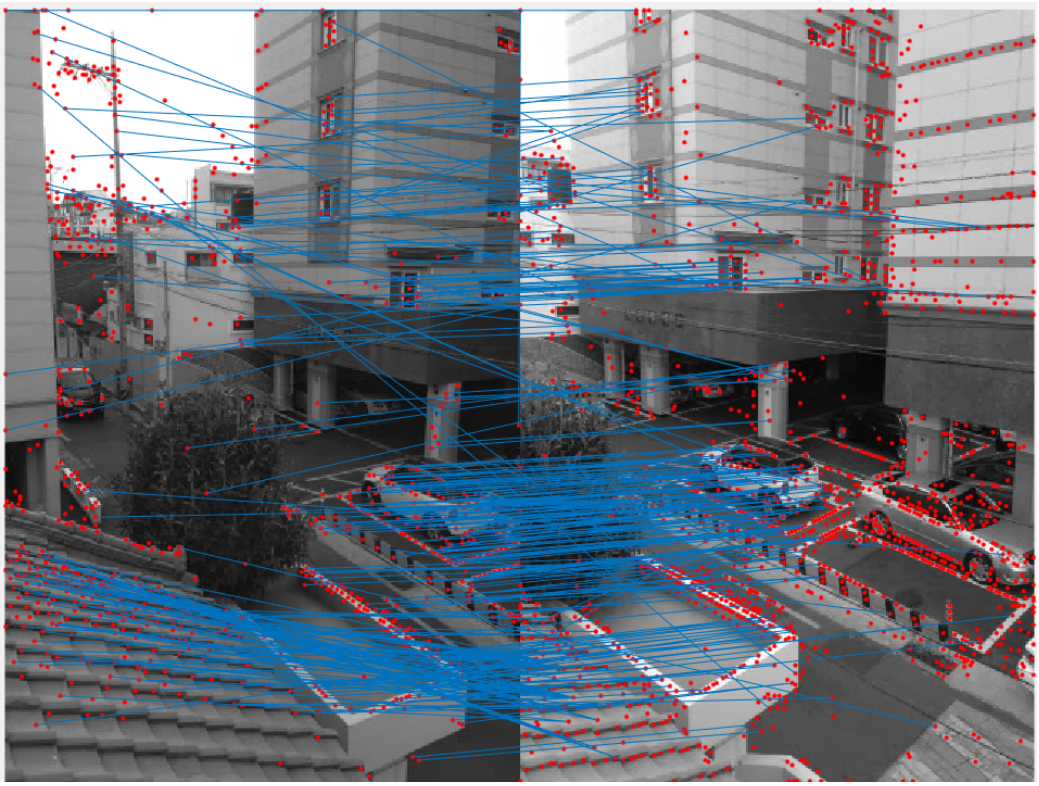


Figure 18: Matches between SIFT features of Building images

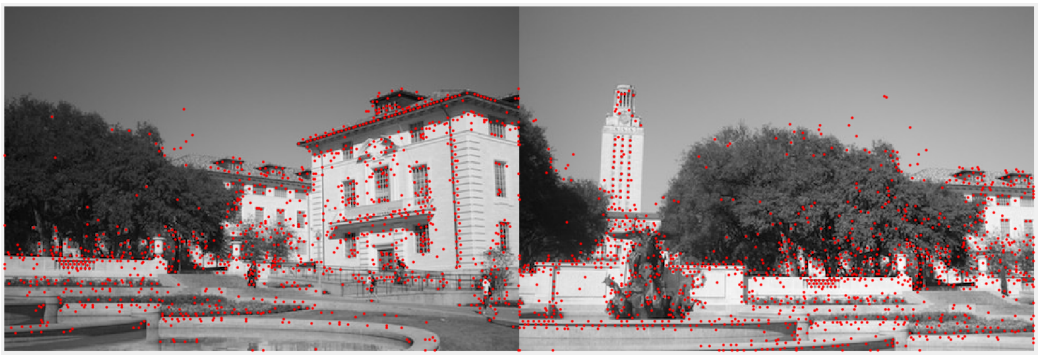


Figure 19: SIFT features of Tower images

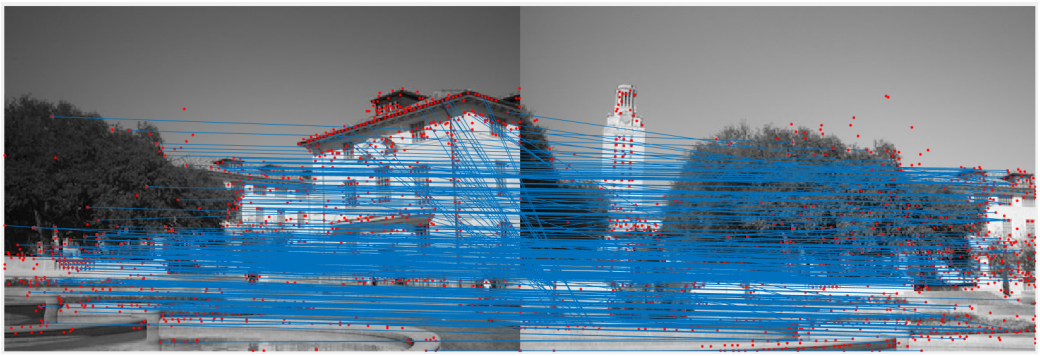


Figure 20: Matches between SIFT features of Tower images

2.5 RANSAC to estimate the best affine transformation

Using RANSAC, we will estimate the AFFINE TRANSLATION that agrees with most matches (inliers).

An affine transformation is any transformation that preserves collinearity (i.e., all points lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation).

The affine transformation of a model point $[x, y]^T$ to an image point $[x', y']^T$ can be written as below

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} m1 & m2 \\ m3 & m4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} tx \\ ty \end{bmatrix}$$

To solve for the transformation parameters the equation above can be rewritten to gather the unknowns into a column vector;

$$\begin{bmatrix} x & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & y & 0 & 1 \\ .... & & & & & \\ .... & & & & & \end{bmatrix} \begin{bmatrix} m1 \\ m2 \\ m3 \\ m4 \\ tx \\ ty \end{bmatrix} = \begin{bmatrix} u \\ v \\ . \\ . \end{bmatrix}$$

This is a linear system with six unknowns. [2]

Each match gives us two linearly independent equations, we need at least three to solve for the transformation parameters.

We implement RANDOM SAMPLE CONSENSUS - RANSAC as follows:

2.5.1 Extracting all the matches between SIFT features

We have been given blobs for image1 and image2 & the matches of SIFT features between them.

So first task will be to find all the non-empty matches from the `matches` list, because some of them are assigned to 0, which are of no use in this context.

We also store the original indices of the feature comparison entries

is that doesn't get lost in compaction, and the size of the non-zero entries to run a loop.

