THE UNIVERSITY OF DANANG
UNIVERSITY OF SCIENCE AND TECHNOLOGY
Faculty of Advanced Science and Technology

Image Processing

Final Examination Report

Instructor : TS. Hồ Phước Tiến
Class : 21ES
Group members : Lê Quốc Duy - 123210016
Trần Vũ Hồng Phúc - 123210021

Da Nang, 27th April 2025

FINAL - 04/27

Image Processing

Students:

1. Le Quoc Duy - 123210016

2. Tran Vu Hong Phuc - 123210021

Problem 1. *Looking for a patch in an image*

In the image “barbara.jpg”, find the position of the patch represented by the image “patch.png”. Draw a bounding box on “barbara.jpg” to indicate this patch. (Hint: use the Euclidean distance between two vectors or two images to look for similar patches.)

Repeat the above process by replacing “patch.png” with “patch_1.png”. Comment.

[Code - Python]

REPLACE patch1.png if you want to run and check this code for patch1

```python
# Le Quoc Duy - Tran Vu Hong Phuc - 27/04/2025
import cv2
import numpy as np
import os
# --- Configuration ---
main_image_path = 'barbara.jpg'
patch_image_path = 'patch.png'
output_image_path = 'barbara_with_patch_box.jpg'
def calculate_euclidean_distance(patch1, patch2):
    """Calculates the Euclidean distance between two image patches."""
    # Ensure patches have the same shape
    if patch1.shape != patch2.shape:
        raise ValueError("Patches must have the same dimensions and channels.")

    # Flatten the patches and calculate distance
    # Alternatively, calculate squared distance directly on arrays for efficiency
    # dist = np.sqrt(np.sum((patch1.astype(float) - patch2.astype(float))**2))
```

```
    # Using squared Euclidean distance for comparison (avoids costly
sqrt)
    # Convert to float to prevent overflow issues with uint8
subtraction
    dist_sq = np.sum((patch1.astype(np.float64) -
patch2.astype(np.float64))**2)
    return dist_sq
print(f"Loading main image: {main_image_path}")
if not os.path.exists(main_image_path):
    print(f"Error: Main image file not found at {main_image_path}")
    exit()
main_img = cv2.imread(main_image_path)
if main_img is None:
    print(f"Error: Could not load main image {main_image_path}")
    exit()

print(f"Loading patch image: {patch_image_path}")
if not os.path.exists(patch_image_path):
    print(f"Error: Patch image file not found at {patch_image_path}")
    exit()
patch_img = cv2.imread(patch_image_path)
if patch_img is None:
    print(f"Error: Could not load patch image {patch_image_path}")
    exit()
main_h, main_w = main_img.shape[:2]
patch_h, patch_w = patch_img.shape[:2]

print(f"Main image dimensions (HxW): {main_h} x {main_w}")
print(f"Patch image dimensions (HxW): {patch_h} x {patch_w}")

# Check if patch is larger than the main image
if patch_h > main_h or patch_w > main_w:
    print("Error: Patch image is larger than the main image.")
    exit()
min_distance = np.inf
best_match_coords = (0, 0) # (y, x) of top-left corner

print("Searching for the best match using Euclidean distance...")
# Iterate through all possible top-left corners (y, x) for the patch
for y in range(main_h - patch_h + 1):
    for x in range(main_w - patch_w + 1):
```

```
        # Extract the current window/candidate patch from the main
image
        candidate_patch = main_img[y : y + patch_h, x : x + patch_w]

        # Calculate the distance (using squared Euclidean for
efficiency)
        distance = calculate_euclidean_distance(patch_img,
candidate_patch)

        # Update if this is a better match
        if distance < min_distance:
            min_distance = distance
            best_match_coords = (y, x) # Store NumPy-style coordinates
(row, col)
best_y, best_x = best_match_coords
top_left = (best_x, best_y)          # OpenCV uses (x, y) for drawing
bottom_right = (best_x + patch_w, best_y + patch_h)

print(f"Best match found!")
print(f" -> Minimum Squared Euclidean Distance: {min_distance:.2f}")
print(f" -> Top-left corner (x, y): {top_left}")
print(f" -> Bottom-right corner (x, y): {bottom_right}")
output_img = main_img.copy()
box_color = (0, 0, 255) # BGR format for Red
box_thickness = 2
cv2.rectangle(output_img, top_left, bottom_right, box_color,
box_thickness)
print(f"Saving result with bounding box to: {output_image_path}")
cv2.imwrite(output_image_path, output_img)

print("Displaying result. Press any key to close.")
cv2.imshow('Main Image', main_img)
cv2.imshow('Patch', patch_img)
cv2.imshow('Found Patch Location', output_img)

# Wait for a key press and then close windows
cv2.waitKey(0)
cv2.destroyAllWindows()

print("Done.")
```

[Figures/Tables]

Figure 1: barbara with the patch image

```
Best match found!
 -> Minimum Squared Euclidean Distance: 0.00
 -> Top-left corner (x, y): (369, 379)
 -> Bottom-right corner (x, y): (400, 410)
```

Figure 2: Best match found using Euclidean of patch.img

Figure 3: Here is the image after we zoom in to the bounding boxes, its match the patch

Figure 4: Barbara with patch1 image

```
Best match found for 'patch_1.png'!
 -> Minimum Squared Euclidean Distance: 373407.00
 -> Top-left corner (x, y): (404, 325)
 -> Bottom-right corner (x, y): (433, 354)
```

