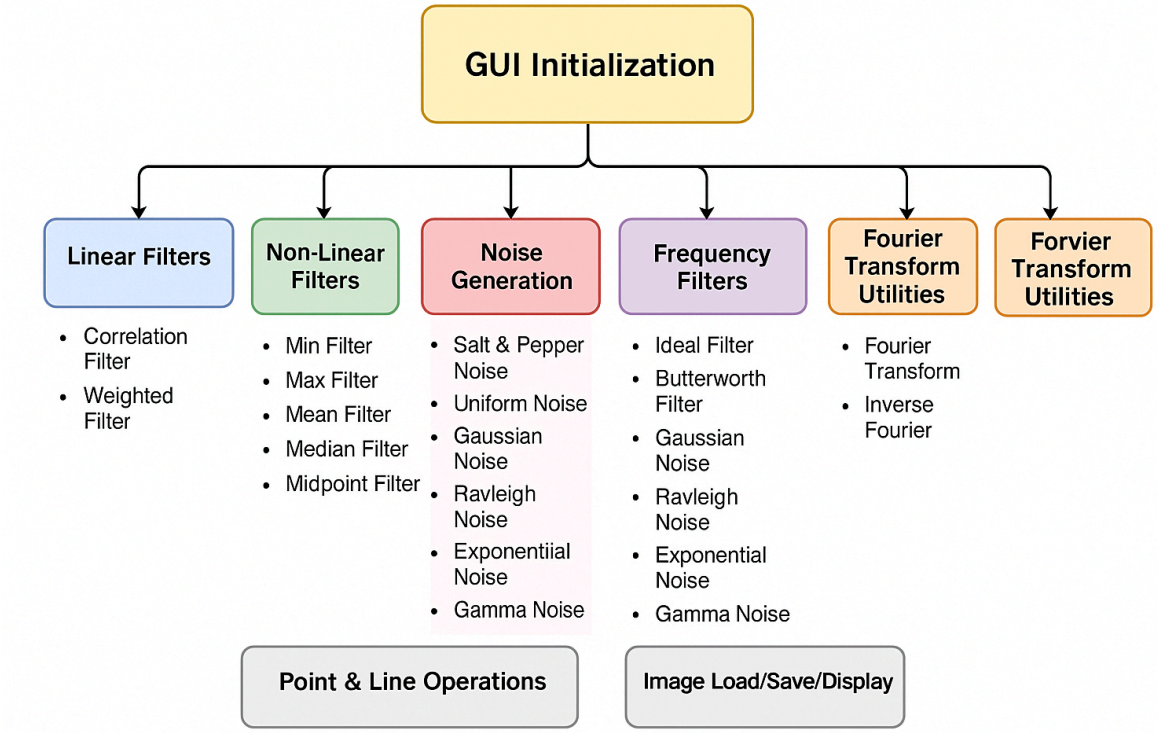
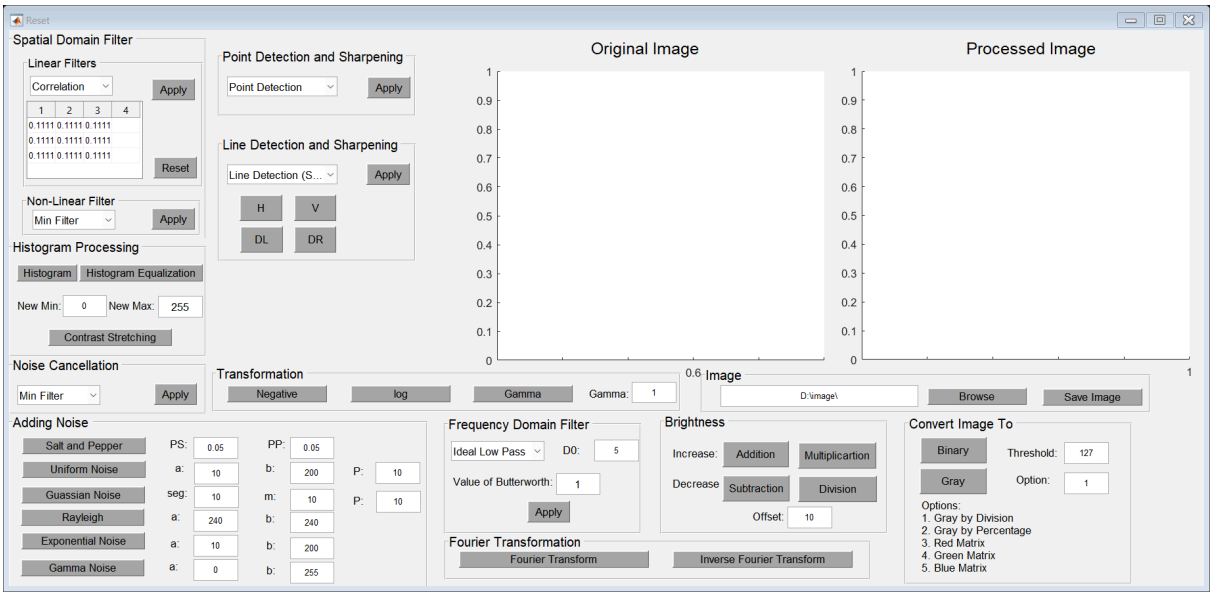
Image Processing Report





▼ 0️⃣ GUI & Functions

```
function varargout = gui(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @gui_OpeningFcn, ...
                   'gui_OutputFcn',  @gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
```

1. `function varargout = gui(varargin)`

This defines the main function of the GUI.

- `varargout` is a variable output argument, meaning it can return multiple values (used when the number of outputs varies).
- `varargin` is a variable input argument, meaning the function can accept a variable number of inputs.

4. `gui_Singleton = 1;`

This line sets a flag `gui_Singleton` to 1. The purpose of this is to ensure that only a single instance of the GUI can be opened at a time. If `gui_Singleton` is set to 1, the GUI will be a singleton.

5. `gui_State = struct('gui_Name', mfilename, ...`

This creates a structure `gui_State` that holds information about the GUI.

- `gui_Name`: This stores the name of the current file, obtained using `mfilename`. It is typically used for referencing the GUI's name.

5. `'gui_Singleton', gui_Singleton, ...`

This assigns the value of `gui_Singleton` (1, as set earlier) to the `gui_Singleton` field of the `gui_State` structure. This indicates that the GUI should be a singleton.

6. `'gui_OpeningFcn', @gui_OpeningFcn, ...`

This sets the `gui_OpeningFcn` field of `gui_State` to a function handle `@gui_OpeningFcn`. This refers to the function that is called when the GUI is first opened.

7. `'gui_OutputFcn', @gui_OutputFcn, ...`

This assigns the `gui_OutputFcn` field of `gui_State` to the function handle `@gui_OutputFcn`. This refers to the function that is responsible for returning output values from the GUI.

8. `'gui_LayoutFcn', [], ...`

This sets the `gui_LayoutFcn` field to an empty array. This is typically used for specifying the layout function for the GUI, but in this case, no layout function is specified.

9. `'gui_Callback', []);`

This sets the `gui_Callback` field to an empty array. This field can store a function handle that will be called when a certain event or callback happens (like a button press), but it is not used in this code.

10. `if nargin && ischar(varargin{1})`

This checks if there is at least one input argument (`nargin`) and if the first argument (`varargin{1}`) is a string (`ischar`). If these conditions are met, it proceeds with the code inside the `if` block.

11. `gui_State.gui_Callback = str2func(varargin{1});`

If the condition in the previous line is true, this line sets the `gui_Callback` field of the `gui_State` structure to a function handle created from the string stored in `varargin{1}` using `str2func`. This allows dynamic function assignment.

12. `if nargout`

This checks if there are any output arguments (`nargout`). If the number of output arguments is greater than zero, the code inside the `if` block is executed.

13. `[varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});`

If the number of output arguments is greater than zero, this line calls the main function of the GUI, `gui_mainfcn`, passing the `gui_State` structure and the input arguments (`varargin`). It returns the outputs in `varargout`.

14. `else`

This marks the start of the `else` block that will be executed if `nargout` is zero (i.e., no output arguments).

15. `gui_mainfcn(gui_State, varargin{:});`

If there are no output arguments, this line simply calls the `gui_mainfcn` without returning anything. It just executes the main GUI function.

```
% --- Executes just before gui is made visible.
function gui_OpeningFcn(hObject, ~, handles, varargin)
% Initialize filter options
set(handles.popupmenu1, 'String', {'Correlation', 'Weighted'});
set(handles.popupmenu1, 'Value', 1);
set(handles.edit1, 'String', '0');
set(handles.edit2, 'String', '255');

% Initialize default kernel
handles.filterKernel = ones(3,3)/9;
if isfield(handles, 'uitable3')
   set(handles.uitable3, 'Data', num2cell(handles.filterKernel));
end

% Initialize non-linear filter options
nonLinearFilters = {
   'Min Filter',
   'Max Filter',
   'Mean Filter',
   'Midpoint Filter',
   'Median Filter'
   };
set(handles.popupmenu3, 'String', nonLinearFilters);
```

```
set(handles.popupmenu3, 'Value', 1); % Default to first option

noiseFilters = {
    'Min Filter',
    'Max Filter',
    'Mean Filter',
    'Median Filter',
    'Midpoint Filter'
};

set(handles.popupmenu6, 'String', noiseFilters); % Assuming popupmenu3
set(handles.popupmenu6, 'Value', 1); % Default to first option

set(handles.edit7, 'String', '0.05');  % Salt & Pepper PS
set(handles.edit8, 'String', '0.05');  % Salt & Pepper PP
set(handles.edit9, 'String', '10');    % Uniform a
set(handles.edit10, 'String', '200');  % Uniform b
set(handles.edit11, 'String', '10');   % Uniform P
set(handles.edit12, 'String', '10');   % Gaussian seg
set(handles.edit13, 'String', '10');   % Gaussian m
set(handles.edit14, 'String', '10');   % Gaussian P
set(handles.edit17, 'String', '10');   % Exponential a
set(handles.edit18, 'String', '200');  % Exponential b
set(handles.edit19, 'String', '240');  % Rayleigh b
set(handles.edit20, 'String', '240');  % Rayleigh a
set(handles.edit21, 'String', '0');    % Gamma a
set(handles.edit22, 'String', '255');  % Gamma b
% Initialize image fields
handles.currentImage = [];
handles.filteredImage = [];
% Initialize point operations menu
pointOperations = {'Point Detection', 'Point Sharpening'};
set(handles.popupmenu7, 'String', pointOperations);
set(handles.popupmenu7, 'Value', 1); % Default to Point Detection
% Initialize line operations menu
lineOperations = {
    'Line Detection (Sobel)',
    'Line Detection (Roberts)',
```

```
    'Line Sharpening'
};
set(handles.popupmenu8, 'String', lineOperations);
set(handles.popupmenu8, 'Value', 1); % Default to Sobel

% Initialize direction buttons as invisible
set([handles.pushbutton31, handles.pushbutton32, handles.pushbutton33,
set(handles.pushbutton31, 'String', 'H');
set(handles.pushbutton32, 'String', 'V');
set(handles.pushbutton33, 'String', 'DL');
set(handles.pushbutton34, 'String', 'DR');
% Initialize transformation parameters
handles.transformationParams = struct(...
    'gammaValue', 1.0, ...      % Default gamma value
    'logConstant', 1.0, ...     % Constant for log transformation
    'negativeActive', false ...  % Flag for negative transformation
);

% Set default gamma value in the edit box
set(handles.edit23, 'String', num2str(handles.transformationParams.gamm
% Initialize frequency domain filter parameters
set(handles.edit26, 'String', '5');  % D0 cutoff frequency
set(handles.edit27, 'String', '1');  % Butterworth order

% Initialize filter type popupmenu
filterTypes = {
    'Ideal Low Pass',
    'Ideal High Pass',
    'Butterworth Low Pass',
    'Butterworth High Pass',
    'Gaussian Low Pass',
    'Gaussian High Pass'
};
set(handles.popupmenu10, 'String', filterTypes);
set(handles.popupmenu10, 'Value', 1); % Default to Ideal Low Pass
% Initialize brightness parameters
set(handles.edit28, 'String', '10');  % Default brightness adjustment value
handles.brightnessValue = 10;
```

```
% Choose default command line output
handles.output = hObject;
handles.currentImage = [];
handles.noisyImage = [];
handles.frequencyFilteredImage = [];
% Initialize Fourier transform variables
handles.fourierData = [];       % Will store Fourier transform data
handles.reconstructedImage = []; % Will store reconstructed image
% Initialize conversion parameters
set(handles.edit29, 'String', '127');  % Default binary threshold
set(handles.edit30, 'String', '1');    % Default grayscale method (1-5)
% Initialize other variables
handles.currentImage = [];
handles.processedImage = [];
% Initialize default values for Fourier display
handles.fourierDisplayMode = 'magnitude'; % Can be 'magnitude' or 'phas
% Initialize kernel size (default 3×3)
handles.kernelSize = 3;

% Update handles structure
guidata(hObject, handles);
```

🔷 Function Header

```
function gui_OpeningFcn(hObject, ~, handles, varargin)
```

This is the GUI Opening Function. It's called just before the GUI window becomes visible.

