Project – Documentation

Mohamed Hany
223101271
Image Processing
DR. Mahmoud Zaki

1. Introduction

This project implements manual Canny Edge Detection using Python, NumPy, and OpenCV. Additional edge detection techniques including Sobel and Laplacian are also implemented. A real-time webcam application allows switching between different filtering modes.

2. Project Components

- Gaussian Blur (Noise Reduction)
- Sobel Gradient Calculation (Edge Direction + Strength)
- Non-Maximum Suppression (Edge Thinning)
- Double Thresholding (Strong vs Weak Edges)
- Hysteresis Tracking (Edge Connectivity)
- Full Custom Canny Function
- Sobel & Laplacian
- Mean Filters (Restoration):
  - Arithmetic, Geometric, and Harmonic
  - Contraharmonic
- Order-Statistics Filters:
  - Median & Alpha-Trimmed
  - Min, Max, and Midpoint
- Ideal, Gaussian, and Butterworth Filters
- Band-Reject & Band-Pass Filters
- Real-Time Webcam

3. Gaussian Blur

The Gaussian blur function smooths the image and reduces noise. This is important because noise can create false edges.

```python
def gaussian_blur(img, ksize=3, sigma=1):
    k = make_odd(max(3, ksize))
    return cv2.GaussianBlur(img, (k, k), sigma)
```

The function uses a kernel size (ksize) and standard deviation (sigma) to control the amount of smoothing.

4. Gradient Computation

The Sobel operator computes gradients in the X and Y directions. These gradients indicate how intensity changes across

the image. The magnitude and direction of the gradient are computed to understand edge strength and orientation.

```python
def compute_gradients(img, ksize=3):
    img = img.astype(np.float64)
    k = make_odd(max(3, ksize))
    grad_x = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=k)
    grad_y = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=k)

    grad_mag = np.sqrt(grad_x**2 + grad_y**2)
    mmax = grad_mag.max()
    grad_mag = (grad_mag / mmax * 255) if mmax > 1e-6 else np.zeros_like(grad_mag)

    grad_dir = np.arctan2(grad_y, grad_x)
    return grad_mag, grad_dir
```

5. Non-Maximum Suppression

After the gradient magnitude is computed, the edges still appear thick because many neighboring pixels have high gradient values.
Therefore, Non-Maximum Suppression (NMS) is applied to thin the edges and keep only the strongest edge pixel across the edge direction.

```python
def non_maximum_suppression(grad_mag, grad_dir):
    M, N = grad_mag.shape
    Z = np.zeros((M, N), dtype=np.float64)
    angle = grad_dir * 180.0 / np.pi
    angle[angle < 0] += 180

    for i in range(1, M-1):
        for j in range(1, N-1):
            q, r = 0, 0
            if (0 <= angle[i, j] < 22.5) or (157.5 <= angle[i, j] <= 180):
                q = grad_mag[i, j+1]; r = grad_mag[i, j-1]
            elif (22.5 <= angle[i, j] < 67.5):
                q = grad_mag[i+1, j-1]; r = grad_mag[i-1, j+1]
            elif (67.5 <= angle[i, j] < 112.5):
                q = grad_mag[i+1, j]; r = grad_mag[i-1, j]
            elif (112.5 <= angle[i, j] < 157.5):
                q = grad_mag[i-1, j-1]; r = grad_mag[i+1, j+1]

            Z[i, j] = grad_mag[i, j] if (grad_mag[i, j] >= q and grad_mag[i, j] >= r) else 0

    return Z
```

How is NMS performed?

For each pixel:

1. Determine the edge direction using the gradient angle.
2. Select the two neighboring pixels **along the same gradient direction**.
3. Compare the gradient magnitude of the current pixel with its two neighbors:

- If the current pixel has a **greater or equal** magnitude → it is kept
- If it is **smaller** → it is suppressed (set to zero)

6. Double Thresholding

Classifies pixels into: strong edges, weak edges, and non-edges based on low and high thresholds. Strong edges are kept, weak edges are marked for potential inclusion, and non-edges are discarded.

```python
def double_threshold(img, low_thresh, high_thresh):
    strong, weak = 255, 75
    res = np.zeros_like(img, dtype=np.uint8)

    strong_i, strong_j = np.where(img >= high_thresh)
    weak_i, weak_j = np.where((img >= low_thresh) & (img < high_thresh))

    res[strong_i, strong_j] = strong
    res[weak_i, weak_j] = weak
    return res, weak, strong
```

7. Hysteresis Tracking

Hysteresis determines which weak edges are true edges. A weak edge becomes a strong edge only if it is connected to a strong edge. This eliminates noise while preserving meaningful edges.

```python
def hysteresis(img, weak=75, strong=255):
    M, N = img.shape
    res = img.copy()
    for i in range(1, M-1):
        for j in range(1, N-1):
            if res[i, j] == weak:
                if np.any(res[i-1:i+2, j-1:j+2] == strong):
                    res[i, j] = strong
                else:
                    res[i, j] = 0
    return res
```

8. Custom Canny Function

Gaussian Blur → Gradient → NMS → Threshold → Hysteresis → Final Edges

```python
def my_canny(img_gray, low_thresh=20, high_thresh=60, ksize=3, sigma=1):
    blurred = gaussian_blur(img_gray, ksize, sigma)
    mag, direction = compute_gradients(blurred, ksize=ksize)
    nms = non_maximum_suppression(mag, direction)
    dt, weak, strong = double_threshold(nms, low_thresh, high_thresh)
    edges = hysteresis(dt, weak=weak, strong=strong)
    return edges.astype(np.uint8)
```

9. Sobel & Laplacian Methods

Sobel: directional edges (X/Y)

```python
def sobel_x(img_gray):
    gx = cv2.Sobel(img_gray, cv2.CV_64F, 1, 0, ksize=3)
    gx = np.absolute(gx)
    m = gx.max()
    gx = (gx / m * 255) if m > 1e-6 else np.zeros_like(img_gray, dtype=np.float64)
    return gx.astype(np.uint8)

def sobel_y(img_gray):
    gy = cv2.Sobel(img_gray, cv2.CV_64F, 0, 1, ksize=3)
    gy = np.absolute(gy)
    m = gy.max()
    gy = (gy / m * 255) if m > 1e-6 else np.zeros_like(img_gray, dtype=np.float64)
    return gy.astype(np.uint8)

def sobel_edges(img_gray):
    gx = cv2.Sobel(img_gray, cv2.CV_64F, 1, 0, ksize=3)
    gy = cv2.Sobel(img_gray, cv2.CV_64F, 0, 1, ksize=3)
    mag = np.sqrt(gx**2 + gy**2)
    m = mag.max()
    mag = (mag / m * 255) if m > 1e-6 else np.zeros_like(img_gray, dtype=np.float64)
    return mag.astype(np.uint8)
```

Laplacian: detects edges in all directions (but noise-sensitive)

```python
def laplacian_filter2d(img_gray):
    kernel = np.array([[0, 1, 0],
                       [1, -4, 1],
                       [0, 1, 0]], dtype=np.float32)
    lap = cv2.filter2D(img_gray, cv2.CV_64F, kernel)
    lap = np.absolute(lap)
    lap = np.clip(lap, 0, 255)
    return lap.astype(np.uint8)
```

10.Mean Filters :

Arithmetic, Geometric, and Contraharmonic means for restoring images from various noise types.

```python
def arithmetic_mean_filter(img_gray, ksize=3):
    k = make_odd(max(3, ksize))
    return cv2.blur(img_gray, (k, k))

def geometric_mean_filter(img_gray, ksize=3):
    k = make_odd(max(3, ksize))
    eps = 1e-6
    g = img_gray.astype(np.float32)
    logg = np.log(g + eps)
    m = cv2.blur(logg, (k, k))
    out = np.exp(m)
    return np.clip(out, 0, 255).astype(np.uint8)

def harmonic_mean_filter(img_gray, ksize=3):
    k = make_odd(max(3, ksize))
    eps = 1e-6
    g = img_gray.astype(np.float32)
    inv = 1.0 / (g + eps)
    s = cv2.blur(inv, (k, k))  # mean of inv
    mn = float(k * k)
    out = mn / (s * mn + eps)
    return np.clip(out, 0, 255).astype(np.uint8)

def contraharmonic_mean_filter(img_gray, ksize=3, q=1.5):
    k = make_odd(max(3, ksize))
    eps = 1e-6
    g = img_gray.astype(np.float32)

    gq = np.power(g + eps, q)
    gq1 = np.power(g + eps, q + 1.0)

    mean_gq = cv2.blur(gq, (k, k))
    mean_gq1 = cv2.blur(gq1, (k, k))

    out = mean_gq1 / (mean_gq + eps)
    return np.clip(out, 0, 255).astype(np.uint8)
```

12 .Order Statistics:

Median, Min, Max, and Alpha-trimmed mean filters for robust noise removal.

```python
def median_filter(img_gray, ksize=3):
    k = make_odd(max(3, ksize))
    return cv2.medianBlur(img_gray, k)

def min_filter(img_gray, ksize=3):
    k = make_odd(max(3, ksize))
    kernel = np.ones((k, k), np.uint8)
    return cv2.erode(img_gray, kernel, iterations=1)

def max_filter(img_gray, ksize=3):
    k = make_odd(max(3, ksize))
    kernel = np.ones((k, k), np.uint8)
    return cv2.dilate(img_gray, kernel, iterations=1)

def midpoint_filter(img_gray, ksize=3):
    mn = min_filter(img_gray, ksize=ksize).astype(np.float32)
    mx = max_filter(img_gray, ksize=ksize).astype(np.float32)
    out = (mn + mx) / 2.0
    return np.clip(out, 0, 255).astype(np.uint8)
```

```python
def alpha_trimmed_mean_filter(img_gray, ksize=3, d=2):
    k = make_odd(max(3, ksize))
    n = k * k
    d = int(max(0, d))
    if d % 2 == 1:
        d += 1
    if d >= n:
        d = n - 1
        if d % 2 == 1:
            d -= 1

    pad = k // 2
    g = img_gray.astype(np.float32)
    gp = np.pad(g, ((pad, pad), (pad, pad)), mode='edge')

    try:
        from numpy.lib.stride_tricks import sliding_window_view
        win = sliding_window_view(gp, (k, k))
        win = win.reshape(win.shape[0], win.shape[1], n)
        win.sort(axis=2)
        lo = d // 2
        hi = n - (d // 2)
        trimmed = win[:, :, lo:hi]
        out = trimmed.mean(axis=2)
        return np.clip(out, 0, 255).astype(np.uint8)
    except Exception:
        h, w = img_gray.shape
        out = np.zeros((h, w), np.float32)
        lo = d // 2
        hi = n - (d // 2)
        for i in range(h):
            for j in range(w):
                block = gp[i:i + k, j:j + k].reshape(-1)
                block.sort()
                out[i, j] = block[lo:hi].mean()
        return np.clip(out, 0, 255).astype(np.uint8)
```

13. Frequency Domain Filtering

```python
def _distance_grid(rows, cols):
    crow, ccol = rows // 2, cols // 2
    y, x = np.ogrid[:rows, :cols]
    D = np.sqrt((y - crow) ** 2 + (x - ccol) ** 2)
    return D

def _lp_mask(rows, cols, filter_name, D0, n=2, W=10):
    D = _distance_grid(rows, cols).astype(np.float32)
    D0 = max(1.0, float(D0))
    n = max(1, int(n))
    W = max(1.0, float(W))

    if filter_name == 'ILPF':
        H = (D <= D0).astype(np.float32)
    elif filter_name == 'GLPF':
        H = np.exp(-(D ** 2) / (2 * (D0 ** 2))).astype(np.float32)
    elif filter_name == 'BLPF':
        H = (1.0 / (1.0 + (D / D0) ** (2 * n))).astype(np.float32)

    elif filter_name == 'IHPF':
        H = (D > D0).astype(np.float32)
    elif filter_name == 'GHPF':
        H = (1.0 - np.exp(-(D ** 2) / (2 * (D0 ** 2)))).astype(np.float32)
    elif filter_name == 'BHPF':
        H = (1.0 - (1.0 / (1.0 + (D / D0) ** (2 * n)))).astype(np.float32)

    else:
        if filter_name == 'IBRF':
            H = np.ones((rows, cols), np.float32)
            H[np.abs(D - D0) <= (W / 2.0)] = 0.0
        elif filter_name == 'IBPF':
            H = np.zeros((rows, cols), np.float32)
            H[np.abs(D - D0) <= (W / 2.0)] = 1.0

        elif filter_name == 'GBRF':
            eps = 1e-6
            term = ((D ** 2 - D0 ** 2) / (np.maximum(D, eps) * W)) ** 2
            H = (1.0 - np.exp(-term)).astype(np.float32)
        elif filter_name == 'GBPF':
            eps = 1e-6
            term = ((D ** 2 - D0 ** 2) / (np.maximum(D, eps) * W)) ** 2
            H = (np.exp(-term)).astype(np.float32)

        elif filter_name == 'BBRF':
            eps = 1e-6
            denom = 1.0 + ((D * W) / (np.maximum(np.abs(D ** 2 - D0 ** 2), eps))) ** (2 * n)
            H = (1.0 / denom).astype(np.float32)
        elif filter_name == 'BBPF':
            eps = 1e-6
            denom = 1.0 + ((D * W) / (np.maximum(np.abs(D ** 2 - D0 ** 2), eps))) ** (2 * n)
            H = (1.0 - (1.0 / denom)).astype(np.float32)
        else:
            H = np.ones((rows, cols), np.float32)

    return H
```

```python
def apply_freq_filter_gray(img_gray, filter_name='ILPF', D0=30, n=2, W=10):
    img = img_gray.astype(np.float32)
    rows, cols = img.shape

    dft = cv2.dft(img, flags=cv2.DFT_COMPLEX_OUTPUT)
    dft_shift = np.fft.fftshift(dft)

    H = _lp_mask(rows, cols, filter_name, D0=D0, n=n, W=W)
    H2 = np.repeat(H[:, :, np.newaxis], 2, axis=2)

    fshift = dft_shift * H2

    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])

    out = cv2.normalize(img_back, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

    mask_vis = (H * 255).astype(np.uint8)

    mag = cv2.magnitude(fshift[:, :, 0], fshift[:, :, 1])
    spec = np.log(mag + 1.0)
    spec = cv2.normalize(spec, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

    return out, mask_vis, spec
```

14.Key Functions

View / Processing Modes

- o Original Image
- g Grayscale Image
- b Gaussian Blur Only
- c Custom Canny Edge Detection
- s Sobel Gradient Magnitude
- x Sobel X (Horizontal Edges)
- y Sobel Y (Vertical Edges)
- l Laplacian Edge Detection

Mean Filters

- e Arithmetic Mean Filter
- r Geometric Mean Filter
- h Harmonic Mean Filter
- v Contraharmonic Mean Filter

Order Statistics Filters

- m Median Filter
- , Min Filter
- . Max Filter
- / Midpoint Filter
- ; Alpha-Trimmed Mean Filter

Frequency Domain Filters

- 1 Ideal Low-Pass Filter (ILPF)
- 2 Gaussian Low-Pass Filter (GLPF)
- 3 Butterworth Low-Pass Filter (BLPF)
- 4 Ideal High-Pass Filter (IHPF)
- 5 Gaussian High-Pass Filter (GHPF)
- 6 Butterworth High-Pass Filter (BHPF)
- 7 Ideal Band-Reject Filter (IBRF)
- 8 Gaussian Band-Reject Filter (GBRF)
- 9 Butterworth Band-Reject Filter (BBRF)
- 0 Ideal Band-Pass Filter (IBPF)

- Gaussian Band-Pass Filter (GBPF)
= Butterworth Band-Pass Filter (BBPF)

Controls

p Increase Kernel Size (Spatial)
m Decrease Kernel Size

i Increase High Threshold (Canny)
k Decrease High Threshold

j Increase Low Threshold (Canny)
n Decrease Low Threshold

z Increase Cutoff Frequency D0
a Decrease Cutoff Frequency D0

u Increase Band Width (W)
d Decrease Band Width (W)

r Increase Butterworth Order (n)
f Decrease Butterworth Order

[Decrease Contraharmonic Q
] Increase Contraharmonic Q

\ Decrease Alpha-Trimmed d
' Increase Alpha-Trimmed d

Exit

q Quit Program

mode: gray | ksize: 3 | low: 20 | high:

60 Comment:

- **This is just the original grayscale image without edge detection.**
- **Used as a reference to visually compare edge extraction results.**

Conclusion: No edge highlights yet — baseline input.

1. Results (Insert Your Images Below)

Image 1
mode: c | ksize: 3 | low: 20 | high: 60

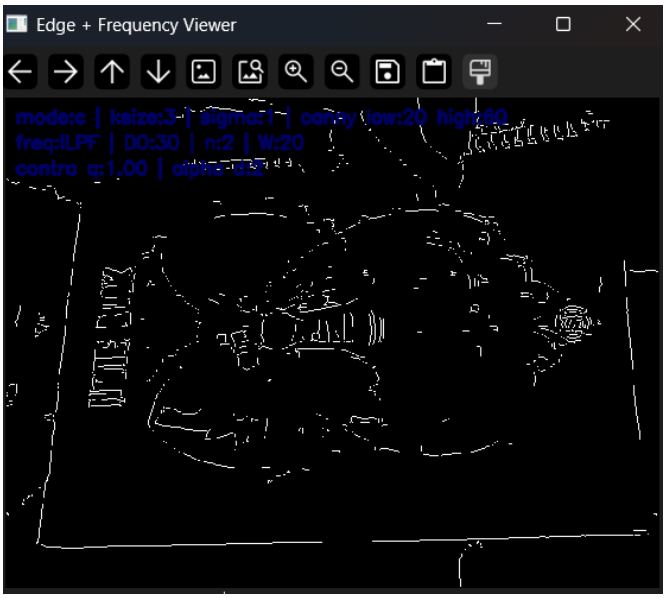
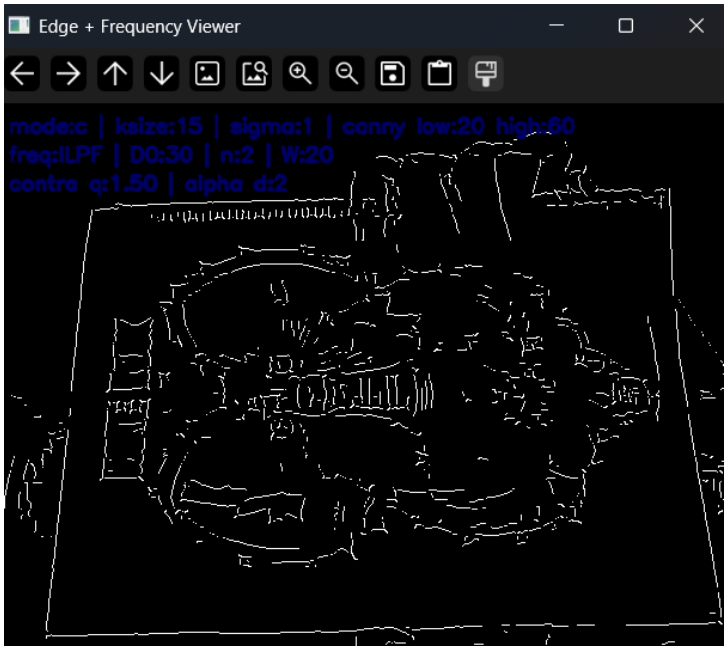