Figure 5: Best match found Euclidean of patch1.img

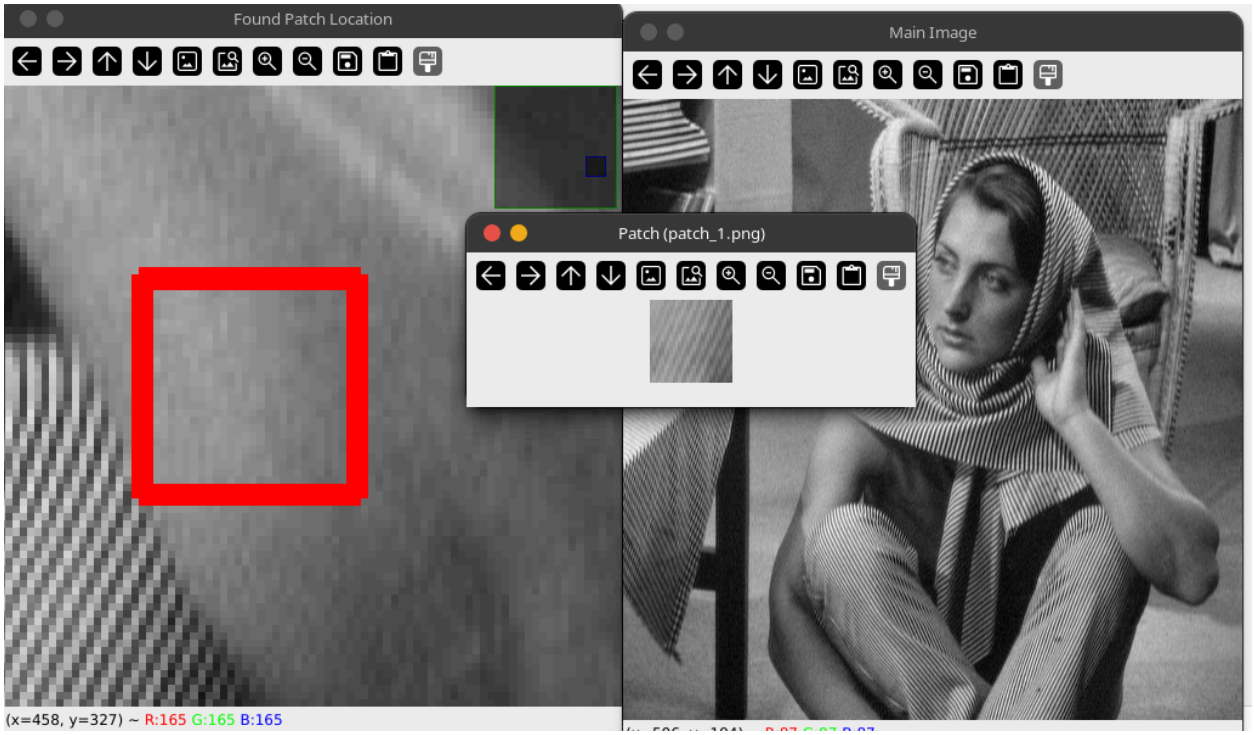

Figure 6: Here is the image barbara after zooming in patch1.img

[Explanation]

Analysis of Result 1 (using patch.png)

1. **Input Patch (patch.png):** A 31x31 patch containing a distinct diagonal striped pattern.
2. **Identified Location:** Top-left corner at (x=369, y=379) on Barbara's trousers/leg.
3. **Minimum Squared Euclidean Distance:** 0.00
4. **Interpretation:** A minimum squared Euclidean distance of zero is definitive. It means that every single corresponding pixel in the 31x31 patch.png was identical to the pixel value at the matched location (369, 379) in barbara.jpg. This overwhelmingly suggests that patch.png was likely created by directly extracting that precise 31x31 block from the original barbara.jpg.

5. **Effectiveness:** This result perfectly illustrates the power of Euclidean distance (and template matching in general) for finding exact duplicates. When the template is identical to a region in the target, the distance metric correctly identifies it with zero error. The distinct nature of the striped pattern likely made it easy for the minimum distance to be clearly identified at this location compared to any other.

Analysis of Result 2 (using patch_1.png)

1. **Input Patch (patch_1.png):** A 29x29 patch, **which visually contains a diagonal striped pattern** similar to patch.png.
2. **Identified Location:** Top-left corner at (x=404, y=325) on Barbara's upper arm/shoulder area.
3. **Minimum Squared Euclidean Distance:** 373,407.00
4. **Interpretation:** The minimum squared Euclidean distance found was **373,407.00**, occurring in a region of skin texture on the arm. Crucially, **this location is visually incorrect**; it does not match the striped pattern present in patch_1.png. The large, non-zero distance value indicates that *no region* in barbara.jpg was a close pixel-by-pixel match to patch_1.png. The algorithm mathematically identified the arm region as having the *lowest* sum of squared differences, even though visually dissimilar areas (like other striped sections) exist.
5. **Quantifying the Difference:** Calculating the average squared error per pixel between patch_1.png and the identified arm region gives 373407 / (29 * 29) ≈ 443.9. This confirms the substantial numerical difference and poor quality of the match *even at the location of minimum distance*.
6. **Nature of the Match Failure:** This demonstrates a significant limitation of the Euclidean distance metric. Despite patch_1.png containing a striped pattern, the algorithm failed to locate a visually corresponding striped region. Instead, it found a **numerically "closer" but visually nonsensical match** on the arm. This can happen if the patch differs slightly from the target instances (e.g., brightness, contrast) or if the metric becomes sensitive to average intensity levels or noise, mistakenly finding lower distances in unrelated regions.

Comparative Commentary and Enhanced Insights from Numerical Data:

1. ***Perfect Match vs. Failed Visual Match:*** *The results starkly contrast finding a perfect, visually correct match (patch.png, distance 0.00) with finding a* ***mathematically "best" but visually incorrect match*** *(patch_1.png, distance 373,407.00). The Euclidean distance succeeded only when an exact duplicate existed.*
2. ***Quality of Match Signal:*** *The magnitude of the minimum distance remains a crucial indicator. Zero indicated perfection. The large non-zero value for patch_1.png correctly signaled a poor quality match, foreshadowing the* ***visual mismatch even before it was explicitly confirmed****.*
3. ***Template Origin & Matchability:*** *The zero distance for patch.png supports direct extraction. The non-zero distance for patch_1.png (which contains stripes) indicates it's not identical to any region in barbara.jpg (perhaps due to noise, slight intensity variation, or being from a subtly different striped area). More importantly, it reveals that these small deviations were enough for the* ***Euclidean distance to fail to locate the*** *.*
4. ***Algorithm Behavior & Limitations:*** *Both results show the algorithm correctly minimizing the squared Euclidean distance mathematically. However, the patch_1.png result vividly demonstrates that minimizing this specific metric* ***does not guarantee finding a visually or semantically correct match****, especially when the template isn't identical to regions in the target. It highlights the metric's sensitivity to factors other than structural pattern similarity, making it unreliable for robust pattern detection.*

