Gabriel Pellegrino da Silva
Damoun Ayman
Image mining github repository
Master M2 IA

OPT10
Image Mining
October 8, 2020

Lab 1: Feature Detection, Description and Matching

0.1 Preliminaries

The aim of this work is to get familiar with the image processing library Python-OpenCV [7] and implement different representation methods for matching local structures in images. The main component of Feature Detection And Matching are listed below:

- **Detection**: Identify the Interest Point.
- **Description**: The local appearance around each feature point is described (changes in illumination, translation, scale, and rotation). We typically end up with a descriptor vector for each feature point.
- **Matching**: Descriptors are compared across the images, to identify similar features.

0.2 Image Format and Convolutions

Question 1
Experiment the convolution code given as example in Convolutions.py. Note the difference between the direct calculation and the calculation using function filter2d from OpenCV. Try to decrypt the OpenCV functions used for image reading and copying, and the MatPlotLib function used for image display. Explain why the convolution kernel provided as example realises a contrast enhacement with respect to the original image.

solution:

Convolution is a simple process during which we apply a matrix (also called a kernel or a filter) to an image so that we can downsize it, or add several padding layers to keep the size the same. Convolution is also used to extract specific features from an image, such as a shape, an edge, and so on.

The central element corresponds to a pixel of interest and the other elements correspond to that pixel's neighbors. On Convolutions.py the proposed kernel calculates enhaces the constrast of an image and to do so the pixel of interest has a weight of 5 and its immediate neighbors each have a weight of -1. For the pixel of interest, the output color will be five times its input color, minus the input colors of all eight adjacent pixels. The direct method for the calculation of the convolution product is expressed in the code below 1.

Listing 1: Direct method

```
for y in range(1,h-1):
  for x in range(1,w-1):
    val = 5*img[y, x] - img[y-1, x] - img[y, x-1] - img[y+1, x] - img[y, x+1]
    direct_method[y,x] = min(max(val,0),255) # fit the value between 0 and 255
```

OpenCV provides a very versatile function, filter2D 2, which applies any kernel or convolution matrix that specified to the given image. The second argument specifies the per-channel depth of the destination image. A negative value means that the destination image has the same depth as the source image. Here we apply a kernel of size 3x3 to the image. Note that the weights sum to 1. This should be the case whenever we want to leave the image's overall brightness unchanged.

Listing 2: filter2d from OpenCV

```
kernel = np.array([[0, -1, 0],[-1, 5, -1],[0, -1, 0]])
filter2d_result = cv2.filter2D(img,-1,kernel)
```

When we apply the code to the standard image FlowerGarden2.png, which is 240x360, the difference of execution time between using the OpenCV function and the Direct method is stark. While OpenCV takes 0.0032 seconds, the Direct method takes 0.2114 seconds to run, then about 100 times slower. Clearly, OpenCV is doing something more than our implementation. In fact, one of the things that's slowing our direct method is the min max step to achieve values inside the grayscale, but that's not the only difference, OpenCV is probably making a better use of the cache when doing the operations, since we acces the top and bottom neighbors of a point from a matrix, it's brought to memory the entire line and thus the cache are filled with not used data.

