Visual Perception

Lab on Problem-Based Learning: Applications of Invariant Features

Abdullah ABDULAZIZ
Taner Gungor
The University of Girona

Supervisor:
Rafael Garcia and Ricard Prados

1 Objective

- Create a testing dataset intended to verify the invariance of the SIFT descriptor.
- Test Lowe's code with a set of images and produce results about the percentage of correct matches for every pair of images.
- Implement other feature descriptors (SURF, FREAK and BRISK) and compare them with the results obtained using the standard SIFT implementation (using Lowe's code).

2 Generating the testing dataset

The dataset will be generated depending on a synthetic underwater scene, from this scene we will generate three sequences correspond to different movements of the camera. The original synthetic image is shown in figure 1:

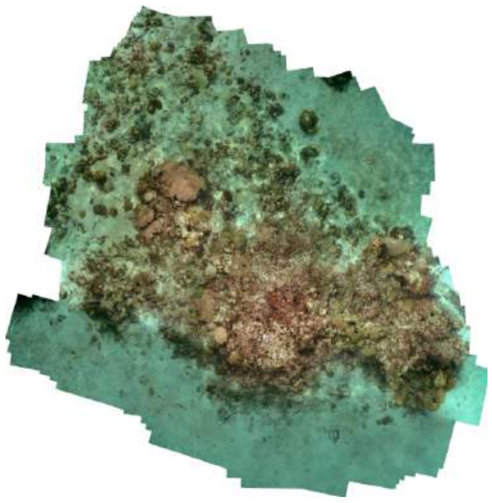


Figure 1: The original synthetic underwater scene

The original image is very big in size so we will crop it in the center to size 2000x2000. This image *Image_small* is shown in fig 2:

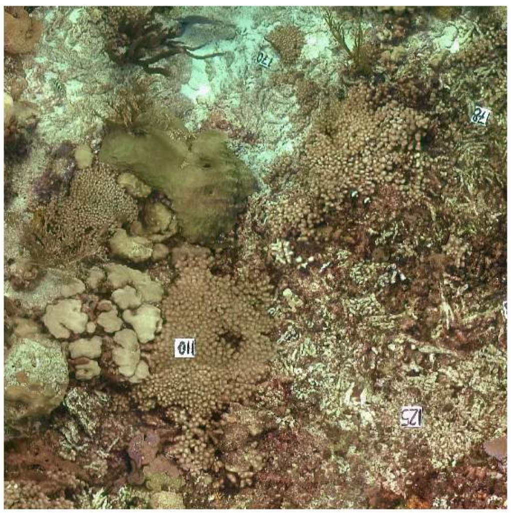


Figure 2: The smaller image of size 2000x2000

Now we will create three sequences of images where each one represents different motion of the camera (rotation, zooming and tilting or projective transformation). Since we know the the homography matrices for the transformations, we will use them to evaluate the invariance performance of different features descriptors.
A reference image *Image_00a.png* will be used for all the sequences. In order to generate this image, we crop *Image_small* in the center to size 750x500. The Matlab code for generating this image is:

```
% Creating the reference image of size 750 x 500 in the center of the
% smaller image
[m,n,c] = size(im1);
row_c = floor(m/2);
col_c = floor(n/2);
rect1 = [500,750];
a1 =  floor(col_c - rect1(2)/2)+1;
a2 =  floor(row_c - rect1(1)/2)+1;
a3 = 750-1;
a4 = 500-1;
im2 =  imcrop(im1, [a1 a2 a3 a4]);
```

As can be seen, the first problem we faced is that *imcrop* crops a rectangle starting from the upper left corner. So the position given to *imcrop* should not be the center of the image but the center shifted by half the size of the rectangle. The reference image is shown in fig 3:

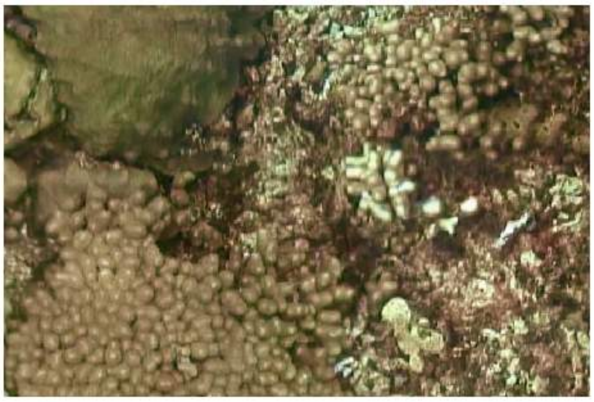


Figure 3: The reference image *Image_00a.png*

2.1 SEQUENCE 1 (projective transformation):

This sequence contains 16 images. Initially, the optical axis of the camera is orthogonal to the observed plane. By tilting the camera from each of the four directions (up, bottom, left, right), we take four images per direction by increasing the tilt angle of the camera.
To generate this sequence, we created a function called *projective*. This function takes three images as input; the reference image of size 750x500, *imx* of size 950x500 and *imy* of size 750x700 and returns the homography matrices.
First of all, the function will create the folder of the sequence (if it is not already existed) and save in this folder the reference image *Image_00a.png* and the homography structure *Sequence1Homographies.mat* that contains the 16 homography matrices of the projective transformation correspond to the 16 images.

```
% Creating the folder that contains the images and the homography matrices
path = [pwd '/SEQUENCE1'];
if exist(path) == 0
    mkdir(path);
end
Sequence1Homographies = struct();
save([path '/Sequence1Homographies.mat']);

% Save the reference image
imwrite(im, [path '/Image_' num2str(00) 'a' '.png']);
```

Looking at figure 4 (tilting the camera from up) that illustrates how to calculate the homography matrix for the projective transformation:

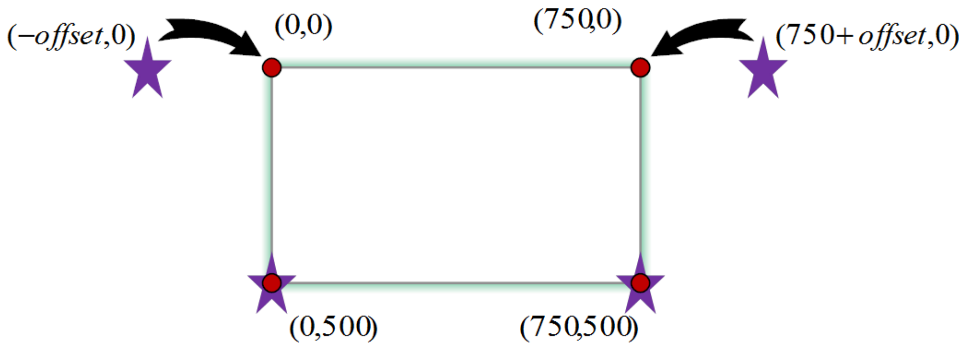


Figure 4: Tilting the camera from up

The homography matrix will bring the star points toward the circle points (compressing the image). We used the way described in Ricard Prados and Rafael Garcia paper for computing the homography matrix:

$$\mathbf{H} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & 1 \end{bmatrix}$$

Firstly, we build the matrix A:

$$\mathbf{A}= \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ & & & & \vdots & & & & \\ x_N & y_N & 1 & 0 & 0 & 0 & -x'_Nx_N & -x'_Ny_N & -x'_N \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_Nx_N & -y'_Ny_N & -y'_N \end{bmatrix}$$

Where we have 8 rows in A, two rows for each point. For instance, when tilting the camera from up (figure 4) x,y are the star points and x',y' are the circle points. After computing the matrix A, by taking the singular value decomposition of it:

$$A = UDV_T$$

The linear solution of the 9x1 unit homography vector $\hat{h}$ is the eigenvector associated with the smallest eigenvalue: $\hat{h} = v_9$. h can be determined by scaling $h = \hat{h}/\hat{h}_9$.
The Matlab code that computes this homography matrix is:

```
    for i = 1 : 4
        A(2*i-1,:) = [p1(i,1) p1(i,2) 1 0 0 0 -p2(i,1)*p1(i,1) -p2(i,1)*p1(i,2) -p2(i,1)
    ];
        A(2*i,:) = [0 0 0 p1(i,1) p1(i,2) 1 -p2(i,2)*p1(i,1) -p2(i,2)*p1(i,2) -p2(i,2)];
    end

    [~,~,V] = svd(A);

    h = V(:,9)/V(9,9);

    H = [h(1) h(2) h(3);h(4) h(5) h(6);h(7) h(8) h(9)];

    % Creating the homography matrix
    Sequence1Homographies(k).H = H;
```

Actually, this homography matrix can only be used for matching the points but we can not use it for building the images cos *imwarp* function does not understand the negative coordinates.
To make all the coordinates positive, we will use a bigger image of the same height (for tilting from up and down) and the same width (for tilting from right and left) and shift all the points to make all the coordinates positive. The following figure (figure 5) shows the case of tilting the camera from up with angle equivalent to 100 pixels:

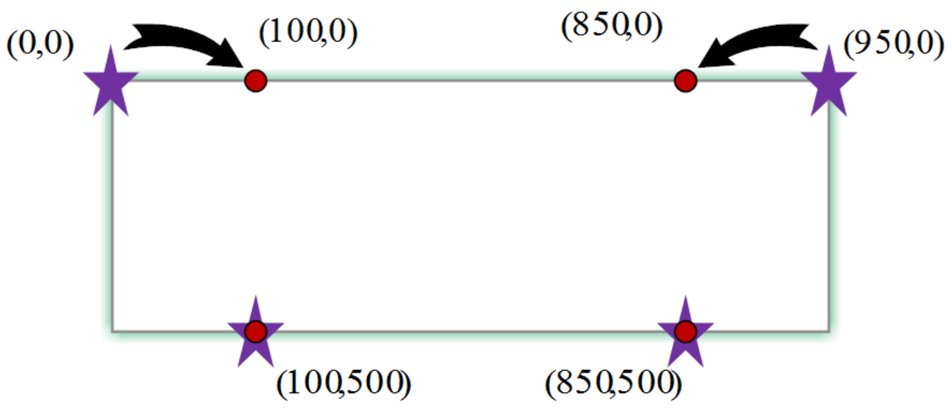


Figure 5: Tilting the camera from up with 100 pixels

Now we compute again the homography matrices in the same SVD way described previously but using the positive coordinates. Then use this homography matrices in *imwarp* for building the images. Finally, we crop the images in the center to size 750x500.
Figure 6 shows *image_04a* before cropping that corresponds to tilting the camera from up with angle equivalent to 100 pixels:

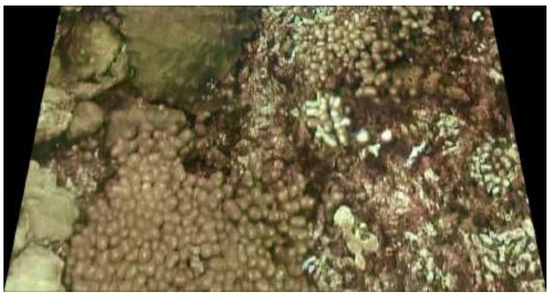


Figure 6: *image_04a* before cropping

Figure 7 shows *image_04a* after cropping in the center to size 750x500:

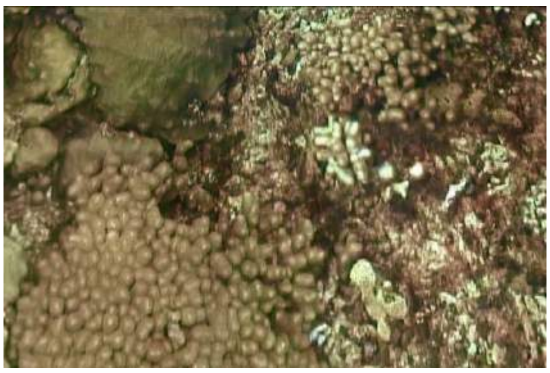


Figure 7: *image_04a* after cropping

Now we will create four versions (a), (b), (c) and (d) for each image and save them in the folder; where (a) corresponds to the image without noise, (b) indicates that the image has additive 0-mean Gaussian noise with standard deviation of 3 gray values, (c) means that the image has additive 0-mean Gaussian noise with standard deviation of 6 grayscale values, and (d) corresponds to a noise with a standard deviation of 18 grayscale values. The following Matlab code is used to create the four versions of each image:

```
    % Creating the set (a) of images without noise
    im_a =  imcrop(B, [a1 a2 a3 a4]);
    imwrite(im_a, [path '/Image_' num2str(k) 'a' '.png']);

    % Creating the set (b) of images that has additive 0-mean Gaussian
    % noise with standard deviation of 3 grayscale values,
    im_b = imnoise(im_a,'gaussian',0,(3/255)^2);
    imwrite(im_b, [path '/Image_' num2str(k) 'b' '.png']);

    % Creating the set (c) of images that has additive 0-mean Gaussian
    % noise with standard deviation of 6 grayscale values,
    im_c = imnoise(im_a,'gaussian',0,(6/255)^2);
    imwrite(im_c, [path '/Image_' num2str(k) 'c' '.png']);

    % Creating the set (d) of images that has additive 0-mean Gaussian
    % noise with standard deviation of 18 grayscale values,
    im_d = imnoise(im_a,'gaussian',0,(18/255)^2);
    imwrite(im_d, [path '/Image_' num2str(k) 'd' '.png']);
```

We should mention that *imnoise* takes the variance so we pass the square of the desired standard deviation. Also we divide by 255 cos *imwrap* converts the image to double [0 1].
The following figures show the four versions of one of the images (for instance image (4)). Figure 8 shows on the left *image_04a* (without noise) and on the right *image_04b* (with Gaussian noise of 0 mean and 3 grayscale values standard deviation):

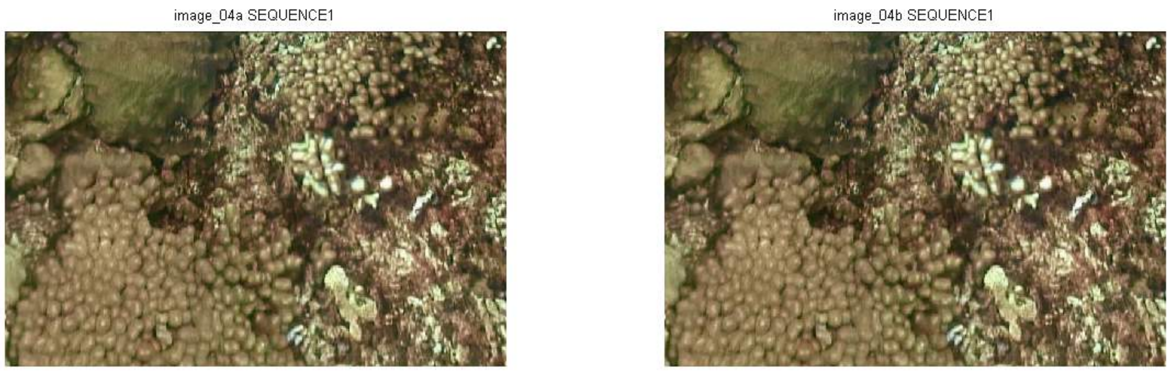


Figure 8: Versions (a) and (b) of image (4)

Figure 9 shows on the left *image_04c* (with Gaussian noise of 0 mean and 6 grayscale values standard deviation) and on the right *image_04d* (with Gaussian noise of 0 mean and 18 grayscale values standard deviation):

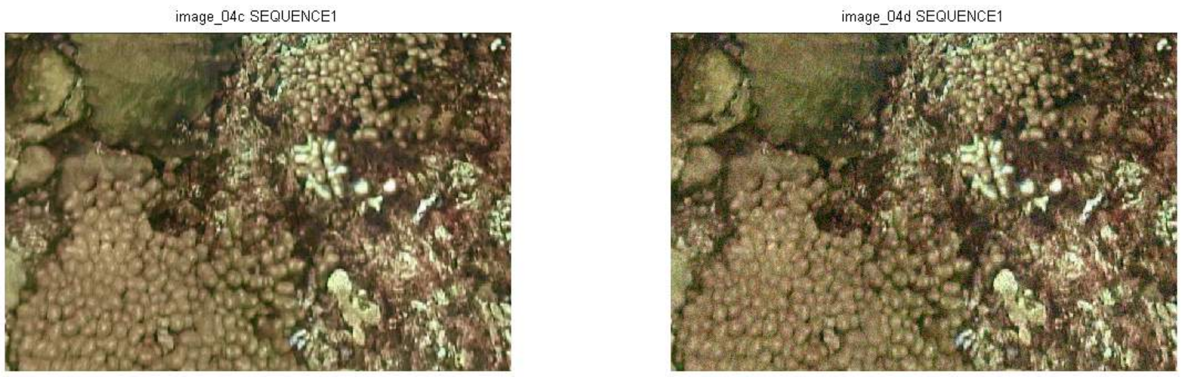


Figure 9: Versions (c) and (d) of image (4)

The following example will demonstrate whether our homography matrices are correct or not. Given a point on the reference image, by multiplying this point with one of the homography matrices (for instance 4) we should get the corresponding point on image (4). We should mention here that we have to normalize the obtained point $p\_04 = p\_04/p\_04(3)$:

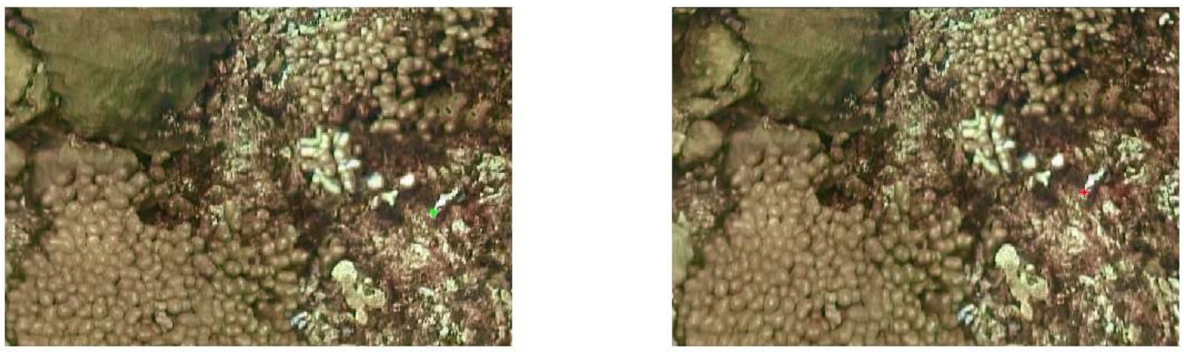


Figure 10: Finding the corresponding point in the projective transformation

It is obvious from the previous figure that the two points are exactly in the same position so we can conclude that our homography matrices are correct and can be used for evaluating the descriptors.