The code for that is shown in the snippet below:

```
%% Get all the non-empty feature matches
[inputs, cols, bases] = find(matches);
% Reset max inliers counter
maxInliers = 0;
% Store the original indices of matches so that
   removing the 0 index entries doesn't change the
   actual indices
originalIndices = find(matches);
% Get the non-zero entries size
inputs_size = size(inputs,1);
```

2.5.2 Affine Transform

We arrange the base and transform points according to the matrix forms shown in above equation. Then we just do

$$AffineTransformation = \frac{BasePoints}{TransformedPoints}$$

```
function [Transform] = computeAffine(basePoints,
   transPoints)
    baseP = zeros(6,6); % input points
    transP = zeros(6,1); % points after transformation
        of input points.
    for j = 1:3
        index = ceil(rand*(size(basePoints,1)-1))+1;
        % Prepare base matrix with points before
           transformation; put them in the position
           shown in affine transform matrix equation
        baseP(2*j-1,:) = [basePoints(index,1),
           basePoints(index,2),1,0,0,0];
        baseP(2*j,:) = [0,0,0,basePoints(index,1),
           basePoints(index,2),1];
```

```
9           % Set transformed points matrice; put them in
                the position shown in affine transform
                matrix equation
10          transP(2*j-1,:) = transPoints(index,1);
11          transP(2*j,:) = transPoints(index,2);
12      end
13      % Matrix Division
14      Transform = baseP\transP;
15  end
```

2.5.3 RANSAC Initial Structures

The structures and matrices which we are using for RANSAC iterations are as shown in the snippet below, the explanation is in comments:

```
1  % This structure stores current point co-ordinates
2  inputPoints = zeros(inputs_size,2);
3  % This structure stores the points whose co-ordinates
       are getting transformed
4  basePoints = zeros(inputs_size,2);
5  % Number of points whose distance from the line is
       less than some constant threshold
6  threshold = 4;
7  iterations = 35;
8  % Store the best affine transformation (with the least
       error)
9  best_model = zeros(2,3);
10 % Store the points who are inlier
11 inliersInfo = zeros(inputs_size,2);
12 % Store the affine transformation for a particular
       iteration
13 transInfo = zeros(2,3,inputs_size);
14
15 % Store all features and locations in inputPoints for
       1st image and in basePoints for 2nd image
16 for ind= 1:inputs_size
17      basePoints(ind,1) = blobs1(bases(ind),1);
18      basePoints(ind,2) = blobs1(bases(ind),2);
```

```
19      inputPoints(ind,1) = blobs2(inputs(ind),1);
20      inputPoints(ind,2) = blobs2(inputs(ind),2);
21  end
```

2.5.4 Running RANSAC

Now we can run the actual RANSAC algorithm.

The pseudocode:

- Randomly select minimal subset of points.
- Compute Affine Transformation for the base and transformed point matrices.
- Reshape the Affine Transform and the Base Point matrices so they can be multiplied to get the Transformed Point matrix HYPOTHESIZING THE MODEL.
- Calculate the error between actual transformed points and the ones which we computed by getting an Affine Transform COMPUTING THE ERROR FUNCTION.
- If the inlier count is more than the 10% of the input size; it counts as a valid model.
- And if the inlier count is more than the previous best, the current Affine Transform is the best now. Store the best transform, the SSD error and the input and transformed points that produce that transform.
- Iterate this procedure arbitrary times.

The code snippet for RANSAC:

```
1  % Try RANSAC for 'iterations' number of times
2  for itr = 1:iterations
3      % Affine Transform for the current iteration
4      T = computeAffine(inputPoints,basePoints);
5      % Reshape the transform matrix in [3x3] for
6      % tranformed_points = tranform x base_points
7      trans = [reshape(T,[3,2])';0,0,1];
```

```
8       % Reshape base_points to [3x1] for
9       % tranformed_points = tranform x base_points
10      inputPt = [inputPoints';ones(1,inputs_size)];
11      % Compute tranformed_points = tranform x
           base_points
12      inputptF =  trans*inputPt;
13      % Take the x-y co-ordinates of every tranformed
           point
14      calPoints = inputptF(1:2,:);
15
16      % Calculate Error between Expected Tranformed
           point co-ordinates and actual Transformed
           points
17      dist = calculateError(calPoints,basePoints');
18      % Only take the points whose distance is lower
           than threshold
19      [rows,cols,error] = find(dist.*(dist < threshold))
           ;
20      % Count the number of inliers
21      inliersCount = size(cols,2);
22      total_error = sum(error,2);
23
24      % Find the best model by inlier count and error
           values
25      if inliersCount > round(0.10*inputs_size) &&
           inliersCount >= maxInliers && inliersCount ~=
           inputs_size
26
27          best_model = reshape(T,[3,2])';
28          if(inliersCount == maxInliers)
29              maxIndex = find(inliersInfo(:,1) ==
                   maxInliers);
30              if(total_error < inliersInfo(maxIndex,2))
31                  best_model = reshape(T,[3,2])';
32              end
33          end
34          inliersC = cols';
35          maxInliers= inliersCount;
36          inliersInfo(itr,1)=maxInliers ;
37          inliersInfo(itr,2)= total_error;
38          transInfo(:,:,itr) = best_model;
```

```
39         end
40 end
```

2.6 Two Stitched images using the Estimated Transformation

THE CODE FOR `computeMatches.m` AND `ransac.m` ARE IN APPENDIX **B.1** AND **B.2**.

Figures 5 to 20 shows intermediate results.

In this section, we show the final output and the affine transforms computed to stitch the two images.
See Figures 21, 22, 23, 24, 25, 26, 27 and 28 & their corresponding Affine Transforms.

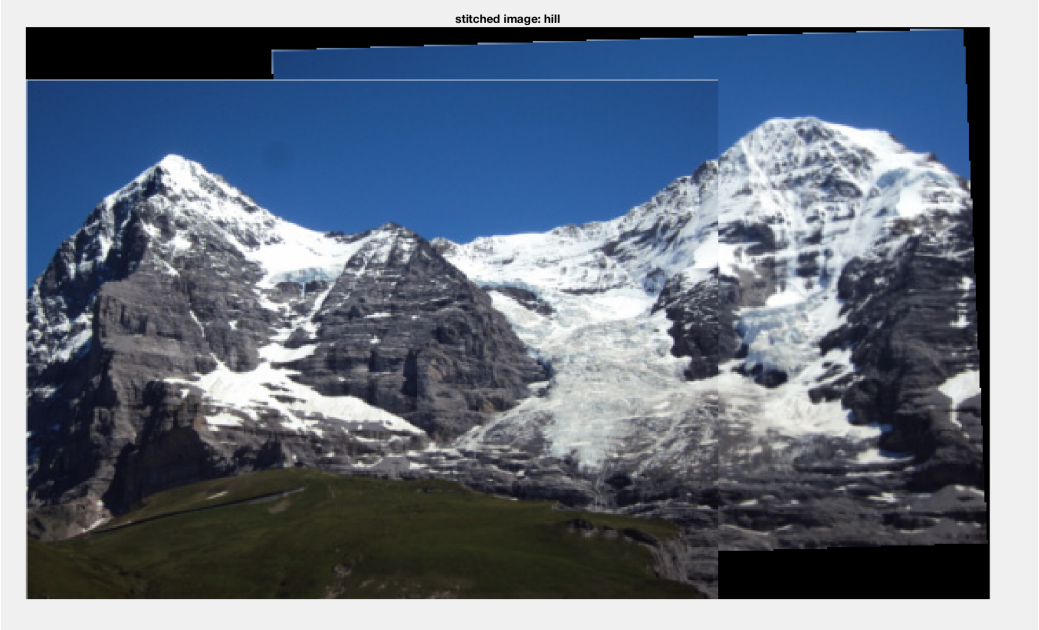


Figure 21: Panoramic Stitching of Hill images

Hill.jpg

Maximum Inliers 226 and Msize is 460
Affine Transformation is:
1.0169 0.0341 139.5263
-0.0597 1.0177 -9.4434

