Linear Convolution

Linear convolution is a mathematical operation that takes two signals, usually in the time domain, and produces a third signal, which is a mathematical combination of the two original signals. The convolution operation is denoted by an asterisk symbol (*). In signal processing, linear convolution is used to calculate the output of a linear time-invariant system.

The linear convolution operation can be represented mathematically as:

$$y(n) = \sum_{k=0}^{N-1} \{ x(k) * h(n-k) \}$$

where x(n) and h(n) are the input signals, y(n) is the output signal, N is the length of the signals, and the sum is taken over the range of k from 0 to N-1.

Algorithm:

1. Prompt the user to enter the input signal x(n) and impulse response h(n).
2. Determine the lengths of x(n) and h(n).
3. Compute the length of the output signal y(n) as N = n1 + n2 - 1.
4. Zero-pad x(n) and h(n) so that they are both of length N.
5. Initialize the output signal y(n) to all zeros.
6. Compute the convolution sum for each value of n from 1 to N. a. For each value of n, loop over k from 1 to n and compute the sum of x(k)h(n-k+1) for k = 1 to n. b. Assign the computed sum to the corresponding element of y(n).
7. Display the resulting output signal y(n).
8. Plot the input signal x(n), impulse response h(n), and output signal y(n) on three subplots.

Circular Convolution

Circular convolution is a type of convolution that is used when the input sequences have a periodic or circular nature. In circular convolution, the end of the sequence is connected to the beginning of the sequence, resulting in a continuous circular sequence.

Circular convolution is a type of convolution that assumes the two sequences being convolved are periodic. It can be computed using the following formula:

$$y(n) = \sum_{i=0}^{N-1} (x(i) * h((n-i) \bmod N))$$

where:

y(n) is the output sequence at index n

x(i) is the input sequence x at index i

h(i) is the impulse response h at index i

N is the length of the input and impulse response sequences

Algorithm for Discrete Convolution of Two Sequences:

Inputs: Two sequences g and h

Outputs: Convolution result y

1. Clear any previously defined variables in the workspace and the command window
2. Prompt the user to input two sequences g and h
3. Determine the lengths of the sequences n1 and n2
4. Determine the length of the output sequence n = max(n1,n2)
5. Check if the lengths of the two sequences are different. If so, pad the shorter sequence with zeros to make it the same length as the longer sequence
6. Compute the convolution of the two sequences using the discrete convolution formula

- Initialize the value of y at each index p to zero
- For p = 1 to n do the following:
- For q = 1 to n do the following:
- Compute the index j for sequence h as j = p-q+1
- Handle wraparound when j becomes negative by setting j = n+j
- Compute the sum of products required by the convolution formula as y(p) = y(p) + g(q)*h(j)

7. Display the output sequence y
8. Plot the output sequence y in a subplot

Image Quantization

Image quantization is a process of reducing the number of distinct colors or gray levels in an image. It is a common technique used in image processing and compression to reduce the amount of data required to represent an image.

The process of image quantization involves dividing the range of pixel values in an image into a set of discrete levels or bins. Each pixel in the image is then assigned to the nearest bin, resulting in a reduced color or gray level resolution of the image.

Algorithm:

1. Clear the command window and workspace.
2. Read an image file from a specific file path and store it in variable "I".
3. Convert image data to double precision floating point format for numerical calculations.
4. Display the contents of "I" in the command window.
5. Find the maximum value in "I" and store it in variable "b".
6. Display the value of "b" in the command window.
7. Prompt the user to input the number of bits to use for quantization.
8. Calculate the size of the quantization intervals.
9. Quantize the image by dividing each pixel value by "c" and rounding down to the nearest integer.
10. Display the quantized image in the command window.
11. Scale the values of "f" to the range [0, 255] using the maximum value of "f".
12. Display the original image in Figure 1.
13. Display the quantized image in Figure 2.

Bit resolution

Bit resolution refers to the number of bits used to represent a signal in digital form. It determines the number of discrete levels that can be used to represent the signal, and hence, the level of detail and accuracy of the digital representation.

