INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

DIGITAL SIGNAL AND IMAGE PROCESSING LABORATORY								
VI Semester: ECE								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
AECD45	**Core**	**L**	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
		0	0	4	2	30	70	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisite: Signals and Systems								

I. COURSE OVERVIEW:

This laboratory course is concerned with the implementation of digital signal processing algorithms using different computational platforms such as MATLAB and Python tools that give core knowledge to develop the realtime applications in signal and image processing. It focuses on the convolution, discrete Fourier trans-form, fast Fourier transform algorithms, digital filter design and multi rate signal processing. Digital signal and image processing applications are used in speech processing, image processing, audio and video data compression, and communication systems.

II.COURSE OBJECTIVES:

The students will try to learn:

I. To familiarize students with Python programming for signal and image processing applications.
II. To understand the implementation of fundamental digital signal processing techniques, such as filtering, transformations, and spectral analysis.
III. To introduce concepts of image processing including enhancement, segmentation, and feature extraction.
IV. To provide hands-on experience in designing and analyzing algorithms for real-time signal and image processing tasks.
V. To enable students to apply theoretical concepts of digital signal and image processing to practical problems.

III. COURSE OUTCOMES:

At the end of the course students will be able to:

CO1 Analyze signals in the time and frequency domains using techniques like FFT, DFT, and filtering
CO2 Develop Python-based projects that integrate signal and image processing techniques to solve engineering problems.
CO3 Implement basic operations on signals and images using Python libraries such as NumPy, SciPy, OpenCV, and Matplotlib.
CO4 Apply spatial and frequency domain techniques for image enhancement and restoration
CO5 Design and implement image segmentation techniques using edge detection, thresholding, and clustering
CO6 Evaluate and optimize algorithms for real-time signal and image processing applications.

IV. COURSE CONTENT:

EXERCISES FOR DIGITAL SIGNAL AND IMAGE PROCESSING LABORATORY

1. Exercises on Discrete-Time Systems

To be proficient in MATLAB programming, students be able to compute:

1. Linear Convolution
2. Circular Convolution
3. Cyclic Convolution
4. Overlap-Add Method to compute linear convolution
5. Overlap-Save Method to compute linear convolution

2. Exercises on FIR and IIR Filter Design

To be proficient in MATLAB programming, students need to design and implement FIR and IIR Filters.

1. FIR filter design (LPF, HPF, BPF).
2. IIR filter design (Butterworth and Chebyshev).

3. Exercises on Optimum Equiripple FIR Digital Filters

To be proficient in MATLAB programming, students need to design and implement the following Optimum Equiripple FIR digital filters.

1. Equiripple FIR Halfband Filter
2. Equiripple FIR Halfband Interpolator

4. Exercises on FIR Digital Filter Using Window Method

To be proficient in MATLAB programming, students need to design and implement the following:

1. FIR Digital Filter Design using Rectangular Window
2. FIR Digital Filter Design using Hanning Window

5. Exercises on IIR Digital Filter using Butterworth and Bilinear Transformation

To be proficient in MATLAB programming, students need to design and implement the following:

1. Designing a Butterworth Low-pass IIR Filter
2. Designing a Chebyshev Type I and bilinear transformation IIR Filter
3. Designing a Chebyshev Type II and bilinear transformation IIR Filter

6. Exercises on Sampling Rate Conversion

To be proficient in MATLAB programming, students need to perform Implementing sampling rate conversion by decimation, interpolation, and a rational factor.

1. Interpolation for Non-Integer Sampling rate

7. Exercise on Sine Wave Generation

Generation of sine waves using a lookup table and designing of FIR filters using the frequency sampling method in MATLAB.

8. Exercises on Introduction to Image Processing

To be proficient in image processing programming, students be able to compute:

1. Intensity Transformations
2. Image edge-detection
3. Visualizing image in different color spaces
4. Gray scaling of Images
5. To find width and height of an image

9. Exercises on Arithmetic Operations on Images

To be proficient in image processing programming, students be able to perform arithmetic operations:

1. Image Addition
2. Image subtraction
3. Image multiplication
4. Image division

10. Exercises on Histogram of an image

1. Histogram of colour image
2. Intensity Transformation Operations on Images
3. Power-Law (Gamma) Transformation
4. Piecewise-Linear Transformation Functions

11. Exercises on Image Enhancement

1. Spatial Filtering and its Types
2. Averaging an Image
3. Gaussian Blur
4. Median blur
5. Bilateral blur

12. Exercises on Compression of an image

1. Coding Interpixel and Psychovisual Redundancy,
2. LZW Coding

13. Exercises on Image Segmentation

1. Edge Detection with Laplacian Mask for Image Segmentation
2. Image Segmentation Using Hough Transform for Edge Detection

14.Exercises on Morphological Processing for Image Segmentation

1. Morphological Processing for Image Segmentation -Erosion
2. Morphological Processing for Image Segmentation -Dilation

EXERCISES FOR DIGITAL SIGNAL PROCESSING LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

1. Exercises on Discrete-Time Systems

Analyzing the response of discrete-time systems is crucial in understanding how these systems process signals. The response can be determined through various methods, depending on the system's representation (e.g., difference equation, transfer function, state-space model). Here, we'll focus on a common approach using the impulse response and the convolution sum for linear time-invariant (LTI) systems.

1. Linear Convolution
2. Circular Convolution
3. Cyclic Convolution
4. Overlap-Add Method to compute linear convolution
5. Overlap-Save Method to compute linear convolution

1.1 Linear Convolution

Linear convolution is a fundamental operation in signal processing that combines two signals to produce a third signal that represents the amount of overlap between the signals as one signal slides past another.

Consider the two discrete-time signals x[n] = [3,5,2,-1] and h[n] = [2,-1,3].

Perform the following operations using MATLAB.

1. Compute the linear convolution of x[n] and h[n] using MATLAB's built-in conv function.
2. Manually implement a function in MATLAB to perform linear convolution without using the conv function. Your function should take two sequences as inputs and return their linear convolution.
3. Compare the results obtained from MATLAB's built-in conv function and your manual implementation for accuracy.
4. Plot the original signals x[n] and h[n], and the convolution result from the built-in conv function. Label each plot with appropriate titles and axis labels.

Hint

```
x = [3, 5, 2, -1];
h = [2, -1, 3];
y_builtin = conv(x, h);
-----
---
function y = myConv(x, h)
    % Initialize the result vector with zeros of appropriate length
---

    % Perform the convolution operation (fill in the details)
    % Hint: Use nested loops to implement the 'flipping and sliding'
    ---
```

Try:

1. Given two sequences are given as follows: x[n] = [1,-1,2,3] and h[n] = [-1,-2,4] , perform the following tasks.
   a) Compute the linear convolution.
   b) Prove the commutative property of convolution.
   c) If the sequence h[n] is shifted right by 2 units, then find the output sequence.

d) Zero-pad the shorter sequence to make both sequences of equal length, then compute the convolution again with this new length.

1.2 Circular Convolution

Circular convolution, also known as cyclic or periodic convolution, plays a crucial role in digital signal processing, especially in the context of implementing linear convolution using the Fast Fourier Transform (FFT) for efficient computation.
Two sequences are given as follows: x[n] = [4,3,2,1] and h[n] = [1,-1,2,3].

Perform the following operations using MATLAB.

1.2.1 Compute the circular convolution of x[n] and h[n] using MATLAB's built-in cconv function.
1.2.2 Compute the linear convolution for comparison.
1.2.3 Plot the original signals x[n] and h[n], and the circular convolution results. Label each plot withappropriate titles and axis labels for clarity.

1.2.4 Analyze and discuss the difference between the results obtained from the circular convolutionand the linear convolution.

Hint

```
% Task 1: Circular Convolution without Zero-Padding
x = [4, 3, 2, 1];
h = [1, 2, 3];
----
% Task 2: Linear Convolution
y_linear = conv(x, h);
---
```

Try:

1. Given two sequences are given as follows: x[n] = [2,1,2,1] and h[n] = [1,2,3] , perform the following tasks.
   a. Compute the circular convolution.
   b. Compute the linear convolution
   c. Analyze and discuss the difference between the results obtained from the circular convolution (with and without zero-padding) and the linear convolution.

1.3 Cyclic Convolution

Cyclic convolution, also known as circular convolution, involves the convolution of two discrete finite-length sequences that wrap around.
Two sequences are given as follows: x[n] = [1,3,2,1,1] and h[n] = [1,0,-1,2,3]

Perform the following operations using MATLAB.

1.3.1 Compute the cyclic convolution of x[n] and h[n]
1.3.2 Compute the circular and linear convolution for comparison.
1.3.3 Plot the original signals x[n] and h[n], and the cyclic, circular, and linear convolution results. Labeleach plot with appropriate titles and axis labels for clarity.
1.3.4 Analyze and discuss the difference between the results obtained from the cyclic, circular, andlinear convolution and the linear convolution.

Hint

```
% Task 1: Cyclic Convolution without Zero-Padding
x = [4, 3, 2, 1];
h = [1, 2, 3];
----
% Task 2: Linear Convolution
y_linear = conv(x, h);
---
```

Try:

1. Given two sequences are given as follows: x[n] = [2,1,2,3,4,-1,1,2] and h[n] = [1,-1,-2,-5,0,2] , perform the following tasks.
   a. Compute the cyclic and circular convolution.
   b. Compute the linear convolution
   c. Analyze and discuss the difference between the results obtained from the cyclic, circular, and linear convolution (with and without zero-padding) and the linear convolution.

1.4 Overlap-Add Method to Compute Linear Convolution

The Overlap-Add method is used to perform the linear convolution of a long input signal with a finite impulse response (FIR) filter by dividing the input signal into shorter segments.

Given an input signal x[n] and an FIR filter with impulse response h[n], use the Overlap-Add method to compute the convolution [n]=x[n]∗h[n]. Let the input signal be x[n]=[1,2,3,4,5,6,7,8,9] and the impulse response of the FIR filter be h[n]=[1,−1].

Hint

```
% Parameters
x = [1, 2, 3, 4, 5, 6, 7, 8, 9];
h = [1, -1];
L = 4;
M = length(h);
N = L + M - 1;

% Zero-pad h
h_padded = [h, zeros(1, N - M)];

% Initialize output
y = [];
---

% Process each segment
for start = 1:L:length(x)
    segment = x(start:min(start+L-1, length(x)));

--
```

Try:

1.4.1 Let the input signal be x[n]=[2,3,1,−1,2,−2,3,0,1,−1] and the impulse response of the FIR filter be h[n]=[1,0,−1]. Compute the convolution between x[n] and h[n] using Overlap-Add method in MATLAB.

1.4.2 Consider two discrete-time signals, x[n] and h[n], where x[n] is an input signal and h[n] is theimpulse response of a finite impulse response (FIR) filter. The signals are defined as follows:

x[n]=[4,3,2,1,2,3,4]

h[n]=[1,−1]

Using the Overlap-Add method with a segment length L=3, analyze the process of convolving

these two signals in MATLAB.

1.5 Overlap-Save Method for Linear Convolution

The Overlap-Save method is another technique for efficiently computing the convolution of a long input signal with an FIR filter. It is particularly well-suited for real-time processing.

Given an input signal x[n] and an FIR filter with impulse response h[n], use the Overlap-Save method to compute the convolution [n]=x[n]∗h[n] in MATLAB. Let the input signal be x[n]=[3,1,2,2,4,1,2,3,4,1] and the impulse response of the FIR filter be h[n]=[2,1].

Hint

```
% Parameters
x = [3,1,2,2,4,1,2,3,4,1];
h = [2,1];
L = 4;
M = length(h);
N = L + M - 1;

---

% Initialize output
y = [];
---

% Process each segment
for start = 1:L:length(x)
    segment = x(start:min(start+L-1, length(x)));

--
```

Try:

1. Consider two discrete-time signals, x[n] and h[n], where x[n] is an input signal and h[n] is the impulse response of a finite impulse response (FIR) filter. The signals are defined as follows:

   x[n]=[4,5,6,7,8,9,10,11,12,13,14]

   h[n]=[1,-1,2]

   Using the Overlap-Save method with a segment size N=8, analyze the process of convolving these two signals in MATLAB.

2. Exercises on FIR and IIR Filter Design

Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters are two fundamental types of digital filters used in signal processing. Each has distinct characteristics and is designed using different approaches.

2.1 FIR filter design

FIR filters have a finite duration response to an impulse input. This means the impulse response settles to zero in a finite time after the input is applied.

Design a low-pass FIR filter with N= 50 as filter order using the window method with a Hamming window.

Hint

```
% Parameters
N = 50; % Filter order
Fc = 0.3; % Cutoff frequency (normalized)
---
freqz(b,1); % Display frequency response
--
```

Try:

1. Design a multi-band FIR filter and simulate in MATLAB, that passes two distinct frequency bands while attenuating others. This kind of filter is useful in applications like audio processing where you might want to isolate certain frequency ranges.

   Specifications:

   Passbands: 0.2 to 0.3 and 0.5 to 0.6 (normalized frequency, where 1 corresponds to the Nyquist rate).