2.2 SEQUENCE 2 (zoom):

This sequence contains 9 images correspond to doing a zoom of the scene from 110% up to a 150% in increments of 5%.
To generate this sequence, we created a function called *zoom*. This function takes two images as input; the smaller image *Image_small* of size 2000x2000, the reference image of size 750x500, and returns the homography matrices.
First of all, the function will create the folder of the sequence (if it is not already existed) and save in this folder the reference image *Image_00a.png* and the homography structure *Sequence2Homographies.mat* that contains the 9 homography matrices of the zooming transformation correspond to the 9 images. To create the images, we used *imresize* function thet takes the *Image_small* of size 2000x2000 and the zooming factor (for instance 1.1 for the first image) and returns the zoomed image. Figure 11 shows *image_04a* before cropping that corresponds to zooming the image by 1.25%:

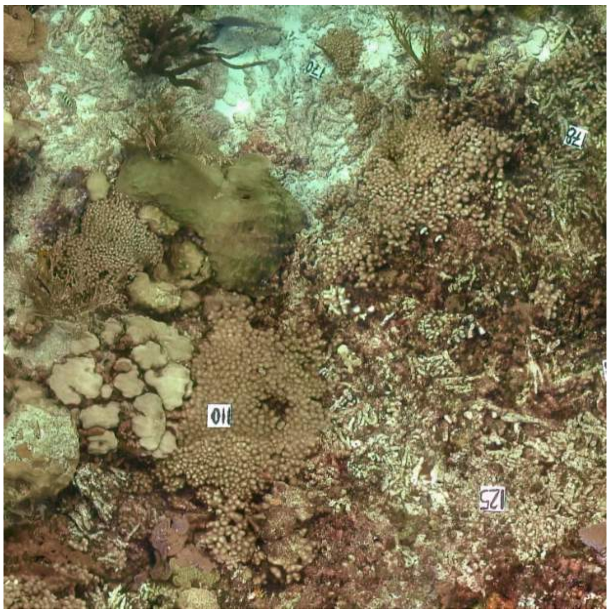


Figure 11: *image_04a* before cropping

Figure 12 shows *image_04a* after cropping in the center to size 750x500:

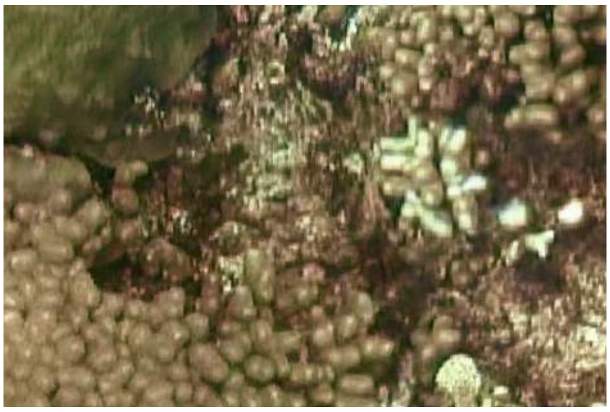


Figure 12: *image_04a* after cropping

The following figures show the four versions of one of the images (for instance image (4)). Figure 13 shows on the left *image_04a* (without noise) and on the right *image_04b* (with Gaussian noise of 0 mean and 3 grayscale values standard deviation):

Figure 13: Versions (a) and (b) of image (4)

Figure 14 shows on the left *image_04c* (with Gaussian noise of 0 mean and 6 grayscale values standard deviation) and on the right *image_04d* (with Gaussian noise of 0 mean and 18 grayscale values standard deviation):

Figure 14: Versions (c) and (d) of image (4)

To create the homography matrices, we have to shift the reference of the image to the center, apply the zooming transformation and then return the reference of the image to the upper left corner. That is because *imresize* zoom the image with respect to the center.

```
    % Creating the corresponding homography matrix:
    center_translation = [1 0 -750/2; 0 1 -500/2; 0 0 1];
    zooming = [zoom 0 0; 0 zoom 0; 0 0 1];
    Sequence2Homographies(i).H = inv(center_translation) * zooming * center_translation;
```

The following example will demonstrate whether our homography matrices are correct or not. Given a point on the reference image, by multiplying this point with one of the homography matrices (for instance 4) we should get the corresponding point on image (4):

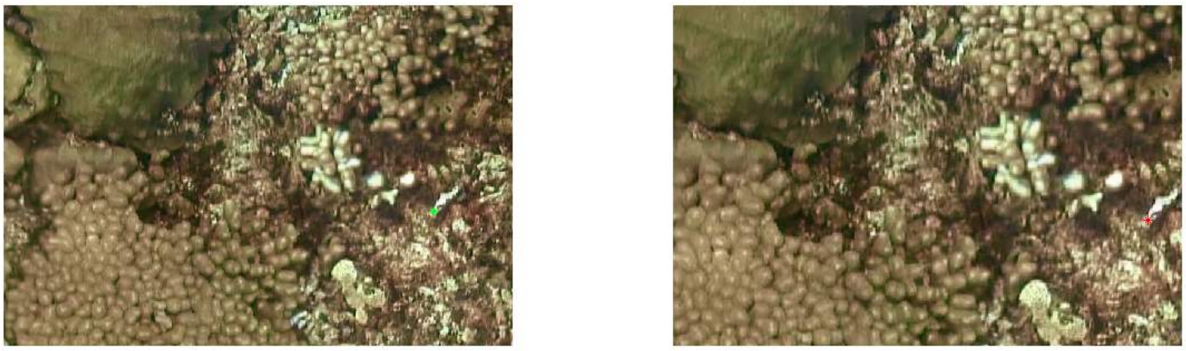


Figure 15: Finding the corresponding point in the zooming transformation

It is clear from the previous figure that the two points are exactly in the same position so we can say that our homography matrices are correct and can be used for evaluating the descriptors.

2.3 SEQUENCE 3 (rotation):

This sequence contains 18 images correspond to rotating the camera around the center of the scene from -45 to 45 degrees.
To generate this sequence, we created a function called *rotation*. This function takes two images as input; the smaller image *Image_small* of size 2000x2000, the reference image of size 750x500, and returns the homography matrices.
First of all, the function will create the folder of the sequence (if it is not already existed) and save in this folder the reference image *Image_00a.png* and the homography structure *Sequence3Homographies.mat* that contains the 18 homography matrices of the rotation transformation correspond to the 18 images.
To create the images, we used *imrotate* function that takes the *Image_small* of size 2000x2000 and the rotation angle (for instance -45 for the first image) and returns the rotated image. Figure 16 shows *image_04a* before cropping that corresponds to rotating the image by -30 degrees:

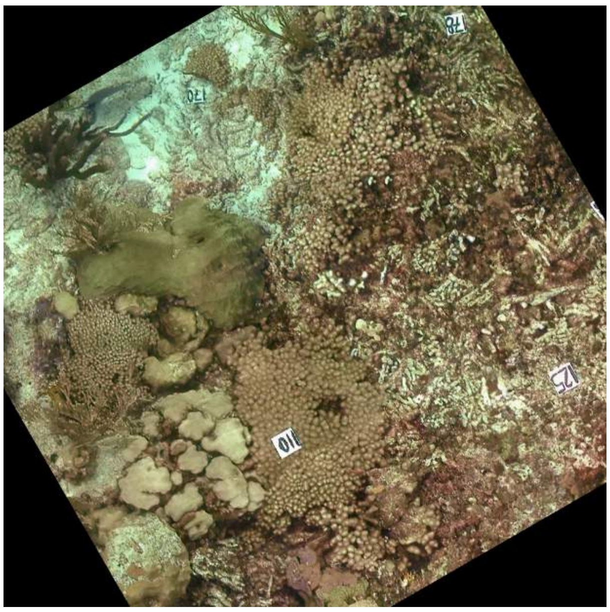


Figure 16: *image_04a* before cropping

Figure 17 shows *image_04a* after cropping in the center to size 750x500:

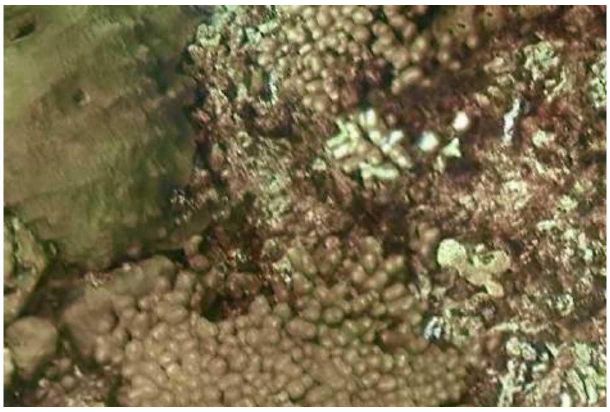


Figure 17: *image_04a* after cropping

The following figures show the four versions of one of the images (for instance image (4)). Figure 18 shows on the left *image_04a* (without noise) and on the right *image_04b* (with Gaussian noise of 0 mean and 3 grayscale values standard deviation):

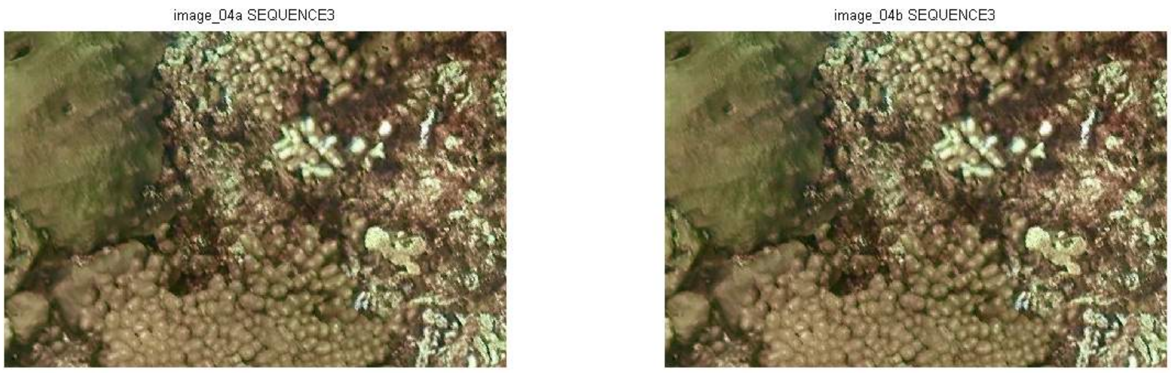


Figure 18: Versions (a) and (b) of image (4)

