**🧠 Step-by-Step Explanation**

**img = imread('your\_image.jpg');**

* Reads the input image from your system.
* This image can be **RGB (color)** or **grayscale**.

**subplot(2,3,1); imshow(img); title('Original Image');**

* Creates a **2-row, 3-column grid** for plotting.
* Displays the **original image** in the **first slot**.
* Helpful for comparing with grayscale and processed versions.

**if size(img, 3) == 3 ...**

* Checks if the image has **3 color channels** (i.e., is RGB).
* If **yes**, converts it to grayscale using rgb2gray.
* If already grayscale or binary, it skips conversion.

**if ~isa(gray\_img, 'uint8') ...**

* Ensures the grayscale image is in **uint8 format**, which most MATLAB image functions require.
* Converts from logical, double, etc., using im2uint8.

**subplot(2,3,2); imshow(gray\_img); title('Grayscale Image');**

* Shows the **converted grayscale** image in the second subplot.
* Useful to see what the image looks like before histogram equalization.

**local\_eq\_img = adapthisteq(gray\_img, 'NumTiles', [8 8], 'ClipLimit', 0.01);**

* Applies **adaptive histogram equalization (CLAHE)**:
  + Enhances **local contrast** in the image.
  + NumTiles = [8 8]: divides the image into 8×8 blocks for processing.
  + ClipLimit = 0.01: prevents over-amplifying noise by limiting contrast.

**subplot(2,3,3); imshow(local\_eq\_img); title('Locally Histogram Equalized');**

* Displays the **contrast-enhanced image** (after local histogram equalization).

**subplot(2,3,5); imhist(gray\_img); title('Histogram of Grayscale');**

* Plots the **histogram of the grayscale image** (before enhancement).
* A histogram shows how pixel intensities are distributed from black (0) to white (255).

**subplot(2,3,6); imhist(local\_eq\_img); title('Histogram after Local Equalization');**

* Plots the **histogram of the equalized image**.
* You should notice the histogram becomes **more spread out**, improving contrast.

## 🔍 What Is Adaptive Histogram Equalization (AHE)?

### Regular Histogram Equalization:

* Works on the **whole image**.
* Spreads out pixel intensity values to improve **global contrast**.
* Can sometimes **over-enhance noise** or **wash out small features**, especially in large or unevenly lit images.

### Adaptive Histogram Equalization (AHE):

* Breaks the image into **small blocks (tiles)** — like 8×8 or 16×16 grids.
* Applies histogram equalization **locally** to each block.
* Helps enhance **local details and textures**, even in shadowed or bright regions.

## 💡 What Makes CLAHE Special?

**CLAHE = Contrast Limited Adaptive Histogram Equalization**

CLAHE solves a key problem with AHE: **noise amplification.**

### ✋ Problem with AHE:

In flat regions (e.g., sky), AHE may overly stretch the histogram, **amplifying noise**.

### ✅ CLAHE Fixes This By:

* **Clipping** the histogram at a certain limit (called the **ClipLimit**).
* Redistributing excess pixels uniformly to prevent over-enhancement.

## 🧪 Parameters in adapthisteq

local\_eq\_img = adapthisteq(gray\_img, 'NumTiles', [8 8], 'ClipLimit', 0.01);

| **Parameter** | **Meaning** |
| --- | --- |
| gray\_img | Input grayscale image |
| 'NumTiles' | Number of tiles in rows and columns (e.g., [8 8] = 8x8 grid) |
| 'ClipLimit' | Controls contrast limit; lower = less contrast, safer from noise |

## 📊 Visual Summary

* CLAHE = Adaptive local contrast boost + smart control over noise
* Think of it as:  
  🔦 "Enhance only where it's meaningful, and don’t let the noise shine!"

## 📸 When to Use CLAHE?

✅ Use CLAHE when:

* The image has **uneven lighting** (e.g., shadows, glare)
* You want to highlight **fine details** in textures (e.g., medical imaging, aerial photos)
* The standard histogram equalization makes your image look **overdone**

**NumTiles** parameter specifies the **number of tiles (windows)** to divide the image into for applying **adaptive histogram equalization**.