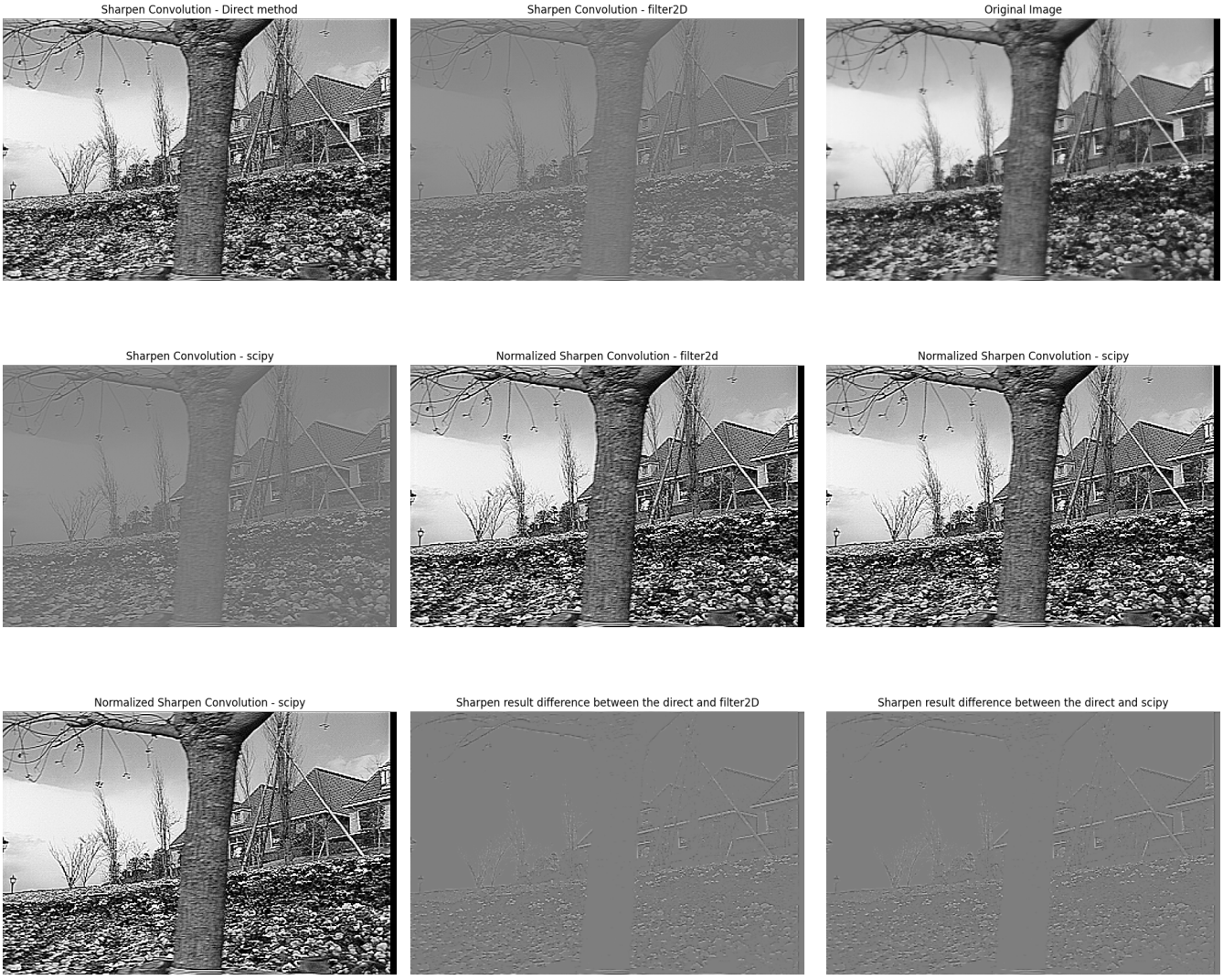

Figure 1: Application with the direct method, OpenCV and Scipy, normalized results and the difference.

The image 1 shows the image produced by both methods and also the image produced by the OpenCV method minus the image produced by our implementation, this minus operation calculates the difference between two images, if the final image is all black, thus they are equal, and where it's not black, it's different.

As one may notice on figure 1 the main difference between the proposed direct method and the function from these libraries, OpenCV and Scipy is that they do not normalize the data. One way to normalize it is to set every pixel with value higher than 255 to 255 and every pixel with value less than 0 to 0.

Two arguments on the imshow function of OpenCV can be used: vmax=255 and vmin=0, but also it's

possible operate on the image itself to get this result. The difference between the normalized version and the direct method is 0.

The function filter2D from OpenCV convolves a given kernel to a given image and matplotlib can be used to create figures, change their size, axis, legends, type of plots and so on. These two libraries are widely used both in small and big projects due to their quality and performance.

The selected kernel is known as the sharpen kernel because it sharpens the output image 1. This kernel can be calculated by calculating the Identity Kernel minus the Laplacian Kernel. The Laplacian Kernel is the sum of the second derivate aproximation both on x and y. By its format, it's seen that the final value after the convolution is an arrangement between the central point, which has a 5 weight and it's neighbors direct (0 manhattan distance), with a -1 weight. So, in a smooth region, where all the values are equal, this convolution kernel operates as an Identity Kernel, because the Laplacian Kernel does not contribute to the final value. However, if the central value is differente from its neighborhood, the final value might change in a factor of 5. One small example illustrate this idea is the matricial operation shown on 1. Notice that if you change the central 3 for a 2 in this image, the final result is 2.

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} * \begin{bmatrix} 2 & 2 & 2 \\ 2 & 3 & 2 \\ 2 & 2 & 2 \end{bmatrix} = 7 \tag{1}$$

1: Application of sharpen kernel on a really sharp region of a matrix.

Question 2
Modify the code to compute, with the two methods, a convolution that approximates the partial derivative I_y. Then use a (fast) method to compute a box sum image, i.e. Display the original and output images. What precautions should be taken in order to get a correct display (i.e. a correct interpretation of the grayscales)?

solution:

The partial derivate on y convolution kernel is $[-\mathbf{1}, \mathbf{1}]^\intercal$. Results for this kernel applied to an image are shown on image 2 and as seen on 3, there's no difference on the output because the output image using the filter2D function is printed using vmax=255 and vmin=0. Remember that the direct method always reference to the direct calculation with multiple for loops.

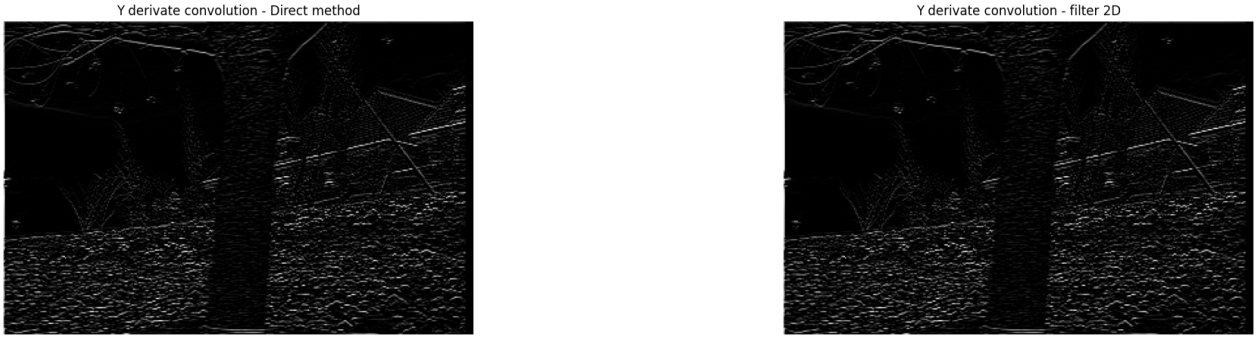

Figure 2: Partial derivate on y applied to an image.