HERE IS THE OPTIONAL CODE WE USING EDGE DETECTION WITH CANNY TO SEARCH FOR THE PATTERN IN THE barbara.img to find the bounding boxes match with patch.img by using patch1.img

```
import cv2
import numpy as np
import os

#Configuration
main_image_path = 'barbara.jpg'
patch_image_path = 'patch_1.png' # Using patch_1
# Changed output name to reflect edge matching
output_image_path = 'barbara_with_patch_1_edge_match_box.jpg'
```

```
#Canny Edge Detector Thresholds
# These thresholds might need tuning for optimal edge detection!
# Lower threshold for linking edges, higher threshold for initial
strong edges.
CANNY_THRESHOLD_1 = 100
CANNY_THRESHOLD_2 = 200

print(f"Loading main image: {main_image_path}")
main_img = cv2.imread(main_image_path)
if main_img is None:
   print(f"Error: Could not load main image {main_image_path}")
   exit()

print(f"Loading patch image: {patch_image_path}")
patch_img = cv2.imread(patch_image_path)
if patch_img is None:
   print(f"Error: Could not load patch image {patch_image_path}")
   exit()

print("Converting images to grayscale...")
main_img_gray = cv2.cvtColor(main_img, cv2.COLOR_BGR2GRAY)
patch_img_gray = cv2.cvtColor(patch_img, cv2.COLOR_BGR2GRAY)

# Get Dimensions (use grayscale dimensions)
main_h, main_w = main_img_gray.shape[:2]
patch_h, patch_w = patch_img_gray.shape[:2]

print(f"Main image dimensions (HxW): {main_h} x {main_w}")
print(f"Patch image dimensions (HxW): {patch_h} x {patch_w}")

# Check if patch is larger than the main image
if patch_h > main_h or patch_w > main_w:
   print("Error: Patch image is larger than the main image.")
   exit()

# Apply Canny Edge Detection
print(f"Applying     Canny     Edge     Detection     (Thresholds:
{CANNY_THRESHOLD_1}, {CANNY_THRESHOLD_2})...")
```

```
# You might need to adjust the thresholds 100 and 200 for your specific images
main_edges = cv2.Canny(main_img_gray, CANNY_THRESHOLD_1, CANNY_THRESHOLD_2)
patch_edges = cv2.Canny(patch_img_gray, CANNY_THRESHOLD_1, CANNY_THRESHOLD_2)

# Display edge images (optional, but helpful for debugging/understanding)
cv2.imshow('Main Image Edges', main_edges)
cv2.imshow('Patch Edges', patch_edges)
cv2.waitKey(1) # Give windows time to display


#Search for the Patch Edges in the Main Image Edges (Sliding Window)
# We will count the number of overlapping edge pixels (simple correlation)
max_match_value = -1 # Initialize with a value lower than any possible match count
best_match_coords = (0, 0) # (y, x) of top-left corner

print("Searching for the best edge overlap...")
# Iterate through all possible top-left corners (y, x) for the patch edge map
for y in range(main_h - patch_h + 1):
    for x in range(main_w - patch_w + 1):
        # Extract the current window from the main edge map
        candidate_window_edges = main_edges[y : y + patch_h, x : x + patch_w]

        # Calculate Match Score: Count Overlapping Edge Pixels
        # Method 1: Using bitwise AND and summing
        overlap = cv2.bitwise_and(patch_edges, candidate_window_edges)
        # Canny outputs 0 or 255, so divide by 255 to count pixels
        match_score = np.sum(overlap // 255)

        # Method 2: Simple element-wise multiplication and sum (similar)
```

```
        # Since edges are 0 or 255:
                # match_score = np.sum((patch_edges / 255) *
(candidate_window_edges / 255))

        # Update if this is a better match (more overlapping edges)
        if match_score > max_match_value:
            max_match_value = match_score
                    best_match_coords = (y, x) # Store NumPy-style
coordinates (row, col)

#Get Coordinates for Drawing
best_y, best_x = best_match_coords
top_left = (best_x, best_y)             # OpenCV uses (x, y) for
drawing
bottom_right = (best_x + patch_w, best_y + patch_h)

print(f"Best match found for '{patch_image_path}' using Edge
Overlap!")
print(f" -> Maximum Overlapping Edge Pixels: {max_match_value}") #
Higher overlap is better
print(f" -> Top-left corner (x, y): {top_left}")
print(f" -> Bottom-right corner (x, y): {bottom_right}")

#Draw Bounding Box on Original Image
# Use the original color main_img to draw on
output_img = main_img.copy()
# Using BLUE color for this method's box to distinguish it
box_color = (255, 0, 0) # BGR format for Blue
box_thickness = 2
cv2.rectangle(output_img, top_left, bottom_right, box_color,
box_thickness)

#Save and Display Result
print(f"Saving result with bounding box to: {output_image_path}")
cv2.imwrite(output_image_path, output_img)

print("Displaying result. Press any key to close all windows.")
cv2.imshow('Main Image (Original)', main_img)
cv2.imshow('Patch (patch_1.png)', patch_img)
```

```
cv2.imshow('Found Patch Location (Edge Match)', output_img) # Updated window title

# Wait for a key press and then close windows
cv2.waitKey(0)
cv2.destroyAllWindows()

print("Done.")
```