Bit resolution is typically expressed in terms of the number of bits used to represent the signal, such as 8-bit, 16-bit, or 24-bit resolution. The number of bits used determines the total number of levels that can be represented, which is equal to 2 to the power of the number of bits.

Algorithm for image processing code:

1. Close all existing figures to start with a clean slate
2. Read an image from the specified file path and display it
3. Get the size of the image and display it
4. Find the maximum value in the image and display it
5. Define an array of downsampling factors
6. Loop through the downsampling factors
7. Divide the image by the downsampling factor and round the result
8. Display the downsampled image
9. Repeat steps 7-8 for each downsampling factor in the array

Idft-dft

The DFT-IDFT method is used to perform convolution in the frequency domain. It involves taking the Discrete Fourier Transform (DFT) of the input sequence and impulse response, multiplying them in the frequency domain, and then taking the Inverse Discrete Fourier Transform (IDFT) of the resulting product to get the convolution result in the time domain.

The DFT of a sequence x(n) of length N can be defined as:

$X(k) = \sum(n=0 \text{ to } N-1)\ x(n) * \exp(-j2pink/N)$

where k is the frequency index and j is the imaginary unit.

The IDFT of a sequence X(k) of length N can be defined as:

$x(n) = (1/N) * \sum(k=0 \text{ to } N-1)\ X(k) * \exp(j2pink/N)$

where n is the time-domain index and j is the imaginary unit.

Algorithm for the provided code:

1. Clear command window, clear all variables, and close all figures.
2. Prompt the user to enter input and impulse sequences.
3. Get the length of the input and impulse sequences.
4. Calculate the length of the output sequence after convolution.
5. Pad the input and impulse sequences with zeros to length N.
6. Perform FFT on the padded input and impulse sequences.
7. Perform frequency domain multiplication of the FFT results.
8. Perform inverse FFT on the frequency domain multiplication result to get the time domain convolution result.
9. Display the convolution result using the DFT-IDFT method.
10. Divide the figure window into 3 rows and 1 column.
11. Plot the input sequence in the first subplot and set a title for it.
12. Plot the impulse sequence in the second subplot and set a title for it.
13. Plot the convolution result in the third subplot and set a title for it.

Image Negative

Image negative refers to an image processing operation that produces a new image where the color or grayscale values of the original image are inverted, i.e., the dark areas become bright, and the bright areas become dark. In other words, it is the complement of the original image.

The image negative can be computed using the formula:

negative = c - original

where 'c' is the maximum pixel intensity value for the image, and 'original' is the original image. For an 8-bit grayscale image, 'c' is 255.

Algorithm for the given MATLAB code:

1. Clear the command window using the 'clc' command.
2. Read in an image file called "ME.jpg" from the specified file path using the 'imread' command and store it in the variable 'original'.
3. Convert the data type of the 'original' image from uint8 to double using the 'double' command and store it in the variable 'imgdouble'.
4. Set the maximum pixel intensity value for an 8-bit grayscale image to 255 and store it in the variable 'c'.
5. Compute the negative image by subtracting each pixel value of the original image from the maximum intensity value (255 in this case) and store it in the variable 'negative'.
6. Create a new figure window with the number '1' using the 'figure' command.

7. Display the original image in the first figure window using the 'imshow' command.
8. Create a new figure window with the number '2' using the 'figure' command.
9. Display the negative image in the second figure window using the 'imshow' command.
10. End of the algorithm.

Threshold

Thresholding is a technique used in image processing and computer vision to simplify an image by reducing the amount of visual information in it. It involves converting a grayscale or color image into a binary image, where each pixel is either black or white, based on a threshold value.

The formula for thresholding can be represented mathematically as follows:

T(x,y) = { 255, if I(x,y) >= threshold_value

{ 0, if I(x,y) < threshold_value

where T(x,y) is the thresholded image, I(x,y) is the input image, and threshold_value is the chosen threshold value.

algorithm are as follows:

1. Load an image from a specified path.
2. Create a copy of the original image.
3. Get the dimensions of the image.
4. Ask the user to input a threshold value.
5. Loop through each pixel in the image.
6. Check if the pixel value is less than the threshold value.
7. If the pixel value is less than the threshold value, set the corresponding pixel in the copy to 0 (black).
8. If the pixel value is greater than or equal to the threshold value, set the corresponding pixel in the copy to 255 (white).
9. Display the original image and the thresholded image.

