Maman 11

Harris Detector

- Using harris detector to detect important keypoints, We have several parameters,
- block_size: size of neighborhood considered for corner detection.
- ksize: k size, define the size / kernel of the sobel operator. (3 for 3x3, 7 for 7x7 etc.)
- k or sigma – a constant that makes the formula work (0.04-0.06)
- threshold_factor – which helps determine which points are important and which are trash.
- Applying the algorithm will extract np.array of keypoints [x, y]. So 2darray.

Problems

- The detector extract several "very close points". We want only 1 point when we do evaluation. We can solve it using non maximum suppression. In the work I solved it using a more sophisticated algorithm DBSCAN which dynamically scanning for clusters of points with certain distance (we use default which is Euclidian).

Rotation

- When using rotation, we get a general problem that image boundary changes. We solve it using parameters from the rotation matrix. I did not delve deep into it to understand the 2x3 rotation matrix. But using these parameters, and some trigonometry, we can calculate the new edges of the rotated image.
- Rotation arises a new problem where, if we want to detect which point is which (matching), we lose the original coordination. Therefor we need to develop a function that given the angle of rotation, restore the points to the original coordinates (it is also possible to use the rotation matrix created above).

Calculation

- After rotating the points to original x, y, We face a computational problem, where we can brute force match the points (~n*~n), or use something more sophisticated like Kdtrees (~nlog(~n)). I use ~n because we expect to find order of n points.
- There is another art, when picking Harris block_size together with DBSCAN eps (proximity of search). You can predict the radius of noise and use a suitable eps value, to reduce computation.

Final point about Harris detector

- After many failed attempts to get the original points with respect to rotation so I can make comparison between original image to rotated image, I gave up. I think the last problem I didn't sold is the offset cause by image expanded after rotation. It just took too much of my time.
- Lastly, I gave up on evaluating this method even though I spent a lot of time over it.
- It also didn't match the general behavior of the code. It needed many cases of it's own to solve.

FAST (Features from Accelerated Segment Test)

- Popular corner detection algorithm, known for its speed and efficiency.
- The algorithm is testing candidate points using their surrounding pixels. Depending on the intensity of brighter/darker around the candidate.
- This algorithm extracts a Keypoints object, which will help me take a DESCRIPTOR algorithm so that I can match the original keypoints with augmented keypoints.

<u>Parameters</u>

- Threshold – helps decide the intensity of pixels around the candidate.
    - High value – fewer detect corners, I tried 10 and 20.
- Non maximum suppression – A general term we've learnt in class, which reduce multiple points that repeat themselves in a very small perimeter to only one point. We used it (True)
- Type: some cv2.CONSTANTS (ints) which determine how many surrounding pixels to test. I used TYPE_9_16

ORB (Oriented FAST and Rotated BRIEF)

- This, and the following algorithms will return descriptors, which will make our life **MUCH** easier in evaluating the algorithms over augmented images.
- Using FAST to detect key points.
- ORB calculates "orientation" which is based on the intensity of the values surrounding the key point. (based on, guess what, **GrAdIeNt** direction)
- Result is binary string rather then floats (which found in SIFT and SURF)
- ORB is rotation invariance
- Can be Scale invariance with some modifications
- Allows fast matching using Hamming Distance (Hamming because binary)
- Very fast.

<u>Paramteres</u>

- Nfeatures: 500
- scaleFactor: 1.2 (how much to reduce the image between each step in the pyramid) 1.2 is 20%
- nlevels: 8 (the levels of the pyramind)
- edgeThreshold: 31
- firstLevel: 0
- WTA_K: 2 – number of points to use for BRIEF dalc
- There are more, I just kept the defaults. These are also the defaults generally recommended.

Akaze(Accelerated KAZE)

- I do not fully understand and did not dive deep on how this algorithm works.
- It's fast.
- Rotation and scale invariant
- Using binary descriptor

- Robust to noisy images.

