# **Assignment 3**

# **Edge Detection, Line & Circle Detection, Matching Using Local Features & Template Matching**

| Total Mark: | 12 marks (6% of the total course grade)   * 9 out of 12: Learn@Seneca Submission (Due: Monday June 30 at 8:00am) * 3 out of 12: Assignment Demo (During the Lab of Week 8) |
| --- | --- |
| Submission file(s): | * Assignment3\_1.py / Assignment3\_1.ipynb * Assignment3\_2.py / Assignment3\_2.ipynb * Assignment3\_3.py / Assignment3\_3.ipynb * Assignment3\_4.py / Assignment3\_4.ipynb * Assignment3.docx (this document with your answers) |

Please work **within your group** to complete this assignment.

This assignment is worth 6% of the total course grade and will be evaluated through your written submission, as well as the assignment demo.

During the assignment demo, group members are *randomly* selected to explain the submitted solution. Group members who are not present during the assignment demo will lose the demo mark.

Please submit the submission file(s) through Learn@Seneca.

***Please paste the resulting images and answers in this document.***

## **Part I: Edge Detection**

Create a program (save as Assignment3\_1). Include code to:

1. Open ‘Credit’ (located in the ‘Assignment3\_Files’ folder) and convert it to greyscale. Paste the result here.

A credit card with a planet in the background

AI-generated content may be incorrect.

1. Use OpenCV to find edge maps on the image. Use a Sobel operator. Display the results for the following parameters as well as two output datatypes, i.e., CV\_8U and CV\_64F. Paste results here.

* dx =1 , dy = 0
* dx =0 , dy = 1
* dx =1 , dy = 1



What do you notice for datatypes? Explain.

* **CV\_8U (8-bit unsigned integer):**
* **Only positive values are represented.** Any negative gradient is clipped to 0, which causes a loss of edge information.
* **Image appears sharper but may lose details,** especially where gradient changes are negative (dark-to-light transitions may vanish).
* **Edges can look more "binary"** and less detailed, since negative gradients are not visualized.
* **CV\_64F (64-bit floating point):**
* **Both positive and negative values are kept.** This means more edge information is preserved.
* **Image may appear more “detailed”** (if scaled properly), and you can see edges regardless of direction (light-to-dark and dark-to-light).
* **Needs to be converted/normalized for display** (since raw values can be negative or exceed 255).

What do dx and dy show? Explain.

* **dx and dy specify the derivative direction for edge detection:**
* **dx=1, dy=0**: Detects edges in the **horizontal direction** (i.e., vertical edges in the image).
* **dx=0, dy=1**: Detects edges in the **vertical direction** (i.e., horizontal edges in the image).
* **dx=1, dy=1**: Detects edges along the **diagonal direction** (combines both vertical and horizontal changes).
* **How it works:**
* **Sobel** computes the gradient (rate of intensity change) in the specified direction(s).
* High gradient values correspond to edges.

1. Compare the results of Sobel operator with the Canny edge detector. Experiment with different low and high threshold values and display two samples here. How is threshold changing the results?

A credit card with a chip and a credit card

AI-generated content may be incorrect.

* The left Canny map (Low=50, High=150) is more detailed, showing almost every line, even faint ones.
* The right Canny map (Low=100, High=200) is less detailed, focusing on the most prominent edges.

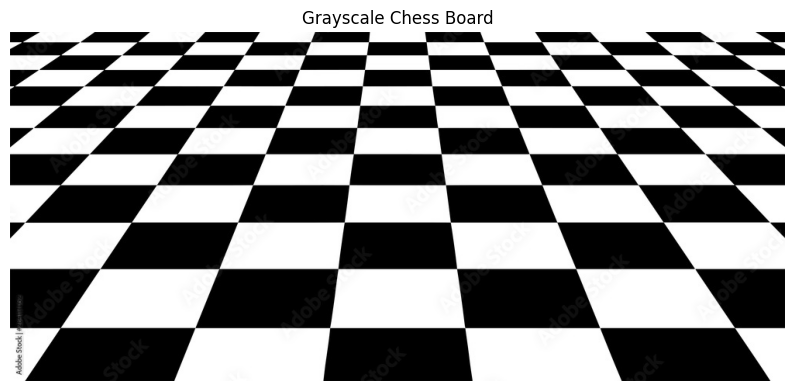
1. Explain how is the Canny map different from the Sobel map?