- `hObject`: handle to the GUI figure
- `handles`: structure containing handles to GUI components
- `varargin`: any additional parameters passed when launching the GUI

🧪 1. Correlation & Weighted Filter Initialization

```
set(handles.popupmenu1, 'String', {'Correlation', 'Weighted'});
set(handles.popupmenu1, 'Value', 1);
```

```
set(handles.edit1, 'String', '0');
set(handles.edit2, 'String', '255');
```

- Initializes the first popup menu (`popupmenu1`) with filter options.
- Default selected value is **Correlation**.
- `edit1` and `edit2` are set to grayscale intensity range (0–255).

📦 2. Linear Filter Kernel Initialization

```
handles.filterKernel = ones(3,3)/9;
if isfield(handles, 'uitable3')
    set(handles.uitable3, 'Data', num2cell(handles.filterKernel));
end
```

- Sets a **default 3×3 average kernel** (used for linear filtering).
- Displays it in `uitable3` (if the table exists).

🌀 3. Non-Linear Filter Options

```
nonLinearFilters = {
    'Min Filter',
    'Max Filter',
    'Mean Filter',
    'Midpoint Filter',
    'Median Filter'
};
set(handles.popupmenu3, 'String', nonLinearFilters);
set(handles.popupmenu3, 'Value', 1);
```

- Initializes **non-linear filter options** (like Min, Max, Median).
- Sets default to **Min Filter** in `popupmenu3`.

---

4. Noise Filter Options

```
noiseFilters = {
    'Min Filter',
    'Max Filter',
    'Mean Filter',
    'Median Filter',
    'Midpoint Filter'
};
set(handles.popupmenu6, 'String', noiseFilters);
set(handles.popupmenu6, 'Value', 1);
```

- Similar to non-linear filters, possibly reused for **denoising filters**.
- `popupmenu6` is set to **Min Filter** by default.

🧂 5. Noise Parameters (Salt & Pepper, Uniform, Gaussian, etc.)

```
set(handles.edit7, 'String', '0.05');  % Salt & Pepper PS
set(handles.edit8, 'String', '0.05');  % Salt & Pepper PP
set(handles.edit9, 'String', '10');    % Uniform a
set(handles.edit10, 'String', '200');  % Uniform b
set(handles.edit11, 'String', '10');   % Uniform P
set(handles.edit12, 'String', '10');   % Gaussian seg
set(handles.edit13, 'String', '10');   % Gaussian m
set(handles.edit14, 'String', '10');   % Gaussian P
set(handles.edit17, 'String', '10');   % Exponential a
set(handles.edit18, 'String', '200');  % Exponential b
set(handles.edit19, 'String', '240');  % Rayleigh b
set(handles.edit20, 'String', '240');  % Rayleigh a
set(handles.edit21, 'String', '0');    % Gamma a
set(handles.edit22, 'String', '255');  % Gamma b
```

- Initializes **parameters for adding synthetic noise**.
- Each pair of edits corresponds to a specific distribution type.

🖼️ 6. Image Placeholders

```
handles.currentImage = [];
handles.filteredImage = [];
```

- Prepares fields to store the **original and filtered images**.

🔍 7. Point Operations Menu

```
pointOperations = {'Point Detection', 'Point Sharpening'};
set(handles.popupmenu7, 'String', pointOperations);
set(handles.popupmenu7, 'Value', 1);
```

- Provides a menu for **point-based image operations**.

📏 8. Line Operations Menu

```
lineOperations = {
    'Line Detection (Sobel)',
    'Line Detection (Roberts)',
    'Line Sharpening'
};
set(handles.popupmenu8, 'String', lineOperations);
set(handles.popupmenu8, 'Value', 1);
```

- Provides **line detection/sharpening methods**.
- Default is **Sobel** operator.

↘️ 9. Direction Buttons (for directional filtering)

```
set([handles.pushbutton31, handles.pushbutton32, handles.pushbutton3
3, handles.pushbutton34], 'Visible', 'on','Enable', 'on');
set(handles.pushbutton31, 'String', 'H');
set(handles.pushbutton32, 'String', 'V');
set(handles.pushbutton33, 'String', 'DL');
set(handles.pushbutton34, 'String', 'DR');
```

- Enables buttons for **direction selection**:
  - H: Horizontal
  - V: Vertical
  - DL: Diagonal Left
  - DR: Diagonal Right

🔁 10. Transformation Parameters

```
handles.transformationParams = struct(...
    'gammaValue', 1.0, ...
    'logConstant', 1.0, ...
    'negativeActive', false ...
);
set(handles.edit23, 'String', num2str(handles.transformationParams.ga
mmaValue));
```

- Initializes default parameters for image transformations like:
  - Gamma correction
  - Log transformation
  - Negative transformation

🧱 11. Frequency Domain Filters

```
set(handles.edit26, 'String', '5');  % D0
set(handles.edit27, 'String', '1');  % Butterworth order

filterTypes = {
    'Ideal Low Pass',
    'Ideal High Pass',
    'Butterworth Low Pass',
    'Butterworth High Pass',
    'Gaussian Low Pass',
    'Gaussian High Pass'
};
```

```
set(handles.popupmenu10, 'String', filterTypes);
set(handles.popupmenu10, 'Value', 1);
```

- Initializes **frequency domain filter** parameters:
  - Cutoff frequency `D0`
  - Filter type selector (`popupmenu10`)

💡 12. Brightness Settings

```
set(handles.edit28, 'String', '10');
handles.brightnessValue = 10;
```

- Sets default **brightness adjustment** value to 10.

🔄 13. More Image Placeholders

```
handles.output = hObject;
handles.currentImage = [];
handles.noisyImage = [];
handles.frequencyFilteredImage = [];
```

- Additional fields to hold different image states.

🔬 14. Fourier Variables

```
handles.fourierData = [];
handles.reconstructedImage = [];
```

- Placeholder for **Fourier Transform result** and **inverse** image.

🎛️ 15. Image Conversion Parameters

```
set(handles.edit29, 'String', '127');  % Threshold for binary conversion
set(handles.edit30, 'String', '1');    % Grayscale conversion method (1 to
```

```
5)
```

- For converting the image to binary or grayscale using specified methods.

📂 16. Final Setup

```
handles.currentImage = [];
handles.processedImage = [];
handles.fourierDisplayMode = 'magnitude';  % Default Fourier display
handles.kernelSize = 3;                    % Default filter kernel size
```

🔁 17. Update `handles` Structure

```
guidata(hObject, handles);
```

- Saves the modified `handles` back into the GUI's internal data.
- Necessary for the changes to persist.

✅ Summary Table

Group	Purpose
Filter Menus	Correlation, Weighted, Non-linear filters
Noise Inputs	Salt & Pepper, Gaussian, Rayleigh, etc.
Line/Point Menus	Point/Line detection & sharpening
Transformation	Gamma, Log, Negative settings
Fourier Filters	Ideal, Butterworth, Gaussian
Direction Buttons	Horizontal, Vertical, Diagonals
Image Data	Stores original, noisy, processed images
Display & Conversion	Brightness, threshold, display modes

▼ 2️⃣ Linear Filters

```
% --- Outputs from this function are returned to the command line.
function varargout = gui\_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;

% --- Helper function to validate numeric input
function valid = validateInput(handle, minVal, maxVal, isInteger)
str = get(handle, 'String');
num = str2double(str);
if isnan(num) || num < minVal || num > maxVal
valid = false;
set(handle, 'String', '');
errordlg(sprintf('Please enter a number between %.2f and %.2f', minVal, m
else
if isInteger
num = round(num);
set(handle, 'String', num2str(num));
end
valid = true;
end

% --- Executes on selection change in popupmenu1.
function popupmenu1\_Callback(hObject, eventdata, handles)
set(handles.uitable3, 'Data', handles.filterKernel);

% --- Executes during object creation, after setting all properties.
function popupmenu1\_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolB
set(hObject,'BackgroundColor','white');
end

% --- Executes on selection change in listbox1.
%function listbox1\_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function listbox1\_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolB
set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on button press in apply.
function apply\_Callback(hObject, eventdata, handles)
% Check if image exists
if \~isfield(handles, 'currentImage') || isempty(handles.currentImage)
errordlg('Please load an image first using the "Load Image" button.', 'No Im
return;
end

try
% Initialize default kernel if not exists
if \~isfield(handles, 'filterKernel')
handles.filterKernel = ones(3,3)/9;
end
```

```
% Get selected filter type
filterTypes = get(handles.popupmenu1, 'String');
selectedFilter = filterTypes{get(handles.popupmenu1, 'Value')};

% Get kernel data safely
if isfield(handles, 'uitable3')
    kernelData = get(handles.uitable3, 'Data');
    if iscell(kernelData)
        % Convert cell to matrix, handling empty/non-numeric values
        kernelData = cellfun(@(x) ifelse(isempty(x)||~isnumeric(x),0,x), kernel
        handles.filterKernel = cell2mat(kernelData);
    else
        handles.filterKernel = kernelData;
    end
end

% Use filtered image if it exists, otherwise use original
if isfield(handles, 'filteredImage') && ~isempty(handles.filteredImage)
    img = handles.filteredImage;
else
    img = handles.currentImage;
end
```

```
% Process image
if size(img, 3) == 3
    img = rgb2gray(img);
end

% Apply selected filter
switch selectedFilter
    case 'Correlation'
        filteredImg = imfilter(double(img), handles.filterKernel, 'corr', 'same');
    case 'Weighted'
        % Normalize kernel
        normalizedKernel = handles.filterKernel / sum(handles.filterKernel(:));
        filteredImg = imfilter(double(img), normalizedKernel, 'conv', 'same');
end

% Always show original in axes1
axes(handles.axes1);
imshow(handles.currentImage);
title('Original Image');

% Show filtered result in axes2
axes(handles.axes2);
imshow(filteredImg, []);
title([selectedFilter ' Result']);

% Store for next operation
handles.filteredImage = filteredImg;
guidata(hObject, handles);
```

```
catch e
errordlg(\['Error applying filter: ' e.message], 'Filter Error');
end

% Helper function
function y = ifelse(condition, trueval, falseval)
if condition
```

```
y = trueval;
else
y = falseval;
end
```

🔷 1. GUI Output Function

This function returns the main output of the GUI (typically the `handles.output`) to the command line.

```
function varargout = gui_OutputFcn(hObject, eventdata, handles)
    varargout{1} = handles.output;
end
```

🔷 2. Validation Helper Function

Validates the input from a UI text box (used when entering numeric values in a kernel, for example).

```
function valid = validateInput(handle, minVal, maxVal, isInteger)
    str = get(handle, 'String');            % Get the input as a string
    num = str2double(str);                  % Convert to number
    if isnan(num) || num < minVal || num > maxVal
        valid = false;
        set(handle, 'String', '');
        errordlg(sprintf('Please enter a number between %.2f and %.2f', mi
nVal, maxVal), 'Invalid Input');
    else
        if isInteger                        % Round if required
            num = round(num);
            set(handle, 'String', num2str(num));
        end
        valid = true;
    end
end
```

🔷 3. Dropdown (Popup Menu) Logic

Used to choose the filter type (Correlation or Weighted). Also sets the kernel data in the UI table.

```
function popupmenu1_Callback(hObject, eventdata, handles)
    set(handles.uitable3, 'Data', handles.filterKernel);
end
```

Creates the popup menu (dropdown list) with a white background.

```
function popupmenu1_CreateFcn(hObject, eventdata, handles)
    if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicon
trolBackgroundColor'))
        set(hObject,'BackgroundColor','white');
    end
end
```

🔷 5. Apply Filter Button Logic

This is where most of the filtering logic is implemented. It gets triggered when the **"Apply"** button is clicked.

