Digitized Notes - Image Processing

Aung Naing Oo

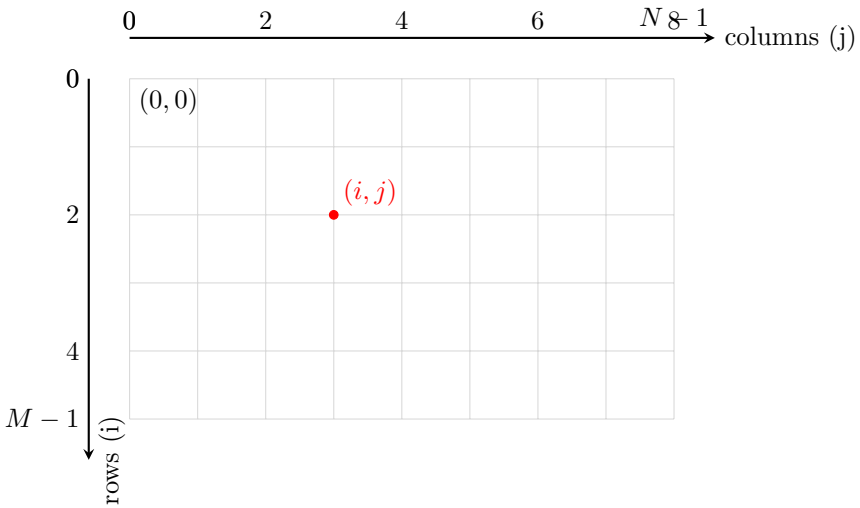
December 10, 2025

Grid and Coordinate System

The notes show a grid with coordinates labeled x_1 through x_5 horizontally and y_1 through y_5 vertically.

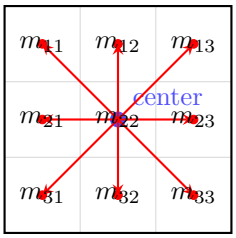
	y_1	y_2	y_3	y_4	y_5
x_1					
x_2					
x_3					
x_4					
x_5					

- Coordinate points would be for example (x_3, y_2), (x_1, y_5), etc.



Binary Representation

- $0 \rightarrow$ black
- $1 \rightarrow$ white



$r = 1$ means all directions from the center

$$\text{sizeM} = 2r + 1, \qquad \text{sizeN} = 2r + 1$$

General Reminder

Need to remember the **grid format** and **coordinate system**.

Clipping/Normalization Formula

The formula for normalization (or scaling/clipping) of a function $f(x, y)$ to the range $[0, 255]$ is noted:

$$g(x, y) \to 255 \cdot \frac{f(x, y) - \min}{\max - \min}$$

- $f(x, y)$ is the original intensity value at (x, y).
- min and max are the minimum and maximum intensity values in the image.

```
rescaledNebulaNoisy = 255*(noisy - Min[noisy])/(Max[noisy] - Min[noisy]);
rescaledNebulaNoisy = Rescale[noisy, MinMax@noisy, {0,255}];
```

Intensity Transformation

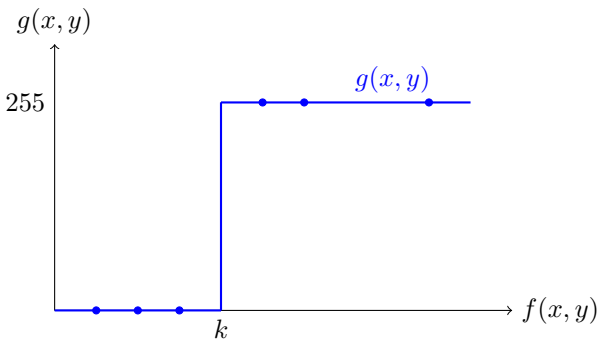
The general form for an intensity transformation where $f(x, y)$ is the input image and $g(x, y)$ is the output image using a transformation T:

$$g(x, y) = T[f(x, y)]$$

Thresholding

Thresholding is the process of producing a **binary image** (2 levels of intensity, e.g., 0 and 1 or black and white).

- The thresholding operation can be visualized as:



- The thresholding function is:

$$g(x, y) = \begin{cases} 0, & f(x, y) < k \\ 255, & f(x, y) \geq k \end{cases}$$

How the Threshold Works

The function processes the input image img and applies the following rule to each pixel's intensity value:

- If the pixel's intensity is less than threshold value, the resulting binary pixel is set to black (intensity 0).
- If the pixel's intensity is greater than the threshold value, the resulting binary pixel is set to white (intensity 1, or maximum).

Example in Mathematica:

```
Binarize[lowPassFiltered, 0.2]
```

Log Transformation

Log transformation is used to expand dark pixel values in an image while compressing higher intensity values. The general formula is:

$$g(x, y) = c \cdot \log\left(1 + f(x, y)\right)$$

where:

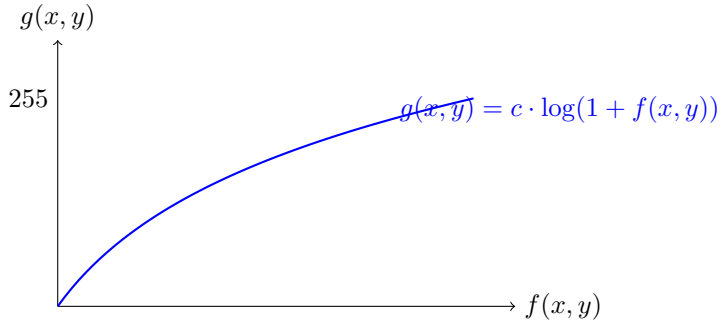
- $f(x, y)$ is the input pixel value.
- $g(x, y)$ is the output pixel value.
- c is a scaling constant, often chosen so that the maximum output value is 255.

Finding the Constant c

When $g(x, y) = 255$ and $f(x, y) = 255$:

$$255 = c \cdot \log(1 + 255)$$

$$c = \frac{255}{\log(256)}$$



Reversing Intensity Levels

Reversing (or inverting) the intensity levels of an image produces a **negative** image. The formula is:

$$g(x, y) = (L - 1) - f(x, y)$$

where:

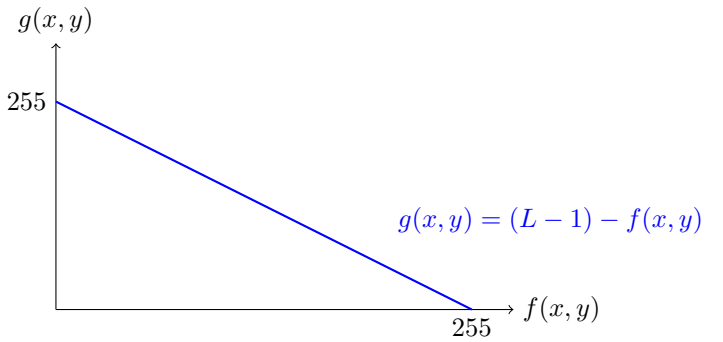
- $f(x, y)$ is the input pixel value.
- $g(x, y)$ is the output pixel value (inverted).
- L is the number of intensity levels (e.g., 256 for an 8-bit image, so $L - 1 = 255$).

Example

For an 8-bit image ($L = 256$):

$$g(x, y) = 255 - f(x, y)$$

This transformation maps dark pixels to bright pixels and vice versa.



Power Law Transformation

Power law transformation is used to enhance image contrast by applying a nonlinear transformation. The general formula is:

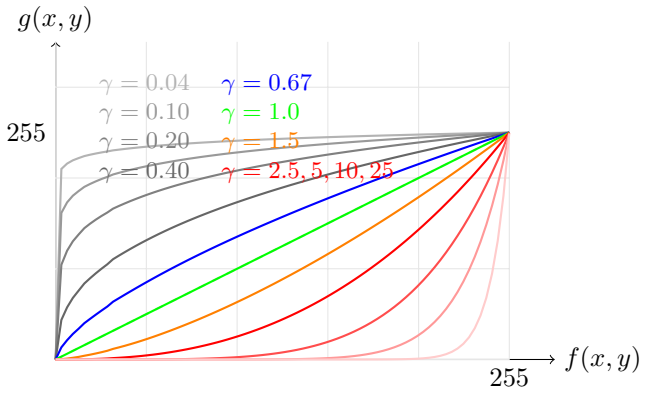
$$g(x, y) = c \cdot [f(x, y)]^{\gamma}$$

where:

- $f(x, y)$ is the input pixel value.
- $g(x, y)$ is the output pixel value.
- c is a scaling constant.
- γ (gamma) is the exponent that controls the transformation:
  - $\gamma > 1$ brightens the image (expands dark values).
  - $\gamma < 1$ darkens the image (compresses bright values).
  - $\gamma = 1$ produces a linear transformation.

Power Law Transformation - Multiple Gamma Values

The following figure shows how different gamma values affect the intensity transformation:



Observations:

- For $\gamma < 1$: curves are above the diagonal, brightening the image.
- For $\gamma = 1$: linear transformation (diagonal line).
- For $\gamma > 1$: curves are below the diagonal, darkening the image and increasing contrast in bright regions.
- As γ increases, the compression of bright values becomes more pronounced.

Contrast Stretching

Contrast stretching is a technique used to expand the range of intensity levels in an image. It uses a piecewise linear transformation to map input intensity levels to output levels.

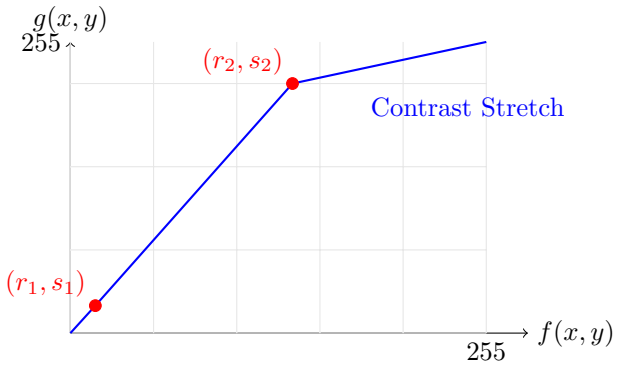
Piecewise Linear Transformation

The transformation is defined by two control points (r_1, s_1) and (r_2, s_2):

$$g(x, y) = \begin{cases} \frac{s_1}{r_1} \cdot f(x, y), & f(x, y) < r_1 \\ \frac{s_2 - s_1}{r_2 - r_1} \cdot (f(x, y) - r_1) + s_1, & r_1 \leq f(x, y) \leq r_2 \\ \frac{255 - s_2}{255 - r_2} \cdot (f(x, y) - r_2) + s_2, & f(x, y) > r_2 \end{cases}$$

Example with Given Values

For $r_1 = 9$, $s_1 = 10$, $r_2 = 80$, and $s_2 = 90$:



- Intensities below $r_1 = 9$ are compressed to $[0, s_1]$.
- Intensities between r_1 and r_2 are stretched to $[s_1, s_2]$.
- Intensities above $r_2 = 80$ are stretched to $[s_2, 255]$.
- This expands the dynamic range and increases contrast.

Intensity Level Slicing

Intensity level slicing is a technique used to highlight specific ranges of intensity values in an image while suppressing others. This method is particularly useful for emphasizing certain features or objects within an image.

Definition

The intensity level slicing operation can be defined as follows:

$$g(x, y) = \begin{cases} 255, & \text{if } L_1 \leq f(x, y) \leq L_2 \\ 0, & \text{otherwise} \end{cases}$$

where:

- $f(x, y)$ is the input pixel value.
- $g(x, y)$ is the output pixel value.
- L_1 and L_2 are the lower and upper bounds of the intensity range to be highlighted.

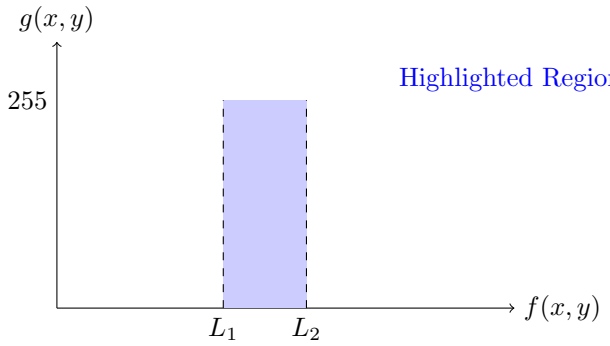
Example

For example, if we want to highlight intensity values between 100 and 150, the transformation would be:

$$g(x, y) = \begin{cases} 255, & \text{if } 100 \leq f(x, y) \leq 150 \\ 0, & \text{otherwise} \end{cases}$$

Visualization

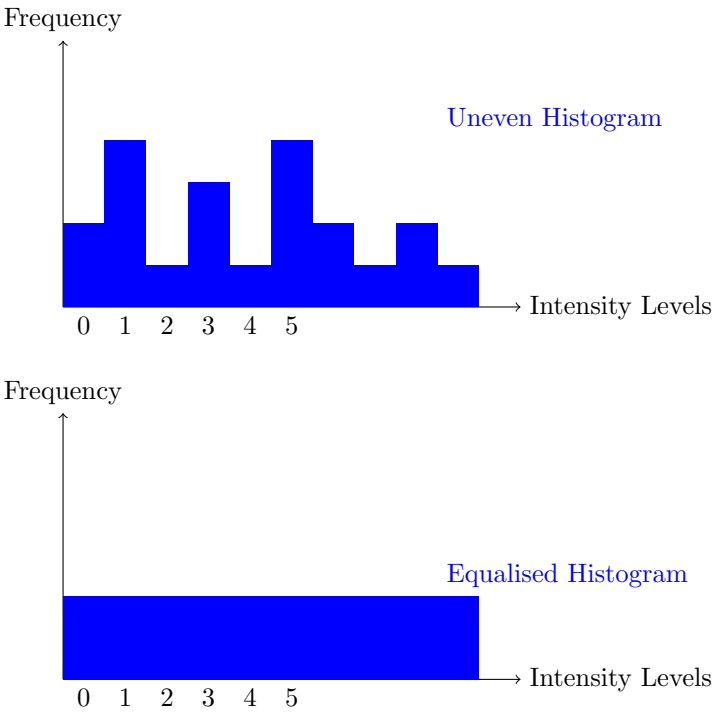
The following figure illustrates the effect of intensity level slicing on an image:



This technique effectively creates a binary image where only the specified intensity range is visible, enhancing the features of interest.

Histogram Techniques

Histogram Equalisation

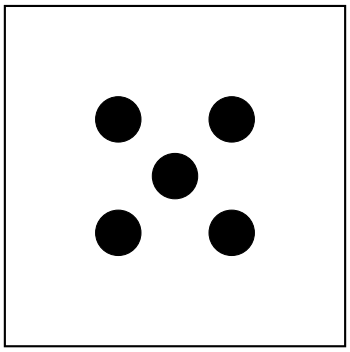


Histogram equalisation makes the histogram of an image as linear as possible.

Histogram Matching

Histogram matching transforms the histogram of the original image to a specific histogram we wish the original image to possess. This technique is useful for adjusting the brightness and contrast of an image to match a desired reference image.

Local Histogram Processing



Local Histogram Processing
Only takes into account a specific patch of the original image
and applies equations such as Mean, Median on it.

Spatial Filtering

Spatial filtering is a technique used in image processing to enhance or suppress certain features in an image. It involves the application of a filter (or kernel) to each pixel in the image, which modifies the pixel's value based on its neighbors.