Figure 22: Panoramic Stitching of Field images

Field.jpg

Maximum Inliers 89 and Msize is 186
Affine Transformation is:
1.0000 0.0000 257.0000
-0.0103 1.0147 11.3895

Figure 23: Panoramic Stitching of Ledge images

Ledge.jpg

Maximum Inliers 92 and Msize is 217
Affine Transformation is:
0.9748 -0.0556 144.7856
0.0553 0.9680 -142.8991

Figure 24: Panoramic Stitching of Pier images

Pier.jpg

Maximum Inliers 207 and Msize is 469
Affine Transformation is:

0.9905	0.0237	287.4163
0.0000	1.0000	27.0000

Figure 25: Panoramic Stitching of River images

River.jpg

Maximum Inliers 51 and Msize is 138
Affine Transformation is:
0.9628 -0.3282 273.7572
0.3540 0.9889 -169.4819

Figure 26: Panoramic Stitching of Roofs images

Roofs.jpg

Maximum Inliers 59 and Msize is 316
Affine Transformation is:
0.9649 0.0504 -160.9132
-0.1228 0.9890 -21.7032

Building.jpg

Maximum Inliers 75 and Msize is 340
Affine Transformation is:
1.1428 -0.0590 93.0511
0.1105 1.0511 -25.1110

Figure 27: Panoramic Stitching of Building images

Figure 28: Panoramic Stitching of Tower images

Tower.jpg

Maximum Inliers 98 and Msize is 383
Affine Transformation is:
1.0377 -0.0680 -203.9290
0.0071 1.0284 -24.6143

3 Extensions for Extra Credit

3.1 Downsample Image and Filter it with Fixed Kernel Size

In this section, we compare the results of first approach "iteratively filter the image with kernel of increasing size" [Problem 1], with a second approach where we iteratively downsample the image and filter it with the kernel of fixed size.

At each level, we downsample(resize) the image by factor k.

Filter it with fixed sized LoG already made in the previous section.

Resize it to its original size with BICUBIC INTERPOLATION. The code snippet is as shown below:

```
imRes = im; % Create a copy of the image
for i = 1:n
    imFiltered = imfilter(imRes, LoG, 'same', '
       replicate'); % Apply the Laplacian of Gaussian
       filter here, the filter is already created in
       the above section
    imFiltered = imFiltered .^ 2; % Square of the LoG
    scaleSpace(:,:,i) = imresize(imFiltered, size(im),
        'bicubic'); % Resize the image to original
       size with bicubic interpolation
    if i < n
        imRes = imresize(im, 1/(k^i), 'bicubic'); %
           Decrease the image size for next iteration
    end
end
```

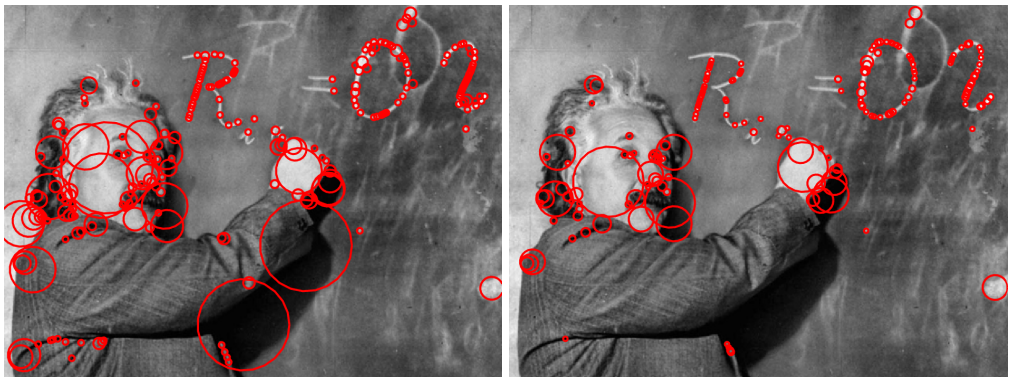
See the blobs for both technique along with the parameter values used in Figures 29, 30, 31 and 32. The timing results are shown in Table 1.

Full code is in Appendix A.1.

From the results shown, we can conclude that downsampling and than interpolating the results to original size is much more time-wise

(a) Increasing Kernel Size (b) Downsampling the Image

Figure 29: Blob Detection on butterfly.jpg with both LoG filtering techniques, at initial $\sigma = 1.6$ and Threshold = 0.009

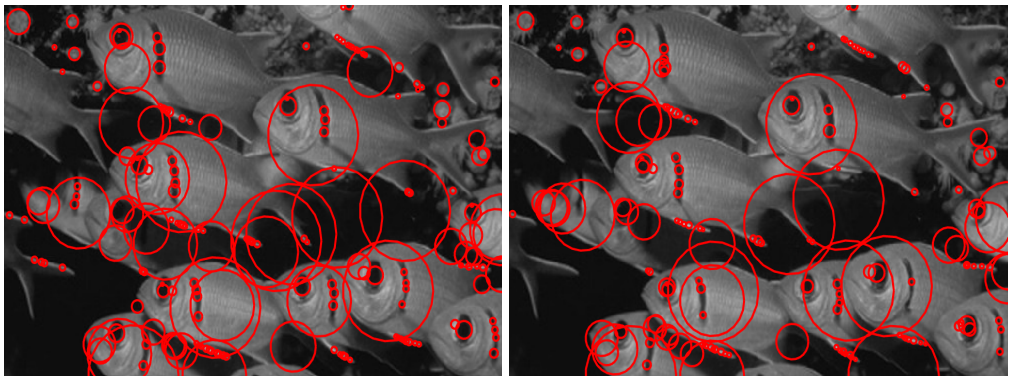


(a) Increasing Kernel Size (b) Downsampling the Image

Figure 30: Blob Detection on einstein.jpg with both LoG filtering techniques, at initial $\sigma = 1.6$ and Threshold = 0.02

