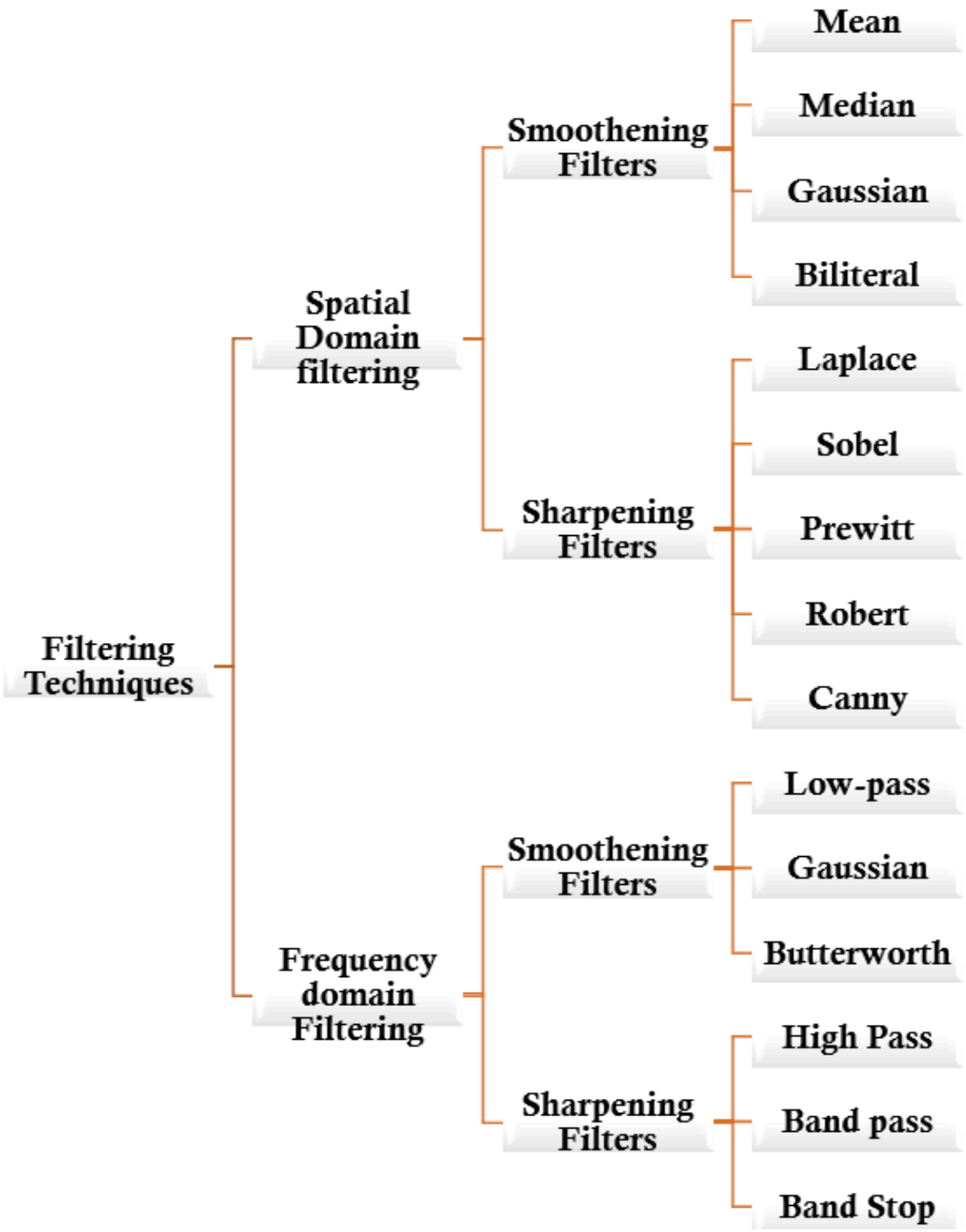
CHAPTER 4: EDGE DETECTION AND FILTERING

4.1 Spatial Domain Filtering
4.2 Frequency Domain Transformation and Filtering
4.3 Edge Detection Techniques



4.1 Spatial Domain Filtering

- Spatial domain filtering works by directly manipulating the pixel values of an image in the spatial domain to achieve various effects, such as smoothing, sharpening, or noise reduction.
- It involves the application of a filter kernel (mask) over the image through convolution or correlation operations.

Convolution is the heart of spatial filtering and enables operations like smoothing, sharpening, and edge detection.
Kernel size affects the result:
Larger kernels = stronger effects (e.g., more smoothing).
Smaller kernels = finer effects.
Adjusting kernel weights tailors the filter for specific tasks.

Spatial filtering is a technique used to enhance the image based on the spatial characteristics of the image.
It can be used for image sharpening, edge detection, blurring, image sharpening and noise reduction.

4.1.1 The Spatial Filtering Process

Step-by-Step Process

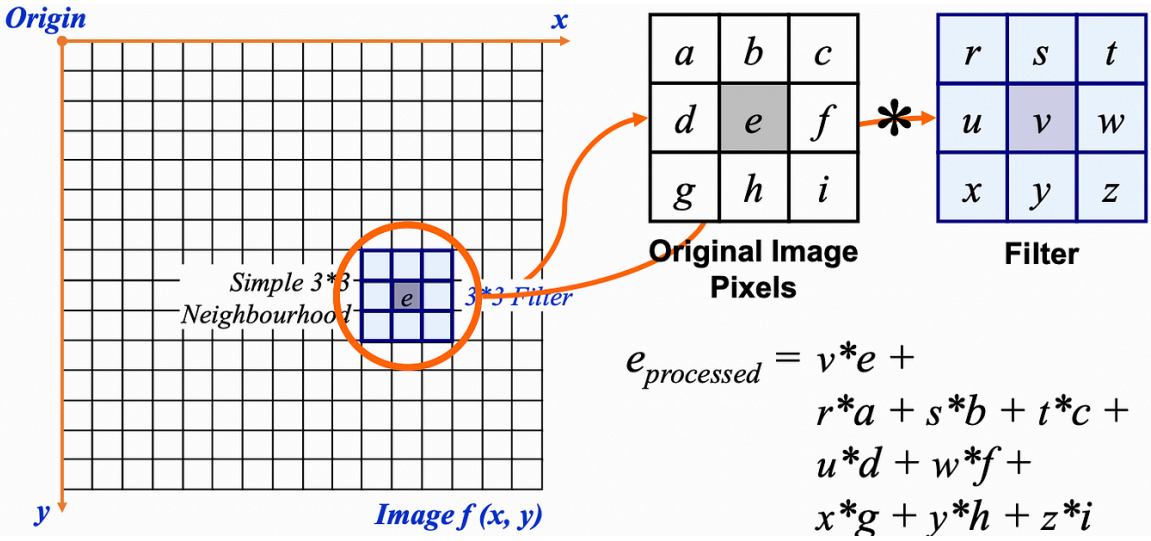
1. Define the Image and Mask

- **Image:** A 2D grid of pixel intensity values, typically represented as $f(x, y)$, where x and y are spatial coordinates.
- **Mask (Kernel):** A small matrix $h(i, j)$ of predefined weights that determines how the neighborhood of a pixel contributes to the output.
  - Example of a 3×3 smoothing kernel:

$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

2. Initialize the Output Image

- Create an output image $g(x, y)$ of the same size as the input image to store the results of filtering.



The spatial filtering process

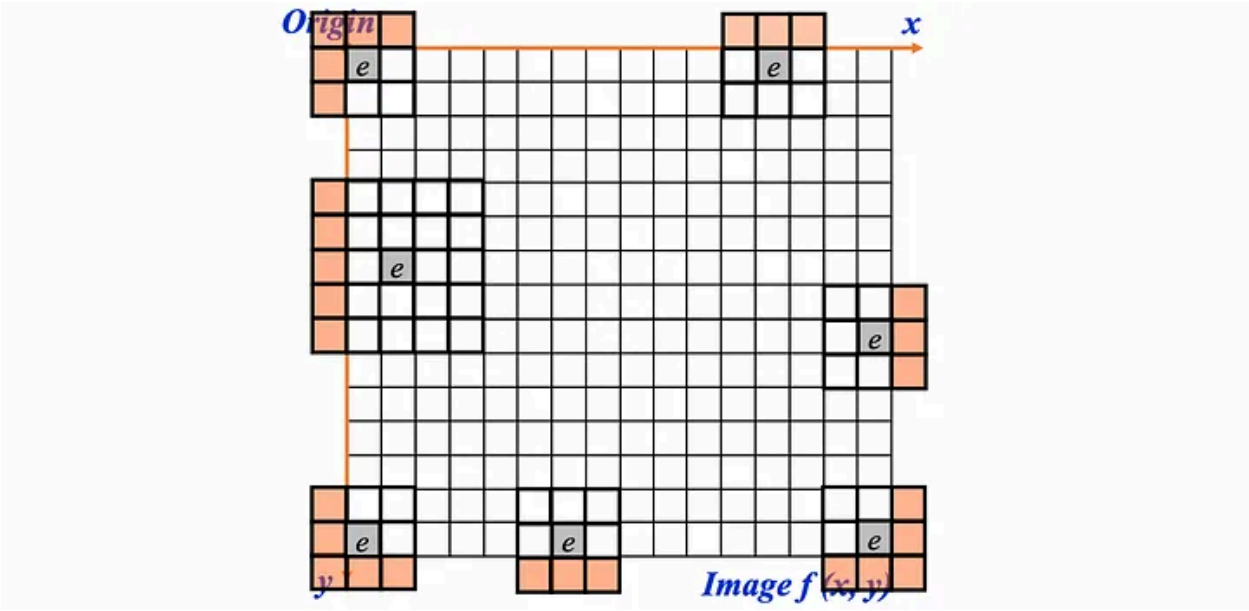
3. Convolution Operation

Convolution involves sliding the kernel over the image and performing the following operations for each pixel:

1. **Overlay the Kernel:**
   - Center the kernel at the current pixel position (x, y) in the input image.
   - Consider a neighborhood around (x, y) defined by the kernel size.
2. **Multiply:**
   - Multiply each value of the kernel $h(i, j)$ with the corresponding pixel intensity $f(x + i, y + j)$ in the neighborhood.
3. **Sum:**
   - Add up all the multiplied values:

$$g(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} h(i, j) \cdot f(x + i, y + j)$$

   Here, k is the kernel radius ($\text{kernel size} = 2k + 1$).
4. **Replace:**
   - Assign the computed sum to the center pixel in the output image at position (x, y).



4. Edge Handling (Padding)

- At the borders of the image, the kernel may partially fall outside. To address this:
  - Use **padding techniques**:
    - **Zero-padding:** Fill missing pixels outside the boundary with zeros.
    - **Replicate-padding:** Extend boundary pixel values.
  - The padding ensures the output image has the same size as the input.

5. Slide the Kernel

- Repeat the convolution operation for every pixel in the image by sliding the kernel from the top-left to the bottom-right.

