Live Canny Edge Detection & Frequency Domain Filtering — Project Documentation

Image Processing (CSE281)

Sama Mohamed Dagher
223101272

1. Introduction

This project implements a real-time demonstration of edge detection and frequency domain filtering using your webcam. The application includes:

- Canny edge detection algorithm
- Sobel edge detection (X and Y gradients)
- Frequency domain filters: Ideal, Gaussian, and Butterworth (both LPF and HPF)
- Real-time parameter adjustment (D0 cutoff frequency and Butterworth order)

---

2. Project Structure

- **main.py**
    - Handles webcam input, mode switching, grayscale conversion
    - Implements real-time visualization of all filters
    - Manages user controls for parameter adjustment
- **canny.py**
    - Implements Canny edge detection steps
    - Functions: non_maximum_suppression(), double_threshold(), edge_tracking_by_hysteresis(), canny_algorithm()
    - Implements frequency domain filters:
        - ILPF() / IHPF() - Ideal Low/High Pass Filters
        - GLPF() / GHPF() - Gaussian Low/High Pass Filters
        - BLPF() / BHPF() - Butterworth Low/High Pass Filters

---

canny.py:

```python
import cv2
import numpy as np
from collections import deque

def non_maximum_suppression(grad_mag, grad_angle):
    """
    Apply non-maximum suppression to gradient magnitudes.

    Args:
        grad_mag: 2D array of gradient magnitudes
        grad_angle: 2D array of gradient angles (in radians)

    Returns:
        Z: Suppressed gradient magnitude array
    """
    M, N = grad_mag.shape
    Z = np.zeros((M, N), dtype=np.float32)

    # Convert gradient angle to degrees and normalize to [0, 180)
    angle = np.rad2deg(grad_angle) % 180

    # Process interior pixels (exclude borders: no padding)
    for i in range(1, M - 1):
        for j in range(1, N - 1):
            q, r = 255, 255  # Initialize comparison values (to suppress pixels
in unexpected cases)

            current_angle = angle[i, j]

            # Determine neighboring pixels based on gradient direction
            # 0° or 180°: horizontal edge
            if (0 <= current_angle < 22.5) or (157.5 <= current_angle <= 180):
                q = grad_mag[i, j + 1]
                r = grad_mag[i, j - 1]

            # 45°: diagonal edge (northeast-southwest)
            elif 22.5 <= current_angle < 67.5:
                q = grad_mag[i + 1, j - 1]
                r = grad_mag[i - 1, j + 1]

            # 90°: vertical edge
            elif 67.5 <= current_angle < 112.5:
                q = grad_mag[i + 1, j]
                r = grad_mag[i - 1, j]

            # 135°: diagonal edge (northwest-southeast)
            elif 112.5 <= current_angle < 157.5:
                q = grad_mag[i - 1, j - 1]
                r = grad_mag[i + 1, j + 1]
```

```python
            # Keep pixel if it's a local maximum along gradient direction
            if (grad_mag[i, j] >= q) and (grad_mag[i, j] >= r):
                Z[i, j] = grad_mag[i, j]
            # Otherwise, suppress it (already 0)

    return Z

def double_threshold(image, low_threshold, high_threshold):
    """
    Apply double thresholding and return binary masks for hysteresis tracking.

    Args:
        image: 2D array of gradient magnitudes
        low_threshold: Lower threshold value
        high_threshold: Upper threshold value

    Returns:
        strong_edges: Binary mask of strong edge pixels
        weak_edges: Binary mask of weak edge pixels
    """
    if low_threshold >= high_threshold:
        raise ValueError("low_threshold must be less than high_threshold")

    strong_edges = (image >= high_threshold)
    weak_edges = (image >= low_threshold) & (image < high_threshold)

    return strong_edges, weak_edges

def edge_tracking_by_hysteresis(strong_edges, weak_edges):
    """
    Connect weak edges to strong edges using hysteresis thresholding.

    Fast BFS implementation - processes each pixel exactly once.

    Args:
        strong_edges: Binary mask of strong edge pixels
        weak_edges: Binary mask of weak edge pixels

    Returns:
        result: Final edge map with all connected edges
    """
    M, N = strong_edges.shape

    strong_edges = strong_edges.astype(bool)
    weak_edges = weak_edges.astype(bool)
    result = strong_edges.copy()
    visited = np.zeros((M, N), dtype=bool)

    # Queue for BFS
    queue = deque() #Uses a deque (double-ended queue) for efficient BFS

    # Add all strong edge pixels to queue as starting points
    strong_coords = np.argwhere(strong_edges)
    for coord in strong_coords:
        i, j = coord
        queue.append((i, j))
        visited[i, j] = True #add all strong coords to visited

    # 8-connected neighbors (all directions)
    neighbors = [(-1, -1), (-1, 0), (-1, 1),
                 (0, -1),           (0, 1),
                 (1, -1),  (1, 0),  (1, 1)]

    # BFS to track connected weak edges
    while queue:
        i, j = queue.popleft()

        # Check all 8 neighbors
        for di, dj in neighbors:
            ni, nj = i + di, j + dj

            # Check bounds (neighbor is inside image)
            if 0 <= ni < M and 0 <= nj < N:
                # If neighbor is weak edge and not yet visited
                if weak_edges[ni, nj] and not visited[ni, nj]:
                    result[ni, nj] = True  # Promote weak edge to strong
                    visited[ni, nj] = True
                    queue.append((ni, nj))  # Continue searching from this pixel

    return result

def canny_algorithm(image, ksize=3):

    # Step 1: Noise reduction
    blurred = cv2.GaussianBlur(image, (ksize,ksize), 0)

    # Step 2: Gradient calculation
    grad_x = cv2.Sobel(blurred, cv2.CV_64F, 1, 0, ksize=3)
    grad_y = cv2.Sobel(blurred, cv2.CV_64F, 0, 1, ksize=3)
    grad_mag = np.sqrt(grad_x**2 + grad_y**2)
    grad_angle = np.arctan2(grad_y, grad_x)

    # Step 3: Non-maximum suppression
    non_max_image = non_maximum_suppression(grad_mag, grad_angle)

    # Step 4: Double thresholding
    strong_edges, weak_edges = double_threshold(non_max_image, 30, 60)

    # Step 5: Hysteresis edge tracking
    final_edges = edge_tracking_by_hysteresis(strong_edges, weak_edges)

    result = final_edges.astype(np.uint8) * 255

    return result

def ILPF(image, D0=50):
    M, N = image.shape
    mask = np.zeros((M, N, 2), np.uint8) #2 channels for real and imaginary parts
(to match DFT output format)
    x, y = np.ogrid[:M, :N] #gives: x with shape (M, 1) containing: 0,1,...,M-1 &
likewise y [alternative for nested loops]
    mask_area = (x-M//2)**2 + (y-N//2)**2 <= D0**2 #True inside circle (low
frequencies), False outside
    mask[mask_area] = 1 #if True -> both channels = 1; gives us a circle of 1s in
the middle, 0/black outside

    dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
    shifted = np.fft.fftshift(dft)
    fshift = shifted*mask #keeps the values of "shifted" within the circle

    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    img_back = cv2.normalize(img_back, None, 0, 255, cv2.NORM_MINMAX) #contrast
stretching

    return img_back

def IHPF(image, D0=50):
    M, N = image.shape
    mask = np.zeros((M, N, 2), np.uint8) #2 channels for real and imaginary parts
(to match DFT output format)
    x, y = np.ogrid[:M, :N] #gives: x with shape (M, 1) containing: 0,1,...,M-1 &
likewise y [alternative for nested loops]
```

```
    mask_area = (x-M//2)**2 + (y-N//2)**2 > D0**2 #True inside circle (low
frequencies), False outside
    mask[mask_area] = 1 #if True -> both channels = 1; gives us a circle of 1s in
the middle, 0/black outside

    dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
    shifted = np.fft.fftshift(dft)
    fshift = shifted*mask #keeps the values of "shifted" within the circle

    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    img_back = cv2.normalize(img_back, None, 0, 255, cv2.NORM_MINMAX) #contrast
stretching

    return img_back

def GLPF(image, D0=50):
    M, N = image.shape
    mask = np.zeros((M, N, 2), np.float32) #gaussian mask contains float weights
bc smoother (not just 0/1)
    x, y = np.ogrid[:M, :N]
    D2 = (x-M//2)**2 + (y-N//2)**2 #D2: D squared
    gaussian = np.exp(-D2/(2*(D0**2))) #instead of sqrting then sqring, just keep
D as it is
    mask[:,:,0] = gaussian
    mask[:,:,1] = gaussian #both real and imaginary parts of every frequency are
multiplied by the same Gaussian weight when we do shifted*mask

    dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
    shifted = np.fft.fftshift(dft)
    fshift = shifted*mask #keeps the values of "shifted" within the circle

    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    img_back = cv2.normalize(img_back, None, 0, 255, cv2.NORM_MINMAX) #contrast
stretching

    return img_back

def GHPF(image, D0=50):
    M, N = image.shape
    mask = np.zeros((M, N, 2), np.float32) #gaussian mask contains float weights
bc smoother (not just 0/1)
```

```
    x, y = np.ogrid[:M, :N]
    D2 = (x-M//2)**2 + (y-N//2)**2 #D2: D squared
    gaussian = 1-np.exp(-D2/(2*(D0**2))) #instead of sqrting then sqring, just
keep D as it is
    mask[:,:,0] = gaussian
    mask[:,:,1] = gaussian #both real and imaginary parts of every frequency are
multiplied by the same Gaussian weight when we do shifted*mask

    dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
    shifted = np.fft.fftshift(dft)
    fshift = shifted*mask

    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    img_back = cv2.normalize(img_back, None, 0, 255, cv2.NORM_MINMAX)

    return img_back

def BLPF(image, D0=50, n=1):
    M, N = image.shape
    mask = np.zeros((M, N, 2), np.float32)
    x, y = np.ogrid[:M, :N]
    D = np.sqrt((x-M//2)**2 + (y-N//2)**2)
    butterworth = 1.0/(1.0+(D/D0)**(2*n))
    mask[:,:,0] = butterworth
    mask[:,:,1] = butterworth

    dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
    shifted = np.fft.fftshift(dft)
    fshift = shifted*mask

    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    img_back = cv2.normalize(img_back, None, 0, 255, cv2.NORM_MINMAX)

    return img_back

def BHPF(image, D0=50, n=1):
    M, N = image.shape
    mask = np.zeros((M, N, 2), np.float32)
    x, y = np.ogrid[:M, :N]
    D = np.sqrt((x-M//2)**2 + (y-N//2)**2)
    butterworth = 1.0/(1.0+(D0/D)**(2*n))
    mask[:,:,0] = butterworth
    mask[:,:,1] = butterworth

    dft = cv2.dft(np.float32(image), flags=cv2.DFT_COMPLEX_OUTPUT)
    shifted = np.fft.fftshift(dft)
    fshift = shifted*mask

    f_ishift = np.fft.ifftshift(fshift)
    img_back = cv2.idft(f_ishift)
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    img_back = cv2.normalize(img_back, None, 0, 255, cv2.NORM_MINMAX)

    return img_back
```