Image 2

mode: c | ksize: 15 | low: 20 | high: 60



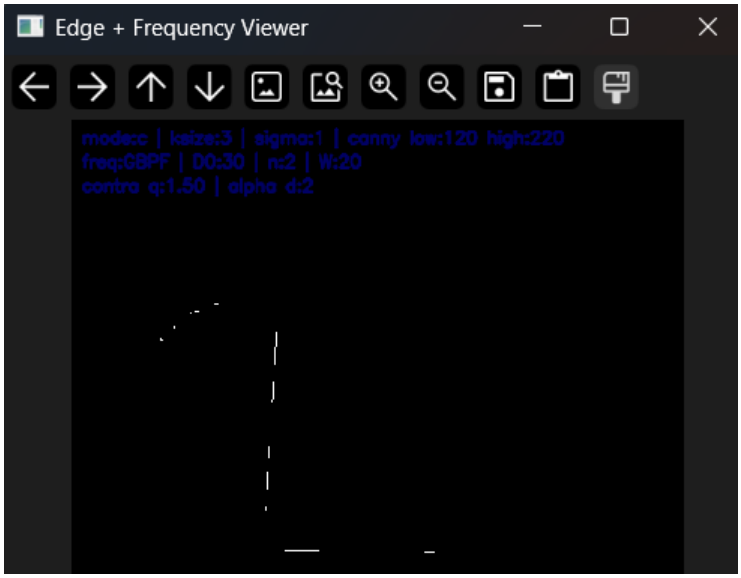
Comment:

- The **Gaussian kernel size is very large (15)** → strong blur removes a lot of important details.
- Low thresholds (20/60) → too many weak edges detected → noise appears.
- Result: **very weak and broken edges**, text is hardly visible, most features are smoothed out.

Conclusion: Increasing ksize too much causes the loss of fine edges.

Image 3

mode: c | ksize: 3 | low: 120 | high: 220

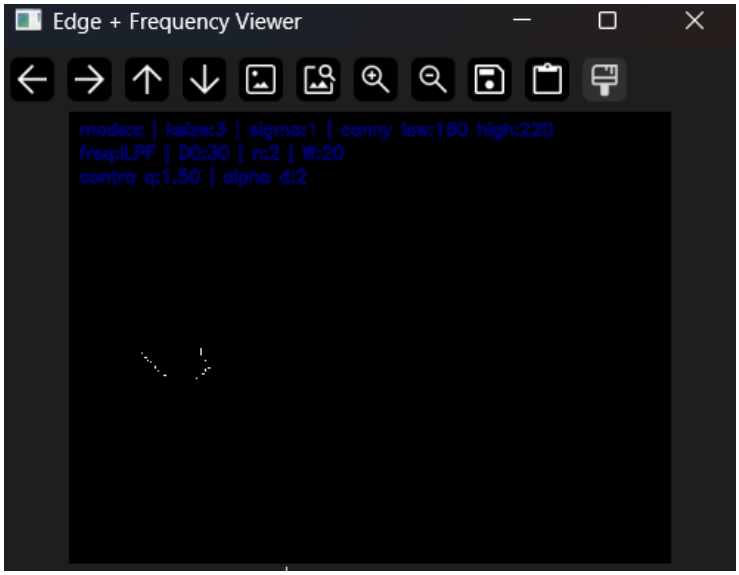


Comment:

- Small blur (ksize=3) keeps details intact.
- High thresholds → only **strong edges** are detected.
- Most thin text and weak features are not detected.

Conclusion: High threshold values reduce noise but **miss many important edges.**

Image 4



mode: c | ksize: 3 | low: 160 | high: 220

Comment:

- Same blur strength as image 2.
- Increasing the low threshold even more → more edges are rejected.
- Result: fewer text edges remain, weaker features disappear completely.

Conclusion: When low threshold increases, Canny becomes more selective and detects **only the strongest edges.**

Image 5

mode: e

Comment (<u>Arithmetic Mean Filter</u> Mode):

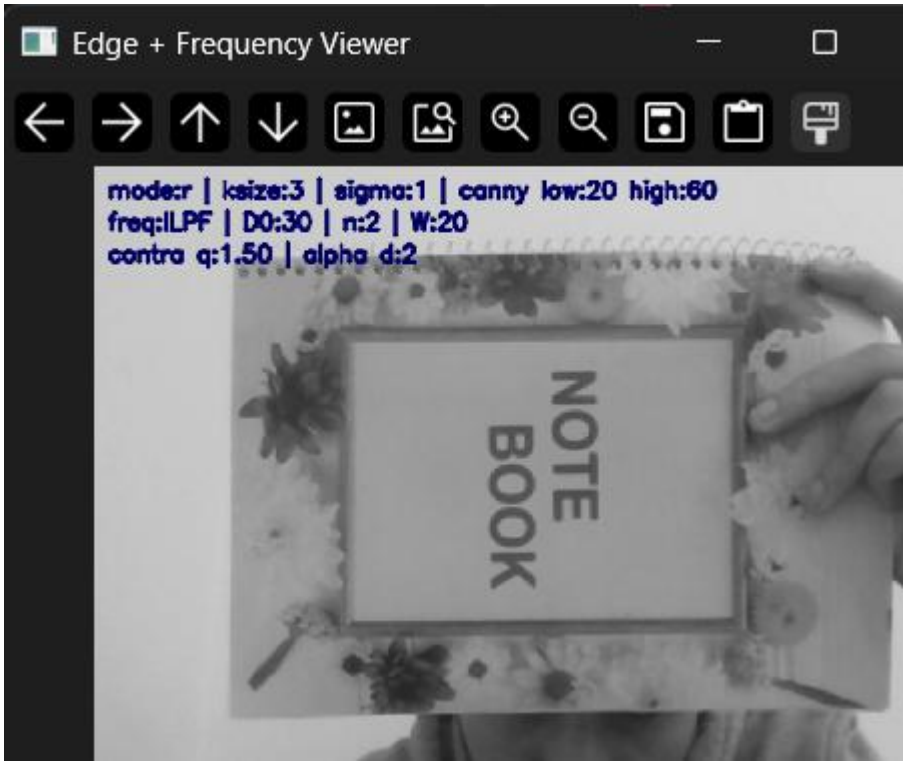
The arithmetic mean filter smooths the image by averaging pixel values within the kernel. As shown, random noise and small intensity variations are reduced, producing a cleaner and more uniform appearance. However, this smoothing comes at the cost of **blurring edges and fine details**, making object boundaries less sharp. This filter is effective for reducing **Gaussian-like noise**, but it is not suitable when edge preservation is critical.

mode: e | ksize = 11

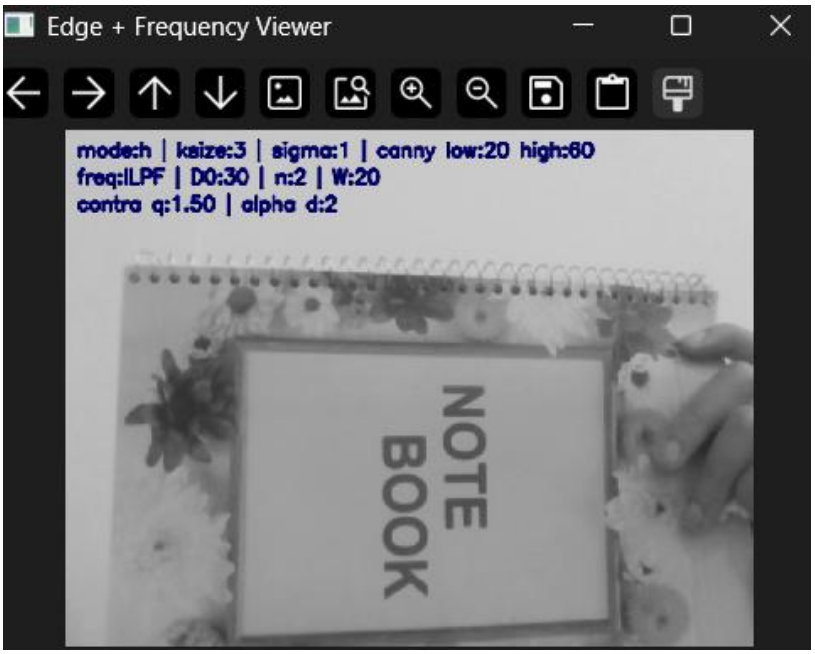
Increasing the kernel size (ksize) increases the amount of blur, which reduces noise but also smooths edges and fine details.

mode r | ksize = 3

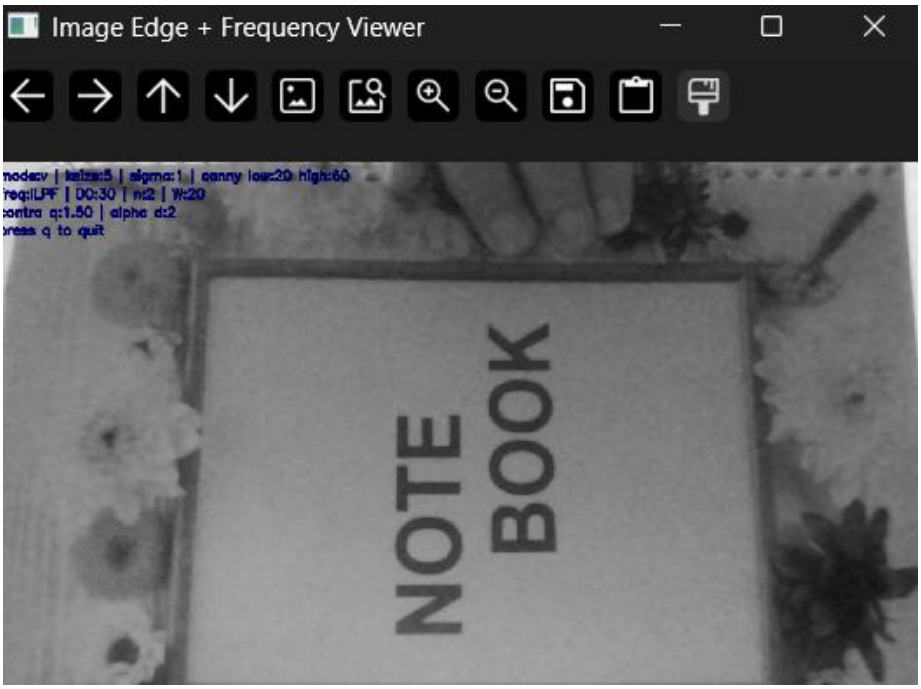
This figure shows the result of applying the geometric mean filter with ksize = 3. The filter smooths the image by reducing noise while preserving edges better than the arithmetic mean filter. Fine details are slightly softened, but important structures remain visible, demonstrating the balance between noise reduction and edge preservation provided by the geometric mean filter.