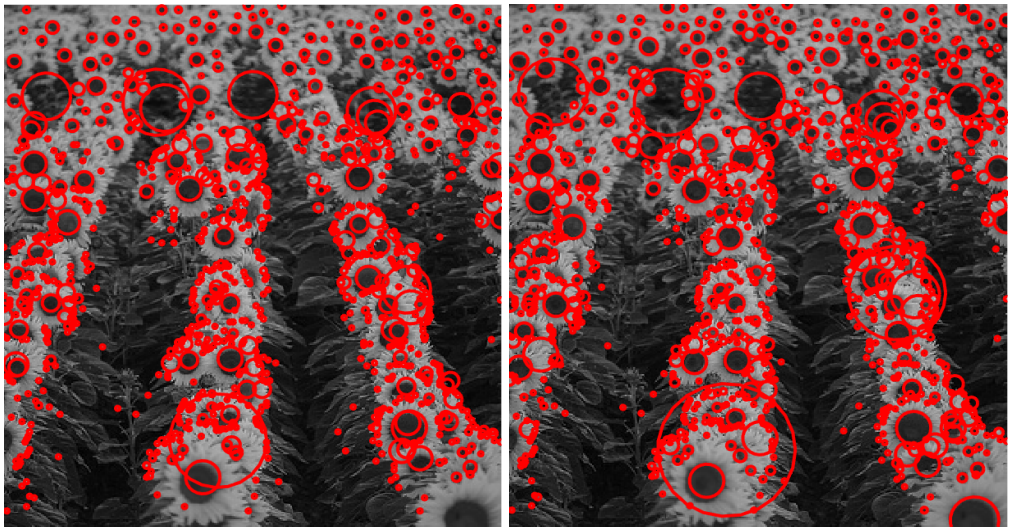
efficient over increasing the kernel size.

Naturally, the convolution is faster when we do it for smaller resolutions. But we might lose some tiny blob details in doing that.



(a) Increasing Kernel Size (b) Downsampling the Image

Figure 31: Blob Detection on fishes.jpg with both LoG filtering techniques, at initial $\sigma = 1.0$ and Threshold = 0.019



(a) Increasing Kernel Size (b) Downsampling the Image

Figure 32: Blob Detection on sunflowers.jpg with both LoG filtering techniques, at initial $\sigma = 1.0$ and Threshold = 0.019

Method →	Increasing Filter Size		Downsampling the Image	
Elapsed Time→ Name↓	Whole Process	Computing Blob Response	Whole Process	Computing Blob Response
butterfly.jpg	102.383 s	98.629974 s	2.037 s	0.268332 s
einstein.jpg	134.837 s	132.56010 s	1.692 s	0.318100 s
fishes.jpg	85.5053 s	82.286573 s	1.206 s	0.194998 s
sunflowers.jpg	61.0202 s	57.543230 s	1.279 s	0.264452 s

Table 1: Elapsed Times for each process using both techniques: Increasing Kernel Size and Downsampling the Image, for whole process and only the blob response computation

3.2 Homography Transformation in RANSAC

We will get base points and transformed points just like before but now we will present them in this form:

$$\begin{bmatrix} x_i & y_i & 1 & 0 & 0 & 0 & -x'_i x_i & -x'_i y_i & -x'_i \\ 0 & 0 & 0 & x_i & y_i & 1 & -y'_i x_i & -y'_i y_i & -y'_i \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The code for finding the h parameters is as follows:

```
function [H] = computeHomography(x1,y1,x2,y2)
    num = numel(x1);
    H = ones(3);
    eq = zeros(num*2, 8);
    sol = zeros(num*2,1);
    for i =1:num

        pts1 = [x1(i),y1(i),1];
        z = [0, 0, 0];
        rem = (-1 * [x2(i);y2(i)]) * [x1(i),y1(i)];
        sol((i-1)*2+1:(i-1)*2+2) = [x2(i); y2(i)];
        eq ((i-1)*2+1:(i-1)*2+2,:) = [[pts1, z;z, pts1
           ] rem];
    end

H = eq\sol;

H(9) = 1;
H = reshape(H,3,3)';
end
```

Now we have got

$$\begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix}$$

we can compute Transformed points from the Base point and Homography Transform and calculate the error between that and the actual Transformed co-ordinate.

```
function [x2, y2] = applyHomography(H, x1,y1)
    p(1,:) = x1(:);
    p(2,:) = y1(:);
    p(3,:) = 1; % Homogenous co-ordinate

    % Compute Transformed Point
    x2 = H(1,:) * p;
    y2 = H(2,:) * p;
    thd  = H(3,:) * p;

    x2 = x2 ./ thd;
    y2 = y2 ./ thd;
end
```

Then we choose the best model which has the least error just like we did in Affine Transform case.

The results are as shown below. See Figures 33, 34, 35, 36, 37, 38, 39 and 40 & their corresponding Homography Transforms.

Full code is in Appendix B.1.

Hill.jpg

. Maximum Inliers 413
Homography Transformation is:

0.9967	0.1063	136.7116
-0.0368	1.0539	-23.7154
-0.0001	0.0002	1.0000

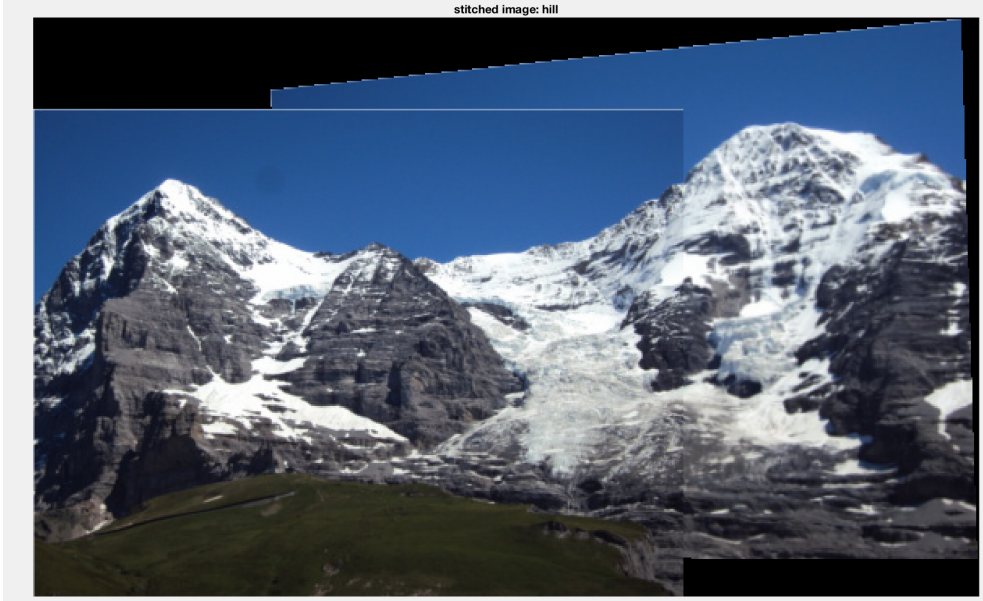


Figure 33: Panoramic Stitching of Hill images with Homography

Figure 34: Panoramic Stitching of Field images with Homography

Field.jpg

Maximum Inliers 116
Homography Transformation is:

0.9068	0.1104	255.2360
-0.0415	1.0482	8.7926
-0.0003	0.0003	1.0000

Figure 35: Panoramic Stitching of Ledge images with Homography

Ledge.jpg

Maximum Inliers 169
Homography Transformation is:

1.0004	0.2301	110.6853
-0.0265	1.2457	-170.1197
-0.0005	0.0009	1.0000

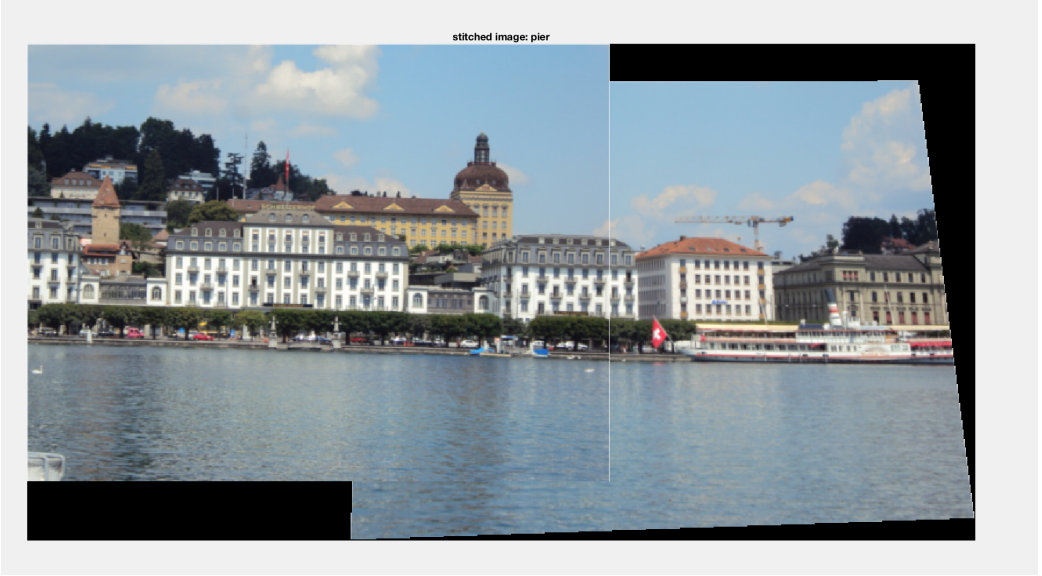


Figure 36: Panoramic Stitching of Pier images with Homography

Pier.jpg

Maximum Inliers 275
Homography Transformation is:

1.0908	-0.0536	285.5045
0.0061	0.9795	28.4132
0.0001	-0.0001	1.0000

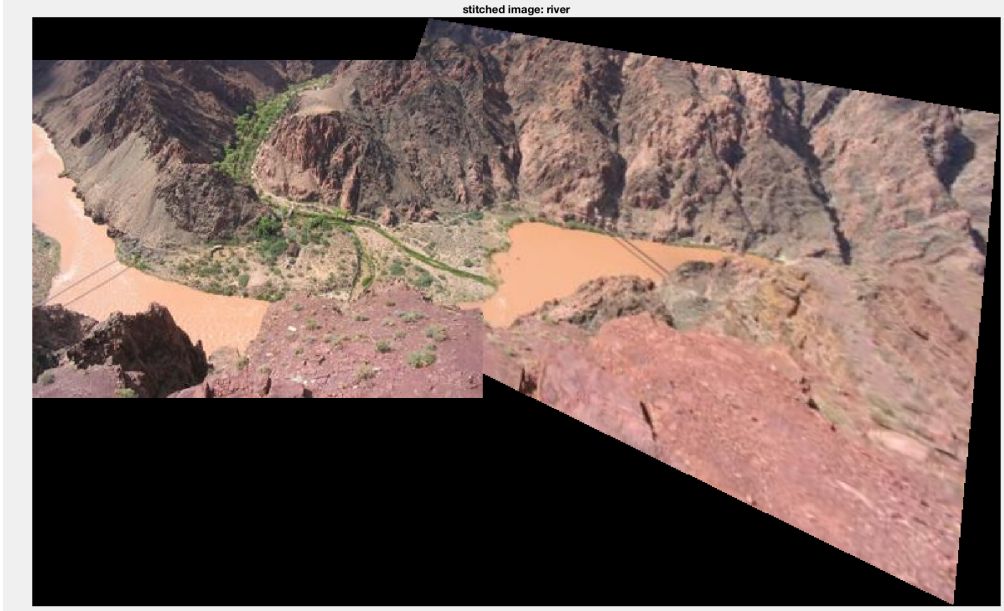


Figure 37: Panoramic Stitching of River images with Homography

River.jpg

Maximum Inliers 87
Homography Transformation is:
0.8239 -0.1951 324.8764
0.3001 1.0629 -72.2569
-0.0005 0.0005 1.0000

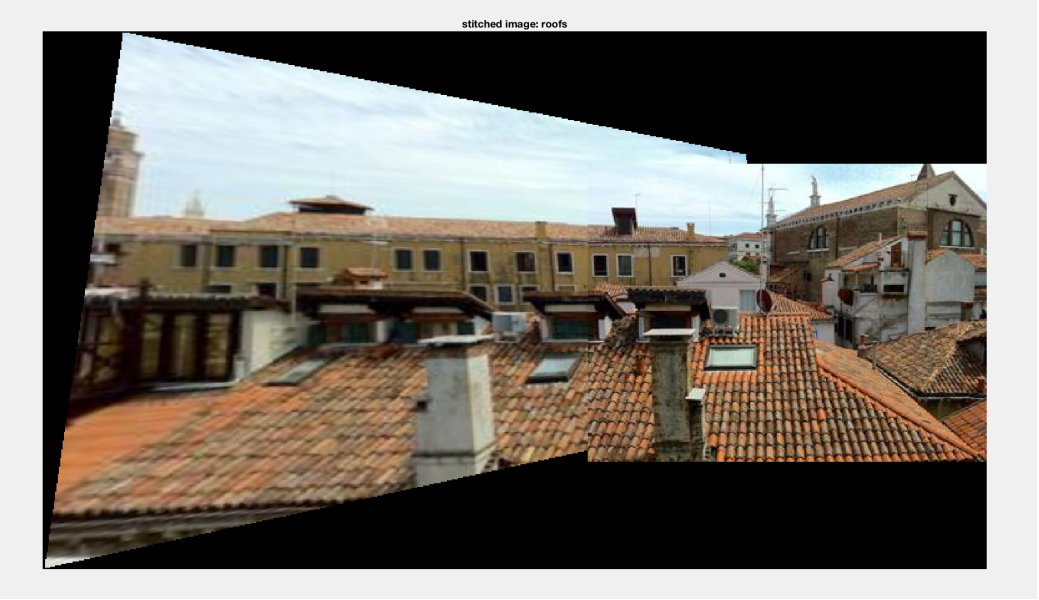


Figure 38: Panoramic Stitching of Roofs images with Homography

Roofs.jpg

Maximum Inliers 215
Homography Transformation is:

2.7722	0.3423	-538.9436
0.4573	2.2519	-183.8534
0.0052	-0.0001	1.0000

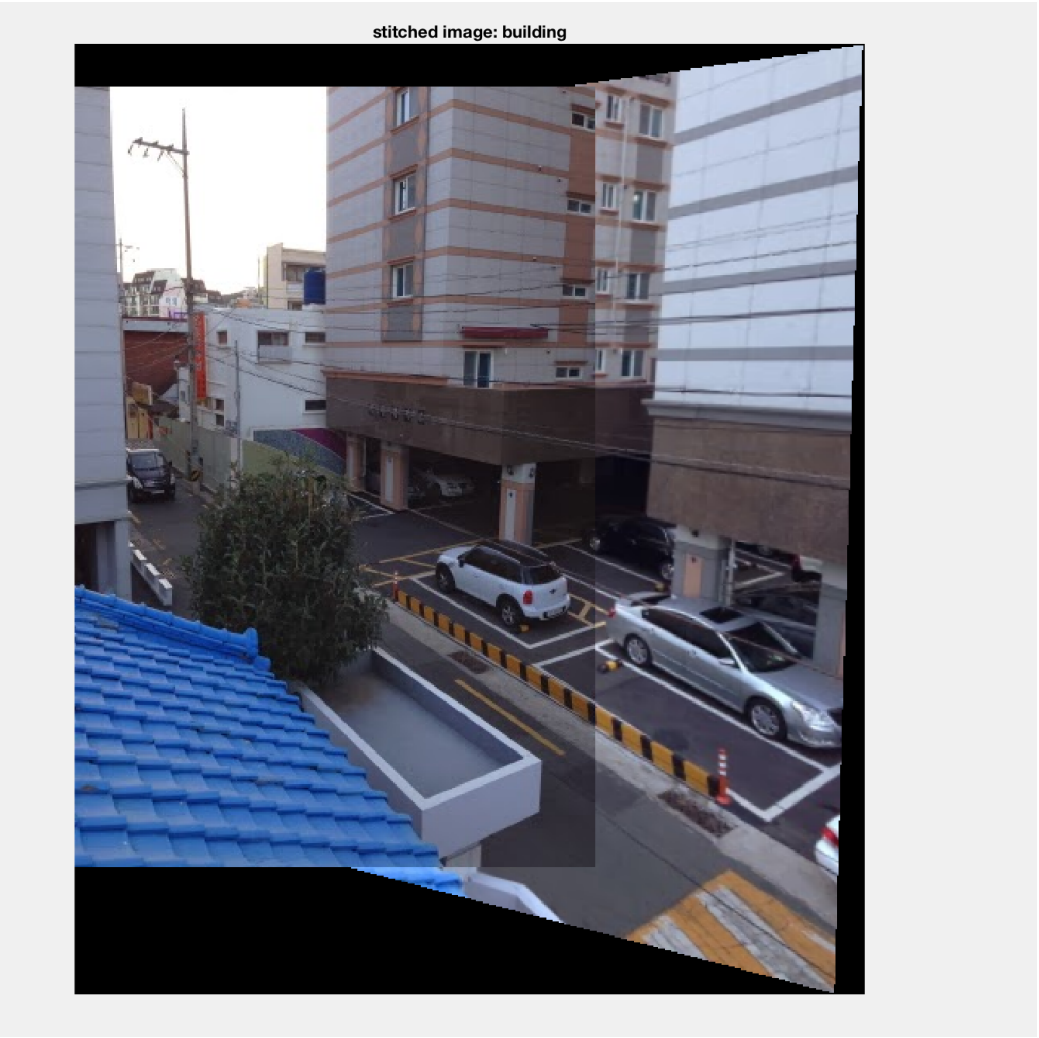


Figure 39: Panoramic Stitching of Building images with Homography

Building.jpg

Maximum Inliers 218
Homography Transformation is:

0.7735	-0.0161	125.7620
-0.0940	0.9434	16.2253
-0.0006	0.0000	1.0000

Figure 40: Panoramic Stitching of Tower images with Homography

Tower.jpg

Maximum Inliers 273
Homography Transformation is:

1.0793	-0.0891	-212.6703
0.0701	1.0257	-35.4482
0.0002	-0.0001	1.0000

References

[1] Blob Detection, https://en.wikipedia.org/wiki/Blob_detection

[2] Subhransu Maji, *Image Alignment* https://www.dropbox.com/s/5flemuwc73wv7o9/lec11\%2B12\%2B13_image_alignment.pdf?dl=0

[3] Richard Szeliski, *Computer Vision: Algorithms and Applications* ISBN: 978-1-848-82934-3

A

Matlab code for Problem 1: Scale-space blob detection

A.1 Implementation of `detectBlobs.m`

```
function blobs = detectBlobs(im, param)
% DETECTBLOBS detects blobs in an image
%   BLOBS = DETECTBLOBSCALEIMAGE(IM, PARAM) detects multi-scale blobs in IM.
%   The method uses the Laplacian of Gaussian filter to find blobs across
%   scale space.
%
% Input:
%   IM - input image
%   PARAM - struct containing the following fields
%       PARAM.SIGMA - sigma of the LoG filter (smallest blob desired)
%       PARAM.INTERVAL - number of intervals in an octave
%       PARAM.THRESHOLD - threshold for blob detection
%       PARAM.DISPLAY - if true then then shows intermediate results
%
% Ouput:
%   BLOBS - n x 4 array with blob in each row in (x, y, radius, score)
%


% Convert image to grayscale and convert it to double [0 1].
if size(im, 3) > 1
    im = rgb2gray(im);
end
if ~isfloat(im)
    im = im2double(im);
end