Types of Spatial Filters

- Linear Filters: These filters compute the output pixel value as a weighted sum of the input pixel values in the neighborhood defined by the filter kernel. Common examples include: - Mean Filter: Averages the pixel values in the neighborhood. - Gaussian Filter: Applies a Gaussian function to weight the pixel values.
- Non-Linear Filters: These filters do not use a linear combination of the input pixel values. Examples include: - Median Filter: Replaces the pixel value with the median of the pixel values in the neighborhood, effectively reducing noise. - Maximum/Minimum Filters: Replace the pixel value with the maximum or minimum value in the neighborhood.

Example of a Mean Filter

The mean filter can be represented by the following kernel:

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

This kernel is applied to each pixel in the image, resulting in a smoothed image.

Example of a Median Filter

The median filter can be represented as follows:

$$g(x, y) = \text{median}\{f(i, j) | (i, j) \in N(x, y)\}$$

where $N(x, y)$ is the neighborhood of the pixel (x, y).

Applications of Spatial Filtering

Spatial filtering is widely used in various applications, including: - Noise reduction - Edge detection - Image sharpening - Feature enhancement

Convolution Example: 1D Array with Kernel

Consider convolving a 1D array with a kernel. This example demonstrates the step-by-step process.

Given:

- Input array: $f = [0, 0, 0, 1, 0, 0, 0, 0]$
- Kernel: $h = [8, 2, 4, 2, 1]$ (size = 5)

Convolution Formula:

$$g[n] = \sum_m f[n + m] \cdot h[m]$$

Step-by-Step Calculation (with zero-padding):

Assuming the kernel is centered and we pad the input with zeros:

Step	Kernel Position	Multiplication	Result
1	$[0, 0, 0, 0, 0]$	$0 \cdot 8 + 0 \cdot 2 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$	0
2	$[0, 0, 0, 0, 1]$	$0 \cdot 8 + 0 \cdot 2 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$	1
3	$[0, 0, 0, 1, 0]$	$0 \cdot 8 + 0 \cdot 2 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1$	2
4	$[0, 0, 1, 0, 0]$	$0 \cdot 8 + 0 \cdot 2 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$	4
5	$[0, 1, 0, 0, 0]$	$0 \cdot 8 + 1 \cdot 2 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$	2
6	$[1, 0, 0, 0, 0]$	$1 \cdot 8 + 0 \cdot 2 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$	8
7	$[0, 0, 0, 0, 0]$	$0 \cdot 8 + 0 \cdot 2 + 0 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$	0

Output: $g = [0, 1, 2, 4, 2, 8, 0]$

The convolution operation slides the kernel across the input array, computing the dot product at each position to produce the filtered result.

Convolution vs. Correlation

Both convolution and correlation are fundamental operations in image processing, but they differ in a key aspect:

Correlation:

$$g[n] = \sum_m f[n + m] \cdot h[m]$$

Correlation computes the similarity between a signal and a kernel by sliding the kernel across the signal and computing the dot product at each position without any reversal of the kernel.

Convolution:

$$g[n] = \sum_m f[n + m] \cdot h[-m]$$

Convolution is similar to correlation but involves **flipping (reversing) the kernel** before sliding it across the signal. This reversal is represented by $h[-m]$ instead of $h[m]$.

Key Differences:

- **Kernel Orientation:** Convolution flips the kernel both horizontally and vertically; correlation does not.
- **Mathematical Property:** Convolution is commutative ($f * h = h * f$), while correlation is not necessarily commutative.
- **Applications:** Convolution is commonly used for filtering and feature extraction in image processing. Correlation is used for template matching and measuring similarity.
- **Implementation:** In many image processing libraries, the "convolution" operation is actually implemented as correlation for efficiency, with the kernel pre-flipped if true convolution is required.

Example: For the input array $f = [0, 0, 0, 1, 0, 0, 0, 0]$ and kernel $h = [8, 2, 4, 2, 1]$:

- **Correlation** uses the kernel as-is: $[8, 2, 4, 2, 1]$
- **Convolution** flips the kernel to: $[1, 2, 4, 2, 8]$ before applying the operation.

This distinction becomes particularly important when working with asymmetric kernels or when the order of operations matters for maintaining mathematical properties.

Smoothing Filters

Smoothing filters are used to reduce noise and blur details in an image by averaging pixel values within a neighborhood. They are commonly applied as a preprocessing step before further image analysis or feature extraction.

Box Filter

The box filter, also known as the mean filter or averaging filter, is one of the simplest smoothing filters. Each output pixel is computed as the average of all pixels in a neighborhood defined by the filter kernel.

Box Filter Kernel

For a 3×3 box filter:

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

For a 5×5 box filter:

$$H = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Properties

- Simple and computationally efficient.
- Reduces high-frequency noise effectively.
- Can blur edges and fine details.
- Equal weight is given to all pixels in the neighborhood.

Gaussian Filter

The Gaussian filter is a more sophisticated smoothing filter that applies a Gaussian function (bell curve) to weight pixel values. Pixels closer to the center receive higher weights, resulting in more natural-looking blur compared to the box filter.

Gaussian Filter Kernel

The Gaussian function is defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

where σ is the standard deviation controlling the spread of the Gaussian.

Example: 3×3 Gaussian Filter

For $\sigma = 1$:

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

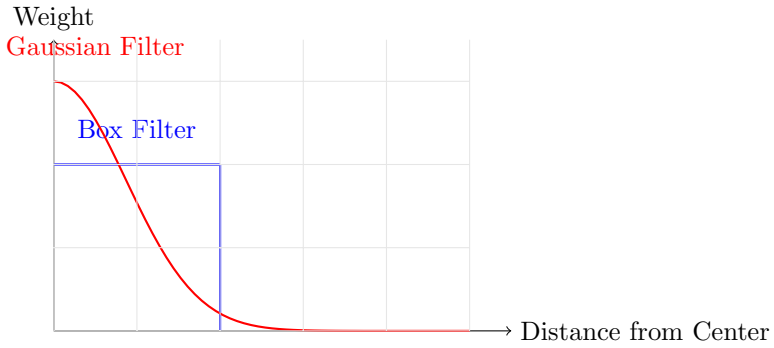
For a 5×5 Gaussian filter with $\sigma = 1$:

$$H = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Properties

- More natural smoothing than box filter due to Gaussian weighting.
- Separable kernel: can be applied as two 1D convolutions (more efficient).
- Controlled by standard deviation σ: larger σ produces stronger blur.
- Preserves edges better than box filter.
- Widely used in computer vision and image processing.

Comparison: Box vs. Gaussian



The Gaussian filter applies exponentially decreasing weights from the center, while the box filter uses uniform weights.

Guideline for Selecting Gaussian Filter Kernel Size

When selecting the kernel size for Gaussian filters, a common guideline is to use:

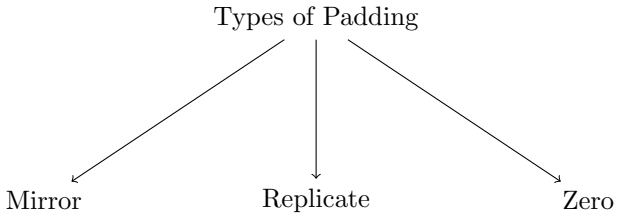
$$\text{Kernel Size} = \lceil 6\sigma \rceil$$

For example, if we have $\sigma = 7$:

$$6 \times 7 = 42$$

In this case, we would choose a kernel size of 43 to maintain an odd preference. However, it is important to note that there is no benefit in using a kernel size greater than $\lceil 6\sigma \rceil$.

Types of Padding



Median Filter Example

The median filter is a non-linear filter used to reduce noise in an image while preserving edges. It works by replacing each pixel value with the median value of the pixels in its neighborhood.

For a 3×3 matrix with the following values:

$$\begin{bmatrix} 4 & 100 & 150 \\ 25 & 78 & 16 \\ 4 & 3 & 160 \end{bmatrix}$$

To apply the median filter, we consider the values in the 3×3 neighborhood of each pixel. For the center pixel (78), the neighborhood values are:

$$\{4, 100, 150, 25, 78, 16, 4, 3, 160\}$$

Sorting these values gives:

$$\{3, 4, 4, 16, 25, 78, 100, 150, 160\}$$

The median value is the middle value in the sorted list, which is 25. Therefore, the center pixel (78) is replaced with 25.

This process is repeated for each pixel in the image, resulting in a new filtered image that reduces noise while maintaining important features.

Sharpening Filters

Sharpening filters are used to enhance edges and fine details in an image by amplifying high-frequency components. Unlike smoothing filters that blur an image, sharpening filters make transitions between pixel values more pronounced.

Blurring vs. Sharpening: Integration and Differentiation

There is a fundamental mathematical relationship between blurring and sharpening operations:

- **Blurring** corresponds to **integration**. Smoothing filters average neighboring pixel values, which is analogous to integrating a signal. This operation reduces high-frequency components and creates a smoother, more averaged representation of the image.
- **Sharpening** corresponds to **differentiation**. Sharpening filters compute differences between neighboring pixel values, which is analogous to differentiating a signal. This operation emphasizes edges and rapid intensity changes, enhancing high-frequency components.

This relationship reflects the inverse nature of these operations:

$$\text{Integration} \leftrightarrow \text{Blurring (Low-pass filtering)}$$

$$\text{Differentiation} \leftrightarrow \text{Sharpening (High-pass filtering)}$$

In practical terms, blurring smooths transitions by computing weighted averages, while sharpening highlights transitions by computing local derivatives or differences. Both operations are essential in image processing for different analytical and enhancement purposes.

First Order Derivatives

First-order derivatives are used to detect edges in images by computing the rate of change of pixel intensity. The most common first-order derivative operators are the Sobel and Prewitt operators.

Sobel Operator

The Sobel operator computes the gradient (first derivative) in both horizontal and vertical directions using two 3×3 kernels:

Horizontal Sobel Kernel (S_x):

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

Vertical Sobel Kernel (S_y):

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

The gradient magnitude is computed as:

$$G = \sqrt{S_x^2 + S_y^2}$$

Example Calculation

Consider the following 3×3 image patch:

$$f = \begin{bmatrix} 10 & 20 & 30 \\ 15 & 25 & 35 \\ 20 & 30 & 40 \end{bmatrix}$$

Applying S_x (Horizontal Gradient):

$$S_x \text{ result} = (-1)(10) + (0)(20) + (1)(30) + (-2)(15) + (0)(25) + (2)(35) + (-1)(20) + (0)(30) + (1)(40)$$

$$= -10 + 0 + 30 - 30 + 0 + 70 - 20 + 0 + 40 = 80$$

Applying S_y (Vertical Gradient):

$$S_y \text{ result} = (-1)(10) + (-2)(20) + (-1)(30) + (0)(15) + (0)(25) + (0)(35) + (1)(20) + (2)(30) + (1)(40)$$

$$= -10 - 40 - 30 + 0 + 0 + 0 + 20 + 60 + 40 = 40$$

Gradient Magnitude:

$$G = \sqrt{80^2 + 40^2} = \sqrt{6400 + 1600} = \sqrt{8000} \approx 89.44$$

The gradient magnitude indicates the strength of the edge at this location. Higher values represent stronger edges.

Second Order Derivatives: Laplacian

The Laplacian operator is a second-order derivative operator used for edge detection and image enhancement. It computes the sum of the second derivatives in both x and y directions and is particularly effective at detecting rapid intensity changes and edges.

Laplacian Definition

The Laplacian operator is defined as:

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

Laplacian Kernels

The Laplacian can be approximated using discrete 3×3 kernels. The most common forms are:

Standard Laplacian Kernel:

$$L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\nabla^2 f(x, y) = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4\, f(x, y)$$

Extended Laplacian Kernel (including diagonals):

$$L = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Example Calculation

Consider the following 3×3 image patch:

$$f = \begin{bmatrix} 10 & 20 & 30 \\ 15 & 25 & 35 \\ 20 & 30 & 40 \end{bmatrix}$$

Applying the Standard Laplacian Kernel:

$$L \text{ result} = (0)(10) + (1)(20) + (0)(30) + (1)(15) + (-4)(25) + (1)(35) + (0)(20) + (1)(30) + (0)(40)$$

$$= 0 + 20 + 0 + 15 - 100 + 35 + 0 + 30 + 0 = 0$$

Properties of the Laplacian

- **Isotropic:** The Laplacian is rotation-invariant, meaning it responds equally to edges in all directions.
- **Edge Detection:** Produces zero values in uniform regions and non-zero values at edges.
- **Sensitive to Noise:** Being a second-order derivative, the Laplacian is more sensitive to noise than first-order operators.
- **Zero Crossing:** Edges are often located at zero-crossings of the Laplacian (transitions from positive to negative or vice versa).
- **Double Edges:** Can produce double edges around actual edge locations due to the second derivative nature.

Comparison: First Order vs. Second Order

Property	First Order (Sobel)	Second Order (Laplacian)
Derivative Type	First derivative (gradient)	Second derivative
Edge Detection	Detects edge location and direction	Detects edge transitions (zero-crossings)
Noise Sensitivity	Less sensitive	More sensitive
Computational Cost	Moderate	Low
Output Type	Gradient magnitude and direction	Scalar value (zero-crossing)

Roberts Operator

The Roberts operator is another first-order derivative operator used for edge detection. It uses two 2×2 kernels to compute gradients in diagonal directions.

Roberts Kernel 1 (Diagonal):

$$R_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Roberts Kernel 2 (Anti-Diagonal):

$$R_2 = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

The gradient magnitude is computed as:

$$G = \sqrt{R_1^2 + R_2^2}$$

Comparison: Sobel vs. Roberts

Property	Sobel	Roberts
Kernel Size	3×3	2×2
Computational Cost	Moderate	Low
Edge Detection	Strong edges	Thin, precise edges
Noise Sensitivity	Moderate	Higher
Smoothing Effect	Slight (averaging)	None

Unsharp Masking and Highboost Filtering

Unsharp masking and highboost filtering are sharpening techniques that enhance edges and fine details by combining the original image with a blurred (unsharp) version of itself.

Unsharp Masking

Unsharp masking works by subtracting a blurred version of the image from the original image to create a high-pass filtered result. This technique is based on the principle that edges and fine details are preserved in the original image but removed in the blurred version.

Formula:

$$g(x, y) = f(x, y) - \text{blur}[f(x, y)]$$

where $\text{blur}[f(x, y)]$ is the blurred version of the image.

Alternatively, the result can be added back to the original to create a sharpened image:

$$g(x, y) = f(x, y) + k \cdot [f(x, y) - \text{blur}[f(x, y)]]$$

where k is a scaling factor controlling the amount of sharpening. $k > 1$ results in highboost filtering while $k = 1$ results in standard unsharp masking.

