MATLAB SELF-STUDY

A good logbook should include explanations, clarifications, what was learnt, and often even some reflections.

LAB1

Explanation and clarification:

Rotating function in matlab:

```
function [Out] = Rotate(In, Theta)
```

```
  [rows, cols] = size(In);
```

This line retrieves the height and width of the input matrix to define the processing area. It is essential to use dynamic variables rather than hardcoding values like 200 or 320 to ensure the function works with any image size.

```
  Out = zeros(rows, cols);
```

This creates an empty black canvas with the same dimensions as the original image to store the rotated result. Pre-allocating the matrix is significantly more computationally efficient in MATLAB than growing the array dynamically inside a loop.

```
  centerX = cols / 2;
```

```
  centerY = rows / 2;
```

Explanation: These variables identify the geometric middle of the image to serve as the rotation axis. Without this step, the image would rotate around the top left corner at (1,1) instead of spinning in place.

```
  % 3. 开始遍历目标图像的每一个像素
```

```
  for y = 1:rows
```

```
    for x = 1:cols
```

This nested structure visits every single pixel coordinate in the target image one by one. We iterate through the destination pixels instead of the source pixels to ensure that every part of the new image is checked for data.

```
      x_old = (x - centerX) * cos(Theta) + (y - centerY) * sin(Theta) + centerX;
```

```
      y_old = -(x - centerX) * sin(Theta) + (y - centerY) * cos(Theta) + centerY;
```

Explanation: This formula applies the inverse rotation matrix to find where a destination pixel originally came from relative to the center point. Subtracting the center before the math and adding it back after is what allows the image to spin around its middle rather than the origin.

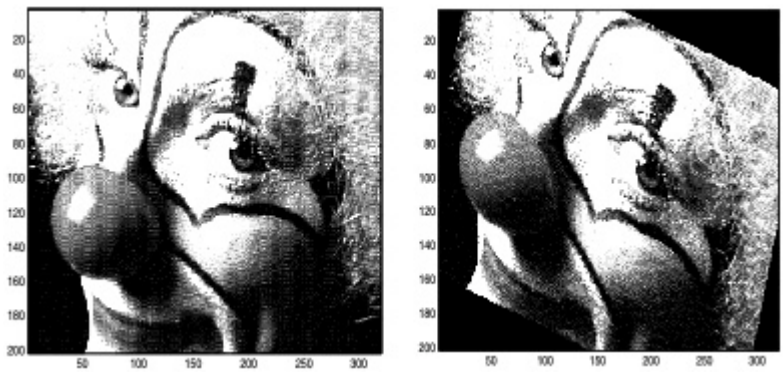
```
        x_source = round(x_old);
        y_source = round(y_old);
```

Explanation: This rounds the calculated decimal coordinate to the nearest whole number to find the closest physical pixel.This step fulfills the 'nearest neighbor' requirement because MATLAB cannot index a matrix using non-integer values.

```
        if (x_source >= 1 && x_source <= cols && y_source >= 1 && y_source <= rows)
```

Explanation: This line copies the brightness value from the identified source pixel to the current destination pixel. Pixels that fail the boundary check remain as 0, which creates the black background seen in the rotated result.

```
            Out(y, x) = In(y_source, x_source);
        end
    end
  end
end
```

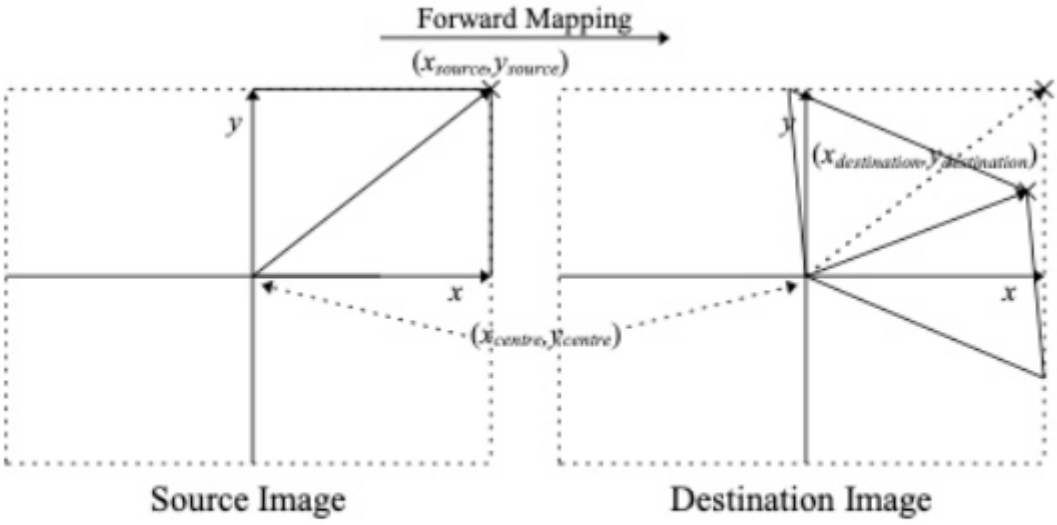


Shearing function in matlab:

Create the Function File

```
function [Out] = Shear(In, Xshear, Yshear)
    Get dimensions of the original image
   [h, w] = size(In);


    Initialize a black image of the same size
```

```
Out = zeros(h, w);


%Define the center of the image (pivot point)
xc = w / 2;
yc = h / 2;
```



Equation 3 - Equation for shearing the image

$$\begin{pmatrix} x_{destination} \\ y_{destination} \end{pmatrix} = \begin{pmatrix} 1 & Xshear \\ Yshear & 1 \end{pmatrix} \left(\begin{pmatrix} x_{source} \\ y_{source} \end{pmatrix} - \begin{pmatrix} x_{centre} \\ y_{centre} \end{pmatrix} \right) + \begin{pmatrix} x_{centre} \\ y_{centre} \end{pmatrix}$$

Define the Transformation Matrix

X-shear slants the image horizontally.

Y-shear slants it vertically.

```
% Define the Forward Shear Matrix
M = [1 Xshear; Yshear 1];


% Calculate the Inverse Matrix for reverse mapping
T_inv = inv(M);
```

The Double Loop

By subtracting `xc` and `yc`, we ensure the middle of the clown's face stays still while the edges slant around it.

The `round()` function is critical. If your calculation says a pixel came from position $(34.4, 46.6)$, MATLAB can't read a "partial" pixel. `round()` tells it to just grab the pixel at (34, 47).

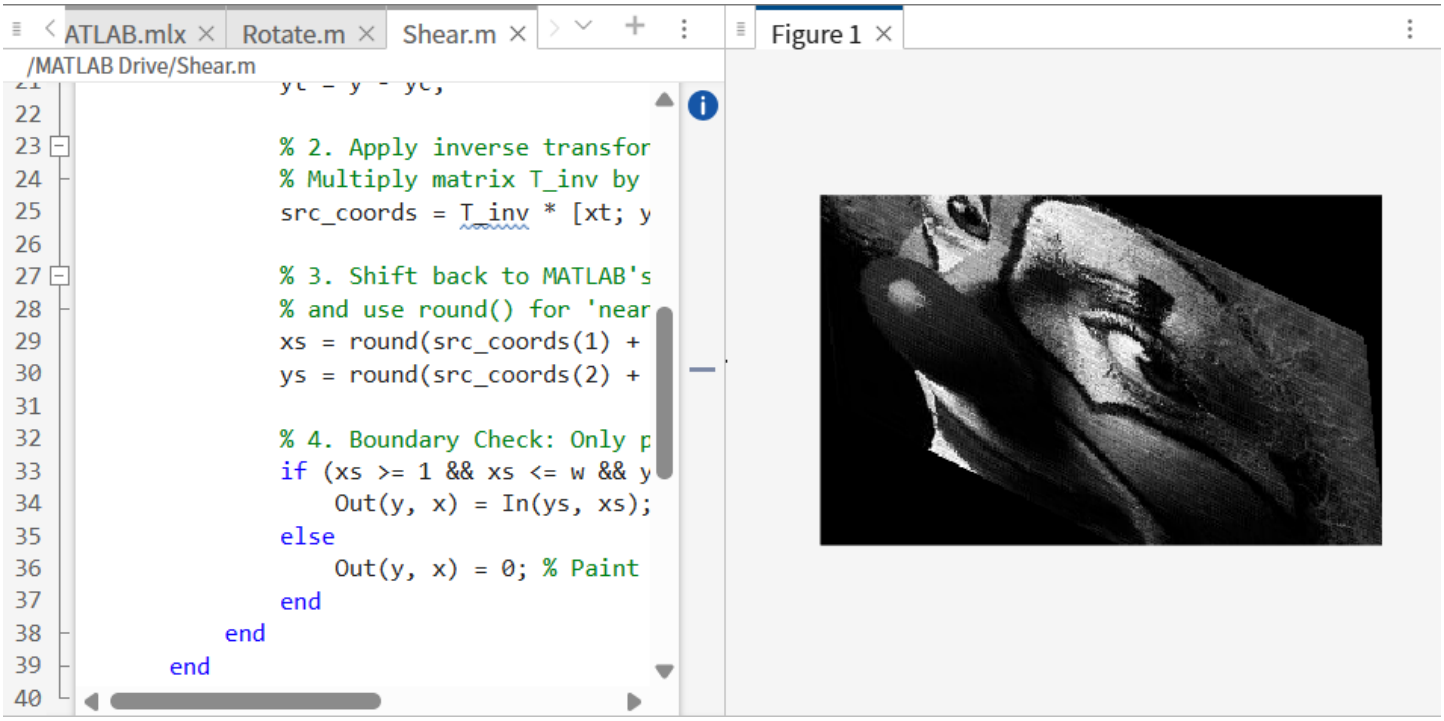
Matrix Order: In the calculation `T_inv * [xt; yt]`, ensure `T_inv` is on the left.

```
for y = 1:h
    for x = 1:w
        % 1. Shift coordinate so the center is (0,0)
        xt = x - xc;
        yt = y - yc;

        % 2. Apply inverse transform to find source coordinates
        % Multiply matrix T_inv by the coordinate vector [xt; yt]
        src_coords = T_inv * [xt; yt];

        % 3. Shift back to MATLAB's 1-based pixel coordinates
        % and use round() for 'nearest pixel' as required
        xs = round(src_coords(1) + xc);
        ys = round(src_coords(2) + yc);

        % 4. Boundary Check: Only paint if the source pixel exists
        if (xs >= 1 && xs <= w && ys >= 1 && ys <= h)
            Out(y, x) = In(ys, xs);
        else
            Out(y, x) = 0; % Paint black if outside the original image
        end
    end
  end
end
```

Run and Test

Command window:

```
% Apply 0.1 X-shear and 0.5 Y-shear as per tutorial example
result = Shear(clown, 0.1, 0.5);

% Display it
imshow(result)
```



What was learnt:

Term	Translation	Definition/Context
Grayscale Image	灰度图像	An image where each pixel represents intensity (0 for black, 1 for white).
Matrix / Array	矩阵 / 数组	In MATLAB, an image is stored as a 2D matrix of numbers.
Pixel	像素	The smallest unit of a digital image.
Transformation	变换	Mathematical operations like rotation, scaling, or shearing.
Forward Mapping	前向映射	Moving pixels from the source to the destination (causes "holes").
Inverse Mapping	反向映射	The standard method: finding where a destination pixel "came from" in the source.
Nearest Neighbor	最近邻插值	Using the closest integer coordinate (e.g., round(34.43) to 34).
Shearing	剪切	A transformation that slants the image along the x or y axis.

☒ **load clown**: Loads the built-in dataset into the **Workspace**.

☒ **imshow(A)**: Displays the matrix A as an image.

☒ **size(In)**: Returns the dimensions (height and width) of the image.

☒ **zeros(h, w)**: Creates an empty black image of a specific size.

☒ **inv(M)**: Calculates the **Inverse Matrix**, essential for reverse mapping.

☒ **round(x)**: Rounds to the nearest integer, used for **Nearest Neighbor** selection.

1. The Hole Problem

> "Direct forward mapping results in multiple pixels mapping to the same location, or some pixels never being written, creating empty gaps."
>
> **Solution:** Use the **Inverse Transformation Matrix** to loop through the *destination* image instead.

2. Centering the Image

Since we want to rotate/shear around the **center**, we must shift the coordinates:

1. **Shift to Origin:** $x_{new} = x - x_{center}$
2. **Transform:** Apply the matrix multiplication.
3. **Shift Back:** $x_{final} = x_{new} + x_{center}$ (so MATLAB can index the pixel at 1, 2, 3...).

3. Boundary Checking

> "If the calculated source pixel lies outside the image boundaries, it should be painted black."
>
> **Logic:** `if (xs >= 1 && xs <= width) ... else Out = 0;`

refelction

A key moment of reflection occurred when I noticed my pixel value at $(20, 319)$ was 0.1686, while the tutorial expected 0.1554. Initially, I viewed this as a mistake, but upon further investigation, I realized it was a lesson in data formats—my use of a compressed `.jpg` file introduced artifacts that changed the numerical reality of the image compared to the raw `.mat` file. In design engineering, this highlights the importance of understanding the provenance of our data; even a small compression can alter the "truth" of an image matrix

Understanding Command Window Syntax

I initially misunderstood the relationship between commands and outputs by attempting to manually type `ans = 0.1554` into the command window. This resulted in an invalid expression error because I was trying to assign a value to a system protected variable instead of

executing a query. I now understand that `ans` is a reactive output from MATLAB and that I should only input the specific coordinate query like `clown(20, 319)` to retrieve data.

The Hole Problem through Logic

Through the implementation of the `Rotate` and `Shear` functions, I experienced firsthand why **Forward Mapping** is insufficient. My initial thought was to move pixels from the source to the target, but the tutorial clarified that this leaves empty gaps or holes in the output. Switching to **Inverse Mapping** allowed me to ensure every pixel in the output image was assigned a value by looking back at the source image, which is a much more robust approach for digital transformations.

Integer Constraints in Matrix Indexing

I learned that while mathematical formulas for rotation and shearing result in decimal coordinates, MATLAB matrices require integer indices. Forgetting to use the `round()` function initially would cause the program to fail because there is no such thing as a partial pixel at index 34.4.

LAB2

Task 1: Find your blind spot

Anatomy of the Blind Spot

The biological reason we have a blind spot lies in the connection between the eye and the brain:

- **The Retina:** The back of the eye is covered with **photoreceptors**—specialized cells that capture light and convert it into signals.
- **The Optic Nerve:** Think of this as a **bundle of wires**. It acts like a massive data cable that carries all those visual signals from the retina to the brain for processing.
- **The Connection Point (Optic Disc):** There is a specific spot where this "bundle of wires" must pass through the retina to exit the eyeball. This exit point is called the **optic disc**.
- **The Result:** Because the optic disc is occupied by the exit of the optic nerve, there is literally no room for any **photoreceptors** in that area.
- **The Blind Spot:** Without photoreceptors to catch light, that specific spot on your retina is functionally blind—it cannot detect any visual information from the world.

Brain Interpolation

When visual signals are missing due to the blind spot, the brain doesn't leave a "black hole." Instead, it performs **interpolation** (calculating and predicting missing data) based on the surrounding **background** to create a seamless image.

- **Solid Color Filling:** The brain identifies the surrounding hue and saturates the blind spot with the same color (e.g., the red background in the video).
- **Geometric Completion (Lines):** If a line or a crosshair is interrupted by the blind spot, the brain automatically "draws" the missing segment to connect the paths.
- **Texture & Pattern Simulation:** The brain can mimic complex textures (like the **Zebra Pattern**) by recognizing the rhythm and flow of the stripes, ensuring visual **continuity**.

Task 2: Ishihara Colour Test

ALL correct

Inside your retina, you have specialized cells called **Photoreceptors**. There are two main types：

Rods (for low light/black and white)

Cones (for color and detail)：

Humans typically have **three** types of cones, named after the wavelengths of light they are **best at absorbing**:

L-Cones (Long Wavelength) 🔴

- **Primary Color:** These are often called "red cones."
- **Function:** They are most sensitive to long wavelengths of light, which our brain perceives as the red end of the spectrum.

M-Cones (Medium Wavelength) 🟢

- **Primary Color:** These are known as "green cones."
- **Function:** They respond best to medium wavelengths, which correspond to green light.

S-Cones (Short Wavelength) 🔵

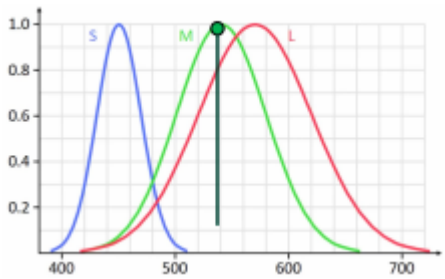
How they work together

It's a common misconception that one cone "sees" one color. In reality, light usually stimulates multiple cone types at once. Your brain compares the signals:

- If your **L-cones** are firing much more than your **M-cones**, your brain says, "That's red!"
- If both are firing equally, you might perceive **Yellow** 🟡

The **Ishihara Test**

(those plates with the dots) is specifically designed to see if your L and M cones are working correctly. In the most common type of color blindness, the sensitivity of the L and M cones overlaps too much, making it impossible for the brain to tell the difference between red and green signals.



question:

To see if this is clear so far: if a person's M-cones (green) weren't working at all, which part of the Ishihara Test do you think they would struggle with the most?

The Initial Misconception (My First Thought) ❌

- **Thought:** If M-cones (Green) don't work, all green objects will simply be perceived as **Red**.

The Core Mechanism: Signal Comparison

- **Brain as a Processor:** The brain doesn't determine color based on a single incoming signal. Instead, it relies on **comparing** signals between different cone types to find the strongest or dominant one.
- **The Red-Green Failure:** Without functional **M-cones**, the brain loses its ability to perform the **Red vs. Green** comparison. It can no longer distinguish between the wavelengths that would normally activate these two separate channels.

The Logic of Exclusion (Why Yellow?)

- **The Blue-Yellow Axis:** While the Red-Green channel is broken, the **S-cones (Blue)** are still functioning. This allows for a different comparison.
- **Process of Elimination:**
  a. The brain detects light but notices the **S-cones** are not strongly activated $\rightarrow$ **"It's not Blue."**
  b. The brain looks at the remaining spectrum (Long wavelengths) where Red and Green used to live.
  c. Because it can no longer tell Red from Green, it collapses these signals into the **complementary color** of blue.

- **Final Perception:** The brain categorizes these ambiguous signals as **"Some degree of Yellow"** (Ochre, Tan, or Brown).