The python library numpy provides a quick way to sum all the elements of a given matrix. So, we define a function $SI = \sum_{i=x-\rho}^{x+\rho} \sum_{j=y-\rho}^{y+\rho} I(i,j)$ as simple as the following code 3.

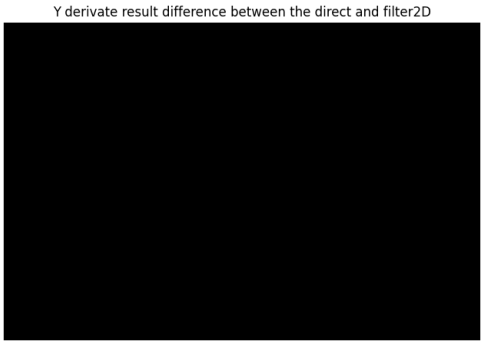

Figure 3: The difference between the two methods, a black image represents no difference.

Listing 3: Partial derivate on y

```
1 def SI(img, x, y, p):
2     val = np.sum(img[y-p:y+p, x-p:x+p])
3     return min(max(val, 0), 255)
```

A common problem when applying convolutional kernels and functions that operates over an image raw pixel values is that sometimes the final output value for a pixel will be higher than 255, however the maximum possible value in the grayscale value is 255; the same happens for negative values while grayscale minimun value is 0. One way to deal with this problem is to renormalize the image in order to make the higher values calculated go to 255 and the smaller values go to 0. As done in code 4.

Listing 4: normalize the image

```
1 x = min(max(val,0), 255)
```

The result is seen on the image 4 and the center is really brigher than the rest of the image because we're summing over an area and thus the pixel value exceeds most of the times the white value (255) and thus after the normalization it becomes white.

Figure 4: SI function applied with p=1 and on a square of side 50 on the center of the image.

0.3 Detectors

Question 3
Complete the code in the Harris.py script to compute the interest function of Harris (at one single scale), and the corresponding interest points. Can you understand how the suppression of non local maxima works in the provided code?
Comment the results obtained with your Harris detector and the effect of the used parameters, in particular the size of the summing window and the value of α. How is it possible to extend this

computation on several scales?

Using this OpenCV guide [3], the code provided by the professor is filled with these lines 5.

Listing 5: OpenCV Harris

```
blockSize = 2
apertureSize = 3
k = 0.04
# Detecting corners
Theta = cv.cornerHarris(Theta, blockSize, apertureSize, k)
```

The suppresion of non-local-maxima points is done using the dilate operation of openCV [6]. As see on the image 5, this operator increases the white area on the image because if there's at least one 1 bit on the kernel area, the final pixel value is 1. With this, the local maxima area are more or less stabilised and gain a little area too. After that the following two lines remove all the non-local maxima points 6.

(a) Toy image (b) Dilated image

Figure 5: Dilation operator example.

Listing 6: Supression of non-local-maxima

```
Theta_maxloc[Theta < Theta_dil] = 0.0 # Suppression of non-local-maxima
Theta_maxloc[Theta < seuil_relatif*Theta.max()] = 0.0 # Remove small values
```

The technique to remove non-local maxima puts to 0 all the points on the Harris function lower than the dilated Harris function. And also puts to 0 all the points lower lower than 1% of the max value of the Harris function. After that, every point that remains non zero is convolved with a kernel very specific 2, this kernel draws a cross on the image on every position that remains non zero.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

2: Cross kernel.

On this figure 6 the original image with the calculated Harris function using the openCV library and the selected Harris points of interests are shown. Also, the same results are obtained using a homemade implementation of the Harris method, look at the difference on image 7. One may notice that this detector is specially interested in corners. The second and third figure are produced using $\alpha = 0.04$ and summing windows size of 3. Furtherless, our version of the Harris algorithm, a threshold of 3000000 is used to filter only the points with high R value, see the equation below.

$$R = det(M) - \alpha trace(M)^2 = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$$

R will only be high when both eigenvalues of M are high. Also, the α parameter lets you influence in this step, trading off precision and recall. So with a bigger α, you will get less false corners but you will also miss more real corners (high precision), with a smaller α you will get a lot more corners, so you will miss less true corners, but get a lot of false ones (high recall).

As we increase the value of α the less the points of interest are found. And if we increase the size of the summing window, the Harris function becomes blurrier and thus the Harris points found start to superpose each other, till the point where the size of the summing window is so high that the Harris function no longer represent the contour of the input and is just a blurry image.