Summary Table of Image Enhancement Techniques

Domain	Technique	Primary Goal / Effect	Key Mechanism
Intensity Transformations (Point Operations)			
Intensity (Point Ops)	Image Negative	Reverses colors to enhance **dark area details**.	$s = (L-1) - r$
Intensity (Point Ops)	Log Transformation	**Expands dark pixels**, compresses light pixels.	Uses a logarithmic function.
Intensity (Point Ops)	Power-Law (γ Transform)	Controls overall brightness/contrast based on γ value.	$s = cr^{\gamma}$
Intensity (Point Ops)	Contrast Stretching	Expands the intensity range to use the **full scale**.	Piecewise linear function.
Histogram Processing			
Histogram Processing	Histogram Equalization (Global)	Enhances image contrast by aiming for a **uniform histogram**.	Discrete Cumulative Distribution Function (CDF).
Histogram Processing	Histogram Matching (Specification)	Changes the image's histogram to match a **specific desired shape**.	Equating two equalized variables.
Histogram Processing	Local Histogram Processing	Enhances **details over small areas**.	Applied globally within a moving local neighborhood.
Spatial Filtering (Neighborhood Operations)			
Smoothing (Lowpass Filters)	Box Filter (Averaging)	**Blurs** the image to reduce noise.	Replaces pixel value with the average of its neighbors.
Smoothing (Lowpass Filters)	Median Filter	Excellent reduction of **salt-and-pepper noise**.	Replaces pixel value with the **median** of its neighbors.
Sharpening (Highpass Filters)	Laplacian (2nd Derivative)	Enhances **fine detail** and sharp transitions.	Uses second derivative convolution kernel.
Sharpening (Highpass Filters)	Gradient (Roberts/Sobel)	Highlights **prominent edges**.	Uses first derivative kernels (vector magnitude).
Sharpening (Highpass Filters)	Unsharp Masking / Highboost	Sharpens image details.	Sharpened = Original + k(Original − Blurred).

Image Condition/Problem	**Recommended Techniques**	**Primary Goal & Mechanism**
Contrast and Intensity Adjustment		
Overall Low Contrast (Clustered histogram)	**Global Histogram Equalization**	Enhances **overall contrast** by forcing the histogram towards a uniform distribution.
Low Contrast + Washed-out/Light Image	**Contrast Stretching**	Expands the intensity range to span the **ideal full intensity range** using piecewise linear transformation.
Low Contrast with Specific Brightness Need	**Histogram Matching (Specification)**	Transforms the histogram to match a **specific desired shape** by equating two equalized transformation variables.
Very Dark Image (Loss of shadow detail)	**Log Transformation** or **Power-Law** ($\gamma < 1$)	**Expands dark pixels** using a non-linear curve that rises steeply at low input values.
White/Gray Details Embedded in Dark Background	**Image Negatives**	**Reverses** the intensity levels using the linear inversion formula: $g(x, y) = (L - 1) - f(x, y)$.
Fine Details Hidden in Dark Areas (Local problem)	**Local Histogram Processing**	Enhances **details over small areas** by applying histogram methods within a moving neighborhood.
Noise Reduction, Smoothing, and Sharpening		
General Noise Reduction (Smoothing)	**Box Filter** or **Gaussian Filter**	Reduces **sharp transitions in intensity** (noise) using linear spatial filtering (convolution).
Salt-and-Pepper Noise (Impulse Noise)	**Median Filter** (Non-linear)	Achieves very good **noise reduction** by replacing the center pixel with the **median** of its neighborhood.
Image is Too Smooth/Blurry (Needs Detail Recovery)	**Sharpening Filters**	Highlights intensity transitions (edges) by computing the **digital differentiation**.
Enhancing Fine Detail (Texture)	**Laplacian** (Second Derivative)	Enhances **fine detail much better** than the first derivative by adding the calculated Laplacian image to the original.
Enhancing Prominent Edges (Object Outlines)	**Gradient** (First Derivative, e.g., Sobel)	Provides a **stronger response to significant intensity transitions** using first-order derivative operators.
Controlled Sharpening/Edge Boost	**Unsharp Masking** or **Highboost Filtering**	Sharpens image details using the difference formula: Sharpened = Original + k(Original − Blurred) with $k \geq 1$.

Mathematica Image Processing Code Reference

Below is a summary of key Mathematica code snippets for image processing, covering spatial filtering, intensity transformations, histogram techniques, noise reduction, and image resizing.

Spatial Filtering and Sharpening

```
// Unsharp Masking and Highboost Filtering
blurfactor = GaussianMatrix[{8,2.6}];
originalImage = ImageData[i7];
blurredImage = ImageConvolve[i7, blurfactor];
mask = originalImage - ImageData[blurredImage];
Image[originalImage + 4.5 * maskNumeric]
Manipulate[Image[originalImage + k * maskNumeric], {k, 1, 100}];

// Sobel Operators (First Order Derivatives)
s1 = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};
s2 = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
i8x = ImageConvolve[i8, s1];
i8y = ImageConvolve[i8, s2];
Ax = ImageData[i8x];
Ay = ImageData[i8y];
AG = Rescale[Sqrt[Ax^2 + Ay^2]];
Image[AG]

// Laplacian Transform (Second Order Derivatives)
Grid[{{i6, ImageDifference[i6, LaplacianFilter[i6, 10]]}}, Frame -> All]

// Median Filter
Manipulate[MedianFilter[i5, r], {r, 1, 10, 1}]
```

Image Padding Effects

```
ImageConvolve[i1, GaussianFilter[1, 1], Padding -> Reflected];
v1 = ImageConvolve[, GaussianMatrix[{21, 7}], Padding -> "Reversed"];
v2 = ImageConvolve[, GaussianMatrix[{21, 7}], Padding -> "Reflected"];
v3 = ImageConvolve[, GaussianMatrix[{21, 7}], Padding -> "Periodic"];
v4 = ImageConvolve[, GaussianMatrix[{21, 7}], Padding -> "Fixed"];
v5 = ImageConvolve[, GaussianMatrix[{21, 7}], Padding -> 0];
Grid[{{"Reversed", "Reflected", "Periodic", "Fixed", "0"}, {v1, v2, v3, v4, v5}}, Frame -> All]
```

Low Pass Filtering and Thresholding

```
lowPassFiltered = ImageConvolve[i3, GaussianMatrix[{16, 5.5}]]
Binarize[lowPassFiltered, 0.2]
```

Image Resizing

```
i1Dat = ImageData[i1];
reducedI1 = i1Dat[[2 ;; ;; 2, 2 ;; ;; 2]];
Image[reducedI1]
ImageConvolve[Image@reducedI1, GaussianMatrix[{21, 7}], Padding -> "Periodic"]
ImageConvolve[i1, GaussianMatrix[{21, 7}], Padding -> "Periodic"]
```

Box and Gaussian Smoothing

```
three = BoxMatrix[1]
eleven = BoxMatrix[5];
twentyone = BoxMatrix[10];
```

```
ImageConvolve[i1, three / 9]
ImageConvolve[i1, eleven / 121]
ImageConvolve[i1, twentyone / (21 * 21)]
ImageConvolve[i1, GaussianMatrix[{10, 3.5}]]
ImageConvolve[i1, GaussianMatrix[{21, 7}]]
```

Histogram Processing

```
ImageHistogram[dice]
ImageFilter[If[Mean[Flatten[#]] > 0.12
    && Mean[Flatten[#]] < 0.22, 6 * #[[3, 3]], #[[3, 3]]] &, dice, 2]
HistogramTransform[i7, i8]
Grid[{{ImageHistogram[i7]}, {ImageHistogram[i9]}}]
HistogramTransform[i2]
ImageHistogram[i2t, "Byte", FrameTicks -> Automatic]
```

Intensity Level Slicing and Contrast Stretching

```
f[x_] := Which[x < 50, 10, 120 >= x >= 50, 100, x > 120, 10];
sliced = Map[f, imageData, {2}];
Image[sliced, "Byte"]
f2[x_] := Which[x < 50, x, 120 >= x >= 50, 200, x > 120, x];
sliced2 = Map[f2, imageData, {2}];
Image[sliced2, "Byte"]

fstretch[x_] := Which[x <= 9, 0, x >= 81, 255, True, (255 / 72) * x - 255 / 8];
stretched = Map[f2, i3Data, {2}]
Image[stretched, "Byte"]
fstretch2[x_] := Which[x < N[L / 4], 0.5 * x, x > N[3 * L / 4],
     x * 0.5 + 0.5 * L, True, -0.25 * L + 1.5 * x];
stretched2 = Map[fstretch2, i3Data, {2}];
Image[stretched2, "Byte"]
fstretch3 = Map[If[# < 9, 0, ((L - 1) * # / (81 - 9)) - (L - 1) * 9 / (81 - 9)] &
    ,i3Data, {2}];
Image[fstretch3, "Byte"]
```

Log Transformation and Negation

```
f[x_] := (255 / Log[255]) * Log[1 + x];
logTransformed = Map[f, imgData, {2}];
Image[logTransformed, "Byte"]

ColorNegate[i1]
i1DatFlipped = 1 - i1Dat
Image[i1DatFlipped]
```

Thresholding and Histogram Plotting

```
threshold[x_] := Which[x > 0.5, 0, True, 1];
Manipulate[Image@Map[Which[# > k, 1, True, 0] &, iDat, {2}], {k, 0, 1, 0.05}]
i2Dat = Counts@Flatten@ImageData[i, "Byte"]
ListPlot[i2Dat, FrameTicks -> Automatic, Filling -> Axis]
```

Gaussian Noise and Averaging

```
noisy = nebulaData + RandomVariate[NormalDistribution[0, 64], Dimensions[nebulaData]];
Image[noisy, "Byte"]
rescaledNebulaNoisy = 255 * (noisy - Min[noisy]) / (Max[noisy] - Min[noisy]);
Image[rescaledNebulaNoisy, "Byte"]
```

```
AddNoise[A0_] := Module[{A = A0, B},
    B = A + RandomVariate[NormalDistribution[0, 64], Dimensions[A]];
    B = 255 * (B - Min[B]) / (Max[B] - Min[B])
];
BList = Table[AddNoise[nebulaData], {i, 20}];
BListMean = Mean[BList];
BListMeanRescaled = Rescale[BListMean, MinMax@BListMean, {0, 255}];
Image[BListMeanRescaled, "Byte"]
```

Image Dimension Reduction

```
imageDatX = ImageData[imgX];
Image[imageDatX[[2;;;;2, 2;; ;;2]]]
Manipulate[Image[imageDatX[[k;;;;k, k;; ;;k]]], {k, 1, 20, 1}]
imgDataY = ImageData@Image[imageDatX[[20;;;;20, 20;; ;;20]]];
Image[Rescale[imgDataY, MinMax@imgDataY, {0, 255}], "Byte"]
```

Note: You can perform subtraction, addition, and division directly on image data arrays for custom processing.

Fourier Series Analysis on Images

The Fourier transform is a fundamental tool in image processing that converts an image from the spatial domain to the frequency domain. This transformation enables analysis and manipulation of image frequencies, which is essential for filtering, compression, and feature extraction.

Spatial Domain vs. Frequency Domain

The Discrete Fourier Transform (DFT) converts spatial domain information into frequency domain representation, revealing the frequency components that compose the image. The Inverse DFT reverses this process to recover the spatial domain image.

Steps for Frequency Domain Filtering

The following procedure describes the complete process of converting an image from the spatial domain to the frequency domain, applying filtering, and converting back:

1. **Pad the Input Image:** Given an input image $f(x, y)$ of size $M \times N$, pad the image to size $P \times Q$, where:
$$P = 2M, \quad Q = 2N$$
This zero-padding prevents circular convolution artifacts and improves computational efficiency.

2. **Multiply by Phase Factor:** Multiply the padded image by $(-1)^{x+y}$ to centre the DFT of $f(x, y)$:
$$f_{\text{padded}}(x, y) \cdot (-1)^{x+y}$$
This shifts the zero-frequency component to the center of the frequency domain.

3. **Compute the DFT:** Calculate the Discrete Fourier Transform to obtain $F(u, v)$:
$$F(u, v) = \sum_{x=0}^{P-1} \sum_{y=0}^{Q-1} f_{\text{padded}}(x, y) \cdot e^{-j2\pi(ux/P + vy/Q)}$$

4. **Construct Filter Transfer Function:** Create a real, symmetric filter transfer function $H(u, v)$ of size $P \times Q$ with centre at $(P/2, Q/2)$. The filter design depends on the desired frequency response (lowpass, highpass, bandpass, etc.).

5. **Apply Filter in Frequency Domain:** Form the product $G(u, v)$ by multiplying the filter and DFT elementwise:
$$G(u, v) = H(u, v) \cdot F(u, v)$$

6. **Compute Inverse DFT:** Calculate the Inverse Discrete Fourier Transform and multiply by $(-1)^{x+y}$ to shift the zero-frequency component back:
$$g_{\text{filtered}}(x, y) = \text{IDFT}[G(u, v)] \cdot (-1)^{x+y}$$

7. **Extract Result:** The final filtered image is obtained by extracting the $M \times N$ region from the top left quadrant of $g_{\text{filtered}}(x, y)$.

Key Considerations

- **Zero-Padding:** Padding prevents aliasing and circular convolution effects that would occur without padding.
- **Phase Centering:** The $(-1)^{x+y}$ multiplication ensures that the zero-frequency (DC) component is at the center of the frequency domain, making visualization and filter design more intuitive.
- **Filter Design:** The transfer function $H(u, v)$ is customized based on the filtering objective (noise reduction, edge enhancement, etc.).
- **Computational Efficiency:** Using FFT (Fast Fourier Transform) significantly reduces computation time compared to direct DFT calculation.

Summary: Centering, DFT, and the 4x4 Example

Here is the complete summary of our journey, tying together the conceptual tricks, the specific Mathematica settings, and the raw math using your concrete 4×4 example.

1. The Concept: Why We Center the Data

The Problem: By default, the standard DFT places the zero-frequency (DC) term—which represents the total energy/brightness of the image—at the origin $(0, 0)$ (top-left corner). This splits the "low frequency" details across the four corners, making it impossible to apply a simple circular filter in the middle.

The Fix: Multiply the input image by a checkerboard pattern $(-1)^{x+y}$.

The Effect: This modulation uses the "Shift Property" of the Fourier Transform, shifting the frequency spectrum by half the image size $(M/2, N/2)$, moving the DC spike from the corner to the geometric center.

2. The Concrete Example (Non-Uniform 4×4 Matrix)

We used this 4×4 matrix to illustrate the process:

$$A = \begin{bmatrix} 1 & 2 & 9 & 7 \\ 99 & 88 & 66 & 22 \\ 55 & 1000 & 999 & 9998 \\ 44 & 22 & 5555 & 199 \end{bmatrix}$$

The Summation Expansion (How We Got 18,166):
The DC component (all exponents become 1):

$$F(0,0) = \sum_{x=1}^{4} \sum_{y=1}^{4} A_{x,y}$$

Row sums:

$$\begin{aligned} &\text{Row 1:} \quad 1 + 2 + 9 + 7 = 19 \\ &\text{Row 2:} \quad 99 + 88 + 66 + 22 = 275 \\ &\text{Row 3:} \quad 55 + 1000 + 999 + 9998 = 12052 \\ &\text{Row 4:} \quad 44 + 22 + 5555 + 199 = 5820 \end{aligned}$$

Total sum: $19 + 275 + 12052 + 5820 = \mathbf{18,166}$.

Resulting Matrices:
Uncentered DFT (Standard): Energy at $(0, 0)$.

$$\begin{bmatrix} \mathbf{18166} & 11153 & 4510 & 11153 \\ 13249 & 14553 & 11279 & 3644 \\ 5976 & 9977 & 15376 & 9977 \\ 13249 & 3644 & 11279 & 14553 \end{bmatrix}$$

Centered DFT (After Trick): Energy at center $(2, 2)$.

$$\begin{bmatrix} 15376 & 9977 & 5976 & 9977 \\ 11279 & 14553 & 13249 & 3644 \\ 4510 & 11153 & \mathbf{18166} & 11153 \\ 11279 & 3644 & 13249 & 14553 \end{bmatrix}$$

3. The Parameters: `FourierParameters -> {1, -1}`

- $a = 1$ (Amplitude): Scaling factor is $1/n^{(1-1)/2} = 1$, so the result is the raw sum (not divided by 16).
- $b = -1$ (Phase): Exponent is $-2\pi i$, the standard forward transform.

