Computer Vision **046746**
HW 01

April 21, 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 $1 \times \text{len}\,(levels)$ where each element is of size $imH \times imW$.

1.3

The output pyramid is as follows:

Figure 3: Difference of Gaussian pyramid

Where each element is received by simply substracting 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 $1 \times (\text{len}\,(levels) - 1)$ where each element is of size $imH \times imW$.

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:

```
mask_c = np.abs(im) > th_contrast
mask_r = np.abs(PrincipalCurvature[level]) < th_r
mask_unified = mask_c & mask_r
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:

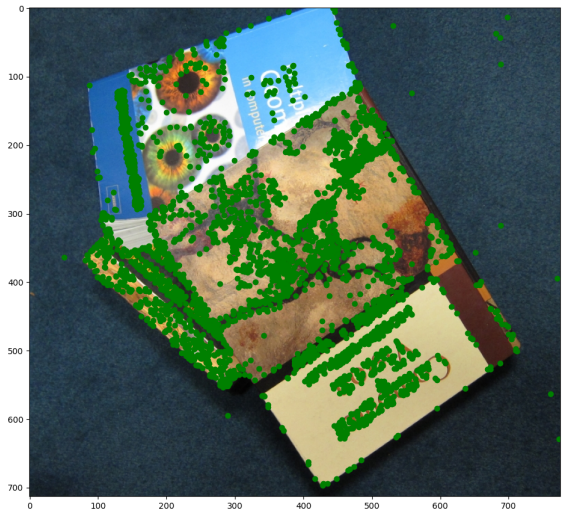

Figure 5: Keypoints for input image *pf_pile.jpg*

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

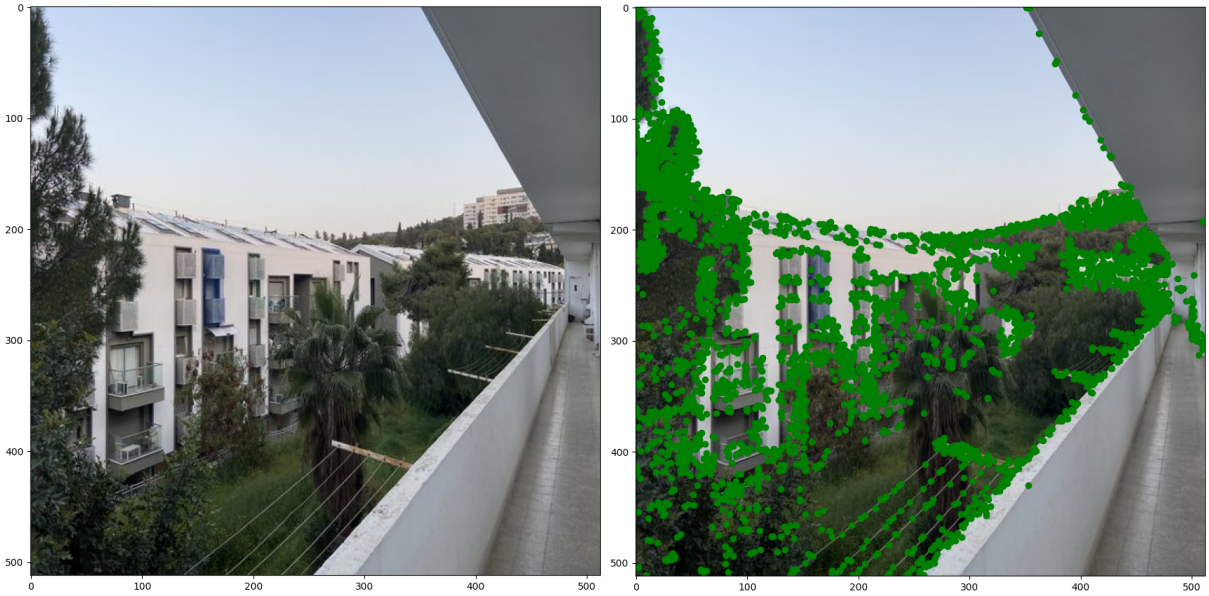

