Exercise 5

Feature Descriptors and Geometric Transformations

By:

Iris Eting 209027333

Nadav Orenstein 312349509

**1.Detect SURF Features MATLAB function**

The MATLAB function detectSURFFeatures is used to detect SURF (Speeded-Up Robust Features) in an image.

these are useful features for many tasks like object recognition and image matching.

* Name-Value pair arguments: The function allows optional parameters which enables control over various aspects of the feature detection.
* optional parameters:
* MetricThreshold: Threshold for strongest features. Lower value will increase blobs. default: 1000
* NumOctaves: Octaves count as integer ≥ 1. Increase to detect larger blobs, 1-4 are the recommended values. default:3
* NumScaleLevels: number of scale levels for the SURF features.
* ROI: Region of interest as [x y width height] vector. lets us detects blobs within only a specific area. default: the whole image

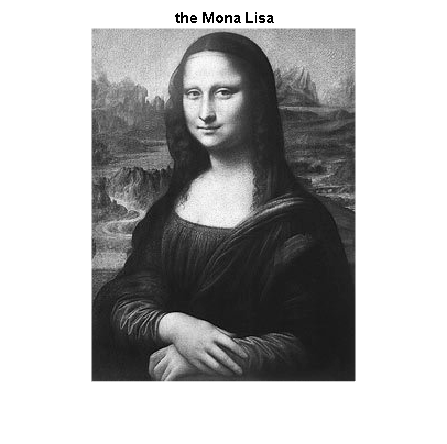
### (2) extracting Features

The Matlab function extracts descriptors and their locations from image points, supporting tasks like recognition and tracking by converting image areas into processable vectors.

-The points argument should come from some other feature detection algorithm (e.g., SIFT, SURF, corners detection), marking special points in an image for more analysis such as matching or tracking.

### (3) Read the images ’mona\_org.jpg’

here we read the image and convert it to grayscale normalized



### (4) Find all the features in the image

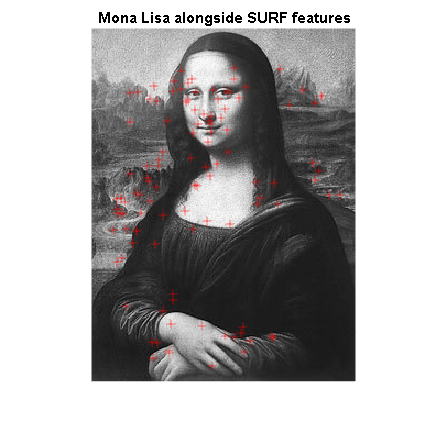
here we first detect the SURF features existing in the image. when timing the operation on our machine we get <<>> seconds runtime.

time\_no\_ROI = 0.0101

this could be considered a very costly operation, especially under realtime constraints. let's what's the number of features we found:

number\_of\_features = 111

and visualize them along with the image.:



it seems that we found as much as 111 features in the image, from trees to facial features. the runtime we got for the operation may seem low, and indeed is for this type of data processing, but it might be heavy on certain real-time and computationally limited systems. (takes approximately 2 frames in 30fps which is a standard framerate)

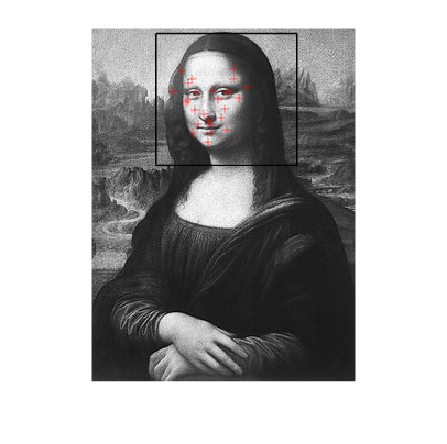
### (5) Find features in the region of interest

using the pair ’ROI’, [59, 5, 128, 120], we can limit the search of SURF feaures to a specific region of interetst. let's see how good a trade of it is to use a smart choice of ROI value in terms of efficiency:

time\_with\_ROI = 0.0039

number\_of\_features = 26

then let's show the result:



In order to know how much we gained in efficiency from this approach, it is a good idea to calculate the speedup we achieved as such:

speedup = time\_no\_ROI / time\_with\_ROI

speedup = 2.5795

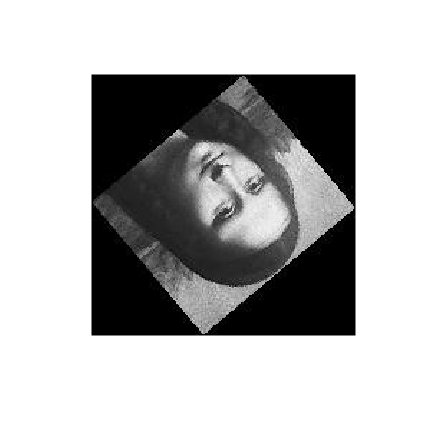
we presented the selected region of interest together with the feature points, the region is the face of the Mona Lisa.

we see the algorithm returned a veriety of segnificant facial features, eyes, nose chin, and more. with the runtime this time we achieved a speedup of nearly 2.5! we found only 26 features this time. this is a tradeoff between realtime computational speed and sizse of detection region, which also corresponds to the amount of features we get. as we agree to lose information about wider regions, we benefit in runtime.

1.2 Make Mona Straight Again

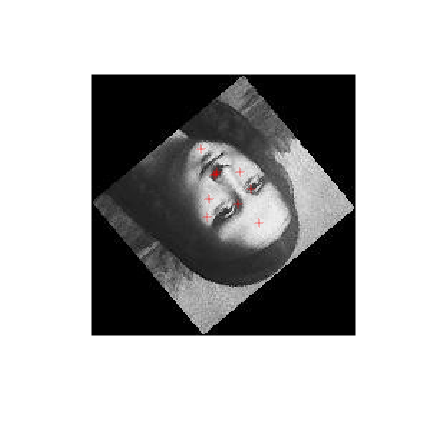
### (1) Read the images

let's read the images ’straight\_mona.PNG’ and ’crooked\_mona.jpg’.



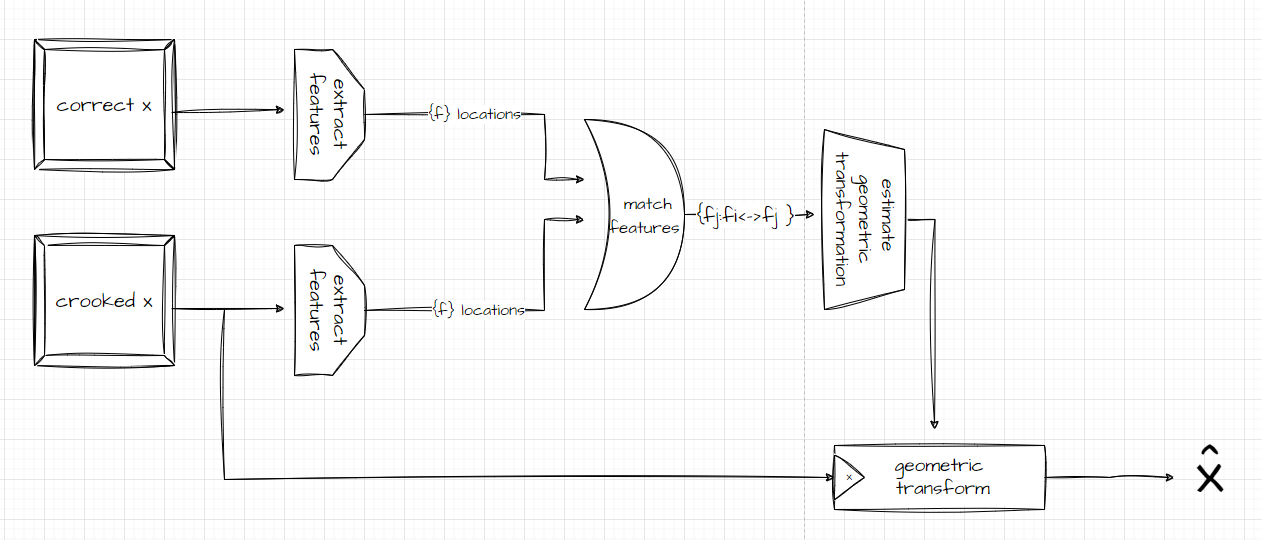
### (2) Extract the SURF feature points

here we extract the SURF feature points of each of the images, as before and display only the ten strongest features for each of the images correspondingly, alongside the images themselves:



### (3) fix the Mona Lisa

We took the approach explained in the following diagram:



Let's break down the code implementing this algorithm, block by block:

* get the locations of the features

% Extract features around each of the detected points

[straight\_features, straight\_validPoints] = extractFeatures(straight\_mona, straight\_points);

[crooked\_features, crooked\_validPoints] = extractFeatures(crooked\_mona, crooked\_points);