* **Sobel:** Simple, detects gradient (change) in a direction; good for showing where image changes rapidly but often result in thicker, less precise edges and more noise.
* **Canny:** More sophisticated, produces ***thinner, cleaner, and more connected edges*.** The two thresholds control which edges are included: only those above the high threshold, and those connected to them above the low threshold.

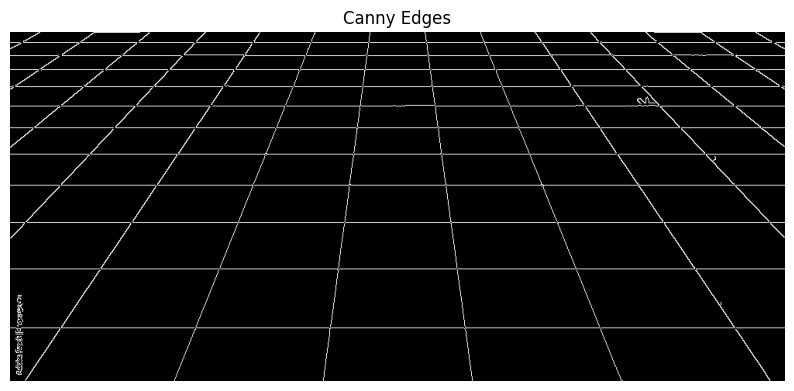
## **Part II: Line & Circle Detection**

Create a program (save as Assignment3\_2). Include code to:

1. Open ‘Chess\_Board’ (located in the ‘Assignment3\_Files’ folder) and convert it to grayscale. Paste the result here.



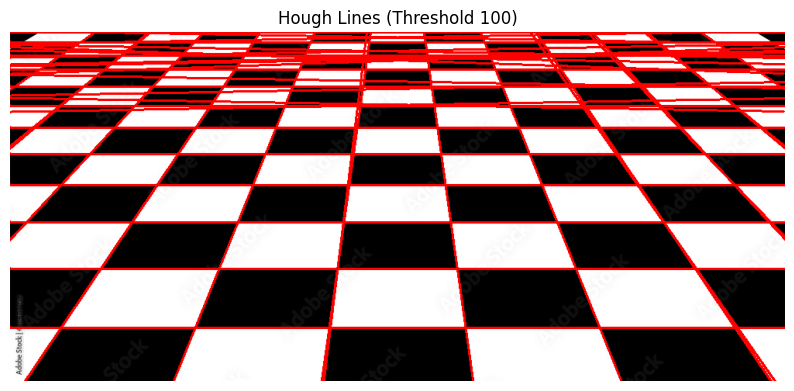
1. Use Canny edge detector to detect edges. Paste the result here. What hyperparameters did you use for Canny?