---

main.py:

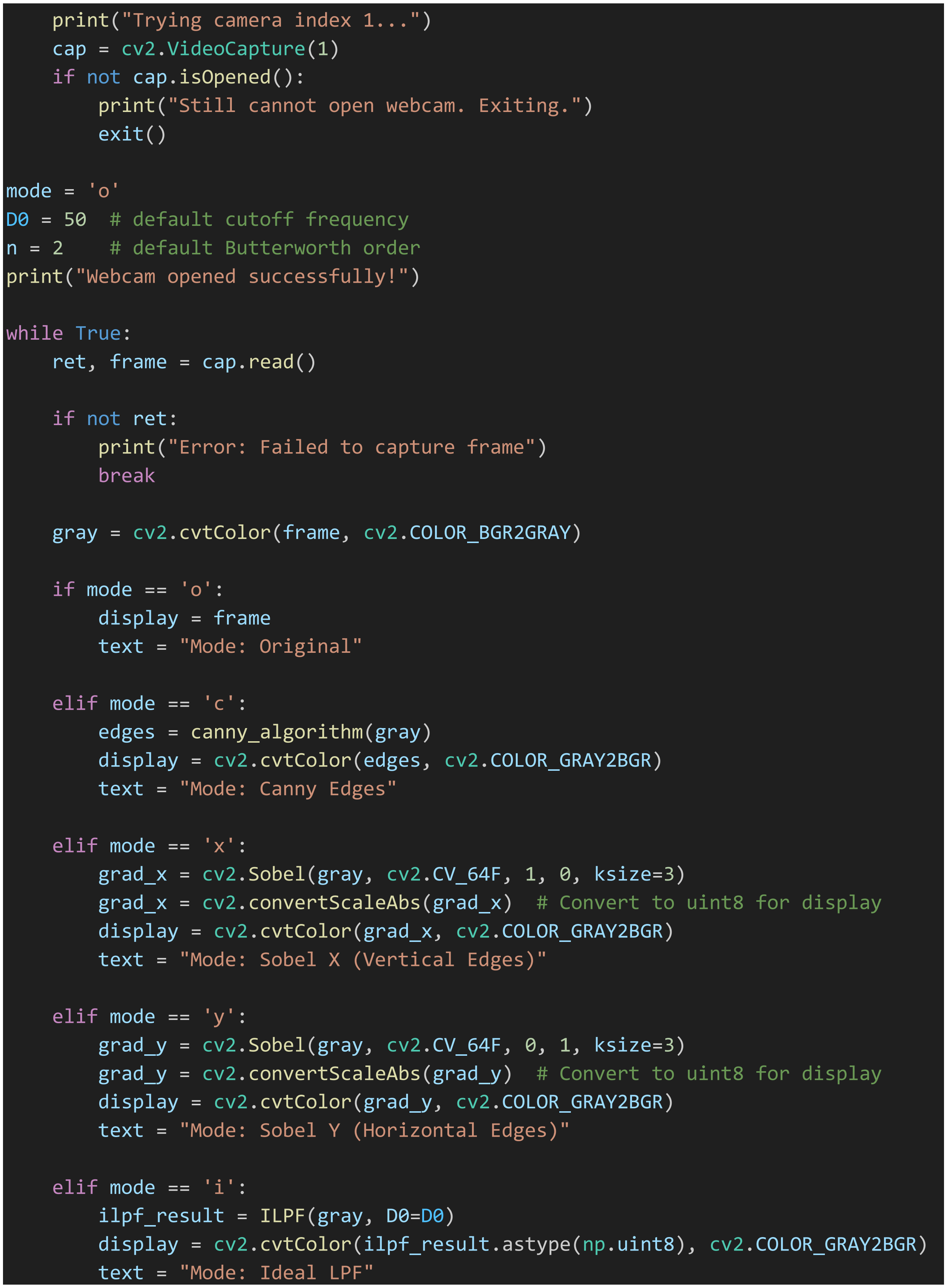
```
"""
Webcam demo for Canny edge detection.

Controls:
'o' - Show original
'i' - Show Ideal LPF
'I' - Show Ideal HPF
'g' - Show Gaussian LPF
'G' - Show Gaussian HPF
'b' - Show Butterworth LPF
'B' - Show Butterworth HPF
'c' - Show Canny edges
'x' - Show Sobel X (vertical edges)
'y' - Show Sobel Y (horizontal edges)
'+' - Increase D0
'-' - Decrease D0
'1'-'9' - Set Butterworth order (n)
'q' - Quit
"""

import cv2
import numpy as np
from canny import canny_algorithm, ILPF, IHPF, GLPF, GHPF, BLPF, BHPF

cap = cv2.VideoCapture(0) # Open webcam (0 is usually the default camera)

if not cap.isOpened():
    print("Error: Could not open webcam")
```



```python
    print("Trying camera index 1...")
    cap = cv2.VideoCapture(1)
    if not cap.isOpened():
        print("Still cannot open webcam. Exiting.")
        exit()

mode = 'o'
D0 = 50  # default cutoff frequency
n = 2    # default Butterworth order
print("Webcam opened successfully!")

while True:
    ret, frame = cap.read()

    if not ret:
        print("Error: Failed to capture frame")
        break

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    if mode == 'o':
        display = frame
        text = "Mode: Original"

    elif mode == 'c':
        edges = canny_algorithm(gray)
        display = cv2.cvtColor(edges, cv2.COLOR_GRAY2BGR)
        text = "Mode: Canny Edges"

    elif mode == 'x':
        grad_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
        grad_x = cv2.convertScaleAbs(grad_x)  # Convert to uint8 for display
        display = cv2.cvtColor(grad_x, cv2.COLOR_GRAY2BGR)
        text = "Mode: Sobel X (Vertical Edges)"

    elif mode == 'y':
        grad_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
        grad_y = cv2.convertScaleAbs(grad_y)  # Convert to uint8 for display
        display = cv2.cvtColor(grad_y, cv2.COLOR_GRAY2BGR)
        text = "Mode: Sobel Y (Horizontal Edges)"

    elif mode == 'i':
        ilpf_result = ILPF(gray, D0=D0)
        display = cv2.cvtColor(ilpf_result.astype(np.uint8), cv2.COLOR_GRAY2BGR)
        text = "Mode: Ideal LPF"
        text2 = f"D0 = {D0}"

    elif mode == 'I':
        ihpf_result = IHPF(gray, D0=D0)
        display = cv2.cvtColor(ihpf_result.astype(np.uint8), cv2.COLOR_GRAY2BGR)
        text = "Mode: Ideal HPF"
        text2 = f"D0 = {D0}"

    elif mode == 'g':
        glpf_result = GLPF(gray, D0=D0)
        display = cv2.cvtColor(glpf_result.astype(np.uint8), cv2.COLOR_GRAY2BGR)
        text = "Mode: Gaussian LPF"
        text2 = f"D0 = {D0}"

    elif mode == 'G':
        ghpf_result = GHPF(gray, D0=D0)
        display = cv2.cvtColor(ghpf_result.astype(np.uint8), cv2.COLOR_GRAY2BGR)
        text = "Mode: Gaussian HPF"
        text2 = f"D0 = {D0}"

    elif mode == 'b':
        blpf_result = BLPF(gray, D0=D0, n=n)
        display = cv2.cvtColor(blpf_result.astype(np.uint8), cv2.COLOR_GRAY2BGR)
        text = "Mode: Butterworth LPF"
        text2 = f"D0 = {D0}, n = {n}"

    elif mode == 'B':
        bhpf_result = BHPF(gray, D0=D0, n=n)
        display = cv2.cvtColor(bhpf_result.astype(np.uint8), cv2.COLOR_GRAY2BGR)
        text = "Mode: Butterworth HPF"
        text2 = f"D0 = {D0}, n = {n}"

    cv2.putText(display, text, (10, 30),
    cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

    if mode in ['i', 'I', 'g', 'G', 'b', 'B']:
        cv2.putText(display, text2, (10, 60),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 2)

    cv2.imshow('Webcam - Canny Edge Detection', display)

    key = cv2.waitKey(1) & 0xFF # keeps only the last byte of the result (for
some systems)

    if key == ord('q'):
        break
    elif key == ord('+'):
        D0 += 5  # Increase D0
    elif key == ord('-'):
        D0 = max(5, D0 - 5)  # Decrease D0, minimum 5
    elif key >= ord('1') and key <= ord('9'):
        n = key - ord('0')  # Set Butterworth order from 1-9
    elif key in [ord('o'), ord('i'), ord('I'), ord('g'), ord('G'),
                ord('b'), ord('B'), ord('c'), ord('x'), ord('y')]:
        mode = chr(key)

cap.release()
cv2.destroyAllWindows()
```

