Programming Assignment - Low Level Vision

Divya Madhuri (ee24d004@smail.iitm.ac.in)
Advaith H Raj (ee21b007@smail.iitm.ac.in)

Due date: March 4, 2026, 11:59 pm

Instructions

The goal of this assignment is to implement classical low level vision methods discussed in class and analyze the effect of noise, smoothing, and parameter selection. The assignment also involves implementing corner detection, blob detection and SIFT from scratch.

Programming Language: Python

Allowed Libraries: `numpy`, `cv2`, `matplotlib`, `scipy.ndimage`

Not Allowed: Using built-in methods as the main solution. (They may be used only for verification and comparison of your results)

Late Submission Policy: Late submissions will be penalized by 10% of the total assignment marks.

- Post any and all doubts on Moodle. This will also be helpful to your peers.
- The assignment primarily involves implementing edge detection, blob detection, and SIFT from scratch. Several online resources may be available for reference; however, refrain from directly copying code. If you take help from any source, cite it clearly in a *References* section.
- You are allowed to use online and offline resources for assistance, but using AI tools such as ChatGPT, Gemini, or similar models to write code is strictly prohibited. If found, you will receive a zero for the assignment.
- Submit the code (both the original notebook and a PDF version) in a zip file named `PA1_RollNumber.zip` using the submission link provided on Moodle.
- Your notebook and pdf should display results strictly adhering to the hyperparameter values given in this assignment.
- This assignment is an extension of the previous tutorials, and the code/ideas implemented in the tutorial will be useful.

Section 1: Edge Detection [Marks: 5]

Task: Canny Edge Detection

Implement the complete Canny edge detection algorithm from scratch according to the steps given below. Use `"cameraman.png"` for this exercise
Implementation Requirements:

1. **Effect of Smoothing and Operator Choice on Gradient Estimation**

   Begin by computing image gradients to analyze the impact of both noise and derivative kernels.

   - **Baseline (Unsmoothed):** Compute the horizontal and vertical derivatives I_x and I_y on the **unsmoothed image** using Finite Difference operator (Filter kernels $K_x = [-1, 0, 1]$, $K_y = [-1, 0, 1]^T$).
   - **Operator Comparison (Unsmoothed):** Compute the derivatives on the **unsmoothed image** using the **Sobel operator**.
   - For both cases, calculate the gradient magnitude $M = \sqrt{I_x^2 + I_y^2}$ and direction $\theta = \text{arctan2}(I_y, I_x)$ (Use `np.arctan2()`).

   Next, apply an explicit Gaussian smoothing filter ($\sigma = 0.33$, kernel size $= 3$) to the unsmoothed image and recompute the gradients using Finite Difference operator.

   Submission Requirements:

   - Plot gradient magnitude using Finite Difference (on Unsmoothed image).
   - Plot gradient magnitude using Sobel (on Unsmoothed image).
   - Plot gradient magnitude using Finite Difference (After applying Gaussian Smoothing).

   Compare the gradient magnitudes obtained from the direct Sobel operator against the two-step Gaussian Smoothing ($\sigma = 0.5, k = 3$) + Finite Difference pipeline.

   Question: Between these two methods, which is more computationally efficient and why?

2. **Directional Quantization (180°):** Using the gradient angle calculations from Sobel filter, categorize gradient orientations into **eight** sectors spanning a 180° range. In this model, opposing directions (e.g., 0° and 180°) are considered equivalent.

   - Map the gradient from the raw range of $(-\pi, \pi)$ (as outputted by `np.arctan2()`) to a positive range of $[0, 180°)$.
   - Quantize the 180° span into 8 equal sectors, each of width 22.5°.

   Repeat the same for **four** sectors (each spanning 45°) as well and elaborate on the results.

   Submission Requirement: Plot two color-coded visualizations representing the gradient orientations quantized into 4 and 8 sectors respectively. To accurately represent the angular continuity, where 179° and 0° are neighbors, utilize a cyclic colormap (such as `hsv` or `twilight`) to ensure the ”wrap-around” is visually seamless.

3. **Non-Maximum Suppression (NMS):** Implement NMS to produce thin, one-pixel-wide edges by suppressing gradient responses that are not local maxima along the gradient direction.

