

Computer Vision 046746

HW 01

April 20, 2020

Sahar Carmel 305554453
Nadav Gelfer 304849821

1 Keypoint detector

Summary

After loading the image, converting to gray and normalizing we created the DoG pyramid using the provided code. To implement Q1.4, we used *cv2.Sobel* to calculate D_{xx} , D_{yy} and D_{xy} for each level of the pyramid. Using the derivative matrix, we calculated the curvature ratio, R , pixelwise and stored the results.

In order to implement Q1.5 with reasonable complexity, we used masks to filter possible extrema points which do not satisfy the θ_c and θ_r threshold. Such approach was required because the implementation of spatial extrema point detection has far greater complexity, and because we found empirically that only about 10% – 15% of the extrema point satisfy both thresholds.

Q1.6 simply wraps all of the implemented functions.

1.1

The loaded image is simply:



Figure 1: Loaded image

We've used *cv2.cvtColor* to transform from *BGR* layout to *RGB*.

1.2

We converted the image to grayscale using `cv2.cvtColor` and normalized its values to $[0, 1]$ using division by 255.

Using the attached function we receive the following pyramid:



Figure 2: Gaussian pyramid

The shape of the GaussianPyramid matrix is $1 \times \text{len}(\text{levels})$ where each element is of size $\text{imH} \times \text{imW}$.

1.3

The output pyramid is as follows:



Figure 3: Difference of Gaussian pyramid

Where each element is received by simply subtracting the previous level.

The shape of the matrix is similar to that of the Gaussian pyramid and differs only by having one less level due to the nature of the calculation, namely $1 \times (\text{len}(\text{levels}) - 1)$ where each element is of size $\text{imH} \times \text{imW}$.

1.4

We used a Sobel filter with a kernel size of 3 to calculate the derivatives D_{xx} , D_{yy} and D_{xy} . Then we calculated the curvature ratio, R , pixelwise for each level.

To avoid division by zero in the case of $\det(H) = 0$, we set the value of $\det(H)$ to `numpy's epsilon` in such case.

1.5

In order to satisfy both thresholds and find all spatial extremums, we've used masks:

```

1 mask_c = np.abs(im) > th_contrast
2 mask_r = np.abs(PrincipalCurvature[level]) < th_r
3 mask_unified = mask_c & mask_r
4 valid_pixels = np.where(mask_unified == True)

```

Using masks allowed us to skip the costly action of calculating a 10-neighborhood extrema for each pixel in the image by allowing us to loop over a much smaller set of pixels which satisfy both thresholds.

For each pixel that satisfies both thresholds, we considered the pixel an extrema point if it was either the maximum or the minimum of it's neighbors.

1.6

Detected keypoints for the input image *model_chickenbroth.jpg*:



Figure 4: Keypoints for input image *model_chickenbroth.jpg*

Using the same code for *pf_pile.jpg* yields the following result:

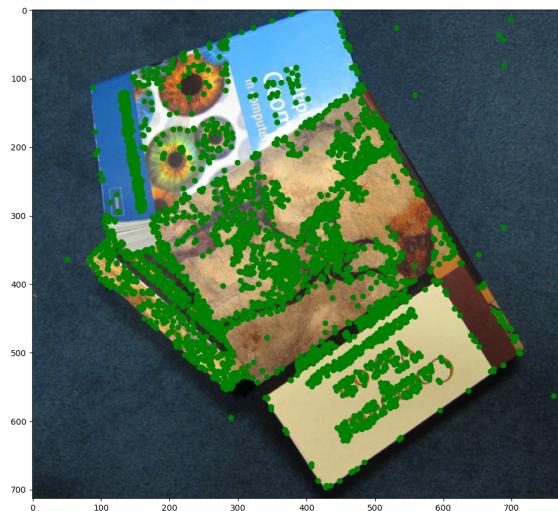


Figure 5: Keypoints for input image *pf_pile.jpg*

Using a picture we've taken outside and resized to 512×512 we get:

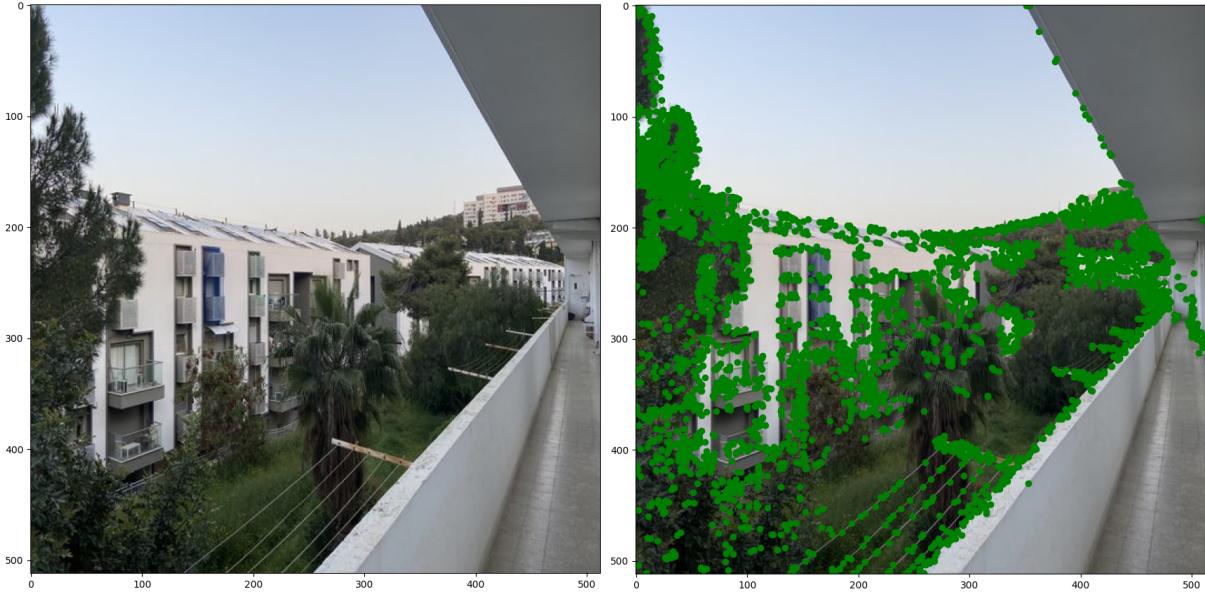


Figure 6: Our image and it's keypoints, side by side

From the above samples it's obvious that there are many keypoints and not all of them describe a desirable feature. In order to improve the results, we used a Gaussian blur filter to smoothen the image. Such action reduces the effect of noise and further enhances the detection of corners over edges. The same picture we've taken after applying `cv2.GaussianBlur` yields the following results:

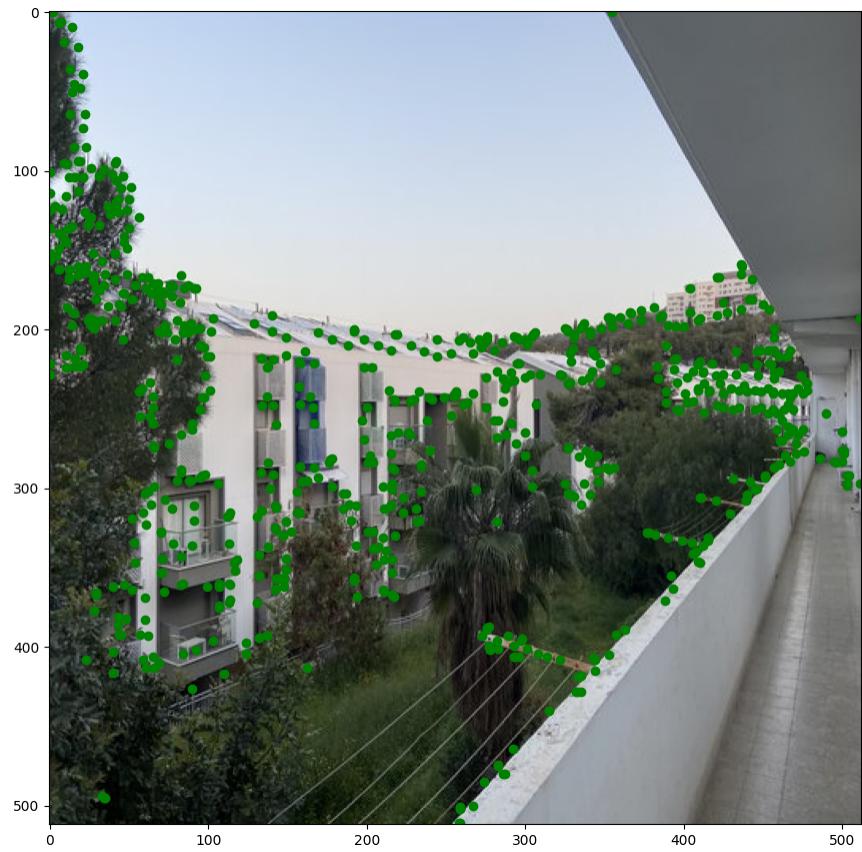


Figure 7: Our image and it's keypoints after applying Gaussian blur

The effect of the blur shows quite well using a simple polygons picture that was posted on Facebook:

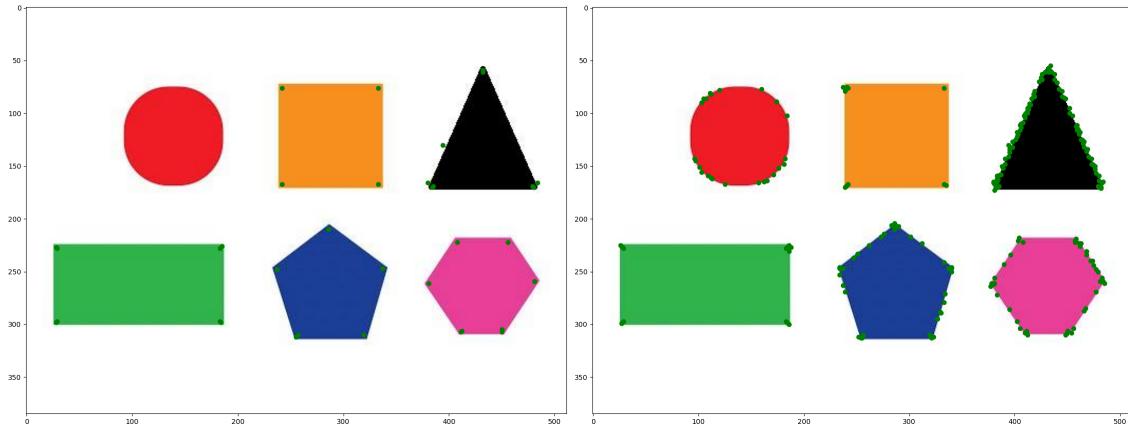


Figure 8: Keypoints for the blurred picture compared to the non-blurred picture

This image demonstrates quite well how smoothing the picture using a Gaussian blur eliminates most false detections which occur due to noise and pixelation along the edges of the polygons.

The effect is shown easily by comparing the corner of the orange square zoomed in:

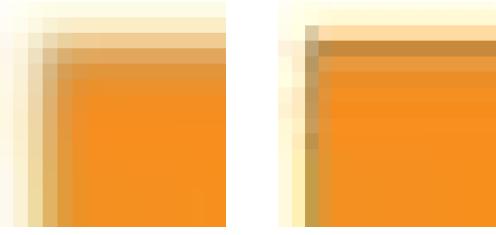


Figure 9: Blurred corner compared to the original corner

2 BRIEF descriptor

Summary

Something

2.1

We've chosen to randomize our X, Y coordinates using a uniform distribution. We used `numpy.random.randint` for both X and Y with a lower bound of $-patchWidth/2$ and an upper bound of $patchWidth/2$.