Grey Level Slicing with Background

Grey level slicing with background is a type of image enhancement technique that is used to highlight a specific range of grey levels in an image. This technique is particularly useful when the areas of interest in an image are within a specific range of grey levels and are surrounded by a background of different grey levels.

Algorithm for the given code is as follows:

1. Clear the command window.
2. Read the image "ME.jpg" and store it in the variable "original".
3. Convert the image to a double-precision array and store it in the variable "doub".
4. Get the dimensions of the array and store the number of rows in the variable "row" and the number of columns in the variable "column".
5. For each row from 1 to the total number of rows, do the following: a. For each column from 1 to the total number of columns, do the following: i. Check if the pixel value at (i, j) is between 50 and 150. ii. If the pixel value is between 50 and 150, set the pixel value to 0.
6. Display the original image in figure 1.
7. Display the modified image in figure 2 by converting it back to uint8 format.

Grey Level Slicing without Background

Grey level slicing without background is a technique used to enhance specific features in an image. This technique works by mapping a certain range of grey levels in the image to a different range of grey levels. Grey level slicing without background enhances the features in the image without affecting the background.

algorithm:

1. Clear the command window and any existing figures.
2. Load the image "cameraman.jpg" from the specified directory and store it in a variable named "original".
3. Convert the image to a matrix of doubles (pixel values ranging from 0 to 255) and store it in a variable named "doub".
4. Get the number of rows and columns of the image matrix.
5. Loop through each pixel in the image matrix: a. Check if the pixel value is between 50 and 150. b. If the pixel value is within the range, replace it with the corresponding pixel value from the original image.
6. Display the original image using the imshow function.
7. Display the modified image (converted back to uint8 format) using the imshow function.

Bitplane

In digital image processing, a "bit plane" refers to a binary image that represents a single bit of the original image. An image is typically represented using 8 bits (i.e., 1 byte) per pixel, with each bit representing a power of 2 from 2^0 (the least significant bit) to 2^7 (the most significant bit).

The algorithm for the given code is as follows:

1. Clear the command window using 'clc'.
2. Read the input image from the file path and store it in 'c' using the 'imread' function.
3. Convert the image information to double format and store it in 'cd'.
4. Extract the binary bit values of each pixel and store them in separate bit planes using modulo division.
5. Recombine the bit planes to form an image equivalent to the original grayscale image.
6. Display the original image in the first subplot using 'subplot' and 'imshow'.
7. Display the bit planes from LSB to MSB in subplots 2 to 9 using 'subplot' and 'imshow'.
8. Display the recombined image in the last subplot using 'subplot' and 'imshow'.
9. Convert the double format to uint8 format for display using 'uint8'.
10. Add appropriate titles to each subplot using 'title'.

Erosion and Dilation

Dilation and erosion are two basic operations in mathematical morphology used to process and analyze images.

Dilation:

Dilation of an image A by a structuring element B can be defined mathematically as follows:

$C = A \oplus B$

where $\oplus$ denotes the dilation operation, C is the output image, A is the input image, and B is the structuring element.

The dilation operation involves sliding the structuring element over each pixel of the input image and computing the maximum value of the overlapping pixels between the input image and the structuring element. The resulting value is assigned to the corresponding output pixel.

Erosion:

Erosion of an image A by a structuring element B can be defined mathematically as follows:

$C = A \ominus B$

where $\ominus$ denotes the erosion operation, C is the output image, A is the input image, and B is the structuring element.

The erosion operation involves sliding the structuring element over each pixel of the input image and computing the minimum value of the overlapping pixels between the input image and the structuring element. The resulting value is assigned to the corresponding output pixel.