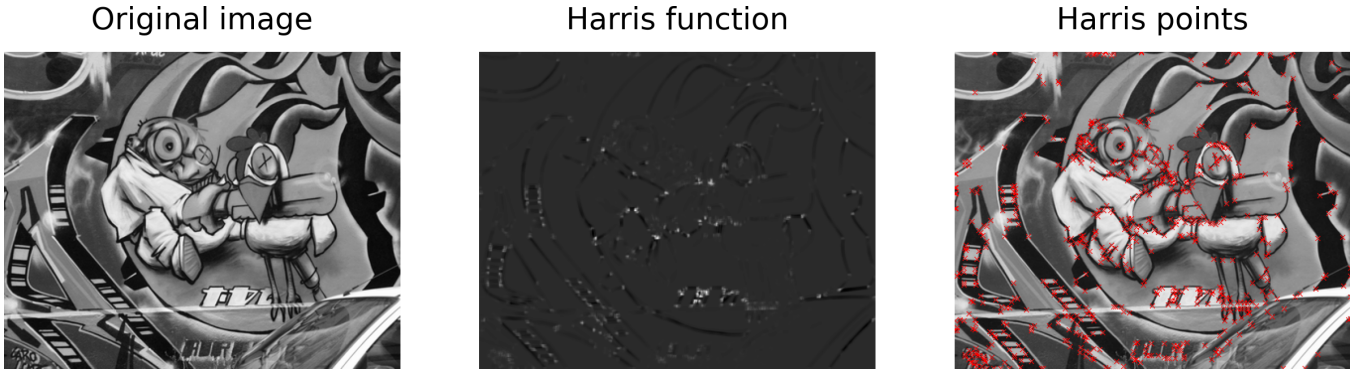

Figure 6: Harris operator aplied to an image and its result, using openCV library.

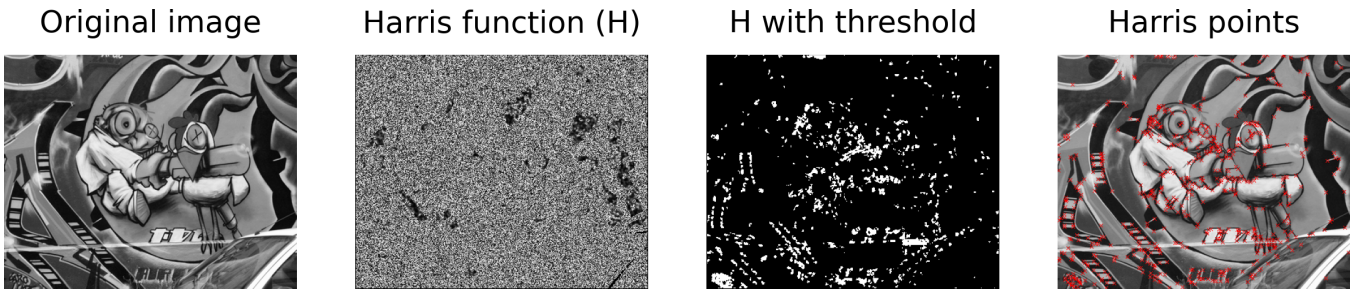

Figure 7: Harris operator aplied to an image and its result, using homemade implementation.

The Harris result displayed on 6 and 7 uses only one scale. On this case, scale means the number in depth of applications of gaussian filters to smooth the image and thus get fewer, but sometimes more precise, number of interest points as the number of scale grows. That's because, as the number of scale increases, the smoother the image and the more it gives the impression that the main aspects of the image are on highlight, as it gains the shape as if it was observed from a bigger distance [4].

The following implementation of the Harris OpenCV detector is used 7. This function calculates the Harris function values and apply the defined threshold, as seen on 7. To get the final result of the interest points, the same metodology is applied using the code provided by the professor, Harris.py.

Listing 7: Harris function

```
1 def harris(input_img, k, window_size, threshold):
2
3     harris_function = input_img.copy()
4     harris_function[:,] = 0 # make it black
5
6     thresh_harris_function = input_img.copy()
7     thresh_harris_function[:,] = 0 # make it black
```

```
8
9       offset = int(window_size/2)
10      y_range = input_img.shape[0] - offset
11      x_range = input_img.shape[1] - offset
12
13
14      dy, dx = np.gradient(input_img)
15      Ixx = dx**2
16      Ixy = dy*dx
17      Iyy = dy**2
18
19      for y in range(offset, y_range):
20          for x in range(offset, x_range):
21
22              #Values of sliding window
23              start_y = y - offset
24              end_y = y + offset + 1
25              start_x = x - offset
26              end_x = x + offset + 1
27
28              #The variable names are representative to
29              #the variable of the Harris corner equation
30              windowIxx = Ixx[start_y : end_y, start_x : end_x]
31              windowIxy = Ixy[start_y : end_y, start_x : end_x]
32              windowIyy = Iyy[start_y : end_y, start_x : end_x]
33
34              #Sum of squares of intensities of partial derevatives
35              Sxx = windowIxx.sum()
36              Sxy = windowIxy.sum()
37              Syy = windowIyy.sum()
38
39              #Calculate determinant and trace of the matrix
40              det = (Sxx * Syy) - (Sxy**2)
41              trace = Sxx + Syy
42
43              #Calculate r for Harris Corner equation
44              r = det - k*(trace**2)
45              harris_function[y,x] = r
46
47              if r > threshold:
48                  thresh_harris_function[y,x] = 1
49
50      return harris_function, thresh_harris_function
```

Question 4
Experiment and compare the two detectors ORB and KAZE by running the script Features Detect.py. Recall the principles of each detector, and play with their major parameters. How is it possible to visually evaluate the repeatability of each detector applied on a pair of images?

solution:

ORB (Oriented FAST and Rotated BRIEF) is a fast binary descriptor based on the combination of the FAST (Features from Accelerated SegmentTest) keypoint detector [9], [8] and the BRIEF (Binary robust independentelementary features) descriptor [1]. It is rotation invariant and robust to noise.

