Aim: Linear Convolution

Steps:

1. **Accept the input sequence** using the input method and store it in variable x.
2. **Accept another input sequence** to be convolved and store it in variable h.
3. **Find the lengths** of x and h, store them in n1 and n2 respectively.
4. **Find the maximum length** of the input sequences and store it in n.
5. **Calculate the output sequence length** as n = n1 + n2 - 1.
6. **Zero-padding**: Extend x and h with zeros from 1 to n-1.
7. **Obtain the time reversal** of h(k) → h(-k).
8. **Shift the sequence** to obtain h(n-k).
9. **Perform multiplication** of h(n-k) with x(k) and sum over k.
10. **Store the final result** as the convolution output.

Scilab Code for Linear Convolution

```
// Read input sequences
x = input("Enter first sequence x: ");
h = input("Enter second sequence h: ");

// Get lengths
n1 = length(x);
n2 = length(h);

// Output sequence length
n = n1 + n2 - 1;

// Zero-padding
x = [x, zeros(1, n - n1)];
h = [h, zeros(1, n - n2)];

// Convolution operation
```

```
y = zeros(1, n);
for i = 1:n
   for j = 1:i
      y(i) = y(i) + x(j) * h(i-j+1);
   end
end

// Display result
disp("Linear Convolution Output: ");
disp(y);

// Plotting the input and output sequences
subplot(3,1,1);
plot2d3(nx, x);
xtitle("Input Sequence x[n]");

subplot(3,1,2);
plot2d3(nh, h);
xtitle("Impulse Response h[n]");

subplot(3,1,3);
plot2d3(ny, y);
xtitle("Linear Convolution Output y[n]");
```

Example Execution

Input:

x = [1, 2, 3, 4];
h = [1, 2, 1, 2];

Output:

y = [1 4 8 13 12 10 8]

Circular Convolution (Image Processing Context)

Step 1: Input the Sequences

- Accept the first input sequence (image or signal) and store it in variable **x**.

- Accept the second input sequence (filter or kernel) and store it in variable **h**.

Step 2: Compute the Lengths

- Find the length of **x** (let it be **N1**) and **h** (let it be **N2**).
- Compute **N = max(N1, N2)** (for circular convolution, both sequences should have the same length).

Step 3: Zero-Pad the Sequences

- If **x** and **h** are not of length **N**, pad them with zeros to make their lengths equal to **N**.

Step 4: Compute Circular Convolution

- Initialize an output sequence **y** of length **N** with zeros.
- For each index **n = 0 to N-1**, calculate:
  y(n)=∑k=0N−1x(k)⋅h((n−k)mod N)y(n) = \sum_{k=0}^{N-1} x(k) \cdot h((n-k) \mod N)
    - This performs circular convolution using modular arithmetic.

Step 5: Display and Plot the Results

- Display the circular convolution result **y**.
- Plot the original sequences **x** and **h**, along with the output **y**, using **subplot()** in Scilab.

```
g = input("Enter first sequence: ");
o = input("Enter second sequence: ");

n1 = length(g);
n2 = length(o);
n = max(n1, n2);
n3 = n1 - n2;


if (n3 > 0)
  o = [o, zeros(1, n3)];
else
```

```
    g = [g, zeros(1, -n3)];
end

y = zeros(1, n);

for p = 1:n
    y(p) = 0;
    for q = 1:n
        k = p - q + 1;
        if k < 1
            k = k + n;
        end
        y(p) = y(p) + g(q) * o(k);
    end
end

disp("Circular Convolution result: ");
disp(y);

plot(y);
```

Practical 3

Algorithm for Image Quantization in Image Processing

1. Read the input image using `imread()` and convert it to double precision using `double()`.

2. Find the maximum pixel intensity using `max()`.

3. Prompt the user to enter the number of quantization bits using `input()`.

4. Compute the quantization step size as:
   c=max intensity2ac = \frac{\text{max intensity}}{2^a}
5. Apply quantization by dividing pixel values by `c`, using `floor()`.

6. Normalize the quantized image to an 8-bit range by scaling it with `255/max(f)`.

7. Display the original and quantized images using `imshow()`.

```
// Read the input image
img = imread("image1.jpg");
// Convert image to double for calculations
img_double = double(img);
// Find the maximum pixel intensity value
b = max(img_double);
// Get the number of quantization bits from the user
a = input("Enter the number of quantization bits: "); //eg.1
or 2 or 4 or 8
// Define quantization levels
levels = 2^a; // Total quantization levels
step = b / (levels - 1); // Step size for quantization
// Perform quantization
f = floor(img_double / step) * step;
// Normalize and scale the quantized image
f1 = (f / max(f)) * 255;
// Display original and quantized images
figure;
imshow(uint8(img));
title("Original Image");
figure;
imshow(uint8(f1));
title("Quantized Image");
```

Practical: 4

Aim: Perform the FFT and the inverse FFT for the given sequence.