   Your implementation must explicitly perform the following steps:

   - **Directional Mapping:** Using your 8-sector quantized directions (spanning $0°$ to $180°$), identify the two specific neighboring pixels (p_1, p_2) located along the gradient axis.
   - **Local Comparison:** For each pixel, compare its gradient magnitude $M(x, y)$ with the magnitudes of neighbors p_1 and p_2.
   - **Suppression:** Preserve the pixel magnitude only if $M(x, y) > M(p_1)$ **and** $M(x, y) > M(p_2)$. If it is not a local maximum, suppress its value to zero.
   - **Boundary Handling:** Explicitly handle image borders (e.g., via padding or skipping the outermost pixel row/column) to prevent indexing errors.

   Submission Requirement: Plot the resulting edge map after Non-Maximum Suppression. The output should consist of "thinned" edges (1-pixel wide) representing the ridges of the gradient magnitude.

4. **Double Thresholding & Iterative Hysteresis:** Following Non-Maximum Suppression, classify the remaining candidate pixels using a dual-threshold scheme and an iterative tracking algorithm.

   - **Classification:** Categorize pixels into three sets based on their magnitude M:
     - *Strong Edges*: $M > T_{high}$
     - *Weak Edges*: $T_{low} < M \leq T_{high}$
     - *Suppressed*: $M \leq T_{low}$

     Take the threshold values: $T_{high} = 200$, $T_{low} = 50$ scaled to 255.
   - **Recommended:** To ensure robust edge detection, it is recommended to normalize the gradient magnitude using **percentile-based scaling**. Specifically, mapping the 98th percentile of the magnitude values to the maximum of the display range (e.g., 255) prevents a few high-intensity outliers from squashing the signal of legitimate background edges.
   - **Iterative Hysteresis Tracking:** Implement an edge-tracking algorithm using **8-connectivity**. This process must be **iterative**:
     - A weak pixel is promoted to a strong edge if it is connected to a strong pixel.
     - This promotion must propagate; once a weak pixel is promoted, it can subsequently promote neighboring weak pixels in a chain reaction.
     - Continue this process until no more weak pixels can be promoted.

   Submission Requirement: Apply the thresholding and hysteresis pipeline to **both** the 4-sector and 8-sector Non-Maximum Suppression outputs. Plot the resulting edge maps side-by-side and discuss how the granularity of directional quantization affects the final connectivity and structural integrity of the edges.

5. **Comparison with OpenCV Implementation:**

   To validate your results, compare both your custom 4-sector and 8-sector pipelines against the industry-standard OpenCV implementation.

   - Apply the `cv2.Canny` function to the original image. To ensure a fair comparison, use the same threshold values. Note: Since `cv2.Canny` typically expects 8-bit input, you must scale your thresholds relative to 255.

   Submission Requirement: Plot three images side-by-side: your Final 4-sector result, your Final 8-sector result, and the OpenCV Canny result.

   Discussion Point: Compare the "cleanliness" and connectivity of your manual implementations against OpenCV. Specifically, how does the 8-sector neighbor-checking logic approach the quality of OpenCV's implementation compared to the 4-sector version?

Your submission must include the following visualizations and analysis:

- Gradient magnitude **before** smoothing.
- Gradient magnitude **after** smoothing, with a brief explanation of the observed differences.
- A color-coded map showing the quantized gradient direction sectors.
- The Gradient Magnitude **before** Non-Maximum Suppression (NMS).
- The Gradient Magnitude **after** NMS (clearly demonstrating the thinning effect).
- A "Diagnostic Edge Map" where pixels are colored differently based on their origin:
  - *Type A:* Strong pixels above the high threshold.
  - *Type B:* Weak pixels retained via hysteresis.
- A side-by-side comparison between:
  - Your custom Canny implementation
  - `cv2.Canny`

Section 2: Blob Detection [Marks: 2.5]

Blobs are regions in an image that differ in properties, such as brightness or color, compared to surrounding regions. In this section, you will explore the Laplacian of Gaussian (LoG) as a scale-space blob detector. Use `"sunflower.png"` image for this section.

Task 1: Laplacian of Gaussian (LoG)

Edges can be detected using second derivatives, but the Laplacian operator is highly sensitive to noise.