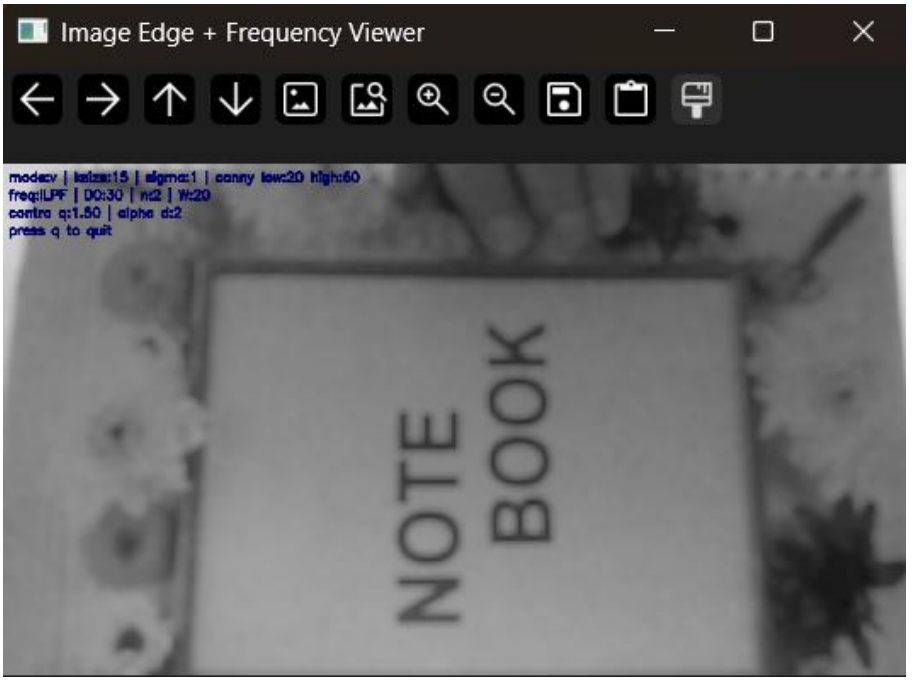
This figure shows the output of the harmonic mean filter with kernel size $k = 3$. The filter is effective in reducing salt noise by giving less weight to large pixel values. As a result, bright noise is suppressed and the image appears smoother, while edges are preserved better than with the arithmetic mean filter.



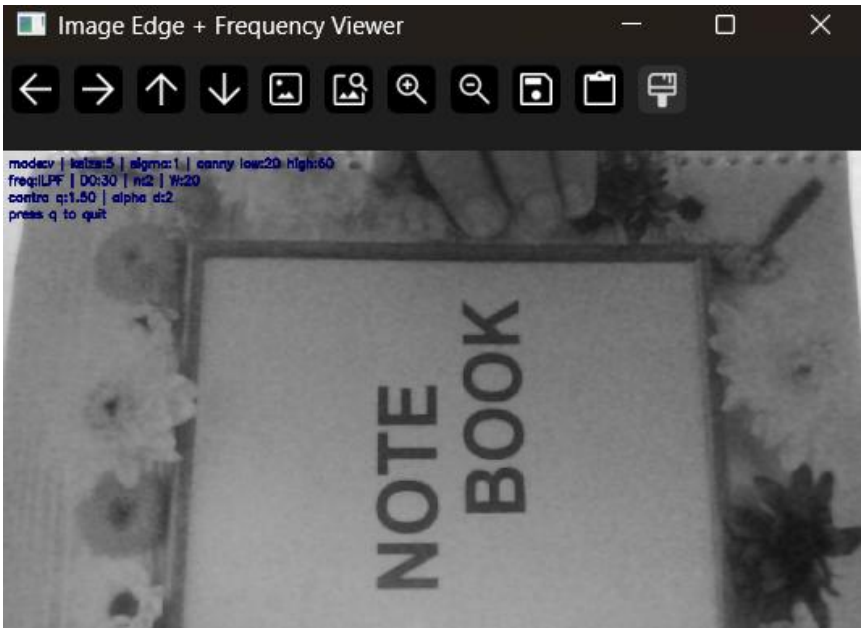
This figure shows the result of applying the contraharmonic mean filter with kernel size $k = 5$. The filter effectively reduces impulse noise depending on the value of the parameter q. With a larger kernel size, noise suppression is increased, producing a smoother image while causing some loss of fine details and slight edge blurring.



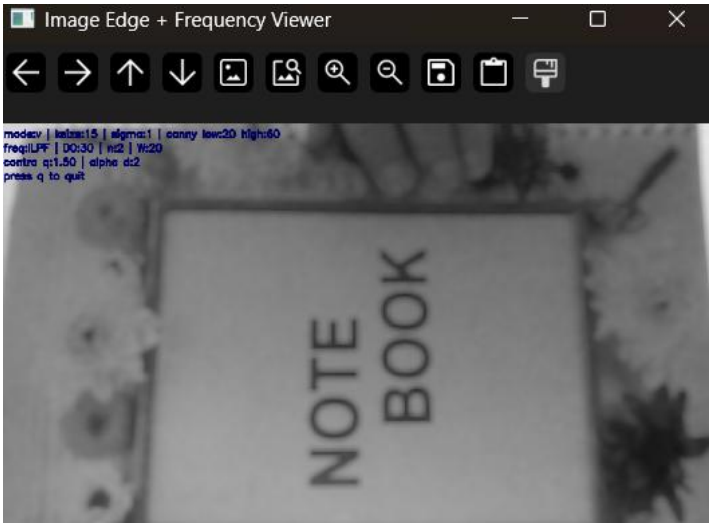
This figure shows the effect of the **contraharmonic mean** filter with a large kernel size ($k = 15$). Increasing the kernel size significantly increases the amount of smoothing, leading to strong noise reduction. However, this also causes heavy blurring and loss of edges and fine details, illustrating the trade-off between noise suppression and image sharpness when using large neighborhood sizes.



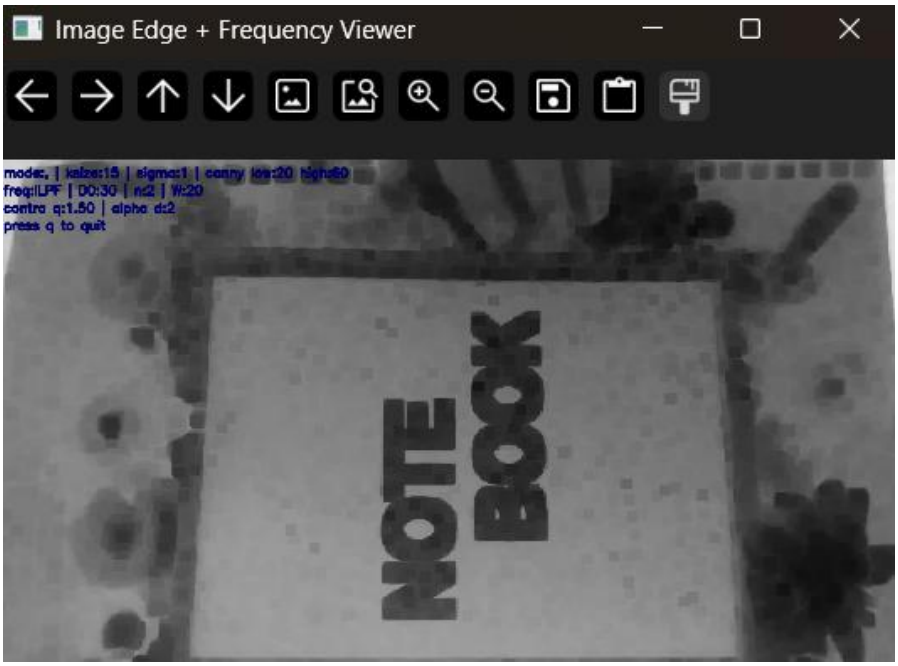
This figure shows the result of applying the **median** filter with kernel size $k = 5$. The median filter effectively removes impulse (salt-and-pepper) noise while preserving edges better than mean-based filters. With a moderate kernel size, noise is reduced without excessive blurring, and important edges and structures remain relatively sharp.



This figure shows the effect of the **median** filter with a large kernel size ($k = 15$). Increasing the kernel size greatly enhances noise removal; however, it also introduces strong smoothing. As a result, edges become less sharp and fine details are lost, demonstrating how excessive kernel sizes can degrade image details even with edge-preserving filters like the median filter.