### 🧠 Understanding NumTiles

* **NumTiles = [8 8]** means the image will be divided into **8×8 blocks** (each block is a small window or tile).
* Each tile will have its own local histogram equalization applied, enhancing contrast in that specific region of the image.

### 🎯 How It Works:

* When you set NumTiles = [8 8], MATLAB will split your image into 8x8 regions. It will then apply histogram equalization individually to each of these blocks.
* If you increase the number of tiles (e.g., [16 16]), each block becomes smaller, and the enhancement will be more localized. This could result in more detailed adjustments but might also increase computational time.

### Example:

* **NumTiles = [8 8]**: If your image is 256×256 pixels, it will be divided into **32 rows and 32 columns**, each containing an 8x8 block.
* **NumTiles = [16 16]**: For the same 256×256 image, the image will be divided into **16 rows and 16 columns**, each with a 16x16 block.

### 📏 Choosing Tile Size:

* **Smaller tiles** (e.g., [16 16]) lead to more detailed local enhancements.
* **Larger tiles** (e.g., [8 8]) may result in smoother adjustments with fewer visible changes at the boundaries between tiles.

Great! You've generated histograms for **local histogram equalization** with different tile sizes: [8×8], [16×16], and [32×32]. Let's break down how to interpret these histograms.

### 📈 What Does the Histogram Show?

Each **histogram** is a plot of:

* **X-axis**: Pixel intensity values from **0 (black)** to **255 (white)**.
* **Y-axis**: **Frequency** — how many pixels in the image have that intensity.

### 🧠 What Should Happen with Histogram Equalization?

* **Goal**: To **redistribute** the pixel intensities so that all values across the grayscale range are more uniformly used.
* This usually results in a **flatter** histogram and improved image contrast.

### 🧪 Analysis of Your 3 Histograms

#### 1. ****8×8 Tiles****

* Local contrast enhancement is very strong.
* Histogram is the most **spread out** — more bins are filled across the grayscale range.
* Since it's very localized, **small details get enhanced** (but may introduce slight noise or graininess in smooth areas).

#### 2. ****16×16 Tiles****

* A balance between local detail and smoothness.
* Histogram still shows good spreading across intensities but is a bit less aggressive than 8×8.
* It enhances **both local features and global structure** more gently.

#### 3. ****32×32 Tiles****

* Larger tiles = less localization.
* Histogram is closer to the original — some intensities dominate more.
* Enhances broader regions, but small details might not be as clearly brought out.

### 📌 Summary

| **Tile Size** | **Enhancement Style** | **Histogram Shape** | **Best For** |
| --- | --- | --- | --- |
| **8×8** | Very local, detailed | Broad, flat distribution | Fine detail and texture contrast |
| **16×16** | Balanced | Moderately flat | Natural look with better contrast |
| **32×32** | Smooth, less detail | Less spread | Large object contrast |

### 🧾 1. ****Global Histogram Equalization****

#### 📷 Image:

* The contrast is enhanced uniformly over the whole image.
* Some areas might look overexposed (too bright) or underexposed (too dark) because **global equalization does not consider local variations**.

#### 📊 Histogram:

* Very **flat and spread out**: this is expected!
* The goal of global equalization is to **distribute the intensities evenly**, so this histogram is close to uniform.
* It stretches the original pixel values to occupy the full dynamic range (0–255).

### 🧾 2. ****Adaptive Histogram Equalization****

#### 📷 Image:

* Looks **more detailed in all regions**, especially where contrast was low before (like shadows or textures).
* Adaptive equalization enhances contrast **locally** using tiles (in your case, 8×8 blocks), which gives **finer contrast control**.

#### 📊 Histogram:

* Still spread out, but **not as flat** as global equalization.
* This is normal—adaptive equalization is based on **local stats**, so global pixel distribution is not forced to be uniform.
* It retains more natural intensity distribution while improving visibility in localized areas.