   Stopbands: 0 to 0.1, 0.4 to 0.5, and 0.7 to 1.

   Passband ripple: ≤ 0.01 dB

   Stopband attenuation: ≥ 60 dB.

2. Design an FIR filter with a very steep transition band, which is a common requirement in applications where It is crucial to minimize interference between adjacent frequency bands, such as in channel selection for telecommunications or isolating specific frequency bands in audio processing. Simulate in MATLAB.

   Specifications:

   Passband: 0 to 0.25 (normalized frequency, where 1 corresponds to the Nyquist rate)

   Stopband: 0.3 to 0.5

   Passband ripple: ≤ 0.5 dB

   Stopband attenuation: ≥ 100 dB

   Transition band: 0.25 to 0.3.

2.2 IIR Filter Design

IIR filters can have an infinite duration response to an impulse input. The impulse response does not become exactly zero after a finite number of steps, theoretically extending indefinitely.

2.3 Butterworth Filter Design

This aims to provide the flattest possible passband response, making it a good choice for applications requiring minimal distortion within the passband. The trade-off is a relatively wide transition band between the passband and stopband.

Design and simulate in MATLAB a low-pass Butterworth IIR filter.
Specifications:

Order of the filter N=5
Cut off frequency Fc=500 Hz
Sampling frequency Fs=1500 Hz

Hint

```
% Parameters
N = 5; % Filter order
Fc = 300; % Cutoff frequency in Hz
Fs = 1000; % Sampling frequency in Hz
----
freqz(b,a); % Display frequency response
```

Try:

2.3.1 Design a 3rd-order Butterworth bandpass filter with passband edges at 500 Hz and 1500 Hz,assuming a sampling frequency of 10 kHz. Plot the magnitude response in MTALB.

2.3.2 Design a 4th-order high-pass Butterworth filter with a cutoff frequency of 2 kHz, assuming asampling frequency of 20 kHz. Plot the magnitude response in MTALB.

2.4 Chebyshev Filter Design

These filters allow for a sharper transition between the passband and stopband compared to Butterworth filters by allowing ripple in either the passband (Type I) or stopband (Type II).

Design a Type I Chebyshev low-pass filter with a 1 dB ripple in the passband, a cutoff frequency of 3 kHz, and a minimum attenuation of 40 dB in the stopband starting at 6 kHz. Assume a sampling frequency of 20 kHz.

Hint

```
% Type1 CB filter design
max_ fc = 3000; % Cutoff frequency in Hz
fs = 6000; % Stopband frequency in Hz
Rp = 1; % Passband ripple in dB
Rs = 40; % Stopband attenuation in dB
Fsample = 20000; % Sampling frequency in Hz
--
title('Type I Chebyshev Low-Pass Filter'); fprintf('The minimum sampling rate required is
```

Try:

2.4.1 Design a Type II Chebyshev high-pass filter with a 20 dB attenuation in the stopband, a stopband cutoff frequency of 500 Hz, and a passband edge starting at 1 kHz with minimal attenuation. Assume a sampling frequency of 5 kHz.

Hint

```
% Type2 CB filter design
fstop = 500; % Stopband cutoff frequency in Hz
fpass = 1000; % Passband edge frequency in Hz
```

```
Rs = 20; % Stopband attenuation in dB
Fsample = 5000; % Sampling frequency in Hz
--
% Determine the filter order and cutoff frequency
--
title('Type II Chebyshev High-Pass Filter');--
```

Try:

1. Design Type I Chebyshev bandpass filter that meets specific group delay requirements. Specifications: Passband: 2 kHz to 4 kHz, Passband Ripple: 3 dB Stopband Attenuation: ≥ 40 db Sampling Frequency: 16 kHz, Group Delay: Minimize variations within the passband.

3. Exercises on Optimum Equiripple FIR Digital Filter

The design of an optimum equiripple Finite Impulse Response (FIR) digital filter involves creating a filter where the magnitude of the ripple is the same (or equiripple) in both the passband and the stopband. This kind of design aims to minimize the maximum error between the desired and the actual filter response, leading to an optimal filter in the sense of minimizing the maximum deviation within the filter's passband and stopband. The Parks-McClellan algorithm, also known as the Remez Exchange algorithm, is a popular method used to design equiripple FIR filters.

Design an optimum equiripple low-pass FIR digital filter with the following specifications:

- Passband edge frequency: 1500 Hz
- Stopband edge frequency: 2000 Hz
- Passband ripple: 1 dB maximum
- Stopband attenuation: 40 dB minimum
- Sampling frequency: 8000 Hz

Hint

```
% EQR FIR lowpass filter design
% Filter specifications
Fs = 8000; % Sampling frequency in Hz
Fpass = 1500; % Passband edge frequency in Hz
Fstop = 2000; % Stopband edge frequency in Hz
Rp = 1; % Passband ripple in dB
Rs = 40; % Stopband attenuation in dB
% Normalize frequencies
--
---
% Frequency response analysis
freqz(b, 1, 1024, Fs)
title('Optimum Equiripple FIR Low-Pass Filter');grid on;
```

Try:

1. Design and simulate in MATLAB an equiripple FIR halfband filter with the following specifications:
   - Sampling frequency: 2000 Hz
   - Transition band: Centered around 2100 Hz (0.25 times the sampling frequency)
   - Passband ripple and Stopband attenuation: Specify as needed for equiripple condition.

3.1 Equiripple FIR Halfband Filter

An equiripple FIR halfband filter is a specific type of filter with the property that every other coefficient (except for the center one) is zero, and the cutoff frequency is at one-quarter of the sampling rate. Halfband filters are efficient in both their design and implementation, especially useful in multirate signal processing for tasks like interpolation and decimation.

Design and simulate in MATLAB an equiripple FIR halfband filter with the following specifications:

Sampling frequency: 8000 Hz

Transition band: Centered around 2000 Hz (0.25 times the sampling frequency)

Passband ripple and Stopband attenuation: Specify as needed for equiripple condition.

Hint

```
% Parameters
% Sampling frequency
Fs = 8000; % Hz
% Define specifications for the halfband filter
--

% Normalize frequencies
Wp = Fpass/(Fs/2);
Ws = Fstop/(Fs/2);
---
% Estimate the filter order and design the filter using the Parks-McClellan
algorithm
[N, Fo, Ao, W] = firpmord([Wp Ws], [1 0], [10^(-Rp/20) 10^(-Rs/20)]);
---
freqz(b, 1, 1024, Fs);
title('Equiripple FIR Halfband Filter')
```

Try:

1. Incorporate a halfband FIR filter into a multirate digital signal processing (DSP) system designed for audio streaming. The system dynamically adjusts between standard (44.1 kHz) and high-resolution (88.2 kHz) audio streams. The halfband filter is used to efficiently transition between these sampling rates.

   Specification:

   Standard Audio Sampling Frequency: 44.1 kHz
   High-Resolution Audio Sampling Frequency: 88.2 kHz

Transition Requirement: Seamless switching between standard and high-resolution audio streams.

3.2 FIR Halfband Interpolator

An FIR (Finite Impulse Response) halfband interpolator is a specialized type of digital filter used to double the sampling rate of a signal. The halfband filter is characterized by its frequency response, where approximately half of its spectrum is passed (the low-pass region) and the other half is attenuated (the stopband region).
The distinctive feature of halfband filters is that their transition band is centered around the Nyquist frequency (Fs/4, where Fs is the new, doubled sampling rate), and they have a symmetric frequency response with respect to Fs/4.

Design an equiripple FIR halfband low-pass filter to be used in a interpolator process, where the goal is to increase the sampling rate of a signal by a factor of 2.
Specifications:

Desired transition band centered around the Nyquist frequency of the original signal.
Passband ripple: 0.01 dB maximum
Stopband attenuation: 80 dB minimum
Original sampling frequency: 32000 Hz

Hint

```
% Parameters
% Filter specifications for polyphase interpolator component
Fs = 32000; % Original sampling frequency
Rp = 0.01; % Passband ripple in dB
Rs = 80; % Stopband attenuation in dB
% Design the equiripple FIR halfband filter
N = 50; % Preliminary filter order, adjust based on design requirements
f = [0 0.25 0.25 0.5]; % Normalized frequency points for halfband
a = [1 1 0 0]; % Desired amplitude response
b = firpm(N, f, a, [1 1]); % Design filter with equal weight in pass and stopband
---
% Frequency response analysis
freqz(b, 1, 1024, Fs*2);
-
title('Equiripple FIR Halfband Filter')
```

Try:

1. Design and simulate in MATLAB: an optimized FIR halfband interpolator for a signal with a maximum frequency component of 750 Hz. Assume the original sampling frequency of the signal is 2000 Hz, and you want to interpolate it to 4000 Hz.

4. Exercises on FIR Digital Filter Using Window Method

4.1 FIR Digital Filter Design using Rectangular Window

The rectangular window, also known as the boxcar or uniform window, is the simplest type of window function used in signal processing.

N = 64; % Window length

rectangularWindow = rectwin(N);

% Plot the rectangular window

stem(rectangularWindow);

title('Rectangular Window');

xlabel('Sample Index');

ylabel('Amplitude');

visualize the frequency response of a rectangular window using the freqz function.

Hint:

N = 64; % Window length

Generate a rectangular window.

Hint:

rectangularWindow = rectwin(N);

% Frequency response of the rectangular window

Fs = 1000; % Sampling frequency

freqResponse = freqz(rectangularWindow, 1, 1024, Fs);

% Plot the frequency response

figure;

subplot(2, 1, 1);

plot(rectangularWindow);

title('Rectangular Window in Time Domain');

xlabel('Sample Index');

ylabel('Amplitude');

subplot(2, 1, 2);

plot(linspace(0, Fs/2, length(freqResponse)), 20*log10(abs(freqResponse)));

title('Frequency Response of Rectangular Window');

xlabel('Frequency (Hz)');

ylabel('Magnitude (dB)');

FIR Filter Design using Rectangular Window Method

Try:

1. Design a digital FIR lowpass filter with the following specifications: wp = 0.2π, ws = 0.3π, Rp = 0.25dB As = 50dB using rectangular window function. Determine the impulse response and provide a plot of the frequency response of the designed filter using MATLAB.

4.2 FIR Digital Filter Design using Hanning Window

The Hanning window, is characterized by a central lobe with smaller sidelobes compared to the rectangular window. visualize the frequency response of the Hanning window using the freqz function

Hint

```
N = 64; % Window length
hanningWindow = hann(N);
% Frequency response of the Hanning window
Fs = 1000; % Sampling frequency
freqResponse = freqz(... ..... );
% Plot the frequency response
figure;
subplot(2, 1, 1);
plot(... ..... );
```

Try:

4.2.1 Design a low-pass FIR filter with a specified cutoff frequency using the Hanning window method, and Analyze its frequency and time-domain characteristics. Sampling frequency = 1000 Hz, Cut off frequency = 150 Hz, Order of the Filter = 50

4.2.2 Using MATLAB program, demonstrate the effect of the Hanning window in reducing spectral leakage by comparing the frequency spectra of a sinusoidal signal when windowed with a Hanning window versus a rectangular window.

Generate a sinusoidal signal with a frequency that does not exactly match the FFT bin frequencies (e.g., f=50.5 Hz, sampling frequency fs=1024 Hz, and a duration of 1 second).

Compute and plot the magnitude spectrum of the signal without any windowing. Apply a Hanning window to the signal and then compute and plot the magnitude spectrum again. Compare the results and discuss the effect of the Hanning window on spectral leakage.

5. Exercises on IIR Digital Filter using Butterworth, Chebyshev,and Bilinear Transformation

Designing an IIR (Infinite Impulse Response) digital filter using the Butterworth method and bilinear transformation involves several steps. The Butterworth filter is known for its maximally flat frequency response in the passband, making it a popular choice for many filtering applications. The bilinear transformation method is used to convert an analog filter design into a digital one without introducing aliasing and to preserve the frequency response characteristics as closely as possible.

5.1 Designing a Butterworth Low-pass IIR Filter

Design a digital low-pass filter using the Butterworth design approach and bilinear transformation
Specifications:

Passband frequency, fp=1kHz
Stopband frequency, fs=1.5kHz

Passband ripple, δp=1dB (maximum allowable attenuation in the passband)
Stopband attenuation, δs=40dB (minimum attenuation in the stopband)
Sampling frequency, fsamp=8kHz

Hint

```
% Butterworth Low-Pass IIR Filter Design Using Bilinear Transformation

% Specifications

fp = 1000; % Passband frequency in Hz

fs = 1500; % Stopband frequency in Hz

delta_p = 1; % Passband ripple in dB

delta_s = 40; % Stopband attenuation in dB

f_samp = 8000; % Sampling frequency in Hz

--

--
```

Try:

1. Design a Butterworth low-pass filter to smooth an image by attenuating high-frequency components.

   Specifications:
   - Passband frequency: 20 Hz
   - Stopband frequency: 50 Hz
   - Passband ripple: 1 dB
   - Stopband attenuation: 20 dB
   - Sampling frequency: 100 Hz

5.2 Designing a Chebyshev Type I and bilinear transformation IIR Filter

Designing an IIR (Infinite Impulse Response) filter using a Chebyshev Type I design approach and bilinear transformation involves specific steps. Chebyshev Type I filters allow for a ripple in the passband but have a steeper roll-off compared to Butterworth filters, making them useful in applications where the steepness of the filter's transition band is more critical than the flatness of the passband.