4. The Final Equation

Input: Use the centered image $B1$ where $B1_{x,y} = A_{x,y} \cdot (-1)^{x+y}$.

Base Formula: The unscaled DFT is $\sum \text{Data} \cdot e^{-2\pi i \cdots}$.

Indexing: Mathematica counts from 1 to N (not 0 to $N-1$), so subtract 1 from every coordinate in the exponent.

Final Equation:

$$BF_{u,v} = \sum_{x=1}^{M} \sum_{y=1}^{N} B1_{x,y}\, e^{-2\pi i \left(\frac{(u-1)(x-1)}{M} + \frac{(v-1)(y-1)}{N} \right)}$$

This equation takes every pixel from your centered image $B1$, compares it against a wave frequency defined by (u, v), sums up the results, and stores it in the output spectrum BF.

Mathematica Implementation Example (Lecture 4.2–4.5)

Below is a step-by-step Mathematica implementation for frequency domain filtering (ideal low pass filter), including spectrum centering, zero-padding, filtering, and reconstruction:

```
(* Load and convert image to grayscale byte matrix *)
i = Import["ExampleData/lena.tif"];
A = ImageData[i, "Byte"];
{Md, Nd} = Dimensions[A];

(* Center the spectrum *)
CM = Table[(-1)^(i + j), {i, 1, Md}, {j, 1, Nd}];
B = A * CM;

(* Zero-pad to double size *)
B1 = ArrayPad[B, {{0, Md}, {0, Nd}}];
{P, Q} = Dimensions[B1];

(* Compute the centered Fourier transform *)
BF = Fourier[B1, FourierParameters -> {1, -1}];

(* Construct ideal low pass filter *)
dis[u_, v_] := Sqrt[(u - Md)^2 + (v - Nd)^2];
D0 = 20;
H = Table[If[dis[u, v] <= D0, 1, 0], {u, 1, P}, {v, 1, Q}];

(* Apply filter in frequency domain *)
G = BF * H;

(* Inverse Fourier transform and take real part *)
R = Re[InverseFourier[G, FourierParameters -> {1, -1}]];

(* Extract top-left quadrant *)
R1 = Take[R, {1, Md}, {1, Nd}];

(* Reverse centering *)
R2 = R1 * CM;

(* Display result *)
Image[R2, "Byte"]
```

This code demonstrates the full process: centering, padding, filtering, and reconstructing the filtered image in Mathematica.

Low Pass Filters

Low pass filters are used to remove high-frequency components from an image while preserving low-frequency information. This results in image smoothing and noise reduction. In the frequency domain, low pass filters attenuate frequencies above a cutoff frequency while allowing lower frequencies to pass through.

Ideal Low Pass Filter

The ideal low pass filter is the simplest type of frequency domain filter. It completely removes all frequency components beyond a specified cutoff distance D_0 from the origin in the frequency domain.

Transfer Function

The transfer function for an ideal low pass filter is defined as:

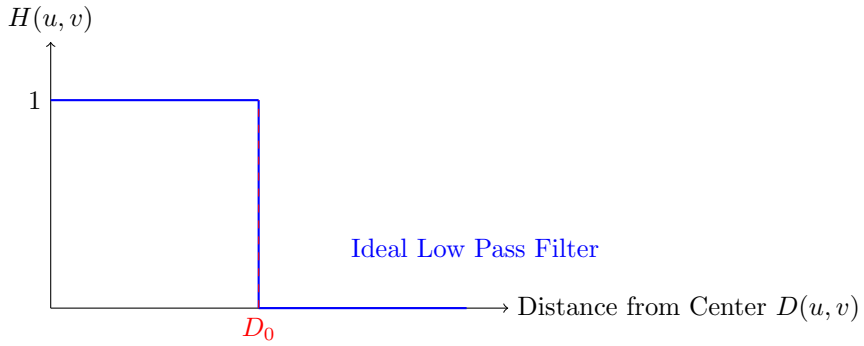
$$H(u,v) = \begin{cases} 1, & \text{if } D(u,v) \leq D_0 \\ 0, & \text{if } D(u,v) > D_0 \end{cases}$$

where $D(u,v) = \sqrt{(u - P/2)^2 + (v - Q/2)^2}$ is the distance from the center of the frequency domain, and D_0 is the cutoff distance.

Characteristics

- **Sharp Cutoff:** The ideal filter has a sharp transition at the cutoff frequency D_0.
- **Ringing Artifacts:** Due to the sharp frequency cutoff, ideal low pass filters produce ringing artifacts (also called Gibbs phenomenon) at edges in the filtered image.
- **Simple Implementation:** Easy to implement and understand.
- **Not Practical:** Rarely used in practice due to the ringing artifacts it introduces.

Visualization



Gaussian Low Pass Filter

The Gaussian low pass filter provides a smooth transition between frequencies that are passed and frequencies that are attenuated. It avoids the sharp cutoff of the ideal filter, resulting in significantly fewer ringing artifacts.

Transfer Function

The transfer function for a Gaussian low pass filter is defined as:

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

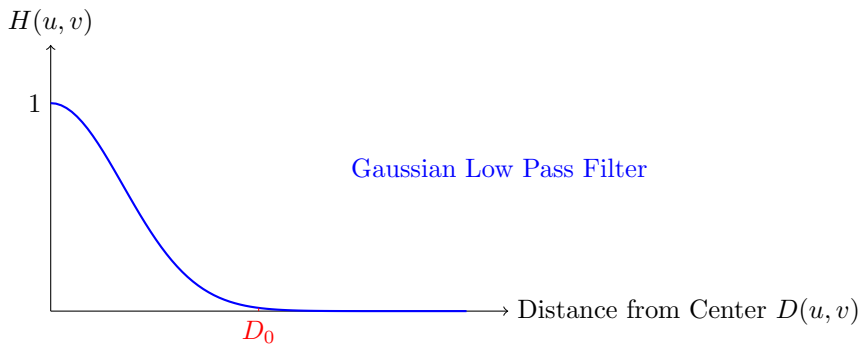
where σ controls the spread of the Gaussian function. Alternatively, using cutoff frequency D_0:

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

Characteristics

- **Smooth Transition:** Provides a gradual transition from pass-band to stop-band.
- **No Ringing:** Eliminates the ringing artifacts associated with ideal filters.
- **Natural Smoothing:** Produces more natural-looking smoothed images.
- **Commonly Used:** Widely used in practical applications due to its smooth frequency response.

Visualization



Butterworth Low Pass Filter

The Butterworth low pass filter provides a compromise between the ideal and Gaussian filters. It has a tunable parameter n (order) that controls the sharpness of the cutoff, allowing adjustment between smoothness and transition sharpness.

Transfer Function

The transfer function for a Butterworth low pass filter of order n is defined as:

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

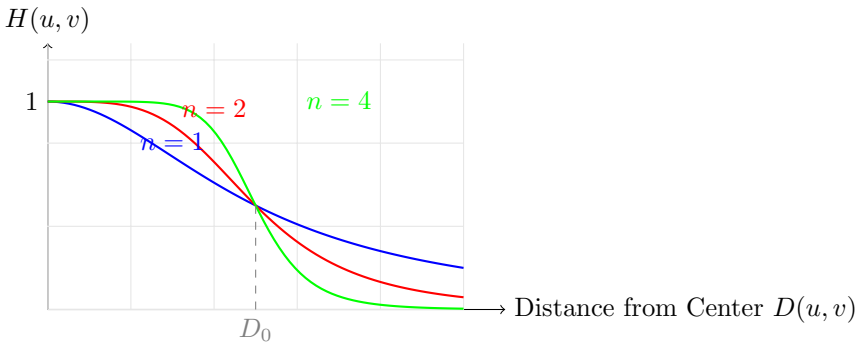
where:

- D_0 is the cutoff frequency (distance at which $H = 0.5$).
- n is the order of the filter, controlling the sharpness of the cutoff.

Characteristics

- **Adjustable Sharpness:** Order n controls the transition sharpness:
  - $n = 1$: Smooth, similar to Gaussian.
  - $n = 2, 3, \ldots$: Progressively sharper cutoff.
  - $n \to \infty$: Approaches ideal filter behavior.
- **Minimal Ringing:** Much less ringing than ideal filter but sharper than Gaussian.
- **Versatile:** Provides flexibility to balance between smoothness and cutoff sharpness.
- **Maximally Flat:** In the pass-band, the frequency response is maximally flat (no ripple).

Visualization



Mathematica Implementation

```
(* Butterworth Low Pass Filter Example *)
BLPF[img_,D0_,order_]:=Module[{A, Md, Nd, CM, B, B1,P,Q,
BF,filterButterworth,filteredDFT,inverseDFT,phaseShift,filteredShifted,resultImg },
A=ImageData[ColorConvert[img,"Grayscale"],"Byte"];{Md,Nd}=Dimensions[A];
CM=Table[(-1)^(i+j),{i,1,Md},{j,1,Nd}];B=A*CM;
B1=ArrayPad[B,{{0,Md},{0,Nd}}];{P,Q}=Dimensions[B1];
BF=Fourier[B1,FourierParameters->{1,-1}];
filterButterworth=Table[1/(1+(Sqrt[(u-P/2)^2+(v-Q/2)^2]/D0)^(2*order)),{u,0,P-1},{v,0,Q-1}];
filteredDFT=BF*filterButterworth;
inverseDFT=InverseFourier[filteredDFT,FourierParameters->{1, -1}];
phaseShift=Table[(-1)^(x+y),{x,0,P-1},{y,0,Q-1}];
filteredShifted=inverseDFT*phaseShift;
resultImg=Take[Re[filteredShifted],{1,Md},{1,Nd}];
Image[resultImg,"Byte"]
]

i = Import["https://boofcv.org/images/thumb/6/66/Kodim17_noisy.jpg/300px-Kodim17_noisy.jpg"]

Manipulate[BLPF[i,cutoff,order],{cutoff,1,255,1},{order,1,4}]
```

Comparison: Low Pass Filters

Property	Ideal	Gaussian	Butterworth
Cutoff Sharpness	Very sharp	Smooth	Adjustable (via n)
Ringing Artifacts	Severe	None	Minimal
Transition Smoothness	Abrupt	Gradual	Tunable
Practical Use	Rare	Very common	Common
Parameter Control	D_0 only	D_0 only	D_0 and n
Computational Cost	Low	Low	Low
Best For	Theoretical analysis	General smoothing	Balanced applications

High Pass Filters

High pass filters are used to enhance high-frequency components in an image while attenuating low-frequency information. This results in edge enhancement and detail sharpening. In the frequency domain, high pass filters attenuate frequencies below a cutoff frequency while allowing higher frequencies to pass through.

Ideal High Pass Filter

The ideal high pass filter is the complementary operation to the ideal low pass filter. It completely removes all frequency components within a specified cutoff distance D_0 from the origin while preserving all components beyond this distance.

Transfer Function

The transfer function for an ideal high pass filter is defined as:

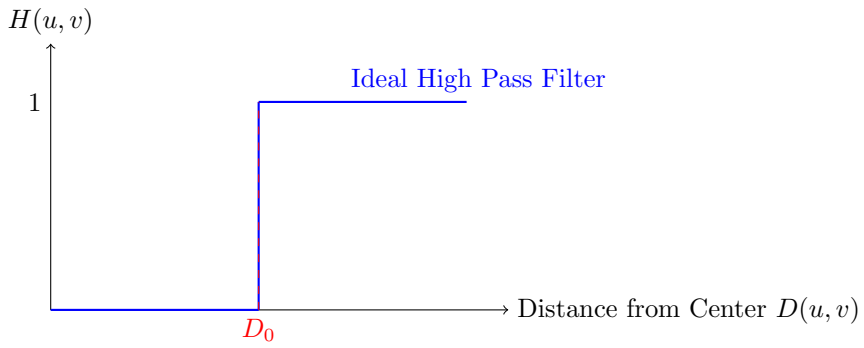
$$H(u,v) = \begin{cases} 0, & \text{if } D(u,v) \leq D_0 \\ 1, & \text{if } D(u,v) > D_0 \end{cases}$$

where $D(u,v) = \sqrt{(u - P/2)^2 + (v - Q/2)^2}$ is the distance from the center of the frequency domain, and D_0 is the cutoff distance.

Characteristics

- **Sharp Cutoff:** The ideal filter has a sharp transition at the cutoff frequency D_0.
- **Ringing Artifacts:** Due to the sharp frequency cutoff, ideal high pass filters produce ringing artifacts at edges in the filtered image.
- **Edge Enhancement:** Emphasizes edges and fine details while suppressing smooth regions.
- **Not Practical:** Rarely used in practice due to the ringing artifacts it introduces.

Visualization



Gaussian High Pass Filter

The Gaussian high pass filter provides a smooth transition between frequencies that are attenuated and frequencies that are passed. It avoids the sharp cutoff of the ideal filter, resulting in significantly fewer ringing artifacts.

Transfer Function

The transfer function for a Gaussian high pass filter is defined as:

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

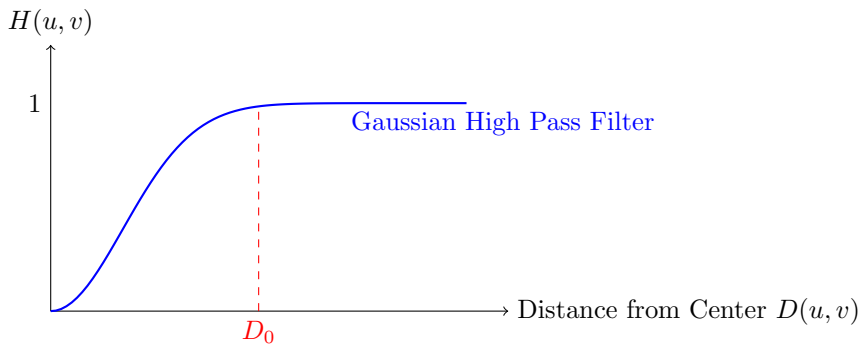
where σ controls the spread of the Gaussian function. Alternatively, using cutoff frequency D_0:

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

Characteristics

- **Smooth Transition:** Provides a gradual transition from stop-band to pass-band.
- **No Ringing:** Eliminates the ringing artifacts associated with ideal filters.
- **Natural Sharpening:** Produces more natural-looking sharpened images.
- **Commonly Used:** Widely used in practical edge detection and sharpening applications.

Visualization



Butterworth High Pass Filter

The Butterworth high pass filter provides a compromise between the ideal and Gaussian filters. It has a tunable parameter n (order) that controls the sharpness of the cutoff, allowing adjustment between smoothness and transition sharpness.