GUI:-

Features that I want are- a slider that slides window options from 8x8 16x16 32x32

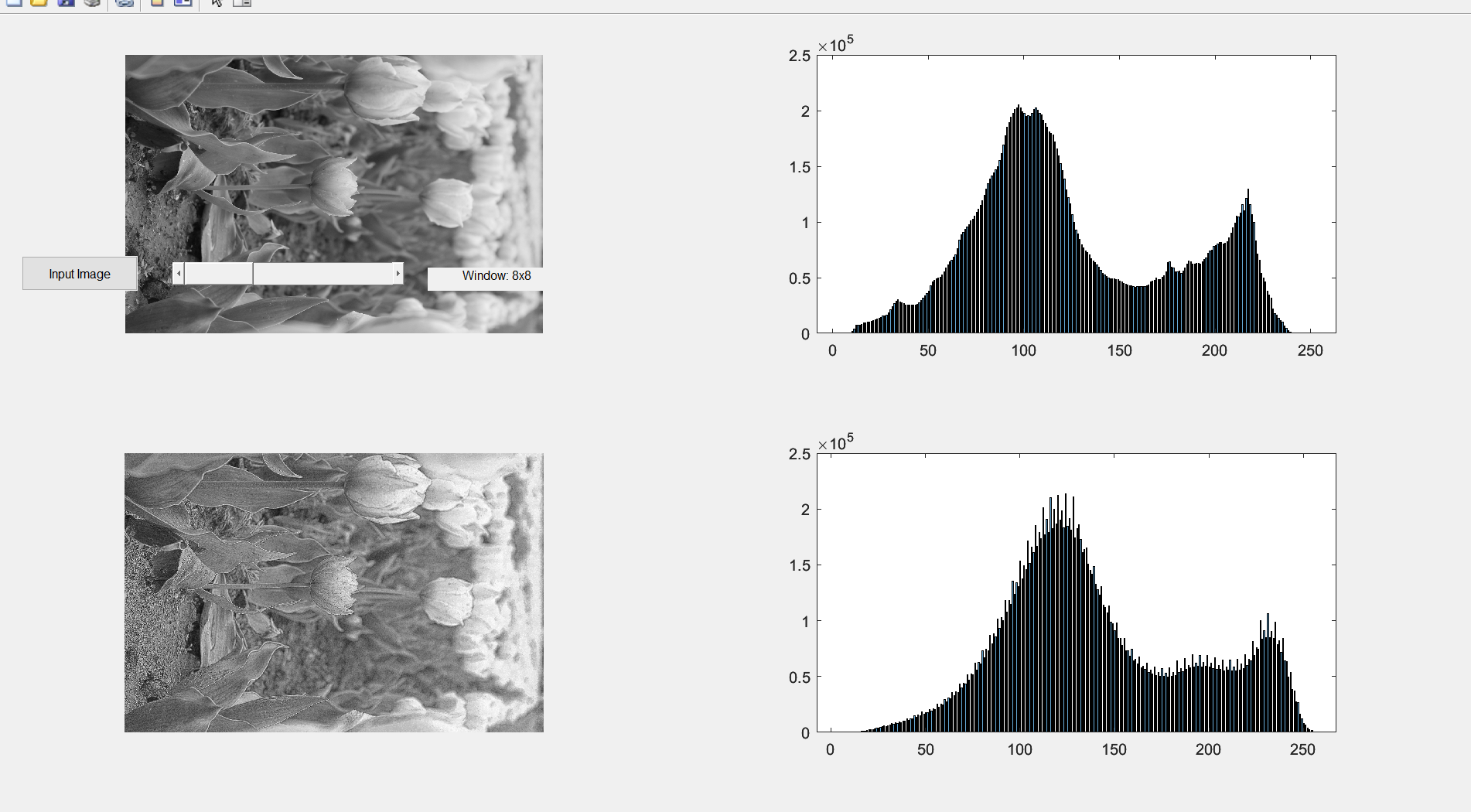
Input the image from the user (select from computer or drag and drop)🡪 after selecting the image it should Displays input image