Design a Chebyshev Type I low-pass filter with the following specifications:

- Passband frequency fp: 2 kHz
- Stopband frequency fs: 2.5 kHz
- Passband ripple δp: 1 dB
- Stopband attenuation δs): 40 dB
- Sampling frequency fsamp: 16 kHz

Hint

```
% Filter Specifications
fp = 2000; % Passband frequency in Hz
fs = 2500; % Stopband frequency in Hz
rp = 1; % Passband ripple in dB
rs = 40; % Stopband attenuation in dB
fsamp = 16000; % Sampling frequency in Hz
--
```

Try:

1. Design a Chebyshev Type I low-pass filter to remove high-frequency noise from an Electrocardiogram (ECG) signal.

   Specifications:

   - Passband frequency: 40 Hz
   - Stopband frequency: 100 Hz
   - Passband ripple: 1 dB
   - Stopband attenuation: 20 dB
   - Sampling frequency: 1 kHz

5.3 Designing a Chebyshev Type II and bilinear transformation IIR Filter

Designing an IIR (Infinite Impulse Response) filter using a Chebyshev Type II design approach and bilinear transformation.

Design a Chebyshev Type II low-pass filter with the following specifications:

- Passband frequency fp: 1 kHz
- Stopband frequency fs: 1.5 kHz
- Passband ripple δp: 0.5 dB
- Stopband attenuation δs): 40 dB
- Sampling frequency fsamp: 8 kHz

Hint

```
% Filter Specifications
fp = 1000; % Passband frequency in Hz
fs = 1500; % Stopband frequency in Hz
rp = 0.5; % Passband ripple in dB
rs = 40; % Stopband attenuation in dB
fsamp = 8000; % Sampling frequency in Hz
--
```

Try:

5.3.1 Design a Chebyshev Type II low-pass filter to enhance speech signals by attenuating high-frequency noise components.

Specifications:
Passband frequency: 3.4 kHz

Stopband frequency: 4 kHz (
Passband ripple: 0.5 dB
Stopband attenuation: 50 dB
Sampling frequency: 16 kHz

5. Exercises on Sampling Rate Conversion

Implementing sampling rate conversion by decimation, interpolation, and a rational factor in MATLAB involves several steps. The process is aimed at converting the sampling rate of a signal from one rate to another, which is specified by a rational factor. The rational factor is represented by two integers, L and M, where L is the interpolation factor (number of samples to insert between existing samples) and M is the decimation factor (number of samples to remove).

Hint

```
% Generate a sample signal (e.g., a sine wave)
Fs = 1000; % Original sampling rate in Hz
t = 0:1/Fs:1-1/Fs; % Time vector
x = sin(2*pi*100*t); % Sine wave at 100 Hz
L = 2; % Interpolation factor
M = 3; % Decimation factor –
----Interpolation
--- decimation
```

Try:

1. Using MATLAB program, generate a cosine signal with a frequency of 100 Hz, sampled at 1000 Hz. Interpolate the signal by a factor of 2, Decimate the interpolated signal by a factor of 2. Plot the original, interpolated, and decimated signals in different subplots and observe the changes.
2. Perform sampling rate conversion using a non-integer (rational) factor. Generate a signal with multiple frequency components. Choose a target sampling rate that is not an integer multiple of the original rate (e.g., convert from 1000 Hz to 750 Hz). Calculate the appropriate interpolation and decimation factors to achieve the desired conversion. Interpolate, filter, and decimate the signal according to the calculated factors. Plot and compare the original and converted signals in both time and frequency domains.

5.1 Interpolation for Non-Integer Sampling rate

Interpolation for non-integer sampling rate conversion is a critical operation in digital signal processing (DSP) for modifying the sampling rate of a signal. This process is particularly important when you need to convert a signal from one sampling rate to another that is not an integer multiple of the original rate, a common requirement in many audio and telecommunications applications.

Write a MATLAB function to perform interpolation by a non-integer factor on a given signal.

Hint

```
% New (non-integer) sampling rate
function interpolatedSignal = interpolateSignal(inputSignal, originalFs, targetFs)
---
% Example usage
fs = 100; % original sampling rate in Hz
t = 0:1/fs:1-1/fs; % time vector
x = cos(2*pi*10*t); % example signal
--


% Plot original and interpolated signals
```

Try:

1. x(n) = cos(πn). Generate samples of x(n) and interpolate them using I = 2, 4, and 8 . Plot originalsignal and sampled signal.

7. Exercises on Introduction to Image Processing

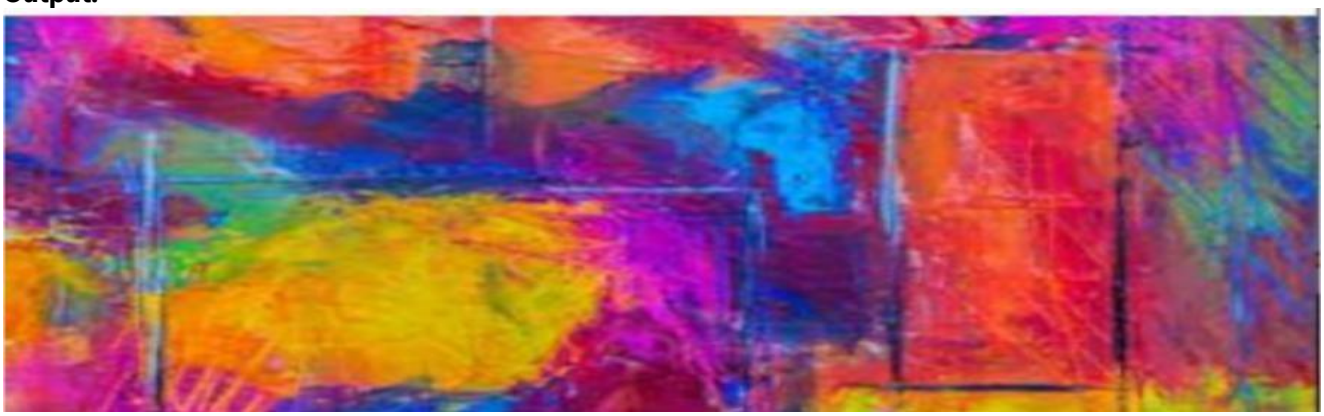
7.1Intensity Transformations

Intensity transformations are applied on images for contrast manipulation or image thresholding. These are in the spatial domain, i.e. they are performed directly on the pixels of the image at hand, as opposed to being performed on the Fourier transform of the image. An image may be defined as a two- dimensional function f(x,y), where x and y are spatial coordinates, and the value of f at any pair of coordinates (x,y) is called the intensity of the image at that point. For a gray-range image, the intensity is given by just one value.

Input:

An image edge-detection algorithm detects the edges in each channel of an image using differences in the intensity values of pixel neighborhoods. Given an input image the task is to prepare this image to feed it to the algorithm for execution with high accuracy and minimal computational complexity.

Output:



```
import cv2
fname = input("Enter the path of the image to display: ") img_arr =
cv2.imread(fname)
cv2.imshow("Displaying Image", img_arr)
cv2.waitKey(0) #wait until a key is pressed
cv2.destroyAllWindows()
```

Try: Produce the code to display the image in different color domains

7.2 Image edge-detection

Intensity transformations are applied on images for contrast manipulation or image thresholding. These are in the spatial domain, i.e. they are performed directly on the pixels of the image at hand, as opposed to being performed on the Fourier transform of the image. An image may be defined as a two- dimensional function f(x,y), where x and y are spatial coordinates, and the value of f at any pair of coordinates (x,y) is called the intensity of the image at that point. For a gray-range image, the intensity is given by just one value.

Input:

An image edge-detection algorithm detects the edges in each channel of an image using differences in the intensity values of pixel neighborhoods. Given an input image the task is to prepare this image to feed it to the algorithm for execution with high accuracy and minimal computational complexity.

Output:

```
import cv2
fname = input("Enter the path of the image to display: ") img_arr =
cv2.imread(fname)
cv2.imshow("Displaying Image", img_arr)
cv2.waitKey(0) #wait until a key is pressed
cv2.destroyAllWindows()
```

Try: Produce the code to display the image in different color domains

7.3 Visualizing image in different color spaces

OpenCV (Open Source Computer Vision) is a computer vision library that contains various functions to perform operations on pictures or videos. It was originally developed by Intel but was later maintained by Willow Garage and is now maintained by Itseez. This library is cross-platform that is it is available on multiple programming languages such as Python, C++ etc. RGB image is represented by linear combination of 3 different channels which are R(Red), G(Green) and B(Blue). Pixel intensities in this color space are represented by values ranging from 0 to 255 for single channel. Thus, number of possibilities for one color represented by a pixel is 16 million approximately [255 x 255 x 255].

Input:

An image edge-detection algorithm detects the edges in each channel of an image using differences in the intensity values of pixel neighborhoods. Given an input image the task is to prepare this image to feed it to the algorithm for execution with high accuracy and minimal computational complexity.

Output:

```
import cv2
img = cv2.imread('g4g.png') plt.imshow(img)
```

Input:

Gray Scale Image :
Grayscale image contains only single channel. Pixel intensities in this color space is represented by values ranging from 0 to 255. Thus, number of possibilities for one color represented by a pixel is 256. **Output:**

```
import cv2
img = cv2.imread('g4g.png', 0)
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) cv2.imshow('image',
img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Try: Produce the code to display the image in different color domains

Input:

YCrCb Color Space :
Y represents Luminance or Luma component, Cb and Cr are Chroma components. Cb represents the blue-difference (difference of blue component and Luma Component). Cr represents the red- difference (difference of red component and Luma Component).

Output:

```
import cv2
img = cv2.imread('g4g.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV) cv2.imshow('image',
img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Input:

LAB color space :
L – Represents Lightness.
A – Color component ranging from Green to Magenta.
B – Color component ranging from Blue to Yellow.

Output:

```
import cv2
```

```
img = cv2.imread('g4g.png')
laplacian = cv2.Laplacian(img, cv2.CV_64F)
cv2.imshow('EdgeMap', laplacian)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Input:

HSV color space :
H : Hue represents dominant wavelength. S : Saturation represents shades of color. V : Value represents Intensity.

Output:

```
import cv2
img = cv2.imread('g4g.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV) cv2.imshow('image',
img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Try: Produce the code to display the image in Edge map of image and Heat map of image

7.4 Gray scaling of Images

Gray scaling is the process of converting an image from other color spaces e.g. RGB, CMYK, HSV, etc. to shades of gray. It varies between complete black and complete white.

Dimension reduction: For example, In RGB images there are three color channels and three dimensions while gray scale images are single-dimensional.

Reduces model complexity: Consider training neural articles on RGB images of 10x10x3 pixels. The input layer will have 300 input nodes. On the other hand, the same neural network will need only 100 input nodes for gray scale images.

Input:

Output:

```
import cv2
image = cv2.imread('C:\\Documents\\full_path\\tomatoes.jpg')
cv2.imshow('Original', image)
cv2.waitKey(0)
# Use the cvtColor() function to grayscale the image gray_image =
cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) cv2.imshow('Grayscale',
gray_image)
cv2.waitKey(0) cv2.imshow('Gray
# Window shown waits for any key pressing event
cv2.destroyAllWindows()
```

Method 2:

Input:

Output:

```
import cv2
# Use the second argument or (flag value) zero
# that specifies the image is to be read in grayscale mode
img = cv2.imread('C:\\Documents\\full_path\\tomatoes.jpg', 0)
cv2.imshow('Grayscale Image', img)
cv2.waitKey(0)
# Window shown waits for any key pressing event
cv2.destroyAllWindows()
```

Method3:

Output:

```
import cv2
# Load the input image
img = cv2.imread('C:\\Documents\\full_path\\tomatoes.jpg') #
  Obtain the dimensions of the image array
# using the shape method
# Take the average of pixel values of the BGR Channels # to
convert the colored image to grayscale image
for i in range(row): for
    j in range(col):
        # Find the average of the BGR pixel values img[i, j]
        = sum(img[i, j]) * 0.33
  cv2.imshow('Grayscale Image', img)
cv2.waitKey(0)
  # Window shown waits for any key pressing event
cv2.destroyAllWindows()
```

Try:Convert an Image from RGB to Grayscale in Python

7.5 To find width and height of an image

In order to find the height and width of an image, there are two approaches. The first approach is by using the PIL(Pillow) library and the second approach is by using the Open-CV library.

PIL.Image.open() is used to open the image and then .width and .height property of Image are used to get the height and width of the image. The same results can be obtained by using .size property.