Figure 6: Our image and it's keypoints, side by side

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:

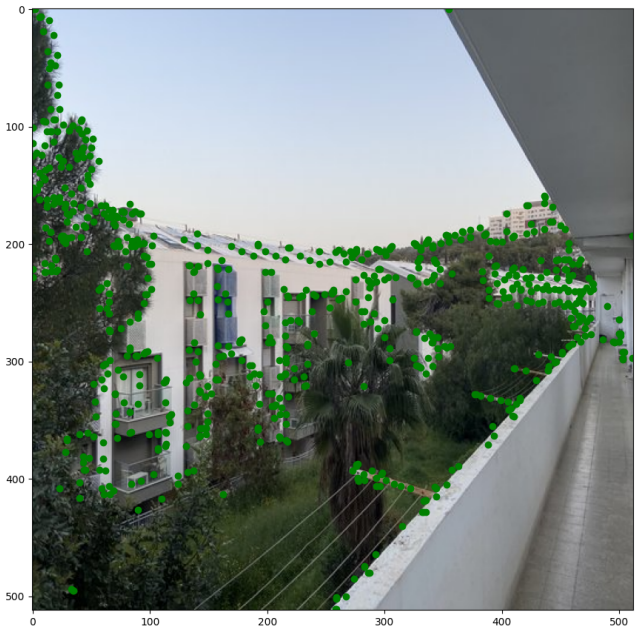

Figure 7: Our image and it's keypoints after applying Gaussian blur

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

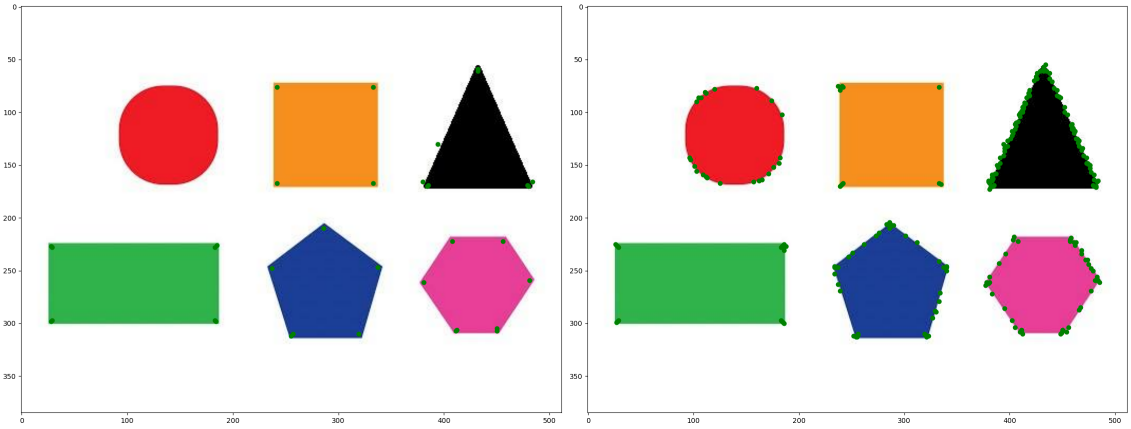

Figure 8: Keypoints for the blurred picture compared to the non-blurred picture

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:

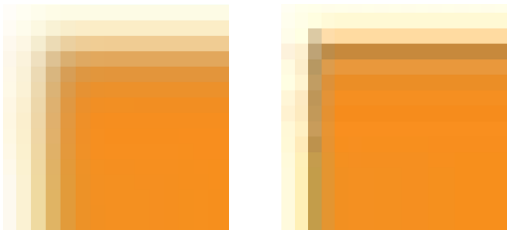

Figure 9: Blurred corner compared to the original corner

2 BRIEF descriptor

Summary

Following the paper, we have fabricated a uniform pattern for comparison and saved it in a mat file. Using the same pattern for each 2 images we have compared the pixel pairs for each image upon each image and created a binary string with a given length each represents if the statement is valid for each pair of pixels.