Figure 19 shows on the left *image_04c* (with Gaussian noise of 0 mean and 6 grayscale values standard deviation) and on the right *image_04d* (with Gaussian noise of 0 mean and 18 grayscale values standard deviation):

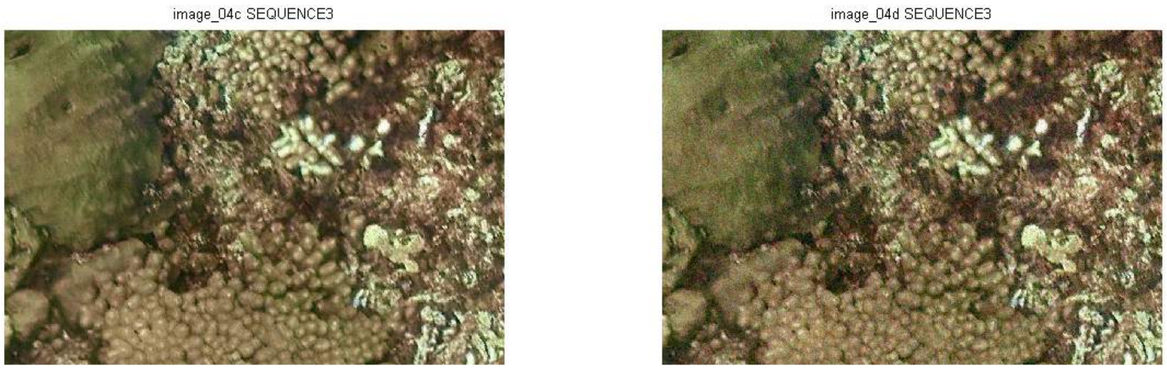


Figure 19: Versions (c) and (d) of image (4)

To create the homography matrices, we have to shift the reference of the image to the center, apply the rotation transformation and then return the reference of the image to the upper left corner. That is because *imrotate* rotating the image with respect to the center.

```
        % Creating the corresponding homography matrix
        center_translation = [1 0 -750/2; 0 1 -500/2; 0 0 1];
        rot = [cos(theta) -sin(theta) 0;sin(theta) cos(theta) 0;0 0 1];
        Sequence3Homographies(k).H = inv(center_translation) * rot * center_translation;
```

The following example will demonstrate whether our homography matrices are correct or not. Given a point on the reference image, by multiplying this point with one of the homography matrices (for instance 4) we should get the corresponding point on image (4):

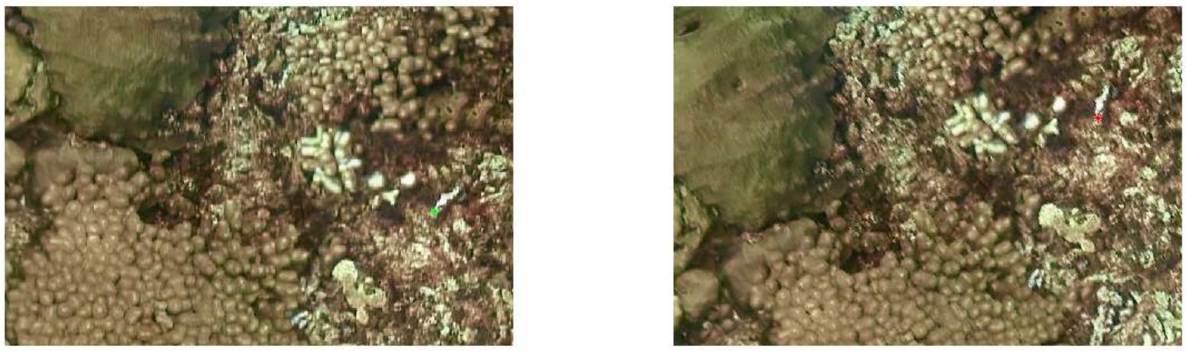


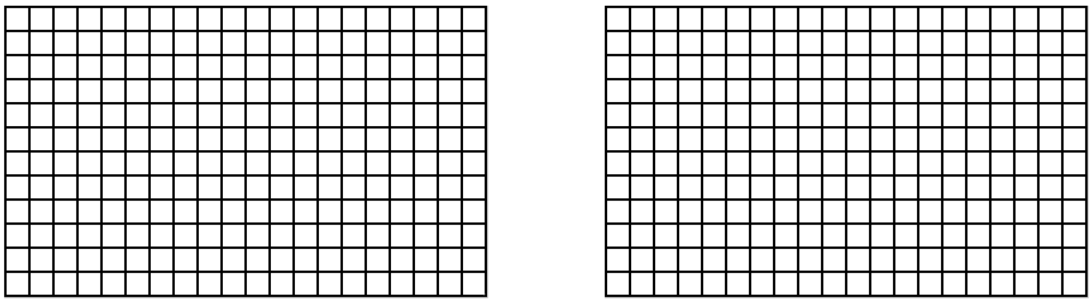
Figure 20: Finding the corresponding point in the rotation transformation

3 Testing the performance of the SIFT descriptor

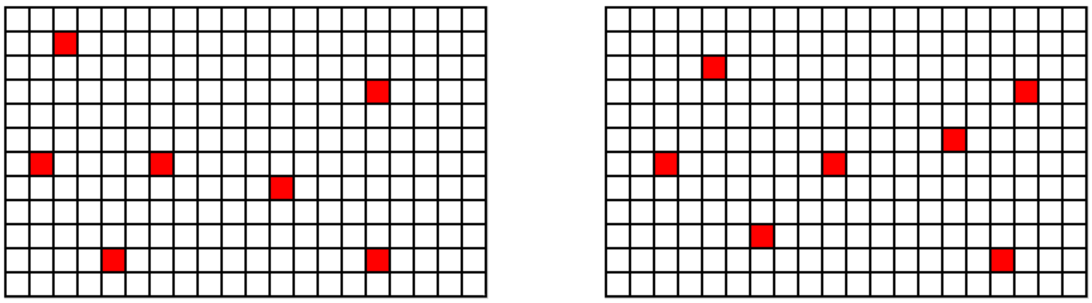
In this chapter, the performance of the SIFT descriptor will be tested by checking the correct matches. After reading the paper of David Lowe, a testing system was implemented to generate performance graphs for every pair of images of all sequences.
Indeed, the evaluation will rely on the homography matrices computed in the previous sections.

The example below will give a clear idea about the operation strategy of the testing system:

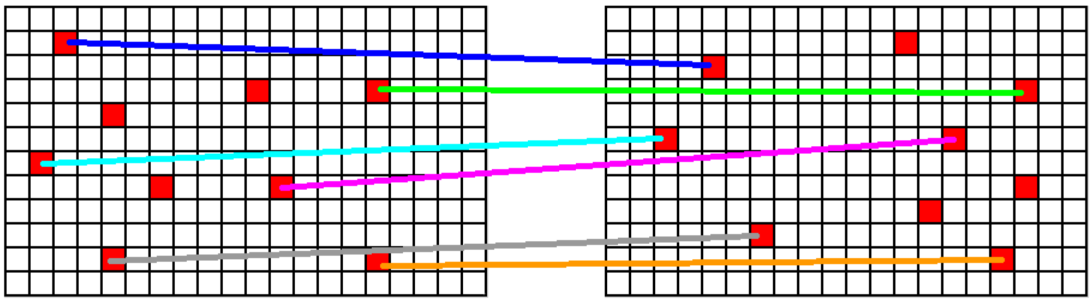
1 - First of all, two images will be loaded: one of them is the original image, the other one is one of three sequences with rotation, zooming or projective transformation.



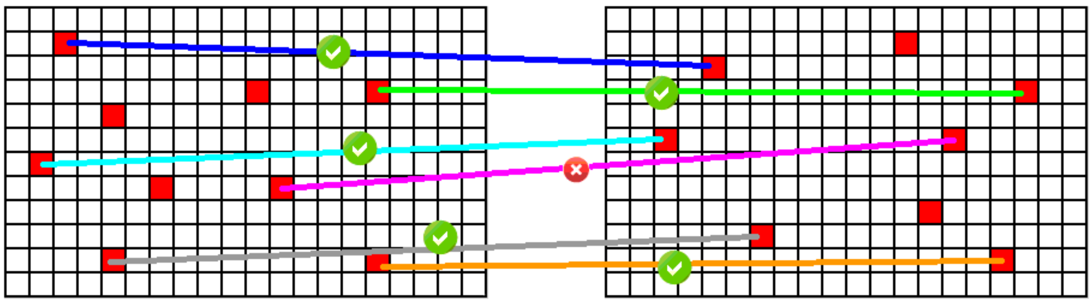
2 - Find the key point descriptors for each image.



3 - After finding the key point descriptors, match all the key points from the first image(Original) with the second one(with rotation, zooming or projective transformation).



4 - Then, matched points in the second image have to be checked by multiplying the location of the key points in the original image with the corresponding homography. If the result is satisfied following condition, it is said that the matching is correct, otherwise it is wrong.