✅ General Steps:

1. Check if an image is loaded.
2. Get the selected filter (Correlation or Weighted).
3. Get and validate the kernel matrix from `uitable3`.
4. Convert image to grayscale if needed.
5. Apply the selected filter.
6. Display original and filtered images.

🔶 1. Check if an image is loaded

```
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first using the "Load Image" button.',
'No Image Found');
```

```
    return;
end
```

✅ What this does:

- Checks if the `handles` structure has a field called `currentImage` **and** that it's not empty.
- If not, it shows an error dialog to the user saying **"No Image Found"**.
- `return` exits the callback function early to prevent errors (you can't filter a nonexistent image).

🔶 2. Create a default filter kernel if one doesn't exist

```
if ~isfield(handles, 'filterKernel')
    handles.filterKernel = ones(3,3)/9;
end
```

✅ What this does:

- Checks if a field called `filterKernel` exists in the `handles` structure.
- If not, it creates a **3×3 average kernel** (each value is `1/9`).
- This kernel is often used for **blurring or smoothing**.

🔶 3. Get the selected filter name from the popup menu

```
filterTypes = get(handles.popupmenu1, 'String');
selectedFilter = filterTypes{get(handles.popupmenu1, 'Value')};
```

✅ What this does:

- Retrieves the list of available filters from `popupmenu1` (like `'Correlation'`, `'Weighted'`, etc.).
- Gets the currently selected filter using `get(..., 'Value')`.
- Stores it in the variable `selectedFilter`.

For example:

If the dropdown has: `{'Correlation', 'Weighted'}` and the user chooses the second one, `selectedFilter` will be `'Weighted'`.

🔶 4. Read the kernel matrix from the UI table (`uitable3`)

```
if isfield(handles, 'uitable3')
    kernelData = get(handles.uitable3, 'Data');
```

- This checks if the `uitable3` field exists (i.e., the GUI contains a table for kernel editing).
- Retrieves the **data inside the table**, which could be a cell array or numeric matrix.

🔶 4.a. If kernel is a cell (like editable table), convert it

```
if iscell(kernelData)
    kernelData = cellfun(@(x) ifelse(isempty(x)||~isnumeric(x), 0, x), kern
elData, 'UniformOutput', false);
    handles.filterKernel = cell2mat(kernelData);
```

✅ What this does:

- `cellfun` goes through each element of the table.
- If the cell is **empty or not numeric**, it's replaced with `0`.
- Otherwise, it keeps the number.
- The result is converted into a numeric matrix using `cell2mat`.

This is important because `uitable` allows any kind of input (strings, empty), but filters need a pure **numeric matrix**.

🔶 4.b. If it's already a numeric matrix

```
else
    handles.filterKernel = kernelData;
end
```

- If the table data was already numeric, no conversion is needed.

🔶 5. Choose which image to apply the filter to

```
if isfield(handles, 'filteredImage') && ~isempty(handles.filteredImage)
    img = handles.filteredImage;
else
    img = handles.currentImage;
end
```

✅ What this does:

- If there's a previously filtered image (`handles.filteredImage`), use that as the input for the new filter.
- Otherwise, use the original image (`handles.currentImage`).
- This allows **chaining multiple filters** together.

🔶 6. Convert color image to grayscale (if necessary)

```
if size(img, 3) == 3
    img = rgb2gray(img);
end
```

✅ What this does:

- Checks if the image has 3 color channels (RGB).
- If so, converts it to **grayscale**, because most basic filters work on single-channel images.
- `rgb2gray()` reduces an RGB image to shades of gray using a weighted sum of R, G, and B.

✅ Summary of Workflow:

Step	Purpose
1️⃣	Check if the image is loaded
2️⃣	Set a default kernel if none is provided
3️⃣	Read the selected filter name

4️⃣	Safely read and clean the kernel from the UI table
5️⃣	Use the latest filtered image (if exists), otherwise use original
6️⃣	Convert image to grayscale if it's colored

🟧 Filter Type 1: Correlation

```
case 'Correlation'
    filteredImg = imfilter(double(img), handles.filterKernel, 'corr', 'same');
```

- `imfilter` applies the **correlation operation** using the provided kernel.
- `'corr'` means no flipping of kernel (standard correlation).
- `'same'` keeps output size equal to input size.

🟦 Filter Type 2: Weighted Average (Normalized Convolution)

```
case 'Weighted'
    normalizedKernel = handles.filterKernel / sum(handles.filterKernel(:));
    filteredImg = imfilter(double(img), normalizedKernel, 'conv', 'same');
```

- Kernel is **normalized** (each element is divided by the total sum).
- `'conv'` means convolution (kernel is flipped).
- Often used for **blurring/smoothing** operations.

🔷 Display and Store the Output

```
% Display original image
axes(handles.axes1);
imshow(handles.currentImage);
title('Original Image');

% Display filtered image
axes(handles.axes2);
imshow(filteredImg, []);
title([selectedFilter ' Result']);
```

```
% Save filtered image to handles
handles.filteredImage = filteredImg;
guidata(hObject, handles);
```

🔷 6. Helper Function - ifelse

Simplified version of ternary logic used during kernel value checks.

MATLAB doesn't have a built-in ternary operator

That’s why this function is created manually to fill that role.

```
function y = ifelse(condition, trueval, falseval)
    if condition
        y = trueval;
    else
        y = falseval;
    end
end
```

✅ Summary by Filter:

Filter Type	Logic Used	Notes
Correlation	`imfilter(..., 'corr', 'same')` using raw kernel	Used for pattern detection, edge-like operations
Weighted	Normalized kernel + `imfilter(..., 'conv', 'same')`	Used for smoothing, blurring

```
% --- Executes when entered data in editable cell(s) in uitable3.
function uitable3_CellEditCallback(hObject, eventdata, handles)
try
    newData = get(hObject, 'Data');
    % Convert from cell if needed
    if iscell(newData)
        newData = cell2mat(newData);
    end
    handles.filterKernel = newData;
    guidata(hObject, handles);
```

```
catch e
    errordlg(['Invalid kernel value: ' e.message], 'Input Error');
end
```

🔷 Purpose:

This function runs **whenever the user edits any cell** inside the GUI table component `uitable3`, which is used to represent the filter kernel (like for correlation or weighted filtering).

🔶 Detailed Line-by-Line Explanation:

Line	Explanation
`function uitable3_CellEditCallback(...)`	Declares a callback function triggered when a cell in `uitable3` is edited.
`newData = get(hObject, 'Data');`	Retrieves the updated data from the table. The data is likely a **cell array** if edited manually.
`if iscell(newData)`	Checks if the retrieved data is a **cell array** (which it is when you edit uitable directly).
`newData = cell2mat(newData);`	Converts the cell array to a numeric matrix, needed for matrix operations like filtering.
`handles.filterKernel = newData;`	Stores the converted kernel into the global `handles` structure under the field `filterKernel`. This is essential to apply the custom kernel later.
`guidata(hObject, handles);`	Updates the `handles` structure in the GUI so the new kernel value is accessible by other callbacks (e.g., apply filter).
`catch e ...`	If anything fails (like a non-numeric cell), shows an error dialog box with a custom error message.

🧩 Filter Group:

This function belongs to the **Linear Filters Group**:

Filter Group	Components
Linear Filters	- Correlation Filter- Weighted Filter

Related Components	- `uitable3` for kernel input- `popupmenu1` for selecting filter- `apply_Callback` for execution

▼ 3️⃣Non-Linear Filters (Spatial Domain) implementation

🟩 Fixed 3x3 Non-Linear Filter

🔘 `pushbutton30_Callback` (always uses 3×3)

```
% --- Executes on button press in pushbutton30.
function pushbutton30_Callback(hObject, eventdata, handles)
    if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
        errordlg('Please load an image first.', 'No Image Found');
        return;
    end

    % Get selected filter from popupmenu3
    val = get(handles.popupmenu3, 'Value');

    % Use filtered image if it exists, otherwise original
    if isfield(handles, 'filteredImage') && ~isempty(handles.filteredImage)
        img = handles.filteredImage;
    else
        img = handles.currentImage;
    end

    % Convert to grayscale if RGB
    if size(img, 3) == 3
        img = rgb2gray(img);
    end

    switch val
        case 1  % Min filter
            imgOut = ordfilt2(img, 1, ones(3));
        case 2  % Max filter
            imgOut = ordfilt2(img, 9, ones(3));
```

```
        case 3  % Median filter
            imgOut = medfilt2(img, [3 3]);
        case 4  % Mean filter
            h = fspecial('average', [3 3]);
            imgOut = imfilter(img, h);
        case 5  % Midpoint filter
            minImg = ordfilt2(img, 1, ones(3));
            maxImg = ordfilt2(img, 9, ones(3));
            imgOut = (double(minImg) + double(maxImg)) / 2;
            imgOut = uint8(imgOut);
        otherwise
            errordlg('Invalid filter selection.');
            return;
    end

    axes(handles.axes2);
    imshow(imgOut);
    title('Filtered Image');

    handles.filteredImage = imgOut;
    guidata(hObject, handles);

    catch e
    errordlg(['Error applying ' selectedFilter ': ' e.message], 'Filter Error');

end
```

This function runs when the user clicks the button labeled (likely) "Apply Fixed Filter".

```
matlab
Copy code
% --- Executes on button press in pushbutton30.
function pushbutton30_Callback(hObject, eventdata, handles)
```

👉 This is the function header. It's automatically triggered when `pushbutton30` is pressed.

- `hObject`: the handle to the button object.
- `eventdata`: not used here, but stores event data.
- `handles`: structure storing GUI data and handles.

🔍 Check for Image Availability

```
matlab
Copy code
% Check if image exists
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first.', 'No Image Found');
    return;
end
```

✔️ Ensures that an image is loaded before proceeding.

- If `handles.currentImage` doesn't exist or is empty, an error dialog appears and the function exits.

🧠 Get Selected Filter from Dropdown

```
matlab
Copy code
% Get selected filter
filterList = get(handles.popupmenu3, 'String');
selectedFilter = filterList{get(handles.popupmenu3, 'Value')};
```

💡 This part reads the filter name chosen in `popupmenu3`, which contains options like:

- Min Filter
- Max Filter
- Mean Filter

- Median Filter
- Midpoint Filter

It does this in two steps:

- `get(..., 'String')` retrieves the **cell array** of all dropdown entries.
- `get(..., 'Value')` retrieves the **index** of the currently selected item.
- Combining them gives the selected string (e.g., `'Median Filter'`).