Then we have used a given function in order to determine the length between strings for each image and extracted from those the common corners found in each image.

Lastly we tested the descriptor performance over rotated images and we have found a weakness finding common corners. This is mainly due to the pattern we have generated being A-symmetrical.

2.1

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

2.2

We simply iterated over the pattern we have fabricated earlier, compared each of the pixels pairs and assembled a 1's and 0's string for each comparison for each corner.

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:

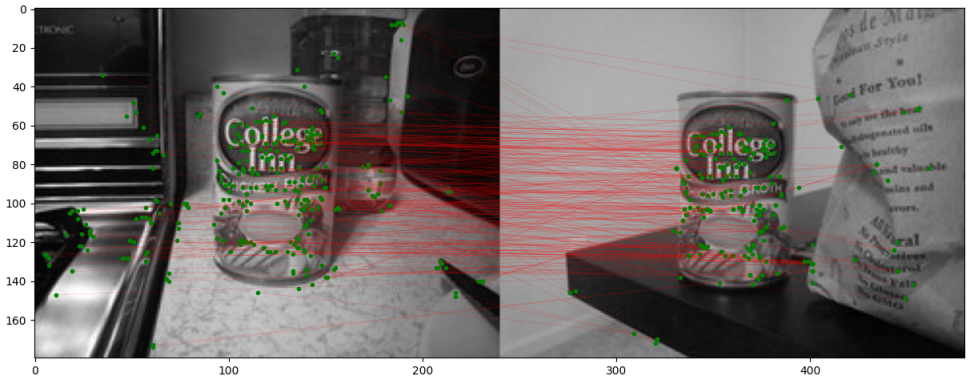

Figure 10: $chickenbroth\_03.jpg$ and $chickenbroth\_04.jpg$ matching

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:

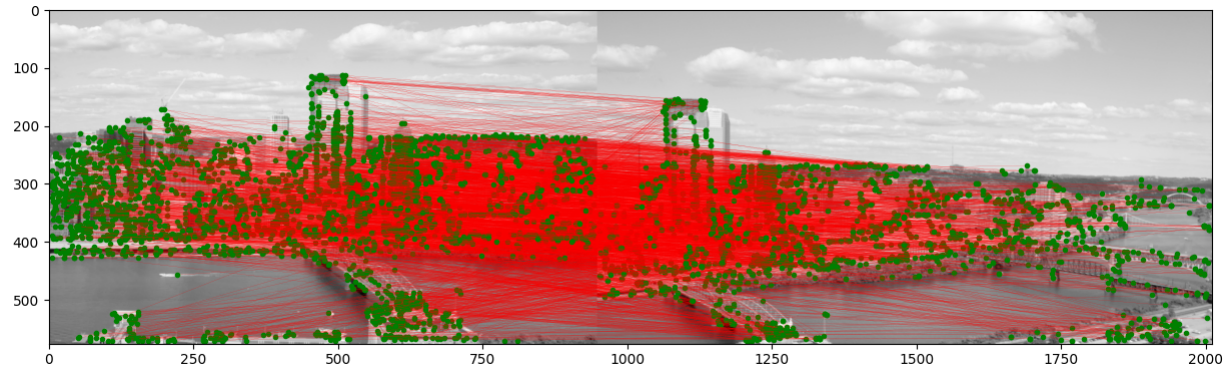

Figure 11: $incline\_L.jpg$ and $incline\_R.jpg$ matching

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.

For matching the Computer Vision book, we tested $pf\_scan\_scaled.jpg$ versus all given images and then did a second test to match how multiple objects match using $pf\_floor.jpg$ and $pf\_floor\_rot.jpg$.