KAZE feature employs the normalized Hessian matrix to detect local maximum and determines the main direction of the feature points in nonlinear scale space. The maxima of detector response are picked up as feature-points using a moving window. Feature description introduces the property of rotation invariance by finding dominant orientation in a circular neighborhood around each detected feature. KAZE features are invariant to rotation, scale, limited affine and have more distinctiveness at varying scales with the cost of moderate increase in computational time.

Figure 8: Feature detection with ORB (image 1 and 2)

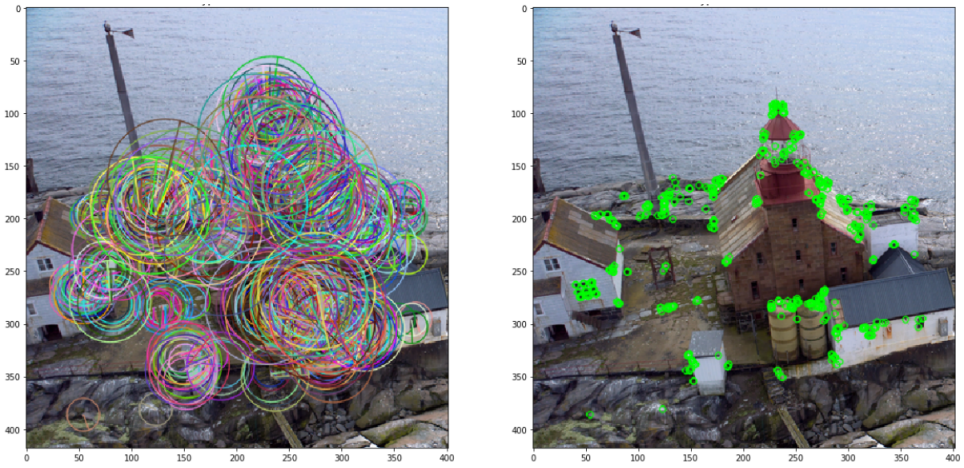

Figure 9: keypoints detected with ORB (with and without cercle size)

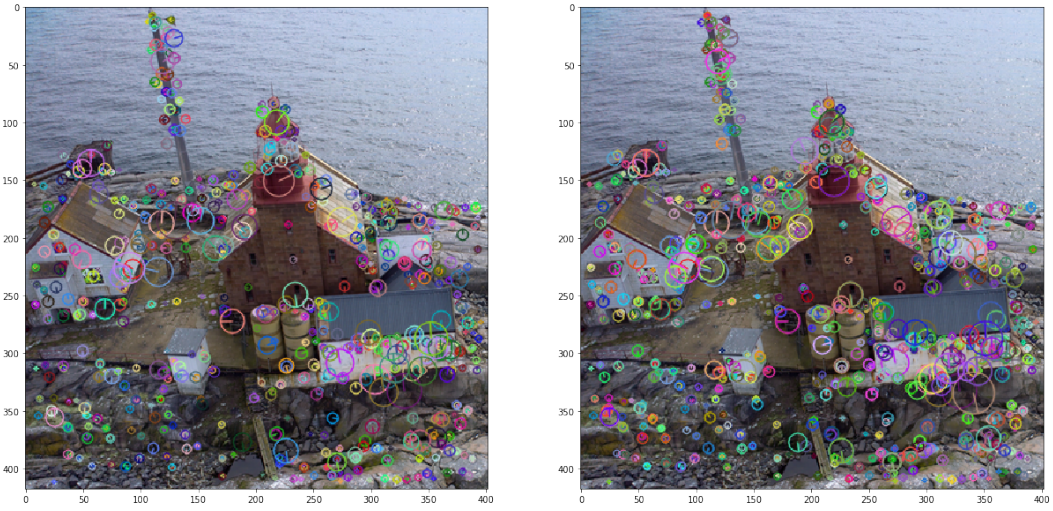

Figure 10: Feature detection with kaze

The table below provides a comparison between the how detectors based on the experiment results

Detector	ORB	KAZE
Detection of keypoints times	0.1946 s	0.3319 s
number of keypoints	500	815

Table 1: Comparison between ORB and KAZE of one trial

Repeatability

To visually evaluate the repeatability of each detector, we can plot the key points detected, for example for 3 tests on the same figure. We can clearly see that on the figures below (11) the three colors (red, blue and green) of each test. This means that for this image both methods present a random parameter that prevents perfect repeatability. We can also see in figure 14, that for the kaze detector the number of keypoints detected varies.