🖼️ Get Current Image (Filtered or Original)

```
matlab
Copy code
try
    % Use filtered image if it exists, otherwise use original
    if isfield(handles, 'filteredImage') && ~isempty(handles.filteredImage)
        img = handles.filteredImage;
    else
        img = handles.currentImage;
    end
```

✔️ Uses the **latest filtered image** if it exists, otherwise falls back to the original image (`currentImage`).

🌈 Convert to Grayscale

```
matlab
Copy code
    % Convert to grayscale if RGB
    if size(img, 3) == 3
        img = rgb2gray(img);
    end
```

✔️ If the image is colored (RGB with 3 channels), convert it to grayscale.

Most spatial filters work on single-channel images.

🧪 Apply the Selected Filter (Fixed 3×3)

```
matlab
Copy code
    % Apply selected filter
    switch selectedFilter
```

🔶 Min Filter

```
matlab
Copy code
        case 'Min Filter'
            filteredImg = ordfilt2(img, 1, ones(3,3));
```

🟢 Takes the **minimum value** in a 3×3 neighborhood using `ordfilt2`.

🔶 Max Filter

```
matlab
Copy code
        case 'Max Filter'
            filteredImg = ordfilt2(img, 9, ones(3,3));
```

🟢 Takes the **maximum value** (9th in sorted list of 9 pixels).

🔶 Mean Filter

```
matlab
Copy code
        case 'Mean Filter'
            h = fspecial('average', 3);
            filteredImg = imfilter(img, h);
```

🟢 Creates a 3×3 averaging kernel and applies it using `imfilter`.

🔶 Midpoint Filter

```
matlab
Copy code
        case 'Midpoint Filter'
            minImg = ordfilt2(img, 1, ones(3,3));
            maxImg = ordfilt2(img, 9, ones(3,3));
            filteredImg = (double(minImg) + double(maxImg)) / 2;
            filteredImg = uint8(filteredImg);
```

🟢 Calculates the **average of min and max** values in the window.

- Converts to `double` to avoid overflow.
- Result is converted back to `uint8`.

🔶 Median Filter

```
matlab
Copy code
        case 'Median Filter'
            filteredImg = medfilt2(img, [3 3]);
```

🟢 Uses `medfilt2` for **median filtering** in a 3×3 window.

🖼️ Display Images

```
matlab
Copy code
    % Always show original in axes1
    axes(handles.axes1);
    imshow(handles.currentImage);
    title('Original Image');
```

Displays the **original image** in `axes1`.

```
matlab
Copy code
    % Show filtered result in axes2
    axes(handles.axes2);
    imshow(filteredImg);
    title([selectedFilter ' Result']);
```

Displays the **filtered image** in `axes2` with a title showing the filter name.

💾 Save Filtered Image

```
matlab
Copy code
    % Store for next operation
    handles.filteredImage = filteredImg;
    guidata(hObject, handles);
```

✔️ Saves the filtered image in `handles` for further processing or chaining operations.

⚠️ Error Handling

```
matlab
Copy code
catch e
    errordlg(['Error applying ' selectedFilter ': ' e.message], 'Filter Error');
end
```

🛑 If an error occurs at any point in the `try` block, display an error dialog with the error message.

🧾 Dropdown Initialization

📌 Inside `gui_OpeningFcn`

```
nonLinearFilters = {
    'Min Filter',
    'Max Filter',
    'Mean Filter',
    'Midpoint Filter',
    'Median Filter'
};
set(handles.popupmenu3, 'String', nonLinearFilters);
set(handles.popupmenu3, 'Value', 1); % Default
```

🔗 UI Elements and Their Roles

🔘 `popupmenu3`

- Dropdown menu for selecting the **non-linear filter**.
- Triggered When Filter Selection Changes
- The selected value is obtained using:

```
val = get(handles.popupmenu3, 'Value');
```

⚙️ `popupmenu3_CreateFcn` – Initialization

```
% --- Executes during object creation, after setting all properties.
function popupmenu3_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontr
olBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

🖥️ Sets the background color of the popup menu to white if the platform is Windows and the color is still default.

✅ Summary Table

Filter	Function	Kernel Size	Implementation Method

Min	`ordfilt2(img, 1, ...)`	User: `k×k` or Fixed: `3×3`	Non-linear Smoothing
Max	`ordfilt2(img, k^2, ...)`	User or Fixed (k^2=9)	Non-linear Edge Enhancing
Median	`medfilt2(img, [...])`	User or Fixed	Salt-and-pepper noise removal
Mean	`fspecial + imfilter`	User or Fixed	Smoothing / Averaging
Midpoint	Average of Min/Max	User or Fixed	Contrast Enhancement

Histogram Processing

🟩 1.📊 Histogram Comparison `pushbutton8_Callback`

```
function pushbutton8_Callback(hObject, eventdata, handles)
```

🔷 This function is called when the "Show Histogram Comparison" button is clicked.

📌 Input Image Checks

```
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first.', 'No Image Found');
    return;
end
```

✔️ Confirms that an original image is loaded. If not, shows an error and exits.

```
if ~isfield(handles, 'filteredImage') || isempty(handles.filteredImage)
    errordlg('No processed image available. Please apply a filter first.', 'N
o Processed Image');
    return;
end
```

✔️ Confirms that the user has already applied a filter (or processed the image). If not, exits.

🖼️ Setup & Display

```
originalImg = handles.currentImage;
filteredImg = handles.filteredImage;

hFig = figure('Name', 'Histogram Comparison', 'NumberTitle', 'off', 'Posit
ion', [100 100 1000 800]);
```

🟩 Creates a new figure window titled "Histogram Comparison" with a size of 1000×800 pixels.

◼️ Original Image Display

```
subplot(2,2,1);
if size(originalImg, 3) == 3
    imshow(originalImg);
    title('Original Color Image');
```

🔶 Shows the original image (color if 3 channels) in the top-left panel.

```
    subplot(2,2,3);
    imhist(originalImg(:,:,1)); hold on;
    imhist(originalImg(:,:,2));
    imhist(originalImg(:,:,3)); hold off;
    title('Original RGB Histograms');
    legend('Red','Green','Blue');
```

🔶 Shows R, G, B channel histograms in the bottom-left panel.

```
else
    imshow(originalImg);
    title('Original Grayscale Image');

    subplot(2,2,3);
    imhist(originalImg);
```

```
    title('Original Histogram');
end
```

🟪 If the original is grayscale, just display its image and grayscale histogram.

🔲 Processed Image Display

```
subplot(2,2,2);
imshow(filteredImg);
title('Processed Image');
```

🟨 Shows the processed image (after filtering, equalization, or contrast stretching) in the top-right.

```
subplot(2,2,4);
if size(filteredImg, 3) == 3
    imhist(filteredImg(:,:,1)); hold on;
    imhist(filteredImg(:,:,2));
    imhist(filteredImg(:,:,3)); hold off;
    title('Processed RGB Histograms');
    legend('Red','Green','Blue');
else
    imhist(filteredImg);
    title('Processed Histogram');
end
```

🟫 Shows RGB or grayscale histograms of the processed image in the bottom-right.

📌 Add Title to the Whole Figure

```
annotation(hFig, 'textbox', [0.3 0.95 0.4 0.05], 'String', ...
    'Histogram Comparison: Original vs Processed Image', ...
    'EdgeColor', 'none', 'HorizontalAlignment', 'center', ...
    'FontSize', 12, 'FontWeight', 'bold');
```

📝 Adds a main title above all subplots.

🛑 Error Handling

```
catch e
    errordlg(['Error displaying histograms: ' e.message], 'Display Error');
end
```

🟥 Catches and shows error if anything fails.

🟩 2. ⚙️ Histogram Equalization

`pushbutton10_Callback`

```
function pushbutton10_Callback(hObject, eventdata, handles)
```

📌 Called when the "Histogram Equalization" button is pressed.

✅ Image Check

```
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first.', 'No Image Found');
    return;
end
```

🟨 Checks for an image before proceeding.

🧠 Grayscale Conversion

```
img = handles.currentImage;

if size(img, 3) == 3
    img = rgb2gray(img);
end
```

🎨 Converts to grayscale because `histeq` only works on 2D images.

⚙️ Histogram Equalization

```
equalizedImg = histeq(img);
```

📈 Stretches contrast using histogram equalization — makes dark/light areas more distinct.

📷 Display in GUI

```
axes(handles.axes2);
imshow(equalizedImg);
title('Histogram Equalization Result');
```

🖼️ Shows result in GUI (`axes2`).

🔍 Compare Histograms in New Window

```
figure;
subplot(2,1,1); imhist(img); title('Original Histogram');
subplot(2,1,2); imhist(equalizedImg); title('Equalized Histogram');
```

📊 Visual comparison between the original and equalized histograms.

💾 Save Result

```
handles.filteredImage = equalizedImg;
guidata(hObject, handles);
```

🧠 Stores the processed image.

❌ Error Handling

```
catch e
    errordlg(['Error in histogram equalization: ' e.message], 'Processing E
rror');
end
```

🟩 3. ✨ Contrast Stretching `pushbutton11_Callback`

```
function pushbutton11_Callback(hObject, eventdata, handles)
```

📌 Called when the user clicks the "Apply Contrast Stretching" button.

🧾 Validation

```
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first.', 'No Image Found');
    return;
end
```

📥 Read Min/Max from GUI Inputs

```
newMin = str2double(get(handles.edit1, 'String'));
newMax = str2double(get(handles.edit2, 'String'));

if isnan(newMin) || isnan(newMax) || newMin >= newMax
    errordlg('Please enter valid min/max values (min < max).', 'Invalid Inp
ut');
    return;
end
```

📌 Reads values from edit boxes and validates them.

🎨 Convert and Stretch Contrast

```
img = handles.currentImage;

if size(img, 3) == 3
    img = rgb2gray(img);
end

minVal = double(min(img(:)));
maxVal = double(max(img(:)));

stretchedImg = (double(img) - minVal) .* ((newMax - newMin)/(maxVal -
```

```
minVal)) + newMin;
stretchedImg = uint8(stretchedImg);
```

✔️ This maps the pixel intensity range from `[minVal, maxVal]` to `[newMin, newMax]` using a linear formula.

📊 Display

```
axes(handles.axes2);
imshow(stretchedImg);
title(['Contrast Stretching [' num2str(newMin) ' to ' num2str(newMax)
']']);
```

💾 Store and Save

```
handles.filteredImage = stretchedImg;
guidata(hObject, handles);
```

❌ Error Handling

```
catch e
    errordlg(['Error in contrast stretching: ' e.message], 'Processing Erro
r');
end
```

🟦 `edit1_Callback` — Input for Min Value

```
function edit1_Callback(hObject, eventdata, handles)
val = str2double(get(hObject, 'String'));
if isnan(val) || val < 0 || val > 255
    set(hObject, 'String', '0');
    errordlg('Please enter a value between 0 and 255', 'Invalid Input');
end
```

🔷 Validates that the user typed a valid number in `edit1` for the **minimum** value.

🟦 `edit2_Callback` — Input for Max Value

```
function edit2_Callback(hObject, eventdata, handles)
val = str2double(get(hObject, 'String'));
if isnan(val) || val < 0 || val > 255
    set(hObject, 'String', '255');
    errordlg('Please enter a value between 0 and 255', 'Invalid Input');
end
```

🔷 Validates that the user typed a valid number in `edit2` for the **maximum** value.

✅ Summary

Feature	Button	Description
Show Histogram	`pushbutton8`	Opens a side-by-side view of original and filtered image + their histograms
Histogram Equalization	`pushbutton10`	Improves image contrast by redistributing intensities
Contrast Stretching	`pushbutton11`	Stretches pixel intensities to new min/max from `edit1`, `edit2`
Min/Max Values	`edit1`, `edit2`	User-input for desired new intensity range

---

▼ 4️⃣ Noise Cancellation

🟩 `pushbutton29_Callback` – Apply Noise Cancellation Filter

```
function pushbutton29_Callback(hObject, eventdata, handles)
```

📌 Triggered when the "Apply Noise Cancellation" button is clicked.

✅ 1. Check for Image Availability

```
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first.', 'No Image Found');
    return;
end
```

✔️ Ensures that an image is loaded. If not, shows error and exits the function.

📥 2. Get Selected Filter from `popupmenu6`

```
filterList = get(handles.popupmenu6, 'String');
selectedFilter = filterList{get(handles.popupmenu6, 'Value')};
```

- Gets the list of all filter names in the dropdown.
- Reads the selected filter based on the current value (index).

🧠 3. Use the Latest Image (Filtered or Original)

```
try
    if isfield(handles, 'filteredImage') && ~isempty(handles.filteredImage)
        img = handles.filteredImage;
    else
        img = handles.currentImage;
    end

    originalImg = img; % Store a copy for histogram comparison
```

✔️ Prefers using a previously filtered image to allow chaining operations.

Stores the input as `originalImg` for later histogram display.

🌈 4. Grayscale Conversion (If RGB)

```
    if size(img, 3) == 3
        img = rgb2gray(img);
        isRGB = true;
```

```
    else
        isRGB = false;
    end
```

✔️ Many filters work best on grayscale images, so it checks if the image is RGB and converts it if needed.

Also stores a flag `isRGB` to convert back later if necessary.

🔧 5. Apply Selected Filter

```
    switch selectedFilter
        case 'Min Filter'
            filteredImg = ordfilt2(img, 1, ones(handles.kernelSize));
```

🟢 **Min Filter:** Uses `ordfilt2` to get the **minimum value** in the neighborhood.