Parameters

- Descriptor_type – KAZE
- Descriptor_size: used 0 which defaults to 64
- Descriptor_channels: 1 (gray)
- Th: 0.001
- N_octaves: 4
- N_otaves_layers: 4

Calculations

repeatability

- Repeatability were computed only on points that matched and had a distance < 5 px. Repeatability dominator was the original number of keypoints (before augmentation)

Localization error

- Also was only calculated once a match was smaller than the above distance. (Because once a match is too far away, it should NOT be considered a match and should not be relied upon.

Match precision

- Iterating over all matches, if their distance was lower then 5 px, this would be a tp. Otherwise fp. Finally, tp / (tp + fp)

Calculation time

- The time from the first kp, descriptors calculation, until and include calculating match precision.

Pythons Code

Attached all the code, images and required dirs.

Requirements.txt
Readme.md
All code is heavily documented.
Just run maman11.py

All the code is in one module maman11.py for comfortability of read. There are 3 sections, helpers functions, augmentations and detector/descriptor. After that it's just the processing.

In the main function there are initializing of the augmentations and descriptors, together with their matching algorithms.

The code is quite generic, and it's easy to change/remove/add augmentations and detectors.

There is additional python file, called legacy.py which holds some of the code previously written and played with. For example, I moved the harris_corner_detection from my work because it wasn't generic enough and required a lot of work to make generic, test it works properly (matching and evaluating).

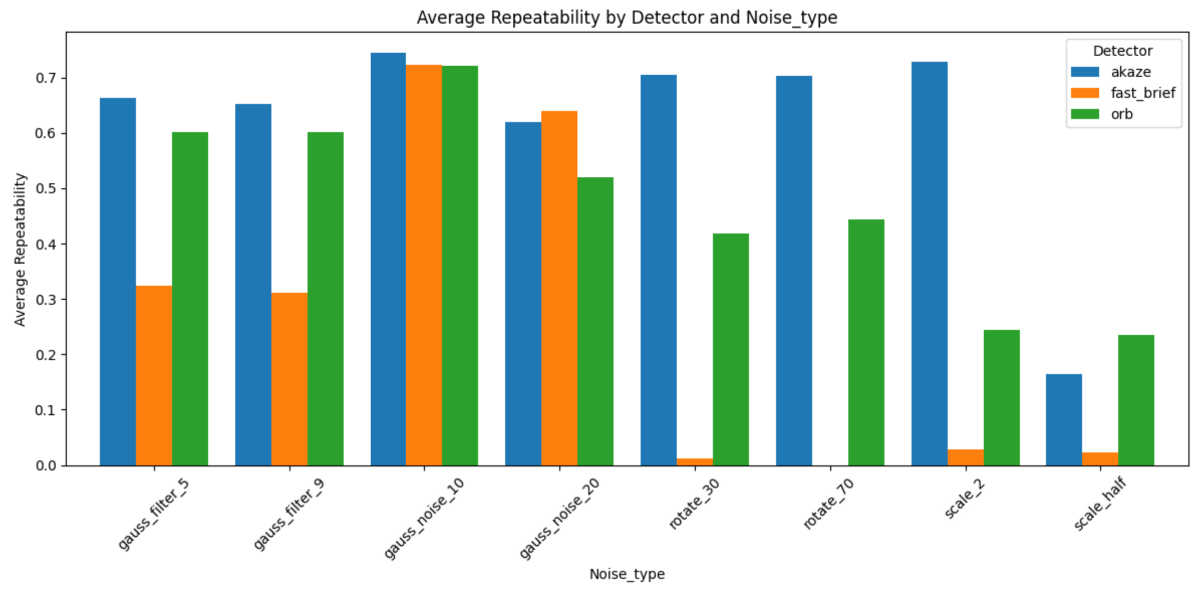
I've also added a rescale to all images to be 512, 512. This can be removed by not sending the **args argument to the main function.