Transfer Function

The transfer function for a Butterworth high pass filter of order n is defined as:

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

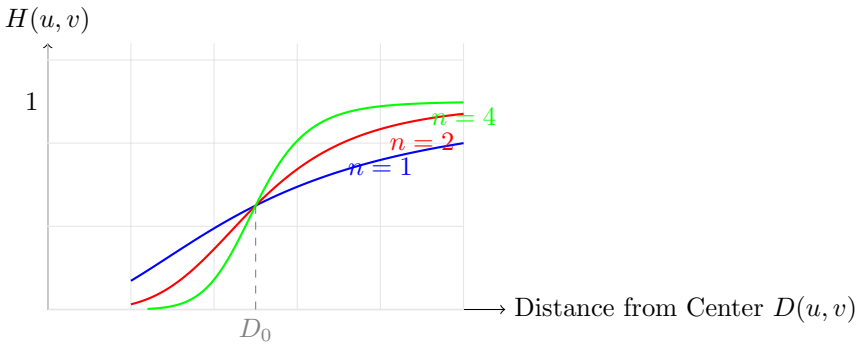
where:

- D_0 is the cutoff frequency (distance at which $H = 0.5$).
- n is the order of the filter, controlling the sharpness of the cutoff.
- $D(u,v)$ is the distance from the center of the frequency domain.

Characteristics

- **Adjustable Sharpness:** Order n controls the transition sharpness:
  - $n = 1$: Smooth, similar to Gaussian.
  - $n = 2, 3, \ldots$: Progressively sharper cutoff.
  - $n \to \infty$: Approaches ideal filter behavior.
- **Minimal Ringing:** Much less ringing than ideal filter but sharper than Gaussian.
- **Versatile:** Provides flexibility to balance between smoothness and cutoff sharpness.
- **Maximally Flat:** In the pass-band, the frequency response is maximally flat (no ripple).

Visualization



Mathematica Implementation

```
BHPF[img_, D0_, order_] :=

 Module[{A, Md, Nd, CM, B, B1, P, Q, BF, filterButterworth,
    filteredDFT, inverseDFT, phaseShift, filteredShifted, resultImg},

  A = ImageData[ColorConvert[img, "Grayscale"], "Byte"]; {Md, Nd} = Dimensions[A];

  CM = Table[(-1)^(i + j), {i, 1, Md}, {j, 1, Nd}]; B = A*CM;
  B1 = ArrayPad[B, {{0, Md}, {0, Nd}}]; {P, Q} = Dimensions[B1];
  BF = Fourier[B1, FourierParameters -> {1, -1}];
  filterButterworth =
   Table[1/(1 + (D0/(Sqrt[(u - P/2)^2 + (v - Q/2)^2] + 0.0001))^(2*
          order)), {u, 0, P - 1}, {v, 0, Q - 1}];
  filteredDFT = BF*filterButterworth;
  inverseDFT =
   InverseFourier[filteredDFT, FourierParameters -> {1, -1}];
  phaseShift = Table[(-1)^(x + y), {x, 0, P - 1}, {y, 0, Q - 1}];
  filteredShifted = inverseDFT*phaseShift;
  resultImg = Take[Re[filteredShifted], {1, Md}, {1, Nd}];
  Image[resultImg, "Byte"]]

i = Import["https://boofcv.org/images/thumb/6/66/Kodim17_noisy.jpg/300px-Kodim17_noisy.jpg"]
Manipulate[BHPF[i, cutoff, order], {cutoff, 1, 255, 1}, {order, 1, 4}]
```

Comparison: High Pass Filters

Property	Ideal	Gaussian	Butterworth
Cutoff Sharpness	Very sharp	Smooth	Adjustable (via n)
Ringing Artifacts	Severe	None	Minimal
Transition Smoothness	Abrupt	Gradual	Tunable
Practical Use	Rare	Very common	Common
Parameter Control	D_0 only	D_0 only	D_0 and n
Computational Cost	Low	Low	Low
Best For	Theoretical analysis	General sharpening	Balanced applications

Comparison: Low Pass vs. High Pass Filters

Property	Low Pass Filter	High Pass Filter
Primary Effect	Smoothing, blur reduction	Sharpening, edge enhancement
Frequency Response	Attenuates high frequencies	Attenuates low frequencies
Removes	Noise, fine details	Smooth regions, DC component
Preserves	Large-scale structure	Edges and fine details
Transfer Function	$H(u,v) = 1$ for $D \leq D_0$	$H(u,v) = 1$ for $D > D_0$
Application	Noise reduction preprocessing	Edge detection, feature enhancement
Output Appearance	Blurred, smoothed image	Enhanced edges, high contrast

Laplacian in Frequency Domain

The Laplacian operator can be implemented in the frequency domain as an alternative to spatial domain convolution. This approach is particularly useful when working with large images or when combining multiple frequency domain operations.

Laplacian Transfer Function

In the frequency domain, the Laplacian operator is represented by its transfer function:

$$H(u,v) = -4\pi^2(u^2 + v^2)$$

However, for practical implementation and to avoid amplifying noise excessively, a normalized version is often used:

$$H(u,v) = -(u^2 + v^2)$$

or equivalently:

$$H(u,v) = -D(u,v)^2 = -[(u - P/2)^2 + (v - Q/2)^2]$$

Properties

- **High-Pass Nature:** The Laplacian is a high-pass filter that enhances high-frequency components (edges and details).
- **Zero at DC:** The transfer function is zero at the origin (DC component), eliminating the average intensity from the output.
- **Isotropic:** Responds equally to edges in all directions.
- **Noise Amplification:** Amplifies high-frequency noise, so preprocessing with smoothing is often recommended.

Frequency Domain Laplacian Implementation

The procedure for applying the Laplacian in the frequency domain follows the standard 7-step frequency domain filtering approach:

1. Pad the input image to size $P \times Q$ where $P = 2M$ and $Q = 2N$.
2. Multiply by $(-1)^{x+y}$ to center the DFT.
3. Compute the DFT of the padded image.
4. Construct the Laplacian transfer function $H(u,v) = -(u - P/2)^2 - (v - Q/2)^2$.
5. Apply the filter: $G(u,v) = H(u,v) \cdot F(u,v)$.
6. Compute the inverse DFT and multiply by $(-1)^{x+y}$.
7. Extract the $M \times N$ result from the top-left quadrant.

Mathematica Implementation

```
(* Laplacian Filter in Frequency Domain *)
(* Step 1: Load and Pad Image *)
img = ExampleData[{"TestImage", "Lena"}];
{M, N} = ImageDimensions[img];
P = 2*M; Q = 2*N;
imgData = ImageData[img, "Real"];
paddedImg = PadRight[imgData, {P, Q}, 0];

(* Step 2: Phase Shift *)
phaseShift = Table[(-1)^(x + y), {x, 0, P-1}, {y, 0, Q-1}];
shiftedImg = paddedImg * phaseShift;

(* Step 3: Compute DFT *)
dftResult = Fourier[shiftedImg];

(* Step 4: Construct Laplacian Transfer Function *)
filterLaplacian = Table[
    -((u - P/2)^2 + (v - Q/2)^2),
    {u, 0, P-1}, {v, 0, Q-1}
];

(* Step 5: Apply Filter *)
filteredDFT = dftResult * filterLaplacian;

(* Step 6: Inverse DFT *)
inverseDFT = InverseFourier[filteredDFT];
filteredShifted = inverseDFT * phaseShift;

(* Step 7: Extract Result *)
resultImg = Take[Re[filteredShifted], {1, M}, {1, N}];
resultLaplacian = Image[Rescale[resultImg, {Min[resultImg], Max[resultImg]},
                                                    {0, 1}]];

(* Display *)
Row[{Image[imgData], resultLaplacian}, Spacer[10]]
```

Comparison: Spatial vs. Frequency Domain Laplacian

Aspect	**Spatial Domain**	**Frequency Domain**
Implementation	Direct convolution with kernel	FFT-based multiplication
Computational Efficiency	Slower for large images	Faster for large images (FFT)
Kernel Size	Fixed 3×3	Flexible (full image size)
Boundary Handling	Requires padding strategy	Automatic (circular)
Intuitive Understanding	Immediate (local differences)	Requires frequency interpretation
Combination with Other Filters	Difficult	Easy (multiply transfer functions)
Noise Sensitivity	Moderate	High (amplifies all frequencies)

Unsharp Masking, Highboost Filtering, and High Frequency Emphasis Filtering (Mathematica Example)

```
UnsharpMaskG[im_, D0_, k1_, k2_] := (
    img = ColorConvert[im, "Grayscale"];
    A = ImageData[img, "Byte"];
    d = Dimensions[A];
    M1 = d[[1]]; (* number of rows *)
    M2 = d[[2]]; (* number of columns *)
    CMat = Array[1 - 2 Mod[#1 + #2, 2] &, {M1, M2}];
    B = A*CMat;
    B1 = ArrayPad[B, {{0, M1 + 1}, {0, M2 + 1}}];
    d = Dimensions[B1];
    P = d[[1]];
    Q = d[[2]];
    BF = Fourier[B1, FourierParameters -> {1, -1}];
    H2 = GaussianMatrix[{{M1, M2}, D0}, Standardized -> False];
    H2n = H2/H2[[M1 + 1, M2 + 1]];
    H = 1 - H2n;
    G = BF*(k1 + k2*H);
    R = Re[InverseFourier[G, FourierParameters -> {1, -1}]];
    R1 = Take[R, {1, M1}, {1, M2}];
    R1 = R1*CMat;
    Grid[{{"Original image", "Filtered image",
         "Filtered image after intensity transformation"}, {im,
         Image[R1, "Byte"], HistogramTransform[Image[R1, "Byte"]]}},
     Spacings -> {2, 2, 2}, Frame -> All]
)
UnsharpMaskG[i4, 70, 0.5, 0.75]
```

Explanation: Unsharp Masking, Highboost Filtering, and High Frequency Emphasis Filtering

Unsharp masking is a classic image sharpening technique. It works by subtracting a blurred (lowpass filtered) version of the image from the original, emphasizing edges and fine details (high frequencies). The result can be added back to the original image for enhanced sharpness.

Highboost filtering generalizes unsharp masking by scaling the high-frequency component before adding it back. The formula is:

$$g(x, y) = f(x, y) + k \cdot [f(x, y) - \text{blur}(f(x, y))]$$

where $k > 1$ produces stronger sharpening.

High frequency emphasis filtering is a further generalization, where both the original and the high-frequency component are scaled independently:

$$G(u, v) = [k_1 + k_2 \cdot H_{HPF}(u, v)] \cdot F(u, v)$$

where H_{HPF} is a highpass filter (e.g., Gaussian), k_1 controls the original image, and k_2 controls the high-frequency emphasis.

In the code above: - The image is converted to grayscale and centered for frequency domain processing. - A Gaussian highpass filter is constructed. - The filter is applied in the frequency domain with weights k_1 and k_2 (high frequency emphasis). - The result is inverse transformed, de-centered, and displayed. - The third column shows the result after histogram equalization for better contrast.

This approach allows flexible sharpening, from subtle (unsharp masking) to aggressive (highboost), and can emphasize high-frequency details as needed.

How to Choose k_1 and k_2 in High Frequency Emphasis Filtering

The parameters k_1 and k_2 in high frequency emphasis filtering control the contribution of the original image and the high-frequency (sharpened) component, respectively:

$$G(u, v) = [k_1 + k_2 \cdot H_{HPF}(u, v)] \cdot F(u, v)$$

where $H_{HPF}(u, v)$ is the highpass filter transfer function.

- k_1 **(Low Frequency Weight):** Controls the amount of the original (low-frequency) image retained. Typical values are $0 \leq k_1 \leq 1$. Setting $k_1 = 1$ keeps the original image fully; $k_1 < 1$ reduces its contribution.
- k_2 **(High Frequency Weight):** Controls the strength of the high-frequency emphasis (sharpening). Typical values are $0 < k_2 \leq 1$. Increasing k_2 increases sharpening, but too large a value can amplify noise and cause artifacts.

Mathematica Code Examples for Frequency Domain Filtering

Calculating the Fourier Series: Small 4×4 Example

Below is Mathematica code to compute the 2D Discrete Fourier Transform (DFT) for your 4×4 matrix example, both uncentered and centered (using the $(-1)^{x+y}$ trick):

```
(* Define the 4x4 matrix *)
A = {{1, 2, 9, 7},
    {99, 88, 66, 22},
    {55, 1000, 999, 9998},
    {44, 22, 5555, 199}};

(* Uncentered DFT *)
DFTuncentered = Fourier[A, FourierParameters -> {1, -1}];

(* Centered DFT: multiply by (-1)^(x+y) *)
CM = Table[(-1)^(i + j), {i, 1, 4}, {j, 1, 4}];
Acentered = A * CM;
DFTcentered = Fourier[Acentered, FourierParameters -> {1, -1}];

(* Display magnitude spectra *)
Grid[{
  {"Original Matrix", "Uncentered DFT (Magnitude)", "Centered DFT (Magnitude)"},
  {MatrixForm[A], MatrixForm[Abs[DFTuncentered]], MatrixForm[Abs[DFTcentered]]}
}]
```

This code shows how to compute the DFT for your matrix, both with and without centering, and displays the magnitude spectra for comparison.

Ideal Low Pass Filter (ILPF)

```
ILPF[im_, D0_] := (
    img = ColorConvert[im, "Grayscale"];
    A = ImageData[img, "Byte"];
    {M1, M2} = Dimensions[A];
    CMat = Table[(-1)^(i + j), {i, 1, M1}, {j, 1, M2}];
    B = A * CMat; (* Centering the image *)
    B1 = ArrayPad[B, {{0, M1}, {0, M2}}];
    {P, Q} = Dimensions[B1];
    BF = Fourier[B1, FourierParameters -> {1, -1}];
    dis[u_, v_] := Sqrt[(u - M1)^2 + (v - M2)^2];
    H = Table[If[dis[u, v] <= D0, 1, 0], {u, 1, P}, {v, 1, Q}];
    G = BF * H;
    R = Re[InverseFourier[G, FourierParameters -> {1, -1}]];
    R1 = Take[R, {1, M1}, {1, M2}];
    R1 = R1 * CMat;
    Grid[{{"Original image", "Filter",
    "Filtered image before de-centering, still with padding", "Filtered image"},
        {img, Image[H], Image[R, "Byte"], Image[R1, "Byte"]}},
        Spacings -> {2, 2, 2, 2}, Frame -> All]
)
```

Ideal High Pass Filter (IHPF)

```
IHPF[im_, D0_] := (
    img = ColorConvert[im, "Grayscale"];
    A = ImageData[img, "Byte"];
    {M1, M2} = Dimensions[A];
    CMat = Table[(-1)^(i + j), {i, 1, M1}, {j, 1, M2}];
    B = A * CMat;
    B1 = ArrayPad[B, {{0, M1}, {0, M2}}];
    {P, Q} = Dimensions[B1];
    BF = Fourier[B1, FourierParameters -> {1, -1}];
    dis[u_, v_] := Sqrt[(u - M1)^2 + (v - M2)^2];
    H = Table[If[dis[u, v] <= D0, 1, 0], {u, 1, P}, {v, 1, Q}];
    G = BF * (1 - H);
    R = Re[InverseFourier[G, FourierParameters -> {1, -1}]];
    R1 = Take[R, {1, M1}, {1, M2}];
    R1 = R1 * CMat;
    Grid[{{"Original image", "Filter",
    "Filtered image before de-centering, still with padding", "Filtered image"},
        {img, Image[H], Image[R, "Byte"],
        Image[R1, "Byte"]}}, Spacings -> {2, 2, 2, 2}, Frame -> All]
)
```