The best results are for $pf\_desk.jpg$ and $pf\_stand.jpg$ since they both show the book facing in the same direction, without any rotation in the XY plane (although there is some angle toward the viewer)

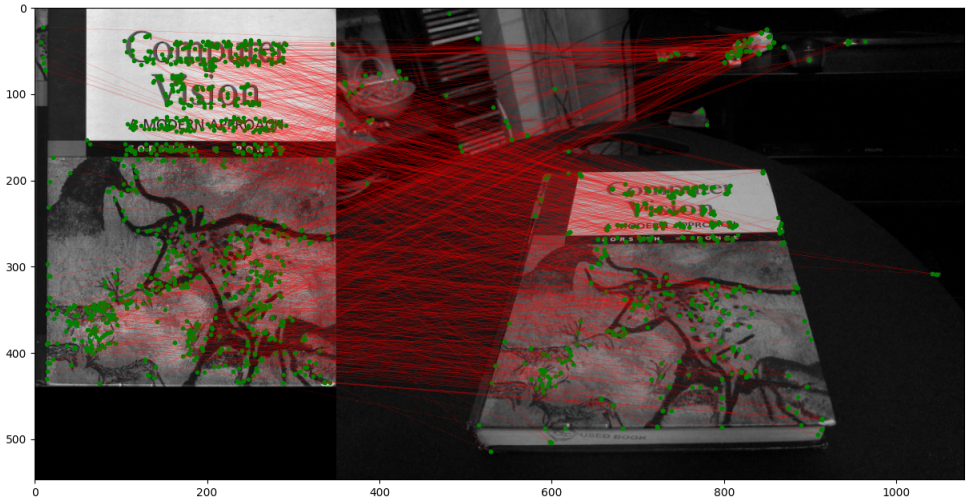

Figure 12: $pf\_scan\_scaled.jpg$ and $pf\_desk.jpg$ matching

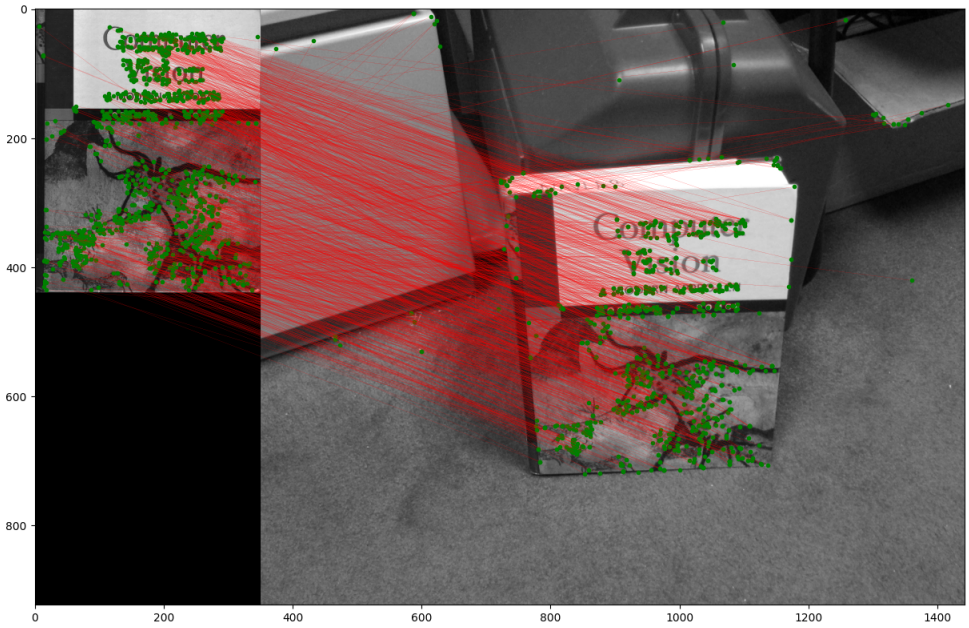

Figure 13: $pf\_scan\_scaled.jpg$ and $pf\_stand.jpg$ matching

We can see that both cases match a good amount of features both in text and in the cover picture with a relatively small number of false matches. Moving to a match with multiple objects, we see some false matches:

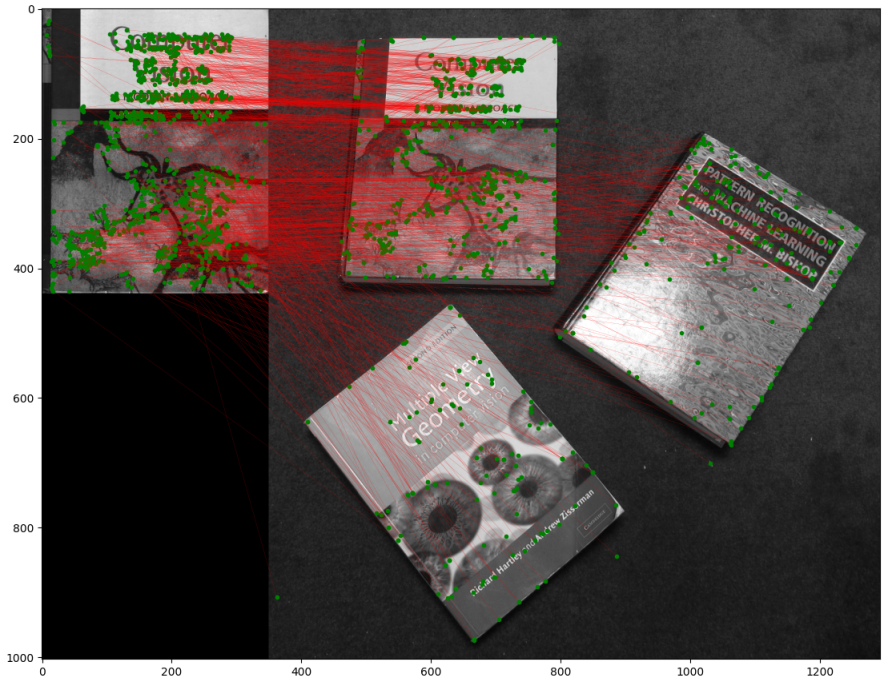