Algorithm:

1. Create a 10x10 matrix 'adilate' with all zeros and set a 4x4 block in the center to 1
2. Create a 3x3 rectangular structuring element 'struele'
3. Dilate the 'adilate' matrix using 'struele' and store the result in 'afterdilate'
4. Open the 'adilate' matrix using 'struele' and store the result in 'afteropen'
5. Create a 10x10 matrix 'aerode' with all zeros and set a 4x4 block in the center to 1
6. Erode the 'aerode' matrix using 'struele' and store the result in 'aftererode'
7. Read an image "cameraman.jpg" from the desktop and store it in 'original'
8. Create a 3x3 rectangular structuring element 'se'
9. Dilate the 'original' image using 'se' and store the result in 'dilate'
10. Erode the 'original' image using 'se' and store the result in 'erode'
11. Open the 'original' image using 'se' and store the result in 'afteropen'
12. Close the 'original' image using 'se' and store the result in 'afterclose'
13. Read an image "morph2.bmp" from the desktop and store it in 'S'
14. Create an 11x11 elliptical structuring element 'se'
15. Close the 'S' image using 'se' and store the result in 'S2'
16. Read an image "morph2.bmp" from the desktop and store it in 'S'
17. Create a 9x9 elliptical structuring element 'se'
18. Open the 'S' image using 'se' and store the result in 'S2'
19. Display the resulting matrices/images using the 'disp' and 'imshow' functions.

LPF and HPF

LPF and HPF stand for Low-Pass Filter and High-Pass Filter, respectively. These are commonly used image processing techniques to filter an image based on its frequency content.

A Low-Pass Filter is a type of filter that allows low-frequency components of the image to pass through while blocking high-frequency components. It is used to remove noise from an image, smoothen it, and reduce its sharpness. LPF can be implemented using different types of filters such as mean filter, Gaussian filter, etc.

On the other hand, a High-Pass Filter allows high-frequency components of the image to pass through while blocking low-frequency components. It is used to sharpen an image and enhance its edges. HPF can be implemented using different types of filters such as Laplacian filter, Sobel filter, etc.

Here is the algorithm for the code:

1. Read the input image and convert it to double format.
2. Define a low-pass filter kernel and a high-pass filter kernel.
3. Apply the low-pass filter and high-pass filter to the input image by convolving the kernel with the image pixels. This is done using nested for-loops, iterating over each pixel of the image except for the boundary pixels.
4. Convert the filtered result to unsigned 8-bit format for display.
5. Display the low-pass filtered image, high-pass filtered image, and original image using the imshow function and specify the title for each.

Prewitt and Sobel operator

The Prewitt and Sobel operators are two commonly used edge detection filters in image processing. They both use a small matrix, called a kernel or filter, to convolve with an image in order to enhance edges or gradients in the image.

The Prewitt operator uses two 3x3 kernels, one for detecting horizontal edges and the other for detecting vertical edges. The kernels are as follows:

Horizontal edge detection kernel:

| -1 -1 -1 |

| 0 0 0 |

| 1 1 1 |

Vertical edge detection kernel:

| -1 0 1 |

| -1 0 1 |

| -1 0 1 |

To apply the Prewitt operator to an image, the two kernels are convolved with the image separately. The resulting images are then combined to form the final edge-enhanced image.

The Sobel operator is similar to the Prewitt operator in that it uses two 3x3 kernels, one for detecting horizontal edges and the other for detecting vertical edges. However, the Sobel operator places more weight on the center pixel in the kernel, resulting in a smoother gradient. The kernels are as follows:

Horizontal edge detection kernel:

| -1 -2 -1 |

| 0 0 0 |

| 1 2 1 |

Vertical edge detection kernel:

| -1 0 1 |

| -2 0 2 |

| -1 0 1 |

To apply the Sobel operator to an image, the two kernels are convolved with the image separately. The resulting images are then combined to form the final edge-enhanced image.

Algorithm:

1. Clear the command window and all variables in the workspace.
2. Read an image and store it in a variable "a".
3. Convert the image to a double-precision floating-point format.
4. Define Prewitt and Sobel filters for vertical and horizontal edges.
5. Loop through all the pixels of the image except the border pixels.
6. Apply Prewitt filter for vertical edges to the image and store the output in "a1".
7. Apply Prewitt filter for horizontal edges to the image and store the output in "a2".
8. Apply Sobel filter for vertical edges to the image and store the output in "a3".
9. Apply Sobel filter for horizontal edges to the image and store the output in "a4".
10. Display the original image, Prewitt Y Gradient, Prewitt X Gradient, and Prewitt Output in a 2x4 subplot.
11. Display the original image, Sobel Y Gradient, Sobel X Gradient, and Sobel Output in a 2x4 subplot.