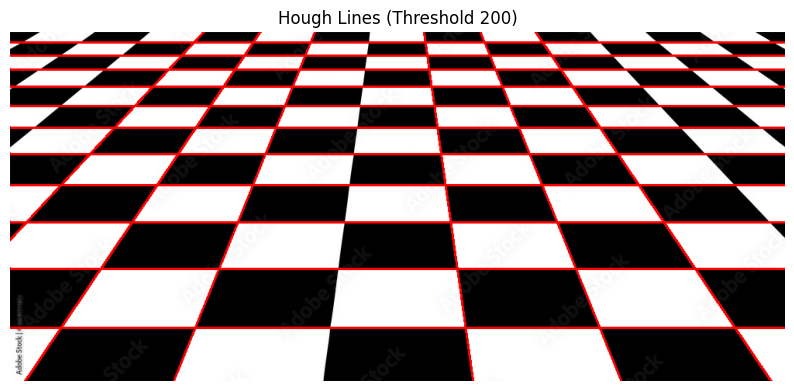


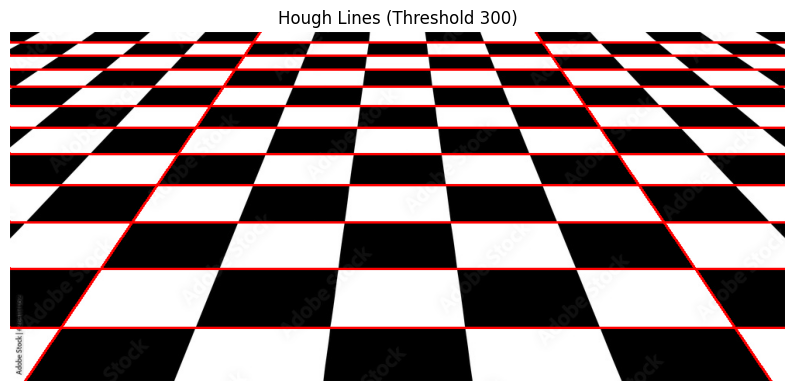
**Hyperparameters Used for Canny Edge Detector:**

* **threshold1:** 50
* **threshold2:** 150

1. Use either the standard ~~or probabilistic~~ Hough transform ([OpenCV: Hough Line Transform](https://docs.opencv.org/4.11.0/d9/db0/tutorial_hough_lines.html)) to detect lines in the above image. Implement a loop to increment the threshold and visualize the lines on the image for various threshold values. Paste the resulting images with lines for three different threshold values.