1. **Accept the two input sequences from the user and store them in variables x and h.**

2. **Find the length of the two sequences by using the `length()` method and store it in the variables m and n, respectively.**

3. **Find the value for the variable N by using the relation:**
   N=m+n−1N = m + n - 1
4. **Append zeros in the variable x and h from m to N-m and n to N-n, respectively.**

5. **Find the FFT of the input sequence x and store it in the variable F1. Also, find the FFT of the sequence h and store it in the variable F2.**

6. **Perform element-wise multiplication of F1 and F2 in the frequency domain and store the result in F3.**

7. **Compute the inverse FFT of F3 and store the result in F4.**

8. **Display the values of F4 using the subplot method, followed by `plot2d()` or `plot()` methods for visualization. Repeat this step for the input sequence x and the impulse response sequence h.**

```
// Take input for two vectors
g = input("Enter the first vector g (as a row vector): ");
h = input("Enter the second vector h (as a row vector): ");

// Get the lengths of the vectors
m = length(g);
n = length(h);

// Compute the length for zero-padding
N = m + n - 1;

// Zero-padding to match the required length
g = [g, zeros(1, N - m)];
h = [h, zeros(1, N - n)];

// Compute FFT of both signals
f1 = fft(g);
f2 = fft(h);

// Perform element-wise multiplication
```

```
f3 = f1 .* f2;

// Compute Inverse FFT to get the convolution result
f4 = ifft(f3);

// Plot results
subplot(3,1,1);
plot2d3(1:N, h);
xtitle("Impulse Sequence");

subplot(3,1,2);
plot2d3(1:N, g);
xtitle("Input Sequence");

subplot(3,1,3);
plot2d3(1:N, f4);
xtitle("Output Sequence (Convolution Result)");
```

Practical: 5

Aim: Write the code for obtaining the negative of the image and perform the thresholding operation on the given image.

Algorithm for Image Negative and Thresholding

1. **Read the input image using `imread()` and convert it to double for precision.**
2. **Initialize the variable for the max gray value (255) and subtract the image pixel values from this value to get the negative image.**
3. **Display the original image and its negative using the `figure()` and `imshow()` functions.**
4. **For the thresholding operation, read the input image again and store it in a separate variable.**
5. **Store the image pixel values in a variable and use the `size()` method to determine the number of rows and columns.**
6. **Ask the user to input the threshold value and store it in a variable.**
7. **Use a for loop to iterate through all pixel values from 1 to max row value and 1 to max column value.**
8. **Use an if conditional statement to check if the pixel value is less than the threshold value, then set the pixel to 0; otherwise, set it to 255.**
9. **Display the original image and the thresholded image using `imshow()` and `figure()`.**

code

```
// Read the input image
img = imread("image1.jpg");
d = double(img);  // Convert to double precision
c = 255;
neg = c - d;  // Compute negative image

// Display Original and Negative Image
figure(1);
imshow(img);
title("Original Image");

figure(2);
imshow(uint8(neg));  // Convert back to uint8 for correct display
title("Negative Image");

// Read image again for thresholding
i = imread("image1.jpg");
d = double(i);  // Convert again to double

// Define a fixed threshold value
threshold = 128;  // Predefined threshold value (Adjust if needed)

// Get image size
[rows, cols] = size(d);

// Apply thresholding
op = zeros(rows, cols);  // Initialize thresholded image

for r = 1:rows
    for c = 1:cols
        if d(r, c) < threshold then
            op(r, c) = 0;  // Set pixel to black
        else
            op(r, c) = 255; // Set pixel to white
        end
    end
end

// Display Thresholded Image
figure(3);
imshow(uint8(op));
title("Thresholded Image");

// Display duplicate of original image
figure(4);
imshow(i);
title("Duplicate of Original Image");
```

Practical: 6 High-Pass and Low-Pass Filtering

Algorithm

1. Read the input image and store it in variable **a1** using `imread()`.
2. Convert the image into a double format and store it in **a** for processing.
3. Determine the size of the image using `size()` and store the values in **m** and **n**.
4. Define a **3×3 high-pass filter (HPF)** for edge detection and a **3×3 low-pass filter (LPF)** for smoothing.
5. Initialize two zero matrices, **hp** and **lp**, of the same size as the image to store the filtered results.
6. Apply filtering using a **3×3 mask** by performing convolution:
   - For each pixel (excluding border pixels), multiply the corresponding 3×3 region of the image with the **HPF** and sum the results to store in **hp**.
   - Repeat the same process using the **LPF** and store the result in **lp**.
7. Convert the filtered images **hp** and **lp** back to an 8-bit format using `uint8()`.
8. Display the **original image, high-pass filtered image, and low-pass filtered image** using `subplot()` and `imshow()`.