2.2

...

2.3

`briefLite` is simply a wrapper of previously implemented functions. It first loads the `testPattern` matrix, calls `DoGdetector` which we implemented in part A and then calls `computeBrief` to compute `locs` and `desc`.

2.4

First we'll match couple of the chicken broth pictures:

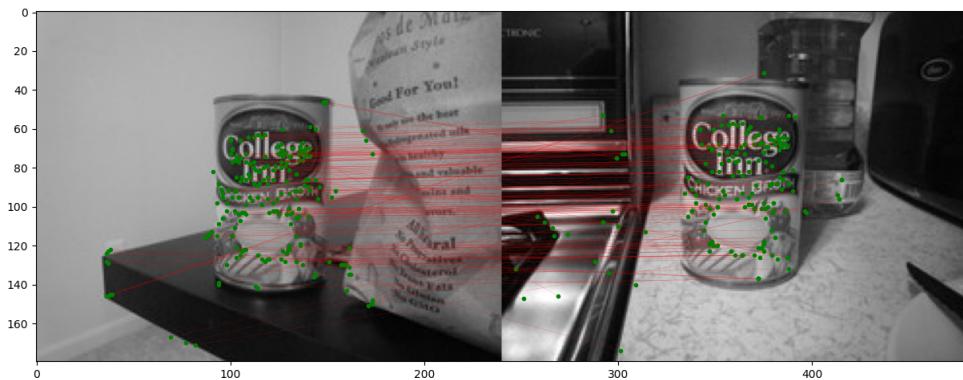


Figure 10: `chickenbroth_03.jpg` and `chickenbroth_04.jpg` matching

Since both cans are facing almost the same direction, we get pretty good matching. We see most of the text matches as well as the edge of the cup in the middle.

Moving forward to the next couple:

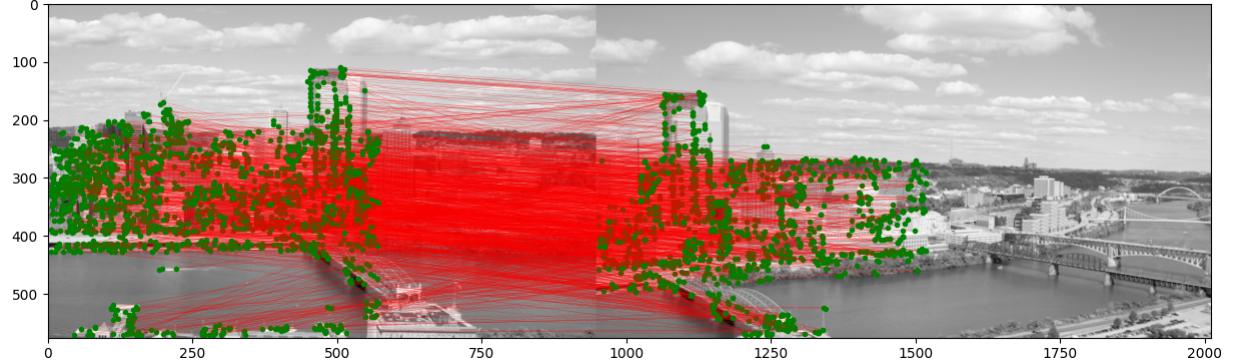


Figure 11: *incline_L.jpg* and *incline_R.jpg* matching

We can see most of the bridge keypoints are matching to one another and also good matchin of the high tower. However, we do see that we get matches for different buildings that are to the right of the tower in one picture and are to the left in ther other.