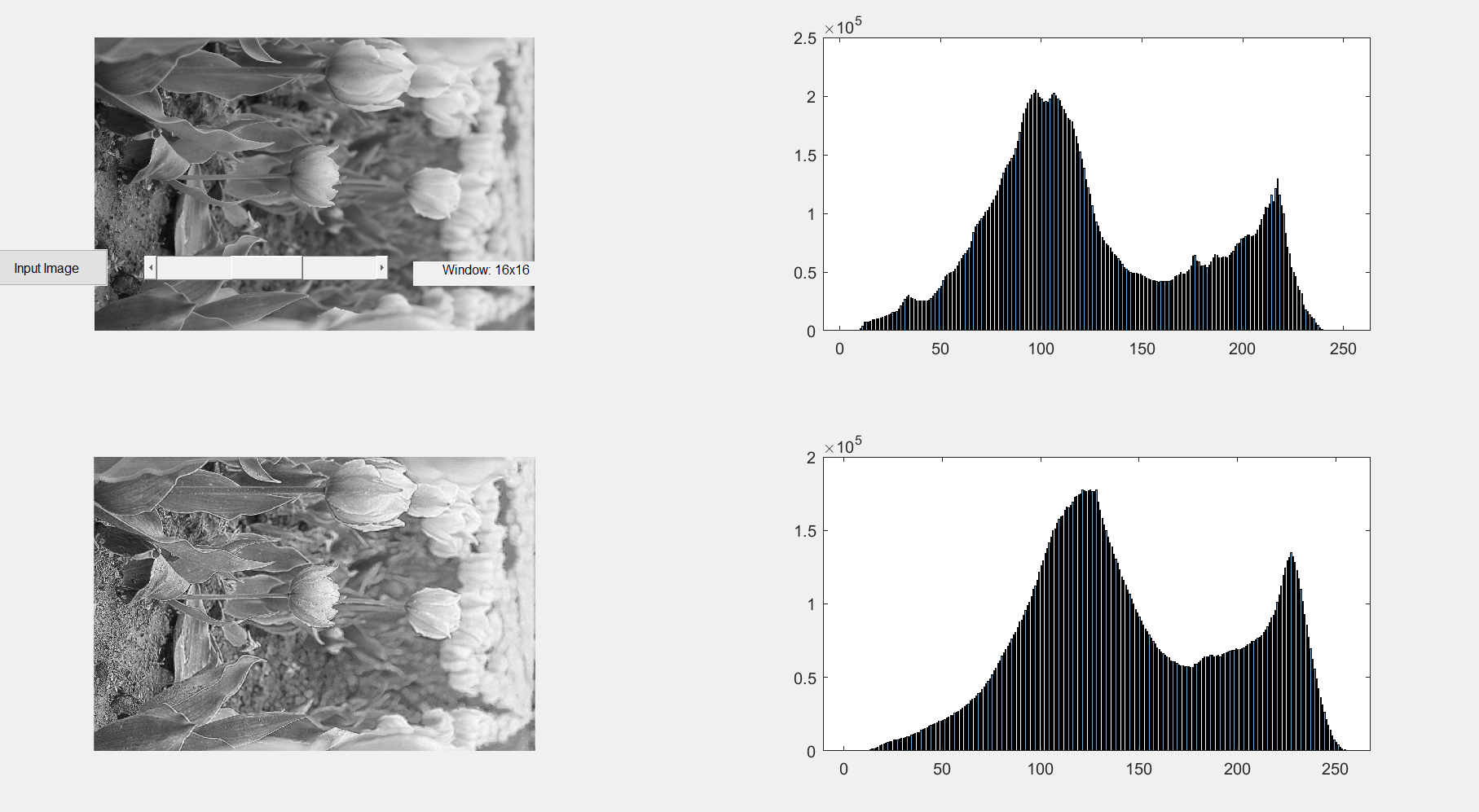
A process button to perform histogram

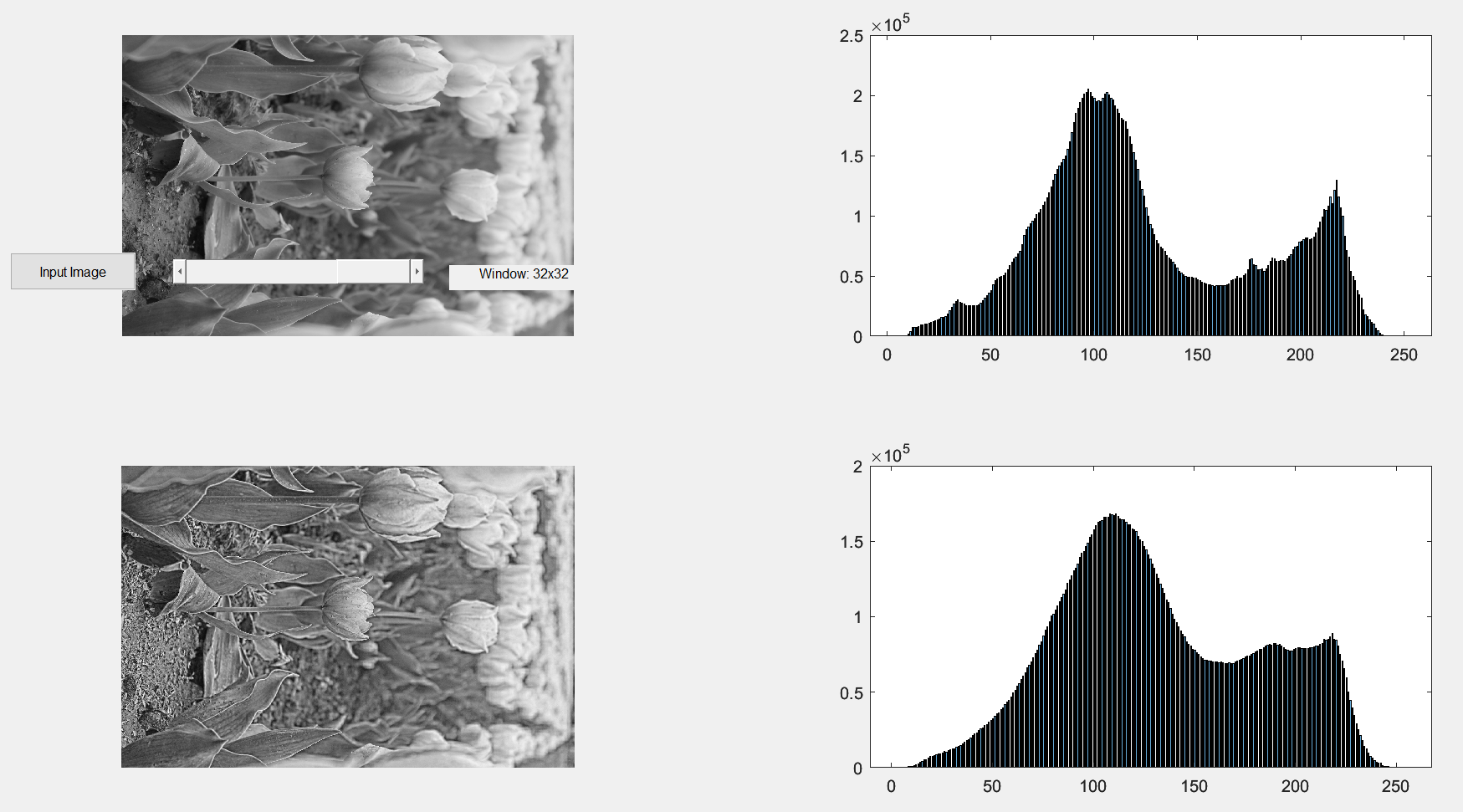
output image, histogram original graph, output histogram graph  
where output and the output graph should be changed depending on the input slider

save this output as an image button

| **Tile Size** | **Effect on Image** | **Histogram Shape** |
| --- | --- | --- |
| 8×8 | Fine-grained contrast | Spiky (more bins jump) |
| 16×16 | Balanced local/global | Moderately smooth |
| 32×32 | Global-like effect | Smoothest |
|  |  |  |