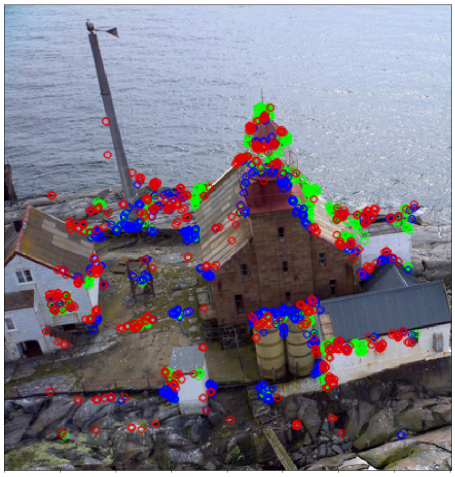

Figure 11: ORB repeatability test with 3 tests

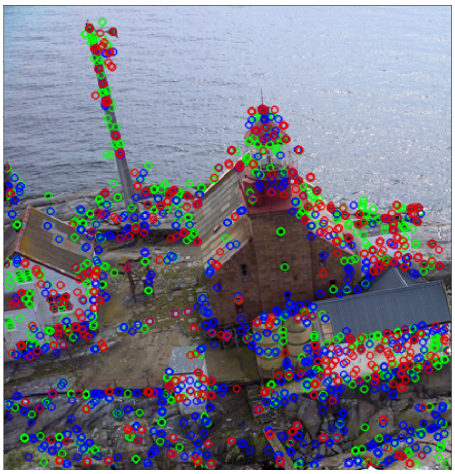

Figure 12: KAZE repeatability test with 3 tests

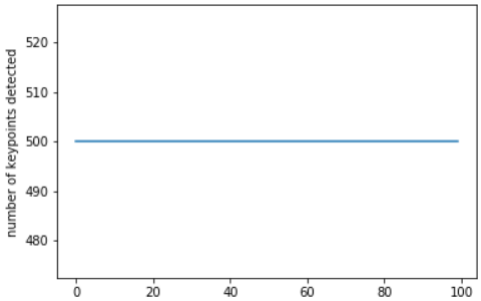

Figure 13: Number of keypoints detected with ORB for 100 repeatition

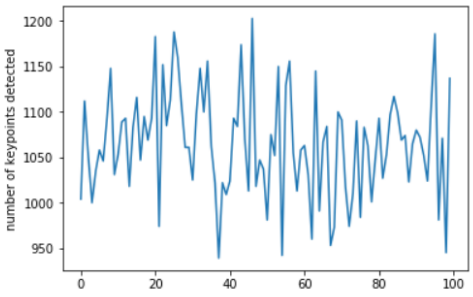

Figure 14: Number of keypoints detected with KAZE for 100 repeatition

0.4 Description and Matching

Question 5
Find the principles of the descriptors attached to points ORB and those attached to points KAZE. What properties of detectors and/or descriptors (distinguish the two aspects in your answer), allow to make the matching scale and rotation invariant?

solution:

KAZE

Steps in KAZE Detection:

- Build nonlinear scale space using AOS and a set of octaves O and sublevels S

$$\sigma_i(o,s) = \sigma_0 2^{o+s/S} \quad , o \in [0...O-1] \, , s \in [0...S-1]$$

- We need to mapscale unitstotime units.

$$t_i = \frac{1}{2}\sigma_i^2, i = 0...N$$

- Detect features in the nonlinear scale space computing the determinant of Hessian response and perform non-maxima suppression in image and scale coordinates.

$$L_{Hessian} = \sigma^2(L_{xx}L_{yy} - Lxy^2)$$

First and second order derivatives are approximated by means of 3×3 Scharr filters of different derivative step sizes σ_i. Scharr filters approximate rotation invariance significantly better than other popular filters.

Steps in KAZE Description:

- For each detected keypoint, estimate adominant orientation if desired or use upright version.
- Compute KAZE descriptor according to the dominant orientation and using multiscale derivatives obtained from the nonlinear scale space.
- Descriptor similar to M-SURF but computed from the non linear scale space.

ORB

basic ORB Algorithm

- Find the position of thr key positions by FAST.
- Selecting N best points by Harris
- Scale-pyramid transform.
- Add a direction of the points in intensity Centroid.
- Extracting Binary descriptor By BRIEF.
- Get steered BRIEF.
- Find low correlative pixel blocks in greedy algorithm.
- Receive a 256 bit descriptor.

ORB mixes the techniques used in the FAST keypoint detector and the BRIEF keypoint descriptor, so it is worth taking a quick look at FAST and BRIEF first.

FAST:The Features from Accelerated Segment Test (FAST) algorithm works by analyzing circular neighborhoods of 16 pixels. It marks each pixel in a neighborhood as brighter or darker than a particular threshold, which is defined relative to the center of the circle. A neighborhood is deemed to be a corner if it contains a number of contiguous pixels marked as brighter or darker.

BRIEF:Binary Robust Independent Elementary Features

- Binary descriptor based on pairwise intensity comparisons
  - Low storage requirements
  - Fast to compute and match. Hamming distance is extremely fast in some architectures
  - BRIEF is not rotation nor scale invariant
- Pairwise comparisons are sampled from an isotropic Gaussian distribution

$$d_K(p) := \sum_{1 \leq i \leq n} 2^{i-1} \tau(p; f(i), f(j))$$
$$\tau(p; f(i), f(j)) := \begin{cases} 1 & , \quad f(i) \geq f(j), i \neq j \\ 0 & , \quad f(i) < f(j) \end{cases}$$

In the literature a more detailed comparison of the different detectors can be found. Based on [11] we can list some differences between ORB and KAZE:

- KAZE has the highest computational cost for feature-detection-description.
- ORB is the most efficient feature-detector-descriptor with least computational cost
- Repeatability of ORB for matching images remains stable and high as long as the scale of test image remains in the range of 60% to 150% with respect to the reference image but beyond this range, its repeatability drops considerably.
- Repeatability of KAZE remains high for up scaling of images while in the case of down scaling, the repeatability of KAZE drops significantly.
- Although KAZE has the highest cost for feature-detection- description, the computational cost of matching KAZE feature-descriptors is surprisingly lower than ORB .