Gaussian

Gaussian is a mathematical function that is commonly used in image processing as a filter. Gaussian filtering is a type of low-pass filtering that smooths images by reducing the high-frequency components of the image. The Gaussian filter is named after the mathematician Carl Friedrich Gauss who first formulated it.

The 1D Gaussian function is defined as:

G(x) = (1 / (sqrt(2 * pi) * sigma)) * exp(-x^2 / (2 * sigma^2))

where x is the distance from the center of the kernel, sigma is the standard deviation, and pi is the mathematical constant pi.

The 2D Gaussian function is defined as:

G(x,y) = (1 / (2 * pi * sigma^2)) * exp(-(x^2 + y^2) / (2 * sigma^2))

where x and y are the distances from the center of the kernel in the horizontal and vertical directions, respectively.

Algorithm for Image Filtering using Ideal Low Pass Filter:

1. Clear the command window.
2. Read the input image and convert it to grayscale.
3. Get the size of the image and store the number of rows and columns in variables.
4. Prompt the user to enter the cutoff frequency for the Ideal Low Pass Filter (LPF).
5. Loop through each pixel of the image, calculate the distance between the current pixel and the center of the image, and use it to calculate the LPF.
6. Perform a 2D Discrete Fourier Transform (DFT) on the image data.
7. Shift the zero-frequency component to the center of the spectrum.
8. Multiply the DFT of the image with the LPF using scalar multiplication, which is equivalent to convolution in the spatial domain.
9. Perform a 2D Inverse DFT on the scalar product to get the filtered image data.
10. Display the original image, LPF mesh plot, LPF 2D image, and the filtered image in four subplots.

Butterworth

Butterworth is a type of low-pass filter that attenuates the high-frequency components of an input signal while allowing the low-frequency components to pass through. It is a type of IIR (Infinite Impulse Response) filter, which means that the impulse response of the filter is infinite in duration.

The Butterworth filter can be described mathematically using the transfer function:

$$H(s) = 1 / [1 + (s / \omega c)^{2N}]^{0.5}$$

where:

s is the complex frequency variable

ωc is the cutoff frequency

N is the order of the filter

The algorithm for the code is:

1. Read the input image from the file path and convert it to grayscale.
2. Convert the pixel values of the input image to double precision.
3. Specify the cutoff frequency and order of the Butterworth filter.
4. Find the center of the image.
5. Initialize the filter kernel as a matrix of zeros with the same size as the input image.
6. Loop over the rows and columns of the filter kernel and compute the distance between the current pixel and the center of the image.
7. Compute the filter kernel value at the current pixel location using the Butterworth filter equation.
8. Compute the 2D Fourier transform of the input image and shift the zero-frequency component to the center of the spectrum.
9. Apply the filter kernel to the Fourier spectrum of the input image.
10. Compute the inverse Fourier transform of the filtered spectrum to obtain the final output image.
11. Display the original image, filter function using `imshow()`, filter function using `surf()` in 3D, and the final output image.

Color Model

A color model is a mathematical model that describes the way colors can be represented as a combination of primary colors or color components. There are several color models used in digital image processing and computer graphics, including RGB (Red, Green, Blue), CMYK (Cyan, Magenta, Yellow, Black), HSL (Hue, Saturation, Lightness), HSV (Hue, Saturation, Value), and YUV (Luminance, Chrominance).

RGB is the most commonly used color model in digital image processing, where each color is represented by its intensity levels of red, green, and blue components.

Here are some commonly used color models and their formulas:

RGB:

Color = (R, G, B)

where R, G, and B are the intensities of red, green, and blue components, respectively.

CMY:

Color = (C, M, Y)

where C, M, and Y are the intensities of cyan, magenta, and yellow components, respectively.