$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \sqrt{(p_1-q_1)^2+(p_2-q_2)^2} \leq 1 \tag{1}$$

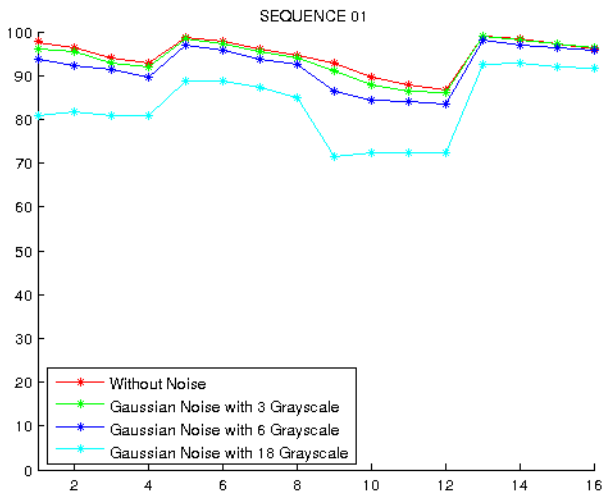
In the **match** method of SIFT, we modified the existing code. While it finds the matched key points between two images. We are checking whether these two points are correspondences or not by calling **is_match** method. When we divide the number of the correct matching to the total number of matching, we can obtain the ratio.

```
.
.
.

% Check if nearest neighbor has angle less than distRatio times 2nd.
if (vals(1) < distRatio * vals(2))
    match(i) = indx(1);

    % [num2str(i) ' - ' num2str(match(i))]
    RATIO = RATIO + is_matched( i , match(i), loc1 , loc2 , homography );

else
    match(i) = 0;
end

.
.
.

num = sum(match > 0);
RATIO = RATIO*100/num;
```

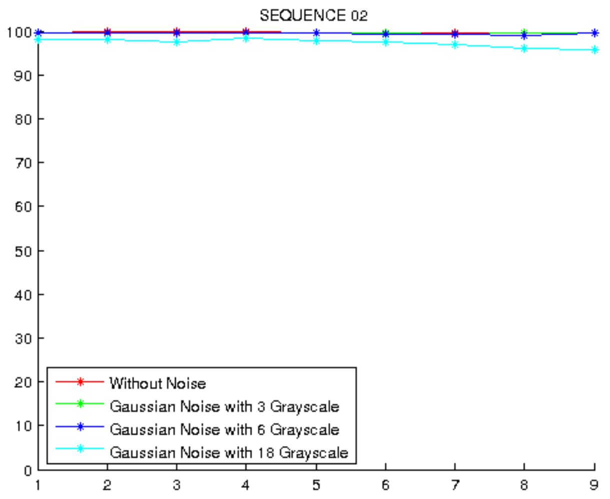
Inside the **is_matched** method we compute the correct point in the second image depending on the homography. If the result is satisfied the condition above, it is said that the matching is correct, otherwise it is wrong. If it is correct return 1, otherwise return 0.

```
function [ result ] = is_matched( ind_1 , ind_2 , loc_1 , loc_2 , homography )
%get_ratio Summary of this function goes here
%   Detailed explanation goes here

    result = 0;

    x_1 = loc_1(ind_1 , 2);
    y_1 = loc_1(ind_1 , 1);

    x_2 = loc_2(ind_2 , 2);
    y_2 = loc_2(ind_2 , 1);

    p_original = [x_1 y_1 1];

    p_final    = homography * p_original';
    p_final    = p_final / p_final(3);
    l_final    = [x_2; y_2; 1];

    % check the distnace between points
    if sqrt((p_final(1) - l_final(1))^2 + (p_final(2) - l_final(2))^2) <= 1.0

        result = 1;

    end

end
```

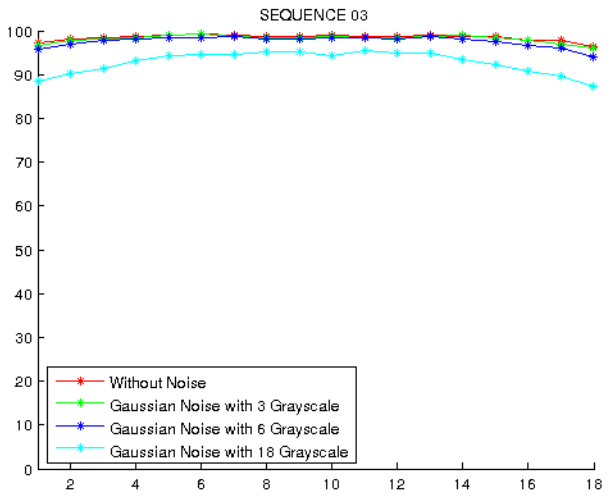
The following figure shows that the performance evaluation of SIFT for each sequence and each version. As you can see from the graph, it has quite high correct matching ratio. But the problem is that computation time of SIFT is extremely high. But, it is robust to match the key points in different images.



(a)



(b)



(c)

Figure 21: (a) SIFT Results for sequence 1, (b) SIFT Results for sequence 2, (c) SIFT Results for sequence 3

4 Implementing a modified SIFT descriptor

In this chapter, the performance of SIFT descriptor is going to be compared with the performance of other 3 descriptors which are BRISK, FREAK and SURF. Like in the previous chapter, in this chapter also the performance of the each descriptor is going to be calculated for each sequence and each version (a), (b), (c) and (d).

4.1 Overview of BRISK Descriptor

BRISK is a 512 bit binary descriptor that computes the weighted Gaussian average over a select pattern of points near the key point (See the figure below). It compares the values of specific pairs of Gaussian windows, leading to either a 1 or a 0, depending on which window in the pair was greater. The pairs to use are preselected in BRISK. This creates binary descriptors that work with hamming distance instead of Euclidean, and run extremely quick in embedded devices.

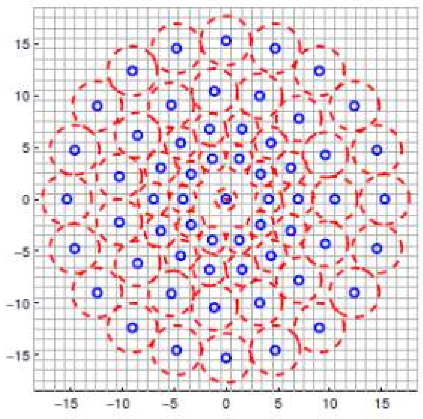


Figure 22: Brisk Sampling Pattern.

4.2 Overview of FREAK Descriptor

FREAK is also a binary descriptor that improves upon the sampling pattern and method of pair selection that BRISK uses. FREAK evaluates 43 weighted Gaussians at locations around the key point, but the pattern formed by these Gaussians is biologically inspired by the retinal pattern in the eye. The pixels being averaged overlap (See the figure below), and are much more concentrated near the key point. This leads to a more accurate description of the key point as analysis will show. The actual FREAK algorithm also uses a cascade for comparing these pairs, and puts the 64 most important bits in front to speed up the matching process.

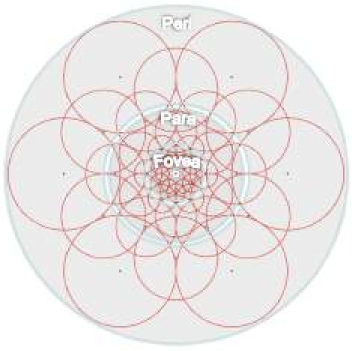


Figure 23: Freak Sampling Pattern.

4.3 Overview of SURF Descriptor

The feature vector of SURF is almost identical to that of SIFT. It creates a grid around the key point and divides each grid cell into sub-grids. At each sub-grid cell, the gradient is calculated and is binned by angle into a histogram whose counts are increased by the magnitude of the gradient, all weighted by a Gaussian. These grid histograms of gradients are concatenated into a 64- dimensional vector. The high dimensionality makes it difficult to use this in real time, so SURF can also use a 36-vector of principle components of the 64 vector (PCA analysis is performed on a large set of training images) for a speedup. SURF also improves on SIFT by using a box filter approximation to the convolution kernel of the Gaussian derivative operator. This convolution is sped up further using integral images to reduce the time spent on this step.

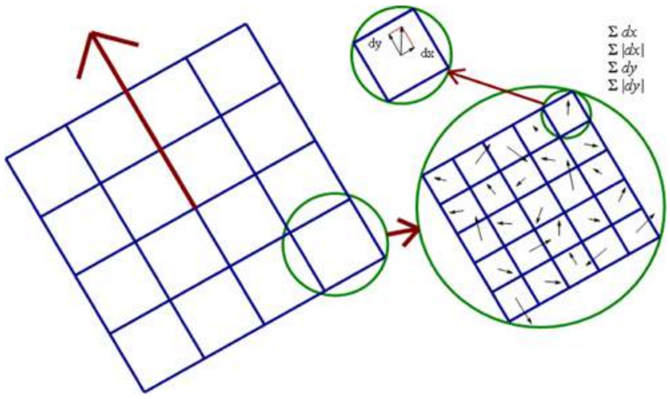


Figure 24: SURF HOG Descriptor.

After giving general information about each key point descriptor, it is the time to show the performance evaluations. The comparisons are made for some cases such as the computation time and the correct matching ratio.

The first case that is going to be evaluated is the correct matching ratio between the key points in different two images. After running each algorithm, key points are going to be obtained and matched, but how many of them are correct, you may not be sure. In order to compute the correct matching, the homography matrix of the each sequence is going to be used.