We did 3 test cases for the Computer Vision book in varying degrees of difficulty:

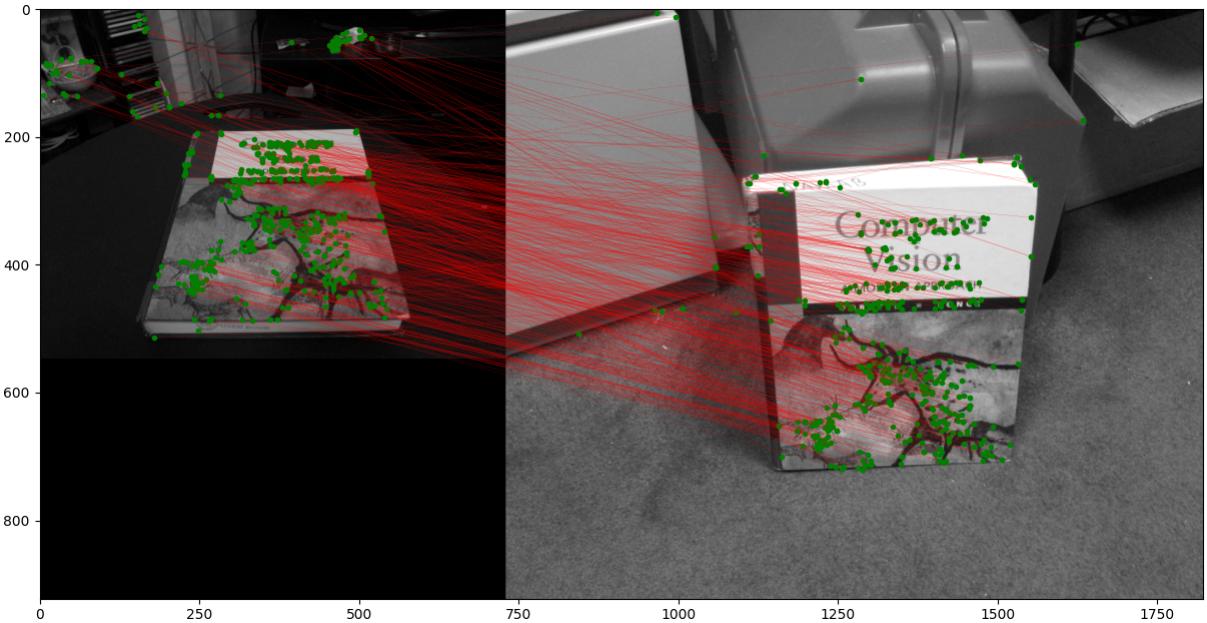


Figure 12: *pf_desk.jpg* and *pf_stand.jpg* matching

The simplest match is between the book laying on the desk and the book standing. Both images are in the same direction even though there is a slight difference in their angle towards us. We can see that indeed we have good matching of both the text and the picture on the cover.

Trying for a harder case, we tested the same desk picture against the picture containing all 3 books:

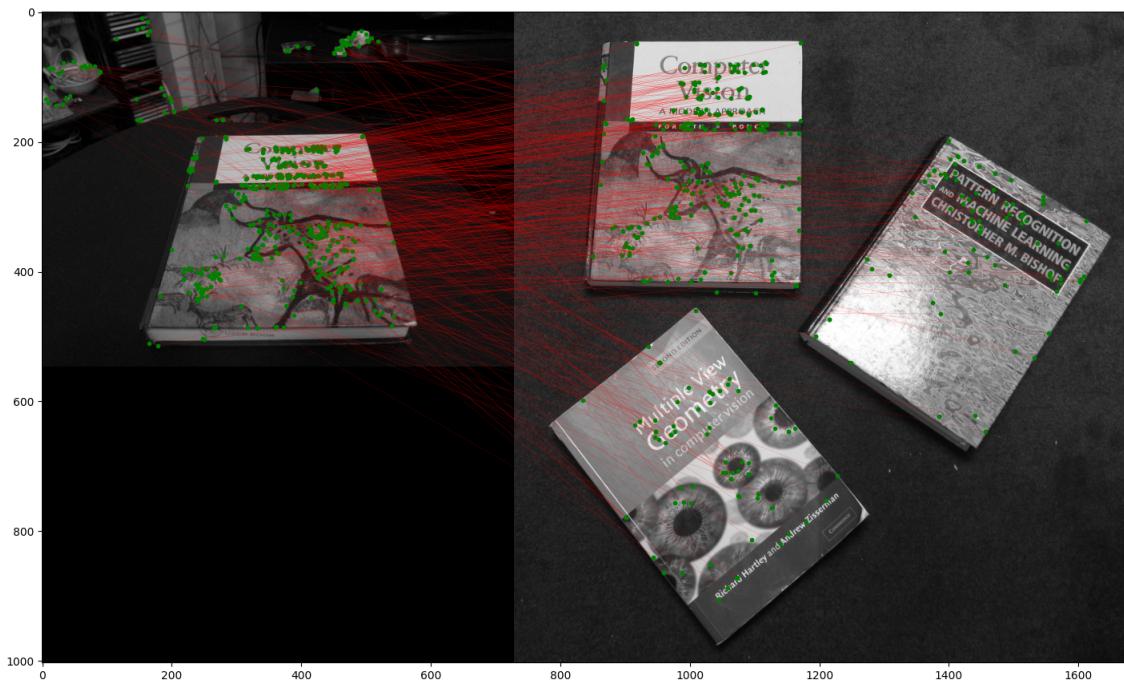


Figure 13: *pf_desk.jpg* and *pf_floor.jpg* matching

In this test we see that while we do have a few matches for the two books which don't appear in the desk image, most of the matches are between the Computer Vision books and once again they match quite well the text and the cover picture.

Our last test was of the same picture rotated:

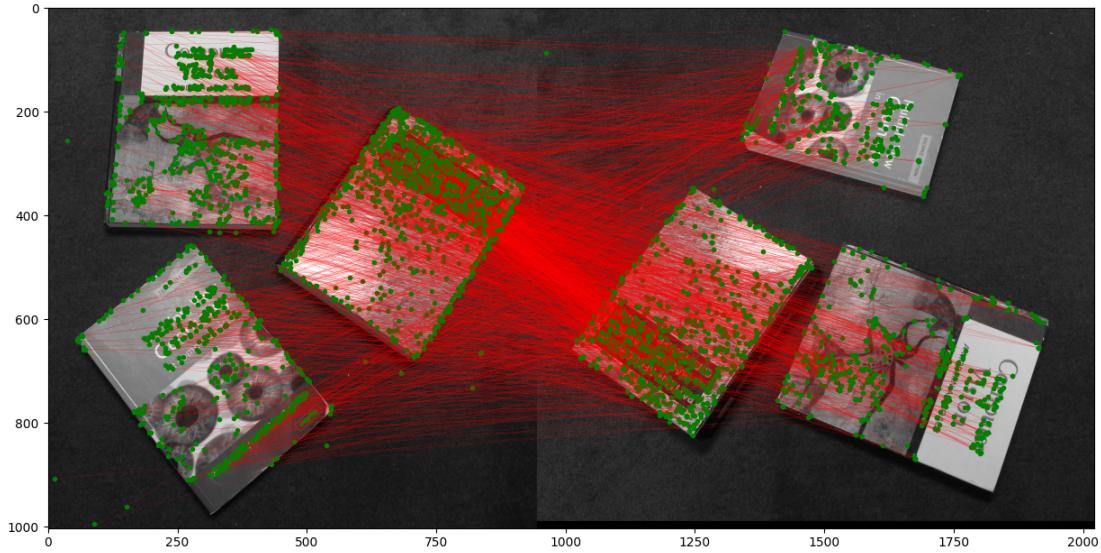


Figure 14: *pf_floor.jpg* and *pf_floor_rot.jpg* matching

We can see that even with a 90° rotation, we match most keypoints.

Judging by the five test cases above, we believe that we get good matches for the books and the chicken broth images because they have far less descriptors than the incline images. The incline image is far more crowded and thus resulting in higher number of descriptors which in turn gets matched to close but different descriptors.

2.5 Bonus

Using 10° rotation, we have received the following...