Common Filters and Their Kernels

- Smoothing (Box Filter):

$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- Sharpening:

$$h = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

- Edge Detection (Sobel X):

$$h = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

For Knowledge: Correlation and Convolution

- *Correlation is the process of moving a filter mask over the image and computing the sum of products at each position. The filtering so far is referred to as correlation with the filter itself referred to as the correlation kernel.*

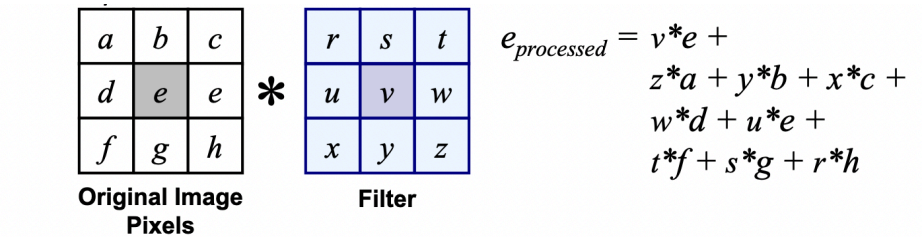
$$g(x,y) = \sum_{s=-a}^{a}\sum_{t=-b}^{b} w(s,t) f(x+s, y+t)$$

correlation formula

- *Convolution is similar operation to correlation with just one subtle difference. The filter is rotated by 180°. For symmetric filters it makes no difference.*

$$g(x,y) = \sum_{s=-a}^{a}\sum_{t=-b}^{b} w(s,t) f(x-s, y-t)$$

convolution formula



Cropped correlation result

0 8 2 3 2 1 0 0

Cropped convolution result

0 1 2 3 2 8 0 0

Correlation vs Convolution

Illustrative Example

Input Image (3x3)

$$f(x,y) = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

Mask (3x3, Averaging Filter)

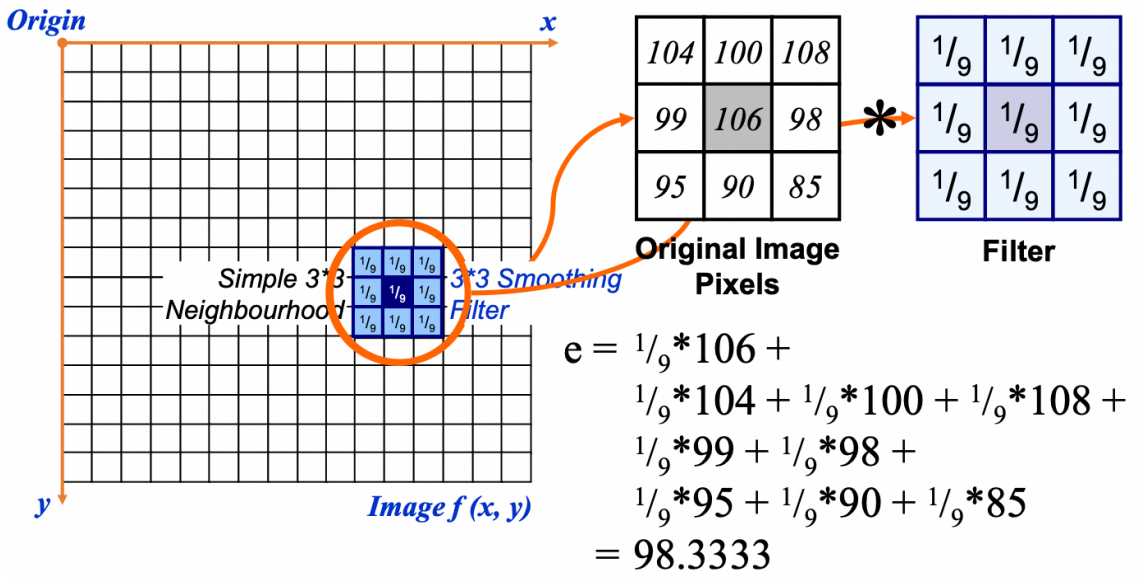
$$h(i,j) = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Filtering the Center Pixel (50)

1. Overlay the kernel at $(1,1)$, the pixel with value 50.
2. Multiply:
   - $10 \cdot \frac{1}{9} + 20 \cdot \frac{1}{9} + 30 \cdot \frac{1}{9} + 40 \cdot \frac{1}{9} + 50 \cdot \frac{1}{9} + 60 \cdot \frac{1}{9} + 70 \cdot \frac{1}{9} + 80 \cdot \frac{1}{9} + 90 \cdot \frac{1}{9}$
3. Sum:
   - $\frac{450}{9} = 50$

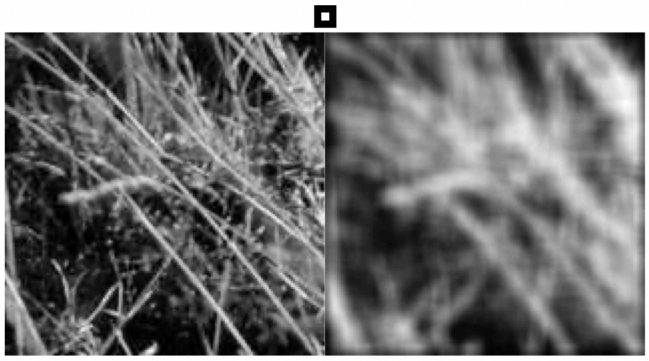
https://uomustansiriyah.edu.iq/media/lectures/6/6_2022_04_15!03_04_27_PM.pdf

4.1.1.1 Smoothing Spatial Filters



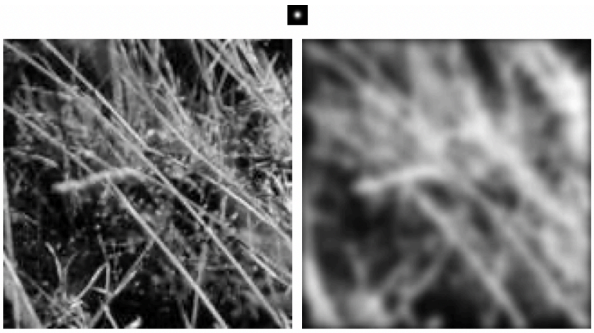
Smoothing with box filter

- Smoothing spatial filters average all of the pixels in a neighbourhood around a central value.
- It is useful in removing noise from images and highlighting gross detail.Details begin to disappear after filtering with an averaging filter of increasing sizes (3, 5, 10 .. etc.).



Smoothing with box filter

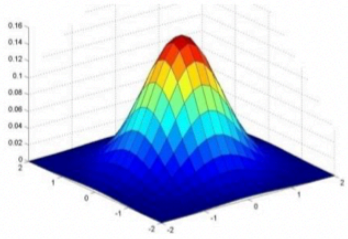
Gaussian Filters

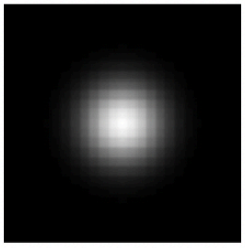


- Gaussian filters remove “high frequency components” from the image. So it is called “low-pass filter”.

- Convolution with self is another Gaussian. So can smooth with small width kernel, repeat, and get same result as larger-width kernel would have.
  - Weight contributions of neighboring pixels by nearness.

Gaussian Filter Formula





0.003	0.013	0.022	0.013	0.003
0.013	0.059	0.097	0.059	0.013
0.022	0.097	0.159	0.097	0.022
0.013	0.059	0.097	0.059	0.013
0.003	0.013	0.022	0.013	0.003

5 x 5, σ = 1

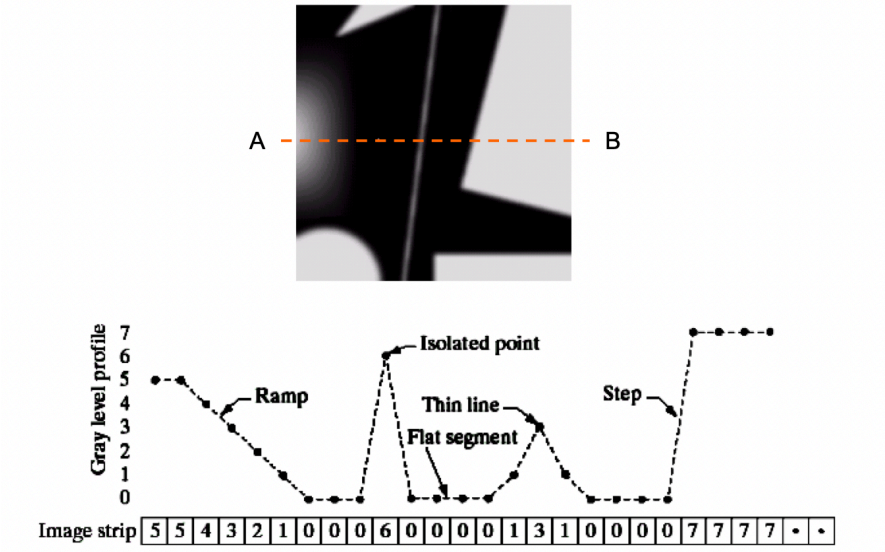
$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

Gaussian filter

4.1.2 Sharpening Spatial Filters

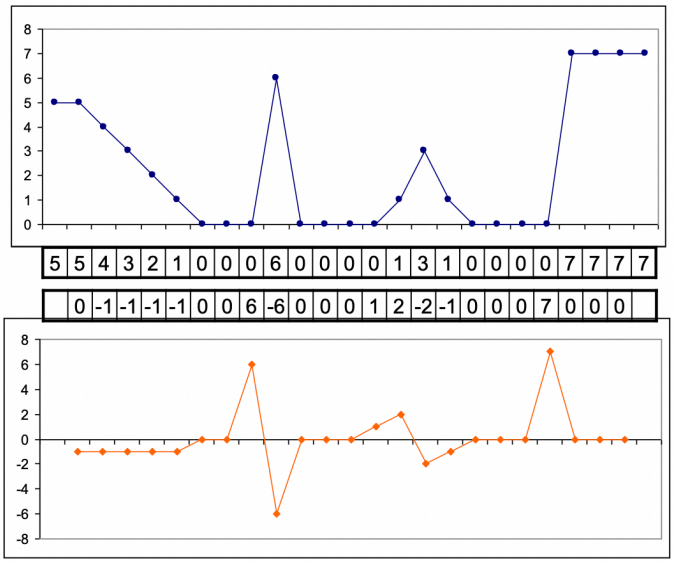
Sharpening spatial filters seek to highlight fine detail, remove blurring from images and highlight edges. Sharpening filters are based on spatial differentiation.

Spatial Differentiation: Differentiation measures the rate of change of a function.



- ***1st Derivative:***It is just the difference between subsequent values and measures the rate of change of the function.

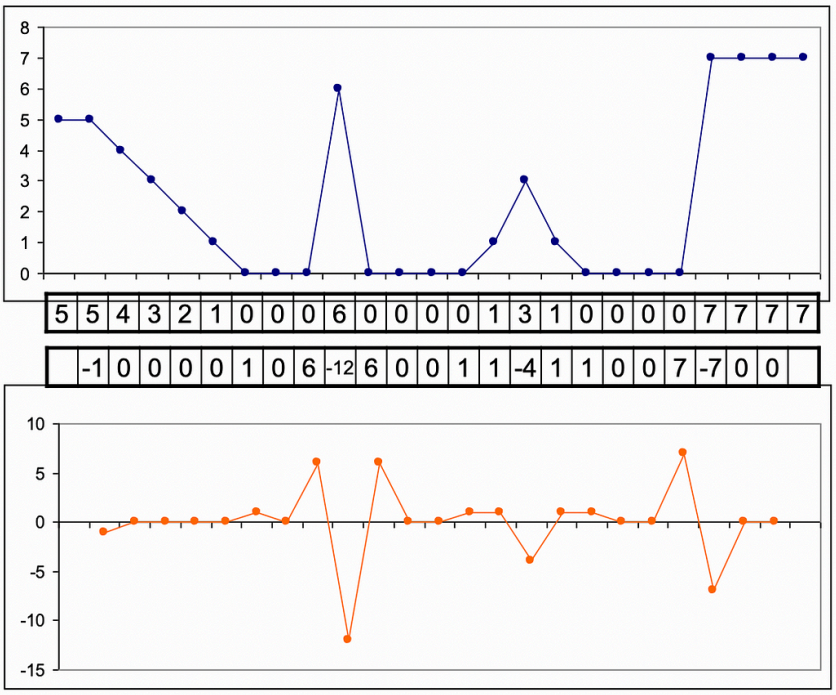
$$\frac{\partial f}{\partial x} = f(x+1) - f(x)$$



1st derivative

2nd Derivative: Takes into account the values both before and after the current value

$$\frac{\partial^2 f}{\partial^2 x} = f(x+1) + f(x-1) - 2f(x)$$



1st and 2nd Derivatives

- 1st order derivatives generally produce **thicker** edges.
- 2nd order derivatives have a stronger response to fine detail e.g. **thin lines.**

- 1st order derivatives have stronger response to grey level step.
- 2nd order derivatives produce a double response at step changes in grey level.