Figure 14: $pf\_scan\_scaled.jpg$ and $pf\_floor.jpg$ matching

We can see that although the Computer Vision book is aligned the same way in both images, we get some false matches to other book covers, but not much more than we had seen in the chicken broth case, for example.

However, once we introduce rotation, we get far less appropriate matches:

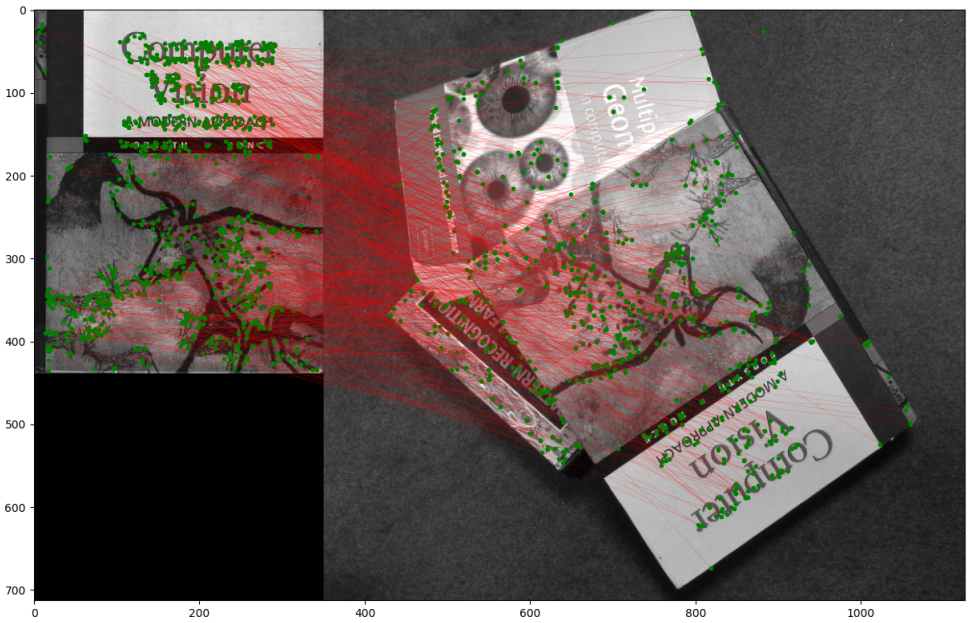

Figure 15: $pf\_scan\_scaled.jpg$ and $pf\_pile.jpg$ matching

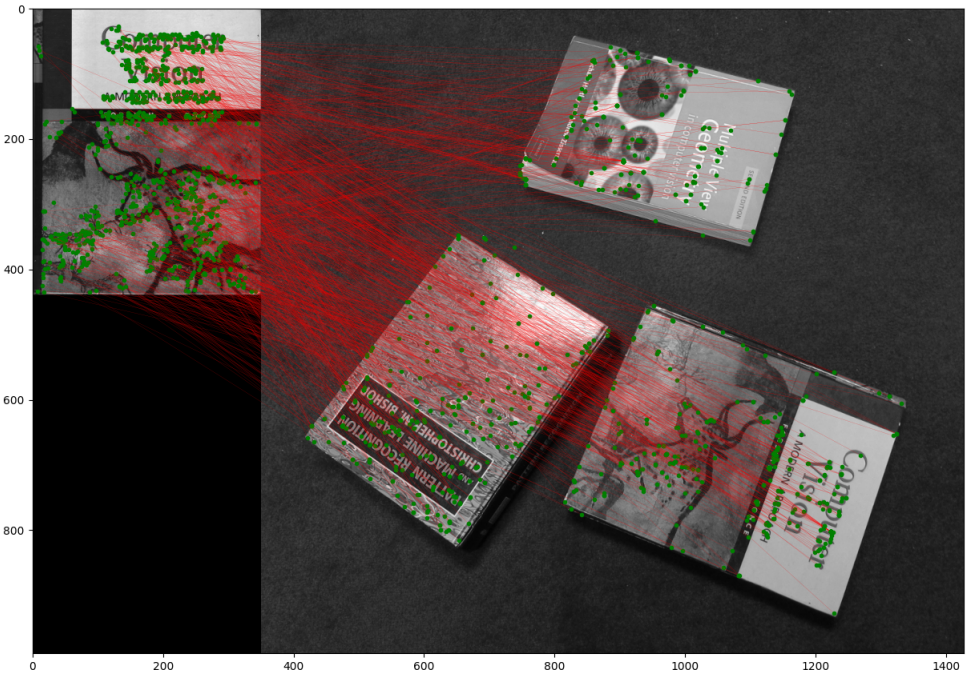

Figure 16: $pf\_scan\_scaled.jpg$ and $pf\_floor\_rot.jpg$ matching