The conversion formula from RGB to CMY is:

C = 1 - R

M = 1 - G

Y = 1 – B

Here is a step-by-step algorithm for this code:

1. Clear the command window.
2. Read an image from a file path and store it in a variable 'a'.
3. Convert the image data to double precision floating point values.
4. Retrieve the dimensions of the image: rows, columns, and color planes (if applicable).
5. Extract the red color plane from the image.
6. Extract the green color plane from the image.
7. Extract the blue color plane from the image.
8. Create a matrix of zeros with the same size as the image.
9. Concatenate the red color plane with two zero planes to create a 24-bit red image.
10. Concatenate the green color plane with two zero planes to create a 24-bit green image.
11. Concatenate the blue color plane with two zero planes to create a 24-bit blue image.
12. Display the original image and the three color planes separately in one figure using subplots.
13. Display the original image and the 24-bit color planes separately in another figure using subplots.
14. End of the code.

Edge Detection

//Edge Detection using Ordinary operator

Edge detection is a fundamental image processing technique that aims to identify the boundaries of objects in an image. It involves identifying the pixels where there is a significant change in intensity, which often correspond to the boundaries between different objects or regions in the image.

In image processing, ordinary operators are a class of linear filters that are applied to an image matrix to extract information from it. They are also known as spatial filters or convolution filters.

An ordinary operator matrix, also known as a kernel or a filter, is a small matrix used in image processing to perform operations such as edge detection, blurring, sharpening, and more. The matrix is applied to each pixel of an image to produce a new output image with modified pixel values.

The algorithm for this MATLAB code is as follows:

1. Clear the command window using the "clc" command.
2. Read in an image file using the "imread" function and store it in the variable "a".
3. Convert the image to grayscale using the "rgb2gray" function and store it back in "a".
4. Convert the pixel values to double for precision using the "double" function.
5. Get the dimensions of the image matrix using the "size" function and store the number of rows and columns in "row" and "col" variables.
6. Define two kernel matrices "w1" and "w2" for detecting vertical and horizontal edges, respectively.
7. Loop through the rows and columns of the image matrix, ignoring the border, using two nested "for" loops.
8. Apply the vertical edge detector using the kernel matrix "w1" and store the result in "a1".
9. Apply the horizontal edge detector using the kernel matrix "w2" and store the result in "a2".
10. Add the results of the two detectors to get the resultant gradient image and store it in "a3".
11. Display the X-gradient image, Y-gradient image, and resultant gradient image using the "subplot" and "imshow" functions with appropriate titles.

//Edge Detection using Roberts operator

Roberts operator is a simple edge detection technique used in digital image processing. It is based on computing the gradient of the image intensity function using a pair of 2x2 convolution kernels. The Roberts operator consists of two kernels, each of which is used to calculate the gradient in the x- and y-directions, respectively.

The two kernels are defined as follows:

w1 = [1 0; 0 -1] (for the x-direction gradient)

w2 = [0 1; -1 0] (for the y-direction gradient)

The following is the algorithm for the given code:

1. Clear the command window using the "clc" command.
2. Read an image and convert it to grayscale using the "imread" and "rgb2gray" functions, respectively.
3. Convert the grayscale image to double precision for calculations using the "double" function.
4. Get the row and column size of the image using the "size" function.

5. Define the Roberts operators w1 and w2.
6. Loop through each pixel in the image, except for the borders.
7. Calculate the x-gradient using the first Roberts operator w1 and store it in a new matrix a1.
8. Calculate the y-gradient using the second Roberts operator w2 and store it in a new matrix a2.
9. Combine the x-gradient and y-gradient to get the resultant gradient image a3 using the "+" operator.
10. Display the original image and the three gradient images using the "subplot" and "imshow" functions.