```
case 'Max Filter'
 filteredImg = ordfilt2(img, handles.kernelSize^2, ones(handles.kernelSi
ze));
```

🟢 **Max Filter:** Gets the **maximum value** (last in sorted window).

```
        case 'Mean Filter'
            h = fspecial('average', handles.kernelSize);
            filteredImg = imfilter(img, h);
```

🟢 **Mean Filter:** Uses averaging kernel and applies it with `imfilter`.

```
        case 'Median Filter'
            filteredImg = medfilt2(img, [handles.kernelSize handles.kernelSiz
e]);
```

🟢 **Median Filter:** Removes salt-and-pepper noise.

```
matlab
Copy code
        case 'Midpoint Filter'
            minImg = ordfilt2(img, 1, ones(handles.kernelSize));
```

```
        maxImg = ordfilt2(img, handles.kernelSize^2, ones(handles.kern
elSize));
        filteredImg = (double(minImg) + double(maxImg)) / 2;
        filteredImg = uint8(filteredImg);
```

🟢 **Midpoint Filter:** Averages the min and max of the neighborhood.

🎨 6. Restore RGB Format If Needed

```
if isRGB
    filteredImg = cat(3, filteredImg, filteredImg, filteredImg);
end
```

📌 If the original image was RGB, rebuild a 3-channel image from the grayscale output.

🖼️ 7. Display Original and Filtered Image

```
axes(handles.axes1);
imshow(handles.currentImage);
title('Original Image');

axes(handles.axes2);
imshow(filteredImg);
title([selectedFilter ' Result']);
```

📷 Shows original on `axes1` and filtered image on `axes2`.

💾 8. Save the Filtered Image

```
handles.filteredImage = filteredImg;
guidata(hObject, handles);
```

📥 Stores the result in `handles` for future steps (like histogram, further filtering, etc.).

📊 9. Create Histogram Comparison Figure

```
hFig = figure('Name', 'Histogram Comparison', 'NumberTitle', 'off', 'Position', [100 100 1000 800]);
```

📌 Creates a new window titled "Histogram Comparison".

🔍 10. Display Original Image and Histogram

```
subplot(2,2,1);
if size(originalImg, 3) == 3
    imshow(originalImg);
    title('Original Color Image');

    subplot(2,2,3);
    imhist(originalImg(:,:,1)); hold on;
    imhist(originalImg(:,:,2));
    imhist(originalImg(:,:,3)); hold off;
    title('Original RGB Histograms');
    legend('Red','Green','Blue');
```

🟥 If RGB:

- Top-left: image
- Bottom-left: histogram of R, G, B channels

```
else
    imshow(originalImg);
    title('Original Grayscale Image');

    subplot(2,2,3);
    imhist(originalImg);
    title('Original Histogram');
end
```

⬜ If grayscale, just show image and single histogram.

🔬 11. Display Processed Image and Histogram

```
    subplot(2,2,2);
    imshow(filteredImg);
    title([selectedFilter ' Result']);

    subplot(2,2,4);
    if isRGB
        imhist(filteredImg(:,:,1)); hold on;
        imhist(filteredImg(:,:,2));
        imhist(filteredImg(:,:,3)); hold off;
        title('Processed RGB Histograms');
        legend('Red','Green','Blue');
```

🟦 If processed image is RGB, show each channel's histogram.

```
    else
        imhist(filteredImg);
        title('Processed Histogram');
    end
```

⬜ If grayscale, single histogram.

📝 12. Add Title to Histogram Window

```
    annotation(hFig, 'textbox', [0.3 0.95 0.4 0.05], 'String', ...
        ['Histogram Comparison: ' selectedFilter ' (Kernel: ' num2str(handle
s.kernelSize) 'x' num2str(handles.kernelSize) ')'], ...
        'EdgeColor', 'none', 'HorizontalAlignment', 'center', ...
        'FontSize', 12, 'FontWeight', 'bold');
```

📌 Adds a title at the top of the histogram window showing the filter name and kernel size.

❌ 13. Error Catching

```
catch e
    errordlg(['Error applying ' selectedFilter ': ' e.message], 'Filter Error');
end
```

🛑 Shows an error message if anything in the `try` block fails.

🟦 `popupmenu6` – Noise Filter Selection

Selection Callback

```
function popupmenu6_Callback(hObject, eventdata, handles)
```

📌 Called when the user selects a new option. Currently, it's empty, so it does nothing unless filled.

Create Function (Style Setup)

```
function popupmenu6_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontr
olBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

📌 Ensures the background is white on Windows platforms for consistent UI look.

✅ Summary Table

Component	Role
`pushbutton29`	Applies selected noise-canceling filter
`popupmenu6`	Lets the user choose filter (Min, Max, Mean, etc.)
`handles.kernelSize`	Defines size of neighborhood window
Filters Supported	Min, Max, Mean, Median, Midpoint
Histograms	Compared between original and filtered images
RGB Handling	Converts to grayscale for filtering, restores later

▼ 5️⃣ Adding Noise

🧪 Group 1: Salt & Pepper Noise

🔗 UI Controls

- Button: `pushbutton18`
- Inputs: `edit7` (PS = Salt Probability), `edit8` (PP = Pepper Probability)

🔁 Callback: `pushbutton18_Callback`

```
% --- Executes on button press in pushbutton18 (Salt & Pepper Noise).
function pushbutton18_Callback(hObject, eventdata, handles)
```

Called when Salt & Pepper button is clicked.

```
ps = str2double(get(handles.edit7, 'String'));
pp = str2double(get(handles.edit8, 'String'));
```

Reads salt (`ps`) and pepper (`pp`) probabilities from the edit boxes.

```
if isnan(ps) || isnan(pp) || ps < 0 || pp < 0 || (ps+pp) > 1
    errordlg('Please enter valid probabilities (0 <= PS,PP <= 1, PS+PP <= 1)', 'Invalid Input');
    return;
end
```

Validates input values: must be between 0 and 1 and their sum ≤ 1.

```
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first!', 'No Image');
    return;
end
```

Checks image existence.

```
if isfield(handles, 'filteredImage') && ~isempty(handles.filteredImage)
    img = handles.filteredImage;
else
    img = handles.currentImage;
end
```

Chooses the image to add noise to.

```
noisy_img = imnoise(img, 'salt & pepper', ps+pp);
```

Adds salt & pepper noise with total probability `ps + pp`.

```
axes(handles.axes1); imshow(handles.currentImage); title('Original Imag
e');
axes(handles.axes2); imshow(noisy_img); title(['Salt & Pepper Noise (PS
=' num2str(ps) ', PP=' num2str(pp) ')']);
```

Displays original and noisy images.

```
handles.filteredImage = noisy_img;
guidata(hObject, handles);
```

Stores result for next processing step.

🧾 `edit7` / `edit8` – Salt & Pepper Inputs

Both validate that values are between 0 and 1:

```
validateInput(hObject, 0, 1, false);
```

🧪 Group 2: Uniform Noise

🧷 UI Controls

- Button: `pushbutton19`
- Inputs: `edit9` (a), `edit10` (b), `edit11` (P%)

🔁 Callback: `pushbutton19_Callback`

```
a = str2double(get(handles.edit9, 'String'));
b = str2double(get(handles.edit10, 'String'));
P = str2double(get(handles.edit11, 'String'))/100;
```

Reads noise bounds `a`, `b` and percentage `P`.

```
if isnan(a) || isnan(b) || isnan(P) || b <= a || P <= 0 || P > 1
```

Validates inputs (a < b, 0 < P ≤ 1).

```
noise = a + (b-a)*rand(rows, cols, ch);
noisy_img = double(img) + P*noise;
```

Generates uniform noise and adds it scaled by `P`.

```
noisy_img = max(0, min(255, noisy_img)); noisy_img = uint8(noisy_img);
```

Clips and converts result to `uint8`.

🧾 Edits for Uniform Noise

Each has `validateInput` with correct bounds:

- `edit9`: a → [-255, 255]
- `edit10`: b → [-255, 255]
- `edit11`: P → [0–100]

🧪 Group 3: Gaussian Noise

🖇️ UI Controls

- Button: `pushbutton20`
- Inputs: `edit12` (σ), `edit13` (μ), `edit14` (P%)

🔁 Callback: `pushbutton20_Callback`

```
seg = str2double(get(handles.edit12, 'String'));
m = str2double(get(handles.edit13, 'String'));
P = str2double(get(handles.edit14, 'String'))/100;
```

Reads standard deviation (σ), mean (μ), and percentage.

```
matlab
Copycode
```

```
noise = m + seg*randn(rows, cols, ch);
noisy_img = double(img) + P*noise;
```

Adds Gaussian noise scaled by `P`.

```
matlab
Copycode
noisy_img = max(0, min(255, noisy_img)); noisy_img = uint8(noisy_img);
```

🧾 Gaussian Inputs

- `edit12`: σ → [0–100]
- `edit13`: μ → [-255, 255]
- `edit14`: P → [0–100]

🧪 Group 4: Rayleigh Noise

📎 UI Controls

- Button: `pushbutton21`
- Inputs: `edit20` (a), `edit19` (b)

🔁 Callback: `pushbutton21_Callback`

```
a = str2double(get(handles.edit20, 'String'));
b = str2double(get(handles.edit19, 'String'));
```

```
noise = a + (-b*log(1 - rand(rows, cols, ch))).^0.5;
noisy_img = double(img) + noise;
```

Generates Rayleigh noise using its formula.

🧾 Rayleigh Inputs

- `edit20`: a → [0–255]
- `edit19`: b → [0.01–255]

🧪 Group 5: Exponential Noise

🔗 UI Controls

- Button: `pushbutton22`
- Inputs: `edit17` (a), `edit18` (b)

🔁 Callback: `pushbutton22_Callback`

```
a = str2double(get(handles.edit17, 'String'));
b = str2double(get(handles.edit18, 'String'));
```

```
noise = (-1/a)*log(1 - rand(rows, cols, ch));
noisy_img = double(img) + b*noise;
```

Applies exponential noise.