The 2nd derivative is more useful for image enhancement than the 1st derivative. It gives stronger response to fine detail and has simpler implementation.

Sharpening Filters

1. **The Laplacian Filter**
   - The Laplacian filter has an isotropic structure.

$$\nabla^2 f = \frac{\partial^2 f}{\partial^2 x} + \frac{\partial^2 f}{\partial^2 y}$$

The Laplacian formula

   - We can easily build a filter based on this:

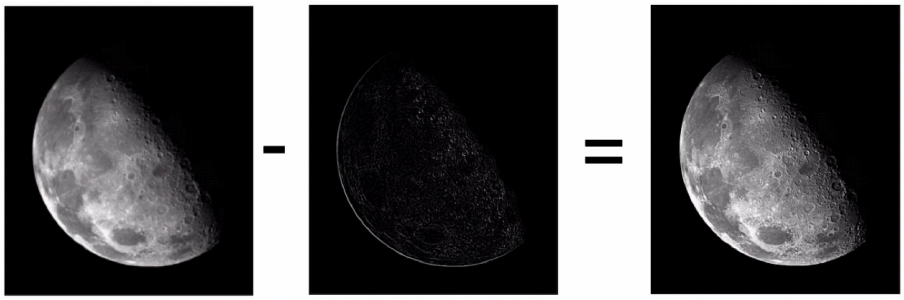
0	1	0
1	-4	1
0	1	0

1	1	1
1	-8	1
1	1	1

Variant of Laplacian

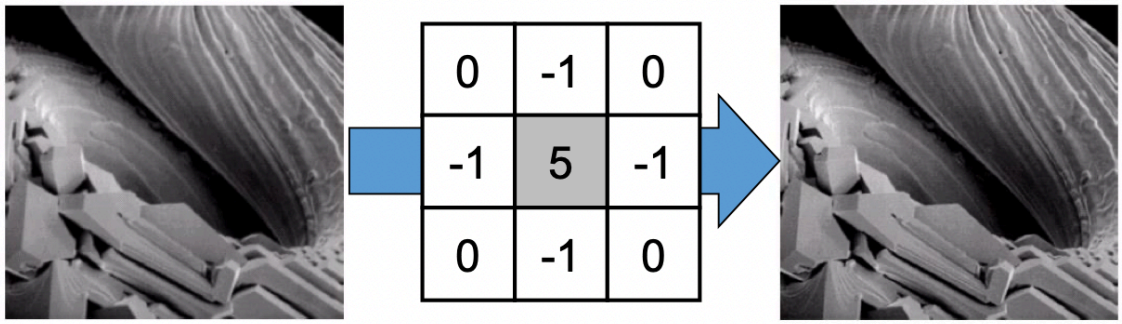
Laplacian Filter

   - The Laplacian highlights edges and other discontinuities. However, the result of a Laplacian filtering is not enhanced image. Laplacian should be subtracted from the original image to generate final sharpened enhanced image.



Original Image | Laplacian Filtered Image | Sharpened Image

g(x,y) = f(x,y)-Laplacian



2. 1st Derivative Filtering

- For practical reasons, this formula can be written as:

$$\nabla f \approx |G_x| + |G_y|$$

Sobel Operators:

- It is a discrete differentiation operator.
- The Sobel edge detection operator extracts all the edges of an image, without worrying about the directions. The main advantage of the Sobel operator is that it provides a differencing and smoothing effect.
- Sobel edge detection operator is implemented as the sum of two directional edges. And the resulting image is a unidirectional outline in the original image.

X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel

-1	-2	-1
0	0	0
1	2	1

$$|G| = |Gx| + |Gy|$$

- Sobel Edge detection operator consists of 3x3 convolution kernels. Gx is a simple kernel and Gy is rotated by 90°
- These Kernels are applied separately to the input image because separate measurements can be produced in each orientation i.e Gx and Gy.
- Advantages:
  - Simple and time efficient computation
  - Very easy at searching for smooth edges
- Limitations:
  - Diagonal direction points are not preserved always
  - Highly sensitive to noise
  - Not very accurate in edge detection
  - Detect with thick and rough edges does not give appropriate results

Prewitt Operator:

- This operator is almost similar to the sobel operator.
- It also detects vertical and horizontal edges of an image. It is one of the best ways to detect the orientation and magnitude of an image.
- It uses the kernels or masks –

$$|G| = |Gx| + |Gy|$$

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad M_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

- Advantages:
  - Good performance on detecting vertical and horizontal edges
  - Best operator to detect the orientation of an image
- Limitations:
  - The magnitude of coefficient is fixed and cannot be changed
  - Diagonal direction points are not preserved always

Robert Operator:

- This gradient-based operator computes the sum of squares of the differences between diagonally adjacent pixels in an image through discrete differentiation.
- Robert's cross operator is used to perform 2-D spatial gradient measurement on an image which is simple and quick to compute. In Robert's cross operator, at each point pixel values represent the absolute magnitude of the input image at that point.
- Robert's cross operator consists of 2x2 convolution kernels. Gx is a simple kernel and Gy is rotated by 90o

$$|G| = |Gx| + |Gy| \qquad |G| = \sqrt{Gx^2 + Gy^2}$$

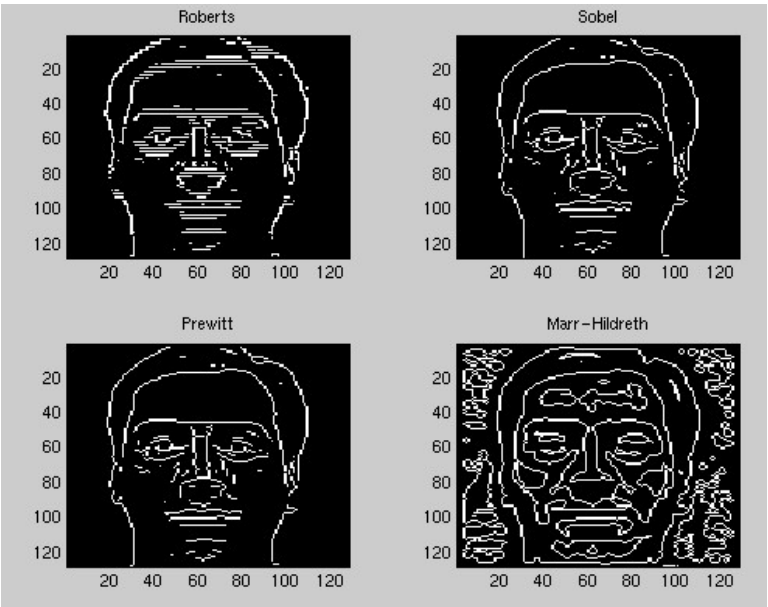
+1	0
0	-1

Gx

0	+1
-1	0

Gy

- Advantages:
  - Detection of edges and orientation are very easy
  - Diagonal direction points are preserved
- Limitations:
  - Very sensitive to noise
  - Not very accurate in edge detection



Example:**3x**3 pixel grayscale image matrix, compute the gradient magnitude using the

Process:

1. **Input Image I:**

$$I = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

2. **Sobel Kernels G_x and G_y:**

- $G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
- $G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

Convolve the Image with Sobel Kernels:

- **Convolution with Sobel X Kernel:**

For the center pixel (50 at (2,2)), calculate:

Gx = (-1)*10 + 0*20 + 1*30 + (-2)*40 + 0*50 + 2*60+ (-1)*70 + 0*80 + 1*90 = 80

90	60	-90
200	80	-200
210	60	-210

Ix

- **Convolution with Sobel Y Kernel:**

For the same center pixel (20 at (2,2)), calculate:

Gy =- 1*10 +- 2*20 + -1*30 + 0*40 + 0*50 + 0*60 +1*70 + 2*80 + 1*90=240

130	200	170
180	240	180
-130	-200	-170

Iy

Calculate Gradient Magnitude:

For the center pixel:

$G = \text{sqrt}(Gx^2 + Gy^2) = \text{sqrt}(0^2 + 0^2) = 0$

Repeat for other pixels:

- After repeating the calculations for the entire matrix, you will get an output matrix representing the gradient magnitude.

$$\text{Gradient Magnitude} = \begin{bmatrix} 158.11 & 208.81 & 192.35 \\ 269.07 & 252.98 & 269.07 \\ 246.98 & 208.81 & 270.19 \end{bmatrix}$$

Result:

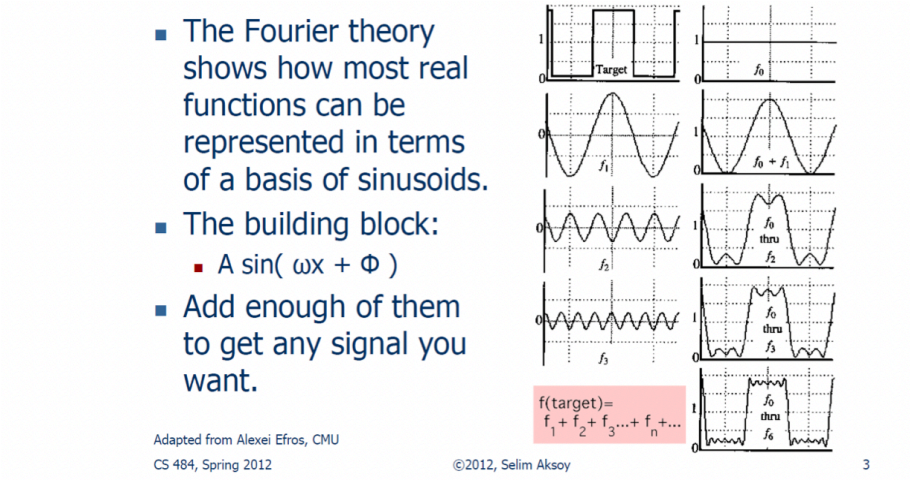
- The Sobel edge detection output will indicate where the edges are located in the image.

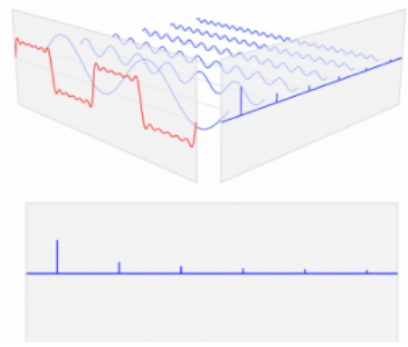
4.2 Frequency Domain Transformation and Filtering

4.2.1 Frequency Domain Transformation

Jean Baptiste Joseph Fourier says any univariate function can be rewritten as a weighted sum of sines and cosines of different frequencies.