Question 6
Explain and compare qualitatively the effects of the three different point matching strategies performed in the three scripts Features Match CrossCheck.py, Features Match RatioTest.py and Features Match FLANN.py. Explain why the used distances are different for the two descriptors. Explain why the strategy FLANN (and, in a lesser extent the strategy RatioTest) does not work well with ORB points. Would it be possible to evaluate quantitatively the different matching strategies? If yes, explain how.

solution:

Feature matching between images can be done with multiple heuristics, here three main methods are presented and discussed. One may hope that the found interest points for a given image might be the same if two different algorithms, if they have the same purpose. And also it's supposed that two images, if they do not differ a lot, they should present similar interest points, because these two similar images are indeed close one to the other. Thus, one way to match to define a threshold and if the distance between a keypoint A on the first image and a keypoint B on the second image is above a defined threshold τ, then a match is made. Also, it's necessary to define what happens to a keypoint with multiple matches [5]. Again, for similar enough images, the number of features detected should be the same or at least close to each other and thus this matching approximates to the famous stable marriage problem.

Now, about algorithms to do matching of features.

- **Brute-Force (BF) matcher**: BF Matcher matches the descriptor of a feature from one image with all other features of another image and returns the match based on the distance. It is slow since it checks match with all the features. Note that many distance metrics can be used. It's common to use hamming distance on this case, but euclidean distance is possible too.

- **FLANN based matcher**: Fast Library for Approximate Nearest Neighbors (FLANN) is optimised to find the matches with search even with large datasets hence its fast when compared to Brute-Force matcher. The FLANN method is based on a training and thus a comparation of a new input to the previous knowledges and the match is made on a nearest neighbor search. Also, a ratio test is done on the proposed matches and if it's below a threshold, they are added up to a list of good matches.
- **Knn based matcher**: a branch of the brute force algorithm, that uses the k-nearest neighbors method (knn) to match the features from one image to another. Also, a ratio test is done on the proposed matches and if it's below a threshold, they are added up to a list of good matches.

Two possible distance metrics used for the brute forces based algorithms.

- **L2 norm**: classical euclidean distance between two points. Used with Kaze descriptor.
- **Hamming distance** [**2**]: calculates the difference between two sequences as the number of changes necessary to go from one element to another. Used with ORB descriptor.

