Daniel Illner
888013562

19.1.2020

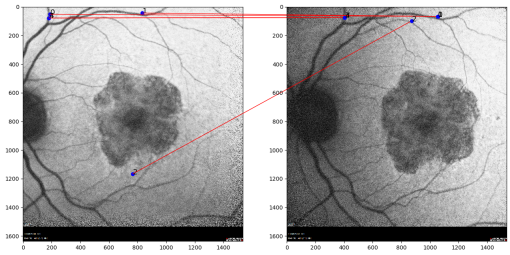
Medical Image Processing - Solution Homework 05

Automatic Registration

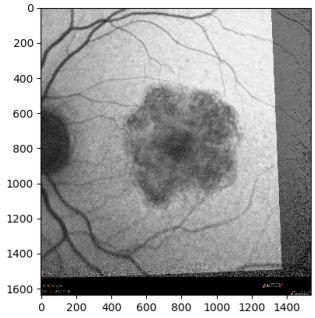
Part 1 - Registration by features

a) algorithm 1

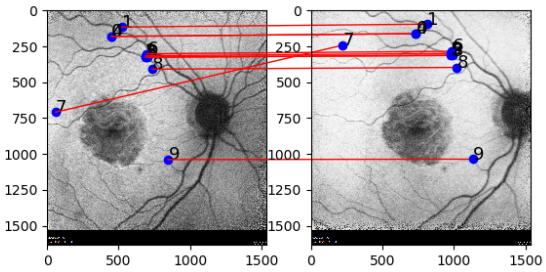
Implemented *def firstalgorithm()* after the steps given in the task sheet. The results for image 1, 3 and 4 are pretty good (worked perfectly). In the other pictures the algorithm has problems finding distinct keypoints and as follows correct matches. The results are:



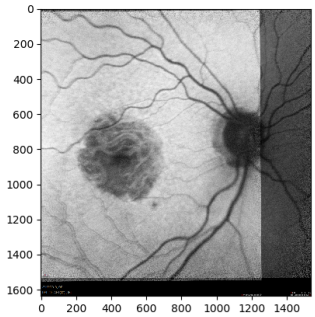
(a) Image 1 - Found matches - ORB - 5 matches



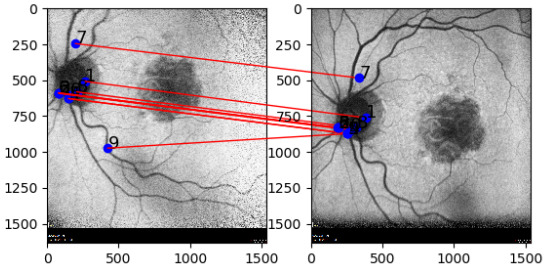
(b) Image 1 - resulting transformation



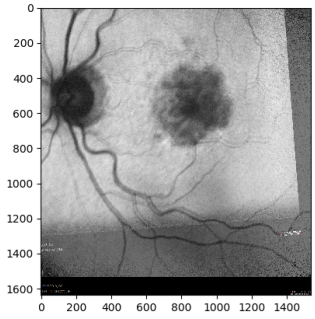
(a) Image 3 - Found matches - ORB - 10 matches



(b) Image 3 - resulting transformation



(a) Image 4 - Found matches - ORB - 10 matches



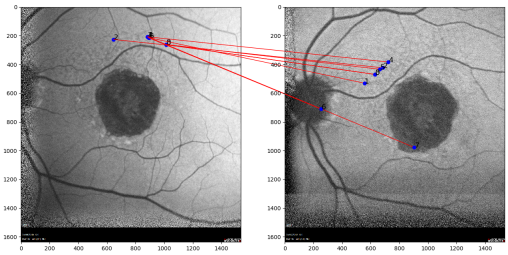
(b) Image 4 - resulting transformation

As I mentioned for the other 2 pictures the matching keypoints process was pretty tough and good matches were barely found. As you can see, so the algorithm produced not a proper result. Also the *RANSAC* function of the *utily.py* sometimes produced a rigid registration matrix all 0 (especially with a lot of found matches put into the function) and as I did not write that function, I am not totally sure why this happens.