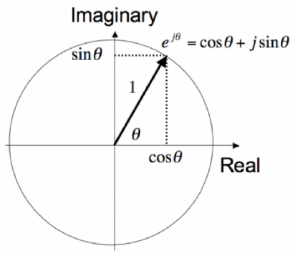
-It is mostly true and called the Fourier Series.





$$e^{j\theta} = \cos\theta + j\sin\theta$$

$$z = x + jy = |z|e^{j\theta}$$



Basics: Euler's Formula

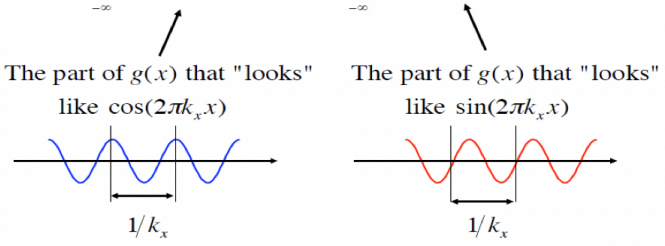
- **Fourier Transform**

The Fourier transform is a mathematical operation that transforms a signal from the time domain to the frequency domain.

a. 1-D Fourier Transform

It is used to analyze signals in order to extract useful information, such as the frequency content of a signal.

$$G(k_x) = \int_{-\infty}^{\infty} g(x)\exp(-j2\pi k_x x)dx$$
$$= \int_{-\infty}^{\infty} g(x)\cos(2\pi k_x x)dx - j\int_{-\infty}^{\infty} g(x)\sin(2\pi k_x x)dx$$



Spatial Coordinates, e.g. x in cm, k_x is spatial frequency in cycles/cm.

$$G(k_x) = \int_{-\infty}^{\infty} g(x)e^{-j2\pi k_x x}dx \qquad \text{Fourier Transform}$$

$$g(x) = \int_{-\infty}^{\infty} G(k_x)e^{j2\pi k_x x}dk_x \qquad \text{Inverse Fourier Transform}$$

b. 2-D Fourier Transform

The 2D Fourier transform is an extension of the 1D Fourier transform, used to analyze two-dimensional signals, such as images.

Fourier Transform

$$G(k_x,k_y) = F\left[g(x,y)\right] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(x,y)e^{-j2\pi\left(k_x x + k_y y\right)}dxdy$$

Inverse Fourier Transform,

$$g(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} G(k_x,k_y)e^{j2\pi\left(k_x x + k_y y\right)}dk_x dk_y$$

Fourier Transform

It is needed to understand the frequency w of the signal. So, reparametrize the signal by w instead of x:

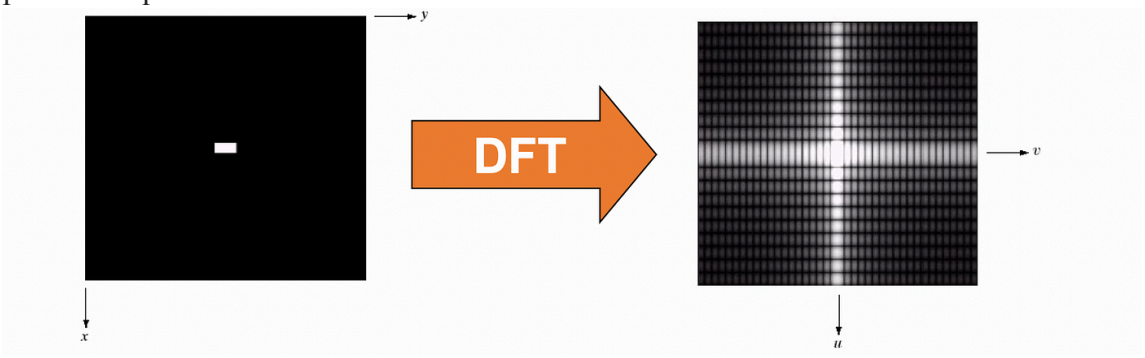
The Fourier transform stores the magnitude and phase at each frequency.

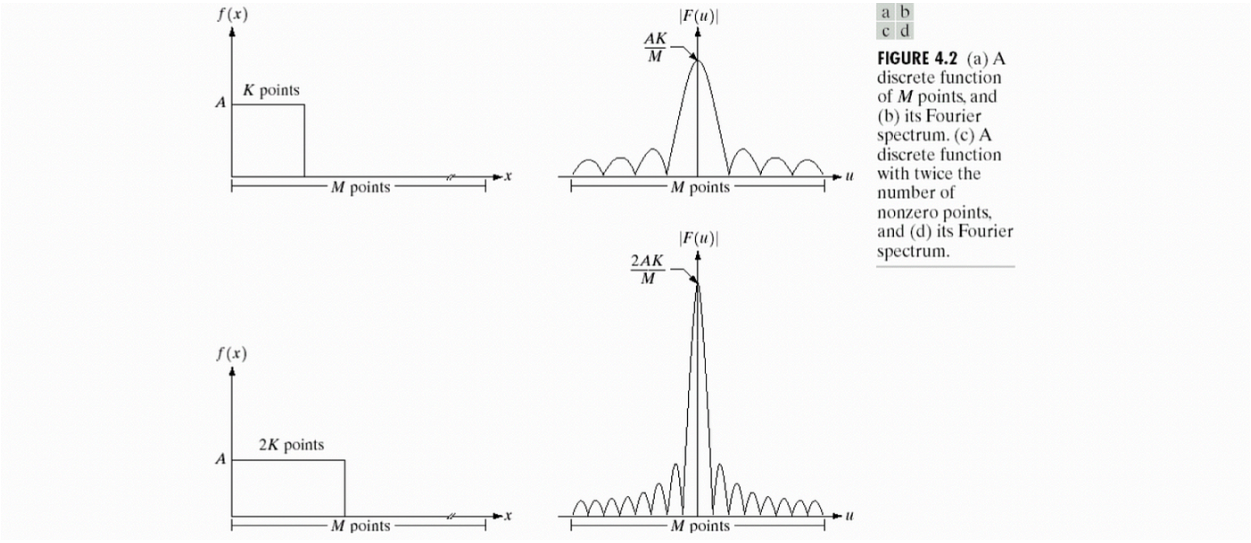
- Magnitude encodes how much signal there is at a particular frequency.
- Phase encodes spatial information (indirectly).

Fourier Transform can always be inverted.

Discrete Fourier Transform

Discrete Fourier Transform is used to analyze discrete signals, such as digital audio and images. It decomposes a discrete signal into its individual frequency components, allowing for the analysis of the frequency content of the signal.
The DFT of a two dimensional image can be visualized by showing the spectrum of the images component frequencies





a b
c d

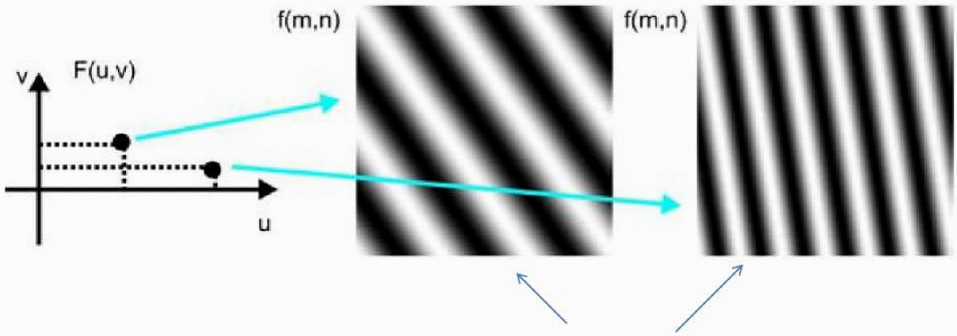
FIGURE 4.2 (a) A discrete function of M points, and (b) its Fourier spectrum. (c) A discrete function with twice the number of nonzero points, and (d) its Fourier spectrum.

Fast Fourier Transform

- The fast Fourier transform (FFT) is an efficient algorithm for computing the discrete Fourier transform (DFT) of a sequence, or its inverse. It can significantly reduce the computational complexity of the DFT, making it possible to perform the transformation on very long sequences in a relatively short amount of time.
- If we take the 2-point DFT and 4-point DFT and generalize them to 8-point, 16-point, …, 2r-point, we get the FFT algorithm.
- To compute the DFT of an N-point sequence using equation (1) would take O(N^2) multiplies and adds. The FFT algorithm computes the DFT using O(N log N) multiplies and adds.
- The FFT is also used for fast extended precision arithmetic (e.g. computing π to a zillion digits), and multiplication of high-degree polynomials, since they also involve convolution. If polynomials p and q have the form:

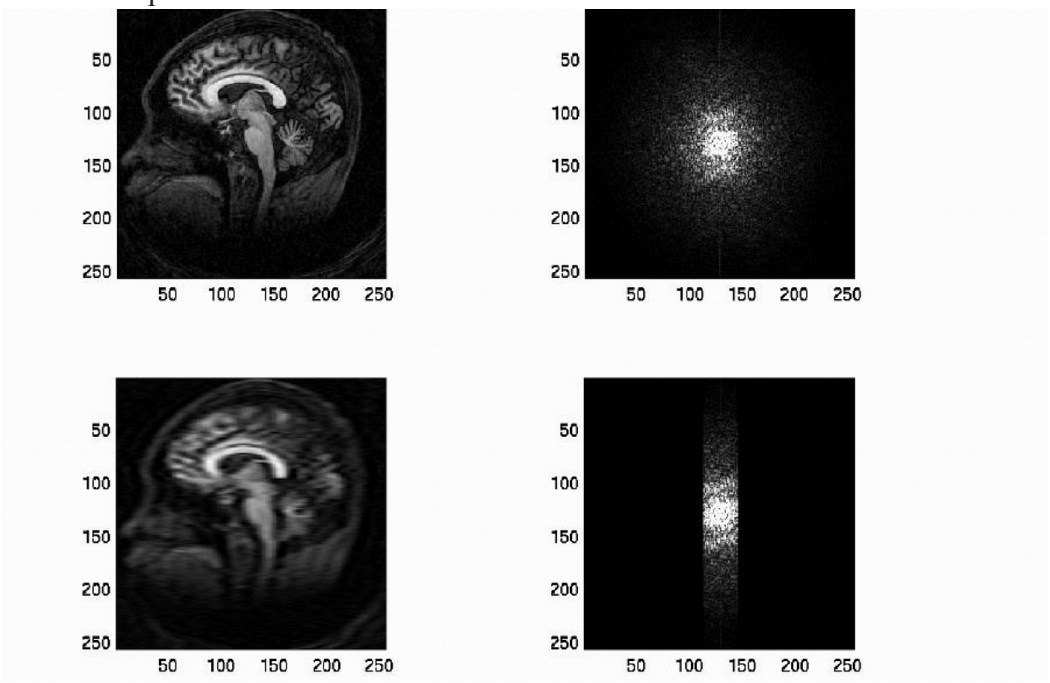
2D Fast Fourier Transform

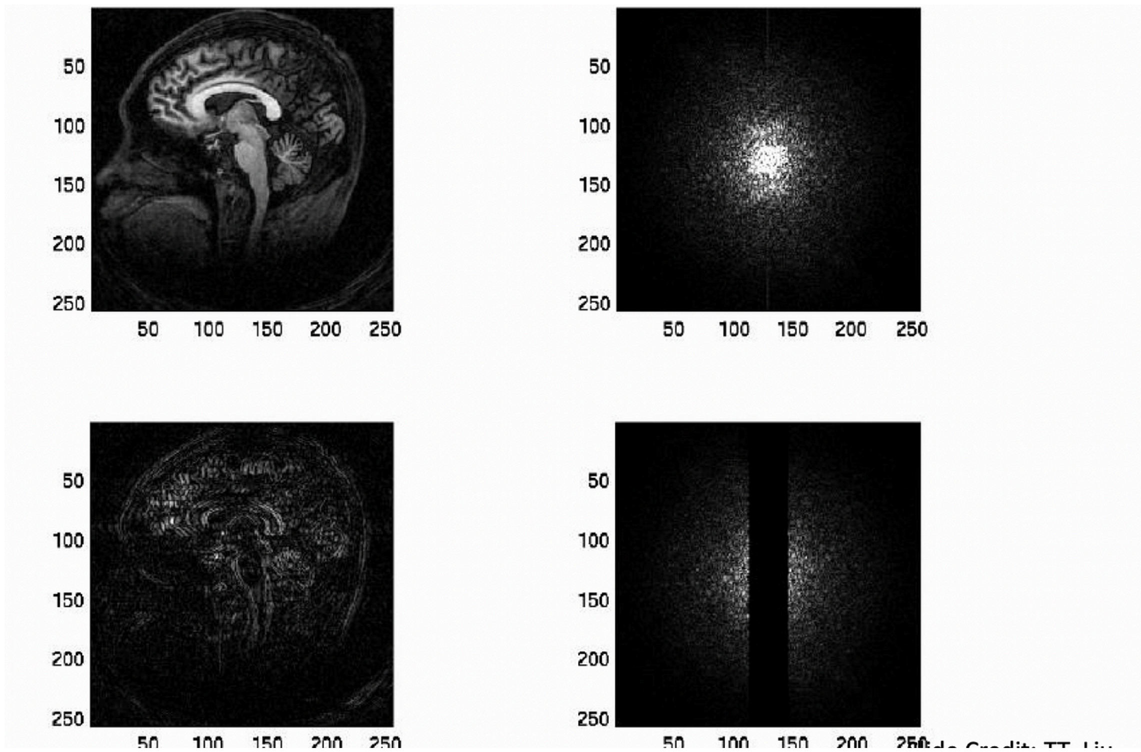
$$F(u,v) = \frac{1}{MN} \cdot \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-i2\pi(xu/M + yv/N)}$$