---

3. User Controls

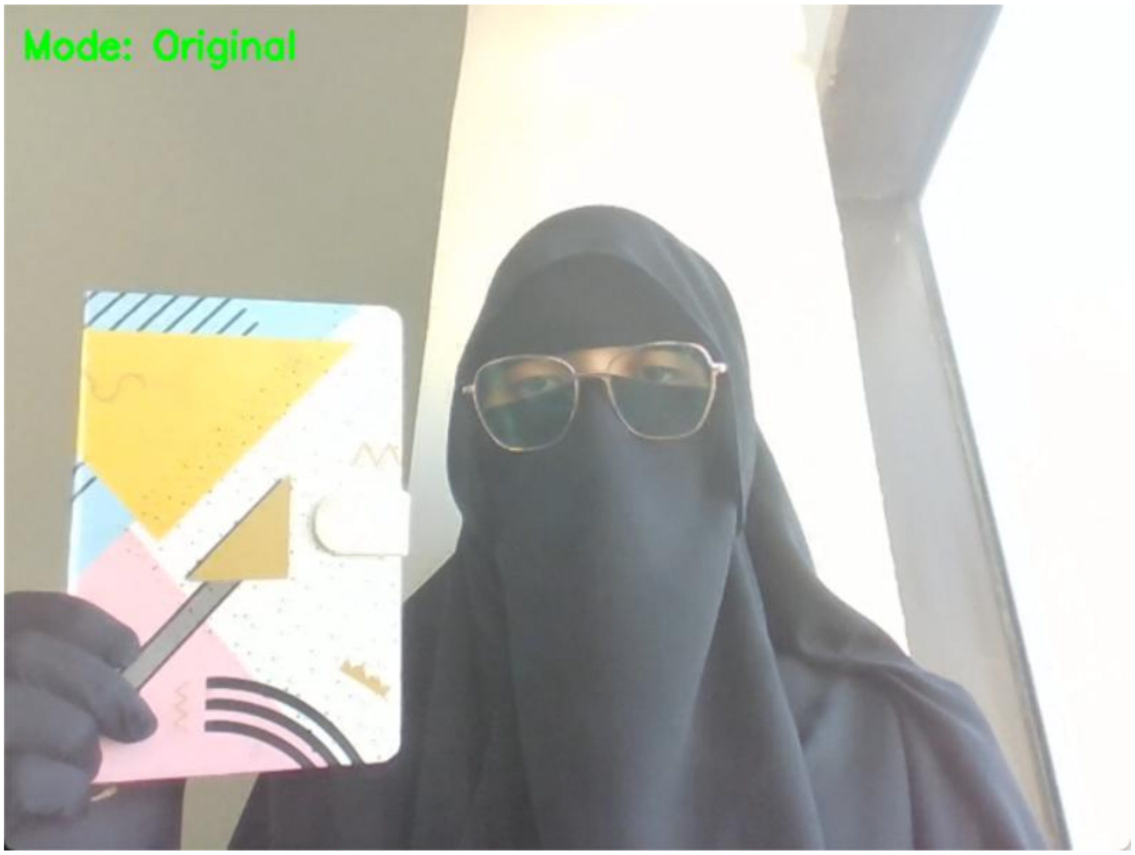
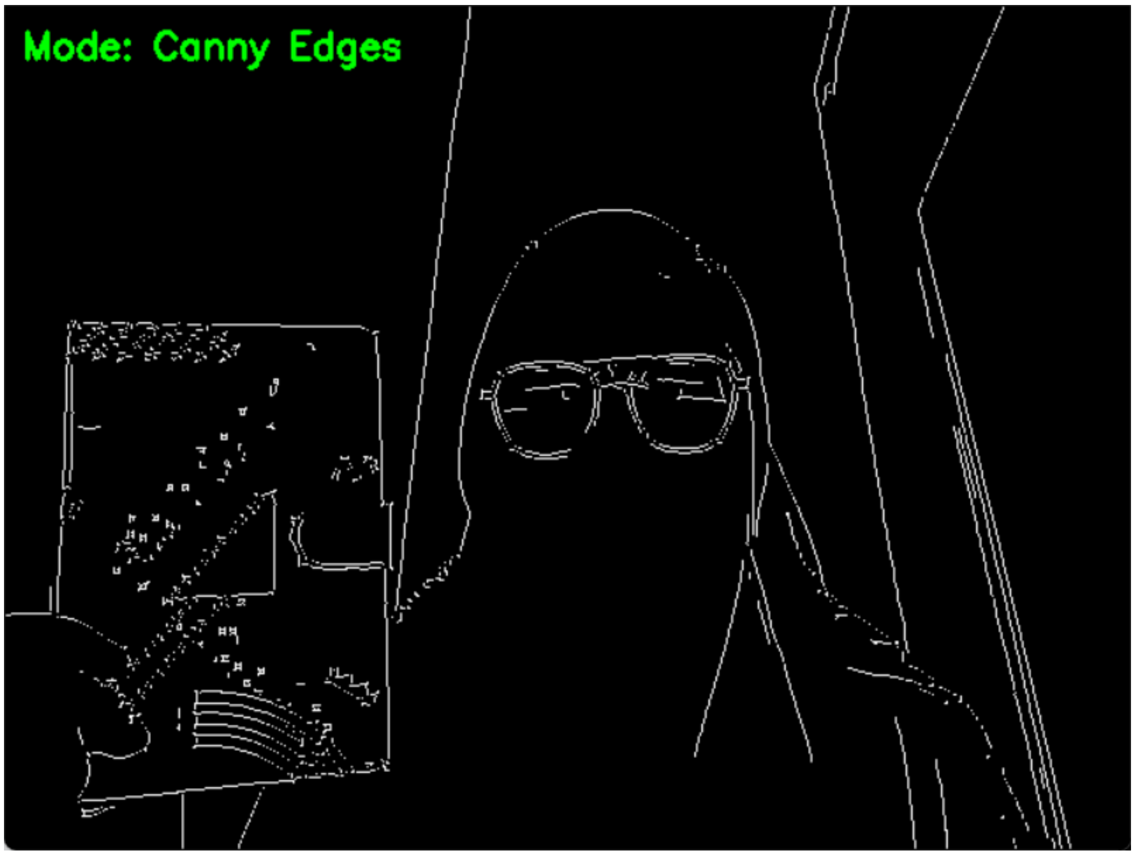
Mode Selection:

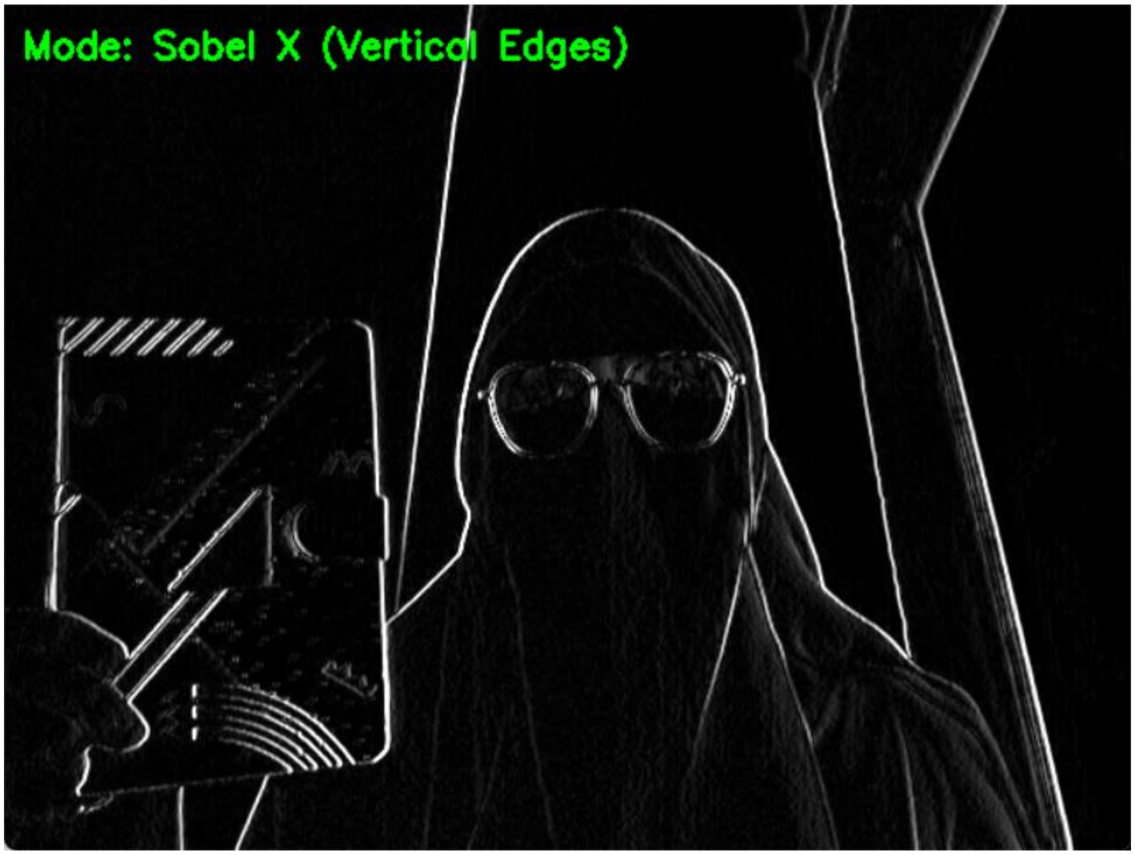
- 'o' - Original image
- 'i' - Ideal LPF
- 'I' - Ideal HPF
- 'g' - Gaussian LPF
- 'G' - Gaussian HPF
- 'b' - Butterworth LPF
- 'B' - Butterworth HPF
- 'c' - Canny edges
- 'x' - Sobel X (vertical edges)
- 'y' - Sobel Y (horizontal edges)

Parameter Adjustment:

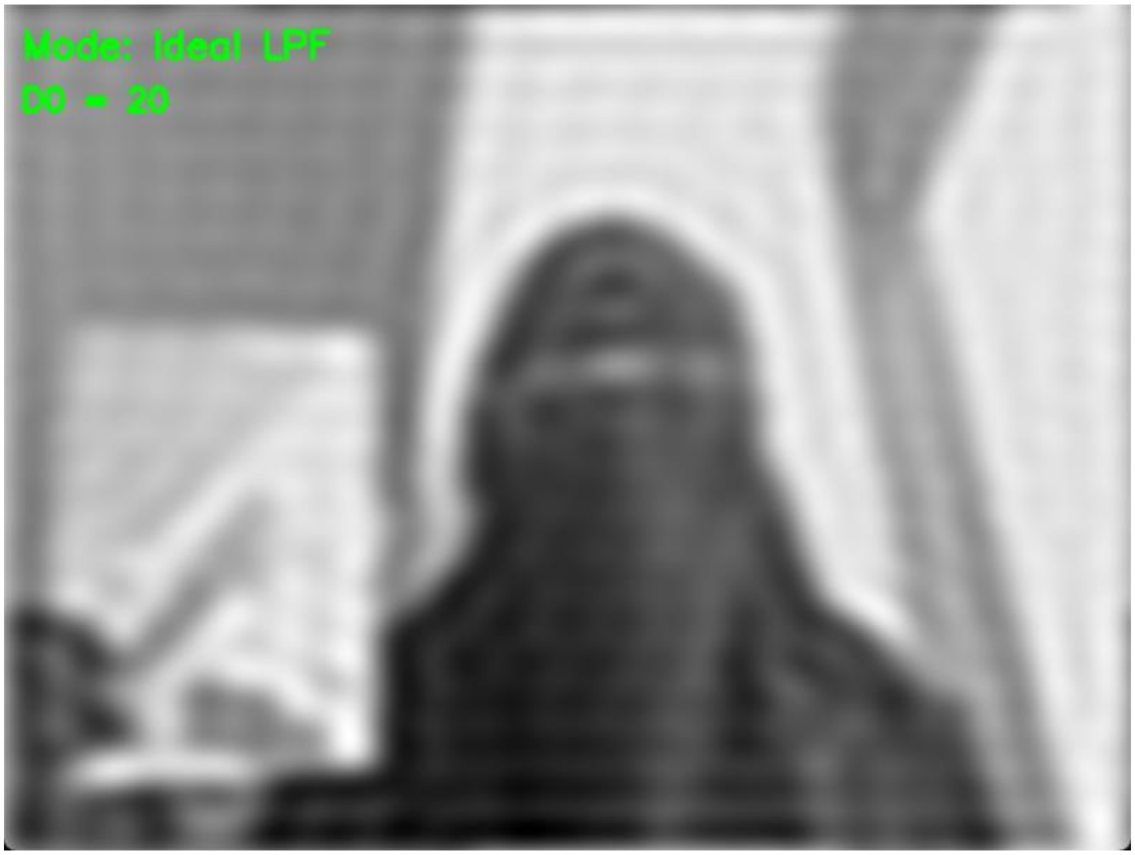
- '+' - Increase D0 (cutoff frequency)
- '-' - Decrease D0
- '1' to '9' - Set Butterworth order (n)
- 'q' - Quit application