The homography matrix allows describing a planar transformation between two images. Given a planar transformation between images 0 and n, the corresponding homography matrix will be defined as ${}^{0}H_n$, that is, a matrix that allows obtaining, from a point represented in the image frame 0, its coordinates on image n. Formally, this can be represented as follows:

$$^{n}p =^{0} H_n \cdot^{0} p_i \tag{2}$$

The homographies describing the transformations from image 0 to image n mentioned in the previous chapter. After calculating the location by using homography matrix, the result(x, y) is going to be compared with the result(x, y) that is obtained by the algorithm using the condition (1):

The second case that is going to be computed is the time measurement. It is going to be represented the elapsed time for each image, each version and each sequence.

```
function [ ratio ] = match_FREAK_BRISK_SURF( I_1 , I_2 , homography, type )
```

First thing is to detect features according to the algorithm. As explained above method takes 4 parameters these are the original image, the transformed image, its homography and the type of the algorithm. The features are going to be detected by *detectBRISKFeatures*, *detectFASTFeatures* or *detectSURFFeatures* which are already built-in functions in MATLAB.

```
    if strcmp(type, 'BRISK')

        points_1 = detectBRISKFeatures(I_1);
        points_2 = detectBRISKFeatures(I_2);

    elseif strcmp(type, 'FREAK')

        points_1 = detectFASTFeatures(I_1);
        points_2 = detectFASTFeatures(I_2);

    elseif strcmp(type, 'SURF')

        points_1 = detectSURFFeatures(I_1);
        points_2 = detectSURFFeatures(I_2);

    end
```

After detecting the features, you should Extract interest point descriptors by calling the *extractFeatures* method which is also already implemented in MATLAB.

```
    [features_1 , valid_points_1] = extractFeatures(I_1 , points_1);
    [features_2 , valid_points_2] = extractFeatures(I_2 , points_2);
```

Then you should find matching features between two images. The *matchFeatures* method returns indices of the matching features in the two input feature sets. Both of the input features must be either binary features objects or matrices.

```
    % match the features
    index_pairs = matchFeatures(features_1 , features_2);
```

Then you retrieve the locations of the corresponding points for each image.

```
    % get the location of the matched descriptors according to the index pairs
    matched_points_1 = valid_points_1(index_pairs(1: end, 1));
    matched_points_2 = valid_points_2(index_pairs(1: end, 2));
```

After that it is almost the same like the previous chapter.

```
    p_original_arr = matched_points_1.Location;
    l_final_arr    = matched_points_2.Location;

    ratio = 0;

    for i = 1: size(p_original_arr , 1);

        p_original = p_original_arr(i, :);
        p_final    = homography * [p_original_arr(i, :) 1]';
        p_final    = p_final / p_final(3);

        l_final    = [l_final_arr(i, 1); l_final_arr(i, 2); 1];

        % check the distnace between points
        if sqrt((p_final(1) - l_final(1))^2 + (p_final(2) - l_final(2))^2) <= 1.0

            ratio = ratio + 1;

        end

    end

    ratio = ratio * 100 / size(p_original_arr , 1);
```

4.4 Comparison

SIFT From the figure 25, it is shown that SIFT is the best key point descriptor. It has the best approximation and the matching rate for each sequence.

For image(a) that correspond to sequence 1 (projective transformation), it is clear that the performance of SIFT is decreasing starting from image 1,5,9,13 correspond to tilting the camera with angle equivalent to 25 pixels to reach image 4,8,12,16 that correspond to tilting the camera with angle equivalent to 100 pixels. We also notice that the performance is highest at set a (without noise) to reach the worst case with set d (noise of 18 grayscale values) since for a, b ,c versions, SIFT gets really high correct matching ratio between almost 90% and 99%. However, when the images have Gaussian noise with 18 grayscale, then the ratio is between 70% and 90%.

For image(b) that correspond to sequence 2 (zooming), it is obvious that the performance of SIFT is very high 100% and consistent for sets a (without noise),b (with noise of 3 grayscale values) and c (with noise of 6 grayscale values). For set d (with noise of 18 grayscale values), the performance of SIFT decreases starting from image 1 corresponds to 110% zooming to reach image 9 that corresponds to 150% zooming.

For image(c) that correspond to sequence 3 (rotation), it is clear that the performance of SIFT is quite high and consistent for sets a (without noise),b (with noise of 3 grayscale values) and c (with noise of 6 grayscale values) to be between 97% and 100%. For set d (with noise of 18 grayscale values), the performance of SIFT increases starting from 89% for image 1 corresponds to -45 degrees rotation to reach 94% for image 9 that corresponds to -5 degrees rotation. Then it start decreasing starting from image 10 corresponds to 5 degrees rotation to be 89% for image 18 that corresponds to 45 degrees rotation .

The only weakness is the speed. The computation time for each image, version and sequence is very high. It is very slow but the most robust. You can see the time measurement for each image, version (a, b, c, d) and sequence below.

Table 1: SIFT Sequence-1 Computation Time Results

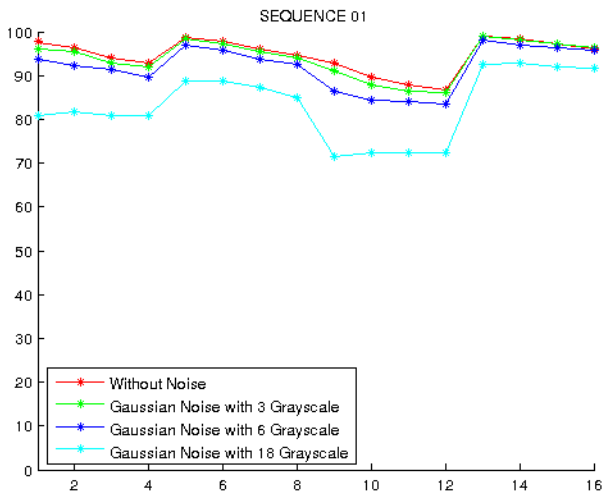
SEQUENCE	Image-1	Image-2	Image-3	.	.	.	Image-14	Image-15	Image-16
WITHOUT NOISE	11.4448	10.1812	11.0894	.	.	.	10.3635	11.3699	10.7506
WITH NOISE 3	10.5534	11.6548	10.5006	.	.	.	10.2364	11.4524	11.9299
WITH NOISE 6	10.3434	10.0549	10.7281	.	.	.	10.2915	10.1790	11.2674
WITH NOISE 18	10.1580	10.4432	9.9322	.	.	.	10.4184	10.4583	9.8571
TOTAL:	687.4525 seconds								

Table 2: SIFT Sequence-2 Computation Time Results

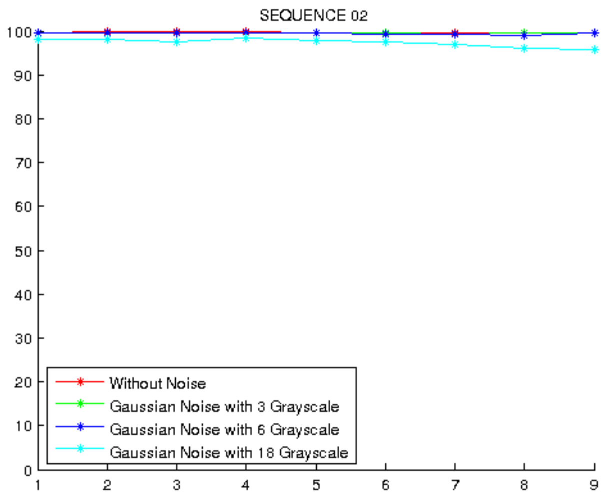
SEQUENCE	Image-1	Image-2	Image-3	Image-4	Image-5	Image-6	Image-7	Image-8	Image-9
WITHOUT NOISE	13.1496	14.5265	13.3357	12.4768	14.1421	11.4554	12.2458	10.7547	8.1227
WITH NOISE 3	19.7670	10.1863	9.2783	9.0948	9.1318	9.2500	8.5213	7.9581	7.8498
WITH NOISE 6	11.4822	11.2329	9.8891	10.2038	8.9488	8.7247	8.3279	9.2383	8.1820
WITH NOISE 18	9.8735	10.1453	9.9490	8.8705	9.1308	8.9118	8.2151	8.9172	8.7158
TOTAL:	360.2054 seconds								

Table 3: SIFT Sequence-3 Computation Time Results

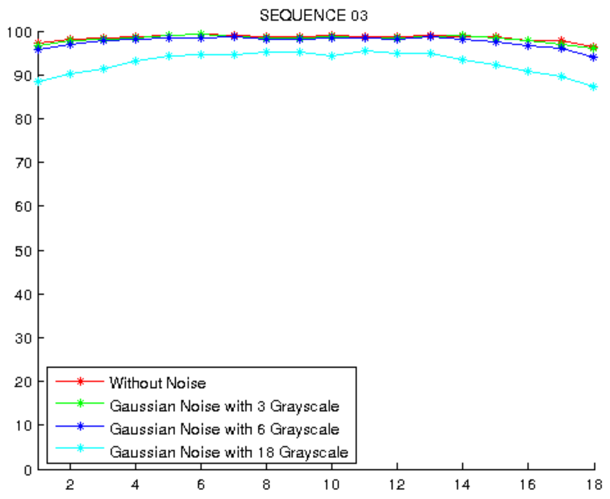
SEQUENCE	Image-1	Image-2	Image-3	.	.	.	Image-16	Image-17	Image-18
WITHOUT NOISE	12.2569	13.6476	11.4037	.	.	.	10.7162	10.1130	11.5221
WITH NOISE 3	110.7353	9.8598	10.2098	.	.	.	10.7834	10.6323	11.5386
WITH NOISE 6	9.9097	11.2336	9.5602	.	.	.	10.6138	10.4585	10.5280
WITH NOISE 18	9.3821	9.3394	9.4228	.	.	.	10.2935	11.4904	9.8703
TOTAL:	746.4725 seconds								



(a)



(b)



(c)

Figure 25: (a) SIFT Results for sequence 1, (b) SIFT Results for sequence 2, (c) SIFT Results for sequence 3

BRISK From the figure 26, it is shown that BRISK has no good approximation and the matching rate like SIFT for each sequence.

When you look at the graph, it is really diffucult to interpret it for the sequence 1 (Projective images). But, the ratio is changing between 55% and 100%.