In both cases, not only that the book we try to match is rotated, but we see similar features in two other books which cause even more false matching. This shows that our implementation is sensitive to rotations.

Our last test was of the same picture rotated:

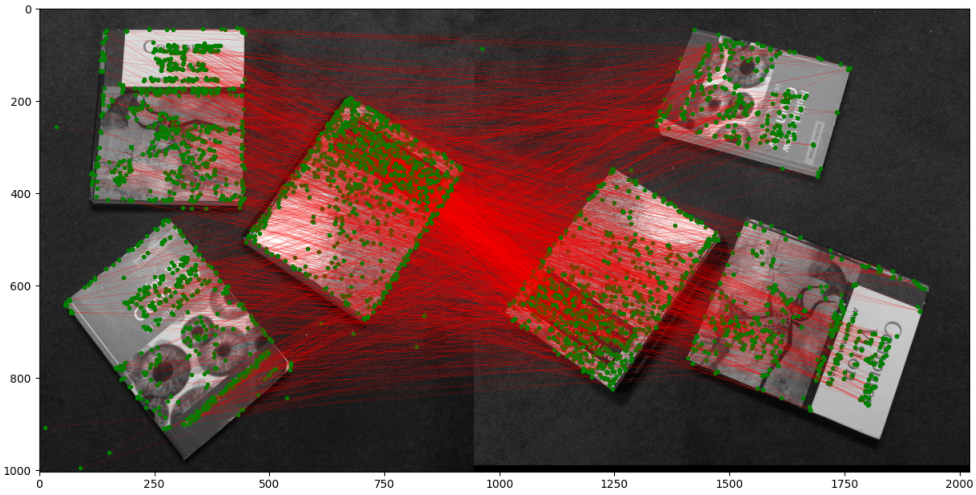

Figure 17: $pf\_floor.jpg$ and $pf\_floor\_rot.jpg$ matching

We can see that even with a $90°$ rotation, we match most keypoints. This is due to the fact that although the picture is rotated and the matching did suffer some false matching, we have far more features that actually exists in both pictures and thus we get better results than matching the Computer Vision book to an image including two additional books.