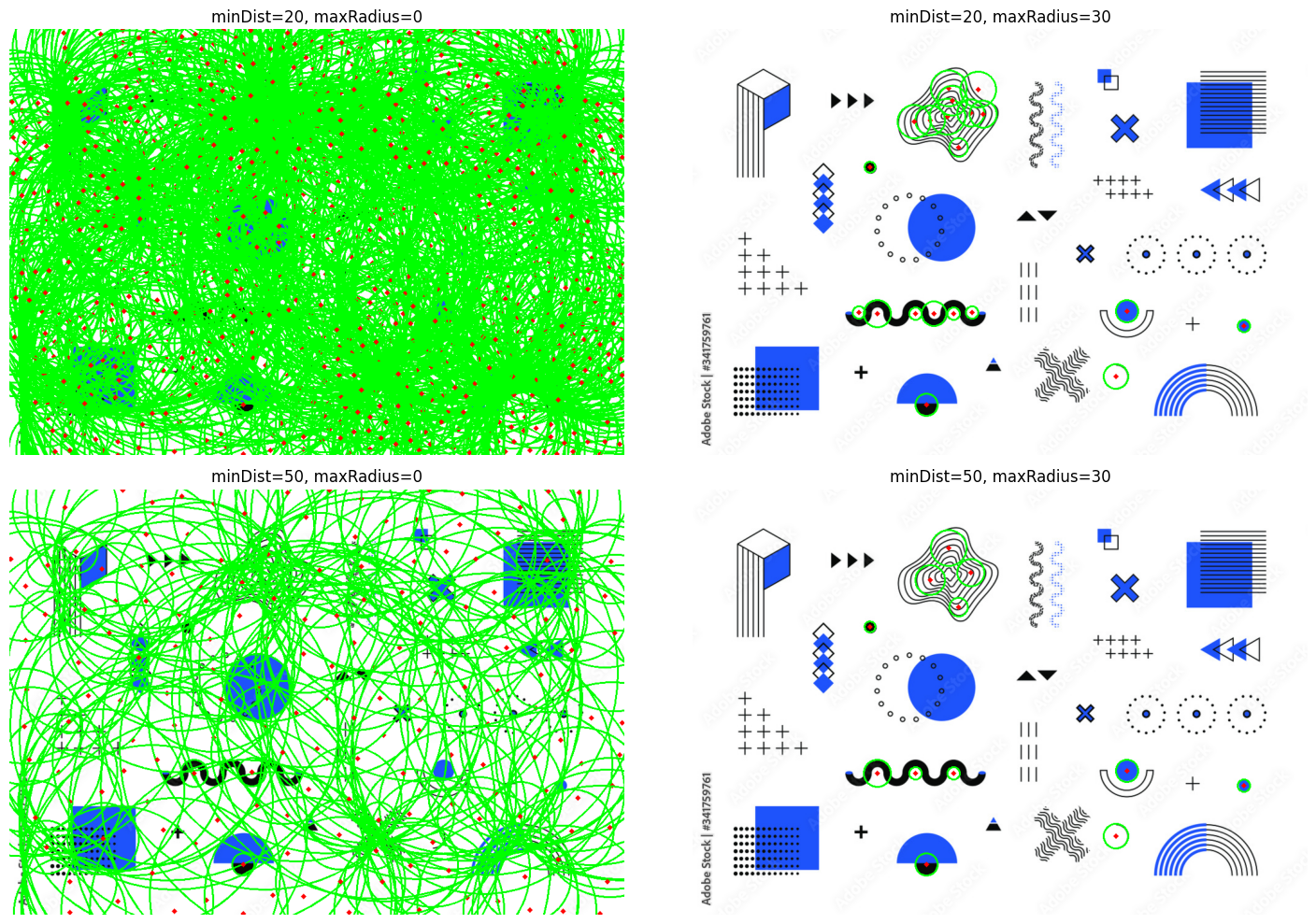


1. Explain what threshold does.

The **threshold** in cv2.HoughLines determines **how many points (votes) in Hough space are required to consider something a line**.

* **Lower threshold** → more lines are detected (including weak or noisy lines).
* **Higher threshold** → only strong, prominent lines (with lots of "votes") are detected.

1. Use Hough transform ([OpenCV: Hough Circle Transform](https://docs.opencv.org/4.11.0/da/d53/tutorial_py_houghcircles.html)) to detect circles in the ‘Shapes’ (located in the ‘Assignment3\_Files’ folder). Implement a loop to visualize the circles detected on the image for various values of minDist and maxRadius. Paste at least four of the resulting images with circles.



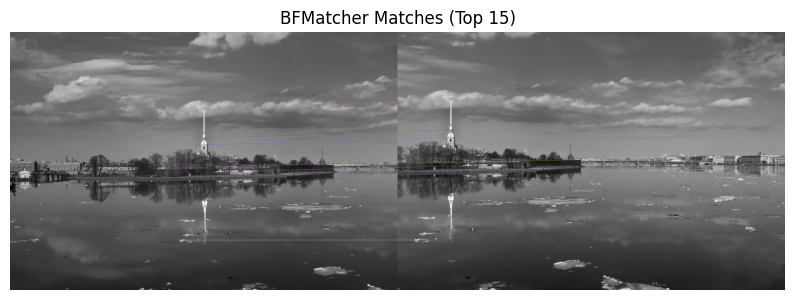
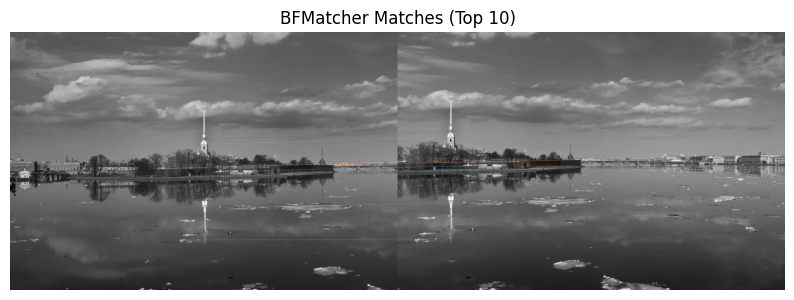
1. In your words, explain how would parameters in cv.HoughCircles() change the result? Choose 3 parameters for this part.