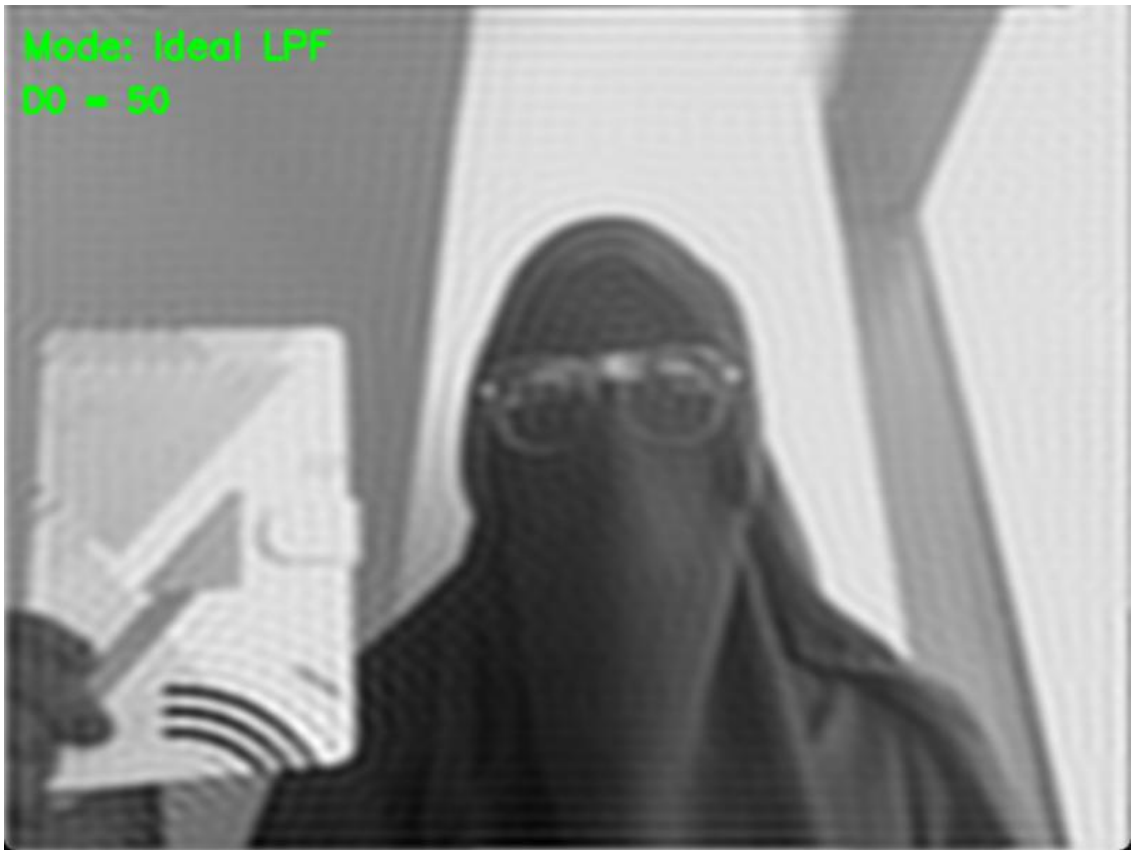
4. Results and Analysis

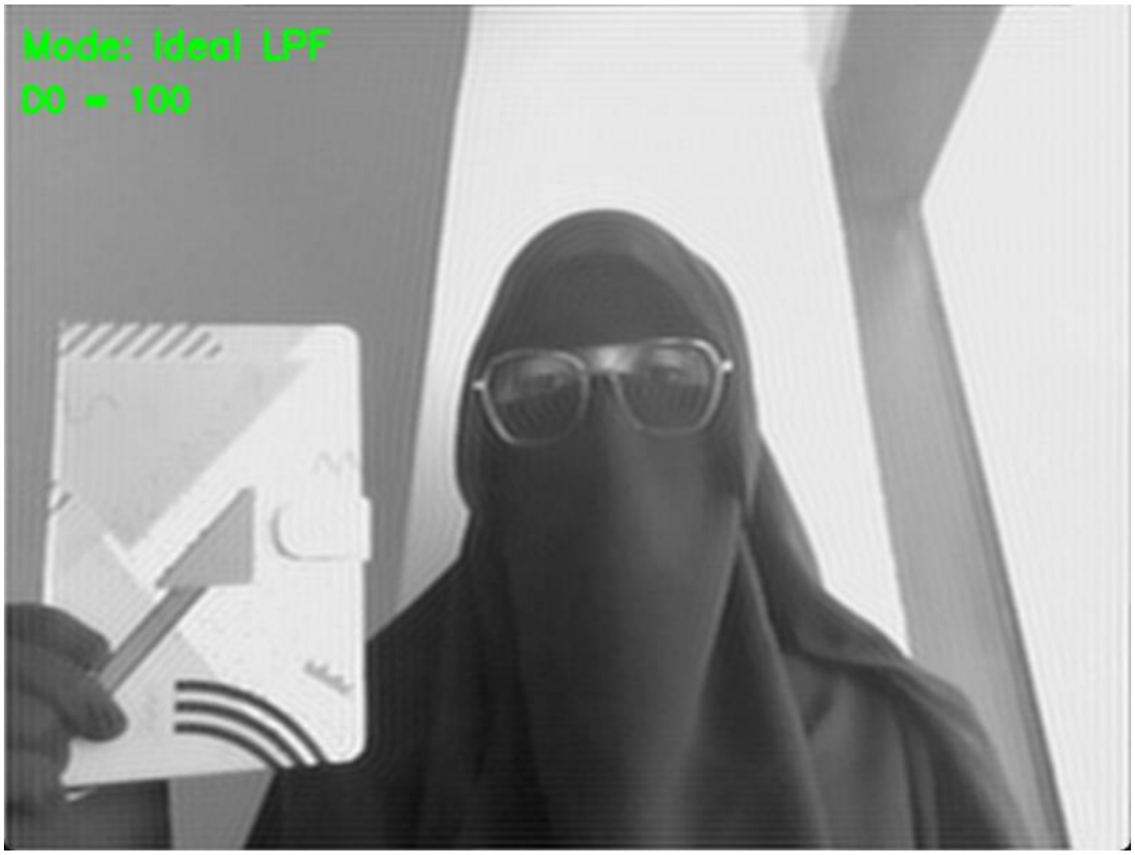
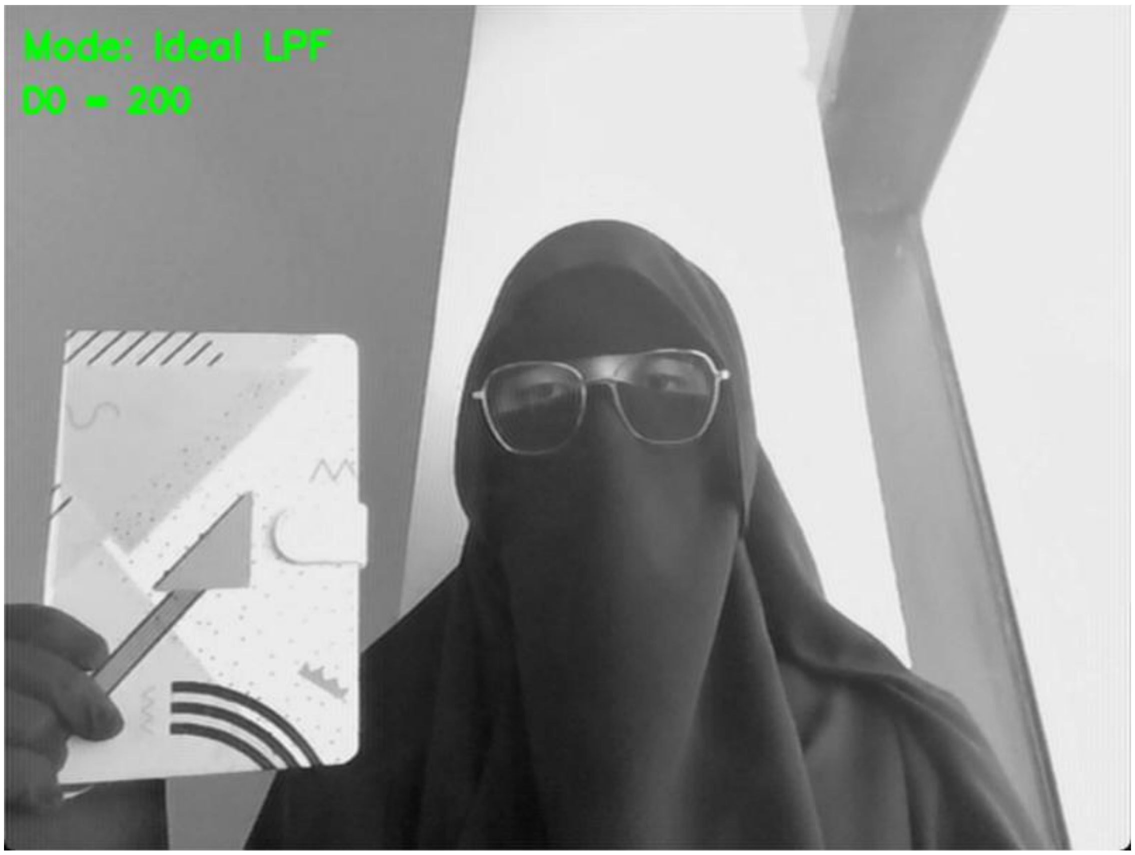

Mode: Original

Mode: Canny Edges

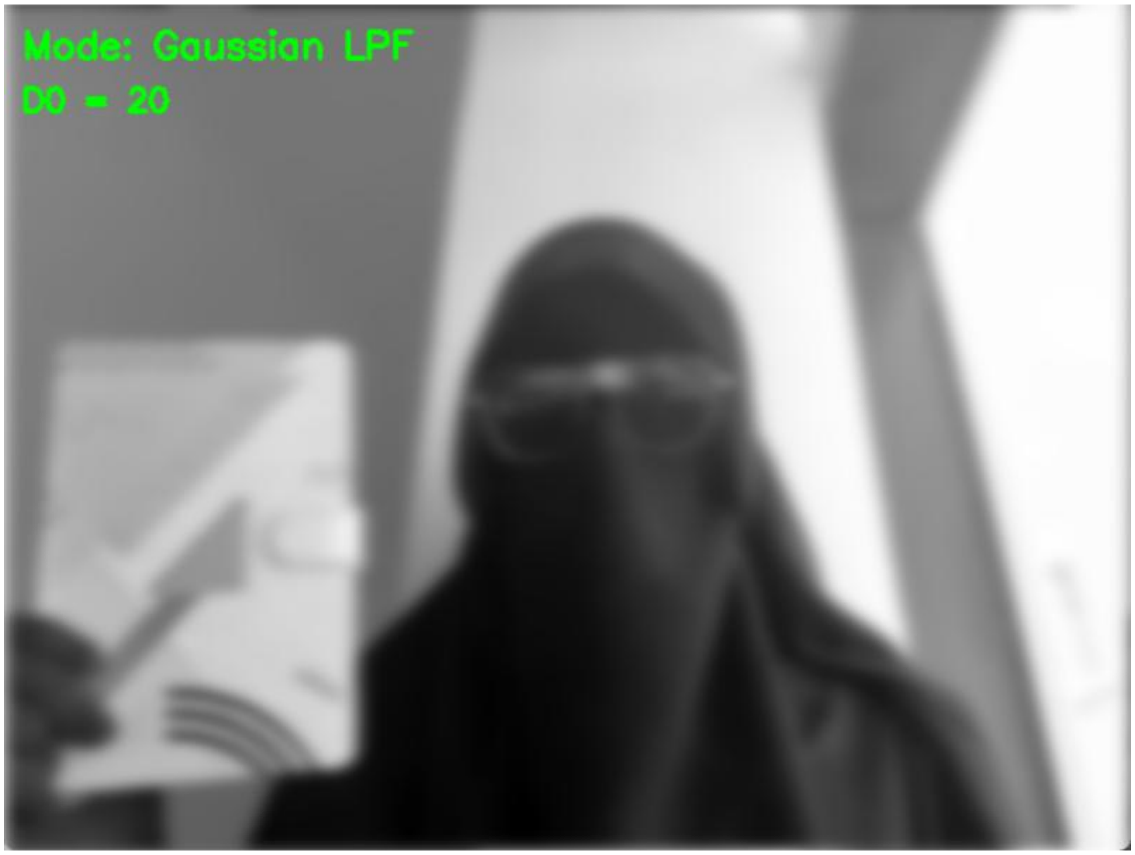

Mode: Sobel X (Vertical Edges)