* match between each feature in each picture

% Match the features based on their descriptors

indexPairs = matchFeatures(straight\_features, crooked\_features,'MatchThreshold',50);

* match between each feature in each picture

% get locations of the matched points

matchedPtsStraight = straight\_validPoints(indexPairs(:, 1));

matchedPtsCrooked = crooked\_validPoints(indexPairs(:, 2));

* now, based on the matches, we can build the transform

% estimate the transform

[tform,inlierIdx] = estgeotform2d(matchedPtsCrooked,matchedPtsStraight,"similarity");

outputsize = imref2d(size(straight\_mona));

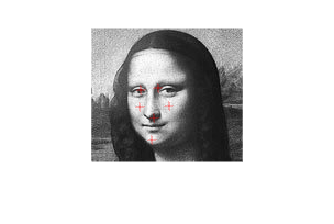
* and then apply it to recover the Mona Lisa

% recover the Mona Lisa using the transform

recoverd\_mona = imwarp(crooked\_mona, tform, "OutputView", outputsize);

imshow(recoverd\_mona);title("recoverd mona")

let's see the locations of the featuers we extracted in first stage:



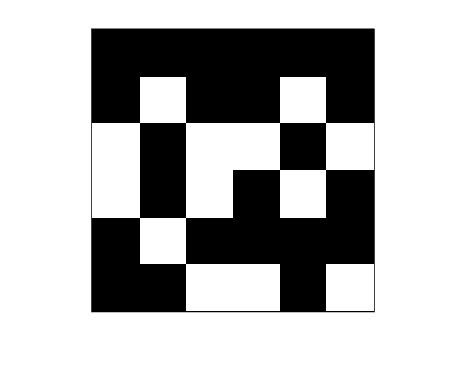
let's display the pairs we got in second stage



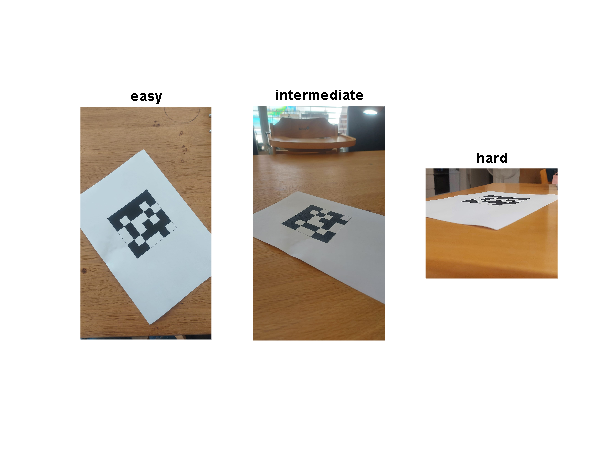
Looks like we have got an eye for an eye and a tooth for a tooth. Now let's see the final product, the recovered Mona Lisa:



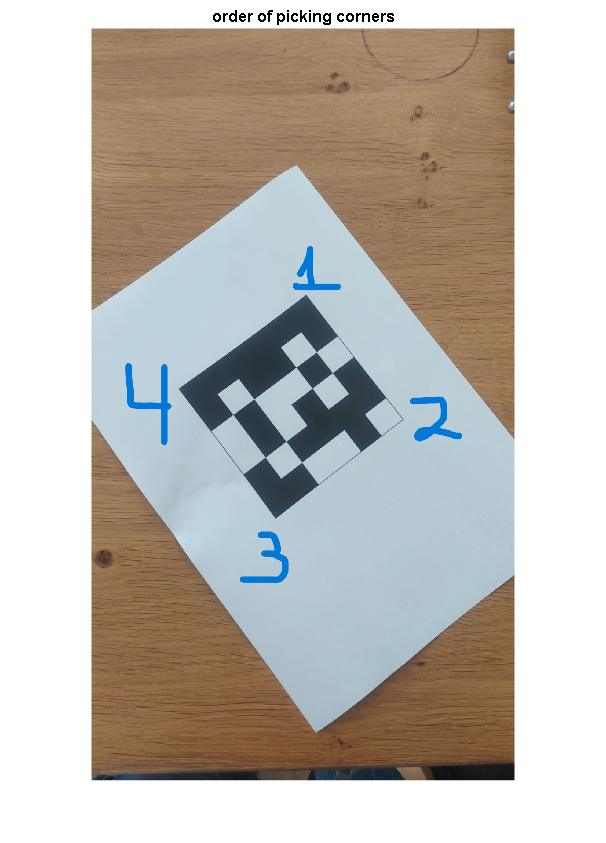
(1) when we insert Nadav's 9 digit ID string to the attached function, this is what we get



(2) we printed it and took three photos of the QR code from three different angles and labled them according to the shooting angle: easy, intermediate and hard.



