COMPUTER VISION COURSE (2022/2023)
Master in Electrical and Computer Engineering
Laboratory Assignment 2

Feature Detection and Matching

1 Learning Goals and Description

Local features and their descriptors are the building blocks of many computer vision algorithms. Their applications include image registration, object detection and classification, tracking, and motion estimation. These algorithms use local features to better handle scale changes, rotation, and occlusion.

`Computer Vision Toolbox` algorithms include the *FAST*, *Harris*, and *Shi & Tomasi* corner detectors, and the *SURF*, *KAZE*, and *MSER* blob detectors. The toolbox includes the *SURF*, *FREAK*, *BRISK*, *LBP*, *ORB*, and *HOG* descriptors. You can mix and match the detectors and the descriptors depending on the requirements of your application.

In this assignment, you will write code to detect discriminating features in an image and find the best matching features in other images. Figure 1 shows the expected output of your code. Your features should be reasonably invariant to translation, rotation, and scale. You will evaluate their performance on a suite of benchmark images.

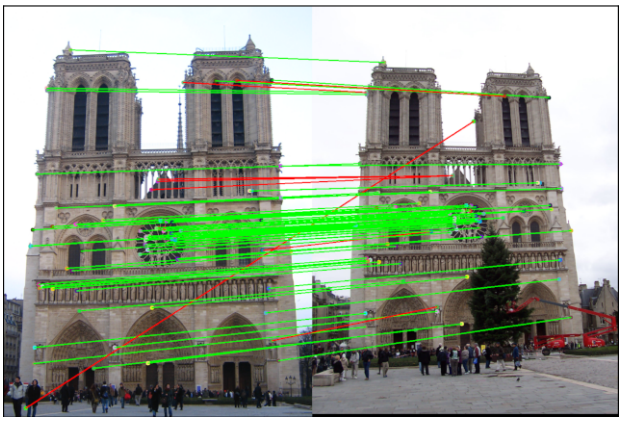

Figure 1: Feature detection and matching.

As a first step for the success of this assigment we strongly recommend you to spend a few minutes exploring the *Feature Detection and Extraction* functions from the Matlab's `Computer Vision Toolbox`.

Your submission for this assignment should be a zip file, `<ClassIDNamesID.zip>`, composed of your report, your Matlab implementations (including any helper functions), and, if the case, your implementations and results for extra credits (optional).

Your final upload should have the files arranged in this layout:

`<ClassIDNamesID>.zip`

- `<NamesID>`

- `<NamesId.pdf> - PDF Assignment report`
- `Matlab`
  - `HarrisCorner.m`
  - `KeypointsDetection.m`
  - `FeatureDescriptor.m`
  - `FeatureMatching.m`
  - `ShowMatching.m`

Please make sure that any file paths that you use are relative and not absolute. Not `imread('/name/Documents/hw1/data/xyz.jpg')` but `imread('../data/xyz.jpg')`.

Every script and function you write in this section should be included in the `matlab/` folder. Please include resulting images in your report.

2 Assignment Pipeline

This assignment has three main parts: feature detection, feature description, and feature matching.

Feature Detection In this step, you will identify points of interest in the image using the Harris corner detection method. The steps are as follows:

- For each point in the image, consider a window of pixels around that point. Compute the Harris matrix H for (the window around) that point, defined as where the

$$H = \sum_p w_p \nabla I_p \left(\nabla I_p\right)^T = \sum_p w_p \begin{pmatrix} I_{x_p}^2 & I_{x_p} I_{y_p} \\ I_{x_p} I_{y_p} & I_{y_p}^2 \end{pmatrix} =$$

$$= \sum_p \begin{pmatrix} w_p I_{x_p}^2 & w_p I_{x_p} I_{y_p} \\ w_p I_{x_p} I_{y_p} & w_p I_{y_p}^2 \end{pmatrix} = \begin{pmatrix} \sum_p w_p I_{x_p}^2 & \sum_p w_p I_{x_p} I_{y_p} \\ \sum_p w_p I_{x_p} I_{y_p} & \sum_p w_p I_{y_p}^2 \end{pmatrix}$$

