Homework Instructions

Release Date: 15.12.2025

Submission Deadline: 30.12.2025 (23:59)

Late submissions will *not* be accepted under any circumstances.

Submission Format

- You may submit the homework **individually** or in **pairs (group of two)**.
- All solutions must be written and executed using a **Jupyter Notebook** (IPython) or **Google Colab**.
- Submit **one notebook file (.ipynb) for each** question containing:
    - The code
    - The output/results
    - Explanations and comments where needed
    - The images you used also in each question

Code Quality Requirements

- Write **clean, readable, and well-structured code**.
- Use **meaningful variable and function names**.
- Add **clear comments** explaining your logic.
- Avoid unnecessary complexity.
- Make sure every code cell runs **without errors**.
- Ensure the notebook is organized into sections using markdown headings.

Notebook Structure (Recommended)

1. **Title and Student Information**
   a. Full name(s)
   b. ID number(s)
2. **Environment Setup**
   a. Import all needed libraries
3. **Task Breakdown**
   a. Each question starts with a markdown explanation
   b. Code cells placed directly under the corresponding question
4. **Results and Discussion**
   a. Summaries, comments, or graphs if required
5. **Final Conclusions (optional)**

Additional Notes

- Test your notebook before submission to ensure all outputs appear correctly.
- If using Google Colab, make sure to download the .ipynb file before submitting.
- When working in pairs, only **one student** uploads the assignment, but **both names and IDs** must appear clearly.
- Keep your answers concise, but demonstrate understanding.
- If the homework includes working with files or datasets, include clear instructions on how to run the notebook.
- If you have any questions, please contact the instructor **before** the submission deadline.

Question 1 — Convolution

Before starting, **load an image of your choice** (grayscale or convert it to grayscale). All tasks below must be performed on this image using your own manual implementations.

1. Manual Convolution Implementation

Implement a **manual 2D convolution function**:

- Do **not** use `cv2.filter2D`, `scipy`, or any other built-in convolution functions.
- Your function must support:
    - **Zero-padding**
    - **Arbitrary kernel sizes**
    - **Both even and odd filter dimensions**

2. Test Your Convolution Function

Apply your manual convolution function using the following kernels:

A. Box filter (simple averaging blur)

Use box filters of sizes:

- **3×3, 5×5, 7×7**

B. Gaussian kernels

Use Gaussian kernels of sizes:

- **3×3, 5×5, 7×7 (compute manually using formula with σ or use standard deviations 0.8, 1, 1.2)**

You may:

- Generate them manually
- Or compute them using the Gaussian formula

C. Sharpening kernels (3×3 only)

Use the following **3×3** sharpening kernels:

• **Basic sharpening kernel:**

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$

• **Sobel kernels** : **Sobel-X and Sobel-Y**

3. Compare With OpenCV

Compare your manual convolution outputs with the built-in OpenCV implementation:

- Use `cv2.filter2D` with the **exact same kernels**
- Show the results **side-by-side images of manual and OpenCV results.**
- Evaluate visually and numerically Compute **numerical differences** (e.g., MAE or MSE) between outputs.

4. Compare Smoothing Behaviors

Compare **box filters vs Gaussian filters**:

- Which filter produces **more excessive smoothing**?
- Why does **Gaussian smoothing preserve edges better** than simple averaging?

Write a short explanation supported by examples.

5. Explain Differences Between Your Output and OpenCV

Discuss why results may differ slightly between your manual method and `cv2.filter2D`:

- Numerical precision
- Border handling differences

Question 2 – Edge Detection

Before starting this part, **load a grayscale image** (or convert a color image to grayscale).
All steps below must be applied to this grayscale image using your own manual implementations from previous sections.

1. Implement Sobel operator manually using your convolution function.

Using your convolution function. Use both Sobel-X and Sobel-Y kernels.

2. Compute the gradient magnitude and gradient orientation.

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

3. Apply thresholding to produce a binary edge map.

You may choose a fixed threshold or experiment with multiple values.

4. Compare with results from cv2.Sobel.

Show images side-by-side for:

- Sobel-X
- Sobel-Y
- Gradient magnitude

5. Include visual comparison and explanation of differences.

Question 3 – Canny Edge Detection

Before starting this part, **load a grayscale image** (or convert a color image to grayscale).
You will manually implement the complete Canny edge detection pipeline step-by-step using your own code (no built-in Canny).

1. Noise Reduction using Gaussian Blur

Apply Gaussian smoothing to reduce image noise before computing edges.
You must:

- Use a **Gaussian kernel** (3×3, 5×5, or 7×7)
- Convolve it manually using your convolution function
- Explain why noise reduction is critical for preventing false edges

Output required:

- Original grayscale image
- Blurred image

2. Gradient Computation using Sobel Operators

Compute horizontal and vertical gradients:

- Apply **Sobel-X** and **Sobel-Y** kernels
- Use your manual convolution implementation

Then compute:

- **Gradient magnitude:**
- **Gradient orientation:**

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Output required:

- Sobel-X result
- Sobel-Y result
- Gradient magnitude image
- Gradient orientation (you may show angle map or explain it)

3. Non-Maximum Suppression (NMS)

Thin the edges by keeping only the local maxima along the gradient direction.

You must:

- Quantize gradient directions to 0°, 45°, 90°, or 135°
- Compare each pixel with the proper two neighbors
- Suppress (set to zero) any pixel that is not the maximum

Output required:

- NMS result (thin edges)

Provide an explanation of why NMS is essential to get clean, single-pixel-wide edges.

4. Double Thresholding

Classify pixels into:

- **Strong edges** (above high threshold)
- **Weak edges** (between low and high threshold)
- **Non-edges** (below low threshold)

Choose reasonable thresholds (ex: 30/100 or your own).
You may experiment and compare different threshold values.

Output required:

- Thresholded image (strong/weak/non-edge visualization)

Explain why double thresholding helps remove noise and false positives.

5. Edge Tracking by Hysteresis

Connect weak edges that are linked to strong edges.
Remove weak edges that are isolated (not connected to a strong edge).

You must:

- Implement hysteresis manually (recursive or BFS/DFS permitted)
- Convert final edges into a clean binary image

Output required:

- Final Canny edge map (binary image)

Explain how hysteresis prevents fragmentation of edges and removes noise.

6. Final Task

Compare your final output with **OpenCV's cv2.Canny**:

- Show your result and OpenCV's result side-by-side
- Explain any differences:
  - Different Gaussian smoothing
  - Threshold settings
  - Precision differences
  - Implementation details (NMS, hysteresis, etc.)