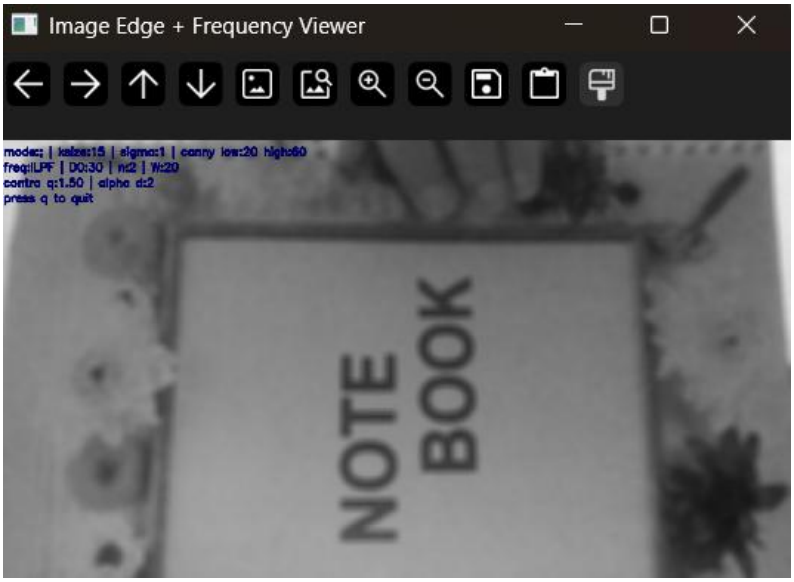


This figure shows the output of the **minimum (erosion)** filter with a large kernel size ($k = 15$). The filter replaces each pixel with the minimum value in its neighborhood, which effectively suppresses bright (salt) noise. However, using a large kernel causes dark regions to expand and edges to shrink, resulting in loss of detail and distortion of object boundaries.

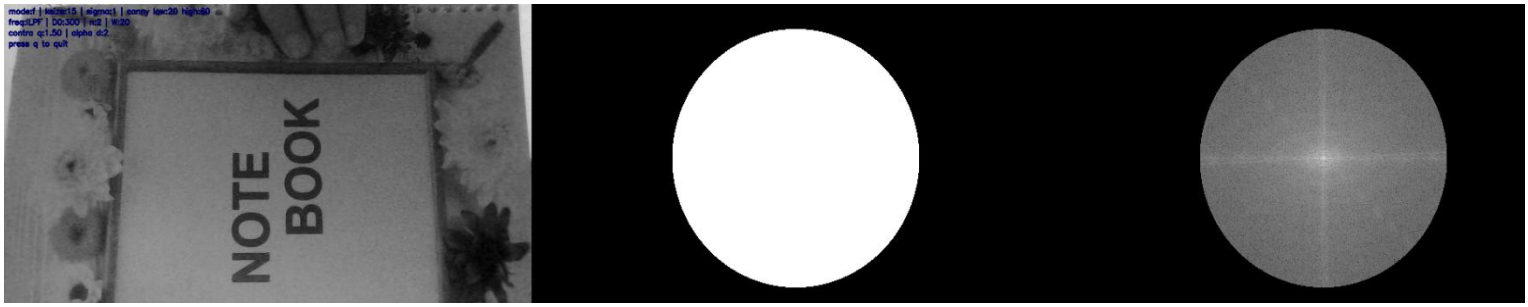


This figure shows the result of the midpoint filter with a large kernel size ($k = 15$). The **midpoint filter** computes the average of the minimum and maximum values within the neighborhood. With a large kernel, noise is reduced; however, strong smoothing occurs, leading to blurred edges and loss of fine details. This demonstrates that increasing the kernel size increases smoothing but reduces edge sharpness.

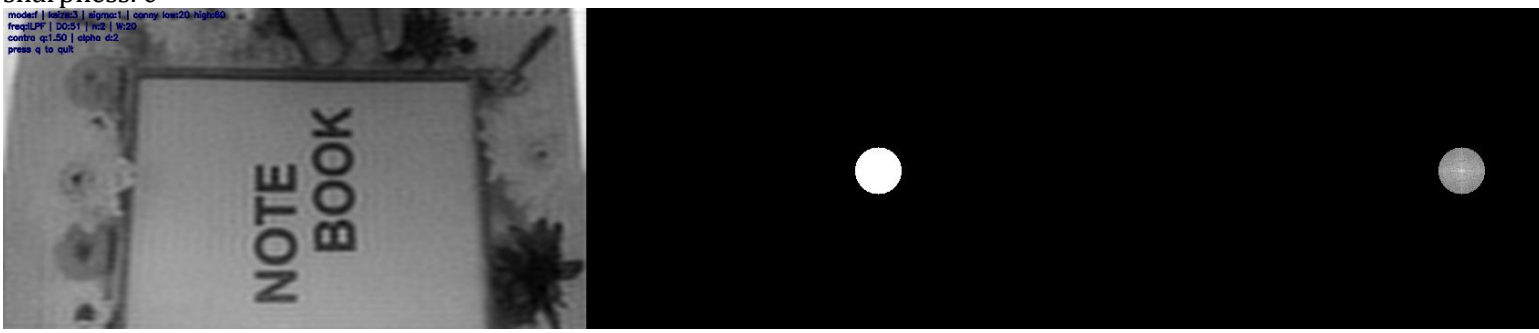
This figure shows the output of the **alpha-trimmed mean** filter with a large kernel size ($k = 15$). The filter removes extreme pixel values before averaging, which helps reduce impulse noise. With a large kernel, noise suppression is strong, but excessive trimming and averaging lead to significant blurring and loss of fine details, illustrating the trade-off between robustness to noise and edge preservation.



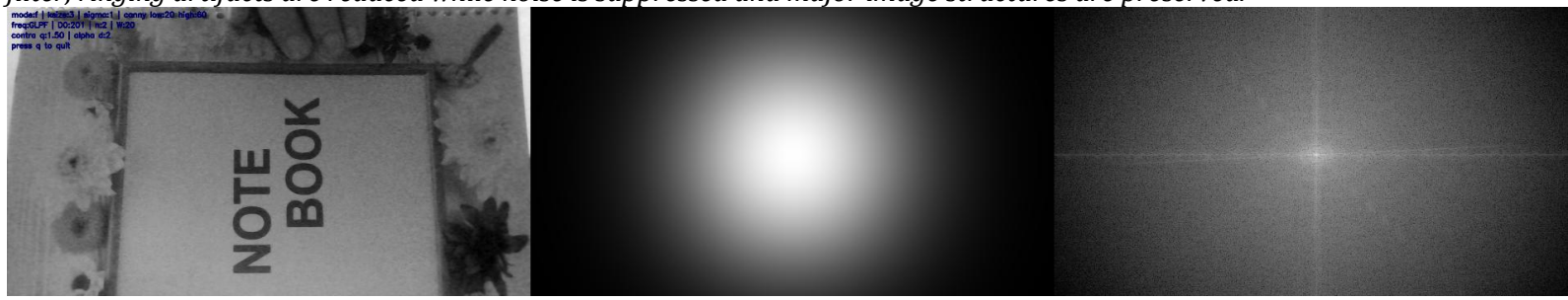
This figure shows the effect of an ***Ideal Low-Pass Filter (ILPF)*** with cutoff frequency $D_0 = 300$. A large D_0allows most low and mid-frequency components to pass, so the filtered image remains close to the original with slight smoothing. High-frequency components (noise and sharp edges) are weakly attenuated, illustrating that increasing D_0reduces the amount of blurring and preserves more image details.



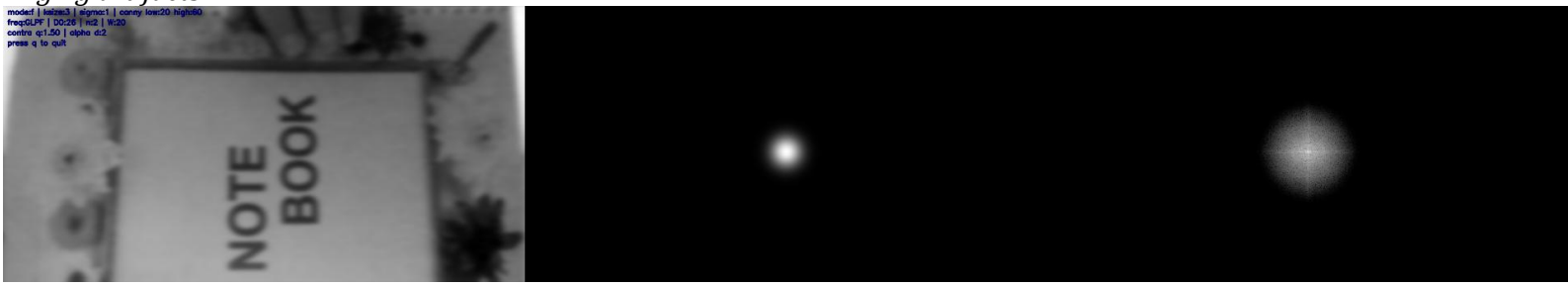
This figure shows the effect of an **Ideal Low-Pass Filter (ILPF)** with a small cutoff frequency $\boldsymbol{D_0 = 51}$. Only very low-frequency components are passed, resulting in strong blurring of the image. High-frequency details such as edges and fine textures are heavily removed, demonstrating that decreasing D_0increases smoothing and significantly reduces image sharpness. c



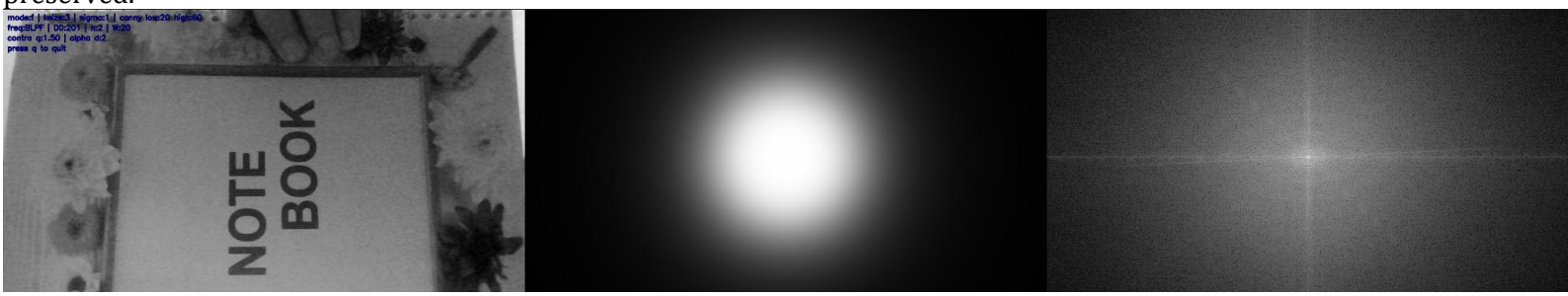
This figure shows the effect of a Gaussian Low-Pass Filter (GLPF) with cutoff frequency $D_0 = 201$. The Gaussian filter provides smooth frequency attenuation without sharp cutoffs, resulting in gradual blurring of the image. Compared to the ideal low-pass filter, ringing artifacts are reduced while noise is suppressed and major image structures are preserved.