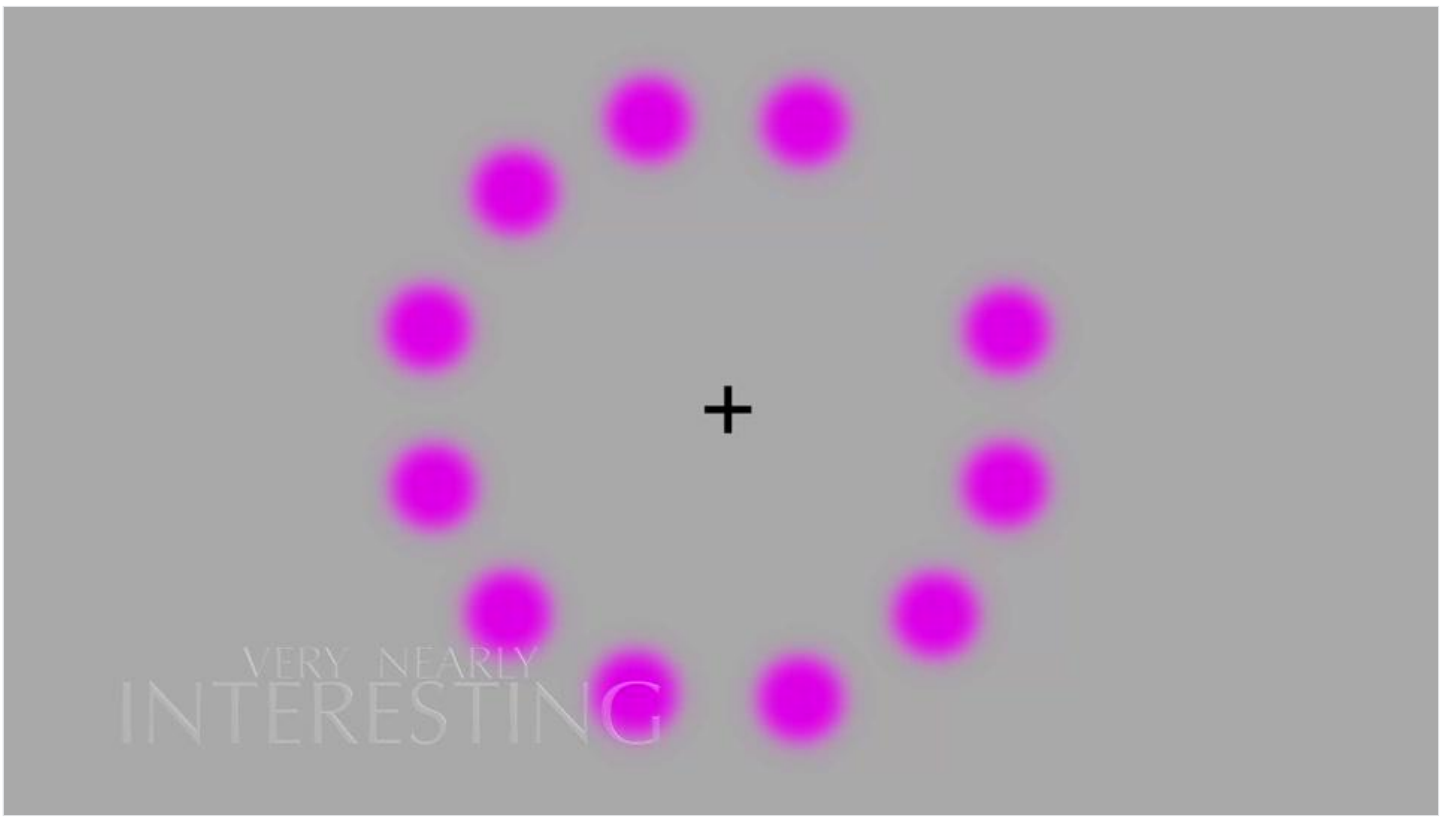
Task 3 - Reverse colour

Inside eyes, you have specialized cells called **cones** that detect color. There are three types, each sensitive to different wavelengths: red, green, and blue. When staring at a specific color (like the bright green in the image) for a long time, **the cones sensitive to that color get tired or fatigued**. They stop sending a strong signal to your brain.

The visual system works in pairs of opposing colors. When one color in a pair is suppressed (due to fatigue), brain automatically perceives its **complementary** (opposite) color to restore balance.

Original Color in Image	Complementary Color (Afterimage)
Green 🟢	Red 🔴
Yellow 🟡	Blue 🔵
Black ⚫	White ⚪

Task 4 - Troxler's Fading



Neural Adaptation:the nervous system is hardwired to prioritize **change**. When a stimulus—like those blurry purple dots—remains constant and unchanging, your brain begins to ignore it

as redundant information. It is similar to how we stop noticing the scent of a room after being inside for a few minutes.

Peripheral Vision Limitations: While we are focused on the central cross, the purple dots are processed by our **peripheral vision**. This area of our sight is much less sensitive to detail and static objects than our central vision, making it more susceptible to fading.

1. Afterimage & Opponent-Process Theory

We observed that staring at a specific color (e.g., green) causes **cone fatigue**. When the signal of one color is suppressed, our brain automatically perceives its **complementary color** (e.g., red) to restore balance.

2. Troxler's Fading & Neural Adaptation

We discovered that unchanging stimuli in our **peripheral vision** tend to disappear. This is because our nervous system prioritizes **change**. When a stimulus remains constant, the brain ignores it as redundant background noise through **neural adaptation**.

3. Fovea vs. Peripheral Vision

We identified a key distinction in retinal topography:

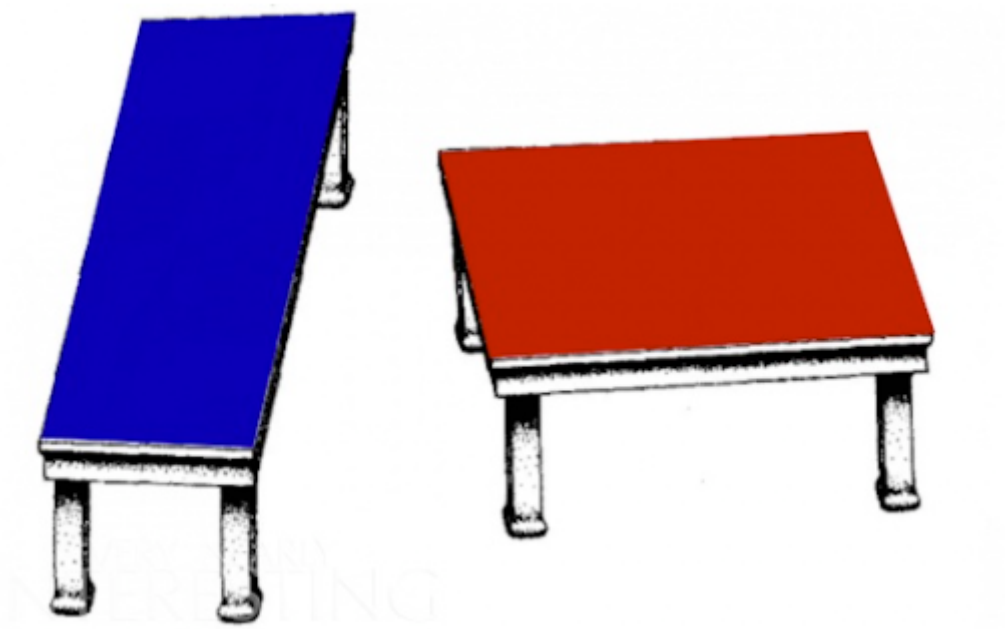
· **Fovea**: 【The area where the eyes focus 】The central focus area where **cones** are highly concentrated. **Microsaccades** (fixational eye movements) constantly refresh the signal here, preventing the central point (e.g., a red dot) from fading.

· **Peripheral Vision**: 【The area of the peripheral vision of the eyes】Dominated by **rods**, this area has lower resolution and is less sensitive to the "refreshing" effect of microsaccades, making static, blurry objects prone to fading.

4. Rods vs. Cones (The Sketch)

· **Cones**: Like colored pencils for fine detail and color in bright light (**photopic vision**).
· **Rods**: Like charcoal sticks for "big relationships" (shading, light/dark contrast, and motion) in dim light (**scotopic vision**). They are highly sensitive but lack color and fine detail.

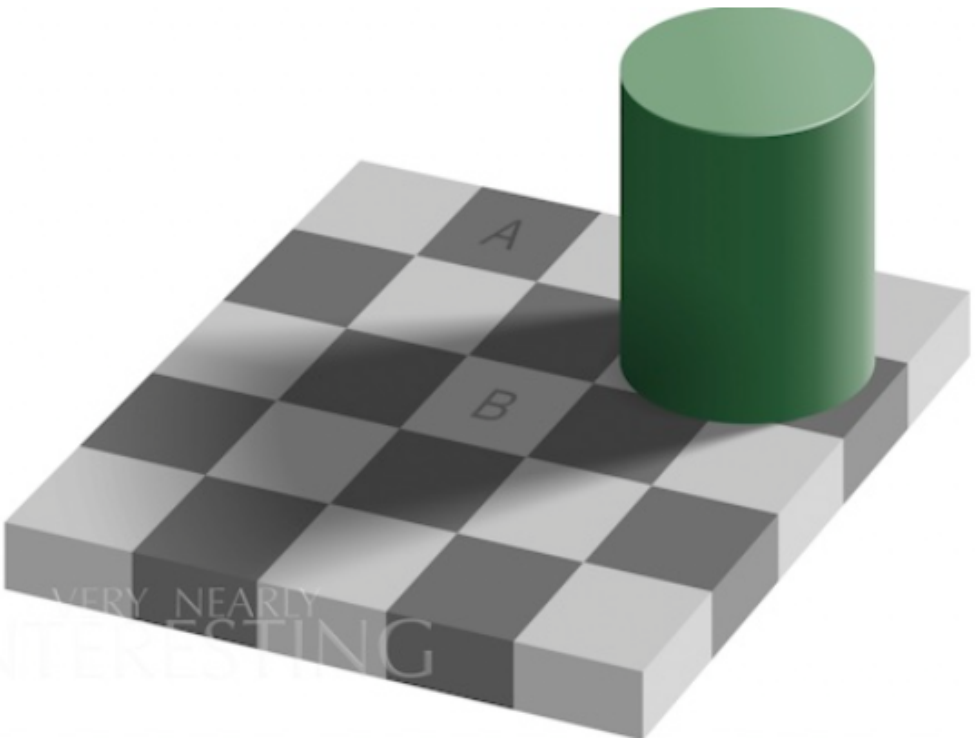
Task 5 - Brain sees what it expects



1. The Bench Illusion (Perspective Inference)

We questioned why two identical shapes on a 2D surface look so different in length.

- **The Brain as a 3D Modeler**: We found that our brain does not just see flat lines; it automatically reconstructs a 3D model of the scene.
- **Foreshortening Compensation**: We realized that in the real world, objects naturally "shrink" on our retina as they recede into the distance (the "near-large, far-small" rule).
- **The Internal Calculation**: Because the blue bench is perceived as having **depth (Z-axis)**, our brain assumes it must be much longer in reality to appear that size on a flat screen. It "stretches" our perception to match this spatial logic.



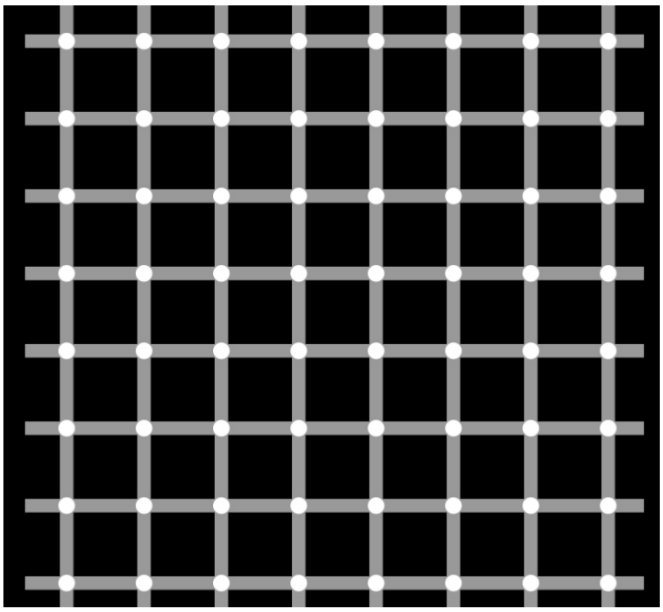
2. The Checkerboard Illusion (Shadow Compensation)

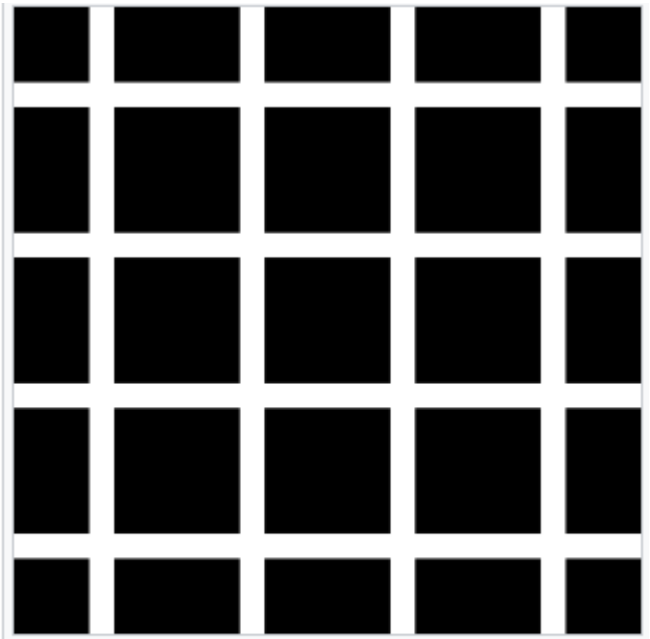
We explored why square A (in the light) looks darker than square B (in the shadow), even though they are the same shade of gray.

- **Identifying the Context**: We observed that our brain first identifies the green cylinder and the shadow it casts.
- **The "Polish" Mechanism**: We concluded that the brain performs an automatic "correction." It thinks: If this square is in a shadow and still reflects this much light, it must be a **light-colored square** in its original state.
- **Prioritizing Reality over Pixels**: To help us recognize the "true" pattern of the checkerboard, our brain "polishes" or brightens Square B in our mind, overriding the actual raw light data hitting our eyes.

Task 6 - The Grid Illusion

When you stare at the centre of the grid below, you should see black dots at the intersection appearing and disappearing. You can read more about it here.





An example of the Hermann grid illusion. Dark blobs appear at the intersections.

1. Lateral Inhibition (The Main Cause)

We see these ghost-like black dots because of how the nerve cells in our retina interact. This process is called **lateral inhibition**.

- **The "Bright" Signal**: When our eyes look at a white intersection, that area is surrounded by more "white" (from the four paths) than a point in the middle of a single white line.
- **The "Inhibition"**: The nerve cells detecting the bright white paths send signals to "turn down" or inhibit the cells next to them.
- **The Result**: Because the intersections have the most white around them, they receive the strongest "turn down" signals. This makes our brain think the intersection is less bright than it actually is, creating a dark spot.

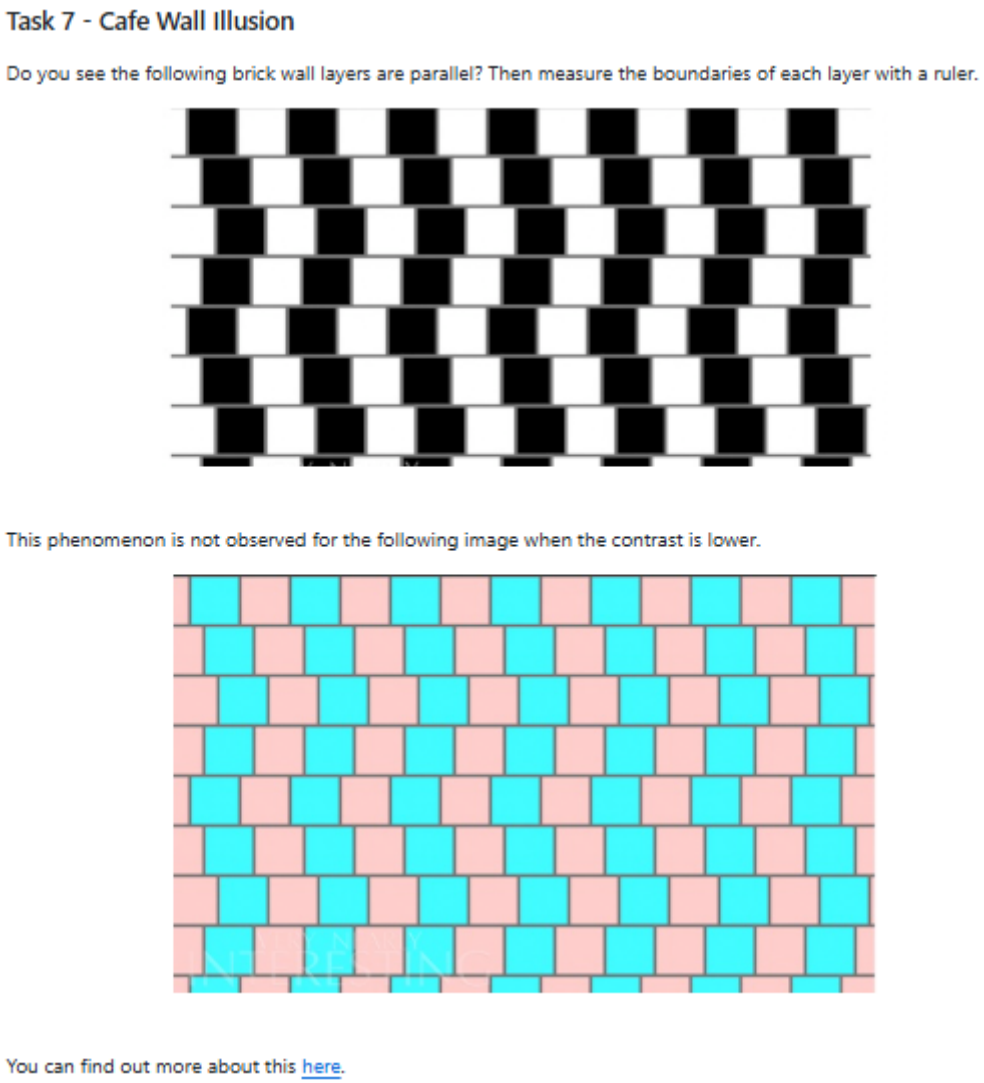
2. Peripheral vs. Central Vision

We noticed that the dots only appear in our **peripheral vision** and disappear when we look directly at them.

- **The Fovea (High Detail)**: When we stare directly at an intersection, the image falls on our **fovea**. Here, the nerve cells are packed so tightly and have such small "receptive fields" that the lateral inhibition isn't strong enough to trick us.
- **The Edge (Low Detail)**: In our peripheral vision, the receptive fields are much larger. The brain "averages out" the light and dark areas over a bigger space, making it much easier for the lateral inhibition to create the illusion of a black dot.

Brain Over-Correction: Just like the "polish" effect in the shadow illusion, we see that the brain sometimes over-corrects for brightness, leading us to see shadows (dots) that don't exist in the physical image.

Task 7 - Cafe Wall Illusion



1. The Visual Paradox

We observed that the horizontal lines between rows of staggered black and white bricks appear to be tilted or wedge-shaped. However, when we measure them with a ruler, we find that the lines are **perfectly horizontal and parallel**.

2. The Role of High Contrast

We identified that this illusion is strongest when there is high contrast (black and white) separated by a thin, neutral "mortar" line.

- **Neural Conflict**: The high contrast between the bricks tricks the neurons in our visual system into misinterpreting the position of the horizontal lines.

- **Low Contrast Effect**: We noticed that when the contrast is lowered (as seen in the pink and blue example), the brain can process the boundaries more accurately, and the illusion of tilting disappears.

3. Edge Detection Errors

We discovered that this happens because of how our brain detects **edges and boundaries**.