RESULTS

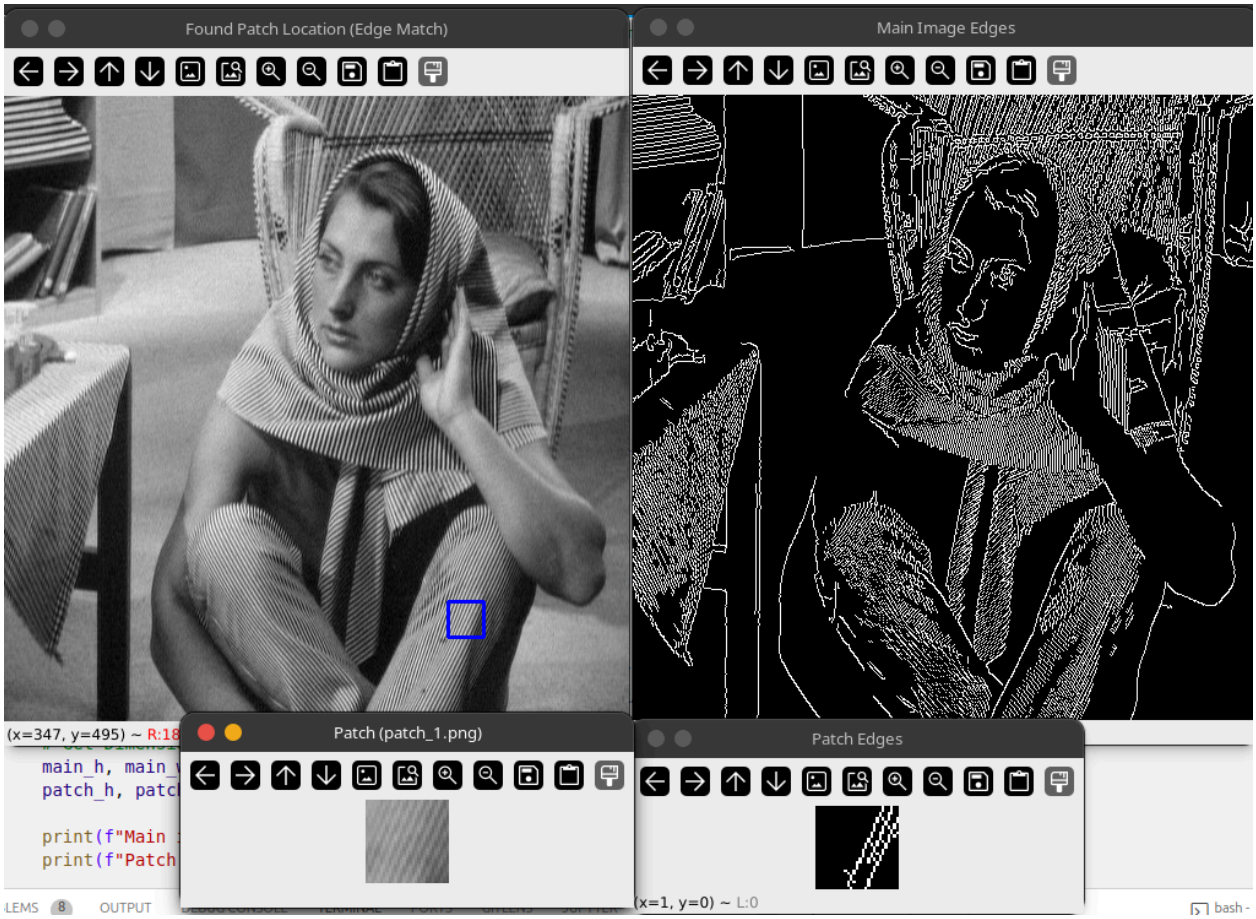

Figure 7: Here is the image after we apply edge detection

→ We found the value quite close to patch.img

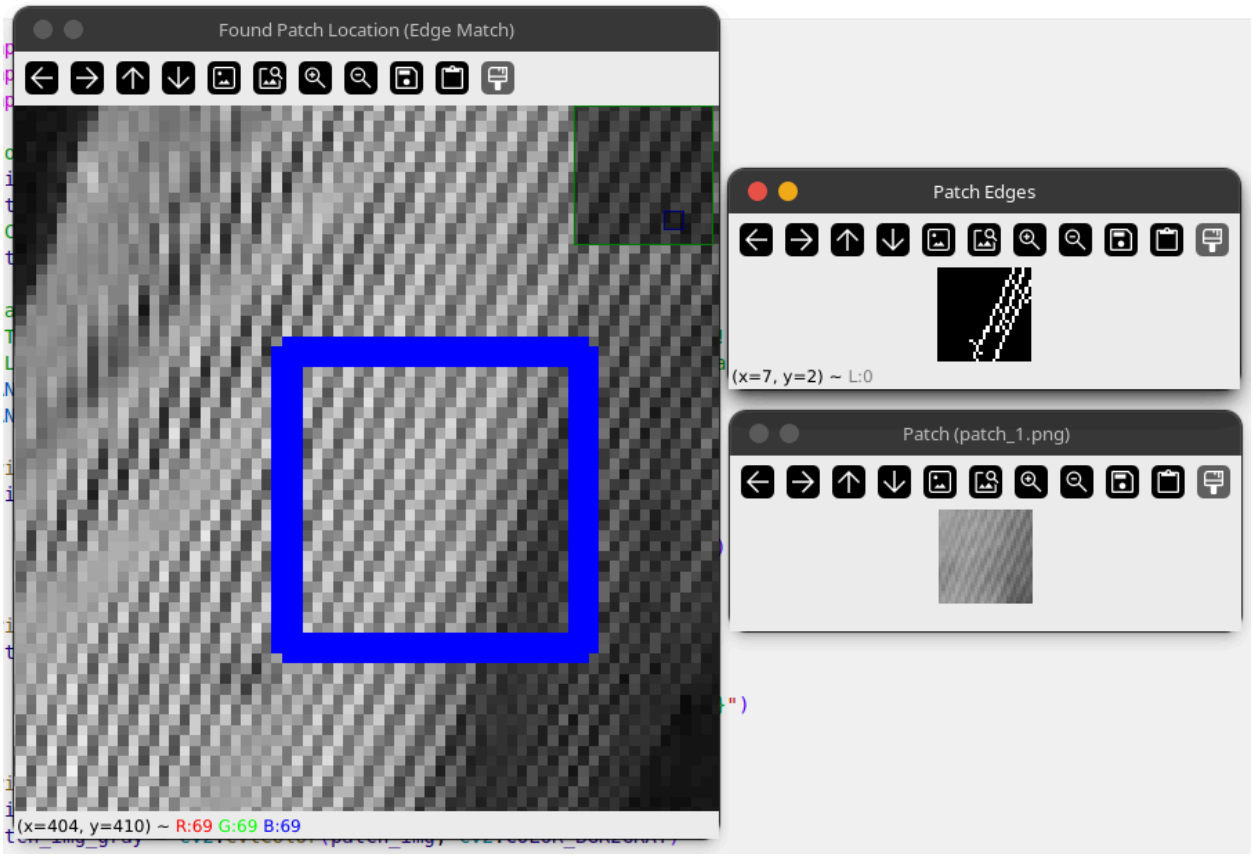

Figure 8: Zooming for the bounding boxes we found with patch1.img

Here is the Best match we found using edge detection:

```
Best match found for 'patch_1.png' using Edge Overlap!
 -> Maximum Overlapping Edge Pixels: 73
 -> Top-left corner (x, y): (362, 412)
 -> Bottom-right corner (x, y): (391, 441)
```

COMMENTS

Comparison: Euclidean Distance vs. Edge Detection for patch_1.png

1. **The Core Problem**: As observed earlier, patch_1.png, while containing stripes, appears somewhat blurred or less sharp than the corresponding features in the main barbara.jpg image (and less sharp than the original patch.png). This slight difference is crucial.
2. **Why Euclidean Distance Failed:**
    a. **Measures Pixel Intensity**: Euclidean distance calculates the difference between the exact brightness/intensity value of *each pixel* in the patch and

the corresponding pixel in the main image window. It squares these differences and sums them up.

b. **Sensitivity to Intensity Changes**: Because patch_1.png is blurred, its *absolute pixel values* are different from the sharp stripes in the main image. Even if the pattern is visually similar, the numerical differences are large. Sharp stripes have high contrast (big differences between light and dark pixel values), while the blurred patch has smoother transitions.

c. **Incorrect Minimum Found**: The process of blurring might have inadvertently made the *average* intensity or the subtle intensity variations within the blurred patch_1.png numerically *closer* (i.e., have a lower sum of squared differences) to the relatively uniform texture of Barbara's arm than to the high-contrast *sharp* stripes elsewhere. The sharp intensity jumps in the correct striped areas created *large* squared differences when compared to the smoother blurred patch, leading to a higher overall Euclidean distance there. The algorithm correctly found the mathematical minimum, but it occurred at the visually wrong location.

3. **Why Edge Detection (Canny + Overlap) Succeeded**:

a. **Focuses on Structure**: The Canny edge detector doesn't look at absolute pixel values. Instead, it identifies locations where there are sharp changes (high gradients) in intensity – essentially, the outlines or borders of features.

b. **Preserving Structure Despite Blur**: While blurring smooth transitions, the underlying structure of the stripes in patch_1.png still creates significant intensity gradients. The "Patch Edges" window confirms this – Canny successfully extracted the pattern of parallel diagonal lines, even from the blurred patch.

c. **Matching Edge Patterns**: The matching process then compares the *edge map* of the patch to the *edge map* of the main image. It searched for the location where the spatial arrangement of edges aligned best (maximizing the overlap count – 73 pixels in your result). The region on Barbara's leg contains a similar spatial arrangement of parallel diagonal edges.

d. **Robustness**: This method effectively ignored the precise intensity values and focused on the higher-level structural information (the layout of the stripes represented by their edges). This structure was less affected by the blur than the raw pixel values were, allowing the algorithm to find the visually correct corresponding pattern.

Problem 2. *Fourier Transform*

Given a grayscale image I, $S(u,v)$ is the Fourier Transform of I. We can represent $S(u,v)$ as follows