To use pillow library run the following command: pip

install pillow

Input:

Output: The height of the Image is The
Width of the Image is

```
# import required module from
PIL import Image
 # get image
filepath = "geeksforgeeks.png" img =
Image.open(filepath)
# get width and height
# display width and height
print("The height of the image is: ", height)
print("The width of the image is: ", width)
```

Second Method:

```
# import required module from
PIL import Image

# get image
filepath = "geeksforgeeks.png" img =
Image.open(filepath)

# get width and height
width,height = img.size
# display width and height
print("The height of the image is: ", height)
print("The width of the image is: ", width)
```

Output: The height of the Image is The
Width of the Image is

Third Method:

The imread(filepath) function is used to load an image from the file path specified. The .shape stores a tuple of height, width and no of channels for each pixel. The .shape[:2] will get the height and width of the image.

```
# import required module
import cv2

# get image
filepath = "geeksforgeeks.jpg"
image = cv2.imread(filepath)
```

```
#print(image.shape)

# get width and height
height, width = image.shape[:2]

# display width and height
print("The height of the image is: ", height)
print("The width of the image is: ", width)
```

Output: The height of the Image is The
Width of the Image is

Try: Find the Size (Resolution) of an Image

8. Exercises on Arithmetic Operations on Images

Image arithmetic refers to the arithmetic operations on images. Whenever we perform any arithmetic operation on an image, it is performed on individual pixel values. To perform any arithmetic operation on an image first, we have to load the image using the cv2.imread() method.

If the arithmetic operations are performed on two or more images then all the images should be of the same type like jpeg, jpg, png, etc.,.

8.1 Image Addition

We can either add two images or add a constant value to an image. Image addition is commonly used as an intermediate step in some complicated processes rather than as a useful operation on its own. We can perform image addition in two ways:

NumPy Addition: In this, we simply load the image files and add the NumPy N-d arrays returned after loading the images using the (+) operator. It is a modulo operation that means if the resultant pixel value is greater than 255 after the addition of the pixel values of the input (loaded) images then modulo (%) of the resultant pixel value with 256 (for 8-bit image format) is calculated and assigned to the resultant pixel value to keep it below 255 or 255 as any pixel value cannot exceed 255. For example: 250+10 = 260 => 260 % 256 = 4

Input:

Sample Image 1

Sample Image 2

Output:

Output Image

```
img1 = cv2.imread('sample-img-1.jpg')
img2 = cv2.imread('sample-img-2.jpg') #
Applying NumPy addition on images #
Saving the output image
cv2.imwrite('output.jpg', fimg)
```

OpenCV addition: In this, we simply load the image files and pass the NumPy N-d arrays returned after loading the images to the cv2.add() method as arguments. It is a saturated operation that means if the resultant pixel value is greater than 255 after the addition of the pixel values of the input (loaded) images then it is saturated to 255 so that any pixel value cannot exceed 255. This is called
**saturation. For example: 250+10 = 260 => 255.

Output:

Output Image

```
# Reading image files
img1 = cv2.imread('sample-img-1.jpg')
img2 = cv2.imread('sample-img-2.jpg')

# Applying OpenCV addition on images
fimg = cv2.add(img1, img2)

# Saving the output image
cv2.imwrite('output.jpg', fimg)
```

8.2 Image subtraction

Image subtraction is simply the pixel subtraction that takes two images as input and produces a third image as output whose pixel values are simply those of the first image minus the corresponding pixel values from the second image. We can also use a single image as input and subtract a constant value from all its pixel values. Some versions of the operator will output the absolute difference between pixel values, rather than the straightforward signed output.

NumPy Subtraction and OpenCV Subtraction.

We will only use the OpenCV subtraction as it produces better results and is widely used. The cv2.subtract() method is used for image subtraction and the result will be like res = img1 - img2 where img1 & img2 are the images of the same depth and type.

Image subtraction is used both as an intermediate step in complicated image processing techniques and also as an important operation on its own. One most common use of image subtraction is to subtract background variations in illumination from a scene so that the objects in foreground can be analyzed more easily and clearly.

Output Image

```
# Reading image files
img1 = cv2.imread('sample-img-1.jpg')
img2 = cv2.imread('sample-img-2.jpg')

# Applying OpenCV subtraction on images
fimg = cv2.subtract(img1, img2)

# Saving the output image
cv2.imwrite('output.jpg', fimg)
```

8.3 Image multiplication

Like other arithmetic operations on images, image multiplication can also be implemented in forms. The first form of image multiplication takes two input images and produces an output image in which the pixel values are the product of the corresponding pixel values of the input images and the second form takes a single input image and produces output in which each pixel value is the product of the corresponding pixel values of the input image and a specified constant (scaling factor). This second form of image multiplication is more widely used and is generally called scaling.

Input:

Sample Image

```
# Reading image file
img = cv2.imread('sample_img.jpg')

# Applying NumPy scalar multiplication on image
fimg = img * 1.5

# Saving the output image
cv2.imwrite('output.jpg', fimg)
```

Output Image

Output:

OpenCV image multiplication using the cv2.multiply() method which usually takes either two image arrays or one image array and one specified constant.

```
# Reading image file
img = cv2.imread('sample_img.jpg')

# Applying OpenCV scalar multiplication on image
fimg = cv2.multiply(img, 1.5)

# Saving the output image
cv2.imwrite('output.jpg', fimg)
```

Output:

Output Image

8.4 Image division

The image division operation normally takes two images as input and produces a third image whose pixel values are the pixel values of the first image divided by the corresponding pixel values of the second image.

It can also be used with a single input image, in which case every pixel value of the image is divided by a specified constant.

Image division operation can be used for change detection like a subtraction but instead of giving the absolute change for each pixel value from one image to another, division operation gives the fractional change or ratio between corresponding pixel values.

```
# Reading image file
img = cv2.imread('sample_img.jpg')

# Applying NumPy scalar division on image
fimg = img / 2

# Saving the output image
cv2.imwrite('output.jpg', fimg)
```

Output:

Output Image

Try: Produce the code to perform the arithmetic operations by converting the image into pixels, the output of arithmetic operations should be in terms of pixels.

9. Exercises on Histogram of an image

Histogram is considered as a graph or plot which is related to frequency of pixels in an Gray Scale Image with pixel values (ranging from 0 to 255). Grayscale image is an image in which the value of each pixel is a single sample, that is, it carries only intensity information where pixel value varies from 0 to 255. Images of this sort, also known as black-and-white, are composed exclusively of shades of gray, varying from black at the weakest intensity to white at the strongest where Pixel can be considered as a every point in an image.

Use the cv2.calcHist() function to compute the histogram of an image. We can use this function to compute the histogram of a region of the image. To compute a histogram of a region in the image first we define a mask. The white color in the mask is for regions to examine in the original input image and the black color in the mask image is for regions to ignore. Now we calculate the histogram passing this mask as a parameter to the function.

Steps:

Import the required libraries OpenCV, NumPy and matplotlib. Make sure you have already installed them.

Read the input image using cv2.imread() method. Specify the full path of the image.

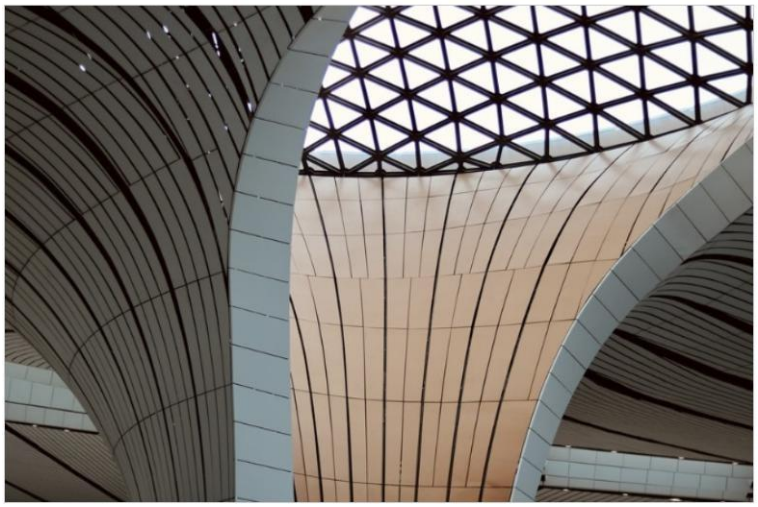
Define a mask for our image. The black color in the mask image is for regions to ignore and white for regions to examine in the original input image.

Split the different channels (blue, green and red) of the input image using cv2.split() function.

Compute the histograms of different channels of the input image using the above defined mask. Plot the histogram of different colors of the input image.

To visualize the masked region of the input image perform cv2.bitwise_and() operation on input image with mask image. It creates a masked region of input image.

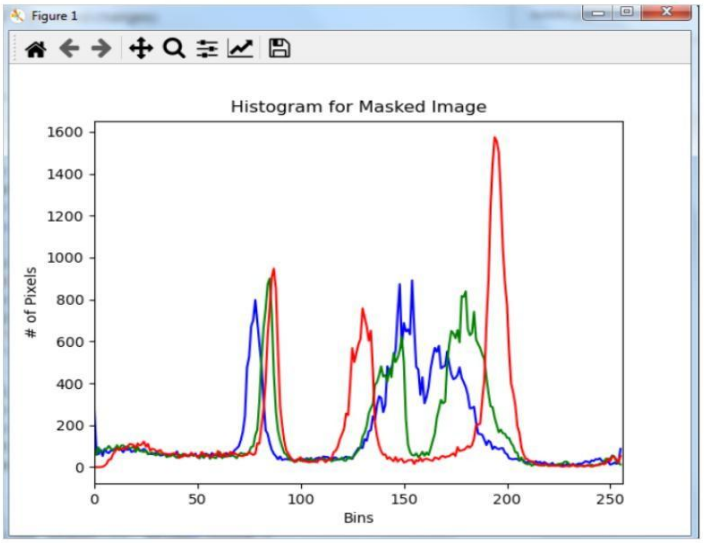
Input:

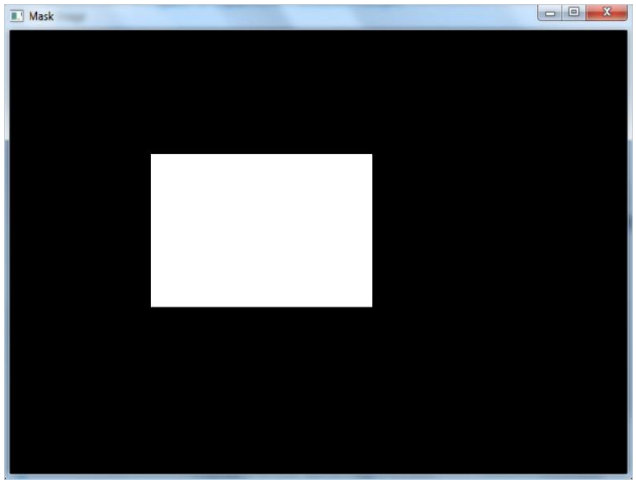


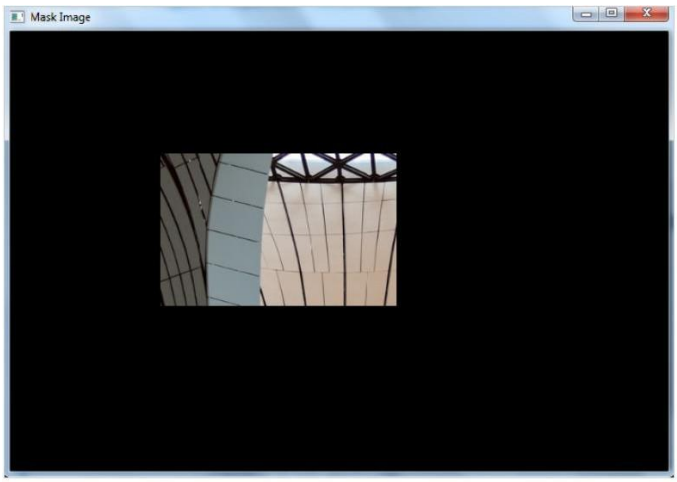
```
# import required libraries
import cv2
from matplotlib import pyplot as plt
import numpy as np

# Read the input image
img = cv2.imread('architecture2.jpg')

# define a function to compute and plot histogram
def plot_histogram(img, title, mask=None):
   # split the image into blue, green and red channels
```

```
  channels = cv2.split(img)
  colors = ("b", "g", "r")
  plt.title(title)
  plt.xlabel("Bins")
  plt.ylabel("# of Pixels")
  # loop over the image channels
  for (channel, color) in zip(channels, colors):
    # compute the histogram for the current channel and plot it
    hist = cv2.calcHist([channel], [0], mask, [256], [0, 256])
    plt.plot(hist, color=color)
    plt.xlim([0, 256])
# define a mask for our image; black for regions to ignore

# and white for regions to examine
mask = np.zeros(img.shape[:2], dtype="uint8")
cv2.rectangle(mask, (160, 130), (410, 290), 255, -1)

# display the masked region
masked = cv2.bitwise_and(img, img, mask=mask)

# compute a histogram for masked image
plot_histogram(img, "Histogram for Masked Image", mask=mask)

# show the plots
plt.show()
cv2.imshow("Mask", mask)
cv2.imshow("Mask Image", masked)
cv2.waitKey(0)
```

Output:





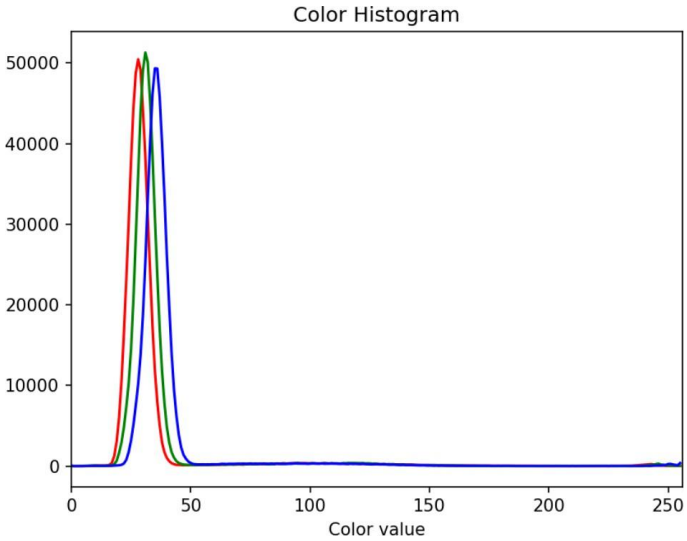


Try : How to plot color channels Histogram of an Image in Python using OpenCV.