When you evaluate the sequence 2 (Scaled images), still there is no uniform behavior for each version (a, b, c, d). But, the general trend for the correct matching ratio is decreasing from the first image corresponds to 110% zooming to the last that corresponds to 150% zooming. And the ratio is very wide which is between 15% and 95%.

The last one is sequence 3 which contains rotated images. In that graph also BRISK does not behave in a uniform way. The ratio is changing between 20% and 85%.

The computation time for each image, version and sequence is quite low when it is compared with SIFT. It is very fast but not robust like SIFT. You can see the time measurement for each image, version (a, b, c, d) and sequence below.

Table 4: BRISK Sequence-1 Computation Time Results

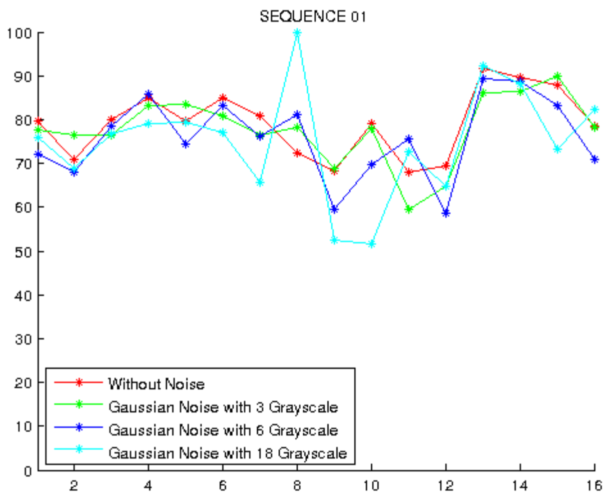
SEQUENCE	Image-1	Image-2	Image-3	.	.	.	Image-14	Image-15	Image-16
WITHOUT NOISE	0.4669	0.1990	0.2242	.	.	.	0.2492	0.2442	0.2502
WITH NOISE 3	0.2527	0.2510	0.2526	.	.	.	0.1848	0.1865	0.1950
WITH NOISE 6	0.1840	0.1828	0.1797	.	.	.	0.1863	0.1873	0.1885
WITH NOISE 18	0.1886	0.1848	0.1912	.	.	.	0.1901	0.1911	0.1917
TOTAL:	13.1725 seconds								

Table 5: BRISK Sequence-2 Computation Time Results

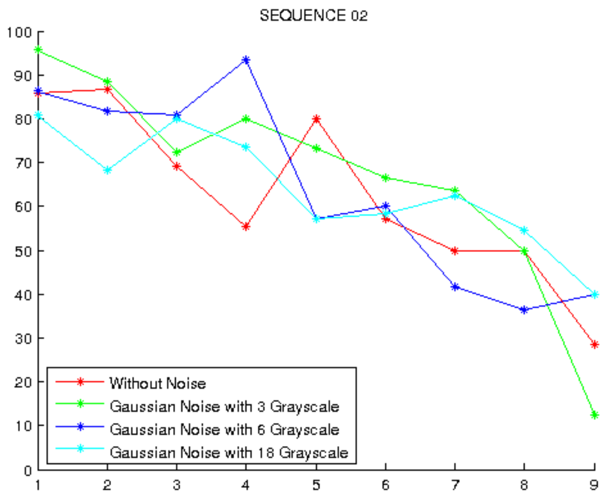
SEQUENCE	Image-1	Image-2	Image-3	Image-4	Image-5	Image-6	Image-7	Image-8	Image-9
WITHOUT NOISE	0.1940	0.1919	0.2175	0.2188	0.2141	0.2535	0.2299	0.2484	0.2160
WITH NOISE 3	0.2107	0.2041	0.2266	0.2128	0.2294	0.2146	0.2032	0.2081	0.2001
WITH NOISE 6	0.2028	0.1981	0.1977	0.1897	0.1983	0.2010	0.1924	0.1913	0.2062
WITH NOISE 18	0.2320	0.2424	0.2072	0.2202	0.2112	0.1970	0.2003	0.1905	0.1979
TOTAL:	7.5697 seconds								

Table 6: BRISK Sequence-3 Computation Time Results

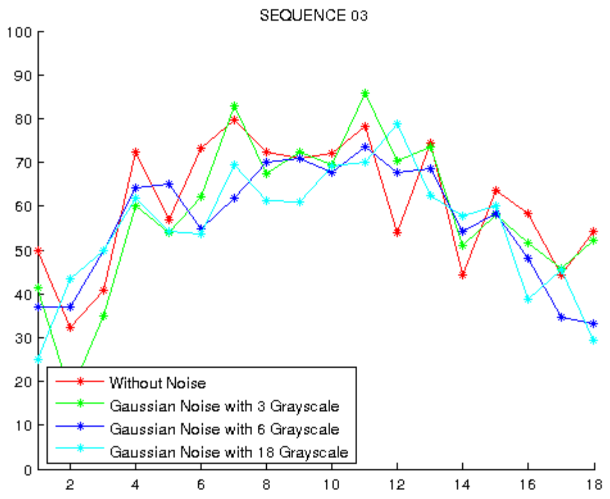
SEQUENCE	Image-1	Image-2	Image-3	.	.	.	Image-16	Image-17	Image-18
WITHOUT NOISE	0.2040	0.2050	0.2037	.	.	.	0.2085	0.2042	0.2080
WITH NOISE 3	0.2180	0.2113	0.2245	.	.	.	0.1990	0.2071	0.2373
WITH NOISE 6	0.2405	0.2464	0.1902	.	.	.	0.2426	0.2376	0.2264
WITH NOISE 18	0.2211	0.1876	0.1871	.	.	.	0.2049	0.2061	0.2047
TOTAL:	15.2495 seconds								



(a)



(b)



(c)

Figure 26: (a) BRISK Results for sequence 1, (b) BRISK Results for sequence 2, (c) BRISK Results for sequence 3

FREAK From the figure 27, it is shown that FREAK has no good approximation and the matching rate like SIFT for each sequence. But, when you compare it with BRISK, it behaves in a uniform way. In other words, the chart that we is obtained by FREAK is not bumpy like BRISK.

When you look at the graph for the sequence 1 (Projective images). the ratio is changing between 60% and 100%. Almost all the images for each version (a, b, c, d) behaviour is almost same. But the ratio of the correct matching for the images that have Gaussian noise with 18 grayscale is a little bit low in some cases as expected.

When you evaluate the sequence 2 (Scaled images), there is a uniform behaviour for each version (a, b, c, d). But, the ratio of the correct matching for the images that have Gaussian noise with 18 grayscale is very low(Even 0%) in some cases.(Especially, for the last images)

The last one is sequence 3 which contains rotated images. In that graph, FREAK behave in a uniform way for all the versions (a, b, c, d). The ratio is changing between 35% and 85%. Where the performance of SIFT increases starting from image 1 corresponds to -45 degrees rotation to reach image 9 that corresponds to -5 degrees rotation. Then it start decreasing starting from image 10 corresponds to 5 degrees rotation to reach image 18 that corresponds to 45 degrees rotation .

The computation time for each image, version and sequence is quite low when it is compared with SIFT and a little bit faster than BRISK. It is the fastest one but not robust like SIFT. On the other hand, it provides better approximation and the matching rate results than BRISK. You can see the time measurement for each image, version (a, b, c, d), sequence below.

Table 7: FREAK Sequence-1 Computation Time Results

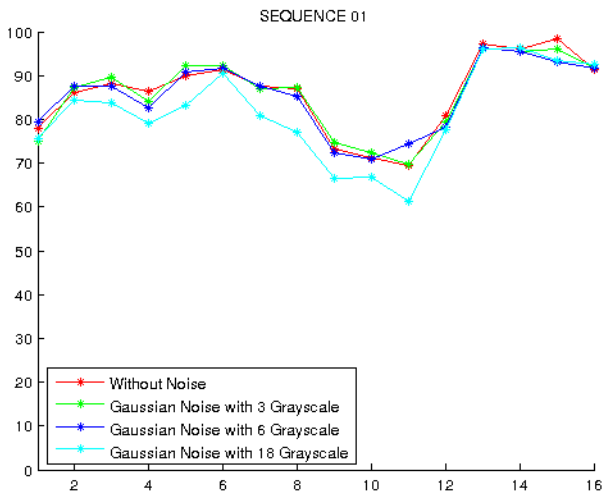
SEQUENCE	Image-1	Image-2	Image-3	.	.	.	Image-14	Image-15	Image-16
WITHOUT NOISE	0.4214	0.1844	0.1804	.	.	.	0.1945	0.1781	0.1795
WITH NOISE 3	0.1858	0.1845	0.1841	.	.	.	0.2322	0.2317	0.2319
WITH NOISE 6	0.1980	0.1758	0.1743	.	.	.	0.1762	0.1795	0.1770
WITH NOISE 18	0.1813	0.1816	0.1819	.	.	.	0.1888	0.1883	0.1918
TOTAL:	12.3782 seconds								

Table 8: FREAK Sequence-2 Computation Time Results

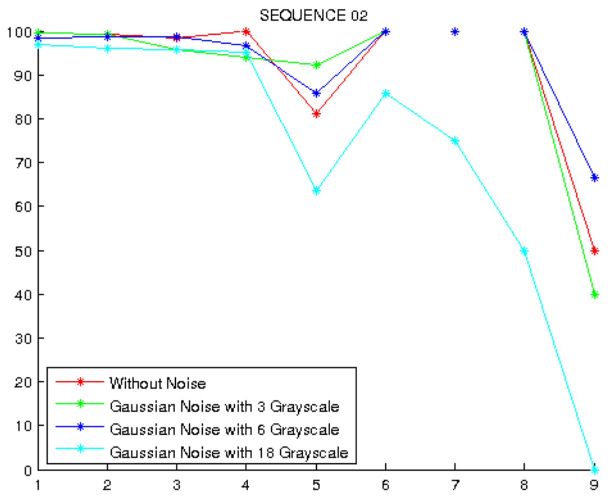
SEQUENCE	Image-1	Image-2	Image-3	Image-4	Image-5	Image-6	Image-7	Image-8	Image-9
WITHOUT NOISE	0.1702	0.1665	0.1635	0.1612	0.1604	0.1596	0.1584	0.1592	0.1578
WITH NOISE 3	0.1724	0.1686	0.1657	0.1677	0.1675	0.1663	0.1652	0.1617	0.1618
WITH NOISE 6	0.1653	0.1614	0.1583	0.1569	0.1594	0.1580	0.1546	0.1575	0.1600
WITH NOISE 18	0.1761	0.1783	0.1708	0.1687	0.1669	0.1663	0.1651	0.1645	0.1645
TOTAL:	5.9065 seconds								