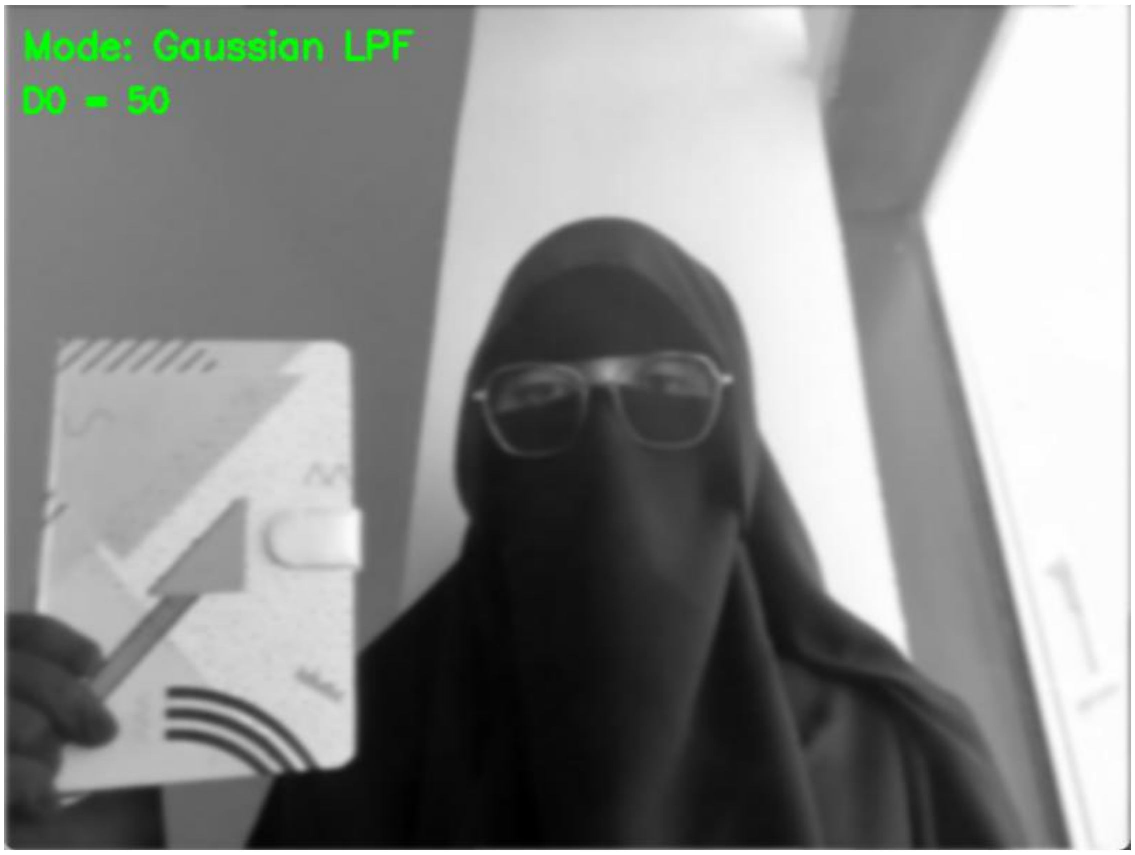
Mode: Sobel X (Vertical Edges)

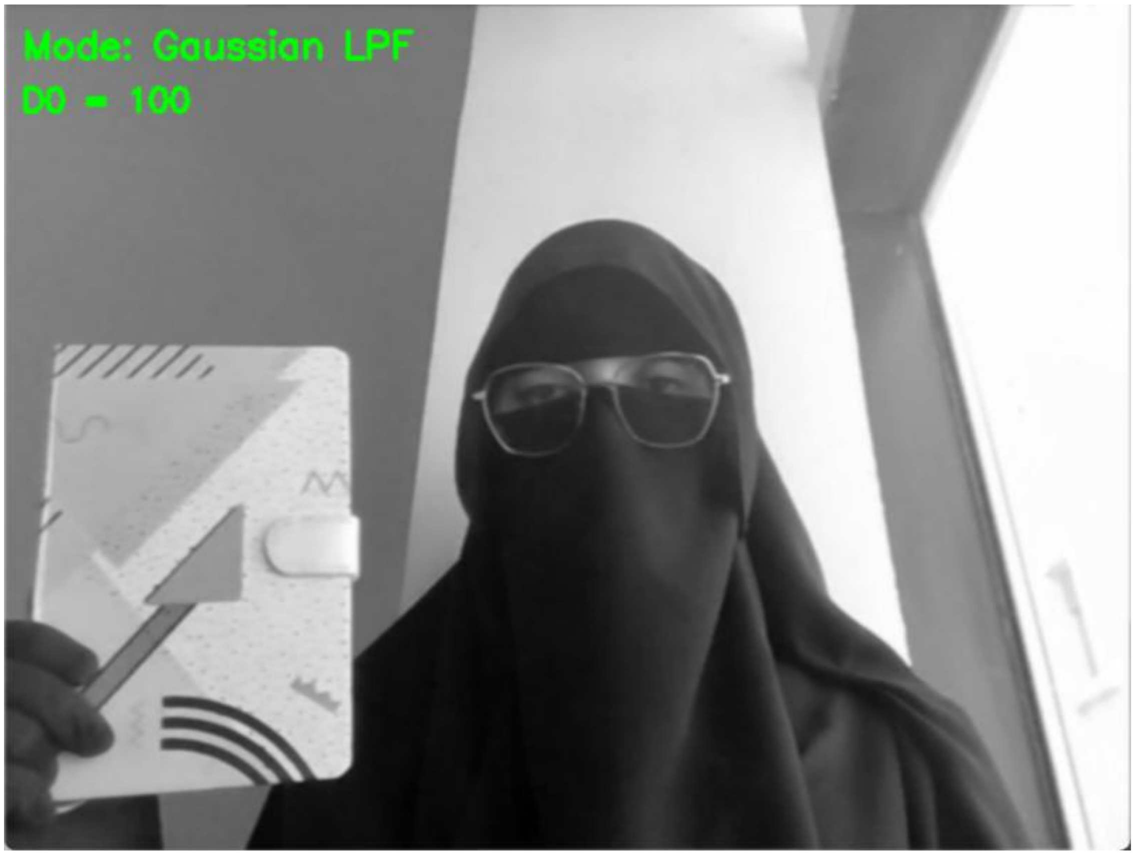
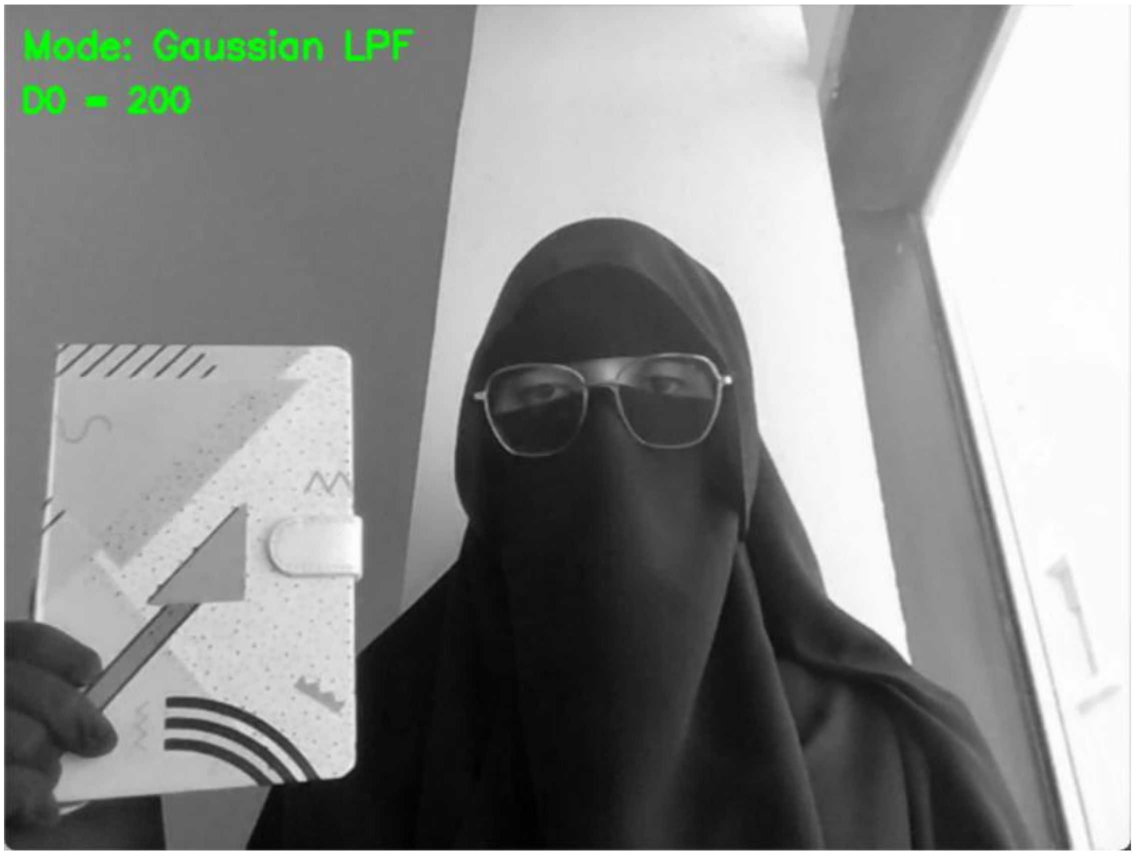