9.1 Histogram of colour image

```
# read original image, in full color
plant_seedling = iio.imread(uri="data/plant-seedling.jpg")

# display the image
fig, ax = plt.subplots()
plt.imshow(plant_seedling)
# tuple to select colors of each channel line
colors = ("red", "green", "blue")

# create the histogram plot, with three lines, one for
# each color
plt.figure()
plt.xlim([0, 256])
for channel_id, color in enumerate(colors):
    histogram, bin_edges = np.histogram(
        plant_seedling[:, :, channel_id], bins=256, range=(0, 256)
    )
    plt.plot(bin_edges[0:-1], histogram, color=color)

plt.title("Color Histogram")
plt.xlabel("Color value")
plt.ylabel("Pixel count")
```

Output:



Try: Produce a code to find intensity and frequency of a histogram image.

9.2 Intensity Transformation Operations on Images

Intensity transformations are applied on images for contrast manipulation or image thresholding. These are in the spatial domain, i.e. they are performed directly on the pixels of the image at hand, as opposed to being performed on the Fourier transform of the image.

The following are commonly used intensity transformations:

i). Image Negatives (Linear)

ii). Log Transformations

iii). Power-Law (Gamma) Transformations

iv). Piecewise-Linear Transformation Functions

When we store an image in computers or digitally, it's corresponding pixel values are stored. So, when we read an image to a variable using OpenCV in Python, the variable stores the pixel values of the image. When we try to negatively transform an image, the brightest areas are transformed into the darkest and the darkest areas are transformed into the brightest.

A color image stores 3 different channels. They are red, green and blue. That's why color images are also known as RGB images. So, if we need a negative transformation of an image then we need to invert these 3 channels.

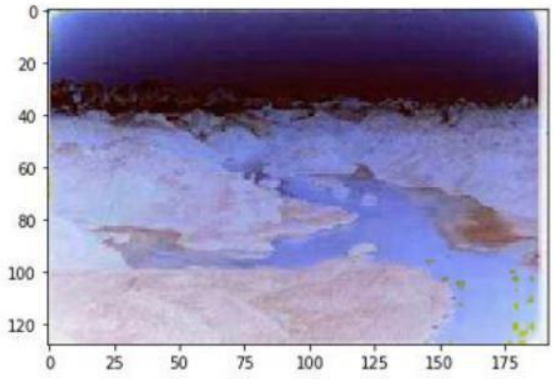
Negative transformation of the image

Let's create a negative transformation of the image. There 2 different ways to transform an image to negative using the OpenCV module. The first method explains negative transformation step by step and the second method explains negative transformation of an image in single line.

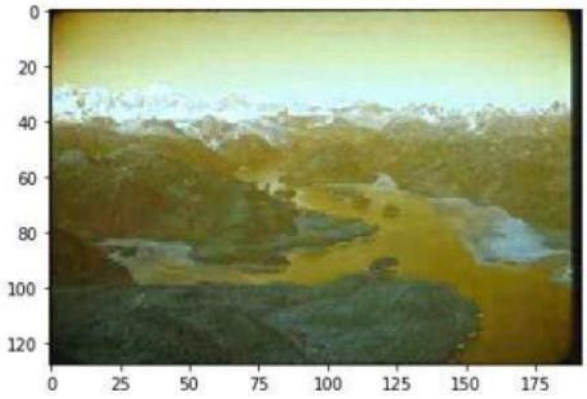
Steps for negative transformation

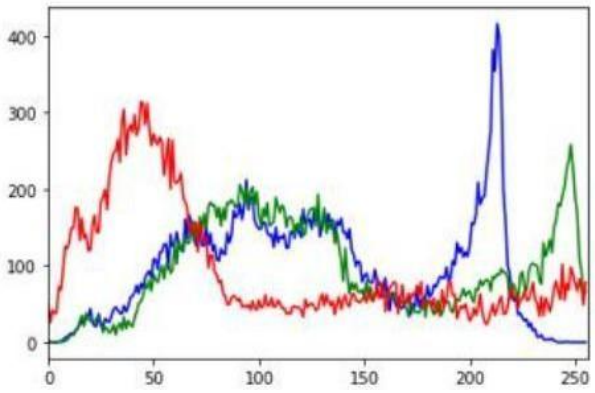
I. Read an image
II. Get height and width of the image
III. Each pixel contains 3 channels. So, take a pixel value and collect 3 channels in 3 different variables.
IV. Negate 3 pixels values from 255 and store them again in pixel used before.
V. Do it for all pixel values present in image.

Input:



Output:





```
import cv2
import matplotlib.pyplot as plt #
    Read an image
img_bgr = cv2.imread('scenary.jpg', 1)
plt.imshow(img_bgr)
plt.show()
  # Histogram plotting of the image
color = ('b', 'g', 'r')

for i, col in enumerate(color):
        histr = cv2.calcHist([img_bgr], [i],
                             None,
                             [256],
                             [0, 256])

    plt.plot(histr, color = col)

    # Limit X - axis to 256
    plt.xlim([0, 256])
    plt.show()
  # get height and width of the image height,
width, _ = img_bgr.shape

for i in range(0, height - 1): for j
    in range(0, width - 1):

        # Get the pixel value
        pixel = img_bgr[i, j]

        # Negate each channel by #
        subtracting it from 255

        # 1st index contains red pixel
        pixel[0] = 255 - pixel[0]

        # 2nd index contains green pixel pixel[1]
        = 255 - pixel[1]

        # 3rd index contains blue pixel
        pixel[2] = 255 - pixel[2]

        # Store new values in the pixel
        img_bgr[i, j] = pixel
  # Display the negative transformed image
plt.imshow(img_bgr)
plt.show()
```

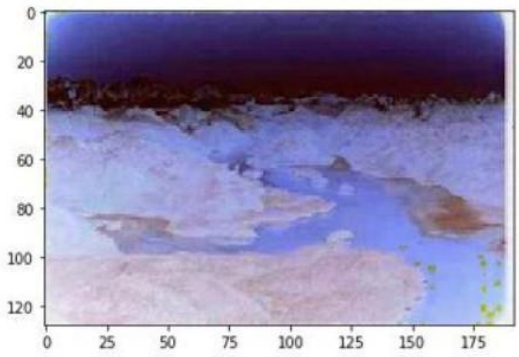
```
  # Histogram plotting of the
# negative transformed image
color = ('b', 'g', 'r')
  for i, col in enumerate(color):
        histr = cv2.calcHist([img_bgr],
                     [i], None,
                     [256],
                     [0, 256])

    plt.plot(histr, color = col)
    plt.xlim([0, 256])
```

Try:

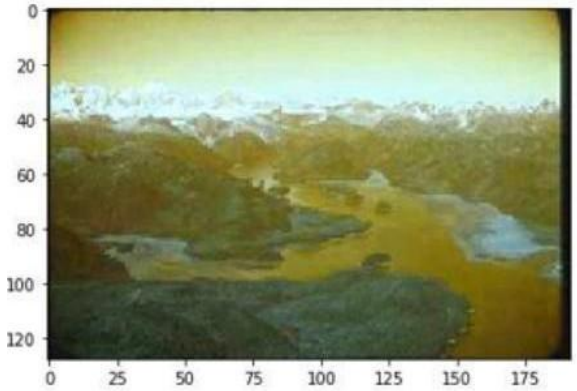
Second method:

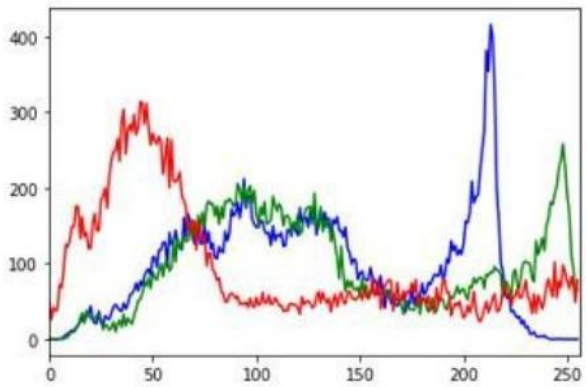
I. Read an image and store it in a variable.
II. Subtract the variable from 1 and store the value in another variable.
III. All done. You successfully done the negative transformation.

Input:



Output:





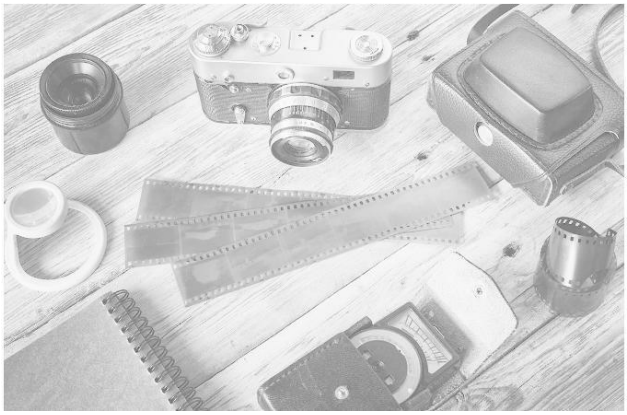
```
import cv2
import matplotlib.pyplot as plt

  # Read an image
img_bgr = cv2.imread('scenary.jpg', 1)

plt.imshow(img_bgr)
plt.show()

# Histogram plotting of original image color =
('b', 'g', 'r')

for i, col in enumerate(color): histr =

    cv2.calcHist([img_bgr],
                         [i], None,
                         [256],
                         [0, 256])

    plt.plot(histr, color = col) #
          Limit X - axis to 256
    plt.xlim([0, 256])
      plt.show()
  # Negate the original image img_neg
= 1 - img_bgr

plt.imshow(img_neg)
plt.show()

# Histogram plotting of
# negative transformed image
color = ('b', 'g', 'r')

for i, col in enumerate(color): histr =

    cv2.calcHist([img_neg],
                         [i], None,
                         [256],
                         [0, 256])

    plt.plot(histr, color = col) plt.xlim([0,
    256])
 plt.show()
```

9.3 Log Transformations

Mathematically, log transformations can be expressed as s = clog(1+r). Here, s is the output intensity, r>=0 is the input intensity of the pixel, and c is a scaling constant. c is given by 255/(log (1 + m)), where m is the maximum pixel value in the image. It is done to ensure that the final pixel value does not exceed (L-1), or 255. Practically, log transformation maps a narrow range of low-intensity input values to a wide range of output values.

Input:

Output:



```
import cv2
import numpy as np

# Open the image.
img = cv2.imread('sample.jpg')

# Apply log transform.
c = 255/(np.log(1 + np.max(img)))
log_transformed = c * np.log(1 + img)

# Specify the data type.
log_transformed = np.array(log_transformed, dtype = np.uint8)

# Save the output.
```

```
cv2.imwrite('log_transformed.jpg',  log_transformed)
```

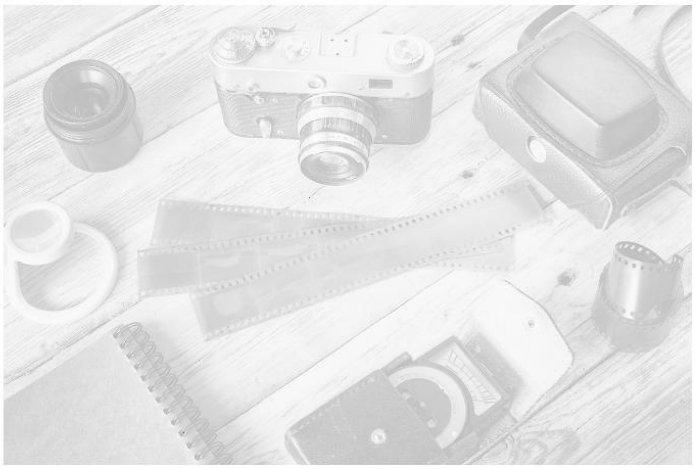
Try: find the log value of a number using Python

9.4 Power-Law (Gamma) Transformation

Power-law (gamma) transformations can be mathematically expressed as s = cr^γ{\gamma}. Gamma correction is important for displaying images on a screen correctly, to prevent bleaching or darkening of images when viewed from different types of monitors with different display settings. This is done because our eyes perceive images in a gamma-shaped curve, whereas cameras capture images in a linear fashion.