- **Staggered Alignment**: Because the bricks are offset (staggered), the brain struggles to integrate the local signals of light and dark along the horizontal gray line.
- **Brain's "Wrong" Calculation**: Instead of seeing a straight line, the brain interprets the varying contrast along the border as a sign of a slope, creating the "wedge" effect.

Summary of Our Insights

- **Parallel Reality**: We confirmed that our perception of "straightness" can be easily distorted by surrounding patterns.
- **The Importance of Contrast**: We concluded that contrast levels are a primary driver for how the brain calculates angles and alignment.
- **Local vs. Global Processing**: This is another example where the brain's attempt to interpret local "clues" (the bricks) results in a global error (the tilted lines).

Task 8 - the Silhouette Illusion

1. The Perceptual Conflict

We observed what appears to be a series of overlapping arcs or a **spiral** leading toward the center. However, by tracing the lines, we discovered that the image is actually composed of **concentric circles**.

2. Directional Cues & Orientation

We found that the illusion is created by the "vines"—the small, tilted segments or patterns within the circles.

- **Misleading Slant**: Each individual segment is tilted at an angle relative to the actual path of the circle.
- **Brain's Integration Error**: Our brain integrates these local directional cues and interprets them as a continuous inward slope, creating the spiral effect.

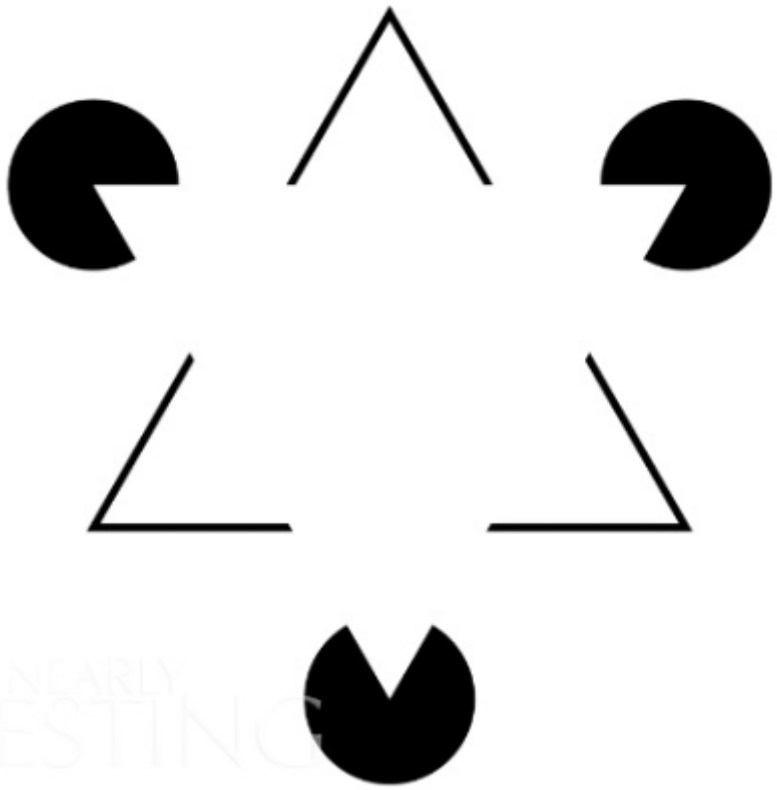
3. Background Interaction

We noticed that the complex, contrasting background pattern further confuses our **edge detection** mechanisms. The brain prioritizes the "flow" of the small patterns over the global geometry of the circles.

Summary

- We learned that the brain can be forced to see a "spiral" if the local segments of a circle are consistently tilted inward.
- We concluded that repetitive, directional patterns can override our perception of basic geometric shapes like circles.
- This reinforces our finding that the brain "reconstructs" what it thinks is a logical path rather than simply recording the physical lines.

Task 9 - the Incomplete Triangles



1. The Perceptual Illusion

We observed what appears to be a solid white triangle pointing downward, overlapping another triangle and three black circles. However, if we look closely at the physical lines, there are **no triangles** and no complete circles—only three "Pac-Man" shapes and three V-shaped angles.

2. Illusory Contours

We found that our brain creates "ghost" edges where none exist.

- **Filling in the Blanks**: Because the "Pac-Man" mouths and the V-shapes are perfectly aligned, the brain assumes there must be an object blocking them.
- **Subjective Brightness**: We might even perceive the central white triangle as being "whiter" or brighter than the surrounding background, even though the white color is uniform across the screen.

3. Gestalt Principle of Closure

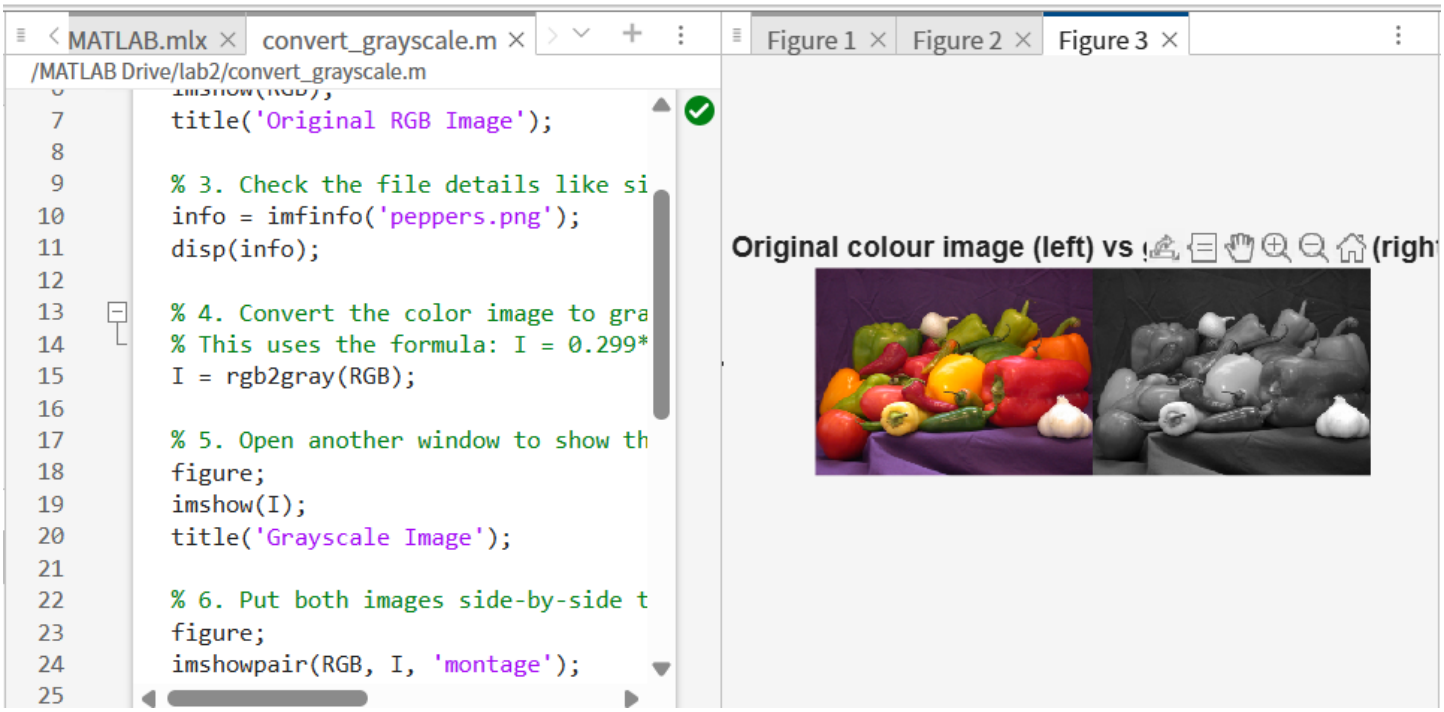
We discovered that the brain has a natural tendency to see complete figures rather than disconnected parts.

- **Creating Order**: Our visual system prefers to see a meaningful shape (a triangle) rather than a random collection of fragmented lines and circles.
- **Predictive Processing**: The brain "predicts" the existence of the triangle to make sense of the gaps in the image.

Summary

- We confirmed that the brain is an active constructor of images, not just a passive receiver of light.
- We learned that when information is missing, the brain uses **context and alignment** to invent edges and shapes to complete the "story."
- We concluded that a few well-placed fragments are enough to trigger a complex visual model in our minds.

Task 10 - Convert RGB image to Grayscale



1. Function code:

```
% 1. Load the original colorful image into MATLAB
RGB = imread('peppers.png');

% 2. Open a new window and show the original picture
figure;
imshow(RGB);
title('Original RGB Image');

% 3. Check the file details like size and format (using imfinfo)
info = imfinfo('peppers.png');
disp(info);

% 4. Convert the color image to grayscale
% This uses the formula: I = 0.299*R + 0.587*G + 0.114*B to calculate brightness
I = rgb2gray(RGB);

% 5. Open another window to show the new gray version
figure;
imshow(I);
```

```
title('Grayscale Image');

% 6. Put both images side-by-side to compare them (using montage mode)
figure;
imshowpair(RGB, I, 'montage');
title('Original colour image (left) vs grayscale image (right)');
```

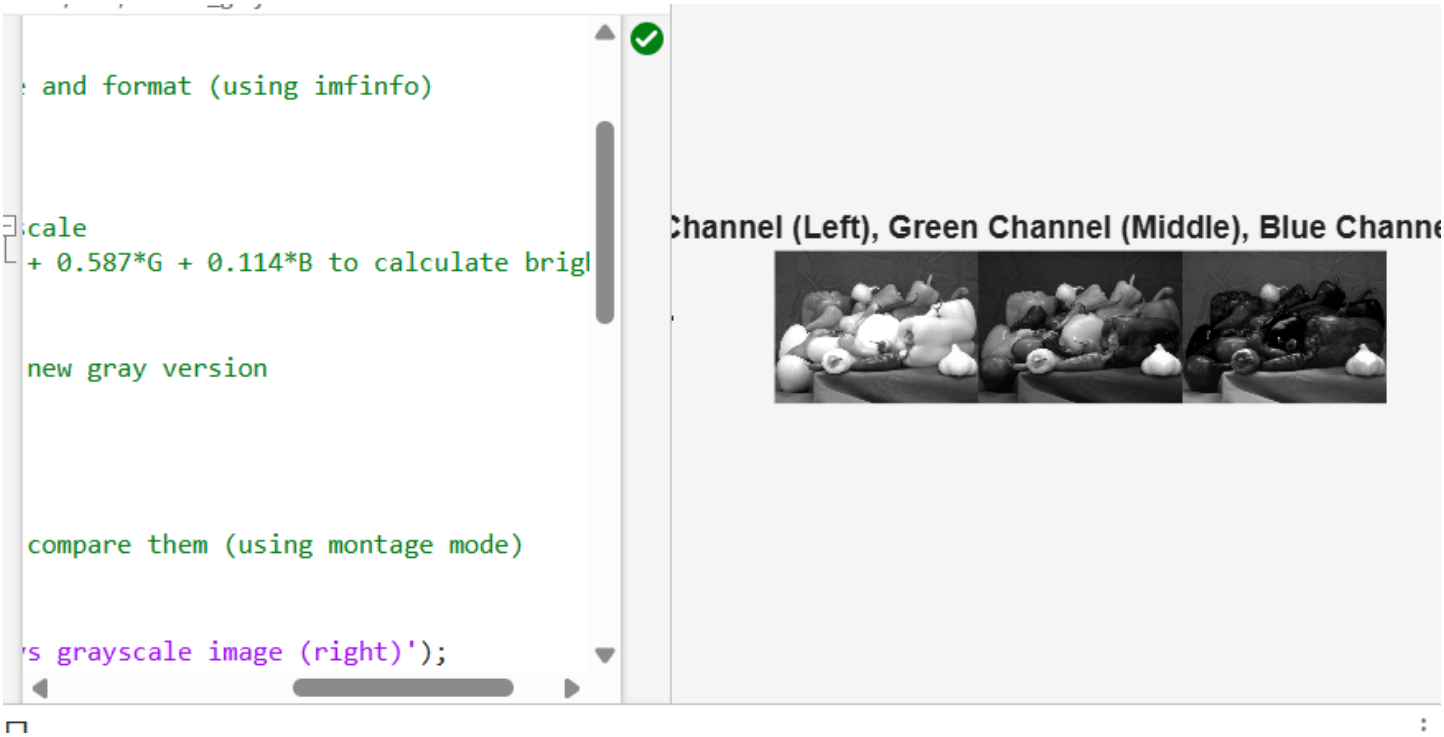
2. Technical Summary (What we did)

- **Image Loading**: We used `imread` to import the standard `peppers.png` file into MATLAB. We found that even on MATLAB Online, the system can automatically find this file in its built-in toolbox.
- **Metadata Analysis**: By using `imfinfo`, we checked the image properties. We learned that a color image is actually a 3D matrix where the third dimension represents the **Red, Green, and Blue (RGB)** channels.
- **Grayscale Transformation**: We applied the `rgb2gray` function. This isn't just a simple average of colors; it uses a specific weighted formula based on human eye sensitivity:
- $I = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$
- **Visual Comparison**: We used `imshowpair` with the `'montage'` parameter to display the color and grayscale images side-by-side for a direct "before and after" look.

3. Reflection (What we learned)

- **Human vs. Computer Vision**: I realized that converting to grayscale is a design choice. We give more weight to the Green channel (58.7%) because our eyes are naturally more sensitive to green light. This ensures the gray version feels right to a human viewer.
- **The "Invisible" Data**: Before this lab, I thought a digital image was just a picture. Now, I see it as a massive set of numbers. When we see gray, the computer is actually doing a fast weighted calculation for every single pixel.
- **Workflow Efficiency**: Using MATLAB Online taught me that managing the Current Folder and understanding the Search Path are essential. If the file and the code aren't in the same mental or physical space, the computer won't find the "ingredients" for the task.

Task 11 - Splitting an RGB image into separate channels



1. Function code

% 1. Use imsplit to break the 3D RGB matrix into three 2D planes

% R, G, and B will each be a grayscale image representing that color's intensity

[R, G, B] = imsplit(RGB);

% 2. Show all three channels side-by-side to compare them

% We use a 1x3 grid (1 row, 3 columns)

figure;

montage({R, G, B}, 'Size', [1 3]);

title('Red Channel (Left), Green Channel (Middle), Blue Channel (Right)');

2. What we see

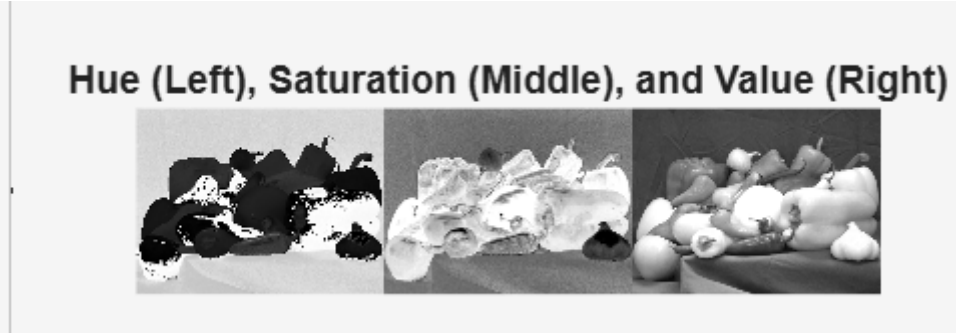
- **Red Peppers**: In the Red channel image, the red peppers look very **bright (white)**. This is because they have a high signal in that channel.
- **Green Peppers**: They look bright in both the Red and Green channels because yellow/green colors are a mix of these two signals.
- **White Garlic**: The garlic looks bright in **all three channels**. This confirms that white light is a strong combination of Red, Green, and Blue.

3. My Insight

- **The Lego of Color**: I realized that any color we see on a screen is just a specific recipe of these three base numbers.

- **Data Structure**: It’s cool to see how `imsplit` changes a 3D object into three separate 2D objects. Understanding the dimensions (like `384x512x3`) helped me visualize how computers actually "store" a memory of a picture.

Task 12 - Map RGB image to HSV space and into separate channels



Function code

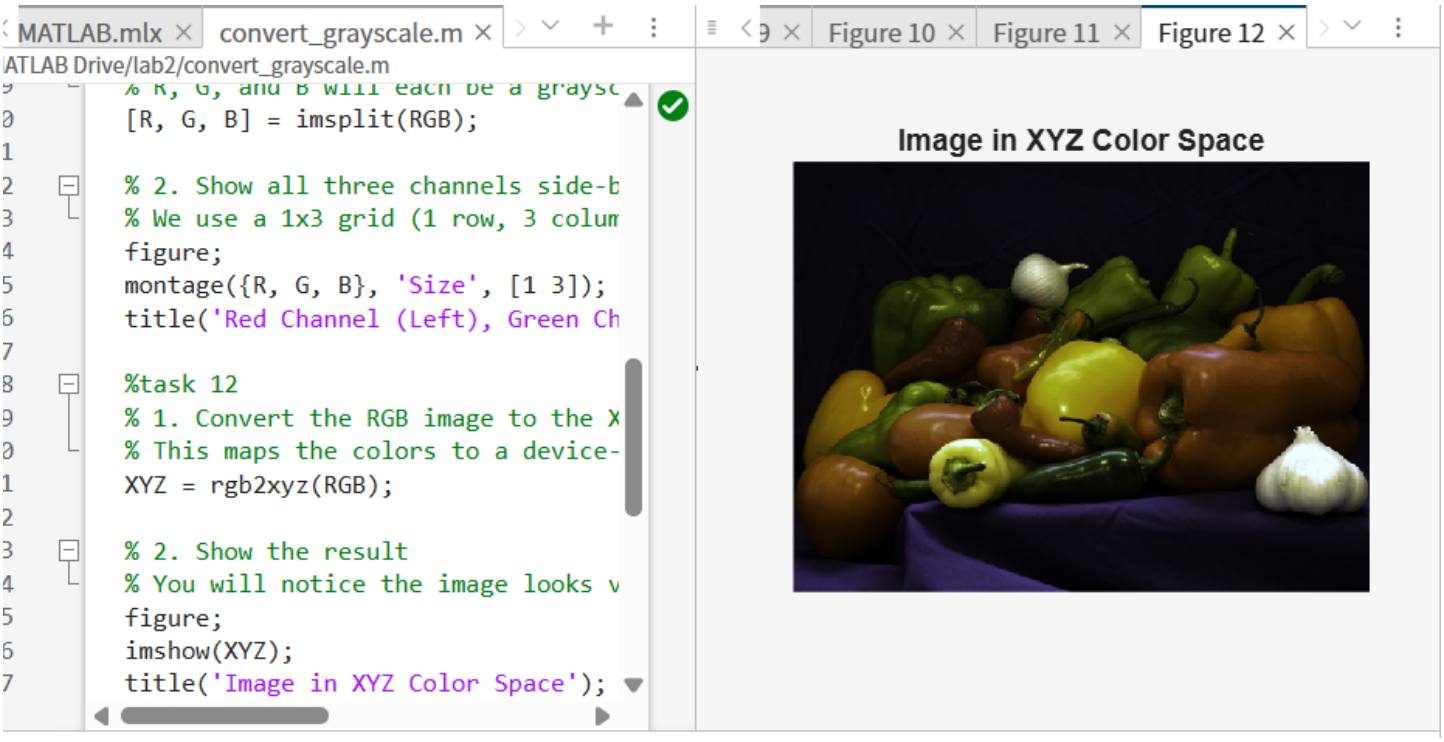
```
% 1. Convert RGB to HSV color space
HSV_img = rgb2hsv(RGB);
% 2. Split the HSV image into its 3 components: Hue, Saturation, and Value
[H, S, V] = imsplit(HSV_img);
% 3. Display them side-by-side to see how they differ
figure;
montage({H, S, V}, 'Size', [1 3]);
title('Hue (Left), Saturation (Middle), and Value (Right)');
```

Hue (H): This channel shows the type of color. I noticed that areas with the same color (like all the red peppers) have similar gray levels here.