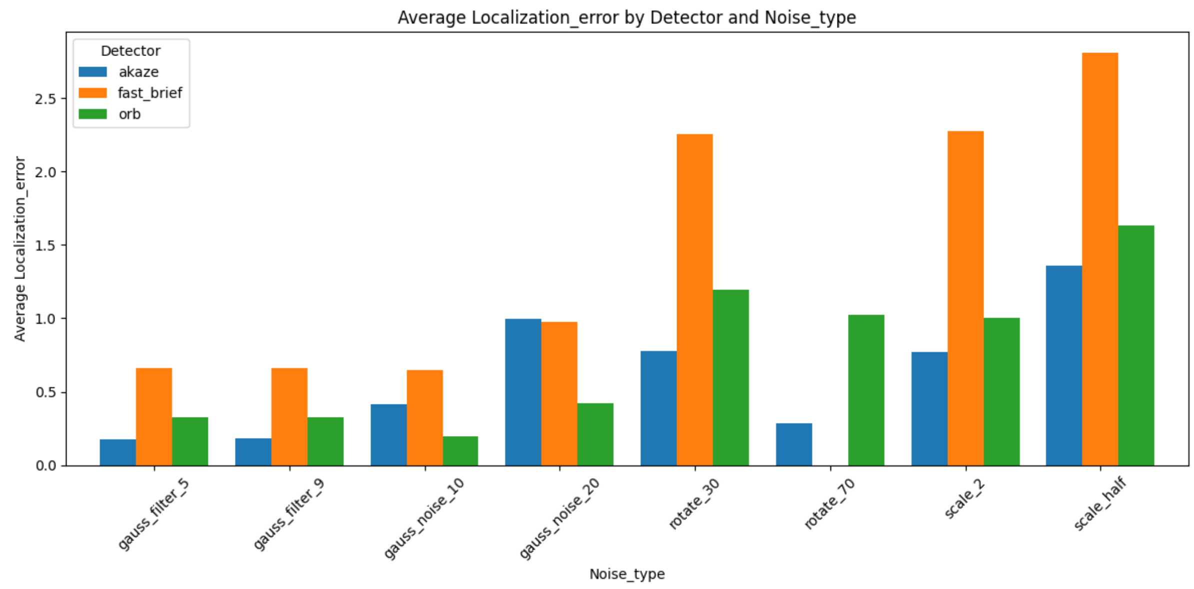
Code result will output a csv containing all data gathered from analyzing the images,

Detector, noise, eval_metric, index(image), value.

Part 1

Drawings:





Strengths and weaknesses:

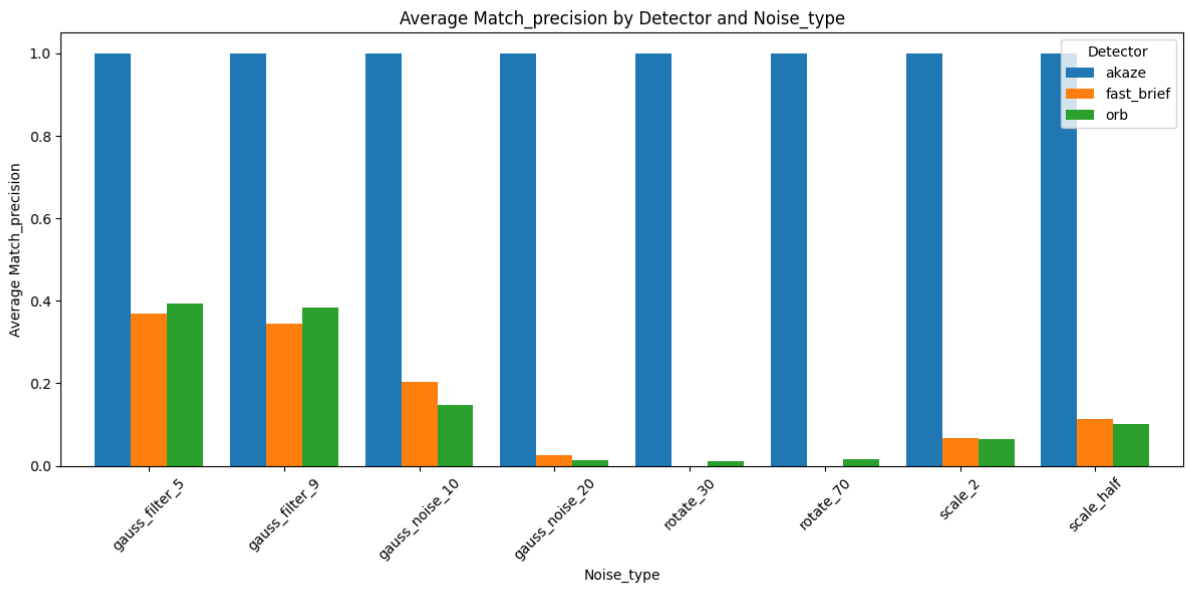
1. We will focus on Akaze vs orb, as it looks like fast + brief is just inferior.
2. In terms of repeatability looks like Akaze and orb are about the same using gauss filter and noise. But when it comes to rotation Akaze prevails. When it comes to scale looks like enlarging the image keeps Akaze ahead but when reducing the image, they are roughly the same.
3. In terms of localization error, we can see fast_brief has the largest errors, especially in rotation and scaling.
4. Akaze has smaller error in gauss filter, rotation and scaling.
5. ORB has smaller error in gauss noise
6. Differences are usually quite small.