//Edge Detection using Prewitts operator

The Prewitt operator is a simple image processing operator used for edge detection. It is a discrete differentiation operator used to calculate the gradient of an image. The operator consists of two 3x3 kernels or matrices, one for horizontal and the other for vertical directions, as shown below:

w1 = [-1 -1 -1; 0 0 0; 1 1 1]; //for horizontal gradient

w2 = [-1 0 1; -1 0 1; -1 0 1]; //for vertical gradient

Algorithm for Prewitt Edge Detection using MATLAB:

1. Clear the command window using the clc command.
2. Read the input image from the specified location using imread command and store it in variable a.
3. Convert the input image to grayscale using rgb2gray command and store it in variable a.
4. Convert the grayscale image to double precision using double command and store it in variable a.
5. Get the number of rows and columns in the input image using size command and store it in variables row and col respectively.
6. Define Prewitt operators for x and y gradients as w1 and w2 respectively.
7. Iterate over each pixel in the input image, excluding the border pixels, using nested for loops.
8. Calculate the x-gradient of the current pixel using the Prewitt operator for x-gradient and store it in variable a1.
9. Calculate the y-gradient of the current pixel using the Prewitt operator for y-gradient and store it in variable a2.

10. Add the x-gradient and y-gradient images to obtain the resultant gradient image and store it in variable a3.
11. Display the original image, x-gradient image, y-gradient image, and resultant gradient image in a 2x2 grid of subplots using subplot and imshow commands.
12. Add titles to each subplot using title command.
13. End of the algorithm.

% Edge Detection using Sobel operator

Sobel operator is a commonly used edge detection filter in image processing. It is used to compute the gradient of an image in the x and y directions. The Sobel operator consists of a small matrix of coefficients that are convolved with the image to obtain the gradient.

The Sobel operator for detecting horizontal edges is defined as:

-1 -2 -1

0 0 0

1 2 1

The Sobel operator for detecting vertical edges is defined as:

-1 0 1

-2 0 2

-1 0 1

The algorithm for this code is as follows:

1. Clear all variables and close all open figures.
2. Read an image from a file and convert it to grayscale.
3. Convert the image to double precision for mathematical operations.
4. Get the dimensions of the image (number of rows and columns).
5. Define Sobel operators for detecting horizontal and vertical edges.
6. Apply the Sobel operator to each pixel of the image by computing the gradient in the x-direction (horizontal gradient) and the gradient in the y-direction (vertical gradient).
7. Combine the horizontal and vertical gradients to obtain the overall gradient magnitude.
8. Display the original image and the gradient images (horizontal gradient, vertical gradient, and resultant gradient image) using subplots.

The code uses four different edge detectors namely Sobel, Prewitt, Log and Canny.

In image processing, the Laplacian of Gaussian (LoG) edge detector is a popular edge detection algorithm. It combines Gaussian smoothing with Laplacian filtering to detect edges in an image.

The LoG edge detector works by first convolving the input image with a Gaussian kernel to smooth out any noise in the image. This creates a blurred version of the image. Then, the Laplacian of the resulting image is computed by taking the second derivative of the image in both the x and y directions. This operation highlights areas where there is a rapid change in intensity, which corresponds to edges in the image.

Laplacian of Gaussian kernel at each point on the grid using the following formula:

LoG(x, y) = (x^2 + y^2 - 2 * sigma^2) / (sigma^4 * pi) * exp(-(x^2 + y^2) / (2 * sigma^2))

where x and y are the coordinates on the grid, sigma is the standard deviation of the Gaussian filter, and pi is the mathematical constant pi.

The Canny edge detection operator is a popular and widely used image processing technique for detecting edges in images. It was developed by John F. Canny in 1986 and is based on the principle of finding the optimal trade-off between edge detection accuracy and noise reduction.

Algorithm for edge detection using different edge detectors and displaying the original and edge-detected images in a subplot figure:

1. Import the necessary libraries: "close" and "clc".
2. Read an image from the specified path using the "imread" function and store it in the variable "a".
3. Convert the image to grayscale using the "rgb2gray" function and store it back in "a".
4. Perform edge detection using different edge detectors - Sobel, Prewitt, Log, and Canny - using the "edge" function and store the results in variables "c", "d", "e", and "f" respectively.
5. Create a subplot figure with 2 rows and 3 columns using the "subplot" function.
6. Display the original image in the first subplot using the "imshow" function and set its title using the "title" function.
7. Display the edge-detected images in the remaining subplots using the "imshow" function and set their titles using the "title" function.
8. End the program.