Saturation (S): This shows how pure or intense the color is. Brightly colored peppers look very white in this channel, while the dull background looks dark.

Value (V): This is basically the brightness. It looks very similar to the grayscale image we made in Task 10

Task 13 - Map RGB image to XYZ space



Function code

% 1. Convert the RGB image to the XYZ color space

% This maps the colors to a device-independent mathematical space

XYZ = rgb2xyz(RGB);

% 2. Show the result

% You will notice the image looks very strange and "washed out"

figure;

imshow(XYZ);

title('Image in XYZ Color Space');

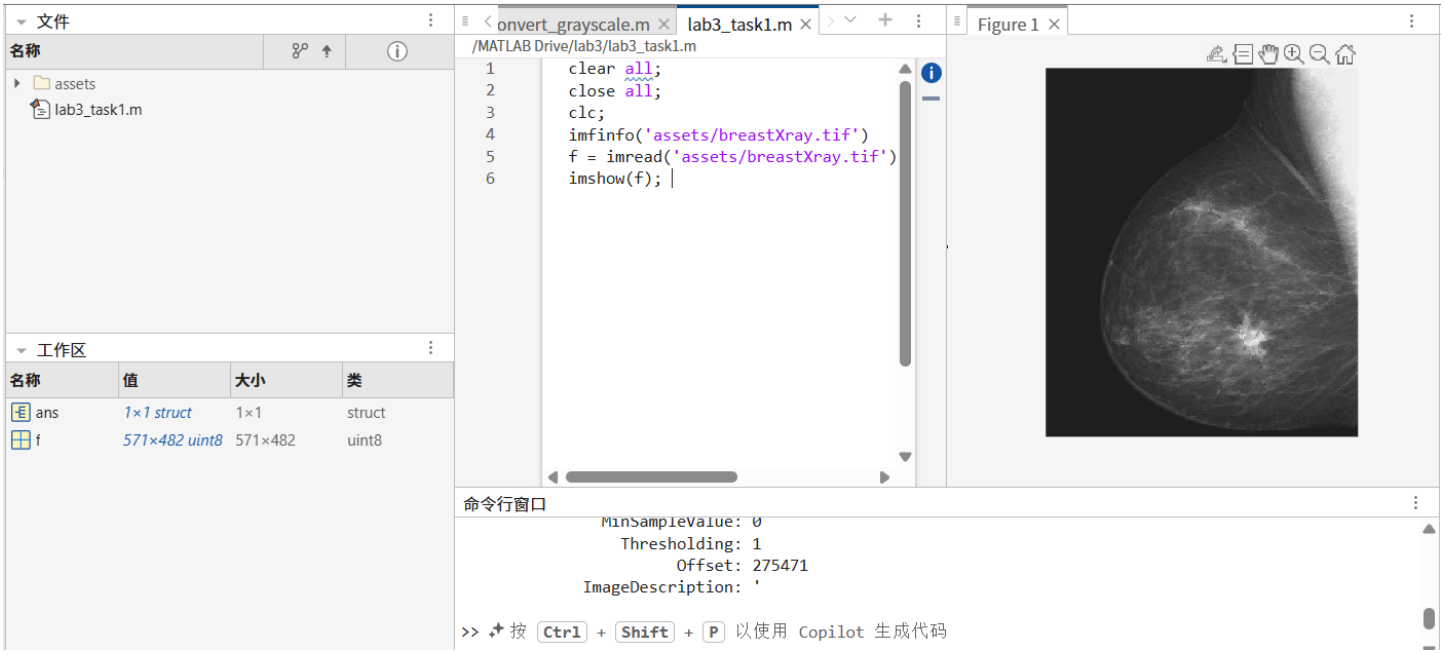
Mathematical Foundation: We used `rgb2xyz` to transform the image into a standard reference space.

Visual Appearance: We observed that the XYZ image looks pale or weird. This is because XYZ is designed for **mathematical calculation**, not for direct human viewing on a standard monitor.

The Master Space: I learned that XYZ is the foundation for almost all other color models because it covers the entire range of colors a human can see.

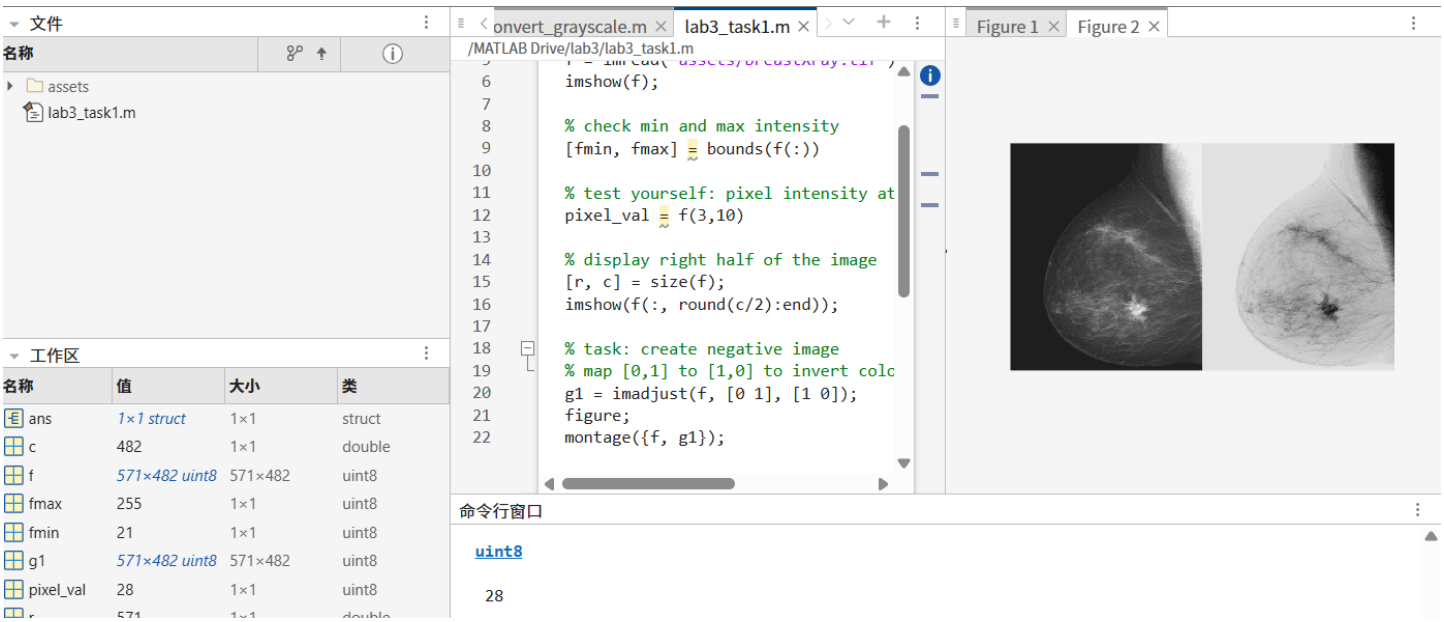
LAB3

Task 1 - Contrast enhancement with function imadjust



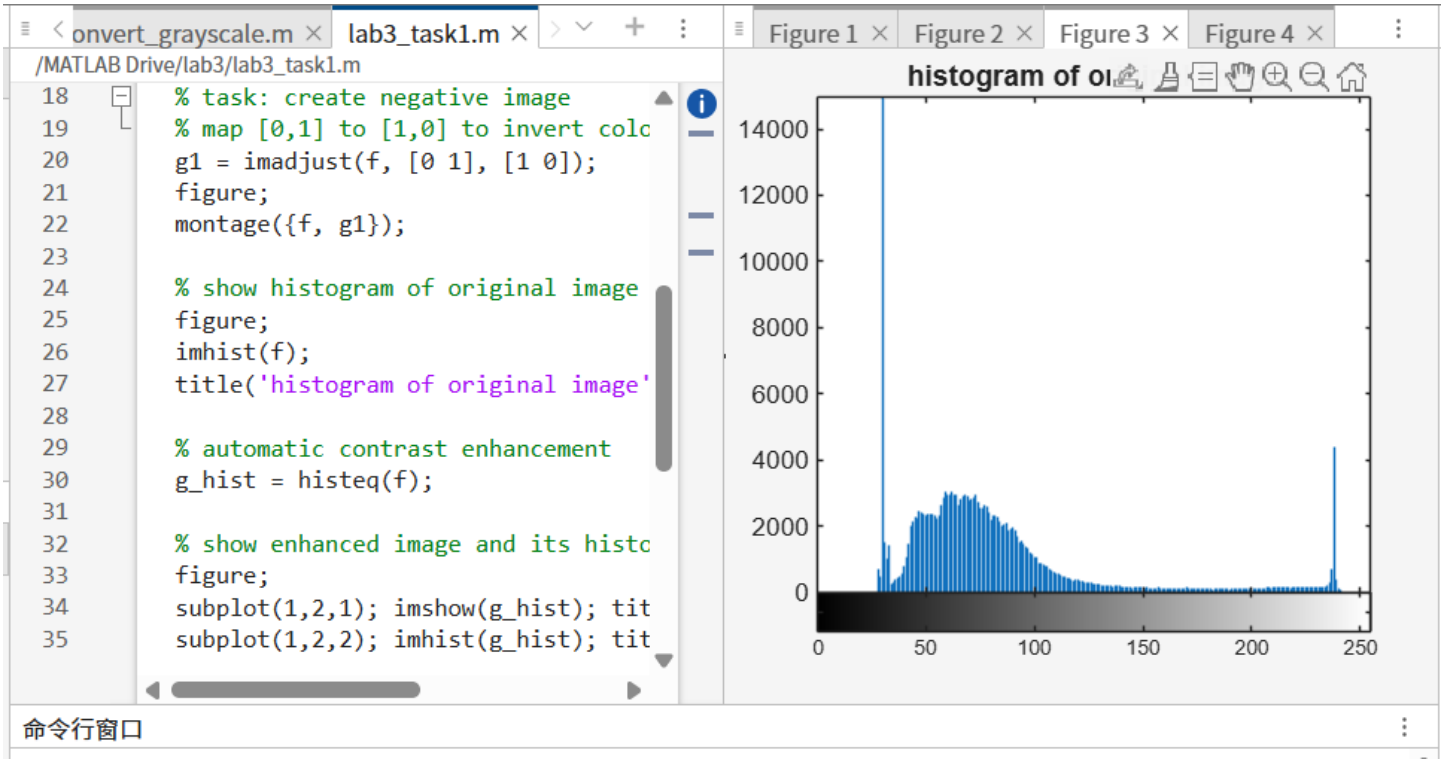
1. Negative Image Insights

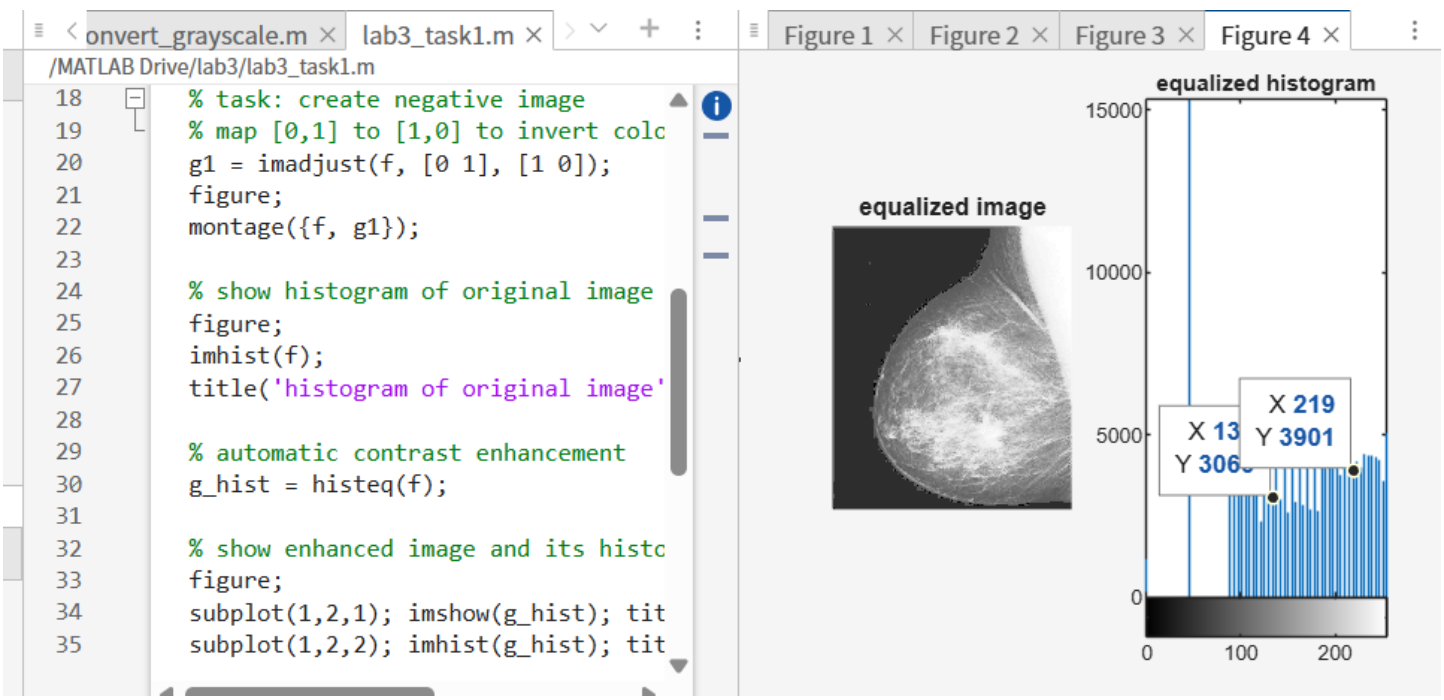
- **Visual Inversion**: This process maps the lightest pixels to the darkest and vice versa, which is why the background turns white and the bone structures turn dark.
- **Feature Detection**: In medical imaging, the negative version (inverse) can sometimes make certain tissue densities or abnormalities easier for the human eye to spot compared to the original.
- **Linear Mapping**: It is a simple linear transformation where s = (L-1) - r. Here, L-1 is 255 for a `uint8` image.



2. Gamma Correction Insights

- **Non-linear Enhancement**: Unlike the negative image, Gamma correction ($s = cr^{\gamma}$) changes the image brightness non-linearly.
- **Contrast Control**:
  - When **gamma > 1** (like your g3 with gamma = 2.0), the mapping curve weighs toward the lower end, making the image darker but stretching the contrast in brighter areas.
  - When **gamma < 1**, the image becomes brighter and reveals more detail in the dark/shadow areas.
- **Human Perception**: Our eyes do not perceive light linearly, so Gamma correction is essential to make digital images look natural or to highlight specific details in medical scans without losing too much data.

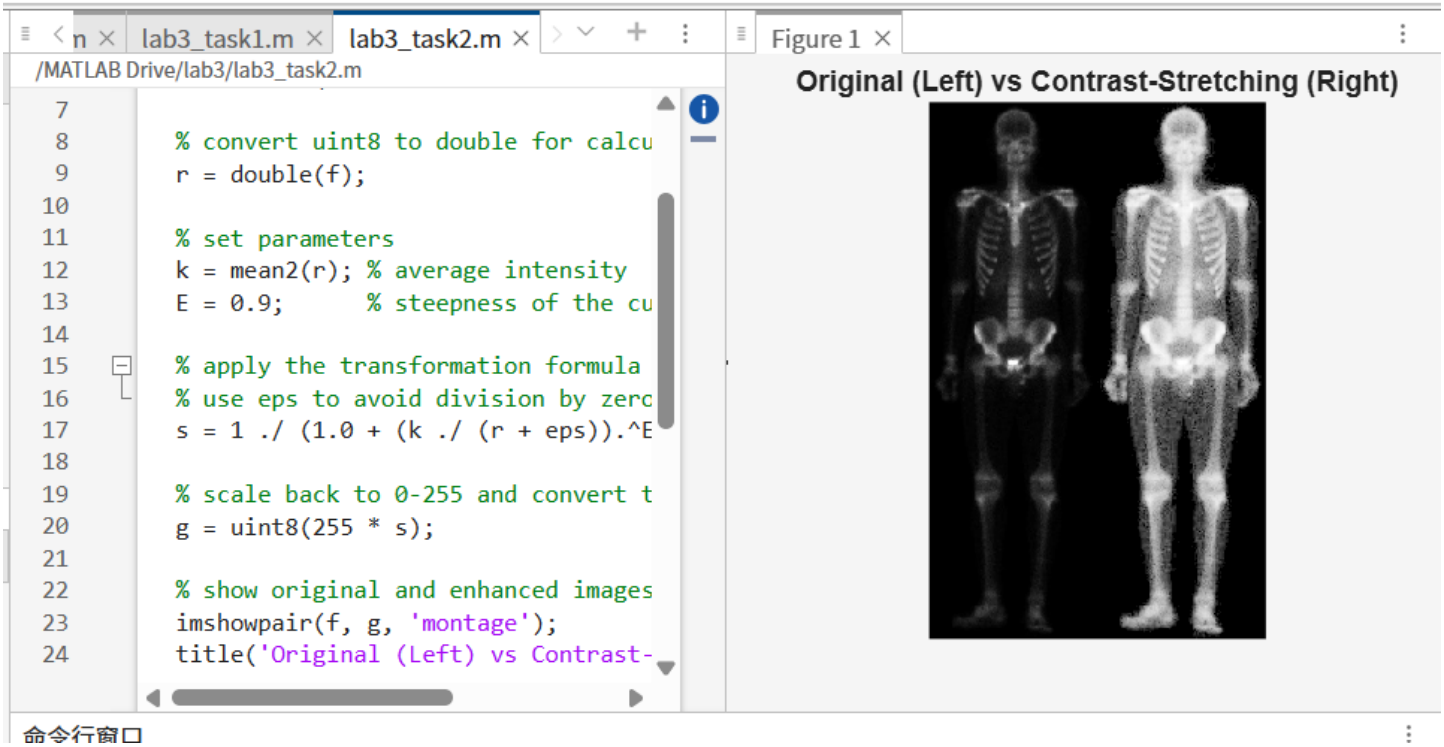




3. Histogram Equalization

- **Automatic Contrast**: Instead of manually guessing the brightness levels, this function automatically finds the best way to stretch the contrast for the whole image.
- **Pixel Redistribution**: It takes the crowded pixels from the dark areas and spreads them out evenly across all possible gray levels (from 0 to 255).
- **Revealing Hidden Info**: In the original X-ray, many details were hidden in the shadows because their brightness values were too similar. After equalization, these tiny differences become much larger and visible to the eye.
- **Flattening the Histogram**: If you look at the new histogram, the peaks are gone and the bars are more spread out. A "flat" histogram usually means the image is using its full range of contrast.
- **Limitations**: While it makes the tissue clearer, it can also amplify background noise. If the image looks too grainy, it is because the equalization stretched the noise along with the actual data.