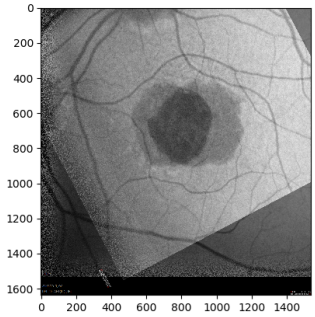
What is also worth mentioning, that you have to play with the number of matches you want to keep, before using RANSAC. The matches with the lowest distance measure are kept. You find the number of found matches I used under each figure. For example image 1, did not produce proper results by keeping 10 matches but did with 5.

Maybe in order to let the algorithm decide if the result is sufficient, you have to implement some measurement (e.g. the rmse would be a good start), to let it rerun with more or less "best matches" kept. And evaluate new afterwards.

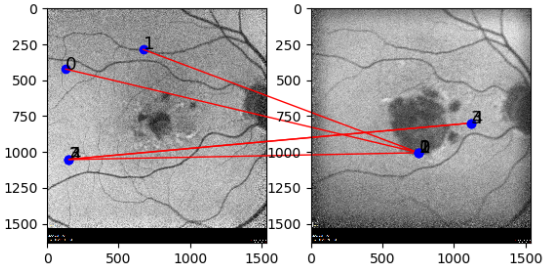
Failed results are:



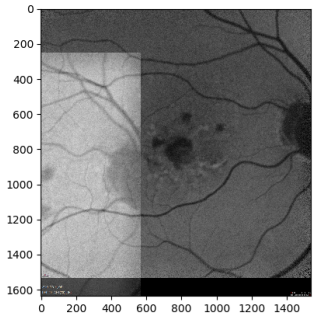
(a) Image 2 - Found matches - ORB - 8 matches



(b) Image 2 - resulting transformation



(a) Image 5 - Found matches - ORB - 5 matches

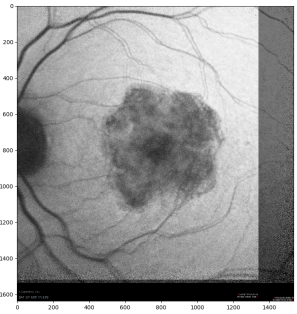


(b) Image 5 - resulting transformation

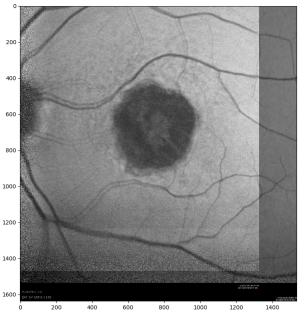
For evaluation check out the code. You can also play around with different algorithms for finding keypoints (SIFT, SURF) in the function additionally. I also implemented the descriptor matching function to adapte it for my purpose.

b) algorithm 2

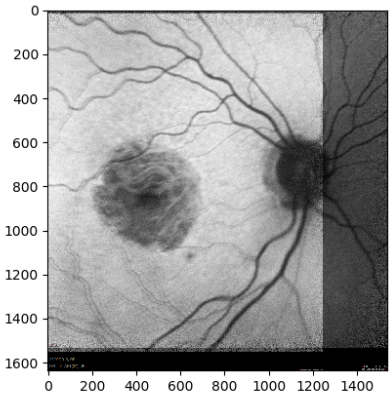
As I did not fully understand the algorithm, you gave me. As I understood it, it is some kind of template matching, so I kind of did a template matching in two steps. Nevertheless, my code sticks almost to the steps you gave me. At first I rotated the original BL image in small angle steps and find the best angle with the maximum correlation result and kept that angle. I rotated the binary image with the angle and did a second step to estimate the translation (used the function *feature.register_translation* from *skimage* for this purpose). In the end I merged the two results to register the image. The results are:



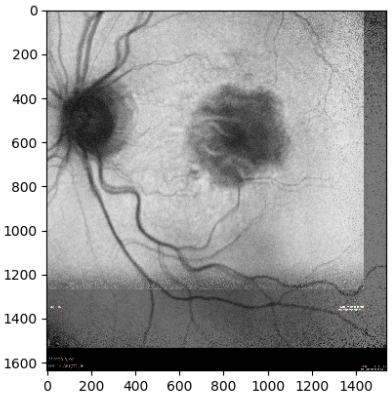
(a) Image 1 - resulting transformation



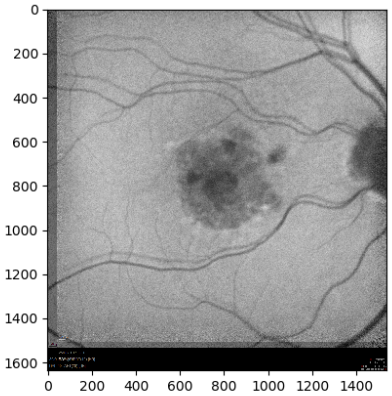
(b) Image 2 - resulting transformation



(c) Image 3 - resulting transformation



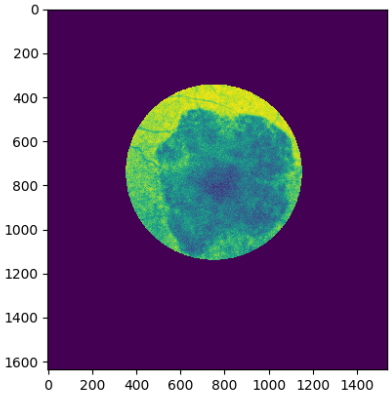
(d) Image 4 - resulting transformation



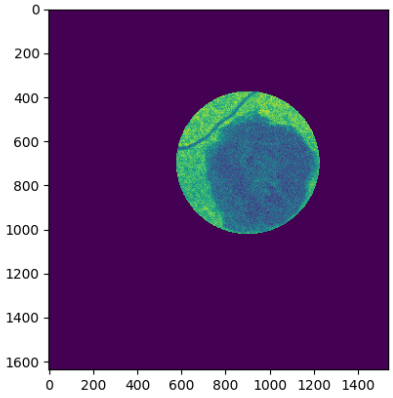
(e) Image 5 - resulting transformation

Part 2 - Detect the changes

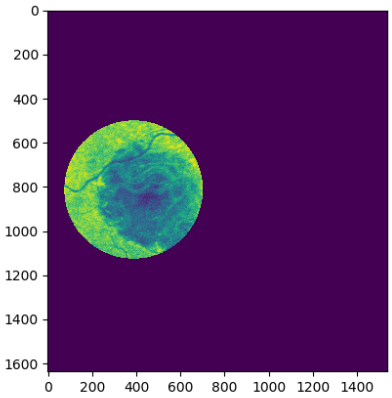
As the original algorithm in the tasksheet did not really work for me, I considered changing it a bit. The main step is to create a ROI by segmenting the original BL or FU image in order to find the lesion and create a circle ROI around it. This is done in the function *segmentLesion()*. For more detail read the code description. There is also a plotting option in it. Finding the ROI worked quite good, but not all the time perfectly. Making this function more robust, helps a lot. It is not done due to time issues. Important is also that I choose BL or FU to find the ROI on each image indepently. This could also be automated, e.g. by comparing the images and choose the image with the bigger lesion (this should give you the better ROI). Results are:



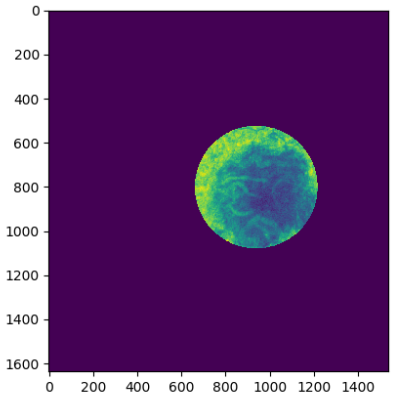
(a) Image 1 - ROI - BL



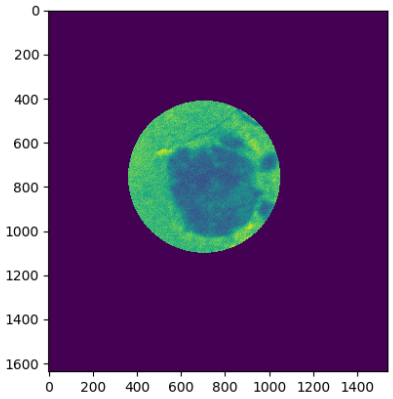
(b) Image 2 - ROI - FU



(c) Image 3 - ROI - BL

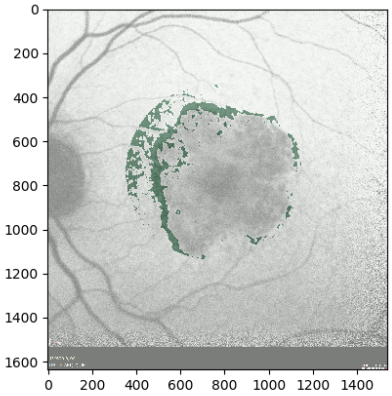


(d) Image 4 - ROI - FU

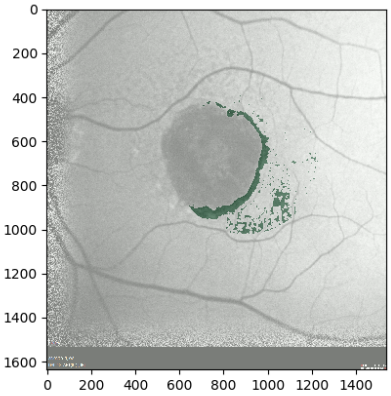


(e) Image 5 - ROI - FU

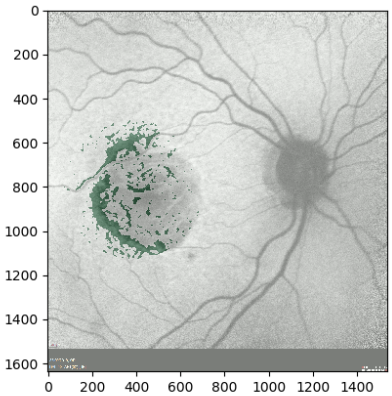
The final results are computed in the function *detectingChange()*. The algorithm is highly oriented to the one in the tasksheet. Please check the steps in the code. At the end of segmentation the ROI is applied. Please note, that the algorithm could be improved by putting more effort and time in it, but in the end I had to decide myshelf by going this way to hand in the code. I think the results are suffiecent but still there is some room to improve. Please note, that I precomputed the tranformed images and just called it from data. Combining task 1 and 2 is also possible if required. This gives the final results:



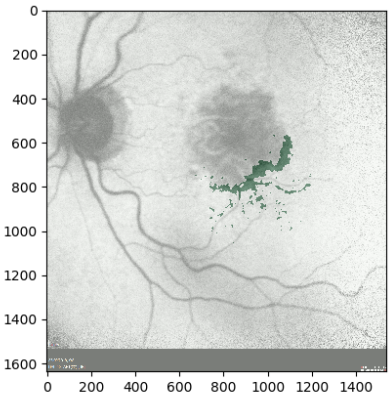
(a) Image 1 - final changes



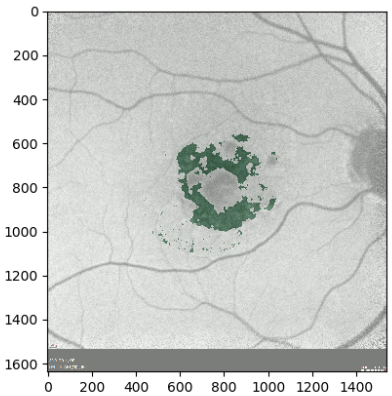
(b) Image 2 - final changes



(c) Image 3 - final changes



(d) Image 4 - final changes



(e) Image 5 - final changes