  summation is over all pixels p in the window. I_{x_p} is the x derivative of the image at point p, the notation is similar for the y derivative. You can use the Sobel operator to compute the x, y derivatives, but improved results are obtained when convolving the image with the horizontal and vertical derivatives of a Gaussian (typically with $\sigma_d = 1.0$). The weights ω_p should be chosen to be circularly symmetric (for rotation invariance). You should use a $n \times n$ Gaussian mask with 2.0 sigma (σ_i).
  Note that H is a 2×2 matrix. To find interest points, first compute the corner strength function

$$c(H) = det(H) - 0.1 \cdot (trace(H))^2. \tag{1}$$

  Once you've computed c for every point in the image, choose points where c is above a *threshold*. The range of corner strength values c varies from image to image which means that no single threshold can be tuned for all images in test. Instead, you should consider an image-based threshold. You are encoraged to define your image-based threshold as a percentage of the maximum corner strenght c_max in an image. You also want c to be a local maximum in a 7×7 neighborhood (region NMS).

- In many situations, detecting features at the finest stable scale possible may not be appropriate. For example, when matching images with little high frequency detail (e.g., clouds), finescale features may not exist. Instead of extracting features at many different scales and then matching all of them, it is more efficient to extract features that are stable in both *location* and *scale*. In order to add a scale selection mechanism to the Harris corner detector, you should evaluate the `Laplacian-of-Gaussian` function at each detected Harris point (in a multi-scale pyramid) and keep only those points for which the Laplacian is extremal (larger or smaller than both its coarser and finer-level values). Use function `fspecial` to obtain the *log* kernel. Figure 2 shows the expected result when applying the LoG in a multi-scale pyramid approach.