$$S(u,v) = |S(u,v)|e^{j\varphi(u,v)}$$

where $\varphi(u,v)$ is the phase spectrum of image I and $|S(u,v)|$ is the magnitude spectrum of image I.

Now we consider two grayscale images I_1 and I_2 (of the same size). Then *we exchange the magnitude spectra of the two images*, i.e. *we combine the magnitude spectrum of one image with the phase spectrum of the other image*.

Requirements:

We reconstruct the resulting images in the spatial domain. Show the results and comment. (The students freely choose the images for simulation.)

What happens if we replace the magnitude spectrum of an image with random values (while keeping the phase spectrum unchanged)?

[Code] - MATLAB

```
% Le Quoc Duy - Tran Vu Hong Phuc - 27/04/2025
clc; clear;
% Problem 2: Fourier Transform - Magnitude and Phase Swapping
% --- Configuration ---
image_file1 = 'cameraman.tif';
image_file2 = 'peppers.png';      % Another alternative standard
image
% --- Load and Prepare Images ---
disp('Loading and preparing images...');
% Load Image 1
I1_orig = imread(image_file1);
if size(I1_orig, 3) == 3
   I1_gray = rgb2gray(I1_orig);
else
   I1_gray = I1_orig;
end
% Load Image 2
try
```

```
    I2_orig = imread(image_file2);
catch ME
    error('Could not read image_file2: %s. Ensure it exists or choose
another image.', image_file2);
end
if size(I2_orig, 3) == 3
    I2_gray = rgb2gray(I2_orig);
else
    I2_gray = I2_orig;
end
% Ensure images are the same size (resize image 2 to match image 1)
if ~isequal(size(I1_gray), size(I2_gray))
    disp('Resizing Image 2 to match Image 1...');
    I2_gray = imresize(I2_gray, size(I1_gray));
end
% Convert to double precision for FFT
I1 = im2double(I1_gray);
I2 = im2double(I2_gray);

% --- Part 1: Swap Magnitude and Phase Spectra ---
disp('Performing Fourier Transform and swapping spectra...');
% 1. Compute 2D FFT
S1 = fft2(I1);
S2 = fft2(I2);
% 2. Extract Magnitude and Phase
Mag1 = abs(S1);
Phase1 = angle(S1);
Mag2 = abs(S2);
Phase2 = angle(S2);
% 3. Create New Spectra by Swapping
% New Spectrum 1 = Magnitude 1 + Phase 2
S_new1 = Mag1 .* exp(1i * Phase2);
% New Spectrum 2 = Magnitude 2 + Phase 1
S_new2 = Mag2 .* exp(1i * Phase1);
% 4. Reconstruct Images using Inverse FFT
I_rec1 = real(ifft2(S_new1)); % Reconstructed image with Mag1, Phase2
```

```
I_rec2 = real(ifft2(S_new2)); % Reconstructed image with Mag2, Phase1

% 5. Display Results for Part 1
figure('Name', 'Part 1: Magnitude and Phase Swapping');
subplot(2, 2, 1);
imshow(I1);
title('Original Image 1');
subplot(2, 2, 2);
imshow(I2);
title('Original Image 2');
subplot(2, 2, 3);
imshow(I_rec1, []); % Use [] for automatic scaling
title('Reconstructed: Mag(I1) + Phase(I2)');
subplot(2, 2, 4);
imshow(I_rec2, []); % Use [] for automatic scaling
title('Reconstructed: Mag(I2) + Phase(I1)');

% --- Part 2: Replace Magnitude with Random Values ---
disp('Replacing magnitude spectrum with random values...');
% Use Image 1 (S1, Mag1, Phase1) for this experiment
% Generate random magnitude spectrum (same size as Mag1)
% Using uniformly distributed random values between 0 and 1
RandMag = rand(size(Mag1));
% Create new spectrum: Random Magnitude + Original Phase 1
S_rand = RandMag .* exp(1i * Phase1);
% Reconstruct the image
I_rand_rec = real(ifft2(S_rand));

% Display Results for Part 2
figure('Name', 'Part 2: Random Magnitude, Original Phase');
subplot(1, 2, 1);
imshow(I1);
title('Original Image 1');
subplot(1, 2, 2);
imshow(I_rand_rec, []); % Use [] for automatic scaling
title('Reconstructed: Random Mag + Phase(I1)');
```