Mode: Ideal LPF
D0 = 20

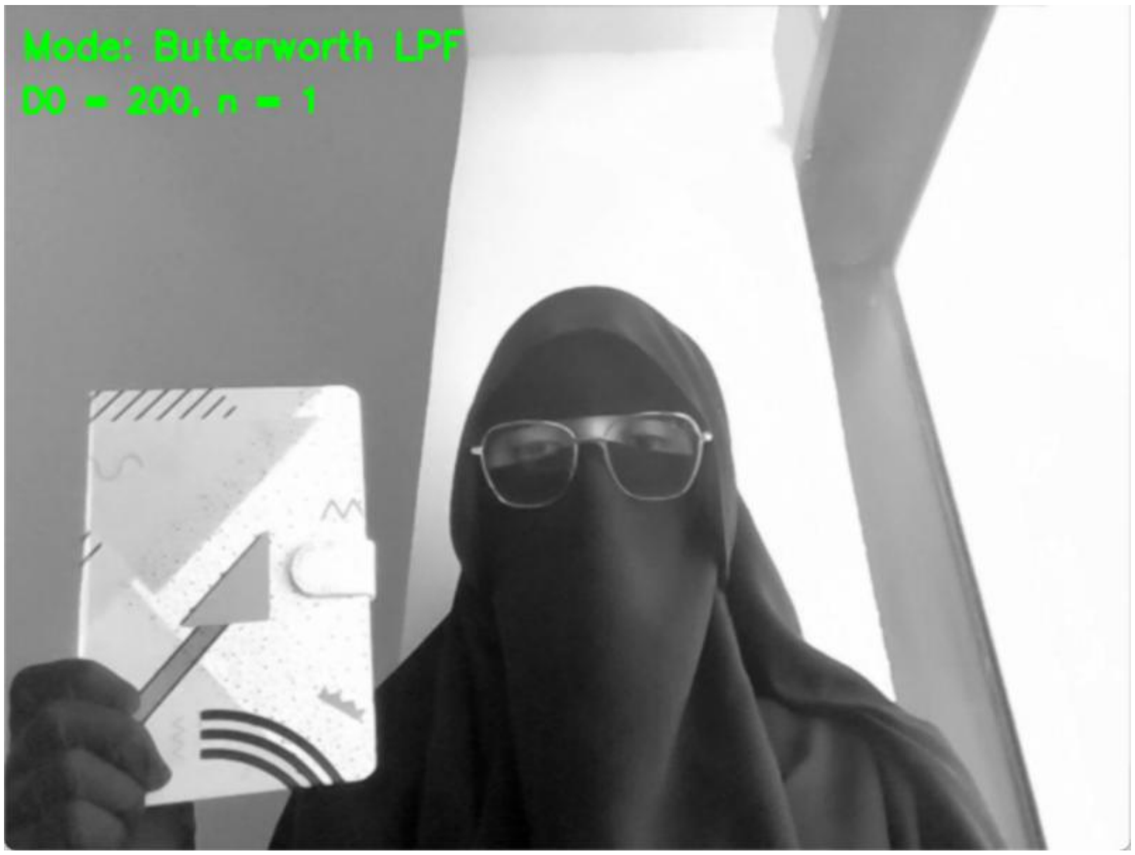

Mode: Ideal LPF
D0 = 50

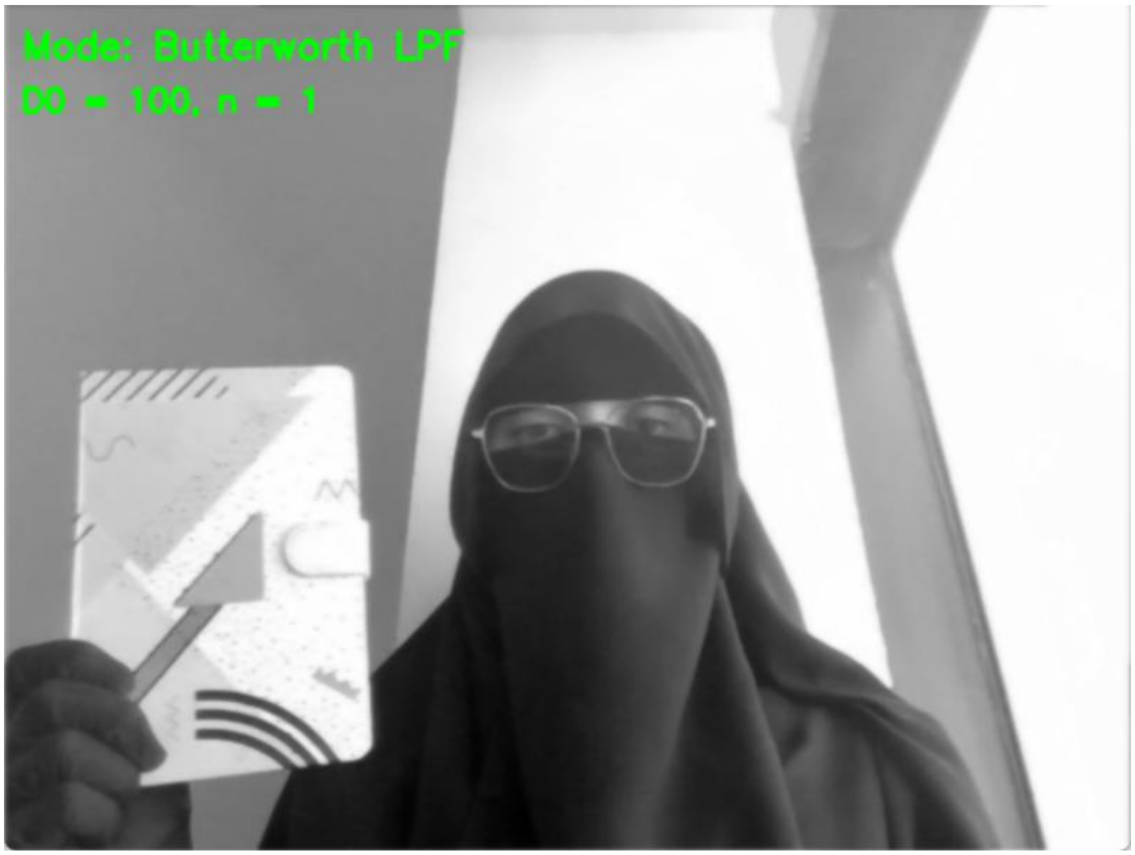

Mode: Ideal LPF
D0 = 100

Mode: Ideal LPF
D0 = 200

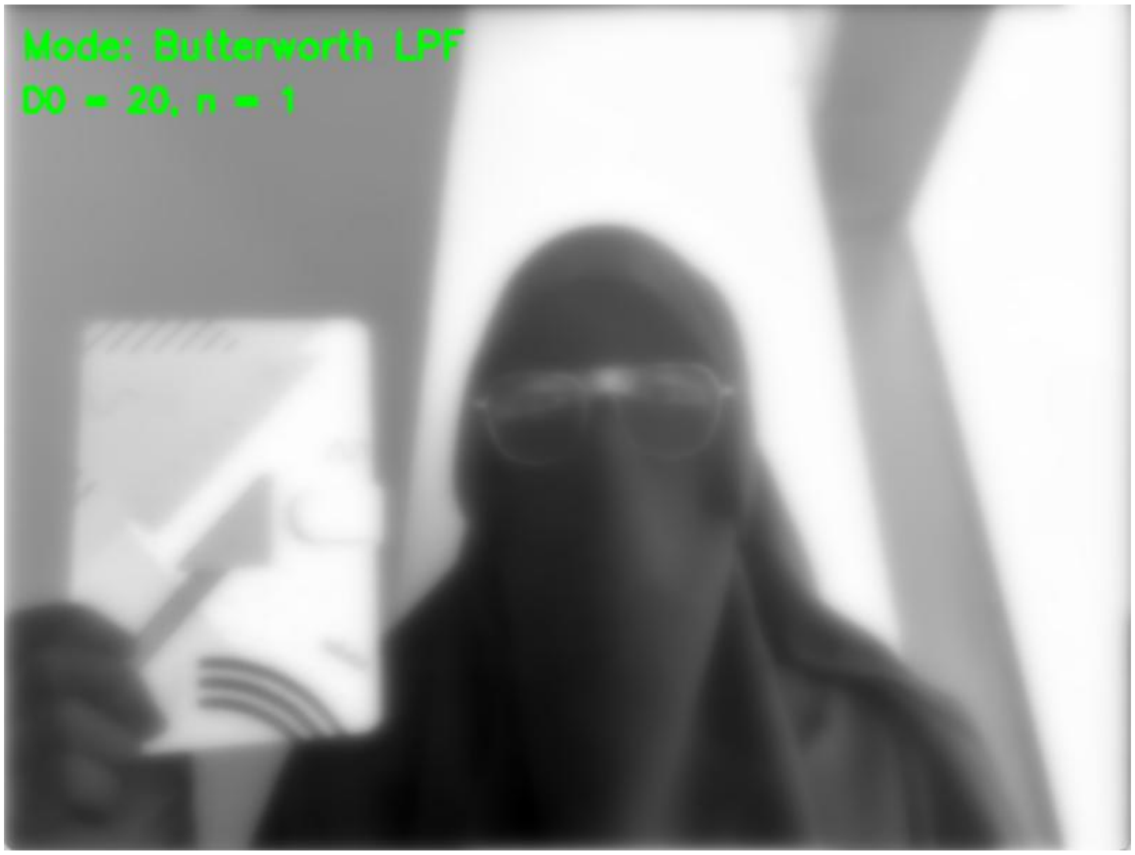

Mode: Gaussian LPF
D0 = 20