Output:

Gamma = 0.1:



```
import cv2
import numpy as np

# Open the image.
img = cv2.imread('sample.jpg')

# Trying 4 gamma values.
for gamma in [0.1, 0.5, 1.2, 2.2]:

    # Apply gamma correction.
    gamma_corrected = np.array(255*(img / 255) ** gamma, dtype = 'uint8')

    # Save edited images.
    cv2.imwrite('gamma_transformed'+str(gamma)+'.jpg', gamma_corrected)
```

Try: Apply Power Law transformation to an Image and produce the code get the output for different values of gamma 0.5, 1.2, 2.2.

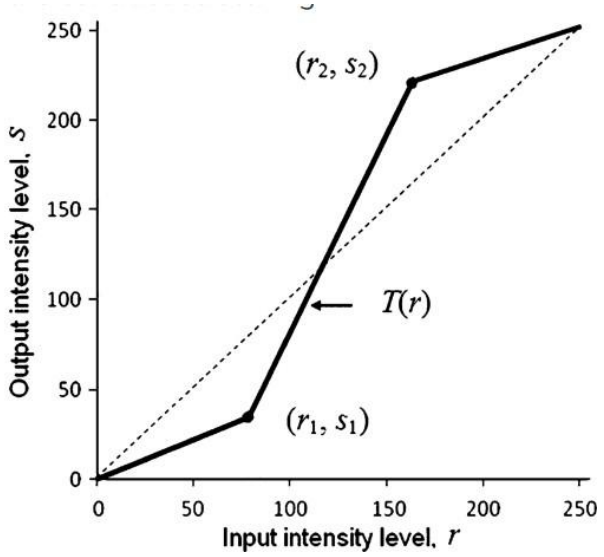
9.5Piecewise-Linear Transformation Functions

One of the most commonly used piecewise-linear transformation functions is contrast stretching. Contrast can

be defined as:

Contrast = (I_max - I_min)/(I_max + I_min)

This process expands the range of intensity levels in an image so that it spans the full intensity of the camera/display. The figure below shows the graph corresponding to the contrast stretching.



With (r1, s1), (r2, s2) as parameters, the function stretches the intensity levels by essentially decreasing the intensity of the dark pixels and increasing the intensity of the light pixels. If r1 = s1 = 0 and r2 = s2 = L-1, the function becomes a straight dotted line in the graph (which gives no effect). The function is monotonically increasing so that the order of intensity levels between pixels is preserved.

Output:

```
import cv2
import numpy as np

# Function to map each intensity level to output intensity level.
def pixelVal(pix, r1, s1, r2, s2):
    if (0 <= pix and pix <= r1):
        return (s1 / r1)*pix
    elif (r1 < pix and pix <= r2):
        return ((s2 - s1)/(r2 - r1)) * (pix - r1) + s1
    else:
        return ((255 - s2)/(255 - r2)) * (pix - r2) + s2
  # Open the image.
img = cv2.imread('sample.jpg')

# Define parameters.
r1 = 70
s1 = 0
r2 = 140
s2 = 255
  # Vectorize the function to apply it to each value in the Numpy array.
pixelVal_vec = np.vectorize(pixelVal)

# Apply contrast stretching.
contrast_stretched = pixelVal_vec(img, r1, s1, r2, s2)

# Save edited image.
cv2.imwrite('contrast_stretch.jpg', contrast_stretched)
```

Try: Apply piecewise linear fit in Python

10. Exercises on Image Enhancement

10.1 Spatial Filtering and its Types

Spatial Filtering technique is used directly on pixels of an image. Mask is usually considered to be added in size so that it has specific center pixel. This mask is moved on the image such that the center of the mask traverses all image pixels.

Classification on the basis of linearity:
There are two types:

1. Linear Spatial Filter
2. Non-linear Spatial Filter

General Classification:

Smoothing Spatial Filter: Smoothing filter is used for blurring and noise reduction in the image. Blurring is pre-processing steps for removal of small details and Noise Reduction is accomplished by blurring.

To smoothen an image with a custom-made kernel we are going to use a function called filter2D() which basically helps us to convolve a custom-made kernel with an image to achieve different image filters like sharpening and blurring and more.

Syntax: filter2D(sourceImage, ddepth, kernel)

Input:

Output:

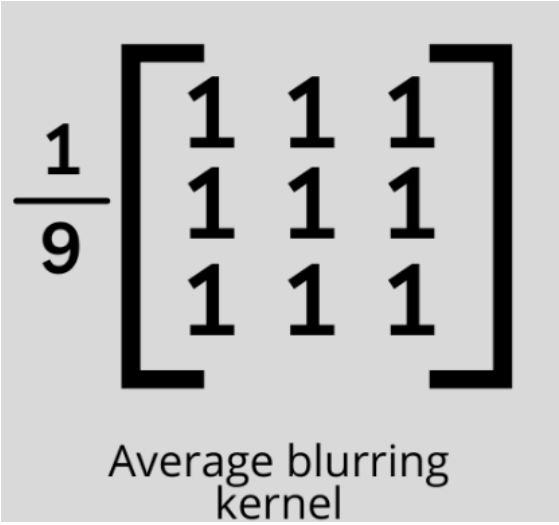
```
# Importing the modules
import cv2
import numpy as np

# Reading the image
image = cv2.imread('image.png')

# Creating the kernel with numpy
kernel2 = np.ones((5, 5), np.float32)/25

# Applying the filter
img = cv2.filter2D(src=image, ddepth=-1, kernel=kernel2)

# showing the image
cv2.imshow('Original', image)
cv2.imshow('Kernel Blur', img)

cv2.waitKey()
cv2.destroyAllWindows()
```

Try : Apply Low Pass Spatial Domain Filtering to observe the blurring effect

10.2 Averaging an Image

The averaging method is very similar to the 2d convolution method as it is following the same rules to smoothen or blur an image and uses the same type of kernel which will basically set the center pixel's value to the average of the kernel weighted surrounding pixels. And by this, we can greatly reduce the noise of the image by reducing the clarity of an image by replacing the group of pixels with similar values which is basically similar color. We can greatly reduce the noise of the image and smoothen the image. The kernel we are using for this method is the desired shape of a matrix with all the values as "1" and the whole matrix is divided by the number of values in the respective shape of the matrix

[which is basically averaging the kernel weighted values in the pixel range]. The kernel we used in this example is,



Input:

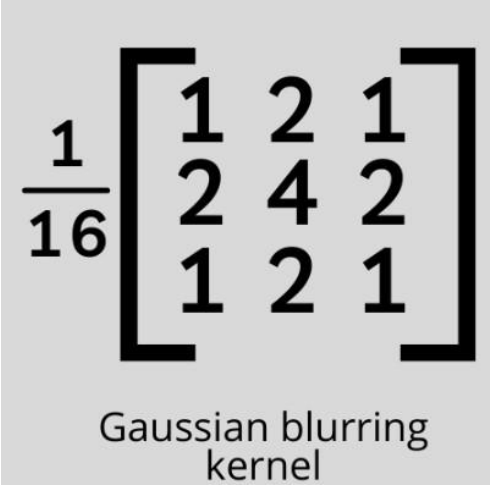
Output:

```
# Importing the modules
```

```
import cv2
import numpy as np

# Reading the image
image = cv2.imread('image.png')

# Applying the filter
averageBlur = cv2.blur(image, (5, 5))

# Showing the image
cv2.imshow('Original', image)
cv2.imshow('Average blur', averageBlur)

cv2.waitKey()
cv2.destroyAllWindows()
```

10.3 Gaussian Blur

In a gaussian blur, instead of using a box filter consisting of similar values inside the kernel which is a simple mean we are going to use a weighted mean. In this type of kernel, the values near the center pixel will have a higher weight. With this type of blurs, we will probably get a less blurred image but a natural blurred image which will look more natural because it handles the edge values very well. Instead of averaging the weighted sum of the pixels here, we will divide it with a specific value which is 16 in the case of a 3 by 3 shaped kernel which will look like this.



Input:

Output:

```
# Importing the module
import cv2
import numpy as np

# Reading the image
image = cv2.imread('image.png')

# Applying the filter
gaussian = cv2.GaussianBlur(image, (3, 3), 0)

# Showing the image
cv2.imshow('Original', image)
cv2.imshow('Gaussian blur', gaussian)

cv2.waitKey()
cv2.destroyAllWindows()
```

10.4Median blur

In this method of smoothing, we will simply take the median of all the pixels inside the kernel window and replace the center value with this value. The one positive of this method over the gaussian and box blur is in these two cases the replaced center value may contain a pixel value that is not even present in the image which will make the image's color different and weird to look, but in case of a median blur though it takes the median of the values that are already present in the image it will look a lot more natural.

Input:

Output:

```
# Importing the modules
import cv2
import numpy as np

# Reading the image
image = cv2.imread('image.png')

# Applying the filter
medianBlur = cv2.medianBlur(image, 9)

# Showing the image
```

```
cv2.imshow('Original', image)
cv2.imshow('Median blur', medianBlur)

cv2.waitKey()
cv2.destroyAllWindows()
```

10.5Bilateral blur

The smoothening methods we saw earlier are fast but we might end up losing the edges of the image which is not so good. But by using this method, this function concerns more about the edges and smoothens the image by preserving the images. This is achieved by performing two gaussian distributions. This might be very slow while comparing to the other methods we discussed so far.

Input:

Output:

```
# Importing the modules
import cv2
import numpy as np

# Reading the image
image = cv2.imread('image.png')
```

```
# Applying the filter
bilateral = cv2.bilateralFilter(image, 9, 75, 75)

# Showing the image
cv2.imshow('Original', image)
cv2.imshow('Bilateral blur', bilateral)

cv2.waitKey()
cv2.destroyAllWindows()
```

Try: Produce a code to smoothen the image using Alpha-trimmed mean filter and Kuwahara filter

11. Exercises on Compression of an image

Image compression is the process of reducing the size of an image file while maintaining an acceptable level of quality or detail. This is achieved by eliminating redundancies in the image data, either through lossless or lossy compression techniques.

Lossless Compression: Reduces file size without losing any information. Examples include PNG and GIF formats.
Lossy Compression: Removes some data, leading to a loss in quality that may be imperceptible to the human eye. Examples include JPEG and WebP formats.

11.1 Coding Interpixel and Psychovisual Redundancy,

Interpixel redundancy arises because neighboring pixels in an image often have similar intensity values due to the smooth nature of natural scenes. Predictive coding is commonly used to reduce this redundancy. Instead of storing the actual pixel values, the differences (residuals) between adjacent pixels are stored. The residual values typically have a smaller dynamic range and are easier to compress. Exploit spatial correlation between pixels to represent the image more efficiently.

Psychovisual redundancy leverages the fact that the human visual system is not equally sensitive to all image details. Quantization reduces the precision of image values, prioritizing perceptually important details while discarding less noticeable variations. The number of quantization levels determines the trade-off between compression and visual quality. Reduce or eliminate information that the human eye is unlikely to perceive, thereby achieving higher compression ratios.

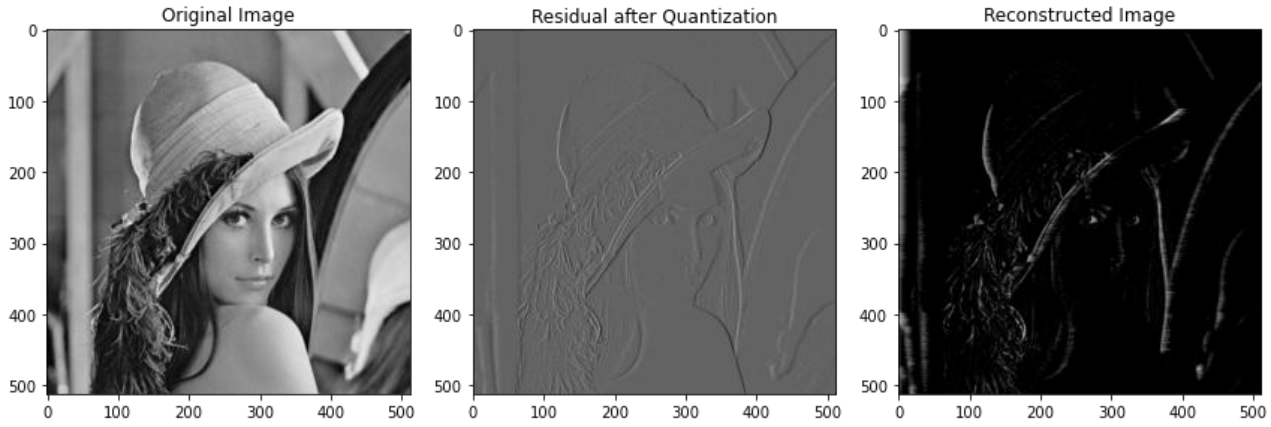
By combining these two concepts, image compression techniques can achieve significant reductions in storage size while maintaining acceptable visual quality.