Code

```
a1 = imread("image3.jpg");
a = double(a1);

// Image size ko 50% reduce karna
//a = imresize(a, 0.5);

[m, n] = size(a);
disp("New Image Size: " + string(m) + " x " + string(n));

hw = [-1, -1, -1;
      -1,  8, -1;
      -1, -1, -1]; // High-pass filter

lw = [1, 1, 1;
      1, 1, 1;
      1, 1, 1] / 9; // Low-pass filter

hp = zeros(m, n);
lp = zeros(m, n);
```

```
for i = 2:m-1
    for j = 2:n-1
        // High-pass filtering (edge detection)
        hp(i, j) = (hw(1) * a(i-1, j-1) + hw(2) * a(i-1, j) + hw(3) * a(i-1, j+1) + ...
                    hw(4) * a(i, j-1)   + hw(5) * a(i, j)   + hw(6) * a(i, j+1) + ...
                    hw(7) * a(i+1, j-1) + hw(8) * a(i+1, j) + hw(9) * a(i+1, j+1));

        // Low-pass filtering (smoothing)
        lp(i, j) = (lw(1) * a(i-1, j-1) + lw(2) * a(i-1, j) + lw(3) * a(i-1, j+1) + ...
                    lw(4) * a(i, j-1)   + lw(5) * a(i, j)   + lw(6) * a(i, j+1) + ...
                    lw(7) * a(i+1, j-1) + lw(8) * a(i+1, j) + lw(9) * a(i+1, j+1));
    end
end

hp = uint8(hp);
lp = uint8(lp);

subplot(1, 3, 1);
imshow(a1);
title("Original Image");

subplot(1, 3, 2);
imshow(hp);
title("High-pass Filtered Image");

subplot(1, 3, 3);
imshow(lp);
title("Low-pass Filtered Image");
```

Here’s your algorithm rewritten in line with your original format, but adapted for Scilab:

Prac 7 Hadamard

Aim: Implement Hadamard technique on the given image and analyze changes with the input.

Steps:

1. Load the image using `imread()` and store it in variable `a`.
2. Convert the image to a double-precision array using `double()` and store it in variable `a`.

3. Extract the number of rows and columns using `size()` and store them in variables `m` and `n`, respectively.

4. Define a 3x3 median filter `w` with all elements set to 1.

5. Use nested loops to iterate through the image pixels (from the second to the second-last row and column to avoid edge issues).

6. For each pixel:

    ○ Create a 3x3 neighborhood of the pixel.

    ○ Multiply the elements of `w` with the corresponding neighborhood pixel values and store the result in variable `b`.

7. Sort `b` in ascending order using `gsort()` and store the sorted array in variable `b2`.

8. Extract the median value (5th element of `b2`) and store it in `d(i, j)`.

9. Convert the `d` matrix to `uint8` for proper image display.

10. Display the original image and the median-filtered image using `imshow()`.

```
// Read the image
img = imread("image3.jpg");
a = double(img);

// Get the size of the image
[m, n] = size(a);

// Define the 3x3 median filter
w = [1 1 1; 1 1 1; 1 1 1];

// Initialize the filtered image
d = zeros(m, n);

// Apply the filter
for i = 2:m-1
    for j = 2:n-1
        b = [
            w(1) * a(i-1, j-1), w(2) * a(i-1, j), w(3) * a(i-1, j+1),
            w(4) * a(i, j-1),   w(5) * a(i, j),   w(6) * a(i, j+1),
            w(7) * a(i+1, j-1), w(8) * a(i+1, j), w(9) * a(i+1, j+1)
        ];

        // Sort and find the median
        b2 = gsort(b, "g", "i"); // Sort in ascending order
        med = b2(5); // 5th element is the median in a sorted 3x3 matrix
        d(i, j) = med;
    end
```

```
end

// Convert the result to uint8
d = uint8(d);

// Display the images
figure(1);
imshow(img);
title("Original Image");

figure(2);
imshow(d);
title("Median Filtered Image");
```