[Figures/Tables]

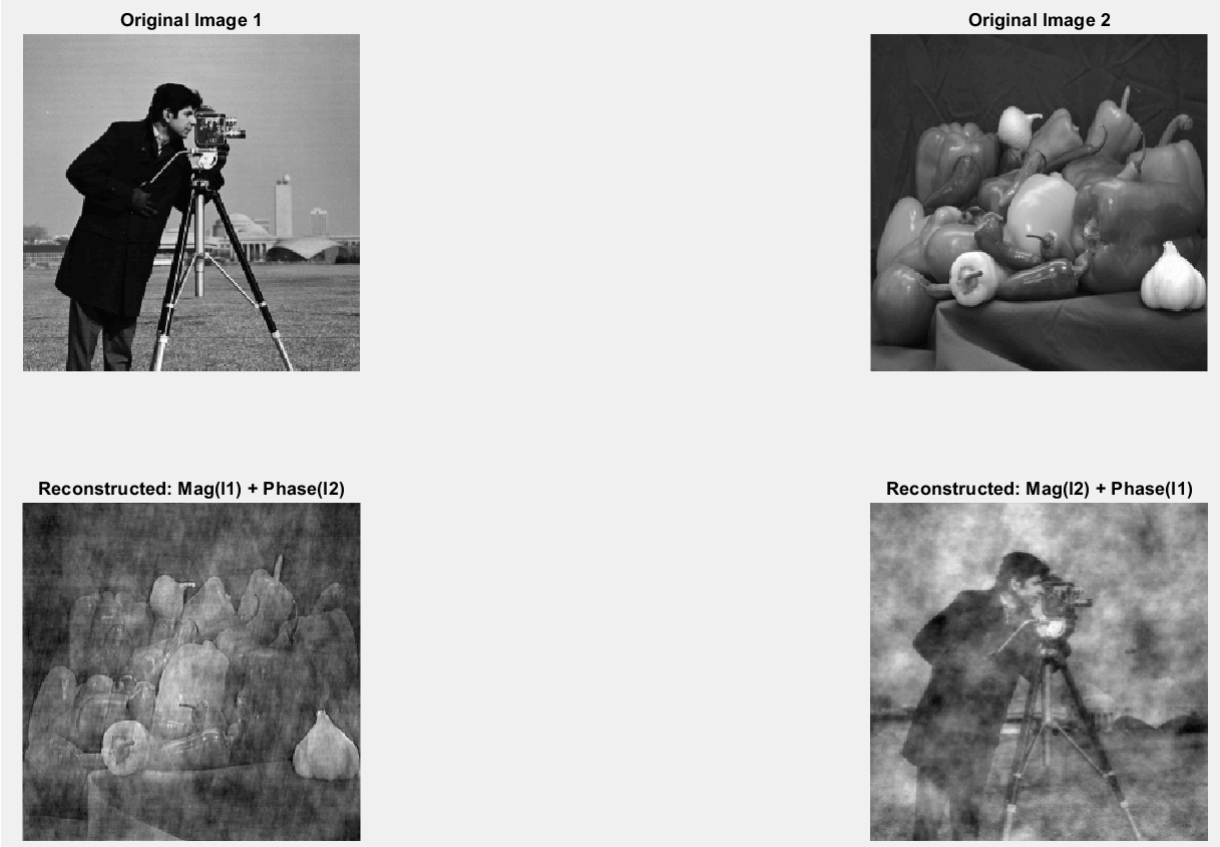

Figure 9: Part 1 - Magnitude and Phase Swapping

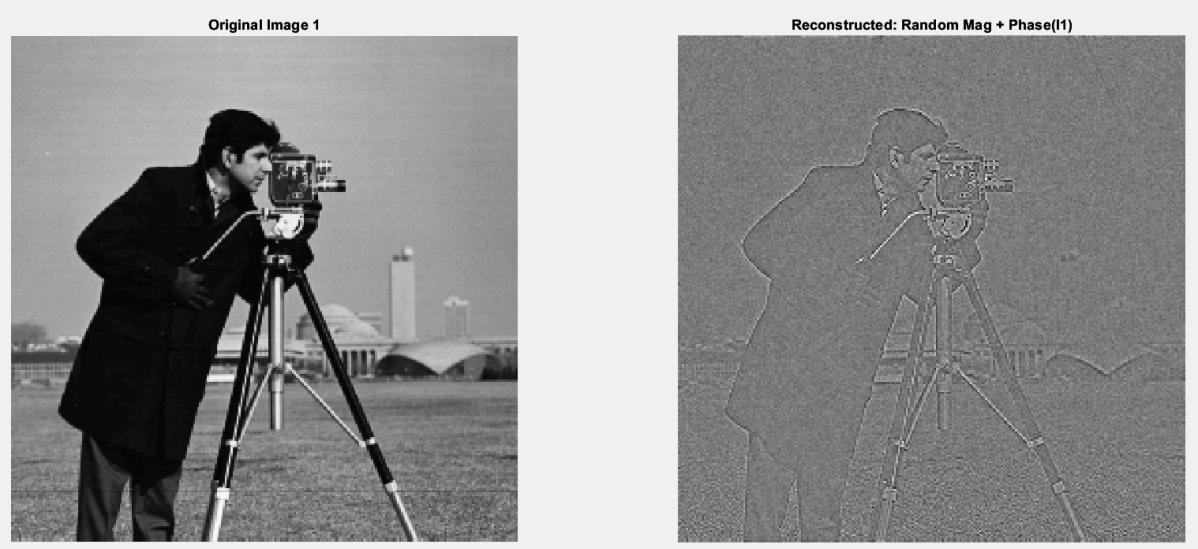

Figure 10: Part 2 - Random Magnitude, Original Phase

[Explanation]

- **Comments for the Figure 9: Part 1 - Magnitude and Phase Swapping:**
  - Observation: The reconstructed images primarily resemble the image whose PHASE spectrum was used.
    ➔ Image "Mag(I1) + Phase(I2)" looks structurally like Original Image 2.
    ➔ Image "Mag(I2) + Phase(I1)" looks structurally like Original Image 1.
  - Conclusion: This demonstrates the dominance of the *phase spectrum* in determining the spatial structure and features (like edges, object shapes) of an image. The *magnitude spectrum* contributes more to the overall contrast and energy distribution, but less to the recognizable structure.

- **Comments for the Figure 10: Part 2 - Random Magnitude, Original Phase:**
  - Observation: When the magnitude spectrum is replaced with random values while keeping the original phase, the reconstructed image loses its original texture and contrast, appearing noise-like. However, faint outlines or ghosts of the original structures (edges, shapes) might still be discernible.
  - Conclusion: This further reinforces the importance of the phase spectrum. Even with a completely random magnitude (random energy distribution across frequencies), the original phase information retains enough spatial positioning data to reconstruct a semblance of the original structure, albeit heavily corrupted by noise.

Problem 3. *K-means segmentation*

Using the K-means method (K = 5) to segment the image "kodim23.png".
a) Represent the data matrix (X) used as input in the K-means method (i.e. write X using mathematical representation).
b) Run the K-means method 3 times and show the segmentation results.
c) How should we do if we want to obtain *the same segmentation result* for several runnings (with the same input and the same value of K)?

[Code] - MATLAB

```
% Problem 3: Le Quoc Duy - Tran Vu Hong Phuc - 27/04/2025
clear; clc; close all;

% --- Configuration ---
image_file = 'kodim23.png';
```

```
K = 5; % Number of clusters
num_runs = 3; % Number of times to run K-means
% --- Load and Prepare Image ---
disp(['Loading image: ', image_file]);