* **minDist**: The Minimum Distance between the centers of the Detected Circles
* **minRadius**: The Minimum Radius of Circles to Look for.
* **maxRadius**: The Maximum Radius of Circles to Look for.

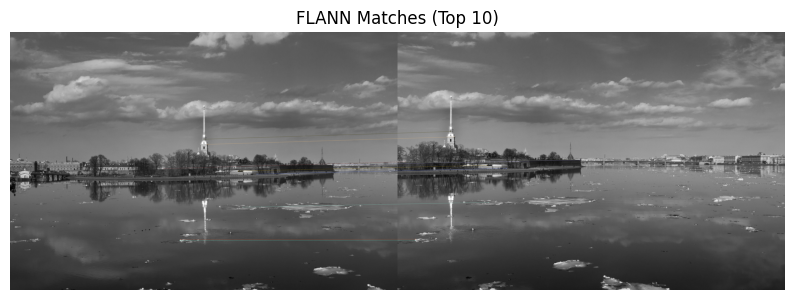
## **Part III: Matching Using Local Features**

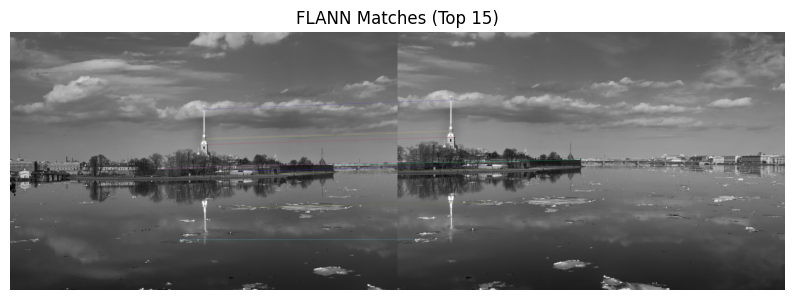
Create a program (save as Assignment3\_3). Include code to:

1. Open ‘boat2’ and ‘boat3’ (located in the ‘Assignment3\_Files’ folder).
2. Use ORB to detect and describe local features for the above images. Match features for the top 10 and the top 15 matches, using:
   1. BFMatcher



* 1. FlannBasedMatcher



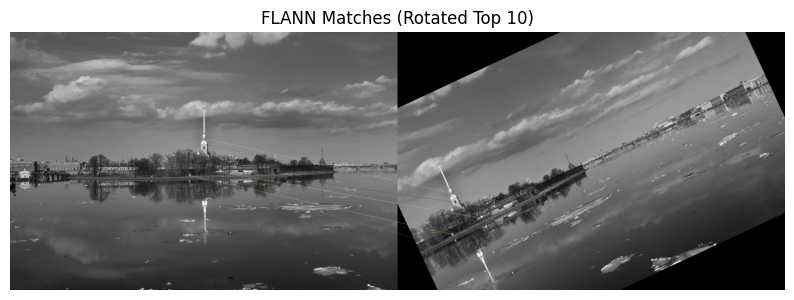


*Paste results here*.

1. How are keypoints matched? What is the difference between these two matchers? Which one performs better, and why?

* Keypoints are matched by comparing their feature descriptors—unique signatures computed for each keypoint in both images. The matching algorithm finds pairs of keypoints with the most similar descriptors, indicating they represent the same point or object across images.
* The main difference is in how they find matches:
* **BFMatcher (Brute Force Matcher)** checks every descriptor in one image against every descriptor in the other, finding the closest one by direct comparison.
* **FlannBasedMatcher** uses optimized data structures to quickly find approximate nearest neighbors instead of comparing every single pair, making it faster for larger datasets but sometimes slightly less precise.
* **BFMatcher** performs better for small or moderate feature sets because it always finds the most accurate matches through brute force comparison. However, as the number of features grows, it becomes slow and inefficient. **FlannBasedMatcher** performs better on large datasets because it uses fast, approximate algorithms to quickly find good matches, making it much faster and more scalable—even if it occasionally sacrifices a bit of matching accuracy. So, for big images or real-world applications, FLANN is preferred due to its speed.