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

def predictive_coding(image):
    """
    Applies predictive coding to reduce interpixel redundancy.
    Uses the pixel to the left as the predictor.
    """
    predictor = np.zeros_like(image, dtype=np.int16)
    predictor[:, 1:] = image[:, :-1]  # Predict next pixel based on the previous one
    residual = image.astype(np.int16) - predictor  # Residual (difference)
    return residual, predictor

def reconstruct_from_residual(residual, predictor):
    """
    Reconstruct the image from residual and predictor.
    """
    reconstructed = np.zeros_like(residual, dtype=np.uint8)
    reconstructed[:, 0] = residual[:, 0]  # First column remains the same
    for i in range(1, residual.shape[1]):
        reconstructed[:, i] = np.clip(residual[:, i] + reconstructed[:, i - 1], 0, 255)
    return reconstructed
```

```
def quantize_image(image, levels):
    """
    Quantizes the image to reduce psychovisual redundancy.
    """
    max_val = 255
    step = max_val // (levels - 1)
    quantized = (image // step) * step
    return quantized

def compression_pipeline(image_path, quantization_levels):
    # Load the image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        raise ValueError("Image not found or unable to load.")

    # Predictive Coding
    residual, predictor = predictive_coding(image)

    # Quantization
    quantized_residual = quantize_image(residual + 128, quantization_levels) - 128  # Shift for signed values

    # Reconstruction
    reconstructed_image = reconstruct_from_residual(quantized_residual, predictor)

    # Display the results
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 3, 1)
    plt.title("Original Image")
    plt.imshow(image, cmap='gray')
    plt.subplot(1, 3, 2)
    plt.title("Residual after Quantization")
    plt.imshow(quantized_residual, cmap='gray')
    plt.subplot(1, 3, 3)
    plt.title("Reconstructed Image")
    plt.imshow(reconstructed_image, cmap='gray')
    plt.tight_layout()
    plt.show()

    # Compression analysis
    original_size = image.size * 8  # bits
    compressed_size = residual.size * np.log2(quantization_levels)
    print(f"Original Size: {original_size} bits")
    print(f"Compressed Size: {compressed_size} bits")
    print(f"Compression Ratio: {original_size / compressed_size:.2f}")

    return reconstructed_image
```

```
# Run the pipeline
image_path = "example.jpg" # Replace with the path to your grayscale image
quantization_levels = 16 # Set the quantization levels
compressed_image = compression_pipeline(image_path, quantization_levels)
```

Results:



11.2 LZW Coding

LZW (Lempel-Ziv-Welch) coding is a lossless data compression algorithm. It works by building a dictionary of substrings encountered in the input data. Each substring is assigned a unique code. As the algorithm processes the input, it replaces substrings with their corresponding codes, thereby reducing redundancy.

Steps:

- **Dictionary Initialization**: Start with a predefined dictionary containing all possible single characters in the input.
- **String Matching**: Traverse the input string, combining characters into longer strings as they are found in the dictionary.
- **Dictionary Expansion**: Add new substrings to the dictionary whenever a new combination is encountered.
- **Output Codes**: Replace substrings with their dictionary codes to achieve compression.

```
import cv2
import numpy as np

def lzw_compress(data):
    """
    Compress data using LZW algorithm.
    """
    # Initialize the dictionary with all single-character entries
    dictionary = {chr(i): i for i in range(256)}
    next_code = 256
    string = ""
    compressed_data = []

    for symbol in data:
        string_plus_symbol = string + symbol
        if string_plus_symbol in dictionary:
            string = string_plus_symbol
        else:
            compressed_data.append(dictionary[string])
            if next_code < 4096:  # Limit dictionary size for simplicity
                dictionary[string_plus_symbol] = next_code
                next_code += 1
            string = symbol

    if string:
        compressed_data.append(dictionary[string])

    return compressed_data
```

Try: Simulate a code to compress and restore an Image using

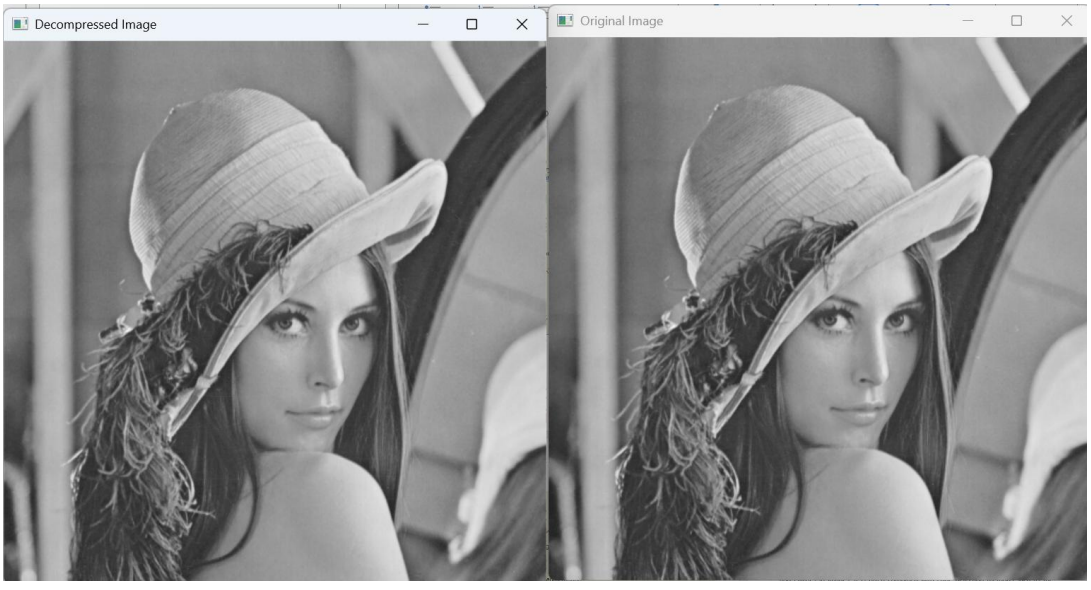
- JPEG 2000 coding
- Transform coding

```
def lzw_decompress(compressed_data):
    """
    Decompress LZW-compressed data.
    """
    # Initialize the dictionary with all single-character entries
    dictionary = {i: chr(i) for i in range(256)}
    next_code = 256
    string = chr(compressed_data[0])
    decompressed_data = string

    for code in compressed_data[1:]:
        if code in dictionary:
            entry = dictionary[code]
        elif code == next_code:
            entry = string + string[0]
        else:
            raise ValueError("Invalid LZW code encountered.")

        decompressed_data += entry

        if next_code < 4096:  # Limit dictionary size for simplicity
            dictionary[next_code] = string + entry[0]
            next_code += 1

        string = entry

    return decompressed_data

def image_to_string(image):
    """
    Convert an image to a string for LZW compression.
    """
    return ''.join(chr(pixel) for pixel in image.flatten())

def string_to_image(data, shape):
    """
    Convert a string back to an image array.
    """
    flat_image = np.array([ord(char) for char in data], dtype=np.uint8)
    return flat_image.reshape(shape)
```

```
def lzw_image_compression(image_path):
    # Load the grayscale image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        raise ValueError("Image not found or unable to load.")

    original_shape = image.shape
    image_string = image_to_string(image)

    # Compress the image using LZW
    compressed_data = lzw_compress(image_string)
    print(f"Compressed Data Length: {len(compressed_data)} codes")

    # Decompress the image
    decompressed_string = lzw_decompress(compressed_data)
    decompressed_image = string_to_image(decompressed_string, original_shape)

    # Display results
    cv2.imshow("Original Image", image)
    cv2.imshow("Decompressed Image", decompressed_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    return compressed_data, decompressed_image

# Run the compression pipeline
image_path = "example.jpg"  # Replace with the path to your grayscale image
compressed_data, decompressed_image = lzw_image_compression(image_path)
```

Results:

Compressed Data Length: 181885 codes



12.1 Edge Detection with Laplacian Mask for Image Segmentation

Edge detection is a fundamental technique in image segmentation, identifying boundaries within images. The Laplacian mask is a second-order derivative operator that highlights regions of rapid intensity change, which are often edges. It is highly sensitive to noise and usually applied on grayscale images.

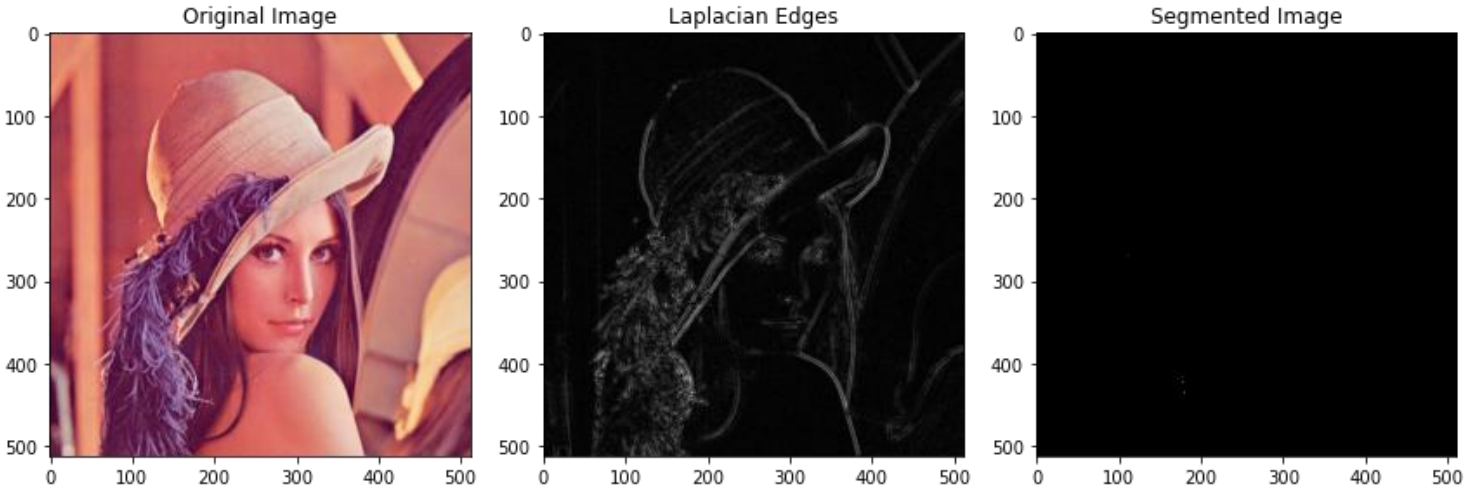
Steps:

- **Convert to Grayscale**: If the input image is colored, it is converted to grayscale to simplify edge detection.
- **Apply Gaussian Blur**: Smooth the image to reduce noise.
- **Laplacian Operator**: Compute the Laplacian of the image, which accentuates edges.
- **Thresholding**: Use thresholding to segment the image into regions based on detected edges.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def laplacian_edge_detection(image_path, threshold=50):
    """
    Perform image segmentation using edge detection with Laplacian mask.

    Parameters:
    - image_path: Path to the input image.
    - threshold: Threshold value for edge segmentation.

    Returns:
    - segmented_image: Binary image showing detected edges.
    """
    # Load the image
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError("Image not found or unable to load.")

    # Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian Blur to reduce noise
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)

    # Apply Laplacian operator
    laplacian = cv2.Laplacian(blurred, cv2.CV_64F)
    abs_laplacian = cv2.convertScaleAbs(laplacian)  # Convert to uint8

    # Thresholding to segment the image
    _, segmented_image = cv2.threshold(abs_laplacian, threshold, 255, cv2.THRESH_BINARY)
```

```
i
    # Display results
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 3, 1)
    plt.title("Original Image")
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    plt.subplot(1, 3, 2)
    plt.title("Laplacian Edges")
    plt.imshow(abs_laplacian, cmap='gray')
    plt.subplot(1, 3, 3)
    plt.title("Segmented Image")
    plt.imshow(segmented_image, cmap='gray')
    plt.tight_layout()
    plt.show()

    return segmented_image

# Run the edge detection
image_path = "example.jpg"  # Replace with your image path
segmented_image = laplacian_edge_detection(image_path, threshold=50)
```

Results:



12.2 Image Segmentation Using Hough Transform for Edge Detection

The **Hough Transform** is a technique for detecting geometric shapes like lines, circles, and ellipses in images. It is often combined with edge detection methods such as the Canny Edge Detector to find edges first and then detect shapes based on these edges.