Task 2: Contrast-stretching transformation



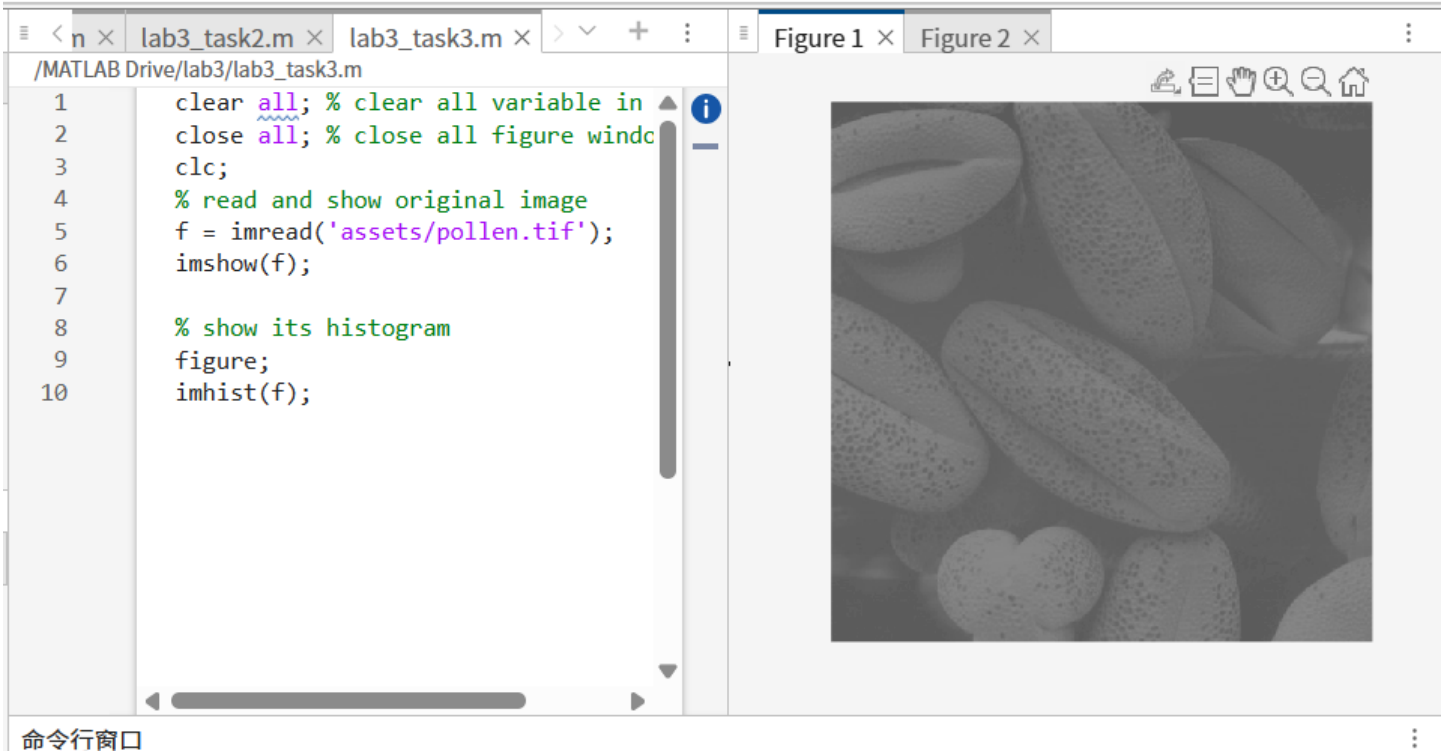
Manual Control: Unlike `imadjust`, this formula lets us use the mean intensity (k) as a pivot point to stretch the contrast.

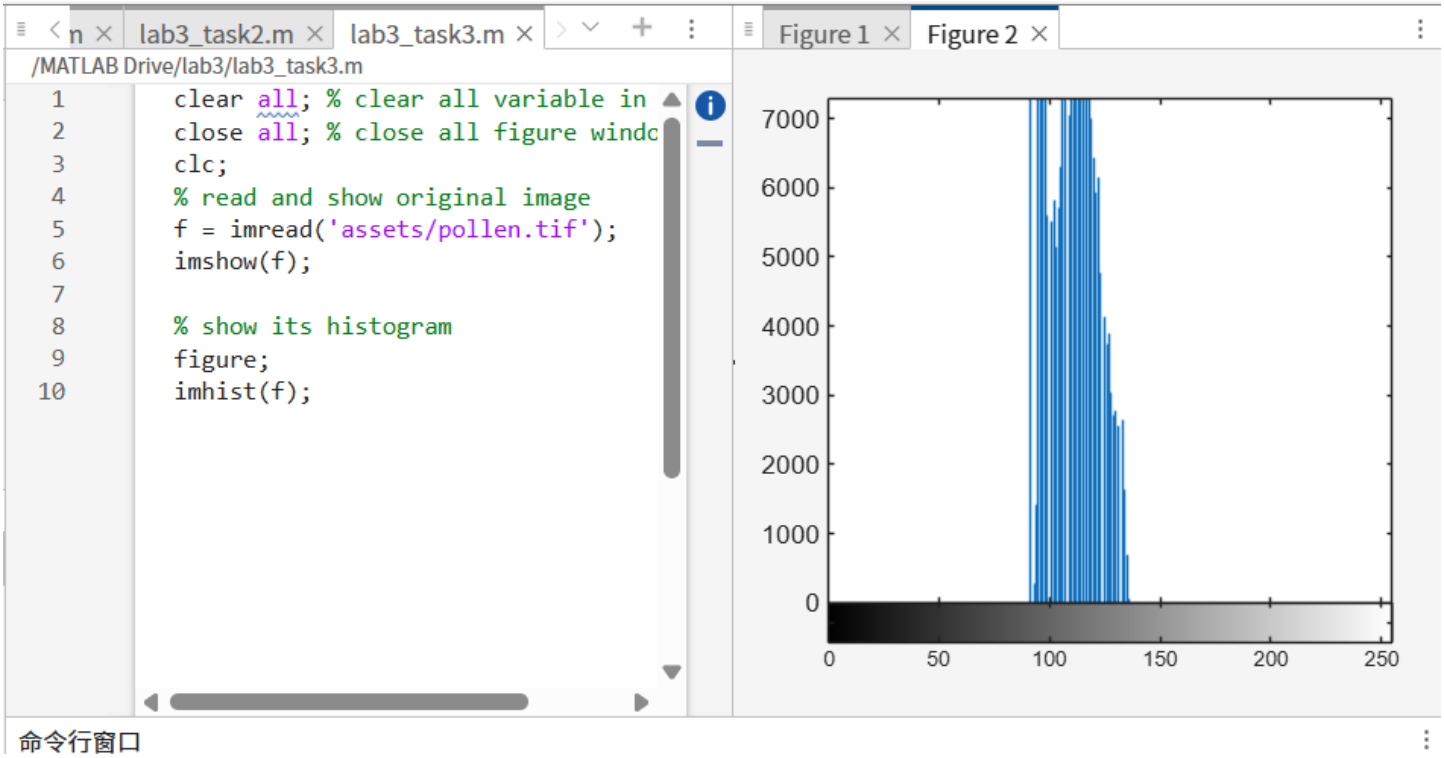
The Role of E: The parameter E controls how steep the transformation is. A higher E makes the contrast change more aggressively around the average brightness.

Data Type Matters: We must convert the image to `double` because the formula involves decimals and powers. If we stay in `uint8`, we would lose all the precision.

Handling Zero: Adding `eps` (a tiny number) is a clever trick to prevent the computer from crashing when it tries to divide by a pixel that has 0 brightness.

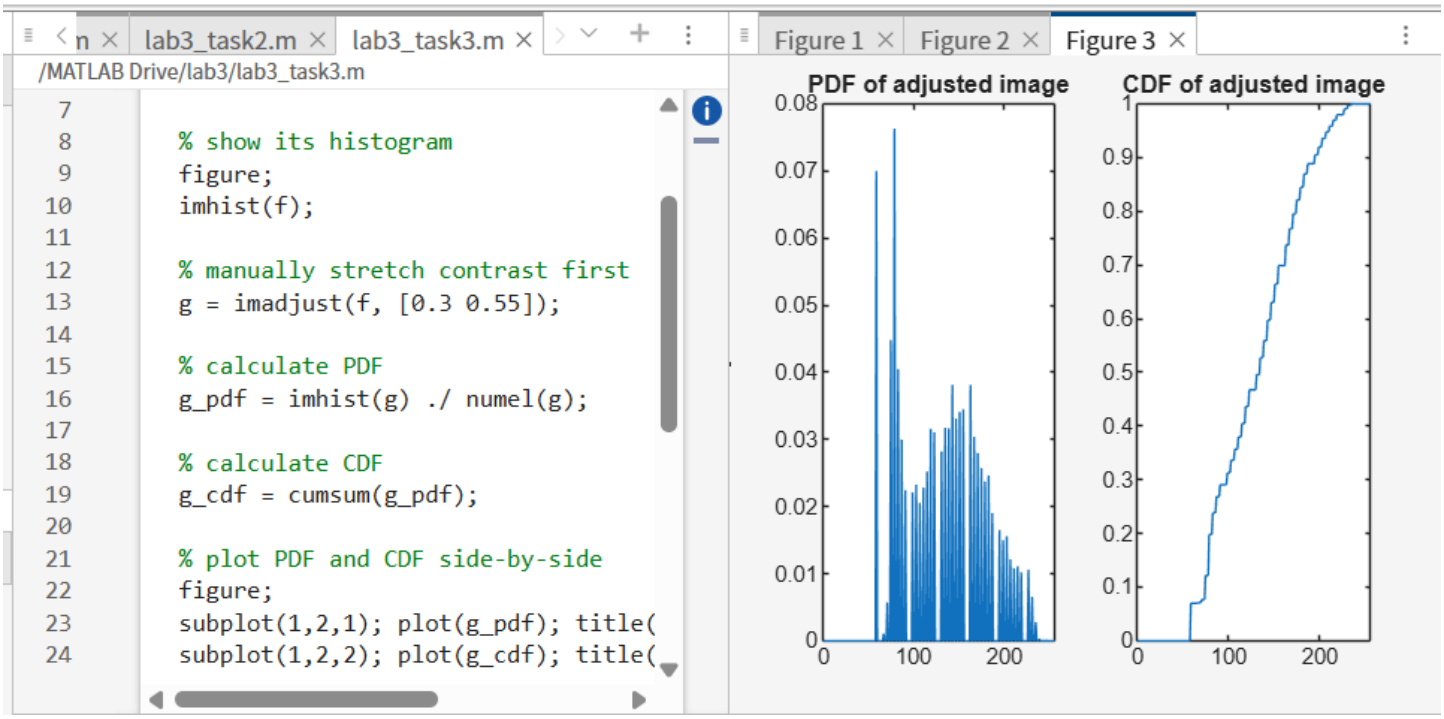
Task 3: Contrast Enhancement using Histogram



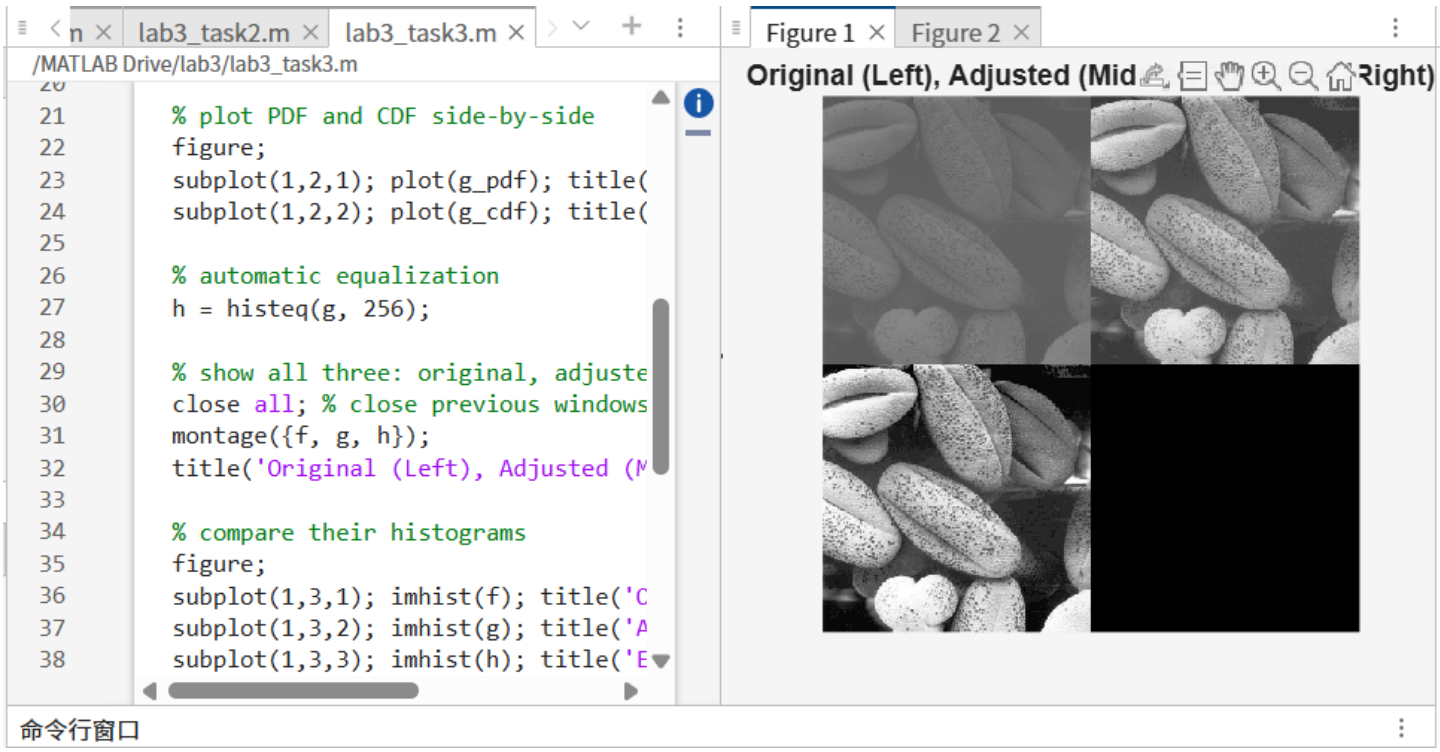


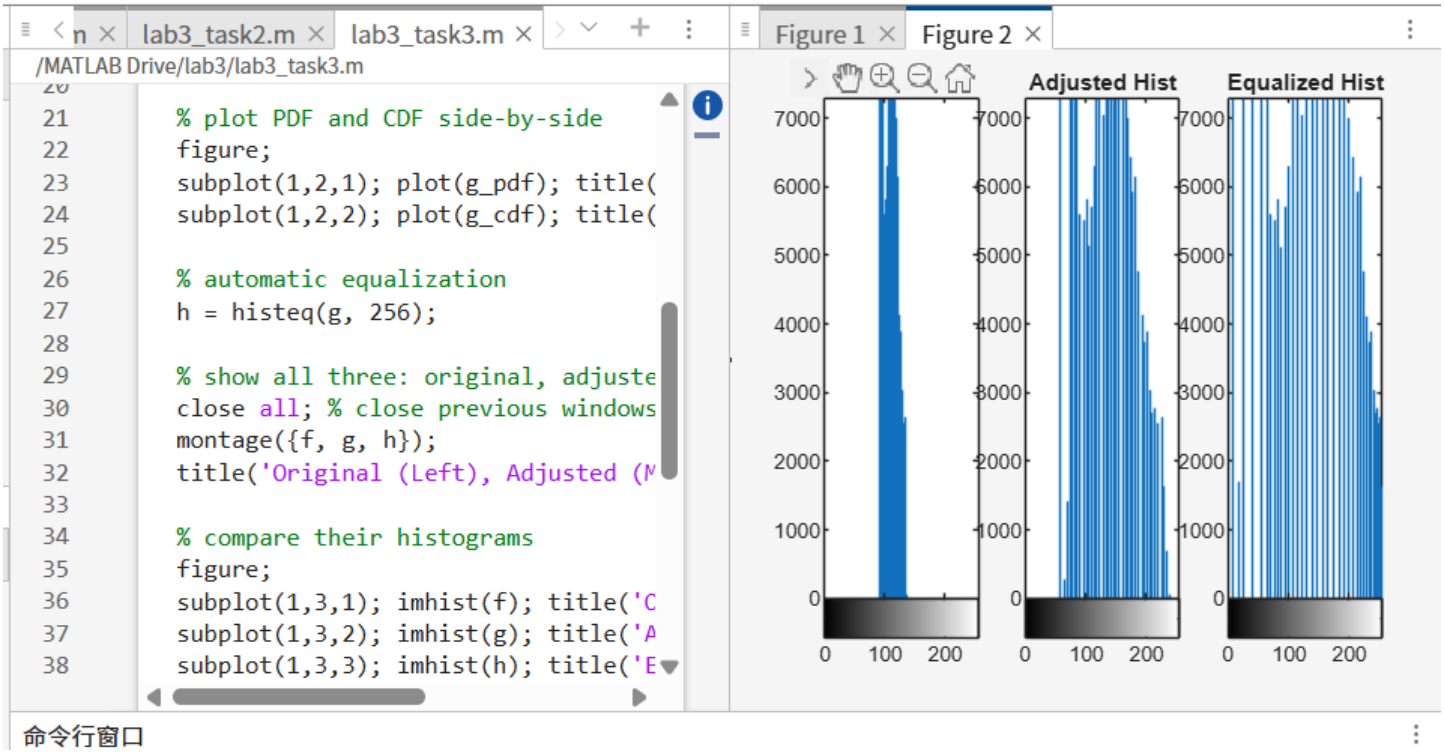
The pixels of the histogram are all concentrated within the range of 70 to 140. This indicates that the image has not fully utilized the contrast range of 0 to 255. Because it doesn't use the full range of 0 to 255, the image looks low-contrast, gray, and blurry.

- By using `imadjust(f, [0.3 0.55])`, we are manually stretching the middle part of the histogram to cover the full scale.
- This makes the image clearer than the original, but it is still not "perfect" because we are just guessing the best range.



- **PDF** : This tells us the percentage of pixels at each intensity level. It's basically a normalized version of the histogram.
- **CDF** : This is the running total of the PDF. In a perfectly equalized image, the CDF should ideally be a straight diagonal line, meaning every intensity has an equal chance of appearing.