try
   I_color = imread(image_file);
catch ME
   error('Could not read image file: %s. Ensure it exists in the
path.', image_file);
end
% Convert image to double precision [0, 1] for calculations
I_double = im2double(I_color);

% (PART a) Reshape the image into the data matrix X (P x 3) as
described in part (a)
num_rows = size(I_double, 1);
num_cols = size(I_double, 2);
X = reshape(I_double, num_rows * num_cols, 3); % X has P rows, 3
columns (R,G,B)

% (PART b) --- Run K-means Multiple Times ---
figure('Name', 'K-means Segmentation Results (K=5)');
set(gcf, 'Position', [100, 100, 1200, 400]); % Adjust figure size
% Display original image
subplot(1, num_runs + 1, 1);
imshow(I_color);
title('Original Image');
segmented_images = cell(1, num_runs); % Store results
fprintf('Running K-means (K=%d) %d times...\n', K, num_runs);
for run = 1:num_runs
   fprintf(' Run %d...\n', run);
   % Perform K-means clustering
   % so results might differ between runs.
   [cluster_indices, centroids] = kmeans(X, K);
   % Reshape the cluster indices back into image dimensions
   clustered_labels = reshape(cluster_indices, num_rows, num_cols);
   % Create the segmented image using centroid colors
   segmented_image = zeros(size(I_double)); % Initialize output image
   for i = 1:K
       mask = (clustered_labels == i);
       [rows, cols] = find(mask);
       num_pixels_in_cluster = length(rows);
       if num_pixels_in_cluster > 0
           centroid_rgb = centroids(i,:);
```

```
            idx_R = sub2ind(size(I_double), rows, cols,
ones(num_pixels_in_cluster, 1));
            idx_G = sub2ind(size(I_double), rows, cols,
2*ones(num_pixels_in_cluster, 1));
            idx_B = sub2ind(size(I_double), rows, cols,
3*ones(num_pixels_in_cluster, 1));
            segmented_image(idx_R) = centroid_rgb(1);
            segmented_image(idx_G) = centroid_rgb(2);
            segmented_image(idx_B) = centroid_rgb(3);
        end
    end
    segmented_images{run} = im2uint8(segmented_image); % Convert to
uint8 for display
    % Display the segmented image for this run
    subplot(1, num_runs + 1, run + 1);
    imshow(segmented_images{run});
    title(sprintf('Segmentation Run %d', run));
end
fprintf('K-means runs complete.\n');
```

[Figures/Tables]