Gaussian Low Pass Filter (GLPF)

```
GLPFfast[im_, D0_] := (
    img = ColorConvert[im, "Grayscale"];
    A = ImageData[img, "Byte"];
    d = Dimensions[A];
    M1 = d[[1]];
    M2 = d[[2]];
    CMat = Array[1 - 2 Mod[#1 + #2, 2] &, {M1, M2}];
    B = A * CMat;
    B1 = ArrayPad[B, {{0, M1 + 1}, {0, M2 + 1}}];
    d = Dimensions[B1];
    P = d[[1]];
    Q = d[[2]];
    BF = Fourier[B1, FourierParameters -> {1, -1}];
    H2 = GaussianMatrix[{{M1, M2}, D0}, Standardized -> False];
    H2n = H2 / H2[[M1 + 1, M2 + 1]];
    G = BF * H2n;
    R = Re[InverseFourier[G, FourierParameters -> {1, -1}]];
    R1 = Take[R, {1, M1}, {1, M2}];
    R1 = R1 * CMat;
    Grid[{{"Original image", "Filter",
    "Filtered image before de-centering, still with padding", "Filtered image"},
        {img, Image[H2n], Image[R, "Byte"], Image[R1, "Byte"]}},
        Spacings -> {2, 2, 2, 2}, Frame -> All]
)
```

Gaussian High Pass Filter (GHPF)

```
GHPFfast[im_, D0_] := (
    img = ColorConvert[im, "Grayscale"];
    A = ImageData[img, "Byte"];
    d = Dimensions[A];
    M1 = d[[1]];
    M2 = d[[2]];
    CMat = Array[1 - 2 Mod[#1 + #2, 2] &, {M1, M2}];
    B = A * CMat;
    B1 = ArrayPad[B, {{0, M1 + 1}, {0, M2 + 1}}];
    d = Dimensions[B1];
    P = d[[1]];
    Q = d[[2]];
    BF = Fourier[B1, FourierParameters -> {1, -1}];
    H2 = GaussianMatrix[{{M1, M2}, D0}, Standardized -> False];
    H2n = H2 / H2[[M1 + 1, M2 + 1]];
    G = BF * (1 - H2n);
    R = Re[InverseFourier[G, FourierParameters -> {1, -1}]];
    R1 = Take[R, {1, M1}, {1, M2}];
    R1 = R1 * CMat;
    Grid[{{"Original image", "Filter",
    "Filtered image before de-centering, still with padding", "Filtered image"},
        {img, Image[1 - H2n], Image[R, "Byte"], Image[R1, "Byte"]}},
        Spacings -> {2, 2, 2, 2}, Frame -> All]
)
```

Unsharp Masking with Gaussian High Pass Filter

```
UnsharpMaskG[im_, D0_, k1_, k2_] := (
    img = ColorConvert[im, "Grayscale"];
    A = ImageData[img, "Byte"];
    d = Dimensions[A];
    M1 = d[[1]];
    M2 = d[[2]];
    CMat = Array[1 - 2 Mod[#1 + #2, 2] &, {M1, M2}];
    B = A * CMat;
    B1 = ArrayPad[B, {{0, M1 + 1}, {0, M2 + 1}}];
    d = Dimensions[B1];
    P = d[[1]];
    Q = d[[2]];
    BF = Fourier[B1, FourierParameters -> {1, -1}];
    H2 = GaussianMatrix[{{M1, M2}, D0}, Standardized -> False];
    H2n = H2 / H2[[M1 + 1, M2 + 1]];
    H = 1 - H2n;
    G = BF * (k1 + k2 * H);
    R = Re[InverseFourier[G, FourierParameters -> {1, -1}]];
    R1 = Take[R, {1, M1}, {1, M2}];
    R1 = R1 * CMat;
    Grid[{{"Original image", "Filtered image", "Filtered image after intensity transformation"},
        {im, Image[R1, "Byte"], HistogramTransform[Image[R1, "Byte"]]}},
        Spacings -> {2, 2, 2}, Frame -> All]
)
```

Comparison: Spatial vs. Frequency Domain Unsharp Masking

Aspect	Spatial Domain	Frequency Domain
Implementation	Explicit blur + subtraction	Single transfer function
Computational Efficiency	Slower (multiple convolutions)	Faster (single FFT)
Filter Flexibility	Limited to kernel size	Flexible frequency control
Control	Direct visual parameters	Frequency-based parameters
Combination	Difficult (sequential ops)	Easy (multiply functions)
Intuitive Understanding	High (visual process)	Lower (frequency interpretation)

Band Reject, Band Pass, and Notch Filters

This section provides Mathematica code for frequency domain band reject, band pass, and notch filters. Each filter is illustrated with a schematic graph showing its frequency response, followed by the corresponding Mathematica implementation.

Mathematica Code Examples

1. Ideal Band Reject Filter

```
IdealBandRejectFilter[img_, D0_, W_] := Module[
    {data, rows, cols, P, Q, paddedData, centeredData,
    F, H, d, G, gInverse, gReal, finalData, result},
    data = ImageData[ColorConvert[img, "Grayscale"]];
    {rows, cols} = Dimensions[data];
    P = 2*rows; Q = 2*cols;
    paddedData = ArrayPad[data, {{0, rows}, {0, cols}}];
    centeredData = Table[paddedData[[x, y]]*(-1)^(x + y), {x, 1, P},
    {y, 1, Q}];
    F = Fourier[centeredData, FourierParameters -> {1, -1}];
    H = Table[
        d = Sqrt[(u - P/2)^2 + (v - Q/2)^2];
        If[D0 - W/2 <= d <= D0 + W/2, 0, 1],
        {u, 1, P}, {v, 1, Q}
    ];
    G = F*H;
    gInverse = InverseFourier[G, FourierParameters -> {1, -1}];
    gReal = Re[gInverse];
    finalData = Table[gReal[[x, y]]*(-1)^(x + y), {x, 1, P}, {y, 1, Q}];
    result = Take[finalData, {1, rows}, {1, cols}];
    Image[result]
]
```

2. Gaussian Band Reject Filter

```
GaussianBandRejectFilter[img_, D0_, W_] := Module[
    {data, rows, cols, P, Q, paddedData, centeredData,
    F, H, d, G, gInverse, gReal, finalData, result},
    data = ImageData[ColorConvert[img, "Grayscale"]];
    {rows, cols} = Dimensions[data];
    P = 2*rows; Q = 2*cols;
    paddedData = ArrayPad[data, {{0, rows}, {0, cols}}];
    centeredData = Table[paddedData[[x, y]]*(-1)^(x + y), {x, 1, P},
    {y, 1, Q}];
    F = Fourier[centeredData, FourierParameters -> {1, -1}];
    H = Table[
        d = Sqrt[(u - P/2)^2 + (v - Q/2)^2];
        If[d == 0, 1, 1 - Exp[-0.5*((d^2 - D0^2)/(d*W))^2]],
```

```
        {u, 1, P}, {v, 1, Q}
    ];
    G = F*H;
    gInverse = InverseFourier[G, FourierParameters -> {1, -1}];
    gReal = Re[gInverse];
    finalData = Table[gReal[[x, y]]*(-1)^(x + y), {x, 1, P}, {y, 1, Q}];
    result = Take[finalData, {1, rows}, {1, cols}];
    Image[result]
]
```

3. Butterworth Band Reject Filter

```
ButterworthBandRejectFilter[img_, D0_, W_, n_] := Module[
    {data, rows, cols, P, Q, paddedData, centeredData,
    F, H, d, denom, G, gInverse, gReal, finalData, result},
    data = ImageData[ColorConvert[img, "Grayscale"]];
    {rows, cols} = Dimensions[data];
    P = 2*rows; Q = 2*cols;
    paddedData = ArrayPad[data, {{0, rows}, {0, cols}}];
    centeredData = Table[paddedData[[x, y]]*(-1)^(x + y),
    {x, 1, P}, {y, 1, Q}];
    F = Fourier[centeredData, FourierParameters -> {1, -1}];
    H = Table[
        d = Sqrt[(u - P/2)^2 + (v - Q/2)^2];
        denom = (d^2 - D0^2) + 0.00001;
        1/(1 + ((d*W)/denom)^(2*n)),
        {u, 1, P}, {v, 1, Q}
    ];
    G = F*H;
    gInverse = InverseFourier[G, FourierParameters -> {1, -1}];
    gReal = Re[gInverse];
    finalData = Table[gReal[[x, y]]*(-1)^(x + y), {x, 1, P}, {y, 1, Q}];
    result = Take[finalData, {1, rows}, {1, cols}];
    Image[result]
]
```

4. Ideal Band Pass Filter

```
IdealBandPassFilter[img_, D0_, W_] := Module[
    {data, rows, cols, P, Q, paddedData, centeredData,
    F, H, d, G, gInverse, gReal, finalData, result},
    data = ImageData[ColorConvert[img, "Grayscale"]];
    {rows, cols} = Dimensions[data];
    P = 2*rows; Q = 2*cols;
    paddedData = ArrayPad[data, {{0, rows}, {0, cols}}];
    centeredData = Table[paddedData[[x, y]]*(-1)^(x + y),
    {x, 1, P}, {y, 1, Q}];
    F = Fourier[centeredData, FourierParameters -> {1, -1}];
    H = Table[
        d = Sqrt[(u - P/2)^2 + (v - Q/2)^2];
        If[D0 - W/2 <= d <= D0 + W/2, 1, 0],
        {u, 1, P}, {v, 1, Q}
    ];
    G = F*H;
    gInverse = InverseFourier[G, FourierParameters -> {1, -1}];
    gReal = Re[gInverse];
    finalData = Table[gReal[[x, y]]*(-1)^(x + y), {x, 1, P},
    {y, 1, Q}];
    result = Take[finalData, {1, rows}, {1, cols}];
    Image[result]
```

```
]
```

5. Gaussian Band Pass Filter

```
GaussianBandPassFilter[img_, D0_, W_] := Module[
    {data, rows, cols, P, Q, paddedData, centeredData,
    F, H, d, G, gInverse, gReal, finalData, result},
    data = ImageData[ColorConvert[img, "Grayscale"]];
    {rows, cols} = Dimensions[data];
    P = 2*rows; Q = 2*cols;
    paddedData = ArrayPad[data, {{0, rows}, {0, cols}}];
    centeredData = Table[paddedData[[x, y]]*(-1)^(x + y), {x, 1, P},
    {y, 1, Q}];
    F = Fourier[centeredData, FourierParameters -> {1, -1}];
    H = Table[
        d = Sqrt[(u - P/2)^2 + (v - Q/2)^2];
        If[d == 0, 0, Exp[-0.5*((d^2 - D0^2)/(d*W))^2]],
        {u, 1, P}, {v, 1, Q}
    ];
    G = F*H;
    gInverse = InverseFourier[G, FourierParameters -> {1, -1}];
    gReal = Re[gInverse];
    finalData = Table[gReal[[x, y]]*(-1)^(x + y), {x, 1, P}, {y, 1, Q}];
    result = Take[finalData, {1, rows}, {1, cols}];
    Image[result]
]
```

6. Butterworth Band Pass Filter

```
ButterworthBandPassFilter[img_, D0_, W_, n_] := Module[
    {data, rows, cols, P, Q, paddedData, centeredData,
    F, H, d, denom, G, gInverse, gReal, finalData, result},
    data = ImageData[ColorConvert[img, "Grayscale"]];
    {rows, cols} = Dimensions[data];
    P = 2*rows; Q = 2*cols;
    paddedData = ArrayPad[data, {{0, rows}, {0, cols}}];
    centeredData = Table[paddedData[[x, y]]*(-1)^(x + y),
    {x, 1, P}, {y, 1, Q}];
    F = Fourier[centeredData, FourierParameters -> {1, -1}];
    H = Table[
        d = Sqrt[(u - P/2)^2 + (v - Q/2)^2];
        denom = (d^2 - D0^2) + 0.00001;
        1 - 1/(1 + ((d*W)/denom)^(2*n)),
        {u, 1, P}, {v, 1, Q}
    ];
    G = F*H;
    gInverse = InverseFourier[G, FourierParameters -> {1, -1}];
    gReal = Re[gInverse];
    finalData = Table[gReal[[x, y]]*(-1)^(x + y), {x, 1, P}, {y, 1, Q}];
    result = Take[finalData, {1, rows}, {1, cols}];
    Image[result]
]
```

7. Butterworth Notch Filter

```
ButterworthNotchFilter[img_, centers_, D0_, n_] := Module[
    {data, rows, cols, P, Q, paddedData, centeredData,
    F, H, d1, d2, G, gInverse, gReal, finalData, result},
    data = ImageData[ColorConvert[img, "Grayscale"]];
    {rows, cols} = Dimensions[data];
    P = 2*rows; Q = 2*cols;
```

```
    paddedData = ArrayPad[data, {{0, rows}, {0, cols}}];
    centeredData = Table[paddedData[[x, y]]*(-1)^(x + y), {x, 1, P},
    {y, 1, Q}];
    F = Fourier[centeredData, FourierParameters -> {1, -1}];
    H = Table[
        Product[
            d1 = Sqrt[(u - P/2 - c[[1]])^2 + (v - Q/2 - c[[2]])^2];
            d2 = Sqrt[(u - P/2 + c[[1]])^2 + (v - Q/2 + c[[2]])^2];
            If[d1*d2 == 0, 0, 1/(1 + (D0^2/(d1*d2))^n)],
            {c, centers}
        ],
        {u, 1, P}, {v, 1, Q}
    ];
    G = F*H;
    gInverse = InverseFourier[G, FourierParameters -> {1, -1}];
    gReal = Re[gInverse];
    finalData = Table[gReal[[x, y]]*(-1)^(x + y), {x, 1, P}, {y, 1, Q}];
    result = Take[finalData, {1, rows}, {1, cols}];
    Image[result]
]
```

Usage Example:

```
(* Some helper functions *)
ImagePeriodogram[imgGray]
centers = {{114, 43}, {52, 205}, {55, 41}, {109, 207},
    {112, 85}, {110, 165}, {56, 81}, {53, 161}};
{w, h} = ImageDimensions[img];
imageCenter = {w/2, h/2};
finalCenters = Map[Reverse[# - imageCenter]*2 &, centers];
finalCenters // MatrixForm
```

```
img = Import["ExampleData/lena.tif"];
D0 = 60; W = 30; n = 2;
notchCenters = {{40, 40}, {40, -40}}; notchRadius = 10;

outIdealBR = IdealBandRejectFilter[img, D0, W];
outGaussBR = GaussianBandRejectFilter[img, D0, W];
outButterBR = ButterworthBandRejectFilter[img, D0, W, n];
outIdealBP = IdealBandPassFilter[img, D0, W];
outGaussBP = GaussianBandPassFilter[img, D0, W];
outButterBP = ButterworthBandPassFilter[img, D0, W, n];
outNotch = ButterworthNotchFilter[img, notchCenters, notchRadius, n];

(* Display results in a grid *)
LabelImg[image_, text_] := Column[{image, Style[text,
FontFamily -> "Helvetica", 12, Gray]}, Center];
Grid[{
    {LabelImg[img, "Original Image"],
    LabelImg[outNotch, "7. Notch Filter"]},
    {LabelImg[outIdealBR, "1. Ideal Band Reject"],
    LabelImg[outGaussBR, "2. Gaussian Band Reject"]},
    {LabelImg[outButterBR, "3. Butterworth Band Reject"], SpanFromLeft},
    {LabelImg[outIdealBP, "4. Ideal Band Pass"],
    LabelImg[outGaussBP, "5. Gaussian Band Pass"]},
    {LabelImg[outButterBP, "6. Butterworth Band Pass"], SpanFromLeft}
}, Frame -> All, Spacings -> {1, 1}, Alignment -> Center]
```

Summary

- Notch filters are essential for removing periodic noise in images.
- The filter is constructed to target specific frequency locations.
- Butterworth and Gaussian notch filters are preferred for smooth transitions and minimal artifacts.
- The process involves identifying noise frequencies, constructing the filter, applying it in the frequency domain, and transforming back to the spatial domain.