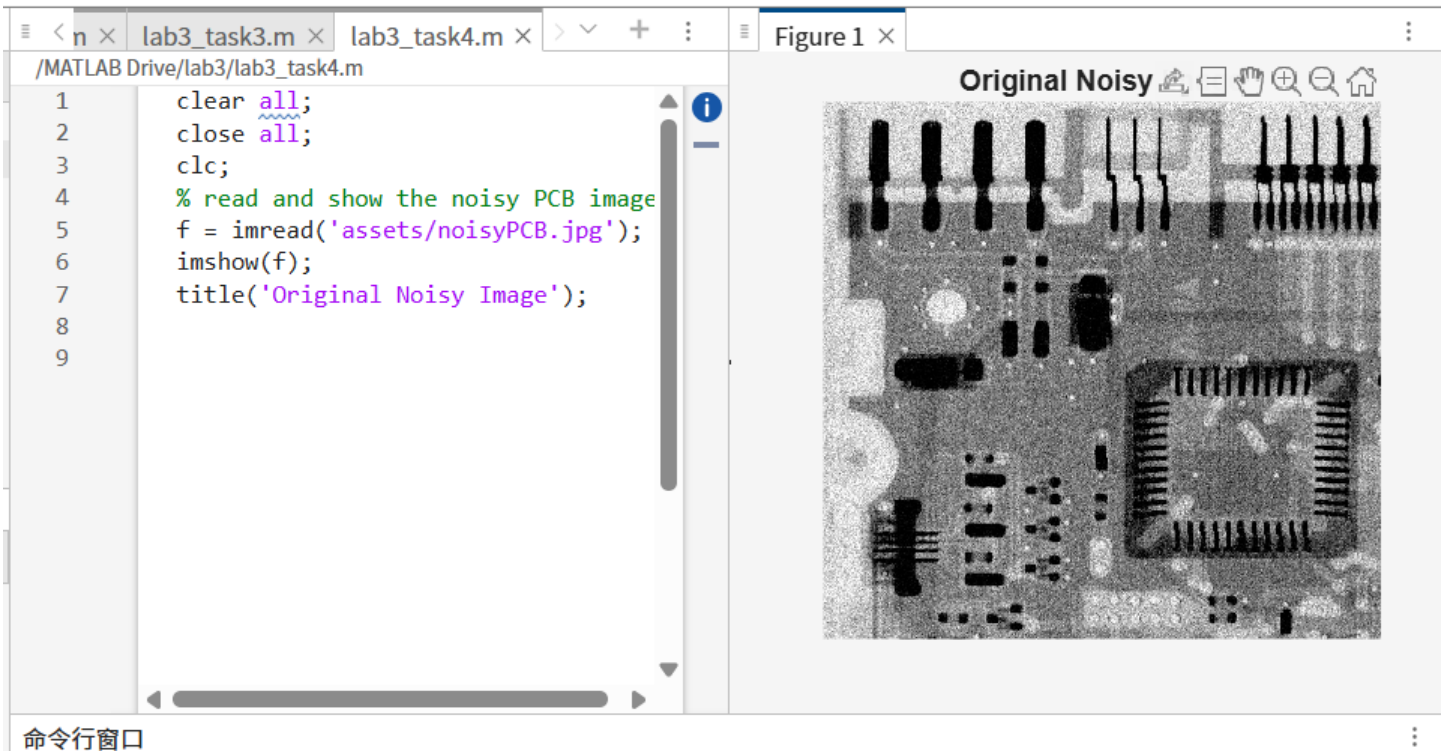


Automatic Optimization: The `histeq` function uses the CDF as a "map" to automatically flatten the histogram.

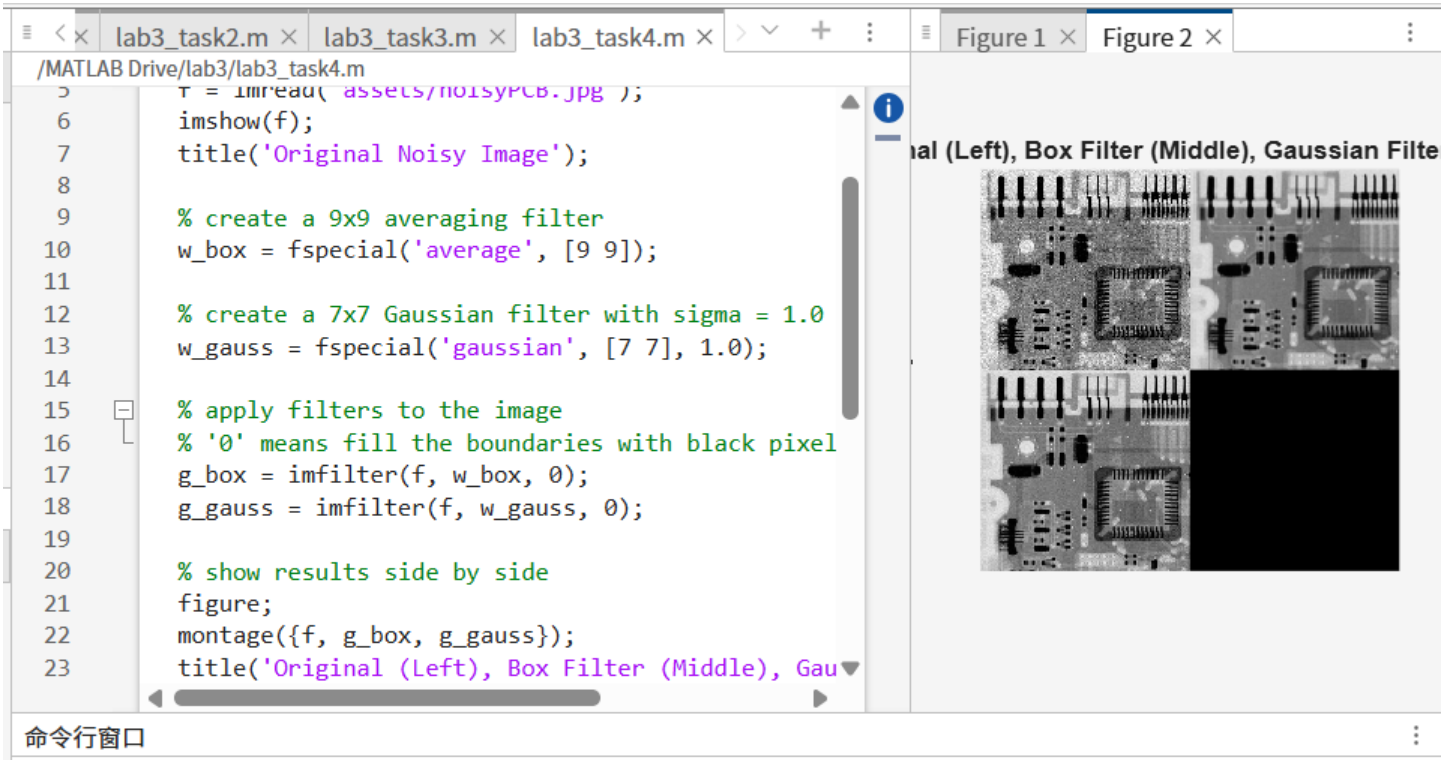
Final Result: The equalized image (h) usually shows much more detail in the pollen grains compared to the manual adjustment (g).

Mathematical Success: By spreading out the pixel intensities, we maximize the information visible to the human eye without having to manually tune parameters.

Task 4 - Noise reduction with lowpass filter



There are a lot of white noise dots.



Noise Reduction: Both filters successfully reduced the "salt and pepper" noise, making the background of the PCB look much cleaner.

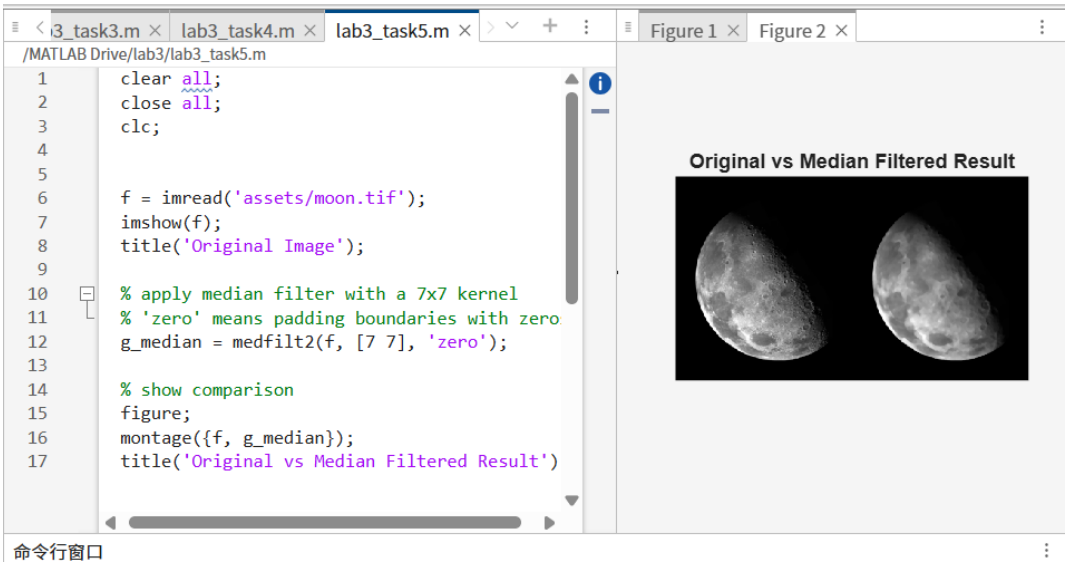
Blurring Trade-off: Notice that while the noise is gone, the image becomes **blurred**. This is the classic trade-off: you lose sharp edges (high-frequency details) to get rid of noise.

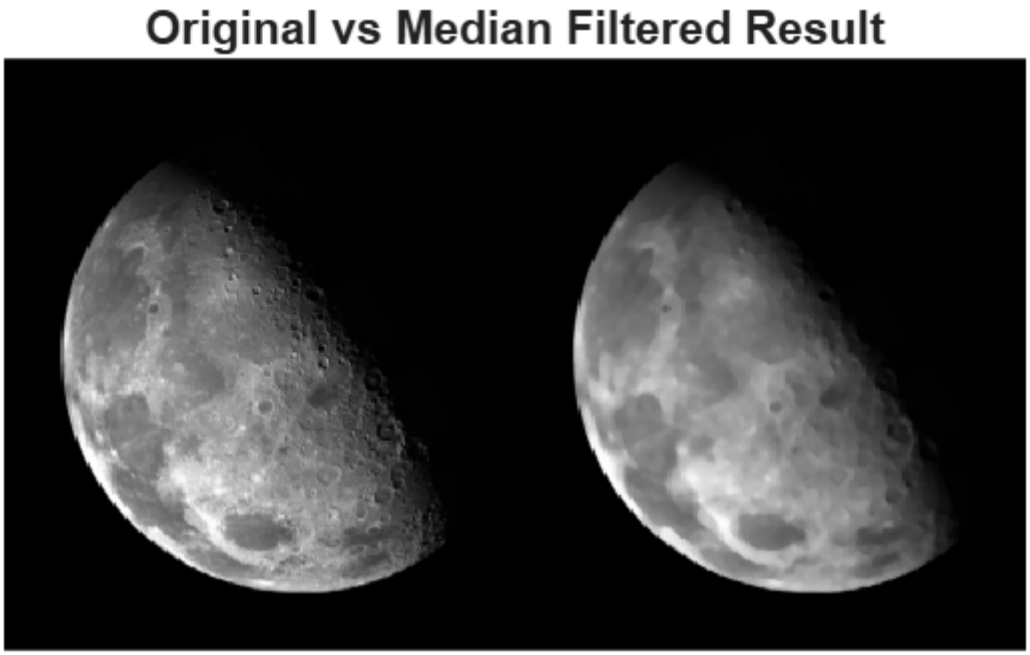
Box vs Gaussian:

- The **Box filter** (middle) usually causes more blurring because it treats all 81 pixels in the 9x9 area equally.
- The **Gaussian filter** (right) tends to preserve more details while smoothing because it gives more importance to the center pixel.

Kernel Size: A larger kernel (like 9x9) removes more noise but makes the image much blurrier than a smaller kernel (like 3x3).

Task 5 - Median Filtering



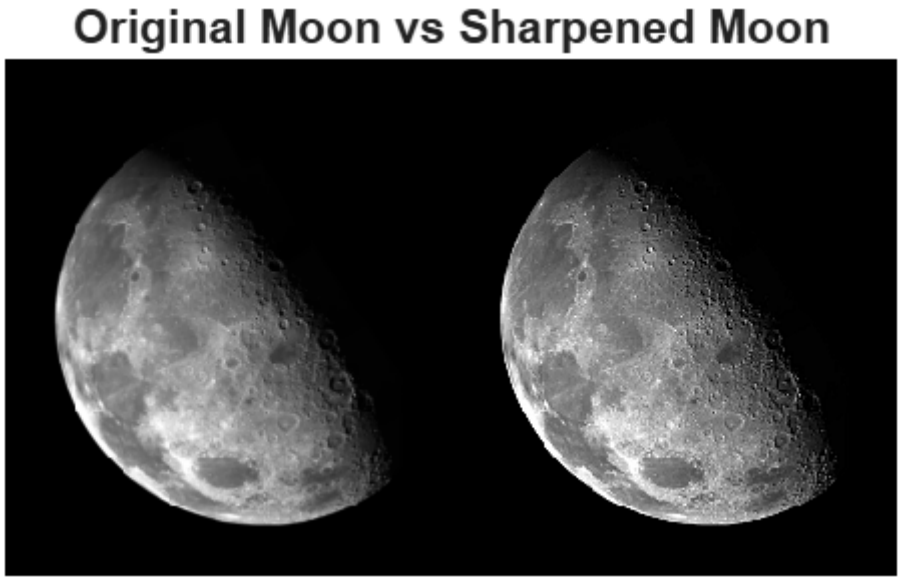


Sharpness Preservation: Compared to the Average or Gaussian filters in Task 4, the **Median filter** is much better at keeping the edges sharp while still removing noise.

Salt and Pepper Noise: Median filtering is specifically famous for removing salt and pepper noise because noise spikes (extreme values) are ignored when picking the median value.

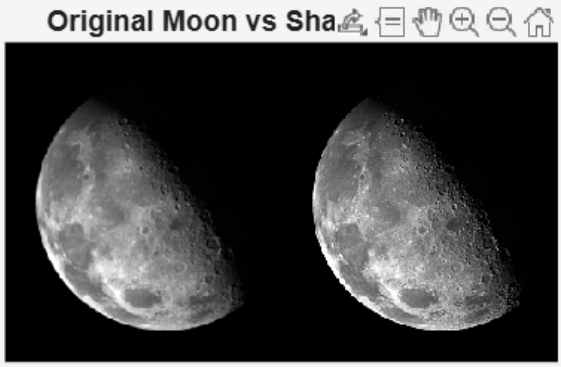
Kernel Choice: We used a `[7 7]` kernel. If the image still looks a bit noisy, you could increase it to `[9 9]`, but remember that a larger kernel might start to make the image look like an oil painting.

Task 6 - Sharpening the image with Laplacian, Sobel and Unsharp filters



/MATLAB Drive/lab3/lab3_task6.m

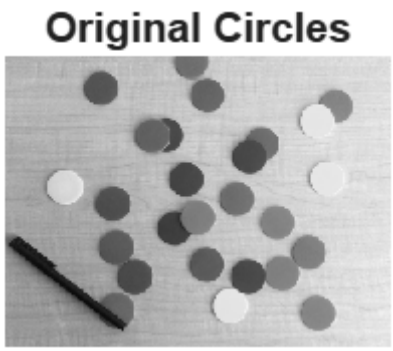
```
1   clear all; close all; clc;
2
3
4   f = imread('assets/moon.tif');
5   f_double = double(f);
6
7
8   w = fspecial('laplacian', 0);
9
10
11  g1 = imfilter(f_double, w, 'replica
12
13
14  g = f_double - g1;
15
16  g = uint8(g);
17  imshowpair(f, g, 'montage');
18  title('Original Moon vs Sharpened M
```

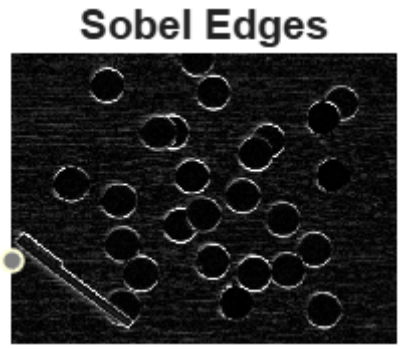


Edge Enhancement: Laplacian filter picks up high-frequency details (edges). By subtracting these from the original, we make the crater boundaries much sharper.

Visual Clarity: You should notice that the small ridges and pits on the moon are much more defined in the sharpened version.

Task 7 - Test yourself Challenges





code:

```
clear all; close all; clc;
%% Challenge 1: Improve Lake & Tree contrast
img_lake = imread('assets/lake&tree.png');
img_lake_en = histeq(img_lake);
%% Challenge 2: Find edges of circles
img_circles = imread('assets/circles.tif');
w_sobel = fspecial('sobel');
img_edges = imfilter(double(img_circles), w_sobel);
%% Display all results
figure;
subplot(2,2,1); imshow(img_lake); title('Original Lake');
subplot(2,2,2); imshow(img_lake_en); title('Enhanced Lake');
subplot(2,2,3); imshow(img_circles); title('Original Circles');
subplot(2,2,4); imshow(uint8(abs(img_edges))); title('Sobel Edges');
```

```
%% Challenge 3: Improve Office lighting
img_office = imread('assets/office.jpg');
% Using Gamma correction (gamma < 1 to brighten shadows)
img_office_en = imadjust(img_office, [], [], 0.5);
figure;
imshowpair(img_office, img_office_en, 'montage');
title('Original Office vs Enhanced Lighting');
```

insights：

Lake and Tree Contrast Improvement: The core logic is to fix the narrow intensity range of the original file. By using the histeq function, the algorithm calculates the Cumulative Distribution Function (CDF) and uses it as a map to redistribute pixels. This ensures that the water and trees, which might look gray or flat, now use the full range from 0 to 255, making the textures and reflections much sharper.

Circle Edge Detection: To find the boundaries of the circles, the code uses a Sobel operator, which is an edge detection tool. The logic uses fspecial to create a kernel that calculates the change in brightness (gradient) across the image. When this kernel slides over the image, it highlights the sharp transitions at the edges of the circles while ignoring the flat, solid colors inside the shapes or in the background.

Office Exposure Correction: For the office photo with poor lighting, the strategy is to adjust the exposure manually. The logic involves using Gamma Correction where the Gamma value is set below 1.0. This non-linearly maps dark pixels to brighter values, which helps reveal details hidden in the shadows of the office without making the already bright areas look washed out.

Data Consistency and Conversion: A critical part of the code logic is converting the image from uint8 to double before doing any math. This ensures that decimal values and calculations are precise. After the processing is done, the data is multiplied by 255 and converted back to uint8 so it can be saved and viewed as a normal image file.