Yes, uicontrol is a **built-in MATLAB function** used to create **user interface controls** in a GUI, such as:

* Push buttons
* Sliders
* Text boxes
* Pop-up menus
* Checkboxes
* Radio buttons
* Editable fields

**🔧 Syntax example:**

uicontrol('Style','pushbutton','String','Click Me','Position',[100 100 80 30],'Callback',@buttonCallback);

This creates a **pushbutton** at position [100 100] with size [80 30] and assigns it a callback function to run when clicked.

**🔍 Common 'Style' options:**

| **Style** | **UI Element** |
| --- | --- |
| 'pushbutton' | Button |
| 'slider' | Slider bar |
| 'text' | Static label text |
| 'edit' | Editable text box |
| 'checkbox' | Checkbox |
| 'radiobutton' | Radio button |
| 'popupmenu' | Dropdown menu |

So yes — it's **built-in**, and central to making custom GUIs with **pure MATLAB code (not App Designer)**.

### ****General Format of**** uicontrol

matlab

CopyEdit

handle = uicontrol('Style', styleType, ...

'Position', [x y width height], ...

'String', 'Label or text', ...

'Callback', @callbackFunction, ...

'OtherProperty', value, ...);

### 🔍 Explanation of Common Properties:

| **Property** | **Description** |
| --- | --- |
| 'Style' | Type of UI element ('pushbutton', 'slider', 'edit', 'text', etc.) |
| 'Position' | [left bottom width height] in pixels within the parent container |
| 'String' | The label/text shown on the control (if applicable) |
| 'Callback' | Function to call when the control is interacted with (like click or slide) |
| 'Min'/'Max' | Used for sliders, sets range |
| 'Value' | Default value (for sliders, checkboxes, etc.) |

### 🧪 Examples

#### 1. Push Button

matlab

CopyEdit

btn = uicontrol('Style', 'pushbutton', ...

'String', 'Click Me', ...

'Position', [20 20 100 30], ...

'Callback', @myButtonCallback);

#### 2. Slider

matlab

CopyEdit

sld = uicontrol('Style', 'slider', ...

'Min', 1, 'Max', 10, 'Value', 5, ...

'SliderStep', [0.1 0.2], ...

'Position', [150 50 200 20], ...

'Callback', @mySliderCallback);

#### 3. Static Text Label

matlab

CopyEdit

lbl = uicontrol('Style', 'text', ...

'String', 'Current Value: 5', ...

'Position', [370 50 120 20]);

### 🧠 About Callback Syntax

You can use:

* @functionName → A direct reference to a local function
* @(src,event) functionName(src, event) → An anonymous function (useful if you want to pass extra data or fix argument mismatch)

**1. The Input Button (btn) - @loadImage Callback:**

This is a **simpler** approach for callback handling:

matlab

CopyEdit

btn = uicontrol('Style','pushbutton', ...

'String','Input Image', ...

'Position',[20 450 100 30], ...

'Callback',@loadImage);

Here’s what happens:

* **'Callback', @loadImage**:
  + The @loadImage refers to a **function handle**.
  + The function handle @loadImage means that MATLAB will automatically call the function loadImage when the button is clicked. You don’t need to pass any additional arguments, because the function is designed to work with the GUI context automatically.
  + MATLAB will automatically pass the necessary inputs to loadImage, such as the source of the event (in this case, the button).