(3) We located the points on the QR code in each image manually... what a way to pick corners.



Not sure what to do with these 😢:

%% THE WAY WE GOT THE CORNERS MANUALLY

%figure();imshow(qr\_easy);title('easy');

%[easy\_x, easy\_y] = ginput(4);

%hold on; scatter(easy\_x, easy\_y,'filled', 'o','LineWidth',1);

%easy\_corners = [easy\_x, easy\_y];

%save('easy\_corners.mat','easy\_corners');

%figure();imshow(qr\_inter);title('intermediate');

%[inter\_x, inter\_y] = ginput(4);

%hold on; scatter(inter\_x, inter\_y,'filled', 'o','LineWidth',1);

%inter\_corners = [inter\_x, inter\_y];

%save('inter\_corners.mat','inter\_corners');

figure();imshow(qr\_hard);title('hard');

[hard\_x, hard\_y] = ginput(4);

hold on; scatter(hard\_x, hard\_y,'filled', 'o','LineWidth',1);

hard\_corners = [hard\_x, hard\_y];

save('hard\_corners.mat','hard\_corners');

pic\_easy\_mat = importdata('easy\_corners.mat');

pic\_med\_mat = importdata('inter\_corners.mat');

pic\_hard\_mat = importdata('hard\_corners.mat');

# (4) transformations

That's ok I got this

here we transform the image such that the QR code is straightened. For each image all three transformations we learned in class are applied:

* Rigid (rotation, translation and scale transformation),
* Affine (shearing added),
* Perspective.

here is the code for the transformations function:

%Iris Eting 209027333

%Nadav Orenstein 312349509

function [si1, si2, si3] = dip\_straight\_image(image, corners)

% Define a set of fixed points for transformation

fixedPoints = [1 1; 1 258; 258 258; 258 1];

% Perform rigid transformation (similarity) and warp the image

tform\_rigid = fitgeotform2d(corners, fixedPoints, 'similarity');

si1\_LR = imwarp(image, tform\_rigid, 'OutputView', imref2d([258 258]));

si1 = fliplr(si1\_LR);

% Perform affine transformation and warp the image

tform\_affine = fitgeotform2d(corners, fixedPoints, 'affine');

si2\_LR = imwarp(image, tform\_affine, 'OutputView', imref2d([258 258]));

si2 = fliplr(si2\_LR);

% Perform perspective transformation and warp the image

tform\_perspective = fitgeotform2d(corners, fixedPoints, 'projective');

si3\_LR = imwarp(image, tform\_perspective, 'OutputView', imref2d([258 258]));

si3 = fliplr(si3\_LR);

% Display the results in a 1x3 subplot

figure();

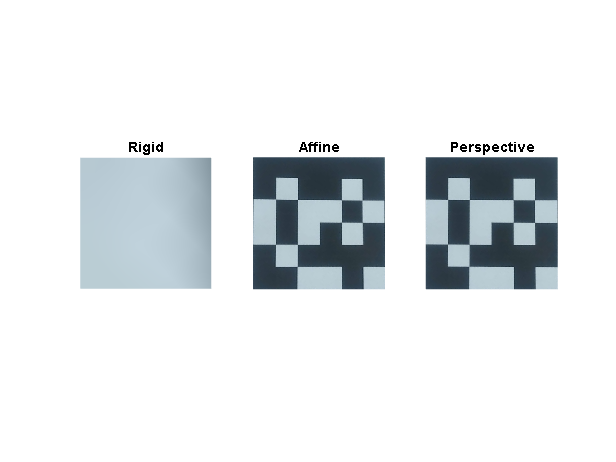
subplot(1, 3, 1); imshow(si1); title('Rigid');

subplot(1, 3, 2); imshow(si2); title('Affine');