1. Implement the Laplacian of Gaussian (LoG) filter in two ways:
   (i) apply Gaussian smoothing for image (f) followed by the Laplacian operator, and
   (ii) use the differentiation property of convolution $\nabla^2(f * G_\sigma) = f * (\nabla^2 G_\sigma)$.
   Verify that both methods produce approximately the same output, and compare their computation time .

2. Implement this using three different values of σ = [1, 1.6, 2] .

Submission Requirements
For the Laplacian of Gaussian (LoG) task, please include the following in your notebook:

1. For each scale $\sigma \in \{1, 1.6, 2\}$, provide a figure displaying:
   - The original input image.
   - The result of Method (i): Sequential Gaussian smoothing and Laplacian operator.
   - The result of Method (ii): Direct convolution with the LoG kernel.
   - A brief verification (e.g., a difference image or mean squared error) confirming the numerical equivalence of both approaches.
   - Computational time of the two methods and comparison between the two.
2. Provide a technical explanation addressing why the second derivative (Laplacian) exhibits higher sensitivity to noise compared to first-order gradient operators.

Task 2: Scale-Space Representation

Real-world blobs appear at different sizes. To detect them, we must look across multiple scales.

1. Create a "stack" of LoG-filtered images (or Difference of Gaussians) by varying σ in a geometric progression (e.g., $\sigma, k\sigma, k^2\sigma ..., k^i\sigma$). Consider number of scales (i) = 6 and k = 1.4.

2. Use the **scale-normalized Laplacian** $\sigma^2\nabla^2 G$ to ensure that the response strength is consistent across different scales.

Submission Requirement: Your notebook should include a dedicated section visualizing the resulting scale-space stack. Display a grid of subplots showing the LoG response at each σ level.

Task 3: Multi-Scale Blob Detection

A blob is defined as a point (x, y) that is a local extremum (maximum or minimum) across both spatial dimensions and the scale (σ) dimension.

1. **3D Non-Maximum Suppression (NMS):** Implement a search algorithm to identify local extrema within the $3 \times 3 \times 3$ neighborhood of the scale-space stack (comparing a pixel to its 8 spatial neighbors and 18 neighbors in the adjacent scales).

- Retain only strict local maxima or minima.
- Apply response threshold ($|LoG| \geq 0.2$) to suppress noise and weak features.

2. **Visualization:** Overlay the detected blobs as circles on the original image. The radius of each circle should be proportional to the scale at which it was detected, $r = \sqrt{2}\sigma$.
3. **Scale Analysis:** Using the three specific σ values from Task 1, identify which scale captures the maximum number of "appropriate" blobs (i.e., those centered accurately on the sunflower heads).

Submission Requirements

Your notebook must include:

- **Detection Plot:** Plot the original image with all detected blobs overlaid as circles. Ensure the circle radii correctly reflect the σ of detection.
- **Scale Discussion:** An explanation of which σ value provided the best "fit" for the objects in the image (in this case, the sunflowers)

Section 3: Corner Detection [Marks: 2.5]

Task 1: Heuristic Corner Detection via Edge Intersection

Before implementing formal detectors, we can approximate corners as points where both horizontal and vertical gradients are high. Use `oswald.png` image for this section

1. Using the gradients I_x and I_y from Section 1, generate three corner maps by identifying pixels where both magnitudes exceed threshold values 0.7, 0.8 and 0.9 respectively.
2. Display the individual I_x and I_y maps and the resulting intersection map overlaid on the original image.

Submission Requirements: For the heuristic corner detection task, your notebook must include:

1. A figure containing the horizontal gradient magnitude $|I_x|$ and the vertical gradient magnitude $|I_y|$. These should clearly show the responses to vertical and horizontal edges, respectively.
2. An overlay plot on the original image where detected "heuristic corners" are marked.

Task 2: Harris Corner Detection

The Harris Corner Detector provides a more robust solution by analyzing the local distribution of gradients through the structure tensor.
Implementation Requirements:

1. From the custom functions used in Section 1, compute the image gradients I_x and I_y using Sobel operators, and then compute products of gradients at every pixel: I_x^2, I_y^2, and $I_{xy} = I_x I_y$.

2. Apply Gaussian smoothing ($\sigma = 0.4$ and kernel size $= 3$) to these product images to compute the local averages (this forms the structure tensor M).