```

```
27 % If param is not specified use default
28 if nargin < 2
29     param.sigma = 2;
30     param.interval = 12;
31     param.threshold = 1e-2;
32     param.display = false;
33 end
34
35 % dummy blob
36 % blobs = [size(im, 2)/2, size(im, 1)/2, 100, 1.0];
37
38
39 %% Implement these:
40
41 %% Compute the scale space representation
42 sigma       = 1.6;      % Initial Scale
43 k           = sqrt(2);  % Factor by which the scale is
       mulitplied each time
44 sigma_final = power(k,16);      % Last Scale
45 % Dynamically decide the interation levels from the
       first and last scale values and multiplication
       factor
46 n           = ceil((log(sigma_final) - log(sigma))/log
       (k)); % Iterations of Laplacian scale space
47 [h, w]      = size(im);
48 scaleSpace  = zeros(h, w, n);   % [h,w] - Dimensions
       of Image, n - Number of Levels in Scale Space
49
50 % Generate the Laplacian of Gaussian for the First
       Scale Level
51 filt_size   = 2 * ceil(3*sigma) + 1;
52 LoG         = fspecial('log', filt_size , sigma);
53
54 %% Compute the blob response
55 % Increase filter size, keep image the same
56 if 0
57     tic
58     for i = 1:n
59         sigma_next = sigma * k^(i-1); % Increament
              sigma for next level
60         filt_size = 2*ceil(3*sigma_next)+1;  % Compute
```

```
            the filter kernel size in the same way as
            previously
61          LoG       = sigma_next^2 * fspecial('log',
            filt_size, sigma_next);   % Making a custom
             LoG filter with trial and error
62          filter_next = imfilter(im, LoG, 'same', '
            replicate');  % Create a replica of the
            previous filter
63          filter_next = filter_next .^ 2;     % Multiply
             it by itself
64          scaleSpace(:,:,i) = filter_next;    % Store
            filter of each level
65      end
66      toc
67  end
68
69  % Faster version: keep the filter size, downsample the
        image
70  if 1
71      tic
72      imRes = im; % Create a copy of the image
73      for i = 1:n
74          imFiltered = imfilter(imRes, LoG, 'same', '
            replicate'); % Apply the Laplacian of
            Gaussian filter here, the filter is already
             created in the above section
75          imFiltered = imFiltered .^ 2; % Square of the
            LoG
76          scaleSpace(:,:,i) = imresize(imFiltered, size(
            im), 'bicubic'); % Resize the image to
            original size with bicubic interpolation
77          if i < n
78              imRes = imresize(im, 1/(k^i), 'bicubic');
                % Decrease the image size for next
                iteration
79          end
80      end
81      toc
82  end
83
84  %% Perform NMS (spatial and scale space)
```

```
85 % Perform Non-Maximum Suppression for each Scale-Space
       Slice
86 suppression_size = 3; % To make it Maximum Filter
87 max_space = zeros(h, w, n); % NMS output for each
     level
88 for i = 1:n
89     max_space(:,:,i) = ordfilt2(scaleSpace(:,:,i),
         suppression_size^2, ones(suppression_size));
90 end
91
92 % Non-Maximum Suppression between Scales and Threshold
93 % For scales, compare the current one with the
     previous and next ones, choose the maximum of the
     three
94 for i = 1:n
95     max_space(:,:,i) = max(max_space(:,:,max(i-1,1):
         min(i+1,n)),[],3);
96 end
97 % Zero Out All Positions that are not the Local Maxima
       of the score (if the Value if not Greater than all
       its Neighbors)
98 max_space = max_space .* (max_space == scaleSpace);
99
100 x = [];   % X - axis of blob's center
101 y = [];   % Y - axis of blob's center
102 radius = []; % Radius of the blob
103 score = [];
104
105 % For each level
106 for i=1:n
107     % Set a Threshold on the Squared Laplacian
         Response above which to report Region
         Detections
108     [rows, cols, value] = find(max_space(:,:,i).*(
         max_space(:,:,i) >= 0.01));
109     numBlobs = length(rows);
110     % Create Lists of each tuple fields separately
111     if(numBlobs > 0)
112         radii =  sigma * k^(i-1) * sqrt(2);
113         radii = repmat(radii, numBlobs, 1);
114         x = [x; rows];
```

```
115             y = [y; cols];
116             score = [score;value];
117             radius = [radius; radii];
118         end
119     end
120 
121     blobs = [y, x, radius, score];
```

B
Matlab code for Problem 2: Image Stitching

B.1 Implementation of `computeMatches.m`

```
function matches = computeMatches(f1,f2)
% COMPUTEMATCHES Match two sets of SIFT features f1 and f2
% This code is part of:
%
%   CMPSCI 670: Computer Vision, Fall 2019
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji
%
%   Mini project 4

% Implement this
f1_size = size(f1, 1);
f2_size = size(f2, 1);

%% Computing Matches using SSD
matches = zeros(f1_size, 1);
% Start Stopwatch Timer
tic
if 0
    fprintf('Computing matching using SSD: \n');
    % For each feature from image 1
    for i = 1 : f1_size
        % Find its nearest match image 2
        bestMatch = inf;
        for j = 1 : f2_size
            % SSD
            match = sum(sum((f1(i,:)-f2(j,:)).^2));
            % Store the lowest SSD feature
            if(match<bestMatch)
                matches(i) = j;
                bestMatch = match;
            end
        end
    end
```

```
35 % Stop Stopwatch Timer
36 toc
37 end
38
39 %% Computing Matches using ratio
40 if 1
41     % For each Descriptor in the First Image, Select
          its Match to Second Image
42     for i = 1 : f1_size
43         % The most likely match
44         bestMatch = inf;
45         % The next most likely match after the best
              one
46         secondMatch = inf;
47         index = inf;
48         for j = 1:f2_size
49             % Compute SSD
50             match = sum(sum((f1(i,:)-f2(j,:)).^2));
51             % compare it with the best match
52             if(match < bestMatch)
53                 secondMatch = bestMatch;
54                 bestMatch = match;
55                 index = j;
56             % compare it with the second best match
57             elseif(match < secondMatch && match ~=
                  bestMatch)
58                     secondMatch = match;
59             end
60         end
61         % Now we have determined best and second best
              match
62
63         % Computer Ratio
64         ratio = bestMatch/secondMatch;
65         % Compare it with a threshold
66         if(~isequal(index,inf) && ratio < 0.8)
67             matches(i)= index;
68         else
69             matches(i) = 0;
70         end
71     end
```

```
72 end
```