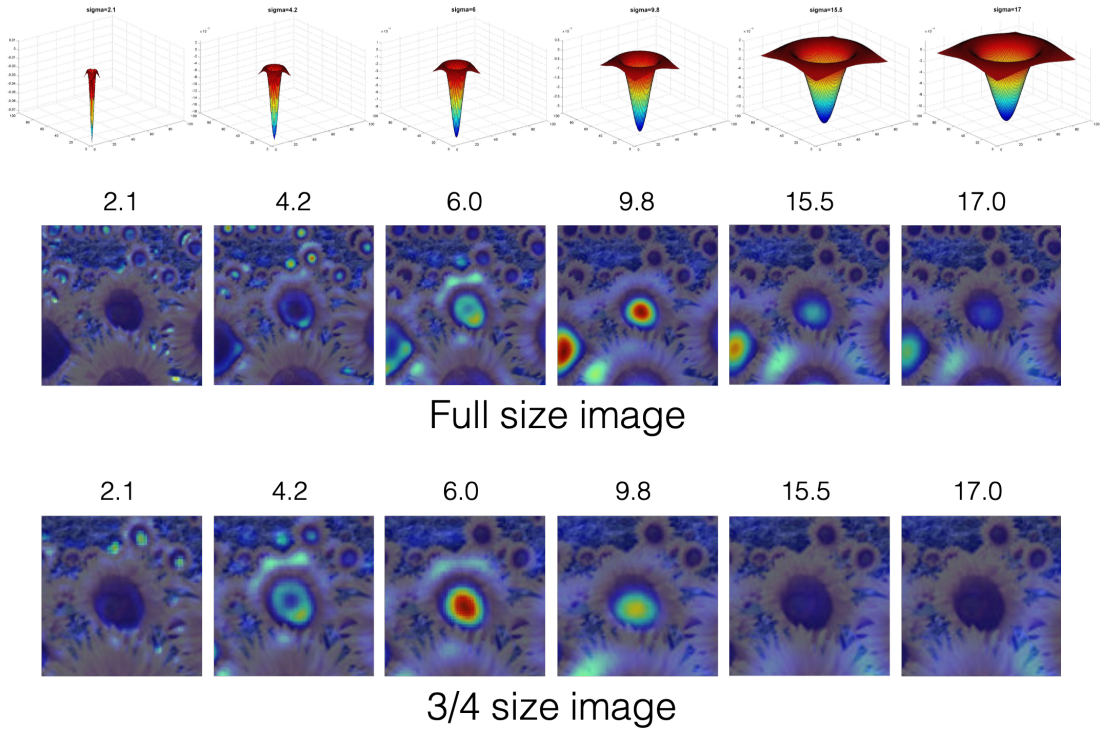

Figure 2: `LoG` multi-scale pyramid convolution in two different scale images. At different scales, the maximum value of the `LoG` convolution is obtained for different scales (σ).

- In addition to computing the feature locations and scale, you'll need to compute a canonical orientation for each feature, and then store this orientation (in degrees) in each feature element. To compute the canonical orientation at each feature you can use different strategies. The dominant orientation of the image patch is given by the eigenvector of H corresponding to $\lambda+$, being $\lambda+$ the larger eigenvalue. Another simple solution is to use first-order steerable filters. What we propose you is to implement a strategy inspired on this latter strategy. Compute the gradient (using the `Sobel` operator again) of a blurred image (with a Gaussian kernel with 13×13 window and 2.0 sigma) and use the angle of the gradient as orientation.

The function `HarrisCorner.m`, which output is used by function `KeypointsDetection.m` is one of the main ones you will complete, along with several helper functions to compute the Harris scores and the scale and orientation for each detected corner in the image.

These set of functions implement the Harris keypoints detector. For filtering the image and computing the gradients, you can use your own functions, implemented in the first assignment.

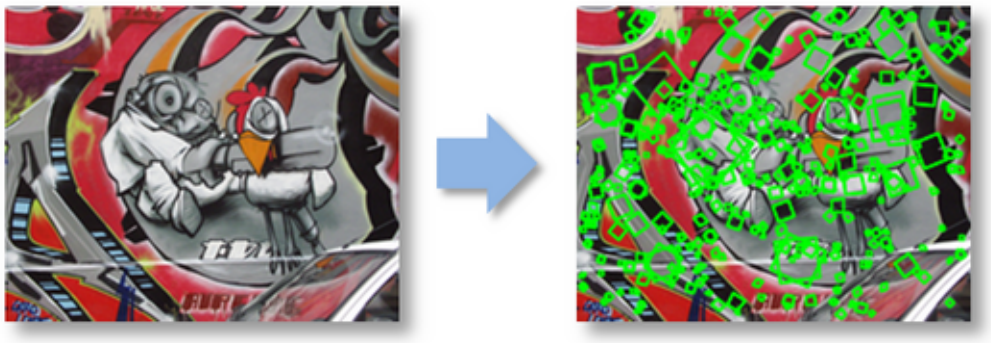

Figure 3: Keypoints (Harris Corners) detection.

Write a function that detects corner keypoints using the Harris Corner detection solution

`function [Pts] = HarrisCorner(img0,thresh,`σ_d`,`σ_i`,NMS)`

As input, the function takes a greyscale image (`img0`) and a set of complementary parameters: `threshold value`, `scale of derivative Gaussian`, `scale of integrative Gaussian` and the `size for region-NMS`. The output of the function will be a data structure `Pts` that stores the coordinates of the detected corners.

The `KeypointsDetection` function takes as input the original greyscale image and the data structure that holds the detected Harris points. The function will output the updated data structure for the keypoints, adding to the previous data the computed keypoint's scale and orientation.

`function [Pts] = KeypointsDetection(img0,Pts)`

The final data structure will be an array of keypoint's data holding: `corner's location` (x, y), `corner's orientation` θ and `corner's scale` σ. Figure 3 shows the expected output of the Keypoints detection function.

Feature description

Now that you've identified points of interest, the next step is to come up with a descriptor for the feature centered at each interest point. This descriptor will be the representation you'll use to compare features in different images to see if they match. Figure 4 shows the idea behind local descriptors centered at keypoints.