The **callback function** loadImage might look like this:

matlab

CopyEdit

function loadImage(~,~)

% Your logic for loading an image

end

* **~ in function arguments** means that the function doesn’t use the input arguments (e.g., the button object). It’s just there to match the callback signature, but we don’t need to use it in the function body.

**Why is it simple here?**

* The callback function loadImage doesn’t need any extra arguments beyond what MATLAB provides automatically (the button object and event data). Hence, it’s just passed as a **simple function handle** (@loadImage).

**2. The Slider (sld) - @(src, event) Callback:**

In this case, the callback is written in a **slightly more complicated** way using an **anonymous function**:

matlab

CopyEdit

sld = uicontrol('Style','slider', ...

'Min',1,'Max',3,'Value',2, ...

'SliderStep',[0.5 0.5], ...

'Position',[150 455 200 20], ...

'Callback', @(src, event) sliderChanged);

Here’s the breakdown:

* **'Callback', @(src, event) sliderChanged**:
  + This is an **anonymous function**. Instead of passing a function handle directly like @loadImage, you're creating a function inline using @(src, event).
  + **@(src, event)**: This syntax defines an anonymous function that takes two inputs, src and event. These correspond to the slider object and event data, respectively.
  + **sliderChanged** is the function that you want to call when the slider is moved. The anonymous function **calls sliderChanged with the arguments src and event**.

In the **callback function** sliderChanged, you might do something like this:

matlab

CopyEdit

function sliderChanged(src, event)

% Use src (slider object) and event (event data) if necessary

% Logic to handle the slider change

end

* **src** refers to the slider object itself (the source of the callback).
* **event** contains information about the event, such as the slider's value or position when the callback is triggered.

**Why use an anonymous function here?**

* The sliderChanged function may need to work with the actual slider object (src) and event data (event) to update the GUI dynamically based on user input.
* The anonymous function is a way to pass these additional parameters (slider object and event data) to the callback.

### Main code

### *size(img, 3):* checks the third dimension of the image matrix img.

* + Images in MATLAB are stored as **matrices** with 3 dimensions: **height, width, and color channels**.
    - For **grayscale images**, the size would be something like [height, width] (2D matrix).
    - For **RGB images**, the size would be something like [height, width, 3], where 3 represents the three color channels: **Red, Green, and Blue**.
    - **if size(img, 3) == 3**: This condition checks if the image has **3 color channels** (i.e., it’s an RGB image). If img has 3 channels, then it is an **RGB image**, and the code inside the if statement is executed.
    - **img = rgb2gray(img);**: This converts the **RGB image** to a **grayscale image**.   
      rgb2gray is a MATLAB function that converts an RGB image to grayscale by applying a formula to combine the red, green, and blue channels into a single intensity value for each pixel.
* *val = round(get(data.slider, 'Value'));*

*winSizes = [8, 16, 32];*

*win = winSizes(val);*

Gets current slider position (1/2/3)

Maps it to 8, 16, or 32  
Updates label

* *enhanced = adapthisteq(data.img, 'NumTiles', numTiles, 'ClipLimit', 0.01);*

Applies local histogram equalization using adapthisteq  
Displays enhanced image and histogram  
**adapthisteq(data.img)**: This function performs adaptive histogram equalization on the image data.img. The idea is to apply histogram equalization to small local regions (tiles) of the image rather than globally across the whole image.  
**'NumTiles', numTiles**: The NumTiles parameter tells adapthisteq how many tiles (regions) to divide the image into for the local histogram equalization process. The numTiles vector specifies the number of tiles in the vertical and horizontal directions. 🡪This helps control the size and number of local regions over which the histogram equalization is applied.  
**'ClipLimit', 0.01**: The ClipLimit parameter is used to control the contrast enhancement. It prevents over-enhancement of local contrast in regions with high intensity. A lower value (like 0.01) will limit the enhancement to avoid excessively bright or dark regions. **If you see graininess**, try gradually **lowering the ClipLimit** (e.g., go from 0.05 → 0.01 → 0.001) and see if the graininess decreases.