🧾 Exponential Inputs

- `edit17`: a → [0.01–100]
- `edit18`: b → [0–255]

🧪 Group 6: Gamma Noise

🔗 UI Controls

- Button: `pushbutton24`
- Inputs: `edit21` (a), `edit22` (b)

🔁 Callback: `pushbutton24_Callback`

```
a = str2double(get(handles.edit21, 'String'));
b = str2double(get(handles.edit22, 'String'));
```

```
if exist('gamrnd', 'file')
    noise = gamrnd(a, b, rows, cols, ch);
else
```

```
    noise = zeros(rows, cols, ch);
    for k = 1:a
        noise = noise - b*log(1 - rand(rows, cols, ch));
    end
end
```

Generates gamma noise with fallback if `gamrnd()` doesn't exist.

🧾 Gamma Inputs

- `edit21`: a → [0–100]
- `edit22`: b → [0–255]

🧾 Summary Table

Noise Type	Button	Inputs	Core Formula / Method
Salt & Pepper	pushbutton18	edit7, edit8	`imnoise(..., 'salt & pepper', ps + pp)`
Uniform	pushbutton19	edit9–edit11	`a + (b-a)*rand`
Gaussian	pushbutton20	edit12–edit14	`μ + σ*randn`
Rayleigh	pushbutton21	edit20, edit19	`a + sqrt(-b*log(1 - rand))`
Exponential	pushbutton22	edit17, edit18	`-1/a * log(1 - rand)`
Gamma	pushbutton24	edit21, edit22	`gamrnd(a, b)` or fallback loop using `log(1-u)`

▼ 6️⃣ Point Detection & Sharpening

🟨 `pushbutton28_Callback` — Apply Point Operation

```
% --- Executes on button press in pushbutton28 (Apply Point Operatio
n)
function pushbutton28_Callback(hObject, eventdata, handles)
```

📌 This function runs when the user clicks the "Apply" button to perform either Point Detection or Point Sharpening.

🧪 1. Check if Image is Loaded

```
try
    if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
        errordlg('Please load an image first!', 'No Image');
        return;
    end
```

✔️ If no image is loaded into the GUI, show an error and cancel processing.

📥 2. Get Selected Operation from Dropdown

```
    operations = get(handles.popupmenu7, 'String');
    selectedOp = operations{get(handles.popupmenu7, 'Value')};
```

- Retrieves the list of options from `popupmenu7`
- Gets the selected string (e.g., `'Point Detection'` or `'Point Sharpening'`)

🎨 3. Convert to Grayscale (if needed)

```
        if size(handles.currentImage, 3) == 3
        img = rgb2gray(handles.currentImage);
    else
        img = handles.currentImage;
    end
```

📌 Ensures the image is single-channel (grayscale) for filtering operations.

🧱 4. Apply Selected Operation

```
    switch selectedOp
```

Starts a switch-case block based on the chosen point operation.

🟡 Case 1: Point Detection

```
    case 'Point Detection'
        h = [1 1 1; 1 -8 1; 1 1 1]; % Laplacian kernel
```

This is a **Laplacian kernel** used for edge/point detection. It enhances regions with rapid intensity changes.

```
        filtered = imfilter(double(img), h, 'same');
```

Applies the kernel filter to the image using `imfilter`. Converts image to `double` for accurate computation.

```
        result = abs(filtered); % Get absolute values
```

✔️ Removes negative values from the result (Laplacian can produce both positive and negative edges).

```
        result = uint8(255 * mat2gray(result)); % Normalize to 0-255
```

- `mat2gray` scales values between 0–1
- Then multiplied by 255 and converted to `uint8`

🟠 Case 2: Point Sharpening

```
    case 'Point Sharpening'
        blurred = imgaussfilt(img, 2);
```

🟧 Uses **Gaussian blur** to smooth the image.

```
        mask = img - blurred;
```

Subtracts the blurred image from the original to isolate **edges and details** (this is the "mask").

```
        result = img + mask; % Add the mask back to original
```

Adds the mask back to sharpen the image = Unsharp Masking.

```
result = uint8(min(max(result, 0), 255)); % Clamp to 0-255
```

Clips any overflows and converts to `uint8`.

📷 5. Display Original and Result

```
axes(handles.axes1);
imshow(handles.currentImage);
title('Original Image');
```

🖼️ Shows the original image in `axes1`.

```
axes(handles.axes2);
imshow(result);
title(selectedOp);
```

🖼️ Shows the result of point detection or sharpening in `axes2`, with title.

💾 6. Save Result for Next Steps

```
handles.filteredImage = result;
guidata(hObject, handles);
```

🧠 Stores the processed image for further processing (e.g., histogram, noise canceling).

🛑 7. Error Handling

```
catch ME
    errordlg(['Error in ' selectedOp ': ' ME.message], 'Processing Error');
end
```

🟥 If something goes wrong, display a detailed error dialog.

🟦 `popupmenu7_Callback` and `CreateFcn`

This function runs when the user changes the dropdown. It's not used actively here (the `pushbutton28` handles the selection).

```
function popupmenu7_CreateFcn(hObject, eventdata, handles)
    if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicon
trolBackgroundColor'))
        set(hObject,'BackgroundColor','white');
    end
```

📌 Ensures the dropdown background color looks correct on Windows systems.

📌 Summary Table

Feature	UI Element	Operation
Point Detection	`pushbutton28` + `'Point Detection'` in `popupmenu7`	Uses Laplacian kernel `[-8 center]` to highlight point features
Point Sharpening	`pushbutton28` + `'Point Sharpening'` in `popupmenu7`	Applies Unsharp Masking (original + details from Gaussian difference)

▼ 7️⃣ Line Detection & Sharpening

✅ Functional Buttons and UI Elements

Feature	Function	Related UI
General Apply	`pushbutton27`	`popupmenu8`
Horizontal Line Detection	`pushbutton31`	
Vertical Line Detection	`pushbutton32`	
Diagonal Left Line Detection	`pushbutton33`	
Diagonal Right Line Detection	`pushbutton34`	

🟦 `pushbutton27_Callback` — Main Line Detection & Sharpening

```
% --- Executes on button press in pushbutton27.
```

```
function pushbutton27_Callback(hObject, eventdata, handles)
```

📌 Triggered when user clicks the "Apply" button for line operations.

✅ 1. Image Existence Check

```
try
    if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
        errordlg('Please load an image first!', 'No Image');
        return;
    end
```

Checks if any image has been loaded. If not, an error is shown and function exits.

🧠 2. Get User Selection from Dropdown

```
    operations = get(handles.popupmenu8, 'String');
    selectedOp = operations{get(handles.popupmenu8, 'Value')};
```

- Retrieves list of options in `popupmenu8`.
- Gets the selected operation name.

🎨 3. Convert to Grayscale (if necessary)

```
    if size(handles.currentImage, 3) == 3
        img = rgb2gray(handles.currentImage);
    else
        img = handles.currentImage;
    end
```

✔️ Ensures the image is grayscale (most edge and line detectors operate on 2D images).

🔧 4. Apply Operation Based on Selection

```
switch selectedOp
```

🟡 Case: Line Detection (Sobel)

```
    case 'Line Detection (Sobel)'
        result = edge(img, 'sobel', 'horizontal');
```

Uses MATLAB's `edge` function with the Sobel operator in the horizontal direction.

🟡 Case: Line Detection (Roberts)

```
    case 'Line Detection (Roberts)'
        result = edge(img, 'roberts', 'horizontal');
```

Applies the Roberts cross operator for line edge detection, defaulted to horizontal.

🟠 Case: Line Sharpening

```
    case 'Line Sharpening'
        blurred = imgaussfilt(img, 1);
        mask = img - blurred;
        result = img + 1.5*mask;
        result = uint8(min(max(result, 0), 255));
```

- Applies **unsharp masking**:
  - Blurs the image.
  - Subtracts blur from original to form a detail-enhancing mask.
  - Adds the mask with a multiplier (1.5x) to increase sharpness.
  - Ensures pixel values are clamped between 0 and 255.

📷 5. Display Original and Processed Images

```
    axes(handles.axes1);
    imshow(handles.currentImage);
    title('Original Image');

    axes(handles.axes2);
    imshow(result);
    title([selectedOp ' Result']);
```

🖼️ Shows original and processed results in separate axes.

💾 6. Save the Result for Further Use

```
    handles.filteredImage = result;
    guidata(hObject, handles);
```

❌ 7. Error Catching

```
catch ME
    errordlg(['Error in ' selectedOp ': ' ME.message], 'Processing Error');
end
```

Catches and shows an error dialog if something fails in the try block.

🟩 `popupmenu8_Callback` — Dropdown Selection Handling

```
% --- Executes on selection change in popupmenu8 line detect and sha
rpening
function popupmenu8_Callback(hObject, eventdata, handles)
    set([handles.pushbutton31, handles.pushbutton32, handles.pushbutto
n33, handles.pushbutton34], 'Visible', 'on');
    guidata(hObject, handles);
```

📌 When user selects a different operation, this ensures the four **direction buttons** remain visible for optional directional detection.

```
function popupmenu8_CreateFcn(hObject, eventdata, handles)
    if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicon
trolBackgroundColor'))
        set(hObject,'BackgroundColor','white');
    end
```

Ensures proper UI background appearance for Windows users.

🧭 Directional Line Detection

Each button (`pushbutton31–34`) uses a common helper function called `applyLineOperation`, which takes a direction code.

🔘 `pushbutton31` → Horizontal (H)

```
function pushbutton31_Callback(hObject, eventdata, handles)
    applyLineOperation(handles, 'H');
```

🔘 `pushbutton32` → Vertical (V)

```
function pushbutton32_Callback(hObject, eventdata, handles)
    applyLineOperation(handles, 'V');
```

🔘 `pushbutton33` → Diagonal Left (DL)

```
function pushbutton33_Callback(hObject, eventdata, handles)
    applyLineOperation(handles, 'DL');
```

🔘 `pushbutton34` → Diagonal Right (DR)

```
function pushbutton34_Callback(hObject, eventdata, handles)
    applyLineOperation(handles, 'DR');
```

🧠 `applyLineOperation(handles, direction)`

```
matlab
Copy code
function applyLineOperation(handles, direction)
```

📌 This is a helper function that performs direction-specific line detection or sharpening.

🧩 Main Steps

1. **Image Check & Preparation**

```
matlab
Copy code
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg(...); return;
end
```

✅ Validates the image exists.

```
matlab
Copy code
operations = get(handles.popupmenu8, 'String');
selectedOp = operations{get(handles.popupmenu8, 'Value')};
img = rgb2gray_if_needed(handles.currentImage);  % Ensures grays
cale
img = double(img);
```

2. **Select Kernel Based on Operation & Direction**

```
matlab
Copy code
switch selectedOp
    case 'Line Detection (Sobel)'
        % Sobel directional kernels
        switch direction
```

```
            case 'H':  [-1 -2 -1; 0 0 0; 1 2 1]
            case 'V':  [-1 0 1; -2 0 2; -1 0 1]
            case 'DL': [0 1 2; -1 0 1; -2 -1 0]
            case 'DR': [-2 -1 0; -1 0 1; 0 1 2]
        end

    case 'Line Detection (Roberts)'
        % Roberts 2×2 directional kernels
        switch direction
            case 'H':  [1 0; 0 -1]
            case 'V':  [0 1; -1 0]
            case 'DL': [0 -1; 1 0]
            case 'DR': [-1 0; 0 1]
        end

    case 'Line Sharpening'
        % Directional sharpening
        switch direction
            case 'H':  [0 -1 0; 0 3 0; 0 -1 0]
            case 'V':  [0 0 0; -1 3 -1; 0 0 0]
            case 'DL': [-1 0 0; 0 3 0; 0 0 -1]
            case 'DR': [0 0 -1; 0 3 0; -1 0 0]
        end
end
result = imfilter(img, kernel);
```

3. **Post-Processing & Display**

```
matlab
Copy code
result = im2uint8(mat2gray(result));  % Normalize and convert
imshow result in axes2 with title
save to handles.filteredImage
```

4. **Error Handling**

```
matlab
Copy code
catch ME
    errordlg(['Error in ' selectedOp ': ' ME.message], 'Processing Erro
r');
end
```

✅ Summary Table

Operation	Direction	Kernel Type	Purpose
Sobel	H/V/DL/DR	3×3	Edge/line detection
Roberts	H/V/DL/DR	2×2	Simple diagonal edge detection
Sharpening	H/V/DL/DR	3×3	Directional detail enhancement

✅ Summary Table

Feature	Trigger	Core Technique	Description
Line Detection (Sobel)	`pushbutton27` + dropdown	`edge(img, 'sobel')`	Detects edges in horizontal direction
Line Detection (Roberts)	`pushbutton27` + dropdown	`edge(img, 'roberts')`	Uses Roberts operator for fine edges
Line Sharpening	`pushbutton27` + dropdown	Unsharp Masking (1.5×)	Enhances edges with boosted difference mask
Directional Detection (H/V/DL/DR)	`pushbutton31–34`	Custom kernel + `applyLineOperation()`	Detects lines in specific directions

▼ 8️⃣ Transformation

📌 Functional Buttons and Inputs:

Transformation	Button	Input(s)
Negative	pushbutton37	—
Log Transform	pushbutton40	—
Gamma	pushbutton41	edit23

🟨 Group 1: Negative Transformation

🔘 Button: pushbutton37

```
matlab
Copy code
function pushbutton37_Callback(hObject, eventdata, handles)
```

Triggered when the **Negative Transform** button is pressed.