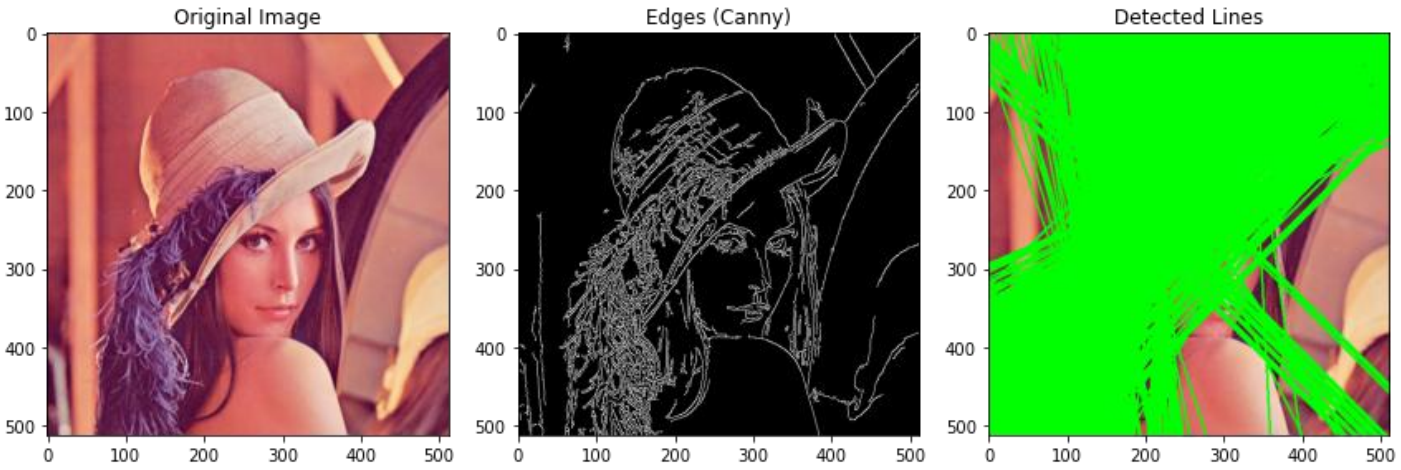
Steps:

- **Convert Image to Grayscale**: Simplifies the input for edge detection.
- **Apply Edge Detection**: Use the Canny Edge Detector to find edges in the image.
- **Apply Hough Transform**:
    - For lines: Uses the parametric form $\rho = x\cos(\theta) + y\sin(\theta)$.
    - For circles or other shapes: Specific versions of the Hough Transform are used.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def hough_transform_edge_detection(image_path, canny_threshold1=50, canny_threshold2=150,
hough_threshold=100):
    """
    Perform image segmentation using edge detection and Hough Transform.

    Parameters:
    - image_path: Path to the input image.
    - canny_threshold1, canny_threshold2: Thresholds for the Canny Edge Detector.
    - hough_threshold: Threshold for the Hough Transform.

    Returns:
    - result_image: Image with detected lines superimposed.
    """
    # Load the image
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError("Image not found or unable to load.")

    # Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

```
# Apply Canny Edge Detection
    edges = cv2.Canny(gray, canny_threshold1, canny_threshold2)

    # Apply Hough Line Transform
    lines = cv2.HoughLines(edges, 1, np.pi / 180, hough_threshold)
    result_image = image.copy()

    if lines is not None:
        for line in lines:
            rho, theta = line[0]
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a * rho
            y0 = b * rho
            x1 = int(x0 + 1000 * (-b))
            y1 = int(y0 + 1000 * (a))
            x2 = int(x0 - 1000 * (-b))
            y2 = int(y0 - 1000 * (a))
            cv2.line(result_image, (x1, y1), (x2, y2), (0, 255, 0), 2)

    # Display results
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 3, 1)
    plt.title("Original Image")
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    plt.subplot(1, 3, 2)
    plt.title("Edges (Canny)")
    plt.imshow(edges, cmap='gray')
    plt.subplot(1, 3, 3)
    plt.title("Detected Lines")
    plt.imshow(cv2.cvtColor(result_image, cv2.COLOR_BGR2RGB))
    plt.tight_layout()
    plt.show()

    return result_image

# Run the Hough Transform-based edge detection
image_path = "example.jpg"  # Replace with your image path
result_image = hough_transform_edge_detection(image_path, canny_threshold1=50, canny_threshold2
hough_threshold=100)
```

Results:



Try: Develop a code to segment the image using edge detection using following masking techniques

1. Robert
2. Sobel
3. Previtt

13. Morphological Processing for Image Segmentation

Morphological processing is a technique in image processing that applies structuring elements to images, focusing on their shape. It is primarily used in binary images but can also be extended to grayscale images.

Erosion:

- Removes pixels from object boundaries.
- Useful for eliminating small noise and separating connected objects.
- Achieved by shrinking objects in the image based on the structuring element.

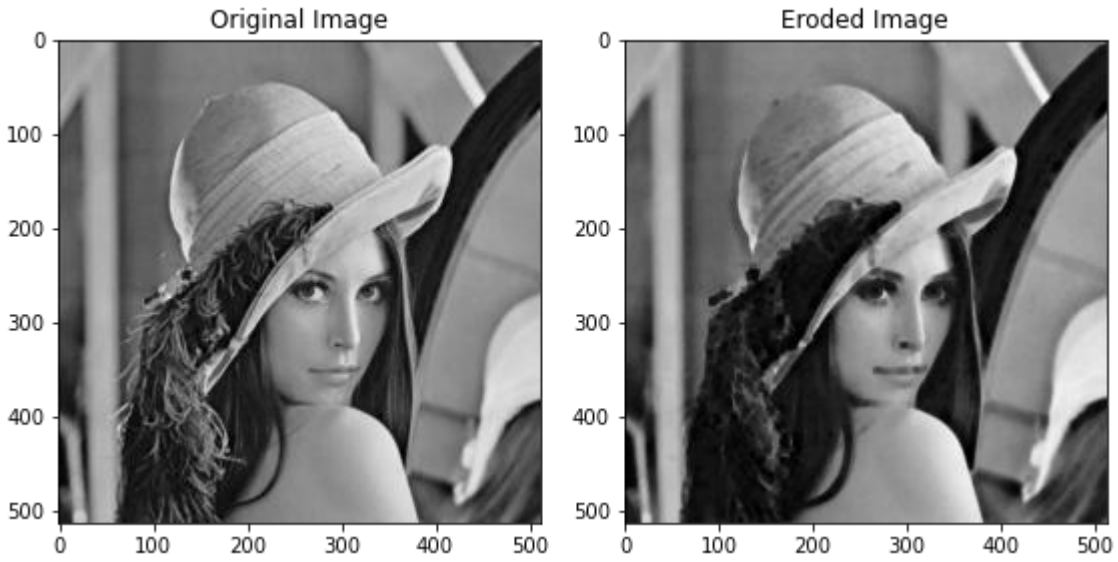
Dilation:

- Adds pixels to object boundaries.
- Useful for filling small holes and connecting disjoint objects.
- Achieved by expanding objects in the image based on the structuring element.

13.1 Morphological Processing for Image Segmentation - Erosion

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morphological_erosion(image_path, kernel_size=(3, 3), iterations=1):
    """
    Perform morphological erosion on the image.

    Parameters:
    - image_path: Path to the input image.
    - kernel_size: Size of the structuring element.
    - iterations: Number of times erosion is applied.

    Returns:
    - eroded_image: Eroded version of the input image.
    """

Load the binary or grayscale image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        raise ValueError("Image not found or unable to load.")

    # Define the structuring element (kernel)
    kernel = np.ones(kernel_size, np.uint8)

    # Apply erosion
    eroded_image = cv2.erode(image, kernel, iterations=iterations)

    # Display results
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.title("Original Image")
    plt.imshow(image, cmap='gray')
    plt.subplot(1, 2, 2)
    plt.title("Eroded Image")
    plt.imshow(eroded_image, cmap='gray')
    plt.tight_layout()
    plt.show()

    return eroded_image

# Run the erosion function
image_path = "example.jpg"  # Replace with your image path
eroded_image = morphological_erosion(image_path, kernel_size=(5, 5), iterations=1)
```

Results:

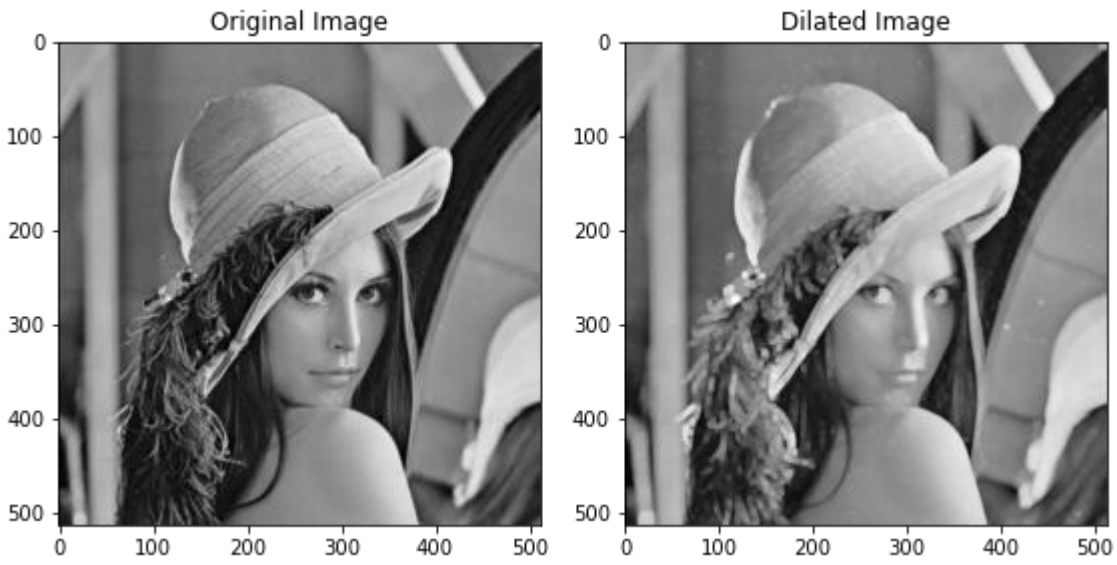


13.2 Morphological Processing for Image Segmentation -Dilation

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def morphological_dilation(image_path, kernel_size=(3, 3), iterations=1):
    """
    Perform morphological dilation on the image.

    Parameters:
    - image_path: Path to the input image.
    - kernel_size: Size of the structuring element.
    - iterations: Number of times dilation is applied.

    Returns:
    - dilated_image: Dilated version of the input image.
    """
    # Load the binary or grayscale image
    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        raise ValueError("Image not found or unable to load.")

    # Define the structuring element (kernel)
    kernel = np.ones(kernel_size, np.uint8)

    # Apply dilation
    dilated_image = cv2.dilate(image, kernel, iterations=iterations)
```

```
    # Display results
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.title("Original Image")
    plt.imshow(image, cmap='gray')
    plt.subplot(1, 2, 2)
    plt.title("Dilated Image")
    plt.imshow(dilated_image, cmap='gray')
    plt.tight_layout()
    plt.show()

    return dilated_image
# Run the edge detection
image_path = "example.jpg"  # Replace with your image path
segmented_image = laplacian_edge_detection(image_path, threshold=50)
```

Results:



Try: Implement the codes for object detection and identification

V. TEXT BOOKS:

1. Discrete-Time Signal Processing, I second edition, A. V. Oppenheim, R. W. Schafer, J. R. Buck, Prentice Hall, 1999.
2. Real-Time Digital Signal Processing, from MATLAB to C With TMS320C6x DSK,î T. B. Welch, C. H. G. Wright, and M. G. Morrow, Taylor & Francis, 2006.
3. Real-time digital signal processing: Implementations and applications, S. M. Kuo, B. H. Lee, and W. Tian, second edition, John Wiley & Sons, 2006.
4. Anil K. Jain, ″Fundamentals of Digital Image Processing″, Pearson, 2002.

VI. REFERENCE BOOKS:

1. Digital Signal Processing Implementation using the TMS320C6000TM Platform, Naim Dahnoun, Prentice Hall, 2000.
2. Digital Signal Processing Using MATLAB ¯, Sanjit Mitra, McGraw-Hill, 1999.
3. CL Wadhwa, *Electrical Circuit Analysis including Passive Network Synthesis*, International, 2nd Edition, 2009.
4. Proakis and Manolakis, "Digital Signal Processing", 3rd edition, Prentice Hall, 1996.
5. Little and Shure, "Signal Processing TOOLBOX User's Guide for use with MATLAB", MathWorks, Inc., 1992.

VII. ELECTRONICS RESOURCES:

1. https://ccrma.stanford.edu/~jos/sasp/Example_Overlap_Add_Convolution.html
2. https://rt-dsp.com/3rd_ed/app_a/App_CCS_5_1_dsk6713.pdf
3. Mathworks. The Mathworks Web Site: http://www.mathworks.com/. Technical report, Mathworks, 2001
4. W. Eaton. Octave web site http://www.che.wisc.edu/octave. Technical report, Univ. Wisconsin, 2001.
5. G. Campbell. DataLab Users' Manual. Technical report, University of Ulster, Interactive Systems Centre, Report isc/94/015/r, available from http://www.cs.qub.ac.uk/~J.Campbell/dl/dlprg.a, 1994.
6. https://sisu.ut.ee/imageprocessing/documents
7. https://www.geeksforgeeks.org/reading-image-opencv-using-python/

VIII. MATERIALS ONLINE

1. Course template
2. Lab manual