2.5 Bonus

We have created a bar graph matching $model\_chickenbroth.jpg$ against itself in increments of 10 degrees up to 90 degrees. The results are as follows:

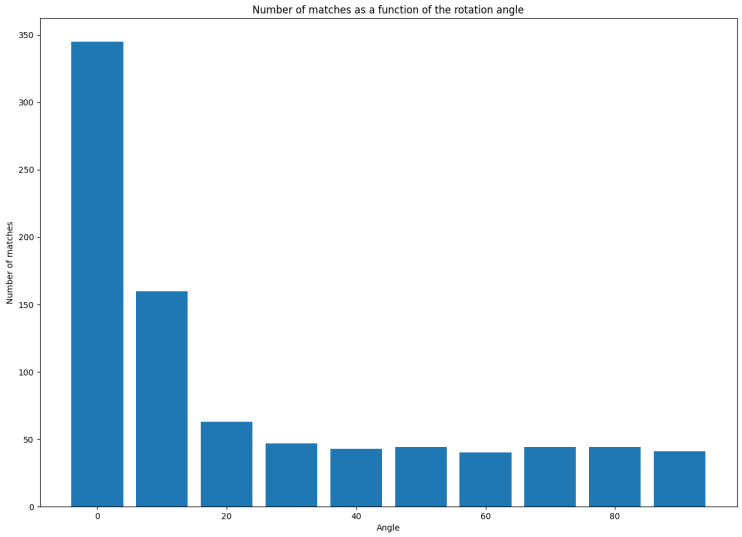

Figure 18: Number of matches as a function of the rotation angle

The graph shows that even a 10 degree rotation drops the number of matches by more than a half, and for a

change of 20 degrees the matching falls down to around 15%, a level in which it stays for all degrees up to 90.

We believe this is the case because of the way we match features using the same coordinates and thus makes it prone to errors when using real images which are non-symmetrical.

If we use a picture which is more symmetrical, such as the polygons we showed at **Q1**, we get much better matching for rotations:

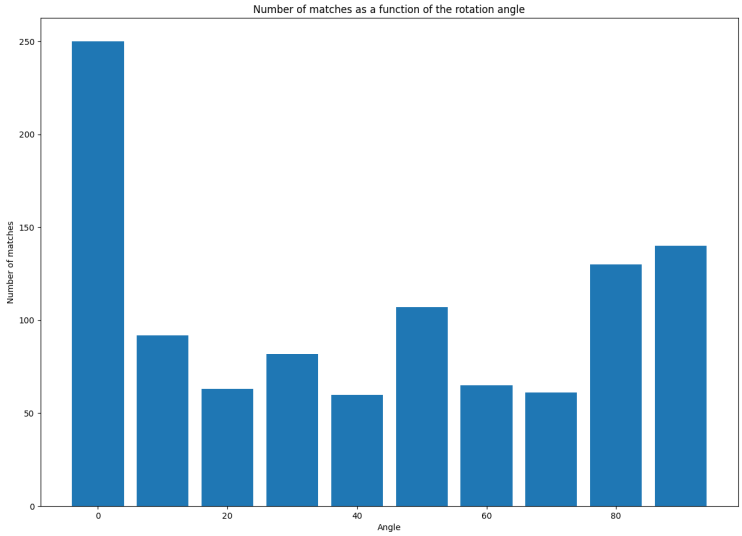

Figure **19:** Number of matches as a function of the rotation angle using $polygons.jpg$