3. Calculate the Harris response R for each pixel:

$$R = \det(M) - k(\text{trace}(M))^2$$

   (Use values $0.04, 0.05$ and 0.06 for k and elaborate on the results).

4. Perform non-maximum suppression to find local peaks in R and apply a threshold to identify corners.

Your submission should include:

- A heatmap of the Harris response R for each value of k used.
- The original image with detected corners overlaid as points, for each R.
- An explanation on how the parameter k and the Gaussian window size σ influence the results.

(Optional) Section 4: SIFT Keypoint Detection and Descriptor Extraction [Marks: 2.5]

SIFT (Scale-Invariant Feature Transform) is a widely used feature detection and description algorithm. It detects stable keypoints across different scales using a Difference of Gaussian (DoG) scale-space representation and generates descriptors for matching. Use `church002.png` and `church003.png` for this section

Task 1: DoG-based Keypoint Detection (SIFT Detector)

1. Construct a Gaussian pyramid (multi-octave scale-space) as follows: Use $s = 3$ scales per octave, base scale $\sigma_0 = 0.6$, and scale factor $k = 2^{1/s}$. For each octave, generate Gaussian blurred images at scales $\sigma_0, k\sigma_0, k^2\sigma_0, \ldots$.

2. After generating the Gaussian images for one octave, downsample the image by a factor of 2 to form the next octave. Repeat this process till we have 3 octaves in total.

3. Compute the Difference of Gaussians (DoG) images for each octave by subtracting consecutive Gaussian blurred images.

4. Detect candidate keypoints by finding local extrema in the DoG scale-space using 3D non-maximum suppression (compare each pixel with its 26 neighbors in a $3 \times 3 \times 3$ neighborhood across space and scale).

5. Apply thresholding to remove low-contrast keypoints: accept points where $|DoG(x, y)| \geq 0.01$.
6. Visualize detected keypoints on the original image as circles with radius proportional to the detected scale.

Task 2: Basic Descriptor Construction

1. For each detected keypoint, extract a 16×16 patch centered at the keypoint at the appropriate scale.
2. Compute the gradient magnitude and orientation at each pixel in the patch.
3. Divide the patch into a 4×4 grid of cells (each cell is 4×4 pixels).
4. For each cell, compute an 8-bin histogram of gradient orientations (45° per bin, covering 0° to 360°), weighted by gradient magnitude.
5. Concatenate the 16 histograms to obtain a 128-dimensional descriptor: $4 \times 4 \times 8 = 128$.
6. Normalize the descriptor to unit length.
7. Visualize a few example descriptors.

Task 3: Descriptor Matching

1. Extract descriptors from two images.
2. Match descriptors using **Lowe's ratio test**:

   $$\frac{d_1}{d_2} < 0.75$$

   where d_1 is the Euclidean distance to the nearest neighbor and d_2 is the distance to the second-nearest neighbor.
3. Visualize matches by drawing lines connecting matched keypoints.

Task 4: Effect of Scales per Octave

1. Vary the number of scales sampled per octave: $s \in \{2, 3, 4, 5\}$.
2. For each value of s, compute $k = 2^{1/s}$ and perform DoG-based keypoint detection on the same test image.
3. Plot a graph of the number of detected keypoints versus s (number of scales per octave).
4. Briefly discuss the observed trend:
   - Does the number of keypoints increase or decrease with more scales?

Task 5: Discussion and Analysis

1. **Scale Invariance:** How does SIFT achieve robustness to scale changes? Which component(s) of the algorithm are responsible?

2. **Rotation Invariance Problem:** Your basic implementation likely fails on rotated images. Explain why this happens. What information is missing from the descriptor that would help with rotation?

3. **Illumination Invariance:** The current normalization (unit length) provides some illumination invariance. Explain what additional normalization step (clipping) could further improve robustness to lighting changes.

Submissions

Your submission must include:

Visualizations

- Gaussian pyramid and DoG pyramid visualizations (Task 1)
- Detected keypoints overlaid on original image with circles proportional to scale (Task 1)
- Example descriptor visualizations showing the 4×4 grid structure (Task 2)
- Plot showing number of detected keypoints versus scales per octave (s) for Task 4