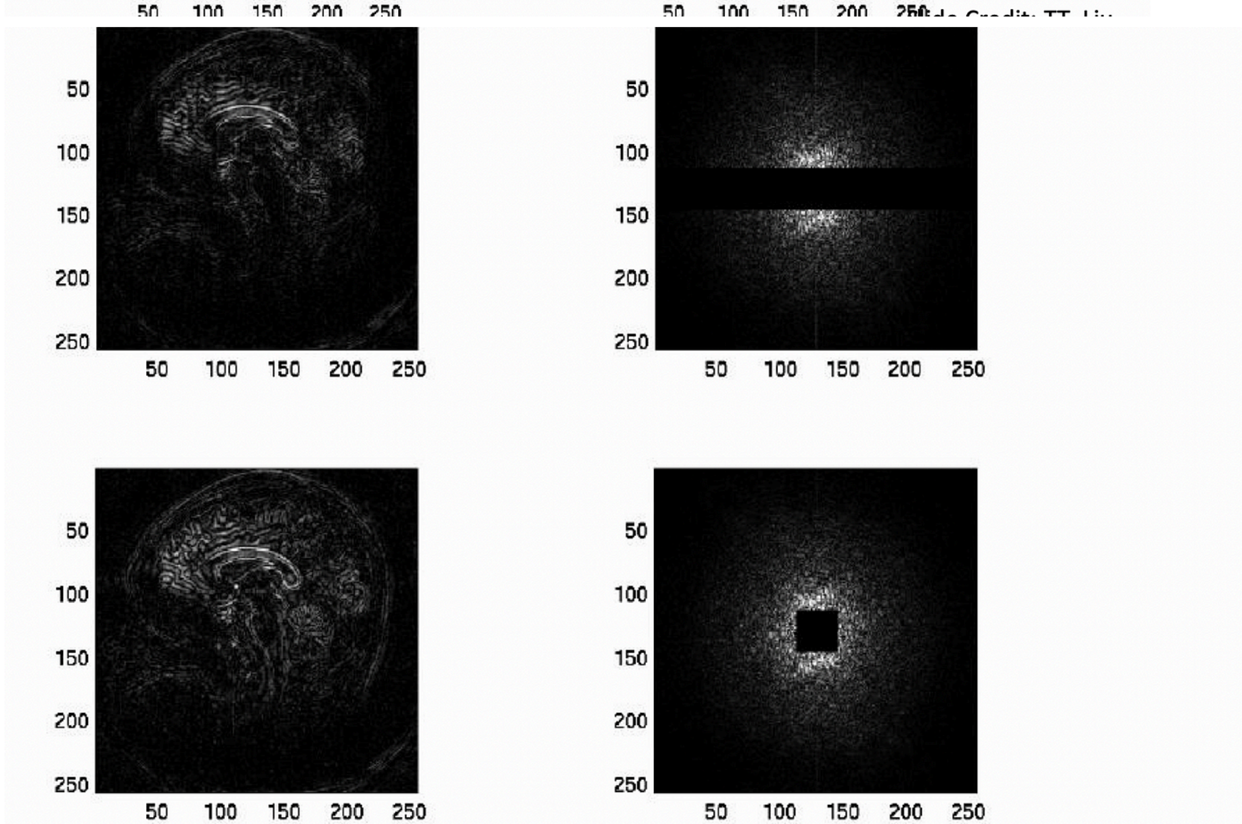


Convolution masks for different frequencies

- Some examples are below:




50
100
150
200
250


50
100
150
200
250

Numerical Example:

Given the 2×2 image matrix:

$$I = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Compute its **2D Fourier Transform (2D FFT)**.

Step 1: Formula for 2D Fourier Transform

The 2D Fourier Transform for a $M \times N$ image $I(x, y)$ is given by:

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} I(x, y) e^{-j2\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)}$$

Where:

- $F(u, v)$ is the frequency domain representation at coordinate (u, v),
- $I(x, y)$ is the spatial domain pixel intensity at coordinate (x, y),
- M, N are the dimensions of the image.

Step 2: Define M and N

For this 2×2 matrix:

- $M = 2$,
- $N = 2$,
- $x, y, u, v \in \{0, 1\}$.

We need to calculate $F(u, v)$ for each combination of (u, v).

Step 3: Compute $F(u, v)$

1. For $F(0, 0)$:

$$F(0, 0) = \sum_{x=0}^{1} \sum_{y=0}^{1} I(x, y) e^{-j2\pi \left(\frac{0 \cdot x}{2} + \frac{0 \cdot y}{2}\right)}$$

Since all exponents are 0, $e^0 = 1$. Therefore:

$$F(0, 0) = I(0, 0) + I(0, 1) + I(1, 0) + I(1, 1)$$

Substituting values:

$$F(0, 0) = 1 + 2 + 3 + 4 = 10$$

2. For $F(0,1)$:

$$F(0,1) = \sum_{x=0}^{1}\sum_{y=0}^{1} I(x,y)e^{-j2\pi\left(\frac{0\cdot x}{2}+\frac{1\cdot y}{2}\right)}$$

The exponent simplifies to $e^{-j\pi y}$:

- For $y = 0$, $e^{-j\pi\cdot 0} = 1$,
- For $y = 1$, $e^{-j\pi\cdot 1} = -1$.

Substituting values:

$$F(0,1) = I(0,0)\cdot 1 + I(0,1)\cdot(-1) + I(1,0)\cdot 1 + I(1,1)\cdot(-1)$$

$$F(0,1) = 1 - 2 + 3 - 4 = -2$$

3. For $F(1,0)$:

$$F(1,0) = \sum_{x=0}^{1}\sum_{y=0}^{1} I(x,y)e^{-j2\pi\left(\frac{1\cdot x}{2}+\frac{0\cdot y}{2}\right)}$$

The exponent simplifies to $e^{-j\pi x}$:

- For $x = 0$, $e^{-j\pi\cdot 0} = 1$,
- For $x = 1$, $e^{-j\pi\cdot 1} = -1$.

Substituting values:

$$F(1,0) = I(0,0)\cdot 1 + I(0,1)\cdot 1 + I(1,0)\cdot(-1) + I(1,1)\cdot(-1)$$

$$F(1,0) = 1 + 2 - 3 - 4 = -4$$

4. For $F(1,1)$:

$$F(1,1) = \sum_{x=0}^{1}\sum_{y=0}^{1} I(x,y)e^{-j2\pi\left(\frac{1\cdot x}{2}+\frac{1\cdot y}{2}\right)}$$

The exponent simplifies to $e^{-j\pi(x+y)}$:

- For $(x,y) = (0,0)$, $e^{-j\pi(0+0)} = 1$,
- For $(x,y) = (0,1)$, $e^{-j\pi(0+1)} = -1$,
- For $(x,y) = (1,0)$, $e^{-j\pi(1+0)} = -1$,
- For $(x,y) = (1,1)$, $e^{-j\pi(1+1)} = 1$.

Substituting values:

$$F(1,1) = I(0,0)\cdot 1 + I(0,1)\cdot(-1) + I(1,0)\cdot(-1) + I(1,1)\cdot 1$$

$$F(1,1) = 1 - 2 - 3 + 4 = 0$$

Step 4: Final Result

The 2D Fourier Transform of the given 2×2 image is:

$$F(u,v) = \begin{bmatrix} 10 & -2 \\ -4 & 0 \end{bmatrix}$$

Here for calculating final F(u,v) will divide all entries with MxN,
Final F(u,v)=[2.5 -0.5]
[-1 0]

4.2.2 Filtering An Image

1. **The Convolution Theorem**

The Fourier transform of the convolution of two functions is the product of their Fourier transforms.

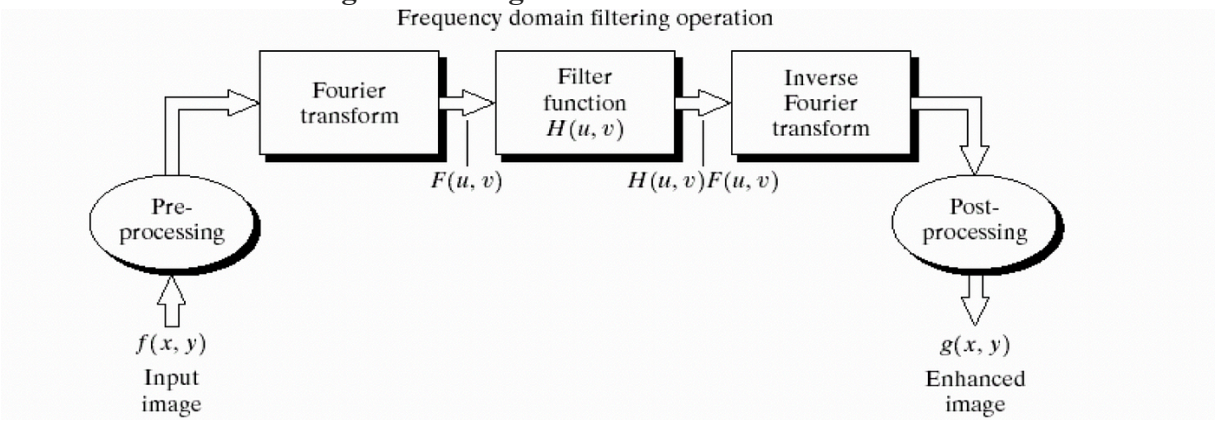
$$\mathrm{F}[g * h] = \mathrm{F}[g]\mathrm{F}[h]$$

The inverse Fourier transform of the product of two Fourier transforms is the convolution of the two inverse Fourier transforms.

$$\mathrm{F}^{-1}[gh] = \mathrm{F}^{-1}[g] * \mathrm{F}^{-1}[h]$$

Convolution in spatial domain is equivalent to multiplication in frequency domain!