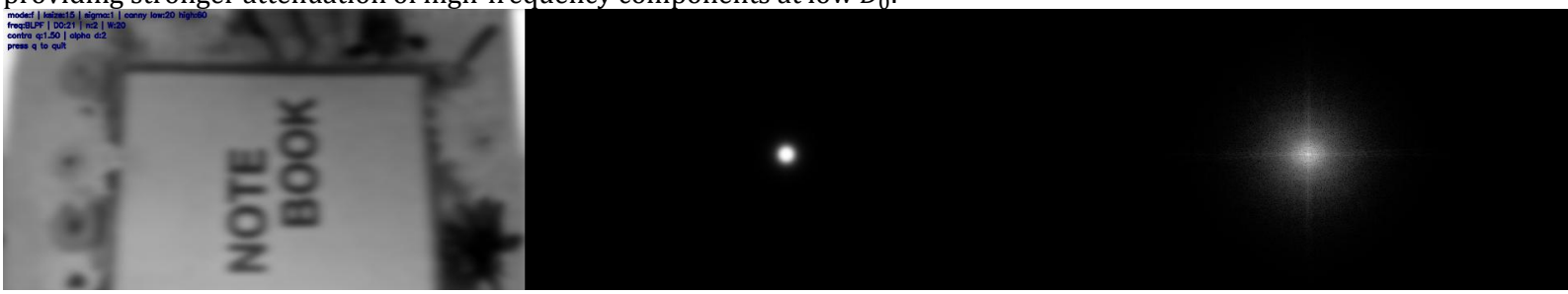
This figure shows the effect of a Gaussian Low-Pass Filter (GLPF) with a small cutoff frequency $D_0 = 26$. Only very low-frequency components are preserved, resulting in strong smoothing and heavy blurring of the image. Edges and fine details are significantly attenuated, while the Gaussian shape of the filter ensures a smooth transition in the frequency domain without sharp cutoffs or ringing artifacts.



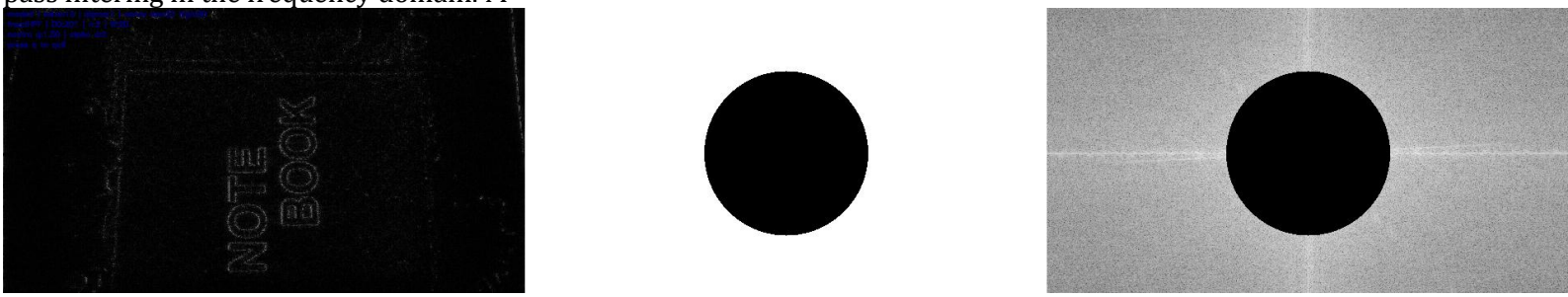
This figure shows the result of applying a Butterworth Low-Pass Filter (BLPF) with cutoff frequency $D_0 = 201$and order $n = 2$. The filter provides a smoother transition than the ideal low-pass filter while maintaining a sharper cutoff than the Gaussian filter. As a result, moderate smoothing is achieved with reduced ringing artifacts, and most of the image's main structures are preserved.



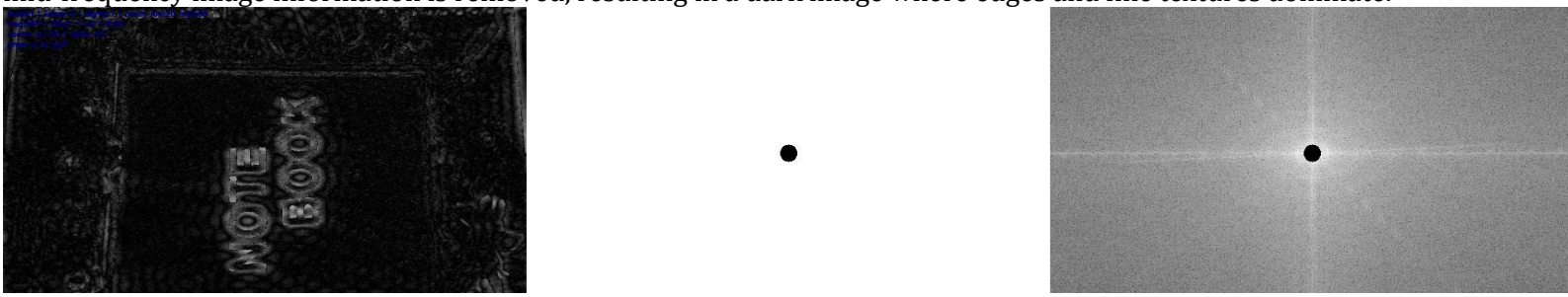
This figure illustrates the effect of a Butterworth Low-Pass Filter (BLPF) with a small cutoff frequency $D_0 = 21$and order $n = 2$. Only very low-frequency components are passed, resulting in strong blurring and significant loss of edges and fine details. Compared to the Gaussian filter, the Butterworth filter maintains a smoother transition than the ideal filter while still providing stronger attenuation of high-frequency components at low D_0.



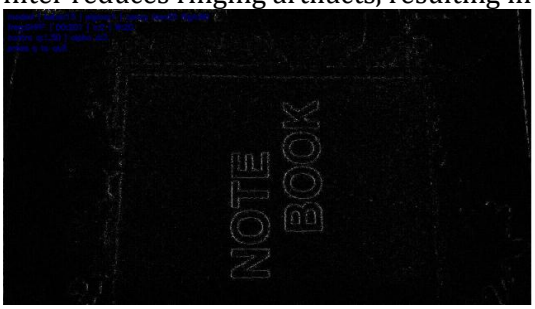
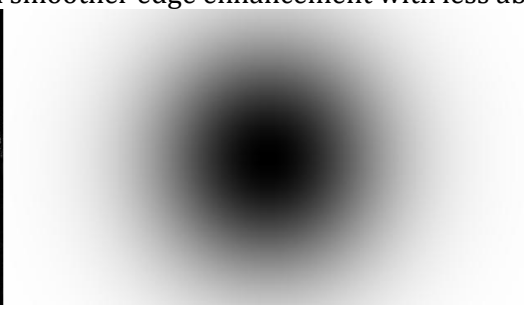
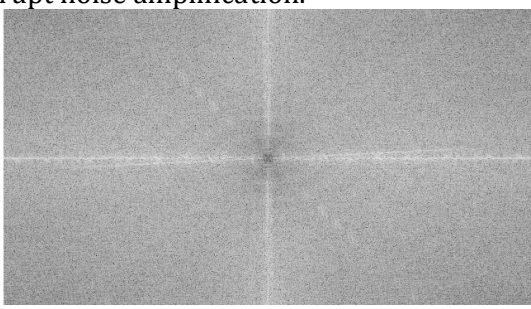
This figure shows the result of applying an Ideal High-Pass Filter (IHPF) with cutoff frequency $D_0 = 201$. Low-frequency components (smooth regions and background) are removed, while high-frequency components such as edges and fine details are preserved. This enhances edges and textures, but also amplifies noise, illustrating the edge-enhancement nature of high-pass filtering in the frequency domain. M



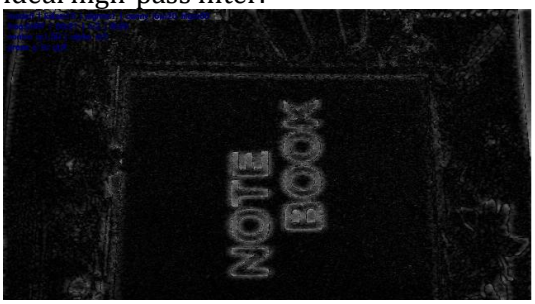
This figure shows the effect of an Ideal High-Pass Filter (IHPF) with a small cutoff frequency $D_0 = 21$. Only very high-frequency components are passed, leading to strong edge enhancement and significant amplification of noise. Most low- and mid-frequency image information is removed, resulting in a dark image where edges and fine textures dominate.