✅ Check Image Exists

```
matlab
Copy code
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first!', 'No Image Loaded');
    return;
end
```

📷 Select Image to Use

```
matlab
Copy code
if isfield(handles, 'filteredImage') && ~isempty(handles.filteredImage)
    img = handles.filteredImage;
else
    img = handles.currentImage;
```

```
end
```

✔️ Uses filtered image if available; otherwise uses the original.

🔄 Apply Negative Transformation

```
matlab
Copy code
negative = 255 - img;
```

Inverts pixel values to create a **photographic negative**.

🖼️ Display

```
matlab
Copy code
axes(handles.axes1); imshow(handles.currentImage); title('Original Imag
e');
axes(handles.axes2); imshow(negative); title('Negative Transform');
```

💾 Store Result

```
matlab
Copy code
handles.filteredImage = negative;
guidata(hObject, handles);
```

❌ Error Handling

```
matlab
Copy code
catch ME
```

```
    errordlg(['Error in negative transformation: ' ME.message], 'Processin
g Error');
end
```

🟦 Group 2: Logarithmic Transformation

🔘 Button: `pushbutton40`

```
matlab
Copy code
function pushbutton40_Callback(hObject, eventdata, handles)
```

Triggered when **Log Transform** is clicked.

✅ Image Check and Selection

Same logic as above:

```
matlab
Copy code
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg(...); return;
end

img = (filteredImage if exists) or currentImage;
```

🔄 Normalize and Apply Log Transform

```
matlab
Copy code
img = im2double(img);  % Convert to range [0,1]
c = 1 / log(1 + max(img(:)));
```

```
logImg = c * log(1 + img);
```

- `log(1 + img)` enhances darker regions.
- `c` scales the image to stay in [0, 1].

📥 Convert to uint8 and Display

```
matlab
Copy code
transformed = im2uint8(logImg);
axes(handles.axes2); imshow(transformed); title('Log Transform');
```

💾 Store Result

```
matlab
Copy code
handles.filteredImage = transformed;
guidata(hObject, handles);
```

❌ Error Handling

```
matlab
Copy code
catch ME
    errordlg(['Error in log transformation: ' ME.message], 'Processing Err
or');
end
```

🟪 Group 3: Gamma Correction

🔘 Button: `pushbutton41`

🔲 Input Box: `edit23`

```
matlab
Copy code
function pushbutton41_Callback(hObject, eventdata, handles)
```

✅ Image Check

```
matlab
Copy code
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg(...); return;
end
```

📥 Read and Validate Gamma Value

```
matlab
Copy code
gammaStr = get(handles.edit23, 'String');
gamma = str2double(gammaStr);

if isnan(gamma) || gamma <= 0
    errordlg(...); set(handles.edit23, 'String', '1.0'); return;
end
```

🔄 Apply Gamma Transformation

```
matlab
Copy code
img = im2double(handles.currentImage);
gammaImg = img.^gamma;
```

```
filtered = im2uint8(gammaImg);
```

✔️ Applies the power-law transformation to enhance either bright or dark areas.

🖼️ Display and Store

```
matlab
Copy code
axes(handles.axes1); imshow(handles.currentImage); title(...);
axes(handles.axes2); imshow(filtered); title(...);
handles.filteredImage = filtered;
guidata(hObject, handles);
```

❌ Error Handling

```
matlab
Copy code
catch ME
    errordlg(['Error in gamma transformation: ' ME.message], 'Processing
Error');
end
```

✏️ `edit23_Callback` — Live Gamma Preview

```
matlab
Copy code
function edit23_Callback(hObject, eventdata, handles)
```

📥 Get and Validate Input

```
matlab
Copy code
gammaStr = get(hObject, 'String');
gamma = str2double(gammaStr);

if isnan(gamma) || gamma <= 0
    errordlg(...); set(hObject, 'String', '1.0'); return;
end
```

🔍 Preview on Small Image

```
matlab
Copy code
if isfield(handles, 'currentImage') && ~isempty(handles.currentImage)
    smallImg = imresize(handles.currentImage, 0.2);
    previewImg = im2uint8(im2double(smallImg).^gamma);
    axes(handles.axes2); imshow(previewImg); title(...);
end
```

🔧 `edit23_CreateFcn`

```
matlab
Copy code
function edit23_CreateFcn(hObject, eventdata, handles)
    set(hObject,'BackgroundColor','white');
    set(hObject, 'String', '1.0'); % Default gamma
end
```

✅ Summary Table

Transformation	Function	Description
Negative	`pushbutton37`	Inverts pixel values: `255 - img`

Log Transform	`pushbutton40`	Enhances dark pixels: `c * log(1 + img)`
Gamma	`pushbutton41` + `edit23`	Nonlinear power correction: `img ^ γ`

▼ 9️⃣ Frequency Domain Filter

🟩 GUI Elements

- **popupmenu10**: Dropdown menu for selecting the filter type:
  - 'Ideal Low Pass'
  - 'Ideal High Pass'
  - 'Butterworth Low Pass'
  - 'Butterworth High Pass'
  - 'Gaussian Low Pass'
  - 'Gaussian High Pass'
- **edit26**: User input for cutoff frequency `D0`.
- **edit27**: User input for Butterworth filter order `n`.
- **pushbutton50**: Executes the selected filter on the image.

🟦 `pushbutton50_Callback` : Apply Frequency Filter

```
matlab
Copy code
function pushbutton50_Callback(hObject, eventdata, handles)
```

Triggered when the user clicks the "Apply Frequency Filter" button.

```
matlab
Copy code
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first!', 'No Image');
    return;
end
```

Checks if an image is loaded.

```
matlab
Copy code
D0 = str2double(get(handles.edit26, 'String'));
n = str2double(get(handles.edit27, 'String'));
filterTypes = get(handles.popupmenu10, 'String');
filterType = get(handles.popupmenu10, 'Value');
selectedFilterName = filterTypes{filterType};
```

Retrieves the cutoff frequency `D0`, filter order `n`, and selected filter type from the GUI.

```
matlab
Copy code
if isnan(D0) || D0 <= 0
    errordlg('Please enter a valid positive D0 value!', 'Invalid Parameter');
    return;
end

if (filterType == 3 || filterType == 4) && (isnan(n) || n <= 0)
    errordlg('Please enter a valid positive Butterworth order!', 'Invalid Par
ameter');
    return;
end
```

Validates the inputs: `D0` must be positive. `n` is required only for Butterworth filters.

```
matlab
Copy code
if size(handles.currentImage, 3) == 3
    img = rgb2gray(handles.currentImage);
else
    img = handles.currentImage;
```

```
end
```

Converts image to grayscale for frequency filtering.

```
matlab
Copy code
switch filterType
    case 1
        filtered = idealFilter(img, D0, 'low');
    case 2
        filtered = idealFilter(img, D0, 'high');
    case 3
        filtered = butterworthFilter(img, D0, n, 'low');
    case 4
        filtered = butterworthFilter(img, D0, n, 'high');
    case 5
        filtered = gaussianFilter(img, D0, 'low');
    case 6
        filtered = gaussianFilter(img, D0, 'high');
end
```

Applies the appropriate filter by calling a helper function.

```
matlab
Copy code
axes(handles.axes1); imshow(handles.currentImage); title('Original Imag
e');
axes(handles.axes2); imshow(filtered); title([selectedFilterName ' Filte
r']);
handles.filteredImage = filtered;
guidata(hObject, handles);
```

Displays the original and filtered images and updates `filteredImage`.

🟦 Helper Functions

📌 idealFilter

```
matlab
Copy code
function output = idealFilter(img, D0, type)
```

Creates a binary mask: pass if distance from center ≤ `D0`.

```
matlab
Copy code
H = double(D <= D0);  % low-pass
H = double(D > D0);   % high-pass
```

📌 butterworthFilter

```
matlab
Copy code
function output = butterworthFilter(img, D0, n, type)
```

Smooth transition based on cutoff `D0` and order `n`:

```
matlab
Copy code
H = 1 ./ (1 + (D./D0).^(2*n));  % low-pass
H = 1 ./ (1 + (D0./D).^(2*n));  % high-pass
```

📌 gaussianFilter

```
matlab
Copy code
function output = gaussianFilter(img, D0, type)
```

Smoothest transition:

```
matlab
Copy code
H = exp(-(D.^2)./(2*D0^2));       % low-pass
H = 1 - exp(-(D.^2)./(2*D0^2));    % high-pass
```

📌 `applyFrequencyFilter`

```
matlab
Copy code
function output = applyFrequencyFilter(img, H)
```

Core operation shared by all filters:

- Converts image to frequency domain via `fft2`
- Applies filter mask `H`
- Returns inverse FFT and normalizes to 0–1

```
matlab
Copy code
F = fftshift(fft2(img));
G = H .* F;
output = real(ifft2(ifftshift(G)));
output = mat2gray(output);
```

✅ Summary

Filter Type	Uses `n`?	Function Called
Ideal Low/High Pass	❌	`idealFilter`
Butterworth Low/High	✅	`butterworthFilter`
Gaussian Low/High	❌	`gaussianFilter`

▼ 🔟 Fourier Transformation

🔵 `pushbutton45_Callback` – Fourier Transform

```
matlab
Copy code
function pushbutton45_Callback(hObject, eventdata, handles)
```

Triggered when the "Fourier Transform" button is clicked.

```
matlab
Copy code
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first!', 'No Image');
    return;
end
```

✅ Checks if an image is loaded. If not, shows an error and exits.

```
matlab
Copy code
if size(handles.currentImage, 3) == 3
    img = rgb2gray(handles.currentImage);
else
    img = handles.currentImage;
end
```

📷 Converts the image to grayscale if it's in color (3-channel RGB).

```
matlab
Copy code
f_img = fft2(double(img));          % Compute 2D Fourier Transform
fshift = fftshift(f_img);          % Shift zero-frequency component to cent
er
```

```
magnitude_spectrum = mat2gray(log(1 + abs(fshift))); % Log scale for vi
sibility
```

📈 Applies the **Fast Fourier Transform**, centers it, and prepares the magnitude spectrum for display using log scaling.

```
matlab
Copy code
handles.fourierData = fshift; % Store for inverse use
```

💾 Saves the frequency domain data for future use.

```
matlab
Copy code
axes(handles.axes2);
imshow(magnitude_spectrum, []);
colormap(handles.axes2, gray);
colorbar(handles.axes2);
title('Fourier Transform (Magnitude Spectrum)');
```

🎨 Displays the frequency spectrum in `axes2`.

```
matlab
Copy code
guidata(hObject, handles);
```

🔁 Updates the GUI state.