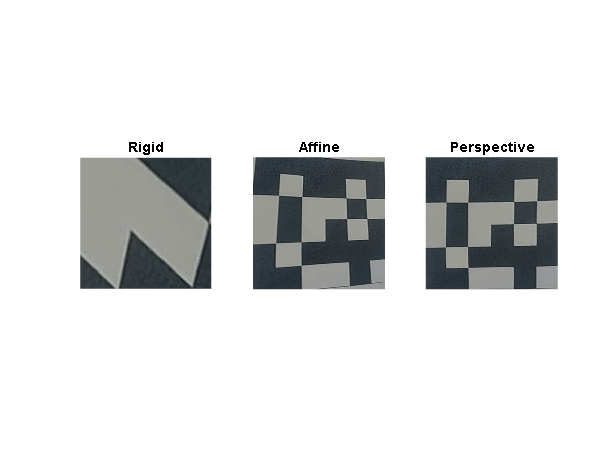
subplot(1, 3, 3); imshow(si3); title('Perspective');

end

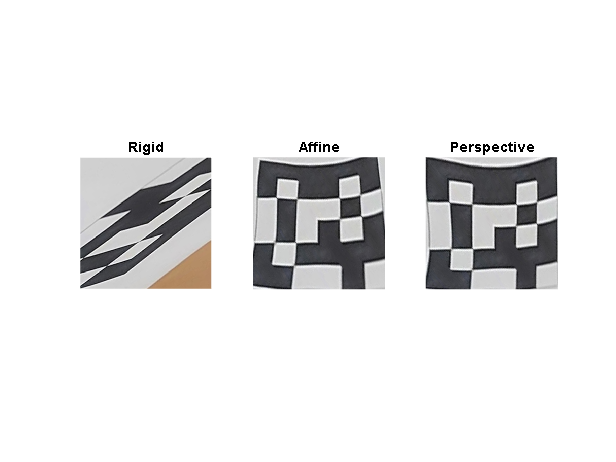
Easy



Med



Hard



Display the results in a 1x3 subplot and save them

fig = figure();

subplot(1, 3, 1); imshow(easy\_str\_1); title('Rigid: Easy');

subplot(1, 3, 2); imshow(easy\_str\_2); title('Affine: Easy');

subplot(1, 3, 3); imshow(easy\_str\_3); title('Perspective: Easy');

saveas(fig,'ex2 Easy transforms.jpeg');

### (5) results

If you have time, if not its ok

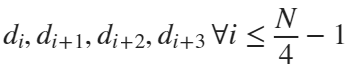
### (6) Extractig the binary values from the straightened QR

first, we calculate block size for our main block processing loop, by deviding the image to 36 regions. we are looking at the image as a 6x6 grid of blocks, where each block represents 1 bit.

in the main loop - for each block  we decide weather it represents a 0 or a 1 by measuring average intensity. if it passes a treshold (140), it is decides that it represents a binary , else, we get a binary .

this way, we get a binary array with the representation of the image's encoded bits.

### (7) converting the binary array to decimals

After that, we loop through all the bits in groups of four and convert them to digits. for each 4 pixels  we extract the decimal value of those 4 bits in order. each can represent togerther any digit 

here is the code for the QR decoder function:

% Decode QR Code

% Iris Eting 209027333

% Nadav Orenstein 312349509

function output = decode\_QR(img)

% Convert the input RGB image to grayscale

I = rgb2gray(img);

% Convert the image to double precision

I = im2double(I);

% Normalize the pixel values to the range [0, 255]

I\_min = min(I(:));

I\_max = max(I(:));

img = ((I - I\_min) / (I\_max - I\_min)) \* 255;

% Input img of size 258\*258 - each column represents 6 binary digits

% Output: str

% Define the length of the ID (4 binary digits \* number of expected digits)

id\_length = 4 \* 9;

% Set a threshold for binary conversion

threshold = 140;

% Initialize binary array to store binary values

binary\_array = ones(1, id\_length);

% Calculate block size for processing

block\_size = size(img, 1) / 6;

% Counter for binary array

n = 1;

% Loop through the 6x6 blocks

for i = 1:6

for j = 1:6

% Extract the current block

block = img(((j - 1) \* block\_size + 1):(j \* block\_size),((i - 1) \* block\_size + 1):(i \* block\_size));

% Calculate the average intensity of the block

avg = mean(block, 'all');

% Determine binary value based on the average intensity

if avg > threshold

binary\_array(n) = 1;

else

binary\_array(n) = 0;

end

% Increment counter

n = n + 1;

end

end

% Initialize output array

output = zeros(1, 9);

% Convert binary values to decimal and store in the output array

for k = 1:9

str\_k = num2str(binary\_array(((k - 1) \* 4 + 1): k \* 4));

output(k) = bin2dec(str\_k);

end

end

here are the results

easy\_id = decode\_QR(easy\_str\_3)

3 1 2 3 4 9 5 0 9

Med\_id = decode\_QR(inter\_str\_3)

3 1 2 3 4 9 5 0 9

Hard\_id = decode\_QR(hard\_str\_3)

3 1 2 3 4 9 5 0 9

Seems to work well!