Quick Reference: Which Frequency Domain Filter to Use?

Scenario / Problem	Recommended Filter	Effect / Rationale	Typical Parameters
General smoothing, noise reduction	**Gaussian Low Pass**	Smooths image, minimal ringing, natural blur	$D_0 = 10$–50 (depends on image size)
Aggressive smoothing, theoretical study	**Ideal Low Pass**	Removes all frequencies above cutoff, but causes ringing	$D_0 = 10$–50
Smoothing with adjustable sharpness	**Butterworth Low Pass**	Tunable transition, balance between ideal and Gaussian	$D_0 = 10$–50, $n = 2$–4
Edge enhancement, detail sharpening	**Gaussian High Pass**	Enhances edges, smooth transition, minimal artifacts	$D_0 = 10$–50
Strong edge enhancement, theoretical	**Ideal High Pass**	Sharp cutoff, strong edge emphasis, severe ringing	$D_0 = 10$–50
Sharpening with tunable sharpness	**Butterworth High Pass**	Adjustable edge enhancement, less ringing than ideal	$D_0 = 10$–50, $n = 2$–4
Controlled sharpening, edge boost	**Unsharp Masking / Highboost**	Adds scaled high-frequency detail, flexible control	$k_1 = 1$, $k_2 = 0.5$–1, $D_0 = 20$–70
Fine detail enhancement	**Laplacian (Frequency Domain)**	Highlights rapid intensity changes, isotropic	No parameter (or scale $H(u, v)$ as needed)
Remove periodic noise (stripes, interference)	**Band Reject / Notch Filter**	Suppresses specific frequency bands or points	D_0 = distance to noise, W = width, $n = 2$–4
Enhance features in a frequency range	**Band Pass Filter**	Keeps only desired frequency band	D_0 = center, W = width, $n = 2$–4

- **Low Pass:** Use for denoising and smoothing. Gaussian is safest for natural images.
- **High Pass:** Use for sharpening and edge enhancement. Gaussian or Butterworth preferred.
- **Band Reject/Notch:** Use to remove periodic or localized frequency noise.
- **Unsharp/Highboost:** Use for controlled sharpening with k_2 for strength.
- **Laplacian:** Use for isotropic edge enhancement.
- **Parameter choice:** D_0 should be $\sim$2–10% of image diagonal for most images; $n = 2$ is a good default for Butterworth.

Color Image Processing

This section demonstrates how to process color images in Mathematica, including color space conversion, channel separation, intensity transformations, color segmentation, and color balancing. Each example is accompanied by code and explanations.

1. Creating an RGB Image from a Matrix

You can create a small color image by specifying a 2×3 matrix, where each entry is a list of three values (R, G, B) between 0 and 1.

```
(* Create a 2x3 RGB matrix with random values *)
m = RandomReal[{0, 1}, {2, 3, 3}];

(* Display as an RGB image *)
Image[m, ColorSpace -> "RGB"]

(* Display as an HSB image (interprets the numbers as HSB, not RGB) *)
Image[m, ColorSpace -> "HSB"]
```

Explanation: The same matrix can be interpreted as RGB or HSB. The appearance will change because the numbers are interpreted differently in each color space.

2. Color Space Analysis and Conversion

Given a color image (e.g., the "Mandrill" test image):

```
(* Load the image *)
img = ExampleData[{"TestImage", "Mandrill"}];

(* a. Find the color space *)
ImageColorSpace[img]  (* Output: "RGB" *)

(* b. Visualize the image as if its data were HSB *)
Image[img, ColorSpace -> "HSB"]

(* c. Convert the image to HSB properly *)
imgHSB = ColorConvert[img, "HSB"];
ImageColorSpace[imgHSB]  (* Output: "HSB" *)

(* d. Extract the underlying data matrices *)
A1 = ImageData[img];      (* RGB data *)
A2 = ImageData[imgHSB];   (* HSB data *)

A1[[1, 1]]  (* First pixel in RGB *)
A2[[1, 1]]  (* First pixel in HSB *)

(* e. Visualize the HSB matrix as if it were RGB *)
Image[A2, ColorSpace -> "RGB"]
```

Explanation: Interpreting the same data in different color spaces produces different images. Converting between spaces changes the data to match the new space.

3. Separating Color Channels

You can extract individual color channels (e.g., R, G, B, or H, S, B) using `ColorSeparate`:

```
(* Separate RGB channels *)
rgbChannels = ColorSeparate[img];
```

```
(* Separate CMYK channels *)
imgCMYK = ColorConvert[img, "CMYK"];
cmykChannels = ColorSeparate[imgCMYK];

(* Separate HSB channels *)
imgHSB = ColorConvert[img, "HSB"];
hsbChannels = ColorSeparate[imgHSB];
```

Explanation: Each channel can be visualized separately to analyze the contribution of each color component.

4. Intensity Transformations on Color Images

You can apply intensity transformations to each channel or to the whole image:

```
(* Define transformation functions *)
f1[r_] := 0.7*r;
f2[r_] := r^2;
f3[r_] := Sqrt[r];
f4[r_] := 1 - r;

(* Apply to all channels *)
imgF1 = ImageApply[f1, img];
imgF2 = ImageApply[f2, img];
imgF3 = ImageApply[f3, img];
imgF4 = ImageApply[f4, img];  (* Equivalent to ColorNegate *)

(* Apply to a single channel (e.g., red) *)
channels = ColorSeparate[img];
redTrans = ImageApply[f4, channels[[1]]];
imgRedNeg = ColorCombine[{redTrans, channels[[2]], channels[[3]]}];
```

Explanation: You can transform all channels equally or just one channel to see the effect on color balance.

5. Color Slicing / Segmentation

To extract a region of a specific color (e.g., the mandrill's nose):

```
(* Select a region (e.g., nose) *)
rows = {334, 356}; columns = {220, 265};
A = ImageData[img];
AS = Take[A, rows, columns];

(* Compute average color in the region *)
meanColor = Mean /@ Transpose[Flatten[AS, 1]];

(* Define a segmentation function *)
f[r_, s_] := If[EuclideanDistance[r, meanColor] >= s*Sqrt[3.], {0.5, 0.5, 0.5}, r];

(* Interactive segmentation *)
Manipulate[
    ImageApply[f[#, s] &, img],
    {s, 0., 1.0, 0.05}
]
```

Explanation: Pixels close to the average color of the selected region are kept; others are replaced with gray.

6. Tonal Correction (Gamma Adjustment)

To correct the tone of an image using a power-law (gamma) transformation:

```
(* Gamma correction with interactive control *)
Manipulate[
    ImageApply[#^gamma &, img],
    {gamma, 0.1, 0.7, 0.05}
]
```

Explanation: Adjusting gamma brightens or darkens the image nonlinearly.

7. Color Balancing

To correct color balance by adjusting individual channels:

```
(* Separate channels *)
channels = ColorSeparate[img];

(* Apply a transformation to the red channel *)
redBalanced = ImageApply[#^3.33 &, channels[[1]]];

(* Combine channels back *)
imgBalanced = ColorCombine[{redBalanced, channels[[2]], channels[[3]]}];
```

Explanation: If one channel dominates (e.g., too much red), compressing its values can restore balance.

8. Sharpening Color Images

To sharpen a color image using a Laplacian filter:

```
(* Apply Laplacian filter *)
imgSharp = LaplacianFilter[img, 10];

(* Subtract to enhance edges *)
imgEnhanced = ImageSubtract[img, imgSharp];
```

Explanation: Subtracting a blurred (Laplacian) version enhances edges and details.

Summary: Color image processing in Mathematica involves manipulating color spaces, separating channels, applying transformations, and combining results. The code examples above illustrate common operations and their pedagogical rationale.

Bit Depth in Digital Images

Bit depth refers to the number of bits used to represent the color or intensity of a single pixel in a digital image. It determines how many distinct colors or gray levels can be displayed.

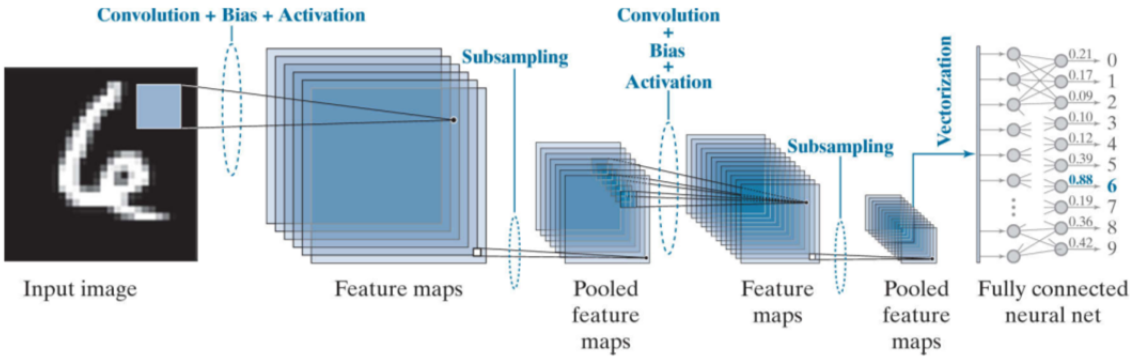
- For a grayscale image, the bit depth is the number of bits per pixel. For example, an 8-bit grayscale image can represent $2^8 = 256$ different intensity levels (from 0 to 255).
- For a color image (such as RGB), the bit depth is typically given as the sum of the bits used for each channel. For example, if each of the Red, Green, and Blue channels uses 8 bits, the total bit depth is $8 + 8 + 8 = 24$ bits per pixel. This allows $2^{24} = 16,777,216$ possible colors.
- Sometimes, you may see a bit depth like 9 bits or 512 colors. This means $2^9 = 512$ possible colors or intensity levels. For example, a 3-3-3 bit RGB image (3 bits per channel) gives $2^3 \times 2^3 \times 2^3 = 8 \times 8 \times 8 = 512$ colors.

Examples:

- **24-bit color:** 8 bits per channel $\times$ 3 channels = 24 bits per pixel. Number of colors: $2^{24} = 16,777,216$.
- **9-bit color:** 3 bits per channel $\times$ 3 channels = 9 bits per pixel. Number of colors: $2^9 = 512$.

In summary: **Bit depth** = (bits per channel) $\times$ (number of channels); **Number of colors/levels** $= 2^{\text{bit depth}}$.

Convolutional Neural Networks (CNNs)



The CNN Process: A 4-Step Summary

Convolutional Neural Networks (CNNs) process images directly as 2D arrays, preserving spatial relationships. The main steps are:

1. **Convolution (Feature Extraction):**
   - Slide a small filter (kernel) over the input image.
   - At each position, compute the sum of products between pixel values and kernel weights.
   - Produces a **feature map** highlighting patterns (edges, lines, etc.).
   - Multiple kernels yield multiple feature maps.
2. **Activation (Non-Linearity):**
   - Add a bias, then apply an activation function (commonly ReLU).
   - Introduces non-linearity, helping the network learn complex features.
3. **Subsampling (Pooling):**
   - Reduce feature map size for efficiency and robustness.
   - Use a window (e.g., 2×2) to keep the maximum (Max-Pooling) or average (Average Pooling) value.
4. **Classification (Fully Connected Layer):**
   - Flatten pooled feature maps into a vector.
   - Feed into a fully connected neural network for final classification (e.g., digit "6" vs "0").

General Formula for Output Size

For both convolution and pooling layers, the output size (width or height) is:

$$\text{Output Size} = \left\lfloor \frac{\text{Input} - \text{Kernel} + 2 \times \text{Padding}}{\text{Stride}} \right\rfloor + 1$$

- **Input:** Size of incoming map.
- **Kernel:** Size of filter or pooling window.
- **Padding:** Pixels added to border (often 0).
- **Stride:** Step size of kernel movement.

Examples:

- **Convolution Layer:** Input = 12, Kernel = 5, Padding = 0, Stride = 1

$$\frac{12 - 5 + 0}{1} + 1 = 8$$

  (Output: 8×8 feature map)

- **Pooling Layer:** Input = 8, Kernel = 2, Padding = 0, Stride = 2

$$\frac{8 - 2 + 0}{2} + 1 = 4$$

  (Output: 4×4 pooled map)

Case Study: Step-by-Step Construction and Evaluation of a CNN in Mathematica (MNIST Example)

This section summarizes the full workflow for building, training, and evaluating a simple Convolutional Neural Network (CNN) for handwritten digit recognition using the MNIST dataset in Mathematica, as illustrated in the attached code.