B.2 Implementation of `ransac.m`

```
1  function [inliers, transf] = ransac(matches, blobs1,
       blobs2)
2  % This code is part of:
3  %
4  %   CMPSCI 670: Computer Vision, Fall 2019
5  %   University of Massachusetts, Amherst
6  %   Instructor: Subhransu Maji
7  %
8  %   Homework 4
9
10 % implement this
11
12 %% Get all the non-empty feature matches
13 [inputs, cols, bases] = find(matches);
14 % Reset max inliers counter
15 maxInliers = 0;
16 % Store the original indices of matches so that
       removing the 0 index entries doesn't change the
       actual indices
17 originalIndices = find(matches);
18 % Get the non-zero entries size
19 inputs_size = size(inputs,1);
20 if nargin< 4
21     method = 'affine';
22 end
23
24 if strcmp(method,'affine')
25     % This structure stores current point co-ordinates
26     inputPoints = zeros(inputs_size,2);
27     % This structure stores the points whose co-
           ordinates are getting transformed
28     basePoints = zeros(inputs_size,2);
29     % Number of points whose distance from the line is
           less than some constant threshold
30     threshold = 4;
31     iterations = 35;
32     % Store the best affine transformation (with the
```

```
        least error)
33      best_model = zeros(2,3);
34      % Store the points who are inlier
35      inliersInfo = zeros(inputs_size,2);
36      % Store the affine transformation for a particular
            iteration
37      transInfo = zeros(2,3,inputs_size);
38
39      % Store all features and locations in inputPoints
            for 1st image and in basePoints for 2nd image
40      for ind= 1:inputs_size
41          basePoints(ind,1) = blobs1(bases(ind),1);
42          basePoints(ind,2) = blobs1(bases(ind),2);
43          inputPoints(ind,1) = blobs2(inputs(ind),1);
44          inputPoints(ind,2) = blobs2(inputs(ind),2);
45      end
46
47      % Try RANSAC for 'iterations' number of times
48      for itr = 1:iterations
49          % Affine Transform for the current iteration
50          T = computeAffine(inputPoints,basePoints);
51          % Reshape the transform matrix in [3x3] for
52          % tranformed_points = tranform x base_points
53          trans = [reshape(T,[3,2])';0,0,1];
54          % Reshape base_points to [3x1] for
55          % tranformed_points = tranform x base_points
56          inputPt = [inputPoints';ones(1,inputs_size)];
57          % Compute tranformed_points = tranform x
                base_points
58          inputptF =  trans*inputPt;
59          % Take the x-y co-ordinates of every
                tranformed point
60          calPoints = inputptF(1:2,:);
61
62          % Calculate Error between Expected Tranformed
                point co-ordinates and actual Transformed
                points
63          dist = calculateError(calPoints,basePoints');
64          % Only take the points whose distance is lower
                than threshold
65          [rows,cols,error] = find(dist.*(dist <
```

```
            threshold));
66          % Count the number of inliers
67          inliersCount = size(cols,2);
68          total_error = sum(error,2);
69
70          % Find the best model by inlier count and
               error values
71          if inliersCount > round(0.10*inputs_size) &&
               inliersCount >= maxInliers && inliersCount
               ~= inputs_size
72
73              best_model = reshape(T,[3,2])';
74              if(inliersCount == maxInliers)
75                  maxIndex = find(inliersInfo(:,1) ==
                       maxInliers);
76                  if(total_error < inliersInfo(maxIndex
                       ,2))
77                      best_model = reshape(T,[3,2])';
78                  end
79              end
80              inliersC = cols';
81              maxInliers= inliersCount;
82              inliersInfo(itr,1)=maxInliers ;
83              inliersInfo(itr,2)= total_error;
84              transInfo(:,:,itr) = best_model;
85          end
86      end
87      fprintf('Maximum Inliers %d and Msize is %d\n',
           maxInliers,inputs_size);
88      transf = [inv(best_model(:,1:2)),-1*best_model
           (:,3)];
89      inliers = originalIndices(inliersC);
90      disp('Affine Transformation is:')
91      disp(transf)
92  end
93
94  if strcmp(method, 'homography')
95      % X co-ordinates of base points
96      xBase = zeros(inputs_size,1);
97      % Y co-ordinates of base points
98      yBase = zeros(inputs_size,1);
```

```
99      % X co-ordinates of transformed points
100     xInput = zeros(inputs_size,1);
101     % Y co-ordinates of transformed points
102     yInput = zeros(inputs_size,1);
103
104     % Blobs of Image 1 are Base Points
105     for i= 1:inputs_size
106         xBase(i,1) = blobs1(inputs(i),1);
107         yBase(i,1) = blobs1(inputs(i),2);
108     end
109
110     % Blobs of Image 2 are Transformed Points
111     for i= 1:inputs_size
112         xInput(i,1) = blobs2(bases(i),1);
113         yInput(i,1) = blobs2(bases(i),2);
114     end
115
116
117     for i = 1:10000
118
119         randBase = zeros(4,2);
120         randInput = zeros(4,2);
121
122         % Picking up random base points for first step
              of RANSAC
123         for j = 1:4
124             index = ceil(rand*(inputs_size-1))+1;
125             randBase(j,1) = xBase(index);
126             randBase(j,2) = yBase(index);
127             randInput(j,1) = xInput(index);
128             randInput(j,2) = yInput(index);
129
130         end
131
132         h = computeHomography(randInput(:,1),randInput
              (:,2),randBase(:,1),randBase(:,2));
133
134         [x,y] = applyHomography(inv(h),xBase,yBase);
135         total = 0;
136         for j = 1:numel(x)
137             sigma = 100;
```

```
138             error = sum(sum(([xInput(j) yInput(j)] - [
                    x(j) y(j)]).^2));
139
140             if  error < sigma
141                 total = total + 1;
142             end
143
144         end
145
146         if total > maxInliers
147
148             maxInliers = total;
149             H = h;
150
151         end
152     end
153     transf = H;
154     inliers = [];
155 end
156 end
157
158 function [x2, y2] = applyHomography(H, x1,y1)
159     p(1,:) = x1(:);
160     p(2,:) = y1(:);
161     p(3,:) = 1;
162
163     x2 = H(1,:) * p;
164     y2 = H(2,:) * p;
165     thd  = H(3,:) * p;
166
167     x2 = x2 ./ thd;
168     y2 = y2 ./ thd;
169 end
170
171 function [H] = computeHomography(x1,y1,x2,y2)
172     num = numel(x1);
173     H = ones(3);
174     eq = zeros(num*2, 8);
175     sol = zeros(num*2,1);
176     for i =1:num
177
```

```
178         pts1 = [x1(i),y1(i),1];
179         z = [0, 0, 0];
180         rem = (-1 * [x2(i);y2(i)]) * [x1(i),y1(i)];
181         sol((i-1)*2+1:(i-1)*2+2) = [x2(i); y2(i)];
182         eq ((i-1)*2+1:(i-1)*2+2,:) = [[pts1, z;z, pts1
                ] rem];
183     end
184
185 H = eq\sol;
186
187 H(9) = 1;
188 H = reshape(H,3,3)';
189 end
190
191 function [dist]= calculateError(calPoints,FinputPoints
    )
192     dist = sum((calPoints-FinputPoints).^2);
193 end
194
195 function [Transform] = computeAffine(basePoints,
    transPoints)
196     baseP = zeros(6,6); % input points
197     transP = zeros(6,1); % points after transformation
            of input points.
198     for j = 1:3
199         index = ceil(rand*(size(basePoints,1)-1))+1;
200         % Prepare base matrix with points before
              transformation; put them in the position
              shown in affine transform matrix equation
201         baseP(2*j-1,:) = [basePoints(index,1),
              basePoints(index,2),1,0,0,0];
202         baseP(2*j,:) = [0,0,0,basePoints(index,1),
              basePoints(index,2),1];
203         % Set transformed points matrice; put them in
              the position shown in affine transform
              matrix equation
204         transP(2*j-1,:) = transPoints(index,1);
205         transP(2*j,:) = transPoints(index,2);
206     end
207     % Matrix Division
208     Transform = baseP\transP;
```

```
209 end
```