Write a function that extract the feature descritors from the keypoints of an image.

`function [Descriptors] = FeatureDescriptor(img0,Pts,'Simple' or 'S-MOPS',N)`

You will implement two descriptors for this function.

1. For starters, you will implement a simple descriptor, a 5×5 square window without orientation centered at each keypoint. This should be very easy to implement and should work well when the images you're comparing are related by a translation. The discriptor is simply the vectorization of the pixel's intensity (Grey-level) inside the 5×5 square patch.

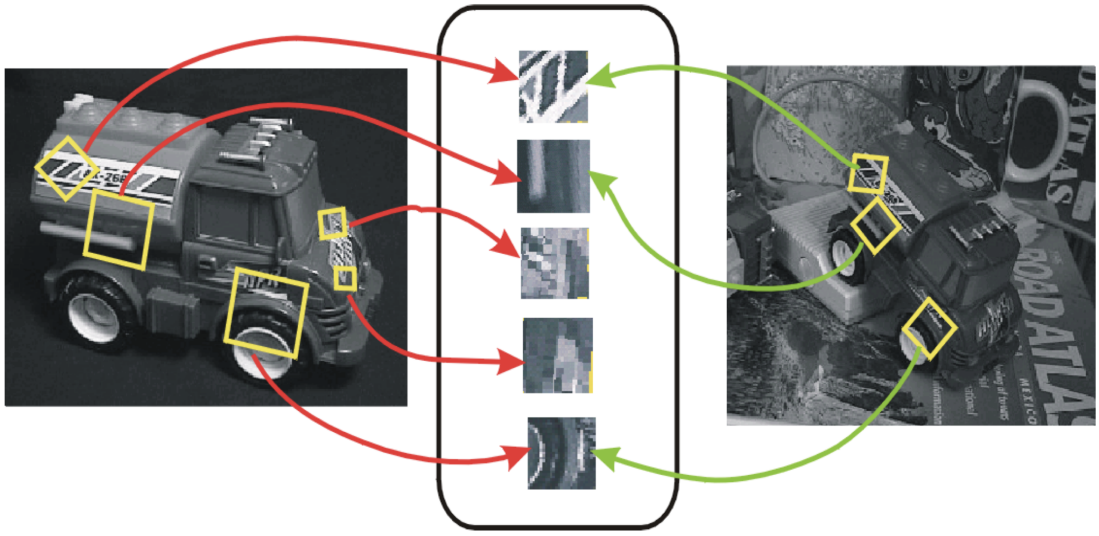

Figure 4: Local descriptors invariant to scale and orientation.

2. Second, you'll implement a simplified version of the `MOPS` descriptor. You'll compute an 8×8 oriented patch sub-sampled from a $N \times N$ pixel region around the feature. You have to come up with a transformation matrix which transforms the $N \times N$ rotated window around the feature to the 8×8 patch, with a canonical orientation where 0 degree corresponds to the $(1, 0)$ direction vector, i.e. the vector points to the right. You should also normalize the patch to have *zero mean* and *unit variance*.

You'll need to implement **two** feature descriptors, in a `SimpleFeatureDescriptor` and `MOPSFeaturesDescriptor` function, respectively. The `FeatureDescriptor` function takes the location and orientation information already stored in a set of keypoints (e.g., Harris corners), and compute descriptors for these keypoints, then store these descriptors in an array which has the same number of rows as the computed keypoints and columns as the dimension of the feature ($5 \times 5 = 25$ for *SimpleFeatureDescriptor*).

For the `MOPS-based` implementation, you have to create a 8×8 matrix, by transforming the $N \times N$ patch around the keypoint to canonical orientation and scales it down to 8×8. You can use the `Matlab` function `B=imwarp(A,TForm)` to transform the patch to a canonical orientation and the function `imresize` to scale down the transformed descriptor's patch.