2. **The DFT and Image Processing**



Steps involved for filtering in the Frequency Domain

1. Given an input f(x,y) of size M X N, obtain the padding parameters P and Q. Typically, we select P = 2M and Q= 2N.

2. Form a padding image, $f_p(x, y)$, of size P X Q by appending the necessary number of zeros to f(x, y).

3. Multiply $f_p(x, y)$ by $(-1)^{x+y}$ to center its transform

4. Compute the DFT, F(u, v), of the image from step 3.

5. Generate a real, symmetric filter function, H(u, v), of size P X Q with center at coordinates (P/2, Q/2). From the product G(u, v) = H(u, v) F(u, V) using array multiplication;

$$G(i, k) = H(i, k)\, F(i, k).$$

6. Obtain the processed image;

7. Obtain the final processed result, g(x, y), by extracting the M X N region from the top, left quadrant of g_p (x, y).

4.2.2.1 Smoothing Frequency Domain Filters

Smoothing is achieved in the frequency domain by dropping out the high frequency components. The basic model for filtering is:

$$G(u,v) = H(u,v)F(u,v)$$

where $F(u,v)$ is the Fourier transform of the image being filtered and $H(u,v)$ is the filter transform function.

Low pass filters – only pass the low frequencies, drop the high ones.

1. **Ideal Low Pass Filter**

Changing the distance changes the behaviour of the filter. The transfer function for the ideal low pass filter can be given as:

$$H(u,v) = \begin{cases} 1 \text{ if } & D(u,v) \le D_0 \\ 0 \text{ if } & D(u,v) > D_0 \end{cases}$$

where D_0 is a positive constant and $D(u,v)$ is the distance between a point (u, v) in the frequency domain and the centre of the frequency rectangle; that is,

$$D(u, v) = [(u - M / 2)^2 + (v - N / 2)^2]^{1/2}$$

Where, as before, P and Q are the padded sizes.

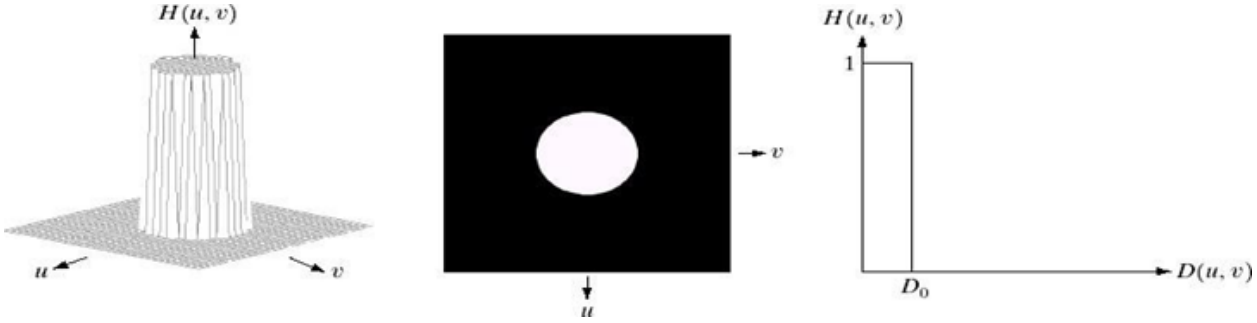


Figure : a. Perspective plot of an ideal lowpass-filter transfer function b. Filter defined as image c. Filter radial cross section

The name ideal indicates that all frequencies on or inside a circle of radius D_0 are passed without attenuation, where as all frequencies outside the circle are completely attenuated. The ideal lowpass filter is rapidly symmetric about the origin, which means that the filter is completely defined by radial cross section by 360^0 yields the filter in 2-D.

For an ILPF cross section, the point of transition between H(u,v) = 1 and H(u, v) = 0 is called the cutoff frequency.

2. **Butterworth Low Pass Filters**

The transfer function of a Butterworth low pass filter of order *n* with cut-off frequency at distance D_0 from the origin is defined as:

$$H(u, v) = \frac{1}{1+[D(u, v) / D_0]^{2n}}$$

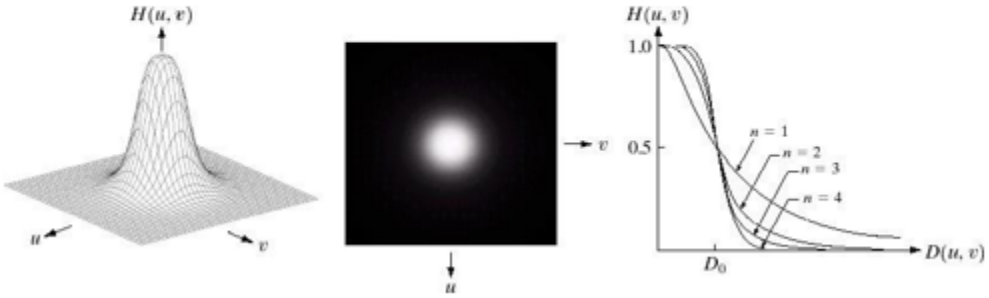


Figure : a. Perspective plot of an Butterworth lowpass-filter transfer function b. Filter defined as image c. Filter radial cross sections of order 1 through 4.

Unlike the ILPF, the BLPF transfer function does not have sharp discontinuity that gives a clear cutoff between passed and filtered frequencies.

3. Gaussian Lowpass Filters

Gaussian lowpass filters (GLPFs) of two dimensions is given by

$$H(u,v) = e^{-D^2(u,v) / 2\sigma^2}$$

Where D(u,v) is the distance from the centre of the frequency rectangle. σ is a measure of spread about the centre By letting $\sigma = D_0$, The transfer function of a Gaussian lowpass filter is defined as:

$$H(u,v) = e^{-D^2(u,v) / 2D_0^2}$$

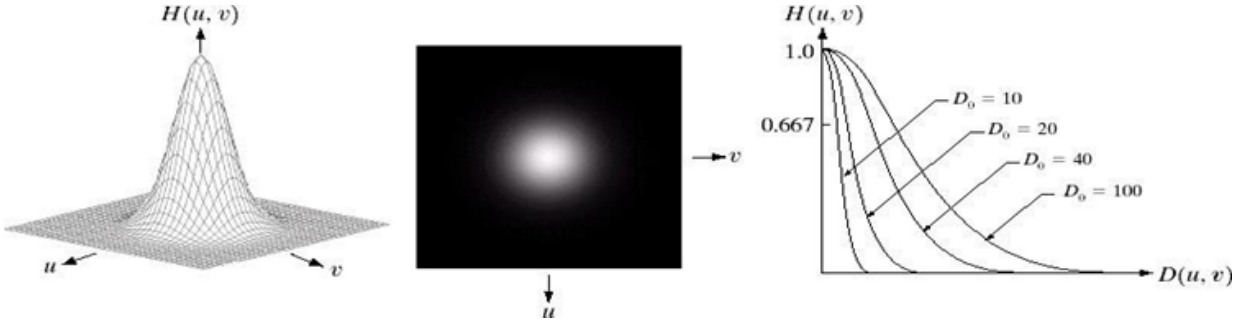


Figure : a. Perspective plot of a Gaussian lowpass-filter transfer function b. Filter defined as image c. Filter radial cross sections for various values of D_0

4.2.2.2 Sharpening in the Frequency Domain Filters using highpass filter

Edges and fine detail in images are associated with high frequency components hence image sharpening can be achieved in the frequency domain by highpass filtering, which attenuates the low frequency components without disturbing high frequency information in the Fourier transform.

High pass filters – only pass the high frequencies, drop the low ones High pass frequencies are precisely the reverse of low pass filters, so:

$$H_{hp}(u, v) = 1 - H_{lp}(u, v)$$

1. Ideal High Pass Filters

The ideal high pass filter is given by:

$$H(u,v) = \begin{cases} 0 \text{ if } & D(u,v) \le D_0 \\ 1 \text{ if } & D(u,v) > D_0 \end{cases}$$

Where D_0 is the cut off frequency.

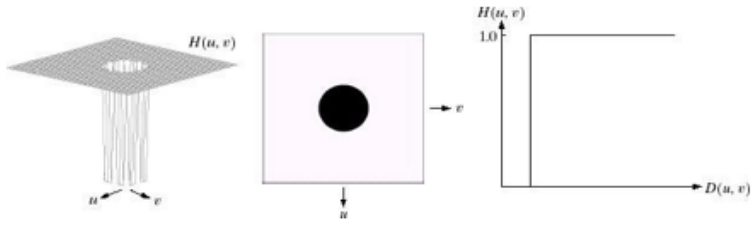


Figure : a. Perspective plot, image representation and cross section of a typical ideal highpass filter

2. Butterworth High Pass Filters

The Butterworth high pass filter is given as:

$$H(u, v) = \frac{1}{1+[D_0 / D(u, v)]^{2n}}$$

n is the order and D_0 is the cut off distance as before.

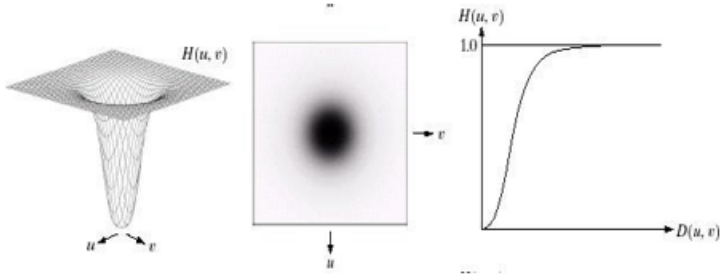


Figure : Butterworth high pass filter

3. Gaussian High Pass Filters

$$H(u, v) = 1 - e^{-D^2(u,v)/2D_0^2}$$

Where D_0 is the cut off distance as before.

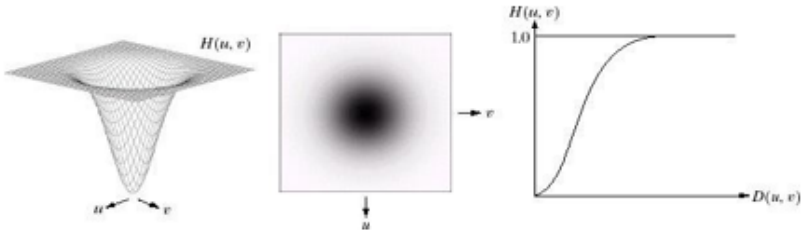
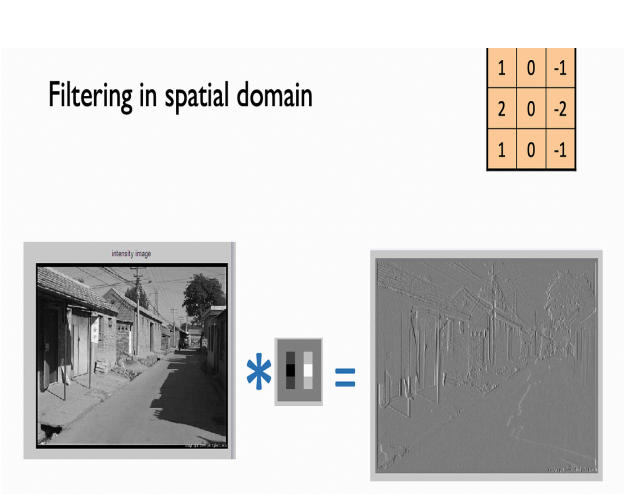
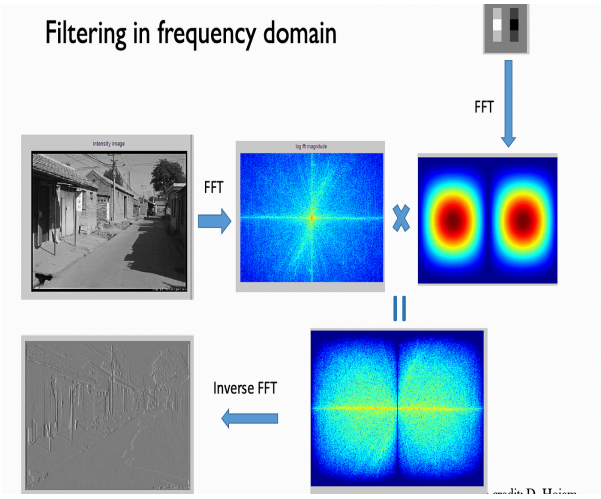