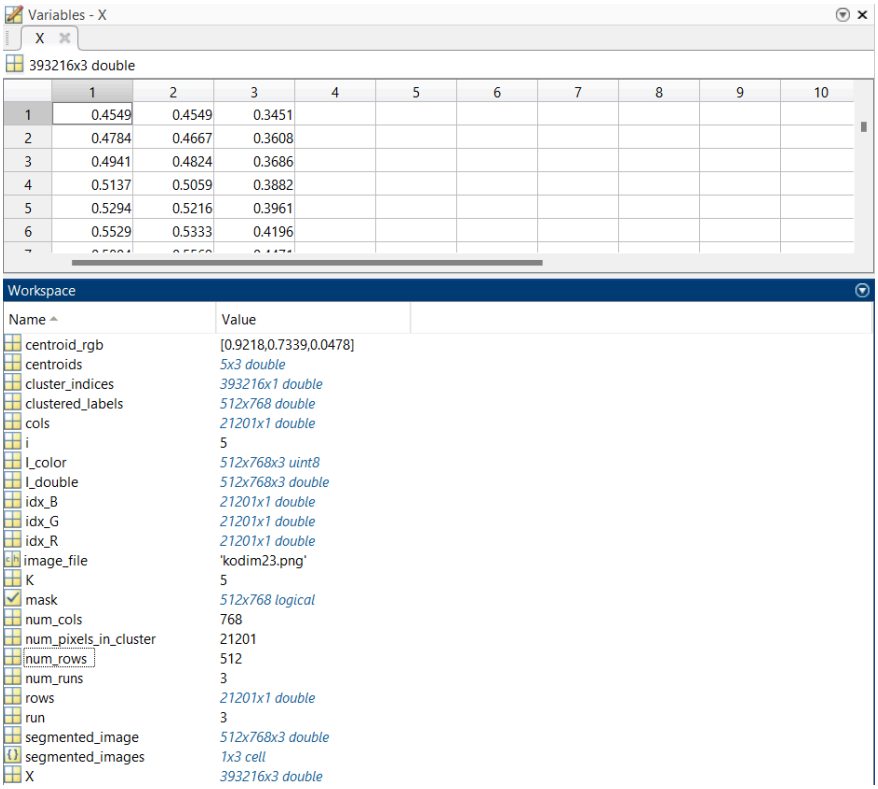

Figure 11: Part a) - Reshape the image into the data matrix X (P x 3)

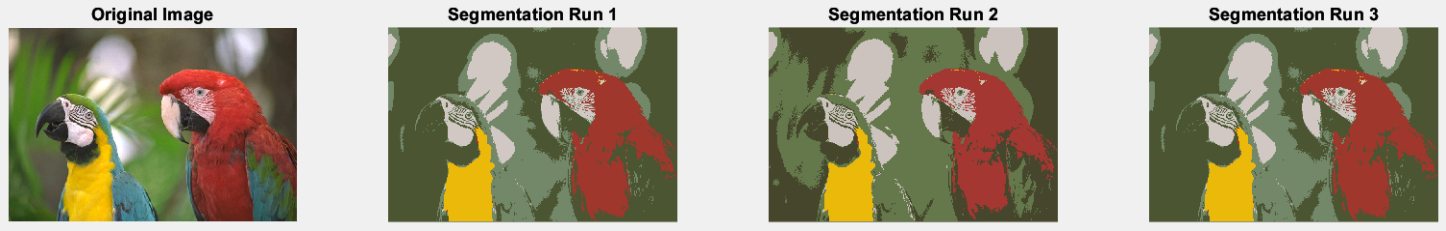

Figure 12: Part b) - K-means Segmentation Results (K = 5)

[Explanation]

- **Comments for the Figure 11: Part a) - Reshape the image into the data matrix X (P x 3):**
  - Let the input color image be denoted by I. The image I has dimensions M x N x 3, where:
    ➔ M is the number of rows (height).
    ➔ N is the number of columns (width).
    ➔ 3 represents the three color channels: Red (R), Green (G), and Blue (B).
  - For K-means clustering, we treat each pixel as a data point, and its color components (R, G, B) as the features. The total number of pixels (data points) is P = M * N.
  - The data matrix X used as input for the K-means algorithm will have dimensions P x 3. Each row of X corresponds to a single pixel, and the three columns correspond to the R, G, and B values of that pixel, respectively.
- **Comments for the Figure 12: Part b) - K-means Segmentation Results (K = 5):**
  - Observe the segmentation results from the 3 runs displayed in the figure.
  - You may notice that the segmented images might look slightly different across the runs.
  - This variation is due to the random nature of the initial centroid selection in the standard K-means algorithm. Different starting points can lead the algorithm to converge to different local optima, resulting in slightly different cluster assignments and boundaries.
- **Part c): What should we do if we want to obtain *the same segmentation result* for several runs (with the same input and the same value of K)?**

- The K-means algorithm typically starts by randomly selecting initial cluster centroids (or randomly sampling data points to be initial centroids). Because this starting point is random, running the algorithm multiple times on the same data with the same K can lead to different final cluster assignments and centroid locations, especially if the clusters are not perfectly distinct or if the algorithm converges to a local minimum instead of the global minimum.
- To obtain the same segmentation result for several runs with the same input image and the same value of K, you need to control the initialization process:
  - ➔ **Fix the Random Number Generator Seed:** Before calling the kmeans function, set the seed for MATLAB's random number generator. This ensures that the same sequence of "random" numbers is used for initialization in each run.
  - ➔ **Provide Explicit Starting Centroids:** You can specify the initial centroid locations directly using the 'Start' parameter in the kmeans function. If you provide the *exact same* initial centroids for each run, the algorithm will start from the same point and, assuming deterministic behavior otherwise, should converge to the same result.
  - ➔ **Increase Replicates (Recommended Practice):** While not guaranteeing *identical* results on every single independent script execution *unless combined with seeding*, using the 'Replicates' option in kmeans is the standard way to get the *best* result out of multiple runs *within a single call*. kmeans runs the algorithm multiple times (default is 1) with different random initial centroids and returns the solution with the lowest total sum of within-cluster distances. This increases the likelihood of finding a better (potentially the global) optimum, making the result more stable and often better quality. To make *this* best result reproducible, you still need to set the random seed (rng) before calling kmeans with replicates.
- Summary: The most direct way to ensure identical results across separate executions of the script is to **set the random number generator seed using rng()** before calling kmeans in each execution.