Office Exposure: For the office image, Gamma correction is superior to simple linear stretching because it non-linearly maps the pixel intensities. Setting gamma = 0.5 helps lift the dark office interior while preserving the colors, which directly addresses the bad exposure mentioned in the task

LAB4

Task 1: Dilation and Erosion

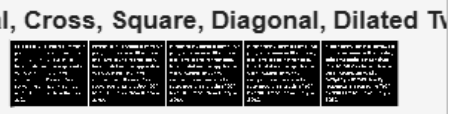
Dilation

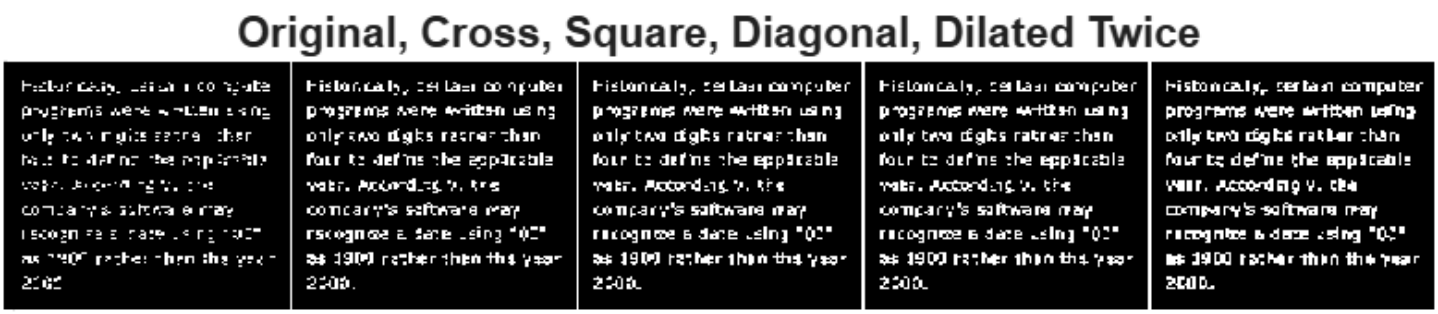
lab4_task1.m

/MATLAB Drive/lab4/lab4_task1.m

```
% 1. Use a cross shape SE
B1 = [0 1 0; 1 1 1; 0 1 0];
A1 = imdilate(A, B1);

% 2. Use a 3x3 square SE (all ones)
B2 = ones(3,3);
A2 = imdilate(A, B2);

% 3. Use a diagonal cross SE
Bx = [1 0 1; 0 1 0; 1 0 1];
Ax = imdilate(A, Bx);

% 4. Dilate twice with B1
A_twice = imdilate(A1, B1);

% Display results
figure;
montage({A, A1, A2, Ax, A_twice}, 'Size', [1 5]);
```

Figure 1





Pixel Expansion: The dilation process adds pixels to the boundaries of the white text. This explains why the characters look bolder or thicker in the processed images compared to the original.

Bridging Gaps: The primary goal for this specific image was to fix broken letters. By expanding the white regions, dilation fills in the small gaps within the characters, making them more complete and easier for OCR (Optical Character Recognition) to read.

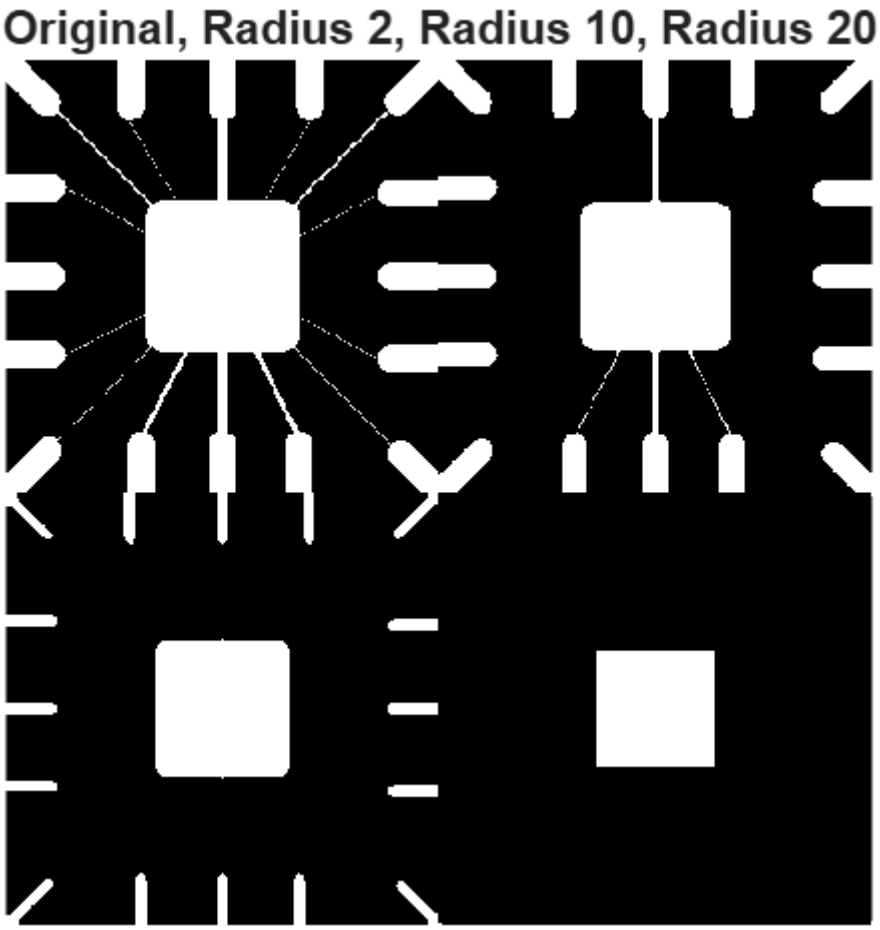
Structuring Element (SE) Influence:

- **Square vs. Cross**: The Square SE (`ones(3,3)`) adds pixels in all directions, including diagonals, resulting in a heavier thickening effect.
- **Diagonal**: The Diagonal Cross SE (`Bx`) specifically grows the pixels along the diagonal paths, which gives the text a slightly different "sharpened" geometric stretch.

Iterative Dilation: The "Dilated Twice" version shows that repeating the operation continues to grow the boundaries. While it bridges larger gaps, it also risks merging separate letters together

into unreadable blocks if done excessively.

Erosion



code：

```
% read wirebond mask image
A_mask = imread('assets/wirebond-mask.tif');

% create disk SE with different radii
SE2 = strel('disk', 2);
SE10 = strel('disk', 10);
SE20 = strel('disk', 20);

% perform erosion
E2 = imerode(A_mask, SE2);
E10 = imerode(A_mask, SE10);
E20 = imerode(A_mask, SE20);
```

% display

figure;

montage({A_mask, E2, E10, E20}, 'Size', [2 2]);

title('Original, Radius 2, Radius 10, Radius 20');

Erosion Effect: Erosion removes pixels on object boundaries. In the `wirebond-mask.tif` experiment, small radius SE (like 2) removes tiny noise, but large radius SE (like 10 or 20) completely deletes thin lines and small objects.

strel Advantage: Using `strel` instead of a manual matrix is more efficient because MATLAB optimizes the calculation for these specific geometric shapes.

Task 2 - Morphological Filtering with Open and Close

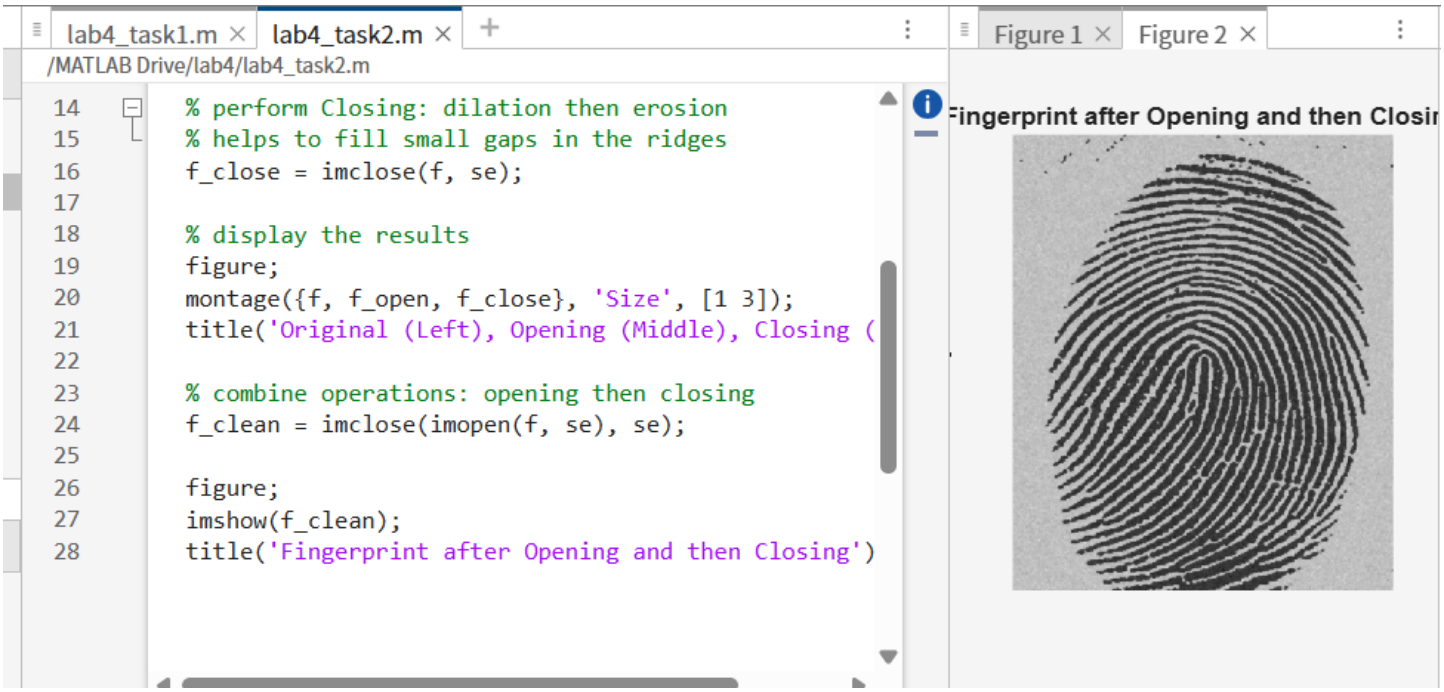
lab4_task1.m × lab4_task2.m ×

/MATLAB Drive/lab4/lab4_task2.m

```
4   % read the fingerprint image
5   f = imread('assets/fingerprint.tif');
6
7   % create a disk structuring element
8   se = strel('disk', 1);
9
10  % perform Opening: erosion then dilation
11  % helps to remove small white noise
12  f_open = imopen(f, se);
13
14  % perform Closing: dilation then erosion
15  % helps to fill small gaps in the ridges
16  f_close = imclose(f, se);
17
18  % display the results
19  figure;
20  montage({f, f_open, f_close}, 'Size', [1 3]);
21  title('Original (Left), Opening (Middle), Closing (Right)')
```

Figure 1 ×

al (Left), Opening (Middle), Closin



Opening Logic: The opening operation starts with erosion to eliminate small isolated white pixels (noise) that are smaller than the structuring element. It then follows with dilation to restore the shape of the larger remaining objects.

Closing Logic: The closing operation begins with dilation to bridge small gaps or holes within the white structures. This is followed by erosion to trim back the boundaries, effectively smoothing the contours and filling in cracks.

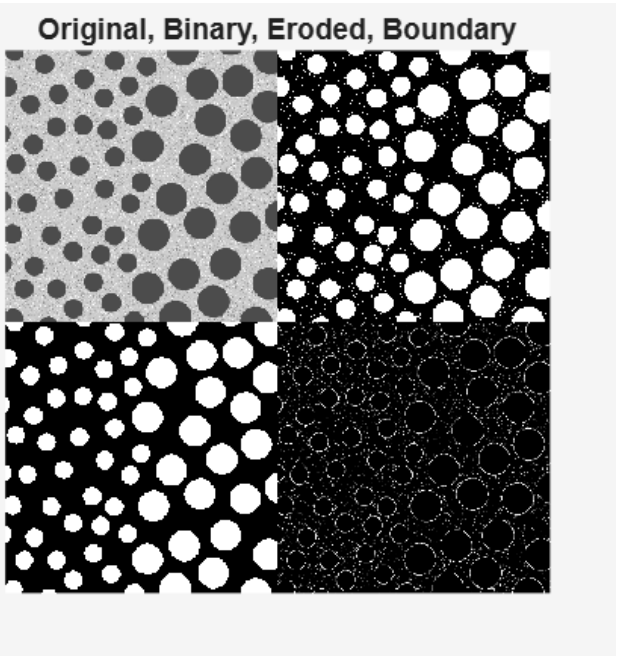
Practical Application: In fingerprint processing, Opening is used to remove "dust" or background noise, while Closing is used to join broken ridge lines, making the fingerprint pattern more continuous for identification.

Morphological Sequence: Using `imopen` and `imclose` is more efficient than calling `imerode` and `imdilate` separately, as MATLAB optimizes these combined routines for better performance.

Task 3 - Boundary detection

/MATLAB Drive/lab4/lab4_task3.m

```
1   clear all; close all; clc;
2
3   % step 1: Pre-processing
4   I = imread('assets/blobs.tif');
5   I_inv = imcomplement(I); % invert:
6   level = graythresh(I_inv); % find c
7   BW = imbinarize(I_inv, level); % cc
8
9   % step 10: Boundary extraction
10  SE = ones(3,3); % 3x3 square struct
11  BW_erode = imerode(BW, SE); % shrin
12  BW_edge = BW - BW_erode; % subtract
13
14  % step 3: Visualization
15  montage({I, BW, BW_erode, BW_edge},
16  title('Original, Binary, Eroded, Bc
```



Logic behind the subtraction: The boundary is found by taking the original binary image and removing its internal pixels. Since `imerode` "eats away" the outer layer of the white blobs, subtracting the eroded version from the original leaves only that outer layer—a one-pixel thick boundary.

Structuring Element (SE) choice: Using a 3 x 3 matrix of 1s ensures we look at all 8 neighbors of a pixel. If you used a larger SE, like 5 x 5, the detected boundary would be much thicker because more layers would be subtracted.

Sensitivity to Noise: One reflection is that this method is highly sensitive to noise. If the background has tiny white dots (noise), each dot will also get a boundary, creating a "messy" result.

Potential Improvement: To get a cleaner boundary, we should use `imfill(BW, 'holes')` before eroding. This ensures that if a blob has a dark spot inside, it doesn't create an internal boundary that we don't want.

The Inversion step: It is crucial to remember that morphological functions work on the white parts of an image. If we forgot `imcomplement`, we would be detecting the boundary of the background instead of the blobs.

Task 4 - Function bwmorph - thinning and thickening

Code

clear all; close all; clc;

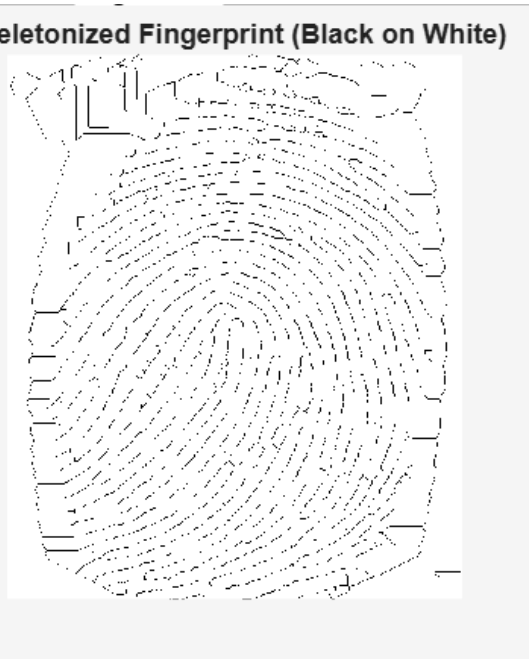
% step 1: load and binarize

f_gray = imread('assets/fingerprint.tif');

```
level = graythresh(f_gray); % find optimal threshold
f = imbinarize(f_gray, level); % create binary image

% step 2: thinning iterations
g1 = bwmorph(f, 'thin', 1);
g2 = bwmorph(f, 'thin', 2);
g3 = bwmorph(f, 'thin', 3);
g4 = bwmorph(f, 'thin', 4);
g5 = bwmorph(f, 'thin', 5);

% step 3: thinning to infinity (skeletonization)
% inf repeats until the image stops changing
g_inf = bwmorph(f, 'thin', inf);

% step 4: visualization (white on black)
figure;
montage({f, g1, g2, g3, g4, g5, g_inf}, 'Size', [1 7]);
title('Original and Thinning Iterations (1, 2, 3, 4, 5, Inf)');

% step 5: display black lines on white background
figure;
imshow(imcomplement(g_inf));
title('Skeletonized Fingerprint (Black on White)');
```

/MATLAB Drive/lab4/lab4_task4.m

```
1   clear all; close all; clc;
2
3   % step 1: load and binarize
4   f_gray = imread('assets/fingerprint.tif'
5   level = graythresh(f_gray); % find optim
6   f = imbinarize(f_gray, level); % create
7
8   % step 2: thinning iterations
9   g1 = bwmorph(f, 'thin', 1);
10  g2 = bwmorph(f, 'thin', 2);
11  g3 = bwmorph(f, 'thin', 3);
12  g4 = bwmorph(f, 'thin', 4);
13  g5 = bwmorph(f, 'thin', 5);
14
15  % step 3: thinning to infinity (skeleton
16  % inf repeats until the image stops chan
17  g_inf = bwmorph(f, 'thin', inf);
18
19  % step 4: visualization (white on black)
```



insights

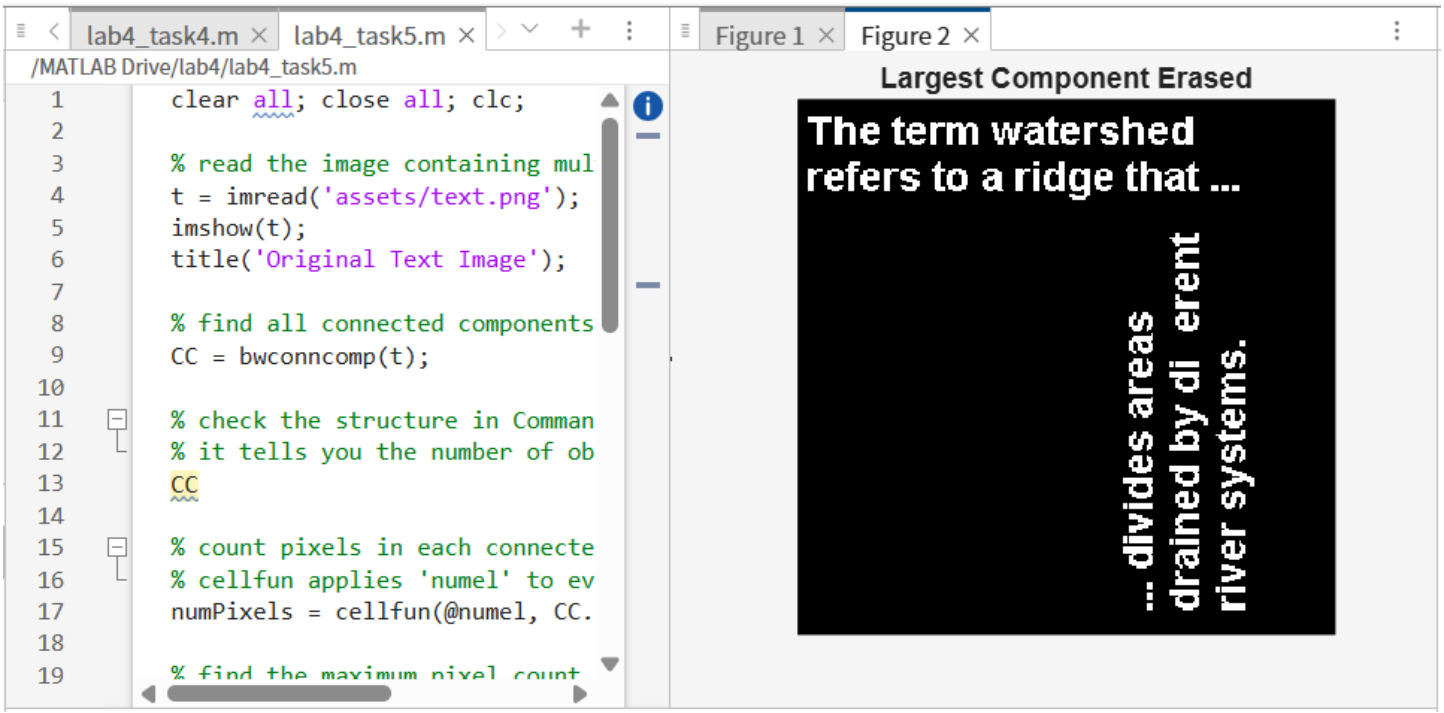
Thinning Process: The `thin` operation removes pixels from the boundaries of foreground objects (white ridges) without breaking their connectivity. Each iteration makes the ridges narrower until they are reduced to a single-pixel width.