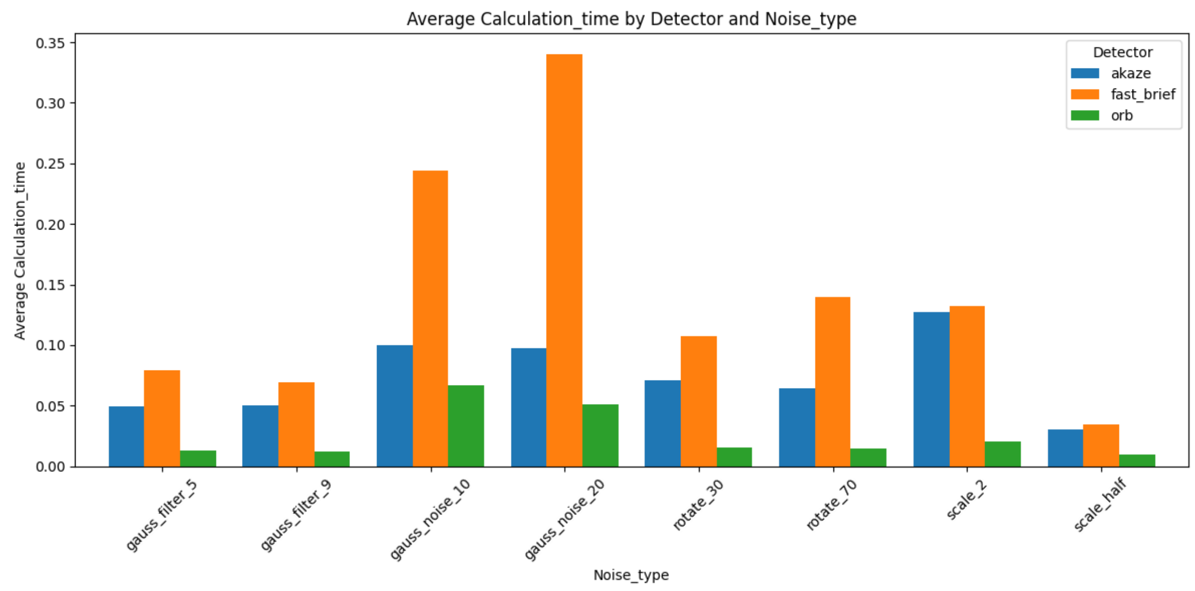
Recommendation:

- I'd probably use Akaze.
- I wouldn't mind using ORB as well.

PART 2

Drawings:





<u>The recommended descriptor is….</u>

Akaze by FAR!!!

All the points were a match (under l2 distance of 5 pixels).