1. Rotate ‘boat3.jpg’ by 25 degrees counterclockwise around its center. Repeat feature matching between the rotated image and ‘boat2.jpg’ selecting the top 10 matches. Show the matches using FlannBasedMatcher.



1. How did the rotation affect the matching? Explain.

* Rotation can reduce the number of correct matches and lower the overall quality of matching, especially if the feature detector is not fully rotation-invariant. In your results, you’ll notice that some lines still connect correct points, but others may not, and overall, there are fewer strong matches compared to unrotated images.

1. (Optional) Use SIFT descriptors to match features between the images. Utilize a FlannBasedMatcher and apply Ratio Matching to filter the matches. Compute the homography matrix using the matched keypoints (a minimum of four corresponding point pairs is required between the two images). Warp one image to align with the other using the computed homography matrix.

You can further attempt to blend and stitch the boat images from the Assignment3\_Files folder to create a single panorama. Note that OpenCV's cv2.Stitcher class ([OpenCV: Images stitching](https://docs.opencv.org/4.11.0/d1/d46/group__stitching.html)) provides built-in functions for panorama creation. It handles feature detection, homography estimation, image warping, and blending automatically. Once completed, paste the result here.

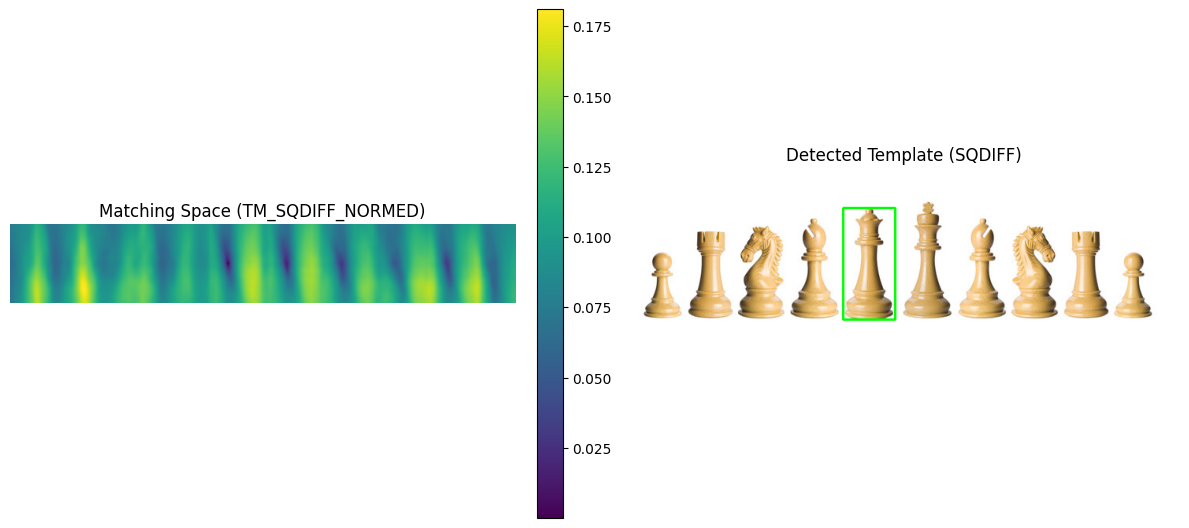
[Images obtained from [opencv\_extra/testdata at 4.x · opencv/opencv\_extra · GitHub](https://github.com/opencv/opencv_extra/tree/4.x/testdata)]