Figure : Gaussian Highpass Filters






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

# Load the image in grayscale
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Get the dimensions of the image
rows, cols = image.shape
crow, ccol = rows // 2, cols // 2  # Center of the image

# Step 1: Perform DFT
dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)  # Shift the zero-frequency component to the center

# Step 2: Define Filter Functions
def create_ideal_filter(shape, filter_type, radius):
    """
    Create an Ideal Low-Pass or High-Pass filter.
    """
    rows, cols = shape
    mask = np.zeros((rows, cols, 2), np.float32)
    for i in range(rows):
        for j in range(cols):
            distance = np.sqrt((i - crow) ** 2 + (j - ccol) ** 2)
            if filter_type == 'low':
                mask[i, j] = 1 if distance <= radius else 0
            elif filter_type == 'high':
                mask[i, j] = 1 if distance > radius else 0
    return mask
```

```
def create_gaussian_filter(shape, filter_type, radius):
    """
    Create a Gaussian Low-Pass or High-Pass filter.
    """
    rows, cols = shape
    mask = np.zeros((rows, cols, 2), np.float32)
    for i in range(rows):
        for j in range(cols):
            distance = np.sqrt((i - crow) ** 2 + (j - ccol) ** 2)
            value = np.exp(-distance ** 2 / (2 * (radius ** 2)))
            mask[i, j] = value if filter_type == 'low' else 1 - value
    return mask

def create_butterworth_filter(shape, filter_type, radius, order):
    """
    Create a Butterworth Low-Pass or High-Pass filter.
    """
    rows, cols = shape
    mask = np.zeros((rows, cols, 2), np.float32)
    for i in range(rows):
        for j in range(cols):
            distance = np.sqrt((i - crow) ** 2 + (j - ccol) ** 2)
            value = 1 / (1 + (distance / radius) ** (2 * order))
            mask[i, j] = value if filter_type == 'low' else 1 - value
    return mask

# Step 3: Apply Filters
radius = 30
order = 2  # For Butterworth filter

# Ideal Filters
ideal_lpf = create_ideal_filter((rows, cols), 'low', radius)
ideal_hpf = create_ideal_filter((rows, cols), 'high', radius)

# Gaussian Filters
gaussian_lpf = create_gaussian_filter((rows, cols), 'low', radius)
gaussian_hpf = create_gaussian_filter((rows, cols), 'high', radius)

# Butterworth Filters
butterworth_lpf = create_butterworth_filter((rows, cols), 'low', radius, order)
butterworth_hpf = create_butterworth_filter((rows, cols), 'high', radius, order)

# Step 4: Perform filtering and inverse DFT
def apply_filter(dft_shift, mask):
    filtered_dft = dft_shift * mask
```

```
    dft_ishift = np.fft.ifftshift(filtered_dft)
    img_back = cv2.idft(dft_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    return img_back

# Apply each filter
results = {
    "Original": image,
    "Ideal Low-Pass": apply_filter(dft_shift, ideal_lpf),
    "Ideal High-Pass": apply_filter(dft_shift, ideal_hpf),
    "Gaussian Low-Pass": apply_filter(dft_shift, gaussian_lpf),
    "Gaussian High-Pass": apply_filter(dft_shift, gaussian_hpf),
    "Butterworth Low-Pass": apply_filter(dft_shift, butterworth_lpf),
    "Butterworth High-Pass": apply_filter(dft_shift, butterworth_hpf)
}

# Step 5: Display Results
plt.figure(figsize=(12, 8))
for i, (key, img) in enumerate(results.items()):
    plt.subplot(2, 4, i + 1), plt.imshow(img, cmap='gray'), plt.title(key)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

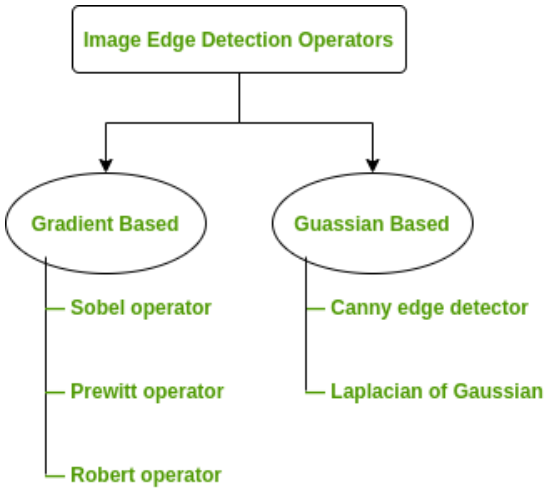
4.3 ***Edge Detection***

- Edges are significant local changes of intensity in a digital image. An edge can be defined as a set of connected pixels that forms a boundary between two disjoint regions. There are three types of edges:
    - Horizontal edges
    - Vertical edges
    - Diagonal edges

- Edge Detection is a method of segmenting an image into regions of discontinuity. It is a widely used technique in digital image processing like
    - pattern recognition
    - image morphology
    - feature extraction
- Edge detection allows users to observe the features of an image for a significant change in the gray level. This texture indicates the end of one region in the image and the beginning of another.
- It reduces the amount of data in an image and preserves the structural properties of an image.
- Steps:Steps:
    - Read the image.

- Convert into grayscale if it is colored.
- Convert into the double format.
- Define the mask or filter.
- Detects the edges along X-axis.
- Detects the edges along the Y-axis.
- Combine the edges detected along X and Y axes.
- Display all the images.

Edge Detection Operators are of two types:



Marr-Hildreth Operator or Laplacian of Gaussian (LoG):

- It is a gaussian-based operator which uses the Laplacian to take the second derivative of an image. This really works well when the transition of the grey level seems to be abrupt.
- It works on the zero-crossing method i.e when the second-order derivative crosses zero, then that particular location corresponds to a maximum level. It is called an edge location.
- Here the **Gaussian operator reduces the noise** and the **Laplacian operator detects the sharp edges**.
  Laplacian, the input image is represented as a set of discrete pixels. 3 commonly used kernels are as following:

0	1	0
1	-4	1
0	1	0

1	1	1
1	-8	1
1	1	1

-1	2	-1
2	-4	2
-1	2	-1

LoG operator is computed from

$$\mathrm{LoG} = \frac{\partial^2}{\partial x^2} G(x,y) + \frac{\partial^2}{\partial y^2} G(x,y) = \frac{x^2 + y^2 - 2\sigma^2}{\sigma^4} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

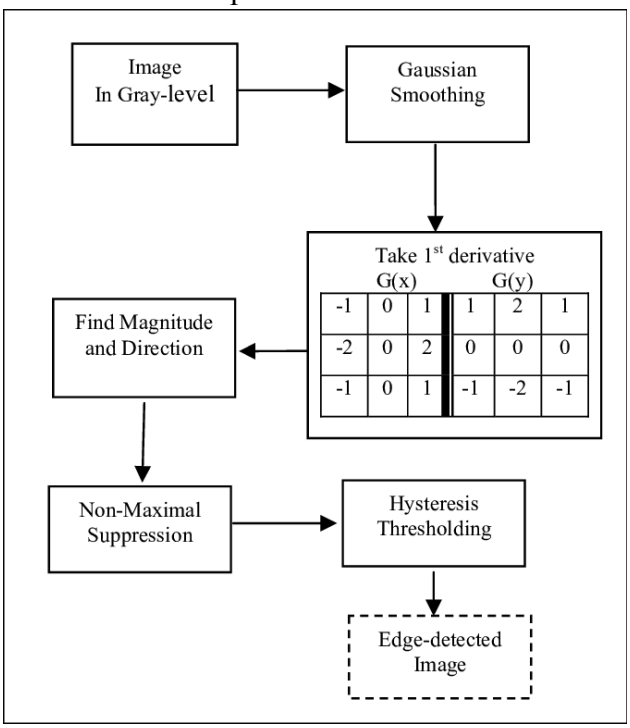
- Advantages:
    - Easy to detect edges and their various orientations
    - There is fixed characteristics in all directions
- Limitations:
    - Very sensitive to noise
    - The localization error may be severe at curved edges
    - It generates noisy responses that do not correspond to edges, so-called “false edges”

Canny Edge Detection:

Canny edge detection algorithm developed by John F. Canny in 1986. Usually, in Matlab and OpenCV we use the canny edge detection for many popular tasks in edge detection such as lane detection, sketching, border removal, now we will learn the internal working and implementation of this algorithm from scratch.

The basic steps involved in this algorithm are:

1. Noise reduction using Gaussian filter
2. Gradient calculation along the horizontal and vertical axis
3. Non-Maximum suppression of false edges
4. Double thresholding for segregating strong and weak edges
5. Edge tracking by hysteresis
6. Now let us understand these concepts in detail:



1. Noise reduction using Gaussian filter

This step is of utmost importance in the Canny edge detection. It uses a Gaussian filter for the removal of noise from the image, it is because this noise can be assumed as edges due to sudden intensity change by the edge detector. The sum of the elements in the Gaussian kernel is 1, so the kernel should be normalized before applying convolution to the image. In this tutorial, we will use a kernel of size 5 X 5 and sigma = 1.4, which will blur the image and remove the noise from it. The equation for Gaussian filter kernel is

$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

2. Gradient calculation

When the image is smoothed, the derivatives Ix and Iy are calculated w.r.t x and y axis. It can be implemented by using the Sobel-Feldman kernels convolution with image as given:

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

Sobel Kernels

after applying these kernel we can use the gradient magnitudes and the angle to further process this step. The magnitude and angle can be calculated as

$$|G| = \sqrt{I_x^2 + I_y^2},$$
$$\theta(x, y) = arctan\left(\frac{I_y}{I_x}\right)$$

Gradient magnitude and angle

3. Non-Maximum Suppression

This step aims at reducing the duplicate merging pixels along the edges to make them uneven. For each pixel find two neighbors in the positive and negative gradient directions, supposing that each neighbor occupies the angle of pi /4, and 0 is the direction straight to the right. If the magnitude of the current pixel is greater than the magnitude of the neighbors, nothing changes, otherwise, the magnitude of the current pixel is set to zero.

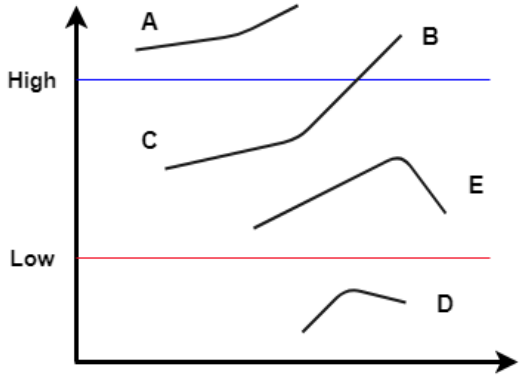
4. Hysteresis Thresholding

Non-max suppression outputs a more accurate representation of real edges in an image. But you can see that some edges are brighter than others. The brighter ones can be considered as strong edges but the lighter ones can actually be edges or they can be because of noise. To solve the problem of ***"which edges are really edges and which are not"*** Canny uses the Hysteresis thresholding. In this, we set two thresholds 'High' and 'Low'.

- Any edges with intensity greater than 'High' are the sure edges.

- Any edges with intensity less than ‘Low’ are sure to be non-edges.
- The edges between ‘High’ and ‘Low’ thresholds are classified as edges only if they are connected to a sure edge otherwise discarded.