The function will input a greyscale image (`img0`) and the keypoint's data struture `Pts` obtained in `KeypointsDetection` function. To enable this function to extract both type of descriptors, a *Type-flag* input string is provived as input: `Simple` or `S-MOPS`. For the `MOPS-based` implementation, the size N of the keypoint's patch is also provided and is defined as a function of the scale σ of the patch, $N = 2\sqrt{2} \cdot \sigma$.

The function will output `Descriptors`, which is a matrix data struture that stores all keypoint's feature descriptors.

Feature matching

Now that you've detected and described your features, the next step is to write code to match them, i.e., given a feature in one query image, find the best matching feature in one or more other test images. This part of the feature detection and matching component is

mainly designed to help you test out your feature descriptor. You will implement a more sophisticated feature matching mechanism in future assignments.

The simplest approach is the following: write a procedure that compares two features and outputs a distance between them. For example, you could simply sum the absolute value of differences between the descriptor elements. You could then use this distance to compute the best match between a feature in one image and the set of features in another image by finding the one with the smallest distance.

Two possible distances are:

1. The SSD (Sum of Squared Distances) distance. This basically means computing the Euclidean distance between the two feature vectors. You must implement this distance.
2. The Ratio test. Compute (SSD distance of the best feature match)/(SSD distance of the second best feature match) and store this value as the distance between the two feature vectors. This is called the "ratio test". You must implement this distance.

Finally, you'll implement a function for matching features. You will implement the `FeaturesMatching` function of `SSDFeatureMatcher` and `RatioFeatureMatcher`.

```
function [Match] = FeaturesMatching(Descpt1,Descpt2,Tresh,'SSD' or 'RATIO')
```

To efficiently compute the $L2$-distances between all feature pairs from the two images, you can take advantage of the matricial representation used to store the descriptors. To determine the best match, you can use `Matlab` functions, such as the `min` function, but you can implement everything by yourself if you prefer.

The function will input the *features descriptors* matrices of both images under comparison and the distance threshold value for matching. Remember that these matrices can have different number of lines (number of features per image), but the number of columns (descriptor dimension) must be the same. The matching distance under use must also be defined as input.

The function will output a list of *Matched* patches. In each element of the list you should store the index of the feature in the query image, the index of the feature in the test image and the distance attribute between the two features as defined by the particular distance metric (e.g., SSD or ratio).

Show matching

To visually evaluate the performance of your algorithm you should add the necessary code to function `ShowMatching` in order to compose both images into a single image, plot all the detected keypoints and a draw the line connecting all matched keypoints. Figure 1 is a good example of the expected result.

The `ShowMatching` function takes as input the list of matched keypoints obtained using `FeaturesMatching` function, both images in comparison and its corresponding keypoint's descriptors.

```
function ShowMatching(Match,img1,img2,Descpt1,Descpt2)
```

By looking at the list of keypoints detected in both images and the corresponding set of matched keypoints you will easily realize that several keypoints pairs were not matched and some matched pairs are incorrectly associated. To better understand the reason

behind this behavior we challenge you to include in the `ShowMatching` function the ability to display the 8×8 (or 5×5) patch of a descriptor. In the canonical configuration both local descriptors (patches) should look invariant to scale and orientation. See figure 4 as an example of the expected result.

3 Experiments

Now you're ready to go!

Using the *HarrisScript* that we provide, you can load in a set of images, view the detected features, and visualize the feature matches that your algorithm computes.

After specifying the path for the directory containing the dataset, the script will run the specified algorithms on all images and compute a list of matching keypoints per image. You may use the outputs from each step to perform individual queries to see if things are working right. First you need to load a query image and check if the detected key points have reasonable number and orientation under the Keypoint Detection stage. Then, you can check if your feature descriptors and matching algorithms work using both the Feature Matching and Show Matching stages.

We are providing a set of benchmark images to be used to test the performance of your algorithm as a function of different types of controlled variation (i.e., rotation, scale, illumination, perspective, blurring).

In your report, you should describe how well your code worked on different images, what effect do the parameters have and any improvements you made to your code to make it work better.