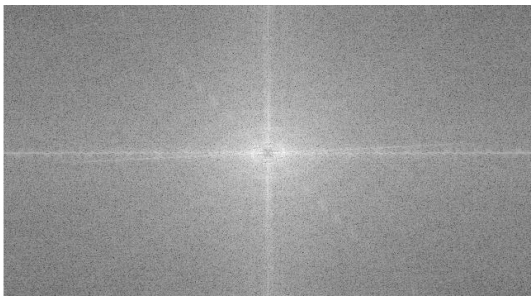


This figure shows the effect of a Gaussian High-Pass Filter (GHPF) with cutoff frequency $D_0 = 201$. Low-frequency components are smoothly attenuated while high-frequency components are preserved. Compared to the ideal high-pass filter, the Gaussian filter reduces ringing artifacts, resulting in smoother edge enhancement with less abrupt noise amplification.

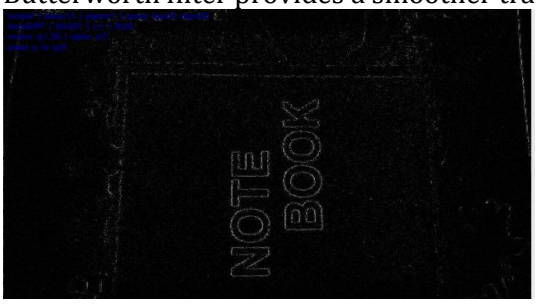
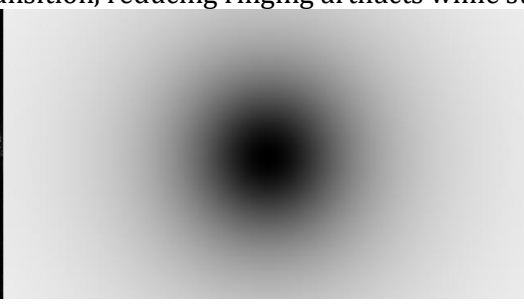
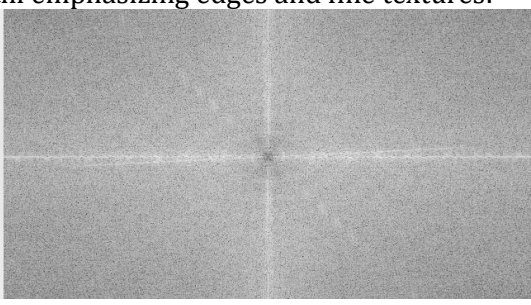




This figure shows the result of a Gaussian High-Pass Filter (GHPF) with a small cutoff frequency $D_0 = 21$. Most low-frequency components are suppressed, leaving only very high-frequency details. Edges are strongly enhanced while noise is also amplified. Due to the Gaussian transition, the enhancement is smoother and produces fewer ringing artifacts compared to the ideal high-pass filter.

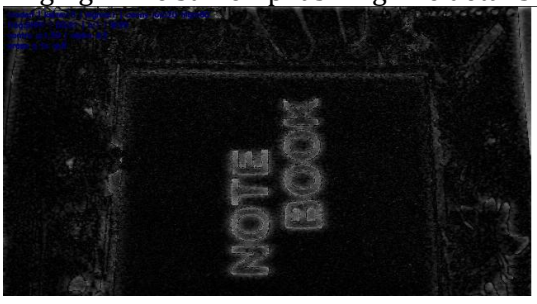
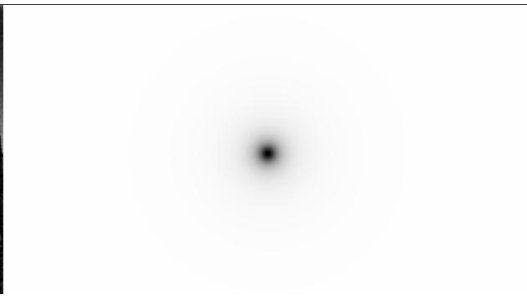
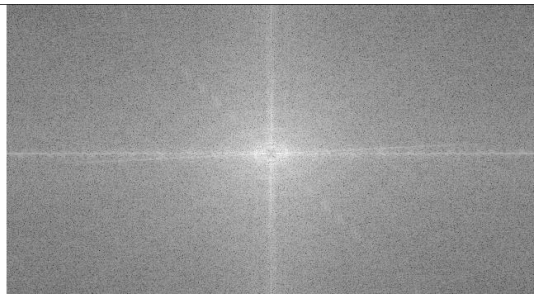




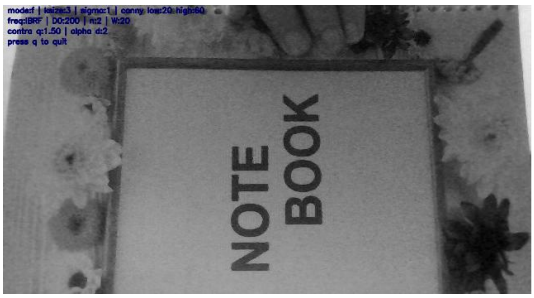
This figure shows the effect of a Butterworth High-Pass Filter (BHPF) with cutoff frequency $D_0 = 201$and order $n = 2$. Low-frequency components are attenuated while high-frequency details are enhanced. Compared to the ideal high-pass filter, the Butterworth filter provides a smoother transition, reducing ringing artifacts while still emphasizing edges and fine textures.

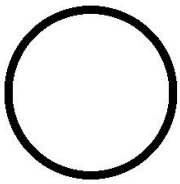
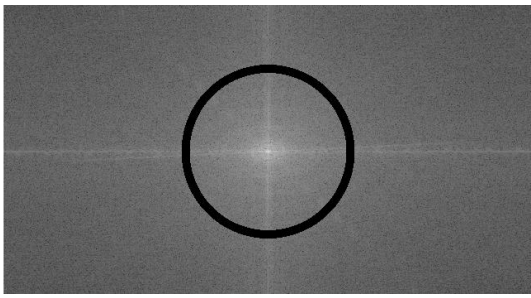




This figure shows the result of a Butterworth High-Pass Filter (BHPF) with a small cutoff frequency $D_0 = 21$and order $n = 2$. Only very high-frequency components are preserved, leading to strong edge enhancement and noticeable noise amplification. Compared to the ideal high-pass filter, the Butterworth response provides smoother attenuation of low frequencies, reducing ringing while still emphasizing fine details.

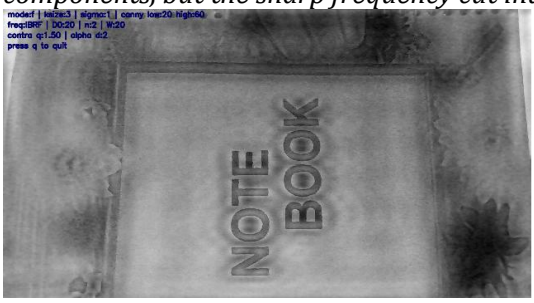




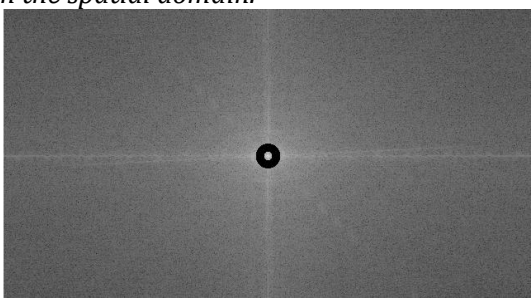
This figure shows the effect of an Ideal Band-Reject Filter (IBRF) with center frequency $D_0 = 200$and bandwidth W. A specific band of frequencies around $D_0$is completely removed, while lower and higher frequencies are preserved. This suppresses periodic noise corresponding to that frequency band, but the ideal sharp cutoff may introduce ringing artifacts in the spatial domain.



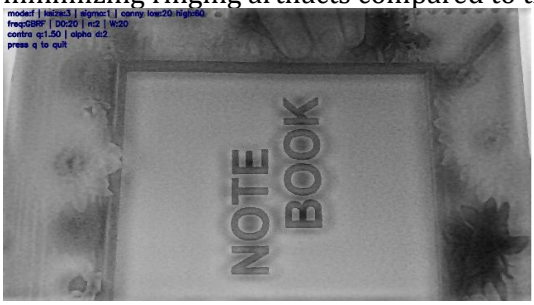



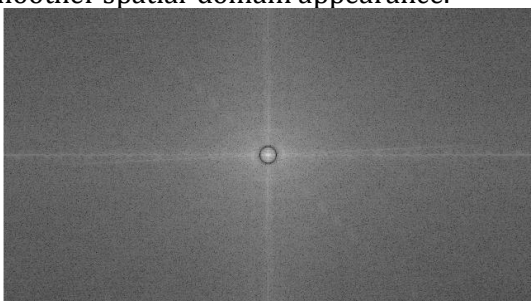
This figure shows the effect of an Ideal Band-Reject Filter (IBRF) with center frequency $D_0 = 20$and bandwidth W. A narrow band of low frequencies around $D_0$is removed, while frequencies outside this band are preserved. This can suppress specific periodic components, but the sharp frequency cut introduces noticeable ringing and distortion in the spatial domain.



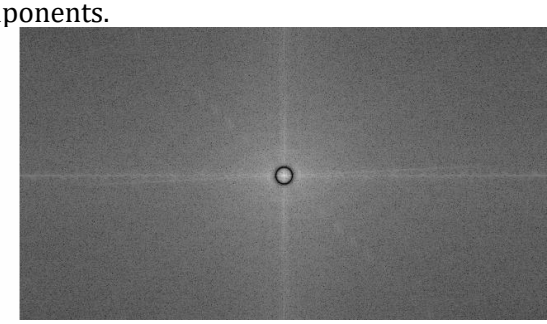



This figure shows the effect of a Gaussian Band-Reject Filter (GBRF) with center frequency $D_0 = 20$. A narrow band of frequencies around $D_0$is smoothly attenuated rather than sharply removed. This reduces specific periodic noise while minimizing ringing artifacts compared to the ideal band-reject filter, resulting in a smoother spatial-domain appearance.