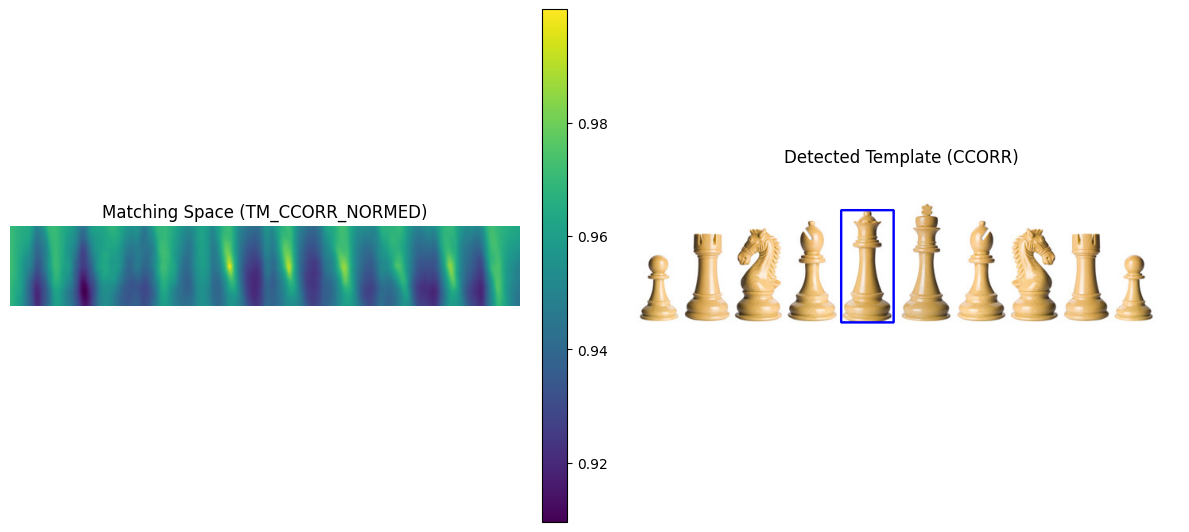
## **Part IV: Template Matching**

Create a program (save as Assignment3\_4). Include code to:

1. Open ‘Chess\_Pieces’ as query image, and ‘Queen’ as template image.
2. Use matchTemplate and TM\_SQDIFF\_NORMED measure to find the best match location. Draw a rectangle around the matching area in the query image. Additionally, display the matching space. Paste the results here.



1. Repeat matching, using TM\_CCORR\_NORMED measure this time. Similarly, paste the results here.



1. How are the two results and the matching spaces different? Explain.

* The two matching spaces differ in how they indicate a good match: **TM\_SQDIFF\_NORMED** shows the best match as the darkest (lowest) spot, since lower values mean higher similarity, while **TM\_CCORR\_NORMED** shows the best match as the brightest (highest) spot, because higher values mean better similarity. So, with SQDIFF you look for the minimum, and with CCORR you look for the maximum.

## **Part V: Project Proposal Submission**

## Please submit your project proposal at the beginning of Week 8, following the study week. You will have the opportunity to discuss and refine your proposed topics during the labs in Weeks 6, 7, and 8.

The goal of this project is to create a computer vision-based Sudoku solver that extracts a Sudoku puzzle from an image, identifies existing digits using OCR (Optical Character Recognition), and solves the puzzle with an algorithm. The user submits a photo of a Sudoku puzzle, and the system applies the solution onto the original image. This project will utilize OpenCV for image preprocessing, pytesseract for OCR, and a backtracking algorithm for solving the puzzle.

## **Part VI: Group Work**

Add this declaration to your file:

We, Aliyyah Jackhan, Jonathan Chacko, Mohammed Aadil (mention assigned group number and your names), declare that the attached assignment is our own work in accordance with the Seneca Academic Policy. We have not copied any part of this assignment, manually or electronically, from any other source including web sites, unless specified as references. We have not distributed our work to other students.

Specify what each member has done towards the completion of this assignment:

|  |  |  |
| --- | --- | --- |
|  | Name | Task(s) |
| 1 | Aliyyah Jackhan | Completed *Assignment3\_1.ipynb* |
| 2 | Jonathan Chacko | Completed *Assignment3\_2.ipynb* |
| 3 | Mohammed Aadil | Completed *Assignment3\_3.ipynb* and *Assignment3\_4.ipynb* |