🔴 `pushbutton46_Callback` – Inverse Fourier Transform

```
matlab
Copy code
```

```
function pushbutton46_Callback(hObject, eventdata, handles)
```

Called when "Inverse Fourier Transform" is clicked.

```
matlab
Copy code
if ~isfield(handles, 'fourierData') || isempty(handles.fourierData)
    errordlg('Please perform Fourier Transform first!', 'No Fourier Data');
    return;
end
```

⚠️ Verifies that Fourier data exists before proceeding.

```
matlab
Copy code
f_ishift = ifftshift(handles.fourierData);  % Unshift the frequency image
img_back = ifft2(f_ishift);                 % Apply inverse FFT
```

🔁 Reverses the frequency shift and performs the **inverse FFT** to reconstruct the image.

```
matlab
Copy code
axes(handles.axes2);
imshow(img_back, []);
title('Inverse Fourier Transform (Reconstructed Image)');
colorbar(handles.axes2, 'off');
```

🖼️ Displays the reconstructed image from frequency domain.

```
matlab
Copy code
handles.reconstructedImage = img_back;
```

```
guidata(hObject, handles);
```

💾 Stores and updates the GUI state.

📌 Summary

Button	Function Name	Description
pushbutton45	`pushbutton45_Callback`	Computes and shows the Fourier magnitude spectrum
pushbutton46	`pushbutton46_Callback`	Reconstructs the image using inverse Fourier Transform

---

▼ 1️⃣1️⃣ Brightness

🔆 Brightness Adjustment Functionalities

📌 Components:

- **Addition** → `pushbutton52`
- **Multiplication** → `pushbutton53`
- **Subtraction** → `pushbutton54`
- **Division** → `pushbutton55`
- **Brightness Value Input** → `edit28`

🔷 `edit28_Callback` & `edit28_CreateFcn`

```
matlab
Copy code
function edit28_Callback(hObject, eventdata, handles)
value = str2double(get(hObject, 'String'));
if isnan(value)
    errordlg('Please enter a valid number!', 'Invalid Input');
    set(hObject, 'String', '10');
end
handles.brightnessValue = value;
```

```
guidata(hObject, handles);
```

- Retrieves and validates the numeric input.
- Stores it in `handles.brightnessValue`.

```
matlab
Copy code
function edit28_CreateFcn(hObject, eventdata, handles)
set(hObject, 'BackgroundColor', 'white');
set(hObject, 'String', '10');
```

- Initializes the brightness edit box with default value 10.

🔷 `pushbutton52_Callback` – Brightness Addition

```
matlab
Copy code
function pushbutton52_Callback(hObject, eventdata, handles)
```

- Adds the value from `edit28` to each pixel.
- Converts to double to avoid overflow during calculation.
- Uses `min(..., 255)` to clamp values.

🔷 `pushbutton53_Callback` – Brightness Multiplication

```
matlab
Copy code
function pushbutton53_Callback(hObject, eventdata, handles)
```

- Multiplies each pixel by the value from `edit28`.
- Input must be positive.

- Clamps the result to 255 to avoid overflow.

🔷 `pushbutton54_Callback` – Brightness Subtraction

```
matlab
Copy code
function pushbutton54_Callback(hObject, eventdata, handles)
```

- Subtracts the value from each pixel.
- Uses `max(..., 0)` to avoid underflow.

🔷 `pushbutton55_Callback` – Brightness Division

```
matlab
Copy code
function pushbutton55_Callback(hObject, eventdata, handles)
```

- Divides each pixel by the value from `edit28`.
- Input must be positive and non-zero.
- Uses `max(..., 0)` to clamp any potential negatives due to rounding.

🧠 Shared Logic Summary

In all four operations:

- If a filtered image exists (`handles.filteredImage`), it is used instead of the original.
- Results are displayed in `axes2`.
- The original is shown in `axes1`.
- Output is saved back to `handles.filteredImage`.

▼ 1️⃣2️⃣ Conversion

🔲 Binary Conversion

- **Button:** `pushbutton56`
- **Input Field (Threshold):** `edit29`

🔷 Code Explanation

```
matlab
Copy code
function pushbutton56_Callback(hObject, eventdata, handles)
```

Triggered when the user clicks the **Binary Conversion** button.

```
matlab
Copy code
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first!', 'No Image');
    return;
end
```

Validates that an image is loaded.

```
matlab
Copy code
threshold = str2double(get(handles.edit29, 'String'));
if isnan(threshold) || threshold < 0 || threshold > 255
    errordlg('Threshold must be between 0-255!', 'Invalid Threshold');
    return;
end
```

Reads and validates the threshold input from `edit29`.

```
matlab
Copy code
if size(handles.currentImage, 3) == 3
    grayImg = rgb2gray(handles.currentImage);
else
```

```
    grayImg = handles.currentImage;
end
```

Converts the image to grayscale if it's RGB.

```
matlab
Copy code
handles.processedImage = imbinarize(grayImg, threshold/255);
```

Applies **binary thresholding** (0 to 1 scale for `imbinarize`).

```
matlab
Copy code
axes(handles.axes2);
imshow(handles.processedImage);
title(['Binary Image (Threshold: ' num2str(threshold) ')']);
```

Displays the binary image in `axes2`.

```
matlab
Copy code
guidata(hObject, handles);
```

Saves changes to the GUI state.

🟫 Grayscale Conversion

- **Button:** `pushbutton58`
- **Input Field (Method):** `edit30`

🔷 Code Explanation

```
matlab
Copy code
```

```
function pushbutton58_Callback(hObject, eventdata, handles)
```

Triggered on pressing the **Grayscale Conversion** button.

```
matlab
Copy code
if ~isfield(handles, 'currentImage') || isempty(handles.currentImage)
    errordlg('Please load an image first!', 'No Image');
    return;
end
```

Checks if an image is loaded.

```
matlab
Copy code
method = round(str2double(get(handles.edit30, 'String')));
if isnan(method) || method < 1 || method > 5
    errordlg('Method must be 1-5!', 'Invalid Method');
    return;
end
```

Reads and validates the grayscale method from `edit30`.

```
matlab
Copy code
if size(handles.currentImage, 3) == 3
    switch method
        case 1 % Average
            grayImg = mean(handles.currentImage, 3);
        case 2 % Weighted (Luminance)
            weights = [0.2989, 0.5870, 0.1140];
            grayImg = handles.currentImage(:,:,1)*weights(1) + ...
                      handles.currentImage(:,:,2)*weights(2) + ...
                      handles.currentImage(:,:,3)*weights(3);
        case 3 % Red channel
```

```
            grayImg = handles.currentImage(:,:,1);
        case 4 % Green channel
            grayImg = handles.currentImage(:,:,2);
        case 5 % Blue channel
            grayImg = handles.currentImage(:,:,3);
    end
    handles.processedImage = uint8(grayImg);
else
    handles.processedImage = handles.currentImage;
end
```

Applies the selected grayscale method:

- **1**: Average all RGB channels.
- **2**: Use standard luminance weights.
- **3-5**: Use individual color channels.

```
matlab
Copy code
axes(handles.axes2);
imshow(handles.processedImage);
methodNames = {'Average', 'Luminance', 'Red Channel', 'Green Channe
l', 'Blue Channel'};
title(['Grayscale (' methodNames{method} ')']);
```

Displays the result with a dynamic title.

```
matlab
Copy code
guidata(hObject, handles);
```

Updates the GUI state.

🧩 Input Field Validators

`edit29` (Binary Threshold)

```
matlab
Copy code
function edit29_Callback(hObject, eventdata, handles)
val = str2double(get(hObject, 'String'));
if isnan(val) || val < 0 || val > 255
    errordlg('Threshold must be 0-255!', 'Invalid Input');
    set(hObject, 'String', '127');
end
```

`edit30` (Grayscale Method)

```
matlab
Copy code
function edit30_Callback(hObject, eventdata, handles)
val = round(str2double(get(hObject, 'String')));
if isnan(val) || val < 1 || val > 5
    errordlg('Method must be 1-5!', 'Invalid Input');
    set(hObject, 'String', '1');
end
```

▼ 1️⃣3️⃣ Browse & Save image

🔷 1. Browse Image — `pushbutton42`

```
matlab
Copy code
function pushbutton42_Callback(hObject, eventdata, handles)
    [filename, pathname] = uigetfile({'*.jpg;*.png;*.bmp;*.tif;*.tiff', 'Image F
iles'}, 'Select an Image');
    if isequal(filename, 0)
        return; % User cancelled
    end

    try
```

```
        img = imread(fullfile(pathname, filename));     % Read image
        handles.currentImage = img;                    % Store it in handles
        axes(handles.axes1); imshow(img);              % Show on axes1
        title('Original Image');
        guidata(hObject, handles);                     % Save handles changes
    catch e
        errordlg(['Error loading image: ' e.message], 'Image Load Error');
    end
end
```

📌 Explanation:

- Opens a file dialog for image selection.
- Reads the image and stores it as `handles.currentImage`.
- Displays the image in `axes1` with the title "Original Image".
- `guidata` is called to store the updated handles.
- If there's an error (e.g., unreadable file), it shows an error dialog.

🔷 2. Save Image — `pushbutton43`

```
matlab
Copy code
function pushbutton43_Callback(hObject, eventdata, handles)
    if ~isfield(handles, 'filteredImage') || isempty(handles.filteredImage)
        errordlg('No filtered image available to save. Please apply a filter fir
st.', 'Nothing to Save');
        return;
    end

    filteredImg = handles.filteredImage;
    defaultName = ['filtered_' datestr(now, 'yyyymmdd_HHMMSS') '.pn
g'];

    [filename, pathname] = uiputfile(...
        {'*.png','PNG files (*.png)'; ...
         '*.jpg','JPEG files (*.jpg)'; ...
```

```
        '*.tif','TIFF files (*.tif)'; ...
        '*.*','All Files (*.*)'}, ...
        'Save Filtered Image As', ...
        defaultName);

    if isequal(filename, 0) || isequal(pathname, 0)
        return; % User cancelled
    end

    try
        [~, ~, ext] = fileparts(filename);
        switch lower(ext)
            case '.png'
                imwrite(filteredImg, fullfile(pathname, filename), 'png');
            case '.jpg'
                imwrite(filteredImg, fullfile(pathname, filename), 'jpg', 'Quality',
90);
            case '.tif'
                imwrite(filteredImg, fullfile(pathname, filename), 'tif');
            otherwise
                imwrite(filteredImg, fullfile(pathname, [filename '.png']), 'pn
g'); % default to PNG
        end
        msgbox(['Image successfully saved to: ' fullfile(pathname, filenam
e)], 'Save Complete');
    catch e
        errordlg(['Error saving image: ' e.message], 'Save Error');
    end
end
```

📌 Explanation:

- Checks if a filtered image exists to save.
- Creates a default filename with timestamp.
- Opens a save file dialog (`uiputfile`) with image format options.
- Uses `imwrite()` to save the image in the selected format.

- If extension is missing or unknown, defaults to PNG.
- Shows confirmation or error dialogs.

🔷 3. Edit Field for Filename (optional) — `edit24`

```
matlab
Copy code
function edit24_Callback(hObject, eventdata, handles)
% Called when user types in edit24 — Not actively used in save logic
end

function edit24_CreateFcn(hObject, eventdata, handles)
    if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicon
trolBackgroundColor'))
        set(hObject,'BackgroundColor','white');
    end
end
```

📌 Explanation:

- `edit24` appears to be unused in the saving logic.
- Standard setup to make sure it has a white background on Windows.
- Might be used for manual filename input in future versions.