Practical 8 Morphological operations

Algorithm for Morphological Operations

1. **Read the Input Image**:
   - Load the input image using the `imread` function.
2. **Create Structuring Elements**:
   - Use `imcreatese` to create a rectangular structuring element (e.g., size 3x3).
   - Use `imcreatese` to create a cross-shaped structuring element (e.g., size 9x9).
3. **Perform Morphological Operations**:
   - Apply **Dilation** using the rectangular structuring element with `imdilate`.
   - Apply **Erosion** using the rectangular structuring element with `imerode`.
   - Apply **Dilation** using the cross-shaped structuring element with `imdilate`.
   - Apply **Erosion** using the cross-shaped structuring element with `imerode`.
   - Perform **Opening** using the cross-shaped structuring element with `imopen`.
   - Perform **Closing** using the cross-shaped structuring element with `imclose`.
4. **Display the Results**:
   - Use `figure()` and `subplot()` to display the original image, dilated images, eroded images, and results of the opening and closing operations.
   - Add appropriate titles for each subplot to identify the operation performed.

```
// Step 1: Read the Input Image
im = imread('image3.jpg');

// Step 2: Create Structuring Elements
seRect = imcreatese('rect', 3, 3);  // Rectangular structuring element (3x3)
seCross = imcreatese('cross', 9, 9); // Cross-shaped structuring element (9x9)

// Step 3: Perform Morphological Operations (Rectangular)
dilateRect = imdilate(im, seRect);  // Dilation with rectangular element
erodeRect = imerode(im, seRect);    // Erosion with rectangular element

// Step 4: Perform Morphological Operations (Cross)
dilateCross = imdilate(im, seCross);  // Dilation with cross element
erodeCross = imerode(im, seCross);    // Erosion with cross element

// Step 5: Perform Opening and Closing Operations
openImage = imopen(im, seCross);   // Opening operation
closeImage = imclose(im, seCross); // Closing operation

// Step 6: Visualize Results
figure(1);

// Subplot 1: Dilation (Rectangular)

subplot(2, 3, 1);

imshow(dilateRect);

title("Dilation (Rectangular)");



// Subplot 2: Erosion (Rectangular)
subplot(2, 3, 2);
imshow(erodeRect);
title("Erosion (Rectangular)");

// Subplot 3: Dilation (Cross)
subplot(2, 3, 3);
imshow(dilateCross);
title("Dilation (Cross)");

// Subplot 4: Erosion (Cross)
subplot(2, 3, 4);
imshow(erodeCross);
title("Erosion (Cross)");

// Subplot 5: Original Image
subplot(2, 3, 5);
imshow(im);
title("Original Image");
```

```
// Subplot 6: Closing Operation
subplot(2, 3, 6);
imshow(closeImage);
title("Closing Operation");
```

Practical 9 Sobel Edge detection

Algorithm for Edge Detection using Sobel Operator

1. **Read the input image** and store it in variable **a**.
2. **Convert the image to grayscale manually** (if needed).
3. **Convert the image to double precision** for accurate calculations.
4. **Get the size** of the image and store it in **m** and **n**.
5. **Define the Sobel kernels** for X and Y directions:
6. **Initialize two matrices** hxh_x and hyh_y with zeros to store gradient values.
7. **Apply the Sobel operator**:
    - Loop through pixels (excluding borders).
    - Compute gradient **h_x** by convolving the image with the X kernel.
    - Compute gradient **h_y** by convolving the image with the Y kernel.
8. **Display the results**:
    - **Original Image**
    - **Gradient in X direction**
    - **Gradient in Y direction**

```
a = imread('image3.jpg');
a = rgb2gray(a)
a = double(a)

[m, n] = size(a);

x = [-1 0 1; -2 0 2; -1 0 1];
y = [-1 -2 -1; 0 0 0; 1 2 1];

h_x = zeros(m, n);
h_y = zeros(m, n);
```

```
for i = 2:m-1
   for j = 2:n-1
      h_x(i,j) = x(1,1)*a(i-1,j-1) + x(1,2)*a(i-1,j) + x(1,3)*a(i-1,j+1) + x(2,1)*a(i,j-1)   +
x(2,2)*a(i,j)   + x(2,3)*a(i,j+1) + x(3,1)*a(i+1,j-1) + x(3,2)*a(i+1,j) + x(3,3)*a(i+1,j+1);

      h_y(i,j) = y(1,1)*a(i-1,j-1) + y(1,2)*a(i-1,j) + y(1,3)*a(i-1,j+1) + y(2,1)*a(i,j-1)   +
y(2,2)*a(i,j)   + y(2,3)*a(i,j+1)   + y(3,1)*a(i+1,j-1) + y(3,2)*a(i+1,j) + y(3,3)*a(i+1,j+1);
   end
end

subplot(2,2,1);
imshow(uint8(a));
title("Original Image");

subplot(2,2,2);
imshow(uint8(h_x));
title("X-axis Gradient");

subplot(2,2,3);
imshow(uint8(h_y));
title("Y-axis Gradient");
```