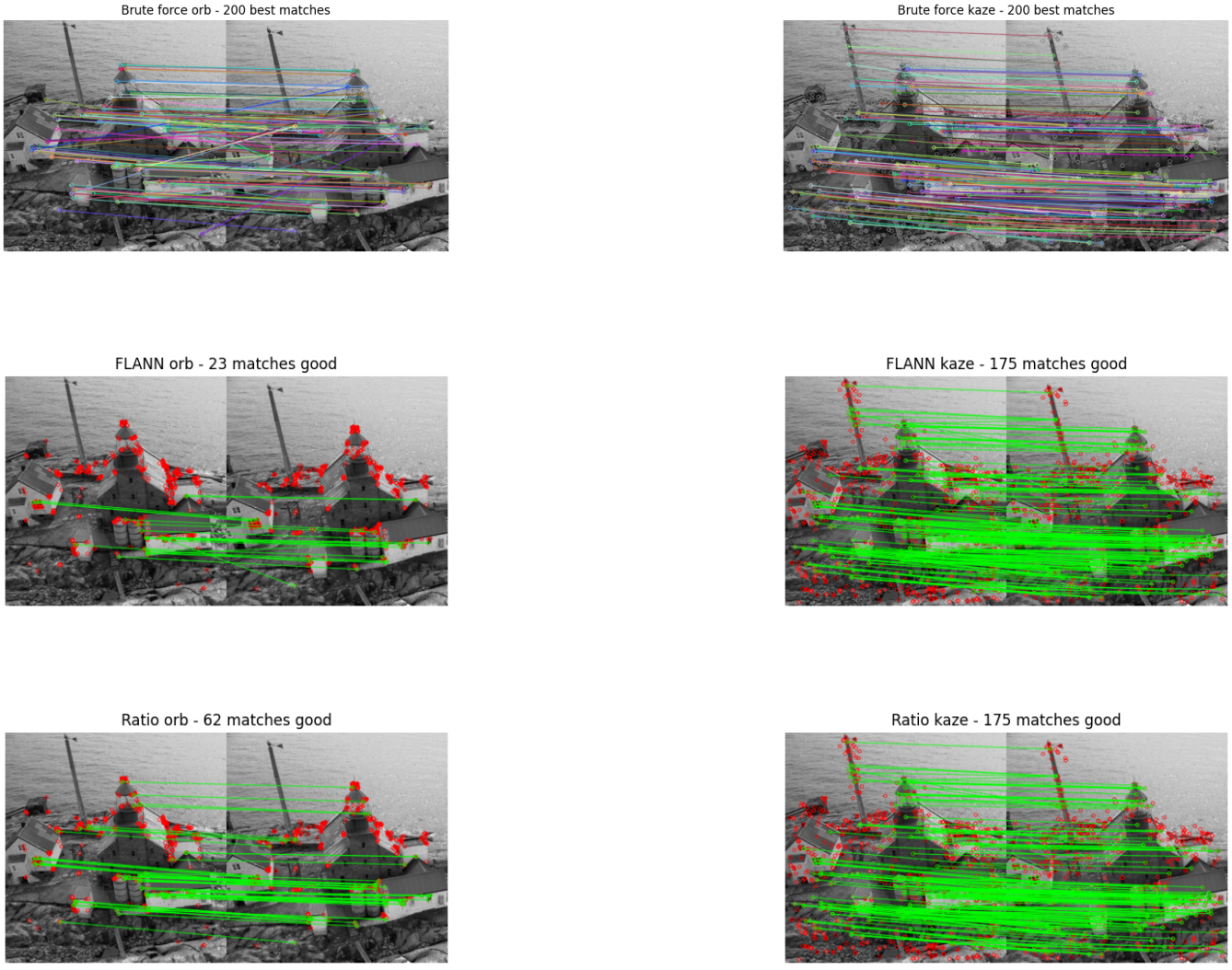

Figure 15: Qualitative results between ORB and KAZE with different matching strategies.

In order to qualitatively evaluate the images present in figure 15, it's possible to analyse them separately and then combine the analysis. In general, kaze display better results, one way to visualize it is seeing that the points matched, the line between them is more or less the same size and have more or less the same inclination. What does not happens for many of the orb descriptor images. As it's possible to see,

there are many lines crossing each other. One may also say that orb produces more matches in general, for the three matching algorithms studied. What isn't a bad thing necessarily, having a few but really representative matching points might be better than many repeated close points found as a match.

More on that, one of the best results are the one produced by the brute force algorithm with kaze, 200 best matches were found.

Also, the orb algorithm uses the hamming distance because orb is a binary string based descriptors and thus to compare the distance between strings of the same size, the hamming distance works more properly than L2 norm. On the other side, the kaze algorithm uses the L2 norm as a distance because the descriptor doesn't operate on a string base description and then a L2 norm is more than enough to calculate the distance between two given features found.

The FLANN methodology for feature matching consists of a comparison between the features of an image and finds out if it is a prospective known feature by matching with some predefined features. One problem of this is that descriptor algorithms, such as orb aren't stable on the sense of get the same features over different iterations and even further, the found features might be close but not intersect each other. And it's because of that that FLANN has dificulties to match ORB generated features [10]. Using the same line of reasoning, the Ratio Test with Brute Force doesn't work so well, but in a lesser way because the Ratio Test approach is using a Knn to found the matches and even when there's no intersection, they might be close enough to be matched correctly.

Many heuristics can come to mind to evaluate quantitatively different matching strategies. A simple idea is to trust on the feature descriptor algorithm and bellieve that it will provide good feature points and based on this, the quality of the matches found should be a ponderation between the number of matches found in relation to the total number of feature found, but you could take in account also the fact finding matches doesn't mean that you find good matches. So, in order to ponderate the quality of your matches, it's possible to score them, as it was proposed earlier, matches should have more or less the same eucliean distance on the image map, more or less the same inclination between them and if they don't, it's a sign that this possibly isn't a traditional match that should be found.

Bibliography

[1] Michael Calonder et al. "BRIEF: Binary Robust Independent Elementary Features". In: *Computer Vision – ECCV 2010*. Ed. by Kostas Daniilidis, Petros Maragos, and Nikos Paragios. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 778–792. ISBN: 978-3-642-15561-1.

[2] *Hamming distance*. `https://en.wikipedia.org/wiki/Hamming_distance`. Accessed: 2020-10-05.

[3] *Harris detector*. `https://docs.opencv.org/3.4/d4/d7d/tutorial_harris_detector.html`.

[4] Agustín Salgado Javiser Sánchez Nelson Monzón. *An Analysis and Implementation of the Harris Corner Detector*. `http://www.ipol.im/pub/art/2018/229/article_lr.pdf`.

[5] Krystian Mikolajczyk and Cordelia Schmid. *A performance evaluation of local descriptors*. `http://lear.inrialpes.fr/pubs/2005/MS05/mikolajczyk_pami05.pdf`.

[6] *Morphological Transformations*. `https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html#dilation`. Accessed: 2020-09-26.

[7] *opencv*. `https://opencv.org/`.

[8] E. Rosten, R. Porter, and T. Drummond. "Faster and Better: A Machine Learning Approach to Corner Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.1 (2010), pp. 105–119.

[9] Edward Rosten and Tom Drummond. "Machine Learning for High-Speed Corner Detection". In: *Computer Vision – ECCV 2006*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443. ISBN: 978-3-540-33833-8.

[10] D. A. Suju and H. Jose. "FLANN: Fast approximate nearest neighbour search algorithm for elucidating human-wildlife conflicts in forest areas". In: *2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*. 2017, pp. 1–6.

[11] Shaharyar Ahmed Khan Tareen and Z. Saleem. "A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK". In: *2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)* (2018), pp. 1–10.