1. Load the MNIST Dataset

```
trainingData = ResourceData["MNIST", "TrainingData"];
testData = ResourceData["MNIST", "TestData"];
Length[testData]  (* Output: 10000 *)
```

2. Explore Data and Prepare Encoders

```
sample = testData[[1]];
img = Keys[sample][[1]];
ImageDimensions[img]  (* Output: {28, 28} *)
enc = NetEncoder[{"Image", {28, 28}, "ColorSpace" -> "Grayscale"}];
enc[img] // Dimensions  (* Output: {1, 28, 28} *)
```

3. Build and Inspect the First CNN Layers

```
conv1 = ConvolutionLayer[6, 5, "Input" -> {1, 28, 28}];
conv1 = NetInitialize[conv1];
activation1 = ElementwiseLayer["ReLU"];
pooling1 = PoolingLayer[{2, 2}, 2];
output1 = pooling1[activation1[conv1[enc[img]]]];
output1 // Length  (* Output: 6 *)
Image[output1[[1]]]  (* Visualize feature map *)
output1[[1]] // Dimensions  (* Output: {12, 12} *)
```

4. Add a Second Convolutional Block

```
conv2 = ConvolutionLayer[12, 5, "Input" -> {6, 12, 12}];
conv2 = NetInitialize[conv2];
activation2 = ElementwiseLayer["ReLU"];
pooling2 = PoolingLayer[{2, 2}, 2];
output2 = pooling2[activation2[conv2[output1]]];
output2 // Dimensions  (* Output: {12, 4, 4} *)
Flatten[output2] // Length  (* Output: 192 *)
```

5. Assemble the Full CNN

```
labels = Range[0, 9];
net1 = NetChain[{
    ConvolutionLayer[6, 5],     (*Layer 1*)
    ElementwiseLayer["ReLU"],
    PoolingLayer[{2, 2}, 2],
    ConvolutionLayer[12, 5, "Input" -> {6, 12, 12}],   (*Layer 4*)
    ElementwiseLayer["ReLU"],
    PoolingLayer[{2, 2}, 2],
    FlattenLayer[],
    LinearLayer[192],
    ElementwiseLayer["ReLU"],
    LinearLayer[10],
    SoftmaxLayer[]      (*Layer 11*)
  },
```

```
  "Input" -> NetEncoder[{"Image", {28, 28}, "ColorSpace" -> "Grayscale"}],
  "Output" -> NetDecoder[{"Class", labels}]
]
```

6. Train the Network and Monitor Performance

```
train = NetTrain[net1, trainingData, All, ValidationSet -> testData]
(* Outputs training/validation loss and error rate per epoch *)
```

7. Evaluate the Trained Network

```
trained = train["TrainedNet"];
trained[Keys[testData[[1]]]]  (* Predicts digit for a test image *)
```

8. Compute Accuracy, Confusion Matrix, and Recall

```
measurements = NetMeasurements[trained, testData, {"Accuracy", "ConfusionMatrixPlot", "Recall"}];
measurements[[1]]  (* Accuracy, e.g., 0.9909 *)
measurements[[2]]  (* Confusion matrix plot *)
BarChart[measurements[[3]], ChartLabels -> Range[0, 9],
  PlotLabel -> "True Positive Rate (Recall) by Digit",
  AxesLabel -> {"Digit Class", "True Positive Rate"},
  PlotTheme -> "Detailed",
  ColorFunction -> "Rainbow",
  LabelingFunction -> (Placed[Round[#, 0.01], Above] &)
]
```

9. Compare with Pretrained LeNet

```
testData = ExampleData[{"MachineLearning", "MNIST"}, "TestData"];
lenet = NetModel["LeNet Trained on MNIST Data"];
lenetresults = NetMeasurements[lenet, testData, {"Accuracy", "ConfusionMatrixPlot", "Recall"}];
BarChart[lenetresults[[3]], ChartLabels -> Range[0, 9],
  PlotLabel -> "True Positive Rate (Recall) by Digit",
  AxesLabel -> {"Digit Class", "True Positive Rate"},
  PlotTheme -> "Detailed",
  ColorFunction -> "Rainbow",
  LabelingFunction -> (Placed[Round[#, 0.01], Above] &)
]
```

10. Visualize Learned Kernels

```
kernelToImage[k_] := Image[Rescale[k]] // ImageResize[#, 50] &;
weights1 = Normal@NetExtract[trained, {1, "Weights"}];
kernels1 = kernelToImage /@ Flatten[weights1, 1];
Row[kernels1, " "]

weights2 = Normal@NetExtract[trained, {4, "Weights"}];
kernels2 = Map[kernelToImage, weights2, {2}];
Column[kernels2, " "]
```

11. Visualize Feature Maps at Each Stage

```
vizMaps[data_] := Row[Map[ImageResize[Image[Rescale[#]], {50, 50}] &, Normal[data]], Spacer[2]]
sampleImage = Keys@testData[[5102]];

featA = NetTake[trained, 2][sampleImage];
Labeled[vizMaps[featA], "a) Conv1 (6 maps)"]
```

```
featB = NetTake[trained, 3][sampleImage];
Labeled[vizMaps[featB], "b) Pool1 (6 maps)"]

featC = NetTake[trained, 5][sampleImage];
Labeled[vizMaps[featC], "c) Conv2 (12 maps)"]

featD = NetTake[trained, 6][sampleImage];
Labeled[vizMaps[featD], "d) Pool2 (12 maps)"]

featE = NetTake[trained, 7][sampleImage];
Labeled[BarChart[featE, PlotTheme -> "Business", ImageSize -> 500, AspectRatio -> 1/4, PlotLabel ->

featF = Normal@NetTake[trained, 8][sampleImage];
Labeled[BarChart[featF, PlotTheme -> "Business", ImageSize -> 500, AspectRatio -> 1/4, PlotLabel ->

featG = Normal@NetTake[trained, 9][sampleImage];
Labeled[BarChart[featG, PlotTheme -> "Business", ImageSize -> 500, AspectRatio -> 1/4, PlotLabel ->

featH = NetTake[trained, 10][sampleImage];
featI = NetTake[trained, 11][sampleImage];
```

Summary

This workflow demonstrates:

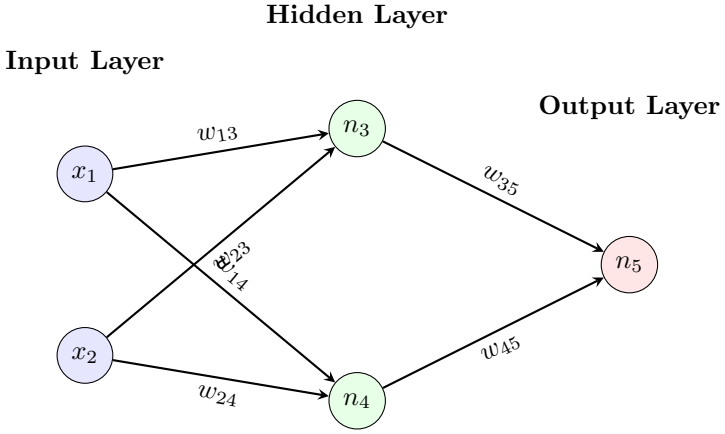
- Loading and preparing image data for CNNs.
- Building and visualizing each layer's output (feature maps, pooling, flattening).
- Training and evaluating a CNN, including accuracy and confusion matrix.
- Comparing with a reference model (LeNet).
- Visualizing learned kernels and feature activations at each stage.

Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of biological neural networks. They consist of interconnected nodes (neurons) organized in layers, which process input data and learn complex patterns through training.

ANN Architecture Example

The following diagram illustrates a simple feedforward neural network with two input nodes, a hidden layer with two neurons, and one output node. This architecture is commonly used for tasks such as binary classification (e.g., XOR problem).



How ANNs Work

- **Input Layer:** Receives raw data (e.g., pixel values, features).
- **Hidden Layer(s):** Each neuron computes a weighted sum of its inputs, applies an activation function (e.g., sigmoid), and passes the result forward.
- **Output Layer:** Produces the final prediction or classification.
- **Weights:** Connections between neurons have adjustable weights (w_{ij}) that are learned during training.
- **Activation Functions:** Nonlinear functions (e.g., sigmoid, ReLU) allow the network to model complex relationships.
- **Training:** The network learns by adjusting weights to minimize the error between predicted and target outputs, typically using backpropagation and gradient descent.

Example: Solving XOR with ANN

The reference code implements a simple ANN to solve the XOR problem:

- Two inputs (x_1, x_2).
- Two hidden neurons (n_3, n_4) with sigmoid activation.
- One output neuron (n_5) with sigmoid activation.
- Weights are updated using backpropagation to minimize output error.

Reference Python code

```
import math

# Activation functions
def sign(a):
    if a >= 0:
        return 1
    else:
        return -1

def sigmoid(a):
    return 1 / (1 + math.exp(-a))

# Input nodes
def n1(x1):
    return x1

def n2(x2):
    return x2

# Hidden layer nodes
def n3(w23, w13, x1, x2):
    return sigmoid(w23 * n2(x2) + w13 * n1(x1))

def n4(w14, w24, x1, x2):
    return sigmoid(w14 * n1(x1) + w24 * n2(x2))

# Output node
def n5(w35, w45, o3, o4):
    return sigmoid(w35 * o3 + w45 * o4)

# Wrapper for network output
def nnOutput(x1, x2):
    return n5(w35, w45, n3(w23, w13, x1, x2), n4(w14, w24, x1, x2))

# Initial weights
w13 = 1
w14 = 2.12
w23 = 4.66
w24 = 0.25
w35 = -0.98
w45 = 11.45

# Learning rate
l = 0.4

def updateWeights(x1, x2, correct_value):
    global w13, w14, w23, w24, w35, w45
    o3 = n3(w23, w13, x1, x2)
    o4 = n4(w14, w24, x1, x2)
    current_out = nnOutput(x1, x2)
    error = correct_value - current_out
    w35 = w35 + l * error * o3
    w45 = w45 + l * error * o4
    w13 = w13 + l * (w35 * error) * n1(x1)
    w14 = w14 + l * (w45 * error) * n1(x1)
    w23 = w23 + l * (w35 * error) * n2(x2)
    w24 = w24 + l * (w45 * error) * n2(x2)
```

```
# Training loop for XOR
for i in range(100):
    updateWeights(0, 0, 0)
    updateWeights(0, 1, 1)
    updateWeights(1, 0, 1)
    updateWeights(1, 1, 0)

# Final evaluation
print("Input(0,0) ->", nnOutput(0, 0)) # Target: ~0
print("Input(0,1) ->", nnOutput(0, 1)) # Target: ~1
print("Input(1,0) ->", nnOutput(1, 0)) # Target: ~1
print("Input(1,1) ->", nnOutput(1, 1)) # Target: ~0
```

Mathematica Example: Building and Training a Neural Network for Classification

Mathematica provides a high-level interface for constructing, training, and evaluating neural networks. Below is a pedagogical example using the classic Iris dataset for classification.

Step-by-Step Construction

1. **Load the Data:**

   ```
   trainingData = ExampleData[{"MachineLearning", "FisherIris"}, "TrainingData"];
   testData = ExampleData[{"MachineLearning", "FisherIris"}, "TestData"];
   ```

2. **Inspect the Data:**

   ```
   RandomSample[trainingData, 5]
   labels = Union[Values[trainingData]]
   ```

3. **Build the Neural Network as a Chain:**

   The network is constructed as a sequence (`NetChain`) of layers:

   - **LinearLayer:** Fully connected layer.
   - **ElementwiseLayer["ReLU"]:** Activation function.
   - **SoftmaxLayer:** Converts outputs to probabilities.
   - **NetDecoder:** Maps output indices to class labels.

   ```
   net = NetChain[{
       LinearLayer[10],
       ElementwiseLayer["ReLU"],
       LinearLayer[20],
       ElementwiseLayer["ReLU"],
       LinearLayer[3],
       SoftmaxLayer[]
     },
     "Input" -> 4,
     "Output" -> NetDecoder[{"Class", labels}]
   ]
   ```

4. **Train the Network:**

```
results = NetTrain[net, trainingData, All, ValidationSet -> testData];
trainedNet = results["TrainedNet"];
```

5. **Evaluate Performance:**

```
NetMeasurements[trainedNet, testData, "Accuracy"]
NetMeasurements[trainedNet, testData, "ConfusionMatrixPlot"]
```

6. **Make Predictions:**

```
trainedNet[{5.1, 3.5, 1.4, 0.2}]
```

Key Points

- **Layer Chaining:** `NetChain` allows you to build a neural network as a sequence of layers, mirroring the conceptual flow from input to output.
- **Activation and Output:** Nonlinear activations (ReLU) and softmax output are standard for classification.
- **Decoder:** `NetDecoder` maps numeric outputs to human-readable class labels.
- **Training and Evaluation:** `NetTrain` and `NetMeasurements` provide a complete workflow for supervised learning.

Summary: Mathematica's neural network framework enables rapid prototyping and experimentation, with clear chaining of layers and integrated training/evaluation tools.

Morphological Image Processing and Segmentation

Morphological image processing is a set of non-linear operations related to the shape or morphology of features in an image. These techniques are especially useful for binary images but can be extended to grayscale images.

Basic Morphological Operations

- **Erosion:** Removes pixels on object boundaries.
- **Dilation:** Adds pixels to object boundaries.
- **Opening:** Erosion followed by dilation; removes small objects.
- **Closing:** Dilation followed by erosion; fills small holes.

Mathematica Examples

Erosion:

```
Manipulate[Erosion[circuit, BoxMatrix[r]], {r, 1, 40, 1}]
Manipulate[Erosion[spheres, DiskMatrix[r]], {r, 1, 10, 1}]
Manipulate[Erosion[spheres, BoxMatrix[r]], {r, 1, 10, 1}]
Manipulate[Erosion[spheres, CrossMatrix[r]], {r, 1, 10, 1}]
Manipulate[Erosion[spheres, DiamondMatrix[r]], {r, 1, 10, 1}]
```

Dilation:

```
Manipulate[Dilation[text, BoxMatrix[r]], {r, 1, 10, 1}]
Dilation[text, ker1]
Dilation[text, ker2]
```

Opening:

```
Opening[triangle, DiskMatrix[9]]
Manipulate[Opening[triangle, DiskMatrix[r]], {r, 1, 10, 1}]
Manipulate[Opening[triangle, BoxMatrix[r]], {r, 1, 10, 1}]
Manipulate[Opening[wood, DiskMatrix[r]], {r, 1, 10, 1}]
```

Closing:

```
Manipulate[Closing[woodn, DiskMatrix[r]], {r, 1, 10, 1}]
Manipulate[Closing[rednose, DiamondMatrix[r]], {r, 1, 15, 1}]
Closing[crednose[[1]], DiamondMatrix[3]]
Closing[crednose[[2]], DiamondMatrix[3]]
Closing[1 - crednose[[2]], DiamondMatrix[3]]
```

Boundary Extraction

```
star1 = Erosion[star, CrossMatrix[1]];
star2 = star - star1;
```

Line Detection (Directional Kernels)

```
kh = {{-1, -1, -1}, {2, 2, 2}, {-1, -1, -1}}; (* horizontal *)
ih = ImageConvolve[circuit, kh];
Image[Rescale[ImageData[ih]]]
Binarize[Image[Rescale[ImageData[ih]]], 0.9]

k45 = {{2, -1, -1}, {-1, 2, -1}, {-1, -1, 2}}; (* 45 degrees *)
i45 = ImageConvolve[circuit, k45];
Manipulate[Binarize[Image[Rescale[ImageData[i45]]], r], {r, 0.72, 1}]
```

```
kv = {{-1, 2, -1}, {-1, 2, -1}, {-1, 2, -1}}; (* vertical *)
iv = ImageConvolve[circuit, kv];
Manipulate[Binarize[Image[Rescale[ImageData[iv]]], r], {r, 0.72, 1}]
```

Edge Detection (Gradient Operators)

```
s1 = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}}; (* vertical Sobel *)
s2 = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}}; (* horizontal Sobel *)
building1 = ImageConvolve[building, s1];
building2 = ImageConvolve[building, s2];
A1 = ImageData[building1];
A2 = ImageData[building2];
A3 = Sqrt[A1^2 + A2^2];
Image[A3]
```

K-Means Clustering for Segmentation

```
i1 = ClusteringComponents[i, 5, Method -> "KMeans"];
Colorize[i1]
j1 = ClusteringComponents[j, 4, Method -> "KMeans"];
Colorize[j1, ColorRules -> {1 -> Blue, 2 -> Green, 3 -> White, 4 -> Black}]
```

Region Growing Segmentation

```
Manipulate[RegionBinarize[scan, {{199.109375, 100.88671875}}, t], {t, 0.05, 0.1, 0.01}]
RegionBinarize[leaf, {{210.203125, 141.142578}, {192.800781, 118.919922}, {207.857422, 90.773438}, {
RegionBinarize[leaf, {{202.40625, 34.914062}}, 0.3, {{229.335938, 161.509766}}]
```

Watershed Segmentation

```
cg = ColorConvert[coins, "Grayscale"];
cb = Binarize[cg, 0.6];
cn = ColorNegate[cb];
dt = ImageAdjust[DistanceTransform[cn]];
dtn = ColorNegate[dt];
wc = WatershedComponents[dtn];
Colorize[wc]
wcb = Image[wc, "Bit"];
ImageMultiply[wcb, cn]
```

Summary: Morphological operations are essential for shape-based image analysis, segmentation, and feature extraction. Mathematica provides built-in functions for these tasks, enabling interactive exploration and visualization.