Table 9: FREAK Sequence-3 Computation Time Results

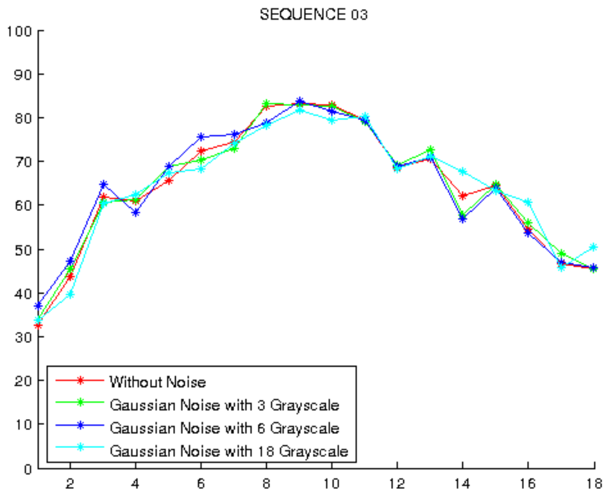
SEQUENCE	Image-1	Image-2	Image-3	.	.	.	Image-16	Image-17	Image-18
WITHOUT NOISE	0.1779	0.1789	0.1760	.	.	.	0.1816	0.1791	0.1788
WITH NOISE 3	0.1744	0.1749	0.1762	.	.	.	0.1811	0.1798	0.1787
WITH NOISE 6	0.1739	0.1747	0.1759	.	.	.	0.1849	0.1841	0.1826
WITH NOISE 18	0.1908	0.1927	0.1930	.	.	.	0.2006	0.1975	0.1952
TOTAL:	13.2988 seconds								



(a)



(b)



(c)

Figure 27: (a) FREAK Results for sequence 1, (b) FREAK Results for sequence 2, (c) FREAK Results for sequence 3

SURF From the figure 28, it is shown that SURF has no good approximation and the matching rate like SIFT for each sequence. But, when you compare it with BRISK, it behaves in a uniform way. In other words, the chart that we is obtained by SURF is not bumpy like BRISK. Its behaviour is uniform like FREAK.

When you look at the graph for the sequence 1 (Projective images). the ratio is changing between 65% and 96%. Almost all the images for each version (a, b, c, d) behaviour is almost same. But the ratio of the correct matching for the images that have Gaussian noise with 18 grayscale is a little bit low in all the cases as expected.

When you evaluate the sequence 2 (Scaled images), there is a uniform behaviour for each version (a, b, c, d). And the ratio is between 75% and 90% for all the versions a, b, c, d.

The last one is sequence 3 which contains rotated images. In that graph, SURF behave in a uniform way for all the versions (a, b, c, d). The ratio is changing between 32% and 83%. But the ratio of the correct matching for the images that have Gaussian noise with 18 grayscale is a little bit low in some cases as expected.

The computation time for each image, version and sequence is quite low when it is compared with SIFT. It is faster than SIFT but slower than FREAK and BRISK as expected and it is also not robust like SIFT. You can see the time measurement for each image, version (a, b, c, d), sequence below.

Table 10: SURF Sequence-1 Computation Time Results

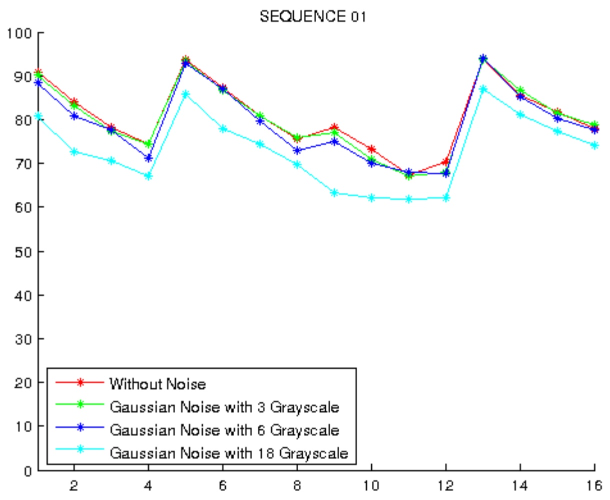
SEQUENCE	Image-1	Image-2	Image-3	.	.	.	Image-14	Image-15	Image-16
WITHOUT NOISE	0.5715	0.4380	0.4480	.	.	.	0.3693	0.4352	0.6989
WITH NOISE 3	0.3676	0.3670	0.3642	.	.	.	0.3731	0.3732	0.3616
WITH NOISE 6	0.3734	0.3731	0.3803	.	.	.	0.3486	0.3404	0.3386
WITH NOISE 18	0.3472	0.3467	0.3610	.	.	.	0.3679	0.3770	0.3616
TOTAL:	24.5500 seconds								

Table 11: SURF Sequence-2 Computation Time Results

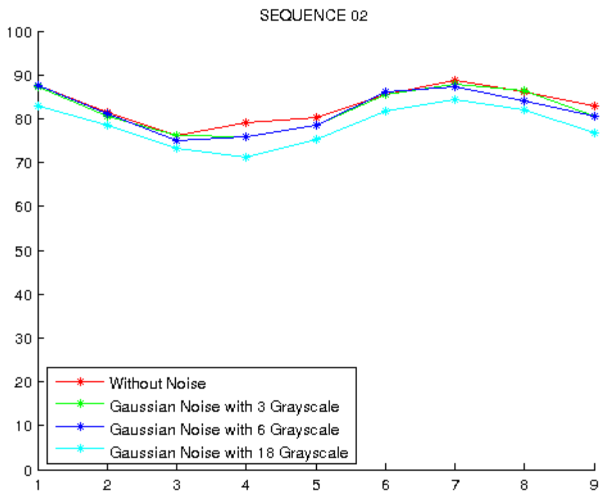
SEQUENCE	Image-1	Image-2	Image-3	Image-4	Image-5	Image-6	Image-7	Image-8	Image-9
WITHOUT NOISE	0.3707	0.4004	0.4481	0.3971	0.3985	0.3311	0.3282	0.3353	0.3196
WITH NOISE 3	0.3615	0.3582	0.3259	0.3260	0.3229	0.3300	0.3220	0.3679	0.3045
WITH NOISE 6	0.3429	0.3551	0.3442	0.3504	0.3400	0.3438	0.3359	0.3355	0.3266
WITH NOISE 18	0.3632	0.3521	0.3544	0.3443	0.3457	0.3450	0.3310	0.3262	0.3326
TOTAL:	12.5165 seconds								

Table 12: SURF Sequence-3 Computation Time Results

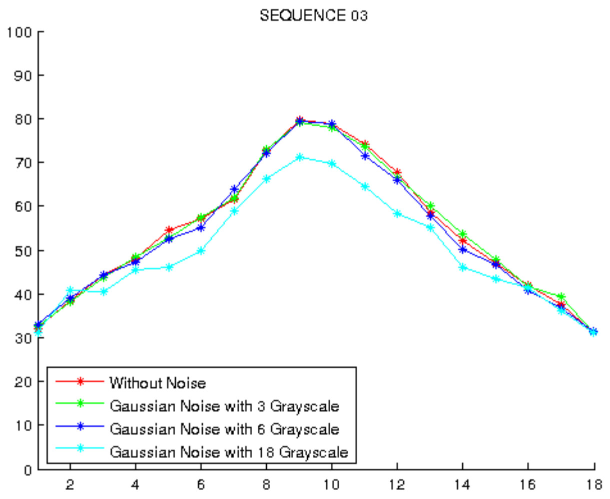
SEQUENCE	Image-1	Image-2	Image-3	.	.	.	Image-16	Image-17	Image-18
WITHOUT NOISE	0.3400	0.3493	0.3664	.	.	.	0.3366	0.3337	0.3336
WITH NOISE 3	0.3503	0.3415	0.3437	.	.	.	0.3528	0.3287	0.3422
WITH NOISE 6	0.3444	0.3661	0.3708	.	.	.	0.3828	0.3974	0.3641
WITH NOISE 18	0.3756	0.3569	0.3450	.	.	.	0.3714	0.3572	0.4340
TOTAL:	26.5192 seconds								



(a)



(b)



(c)

Figure 28: (a) SURF Results for sequence 1, (b) SURF Results for sequence 2, (c) SURF Results for sequence 3

To put it in a nutshell, if the robustness is important for the application, SIFT is advised strongly. However, if the application is running in embedded system or that kind of platform and the speed is the main issue, then FREAK, BRISK or SURF can be used. They are very fast but not robust like SIFT.

5 Conclusion

In the lab, we have successfully generated synthetic dataset to be used for evaluating different feature descriptors. After that, four different descriptors were evaluated (SIFT, SURF, FREAK and BRISK). The performance of all the descriptors was plotted and demonstrated. Finally, the work for the lab was long and even more than expected but we benefited from it a lot and had a practical experience dealing with the most well known descriptors.