Result of `n = inf` : When n is set to infinity, the operation continues until only a **skeleton** remains. This skeleton represents the essential shape and topology of the fingerprint, which is critical for identifying minutiae points in biometric systems.

Relationship between Thinning and Thickening:

- Thinning the foreground (white ridges) is the mathematical dual of thickening the background (black valleys).
- While thinning reduces an object to its minimal "strokes," thickening expands an object's boundary without joining disconnected components.
- By displaying black lines on a white background, we observe that thinning helps clarify the structure of the ridges, whereas thickening would have filled the gaps between them, potentially merging distinct features

Task 5 - Connected Compone



code;

clear all; close all; clc;

% read the image containing multiple characters

t = imread('assets/text.png');

imshow(t);

title('Original Text Image');

% find all connected components

CC = bwconncomp(t);

% check the structure in Command Window

% it tells you the number of objects found

CC

% count pixels in each connected component

% cellfun applies 'numel' to every cell in the list

numPixels = cellfun(@numel, CC.PixelIdxList);

% find the maximum pixel count and its index

[biggest, idx] = max(numPixels);

% erase the biggest component by setting its pixels to 0

t(CC.PixelIdxList{idx}) = 0;

% show the result

figure;

imshow(t);

title('Largest Component Erased');

insights;

Logic of Connected Components: The `bwconncomp` function groups neighboring white pixels (1's) into independent objects. This is the mathematical basis for object recognition in image processing.

The Power of cellfun: Instead of writing a slow "for-loop" to count pixels in hundreds of characters, `cellfun` allows us to apply the `numel` function to the entire list at once, making the code much faster and cleaner.

Data Structure (CC): The `CC.PixelIdxList` is a collection of lists. Each sub-list contains the exact "address" of every pixel belonging to one specific character.

Reflection on Erasing: By setting `t(CC.PixelIdxList{idx}) = 0`, we are using logical indexing to target only the largest object. In a real-world scenario, this technique can be used to remove large background artifacts or focus on the most prominent object in a scan.

Self-Correction/Observation: If your image was "inverted" (black text on white), `bwconncomp` would treat the entire white background as one giant object. Always ensure your objects are white (1) and background is black (0) before using this function.

Task 6 - Morphological Reconstruction

/MATLAB Drive/lab4/lab4_task6.m

```
1   clear all; close all; clc;
2   f = imread('assets/text_bw.tif');
3   se = ones(17,1);
4   g = imerode(f, se);
5   fo = imopen(f, se);
6   fr = imreconstruct(g, f);
7   montage({f, g, fo, fr}, "size", [2 2]);
8   ff = imfill(f, 'holes');
9
10  figure;
11  montage({f, ff});
12  title('Before (Left) vs After (Right) imfill')
```

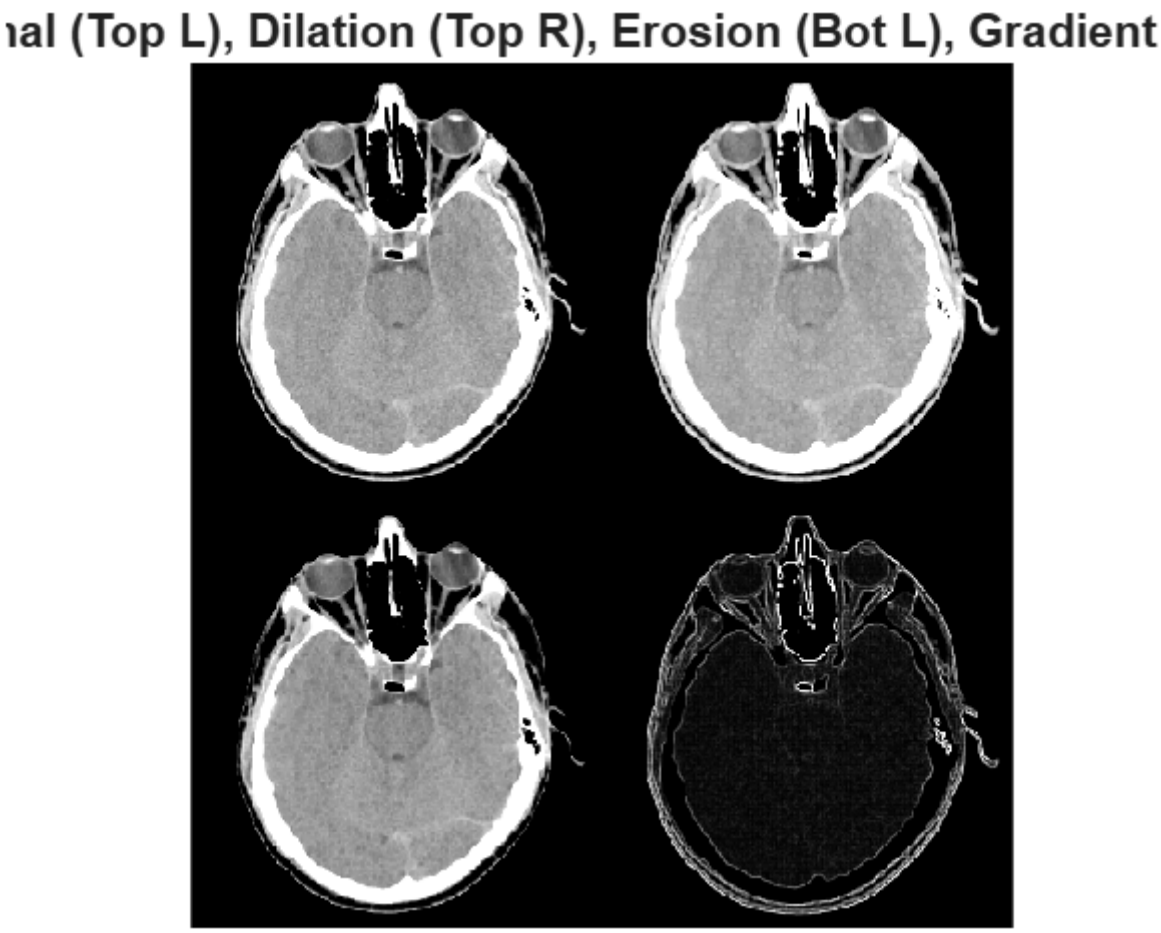
Reconstruction vs. Opening: Regular `imopen` cuts off parts of the characters because it only restores the shape of the structuring element. `imreconstruct`, however, brings back the **entire connected component** from the original image as long as the marker touches it.

Targeted Filtering: we are specifically filtering the image to keep only long, thin vertical elements (like the stems of certain letters) while deleting everything else.

Logical Flow: The marker g tells the computer **where** to look, and the mask f tells the computer **what** the original shape looked like so it can be perfectly restored.

Refining results: Functions like `imfill` are essential pre-processing steps. If a character has a "hole" due to poor binarization, `imfill` ensures the morphological operations treat it as a single solid object.

Task 7 - Morphological Operations on Grayscale images



code;

```
clear all; close all; clc;

% step 1: Load grayscale head CT scan
f = imread('assets/headCT.tif');

% step 2: Create a 3x3 square structuring element
se = strel('square', 3);

% step 3: Perform grayscale dilation and erosion
% Dilation brightens and expands light areas
gd = imdilate(f, se);
% Erosion darkens and shrinks light areas
ge = imerode(f, se);

% step 4: Morphological Gradient (Edge detection)
```

% Subtracting the eroded image from dilated image

gg = gd - ge;

% step 5: Display results as a 2x2 montage

montage({f, gd, ge, gg}, 'size', [2 2]);

title('Original (Top L), Dilation (Top R), Erosion (Bot L), Gradient (Bot R)');

insights

Grayscale Dilation (`gd` **)**: In grayscale images, dilation acts like a **maximum filter**. It chooses the highest pixel value in the 3x3 neighborhood, making the bright structures (like bone in the CT scan) look thicker and brighter.

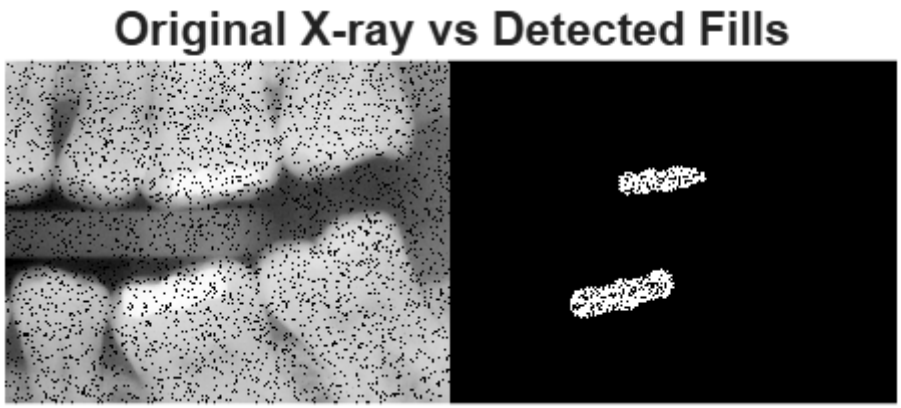
Grayscale Erosion (`ge` **)**: This acts like a **minimum filter**. It chooses the lowest pixel value in the neighborhood, causing dark areas to expand and making the bright features appear smaller and dimmer.

Morphological Gradient (`gg` **)**: The subtraction of the eroded image from the dilated image (gd - ge) highlights the **boundaries** where the intensity changes rapidly. This is a powerful way to perform edge detection on complex grayscale medical images.

Visual Conclusion: While dilation and erosion in binary images simply change the shape of objects, in grayscale images they effectively shift the local brightness levels, allowing for advanced texture and edge analysis.

Challenges1

code;



clear all; close all; clc;

% Step 1: Load the dental X-ray image

% The file is located in the assets folder

```
f = imread('assets/fillings.tif');

% Step 2: Binarization
% Fillings are usually made of metal, so they appear very bright (near white)
% I used a high threshold (0.9) to isolate only the brightest spots
bw = imbinarize(f, 0.9);

% Step 3: Cleaning up noise
% Some small bright spots might just be noise, not real fills
% I used an opening operation with a small disk to remove them
se = strel('disk', 2);
bw_clean = imopen(bw, se);

% Step 4: Counting and Measuring
% bwconncomp finds all separated white objects
cc = bwconncomp(bw_clean);

% Use cellfun to count pixels in each detected fill
numPixels = cellfun(@numel, cc.PixelIdxList);

% Output the results to the Command Window
fprintf('The patient has %d fills in total.\n', cc.NumObjects);
disp('The size of each fill (in pixels) is:');
disp(numPixels);

% Step 5: Final Check
imshowpair(f, bw_clean, 'montage');
title('Original X-ray vs. Detected Fills');
```

insights

Threshold Selection: Choosing the right threshold is the most important step. If the threshold is too low, we might accidentally count the bright parts of the teeth or bone as fills.

The Noise Problem: X-ray images are often grainy. Without the `imopen` step, the computer would detect hundreds of tiny fills that are actually just sensor noise.

Connected Components Logic: `bwconncomp` treats any group of white pixels that touch each other as one single object. This is much more efficient than trying to find objects manually.

Pixel Area as Size: By using `numel` on each component, we get the "area" of the fill. Larger pixel counts mean a larger cavity was filled by the dentist.

Reflection: This method is great for high-contrast objects like metal fills, but it might struggle if the image exposure is bad or if the fills are small and blurry.

challenges2

code

clear all; close all; clc;

% Load the grayscale palm image

f = imread('assets/palm.tif');

% Step 1: Pre-processing

% Palm lines are usually darker, so we binarize and invert

% to make the lines white (foreground) for morphological functions.

bw = imcomplement(imbinarize(f, graythresh(f)));

% Step 2: Create a Marker

% We use a disk SE to erode the image. This "eats away" the thin

% skin textures and small noise, leaving only the "seeds" of the main lines.

se = strel('disk', 3);

marker = imerode(bw, se);

% Step 3: Morphological Reconstruction

% Instead of using imdilate (which would just make the seeds into blobs),

% imreconstruct uses the marker to grow back into the original shape

```
% defined by the mask (bw). This restores the full length of main lines.
main_lines = imreconstruct(marker, bw);

% Step 4: Final Cleanup (Optional)
% Using thinning to get the "skeleton" of the palm lines for a cleaner look.
skeleton = bwmorph(main_lines, 'thin', inf);

% Show our hard work!
figure;
subplot(1,3,1); imshow(f); title('Original Palm');
subplot(1,3,2); imshow(main_lines); title('Extracted Main Lines');
subplot(1,3,3); imshow(imcomplement(skeleton)); title('Line Skeleton');
```

insights;

Logic of Inversion: Morphological functions in MATLAB are designed to process white pixels as the objects of interest. Since palm lines are naturally darker than the skin, using `imcomplement` after binarization is a necessary step to turn those dark lines into white foreground structures.

Marker vs. Mask Strategy: In this script, the binary image `bw` acts as the **mask**, which contains all the data, including noise. The `marker` is created by eroding the mask, effectively "filtering out" skin textures that are too small to survive the `strel('disk', 3)` erosion.

The Power of Reconstruction: Unlike a standard `imdilate` which would just expand the markers into blurry circles, `imreconstruct` uses the markers as starting points to "regrow" the original shapes found in the mask. This ensures that we recover the full, natural length of the main palm lines while completely ignoring the noise that was erased during erosion.

Skeletonization for Clarity: By applying `bwmorph(main_lines, 'thin', inf)`, we reduce the thick extracted lines down to a single-pixel width. This is a classic reflection of how thinning helps simplify complex structures into their basic topological paths, making it much easier to analyze the geometry of the palm.

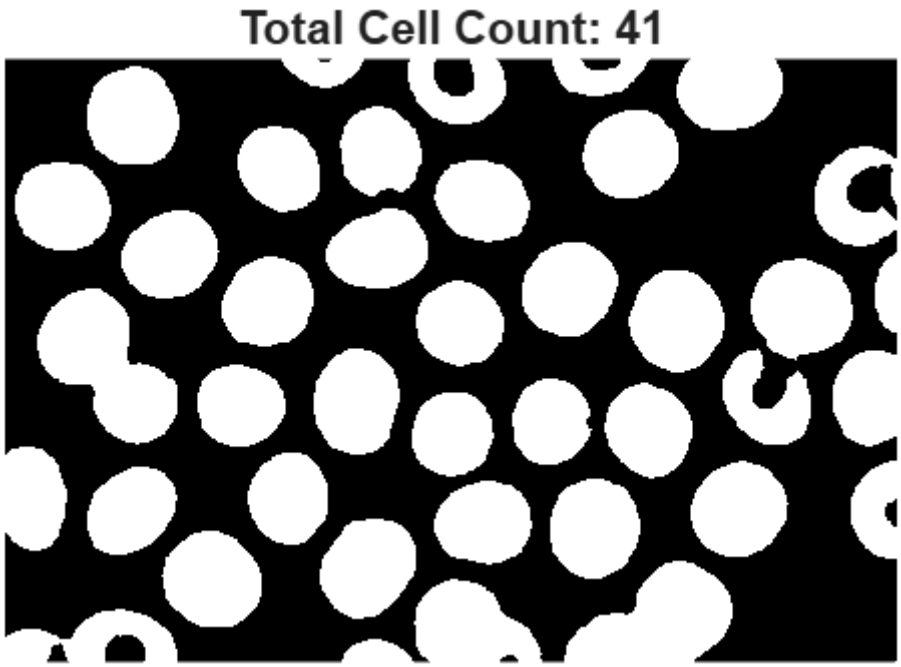
Refining the Result: A key takeaway from this challenge is that the size of the structuring element (SE) is critical. If the disk radius is too small, skin noise will remain; if it is too large, even the main palm lines might be eroded away and lost forever.

challenges3

code

```
clear all; close all; clc;


% step 1: load the image and convert to grayscale
% normal-blood.png is colored, so we need rgb2gray first
img = imread('assets/normal-blood.png');
gray_img = rgb2gray(img);

% step 2: binarization
% the cells are dark on a light background, so we invert it
% after inversion, cells become white (1) and background becomes black (0)
bw = imcomplement(imbinarize(gray_img));

% step 3: clean up the image
% fill the "holes" inside the cells so they are solid white blobs
bw_filled = imfill(bw, 'holes');

% step 4: separate touching cells
% this is the "secret sauce": shrinking the cells slightly
% so that those sticking together become separated
se = strel('disk', 4);
bw_clean = imerode(bw_filled, se);
```

```
% step 5: count the objects
% bwconncomp finds all the independent white blobs
cc = bwconncomp(bw_clean);

% step 6: show the results
fprintf('Result: I found %d red blood cells in this image!\n', cc.NumObjects);
imshow(bw_clean);
title(['Total Cell Count: ', num2str(cc.NumObjects)]);
```



insights;

Why imfill is necessary: When we binarize the cells, some might have dark spots in the middle due to lighting. If we don't fill these holes, `bwconncomp` might mistakenly count one cell as a "ring" or multiple pieces.

The Erosion Trick: Touching cells are the biggest nightmare for counting. By using `imerode` with a small disk, we "shave off" the outer pixels. This breaks the thin bridges between cells that are just barely touching, making the count much more accurate.

Refining the Structuring Element (SE): I found that a `disk` with radius 4 works well. If the radius is too small, cells stay stuck together; if it is too big, the smaller cells might disappear completely.

Logical Indexing Reflection: Using `bwconncomp` is way easier than manually counting. It basically creates a map of the image and assigns a unique ID to every separate white "island" it finds.

AI Acknowledgement

I would like to acknowledge the use of AI assistance (Gemini) in the preparation of this logbook. Specifically, AI was used for the following purposes:

- **Code Documentation**: Assisting in generating concise and clear English comments for the MATLAB scripts to ensure readability.
- **Language Polishing**: Refining the grammar and phrasing of the written English content to ensure professional communication.

However, all **critical insights**, **reflections**, and **experimental analyses** presented in this logbook are my original thoughts, derived from my direct observations during the lab sessions and my understanding of the module's core principles.