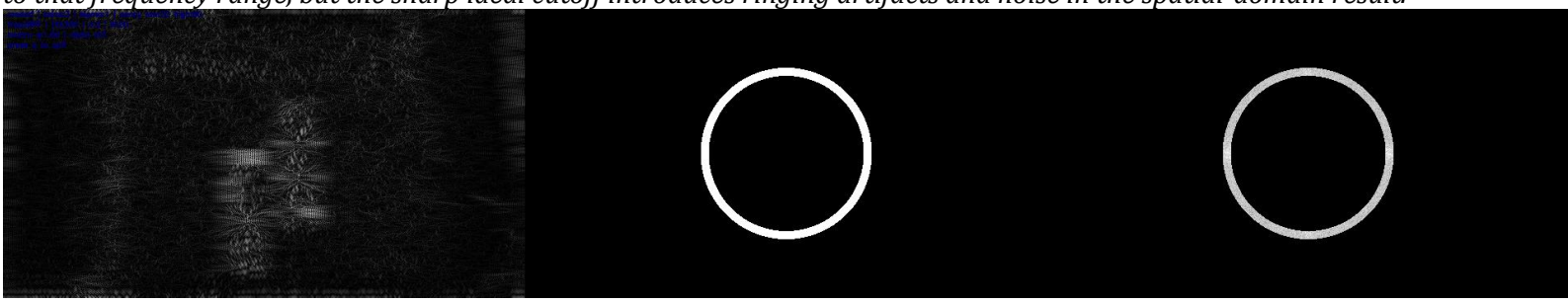


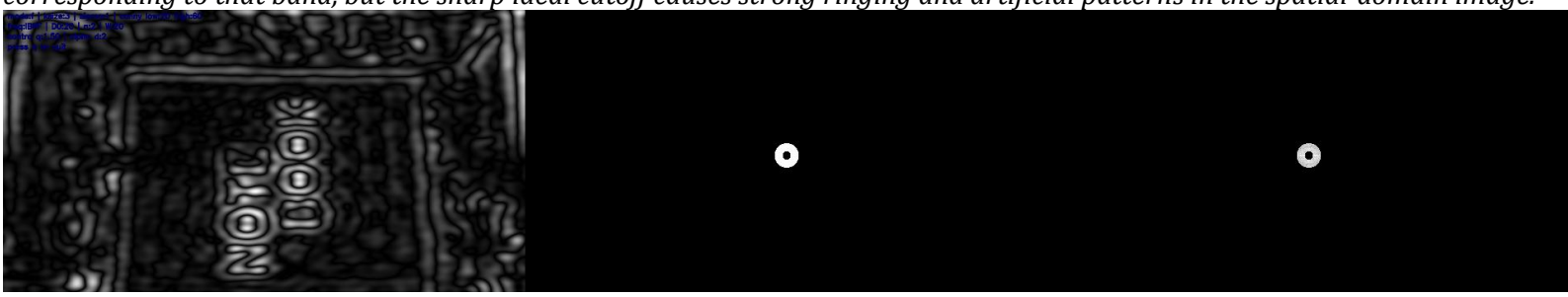
This figure shows the result of a Butterworth Band-Reject Filter (BBRF) with center frequency $D_0 = 20$and order $n = 2$. A band of frequencies around $D_0$is attenuated with a smooth transition. Compared to the ideal band-reject filter, the Butterworth filter reduces ringing artifacts while still effectively suppressing periodic noise components.

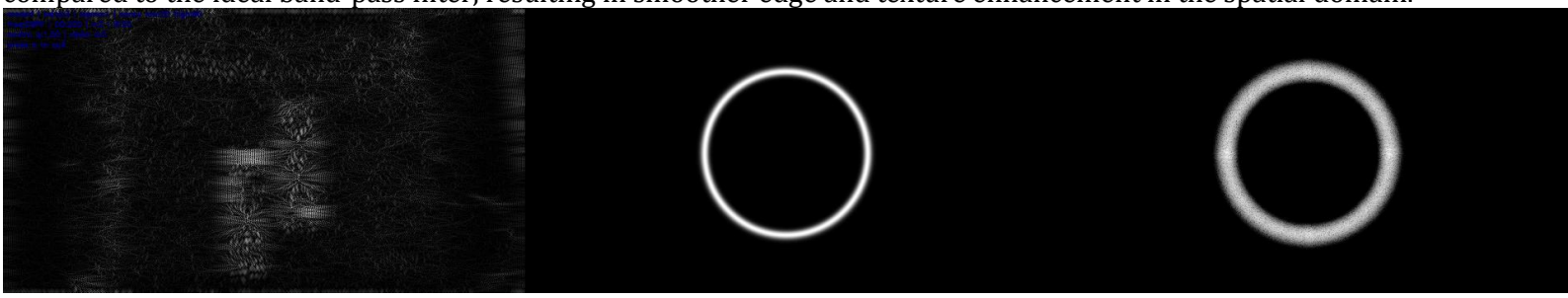
This figure shows the effect of an Ideal Band-Pass Filter (IBPF) with center frequency $D_0 = 200$. Only a specific band of mid-range frequencies is preserved, while both low- and high-frequency components are removed. This emphasizes structures corresponding to that frequency range, but the sharp ideal cutoff introduces ringing artifacts and noise in the spatial-domain result.



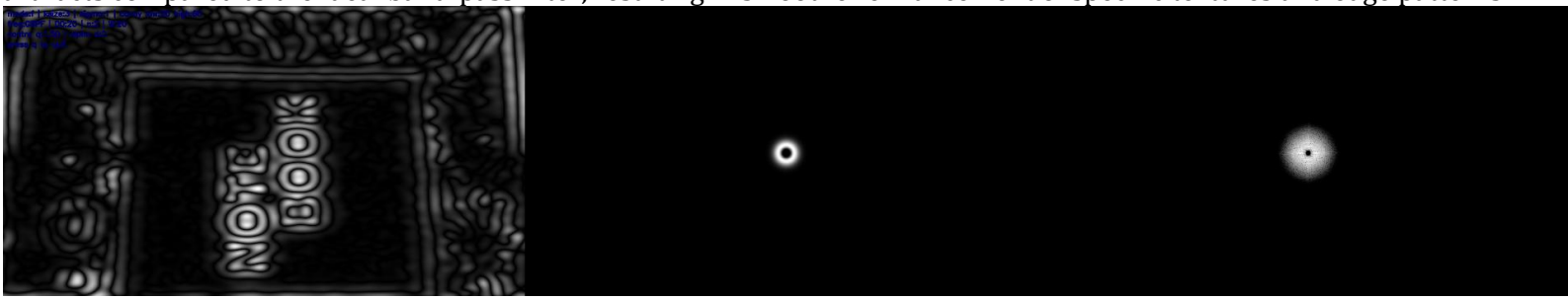
This figure shows the result of an Ideal Band-Pass Filter (IBPF) with center frequency $D_0 = 20$. Only a narrow band of low-frequency components is preserved, while both very low and high frequencies are removed. This enhances specific texture patterns corresponding to that band, but the sharp ideal cutoff causes strong ringing and artificial patterns in the spatial-domain image.



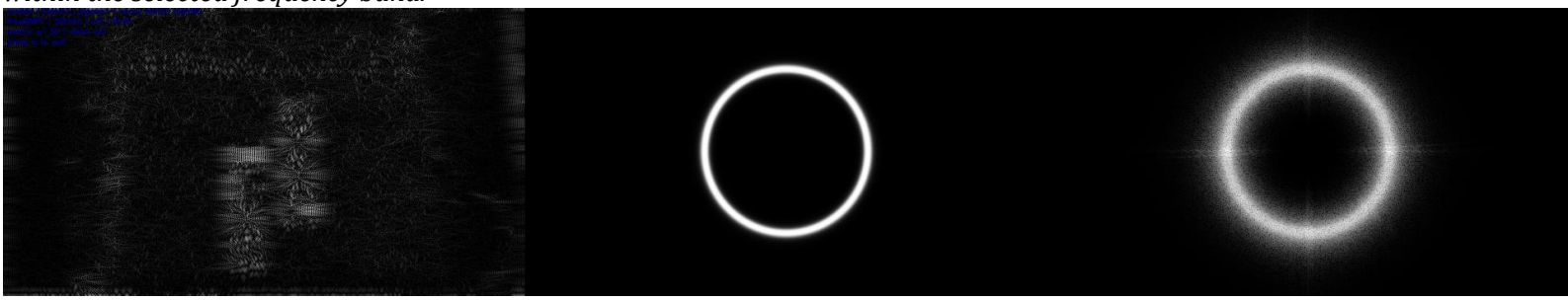
This figure shows the effect of a Gaussian Band-Pass Filter (GBPF) with center frequency $D_0 = 200$. A specific mid-frequency band is smoothly preserved while low and high frequencies are attenuated. The Gaussian transition reduces ringing artifacts compared to the ideal band-pass filter, resulting in smoother edge and texture enhancement in the spatial domain.



This figure shows the effect of a Gaussian Band-Pass Filter (GBPF) with center frequency $D_0 = 20$. Only a narrow band of low-frequency components is smoothly preserved, while other frequencies are attenuated. The Gaussian transition reduces ringing artifacts compared to the ideal band-pass filter, resulting in smoother enhancement of specific textures and edge patterns.



This figure shows the effect of a Butterworth Band-Pass Filter (BBPF) with center frequency $D_0 = 200$and order $n = 2$. A specific mid-frequency band is preserved while low- and high-frequency components are attenuated. Compared to the ideal band-pass filter, the Butterworth filter provides a smoother transition, reducing ringing artifacts while still emphasizing textures and edges within the selected frequency band.



This figure shows the effect of a Butterworth Band-Pass Filter (BBPF) with center frequency $D_0 = 20$and order $n = 2$. A narrow band of low-frequency components is preserved while other frequencies are attenuated. The Butterworth response provides smoother transitions than the ideal band-pass filter, reducing ringing artifacts while still enhancing textures corresponding to the selected frequency band.