Prac 10 Prewitt(edge Detection)

Algorithm for Edge Detection using Prewitt Operator

1. **Read the input image** and store it in variable **a**.
2. **Convert the image to grayscale manually** (if needed).
3. **Convert the image to double precision** for accurate calculations.
4. **Get the size** of the image and store it in **m** and **n**.
5. **Define the Prewitt kernels** for X and Y directions:
6. **Initialize two matrices** hxh_x and hyh_y with zeros to store gradient values.
7. **Apply the Prewitt operator**:
   - Loop through pixels (excluding borders).
   - Compute gradient **h_x** by convolving the image with the X kernel.
   - Compute gradient **h_y** by convolving the image with the Y kernel.
8. **Display the results**:
   - **Original Image**

- **Gradient in X direction (Prewitt Operator)**
- **Gradient in Y direction (Prewitt Operator)**

.

```
a = imread('image3.jpg');
a = rgb2gray(a);
a = double(a);
[m, n] = size(a);

x = [-1 0 1; -1 0 1; -1 0 1];
y = [-1 -1 -1; 0 0 0; 1 1 1];

h_x = zeros(m, n);
h_y = zeros(m, n);

for i = 2:m-1
   for j = 2:n-1
      h_x(i,j) = x(1,1)*a(i-1,j-1) + x(1,2)*a(i-1,j) + x(1,3)*a(i-1,j+1) + x(2,1)*a(i,j-1)  +
x(2,2)*a(i,j)  + x(2,3)*a(i,j+1)  + x(3,1)*a(i+1,j-1) + x(3,2)*a(i+1,j) + x(3,3)*a(i+1,j+1);

      h_y(i,j) = y(1,1)*a(i-1,j-1) + y(1,2)*a(i-1,j) + y(1,3)*a(i-1,j+1) + y(2,1)*a(i,j-1)  +
y(2,2)*a(i,j)  + y(2,3)*a(i,j+1)  + y(3,1)*a(i+1,j-1) + y(3,2)*a(i+1,j) + y(3,3)*a(i+1,j+1);
   end
end

subplot(2,2,1);
imshow(uint8(a));
title("Original Image");

subplot(2,2,2);
imshow(uint8(h_x));
title("X-axis Gradient (Prewitt)");

subplot(2,2,3);
imshow(uint8(h_y));
title("Y-axis Gradient (Prewitt)");
```

Prac 11 Robert(Edge Detection)

Steps for Edge Detection Using Average Gradient Method

1. Load the input image and convert it to grayscale if necessary.
2. Convert the image to double precision for accurate numerical operations.
3. Determine the size of the image (rows and columns).

4. Define the Roberts operator kernels for X-gradient and Y-gradient.
5. Initialize matrices to store the X-gradient and Y-gradient values.
6. Iterate through the image (excluding boundaries) and compute gradients using the kernels.
7. Combine the gradients to calculate the average gradient magnitude.
8. Normalize the gradient matrices for display purposes.
9. Display the original image, X-gradient, Y-gradient, and average gradient results.

```
a = imread('image3.jpg');
a = rgb2gray(a);
a = double(a);

[m, n] = size(a);

x = [1 0; 0 -1];
y = [0 1; -1 0];

h_x = zeros(m, n);
h_y = zeros(m, n);

for i = 1:m-1
   for j = 1:n-1
      h_x(i,j) = x(1,1)*a(i,j) + x(1,2)*a(i,j+1) + x(2,1)*a(i+1,j) + x(2,2)*a(i+1,j+1);

      h_y(i,j) = y(1,1)*a(i,j) + y(1,2)*a(i,j+1) + y(2,1)*a(i+1,j) + y(2,2)*a(i+1,j+1);
   end
end



subplot(2,2,1);
imshow(uint8(a));
title("Original Image");

subplot(2,2,2);
imshow(uint8(abs(h_x)));
title("X-axis Gradient (Roberts)");
subplot(2,2,3);

imshow(uint8(abs(h_y)));

title("Y-axis Gradient (Roberts)");
```