Let’s take an example to understand



Here, A and B are sure-edges as they are above ‘High’ threshold. Similarly, D is a sure non-edge. Both ‘E’ and ‘C’ are weak edges but since ‘C’ is connected to ‘B’ which is a sure edge, ‘C’ is also considered as a strong edge. Using the same logic ‘E’ is discarded. This way we will get only the strong edges in the image.

- Advantages:
  - It has good localization
  - It extract image features without altering the features
  - Less Sensitive to noise
- Limitations:
  - There is false zero crossing
  - Complex computation and time consuming

Some Real-world Applications of Image Edge Detection:

- medical imaging, study of anatomical structure
- locate an object in satellite images
- automatic traffic controlling systems ,
- face recognition, and fingerprint recognition

Numerical Example:

Step 1: Input Image

- Let’s consider a simple grayscale image represented as a 3x3 pixel matrix:

Step 2: Apply Gaussian Blur

- The first step in the Canny edge detection algorithm is to reduce noise in the image using a

Input Image:

$$I = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

Gaussian Kernel (Mask):

$$G = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Gaussian Kernel:

- Convolve the Gaussian kernel with the original image matrix. The output of the Gaussian blur will reduce noise and make the edge detection more robust.

  Step 1: For Pixel at (1, 1)

  The region for convolution (zero-padded):

$$\text{Region} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 10 & 20 \\ 0 & 40 & 50 \end{bmatrix}$$

  Element-wise multiplication with G:

$$\frac{1}{16} \begin{bmatrix} 0 \cdot 1 & 0 \cdot 2 & 0 \cdot 1 \\ 0 \cdot 2 & 10 \cdot 4 & 20 \cdot 2 \\ 0 \cdot 1 & 40 \cdot 2 & 50 \cdot 1 \end{bmatrix} = \frac{1}{16} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 40 & 40 \\ 0 & 80 & 50 \end{bmatrix}$$

  Summation:

$$\frac{1}{16}(0 + 0 + 0 + 0 + 40 + 40 + 0 + 80 + 50) = \frac{210}{16} = 13.125$$

  Result for pixel (1, 1): **13**

Result of Gaussian Blur: After applying Gaussian blur**, let's assume the resulting image matrix** [

Final Smoothed Image:

$$I_{\text{smoothed}} = \begin{bmatrix} 13 & 23 & 21 \\ 33 & 50 & 43 \\ 41 & 66 & 54 \end{bmatrix}$$

Step 3: Compute the Gradient Magnitude and Direction

- The next step is to calculate the gradient magnitude and direction at each pixel using the Sobel operator.

$$I_{\text{smoothed}} = \begin{bmatrix} 13 & 23 & 21 \\ 33 & 50 & 43 \\ 41 & 66 & 54 \end{bmatrix}$$

Sobel Kernels:

- **Sobel kernel in x-direction (G_x):**

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

- **Sobel kernel in y-direction (G_y):**

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Gradient Calculation:

- Convolve the Sobel X and Y kernels with the blurred image matrix.

  - **Pixel (3,2) - Convolution with G_x:**

$$G_x(3,2) = (-1)(66) + (0)(54) + (1)(0) + (-2)(0) + (0)(0) + (2)(0) + (-1)(0) + (0)(0) + (1)(0)$$

$$G_x(3,2) = -66 + 0 = -66$$

  - **Pixel (3,3) - Convolution with G_x:**

$$G_x(3,3) = (-1)(54) + (0)(0) + (1)(0) + (-2)(0) + (0)(0) + (2)(0) + (-1)(0) + (0)(0) + (1)(0)$$

$$G_x(3,3) = -54 + 0 = -54$$

Gradient in x-direction (G_x) - Full 3x3 matrix:

$$G_x = \begin{bmatrix} 41 & -189 & -161 \\ 36 & -182 & -151 \\ 13 & -66 & -54 \end{bmatrix}$$

- **Pixel (3,2) - Convolution with G_y:**

$$G_y(3,2) = (-1)(66) + (-2)(54) + (-1)(0) + (0)(0) + (0)(0) + (0)(0) + (1)(0) + (2)(0) + (1)(0)$$
$$G_y(3,2) = -66 - 108 + 0 = -174$$

- **Pixel (3,3) - Convolution with G_y:**

$$G_y(3,3) = (-1)(54) + (-2)(0) + (-1)(0) + (0)(0) + (0)(0) + (0)(0) + (1)(0) + (2)(0) + (1)(0)$$
$$G_y(3,3) = -54 + 0 = -54$$

Gradient in y-direction (G_y) - Full 3x3 matrix:

$$G_y = \begin{bmatrix} 147 & 109 & 33 \\ -176 & -70 & -43 \\ -227 & -174 & -54 \end{bmatrix}$$

Gradient Magnitude: G = sqrt(Gx² + Gy²)

- **Pixel (3,2):**

$$G(3,2) = \sqrt{(-66)^2 + (-174)^2} = \sqrt{4356 + 30276} = \sqrt{34632} \approx 186.03$$

- **Pixel (3,3):**

$$G(3,3) = \sqrt{(-54)^2 + (-54)^2} = \sqrt{2916 + 2916} = \sqrt{5832} \approx 76.37$$

Gradient Magnitude (G) - Full 3x3 matrix:

$$G = \begin{bmatrix} 152.59 & 218.23 & 164.30 \\ 179.82 & 194.93 & 157.04 \\ 227.69 & 186.03 & 76.37 \end{bmatrix}$$

Gradient Direction: θ = atan(Gy/Gx)

Direction Calculations for Selected Pixels

1. Pixel (1,1):

$$\text{Direction}_{1,1} = \arctan\left(\frac{0}{0}\right) \text{ (undefined). Set direction to } 0^\circ \text{ or skip.}$$

2. Pixel (1,2):

$$\text{Direction}_{1,2} = \arctan\left(\frac{10}{-10}\right) = \arctan(-1) = -45^\circ \text{ (or } 315^\circ)$$

3. Pixel (1,3):

$$\text{Direction}_{1,3} = \arctan\left(\frac{20}{-20}\right) = \arctan(-1) = -45^\circ \text{ (or } 315^\circ)$$

4. Pixel (2,2):

$$\text{Direction}_{2,2} = \arctan\left(\frac{-17}{17}\right) = \arctan(-1) = -45^\circ \text{ (or } 315^\circ)$$

Continue this calculation for all pixels.

Complete Direction Matrix

After calculating the directions for all pixels:

$$\text{Direction (degrees)} = \begin{bmatrix} 0 & -45 & -45 & -45 & 0 \\ 45 & -45 & 0 & 45 & 45 \\ 63.4 & -90 & 0 & 90 & 63.4 \\ 135 & -135 & -180 & -135 & 135 \\ 0 & 45 & 45 & 45 & 0 \end{bmatrix}$$

- **Pixel (3,2):**

$$\theta(3,2) = \arctan\left(\frac{-174}{-66}\right) = \arctan(2.636) \approx 69.66^\circ$$

- **Pixel (3,3):**

$$\theta(3,3) = \arctan\left(\frac{-54}{-54}\right) = \arctan(1) \approx 45^\circ$$

Gradient Direction (θ) - Full 3x3 matrix:

$$\theta = \begin{bmatrix} 75.45^\circ & -30.96^\circ & -11.58^\circ \\ -78.51^\circ & 21.11^\circ & 15.91^\circ \\ -86.44^\circ & 69.66^\circ & 45^\circ \end{bmatrix}$$

Step 4: Non-Maximum Suppression

Rounding the Gradient Directions:

- 0°: **Horizontal edges (left to right)**
- 45°: **Diagonal edges (top-left to bottom-right)**
- 90°: **Vertical edges (top to bottom)**
- 135°: **Diagonal edges (top-right to bottom-left)**

Now, we'll **round** the gradient directions and apply non-maximum suppression.

- **Gradient Direction (θ)**: -11.58° (rounded to 0° for horizontal edge).
- **Pixel Magnitudes**: 164.30.

We need to compare the magnitude of the pixel (1,3) with the two neighboring pixels in the 0° direction (horizontal):

- Left pixel (1,2): Magnitude = 218.23.
- Right pixel (1,4): Not applicable since we are at the image border.

Since 218.23 > 164.30, **(1,3)** will be suppressed (set to 0).

Final Result After Non-Maximum Suppression:

$$\text{Suppressed Image} = \begin{bmatrix} 0 & 218.23 & 0 \\ 0 & 194.93 & 0 \\ 227.69 & 186.03 & 0 \end{bmatrix}$$

Step 5: Hysteresis Thresholding

Step-by-Step Calculation for Hysteresis Thresholding:

Given:

- **High Threshold** = 200
- **Low Threshold** = 190
- **Strong Edge Value** = 255 (will be assigned to pixels considered strong edges)
- **Other Pixels** = 0 (weak edges or non-edge pixels)

Step 1: Identify Strong Edges

Strong edges are pixels with gradient magnitudes greater than the **High Threshold** (200).

From the suppressed image, the strong edges are:

- (1,2) = 218.23 (strong edge)
- (2,1) = 227.69 (strong edge)

Step 2: Identify Weak Edges

Weak edges are pixels with gradient magnitudes between the **Low Threshold** (190) and **High Threshold** (200).

From the suppressed image, the weak edge is:

- (2,2) = 194.93 (weak edge)

Step 3: Connect Weak Edges to Strong Edges

Any weak edge connected to a strong edge will be retained as an edge.

- **(2,2)** is a weak edge, and it is connected to the strong edges at **(1,2)** (218.23) and **(2,1)** (227.69). Hence, **(2,2)** will be retained as an edge.

Step 4: Set Edge Pixels

- Strong edge pixels will be set to **255**.
- All other pixels (weak edges or non-edges) will be set to **0**.

Final Image After Hysteresis Thresholding (with Strong Edges set to 255):

$$\text{Thresholded Image} = \begin{bmatrix} 0 & 255 & 0 \\ 0 & 255 & 0 \\ 255 & 0 & 0 \end{bmatrix}$$

Explanation:

- **Strong edges** at **(1,2)** and **(2,1)** are set to **255**.
- **Weak edge** at **(2,2)** is connected to strong edges and is also set to **255**.
- All other pixels are set to **0**.

Step 6: Edge Tracking by Hysteresis

- Finally, we connect the weak edges to strong edges if they are connected. If a weak edge is connected to a strong edge, we keep it; otherwise, we discard it.

1. Sobel Edge Detection

```
import cv2
import numpy as np

# Read the image
img = cv2.imread('image.png', cv2.IMREAD_GRAYSCALE)

# Apply Sobel operator
sobel_x = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)
sobel_edge = np.hypot(sobel_x, sobel_y)

# Display result
cv2.imshow('Sobel Edge Detection', sobel_edge)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2. Prewitt Edge Detection

```
# Define Prewitt kernels
prewitt_x = np.array([[1, 0, -1], [1, 0, -1], [1, 0, -1]])
prewitt_y = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])

# Apply Prewitt operator
prewitt_edge_x = cv2.filter2D(img, -1, prewitt_x)
prewitt_edge_y = cv2.filter2D(img, -1, prewitt_y)
prewitt_edge = np.hypot(prewitt_edge_x, prewitt_edge_y)

# Display result
cv2.imshow('Prewitt Edge Detection', prewitt_edge)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

3. Canny Edge Detection

```
# Apply Canny edge detection
canny_edges = cv2.Canny(img, 100, 200)

# Display result
cv2.imshow('Canny Edge Detection', canny_edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

4. Laplacian Edge Detection

```
# Apply Laplacian operator
laplacian_edges = cv2.Laplacian(img, cv2.CV_64F)

# Display result
cv2.imshow('Laplacian Edge Detection', laplacian_edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```