Mode: Gaussian LPF
D0 = 50

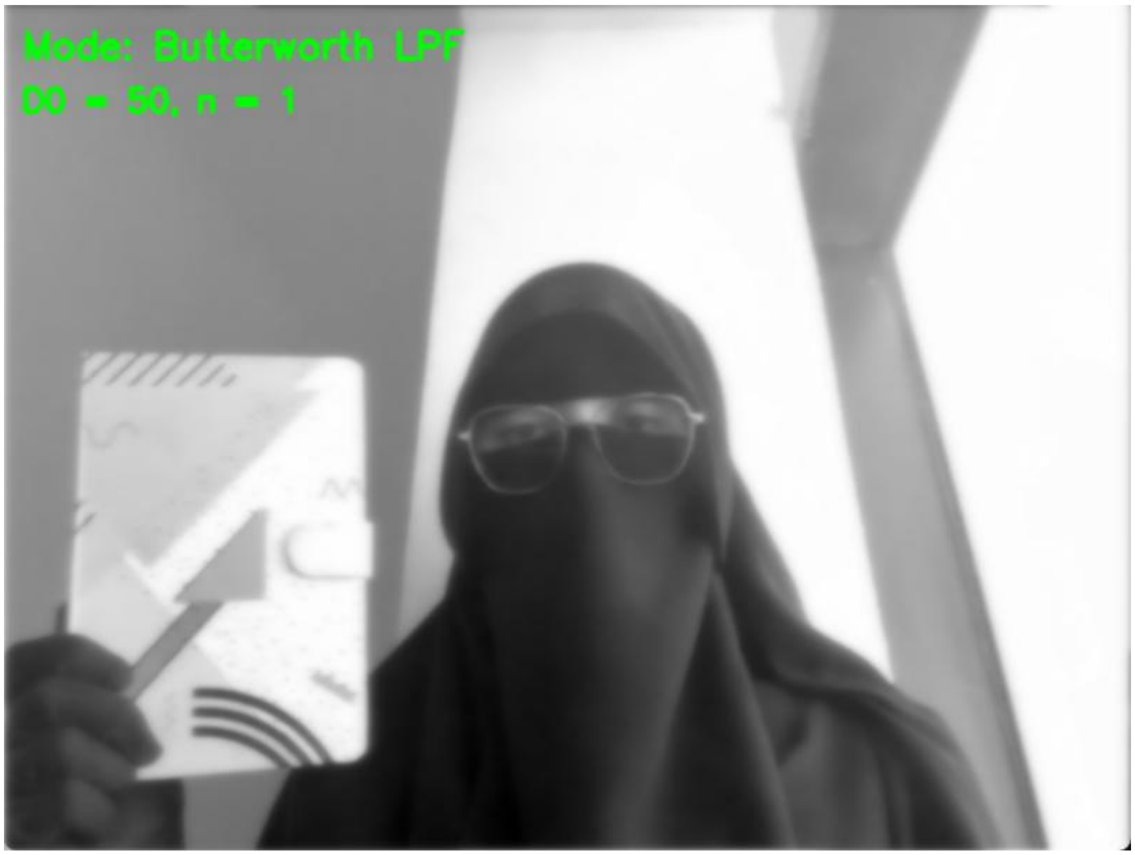

Mode: Gaussian LPF
D0 = 100

Mode: Gaussian LPF
D0 = 200


Mode: Butterworth LPF
D0 = 200, n = 1

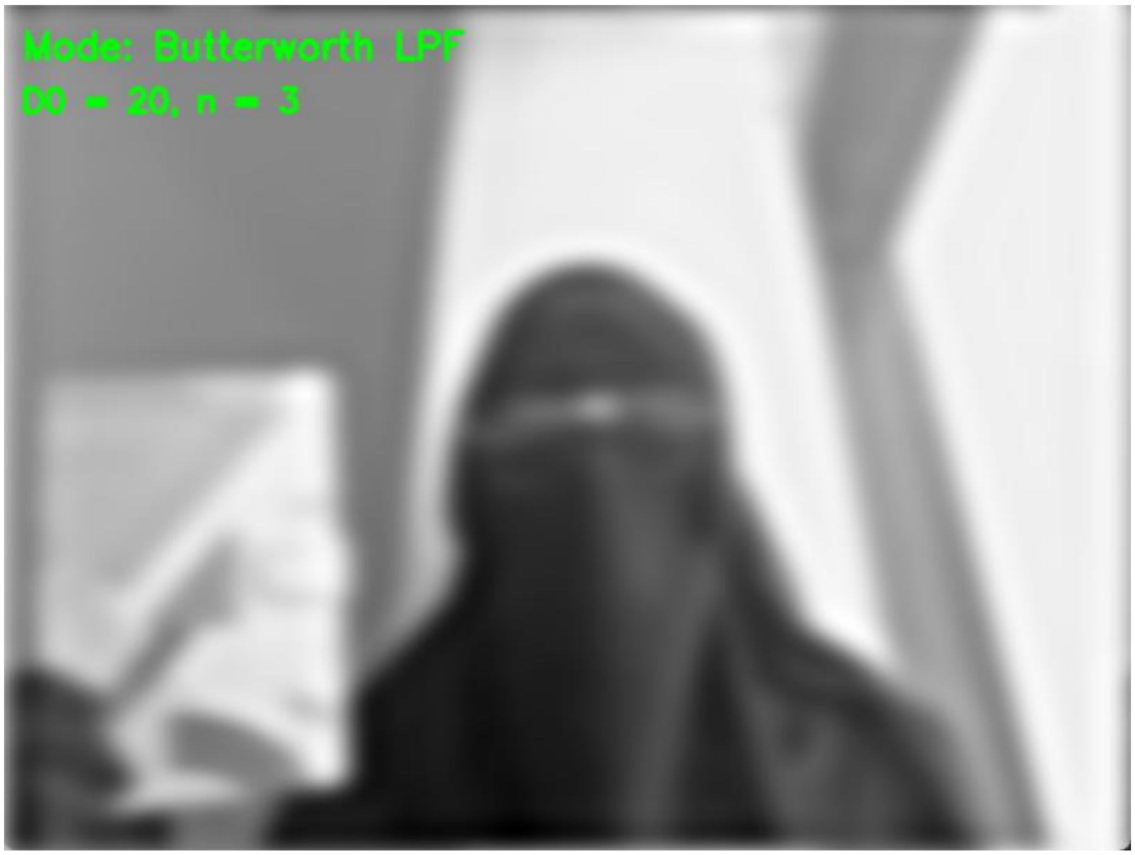

Mode: Butterworth LPF
D0 = 100, n = 1

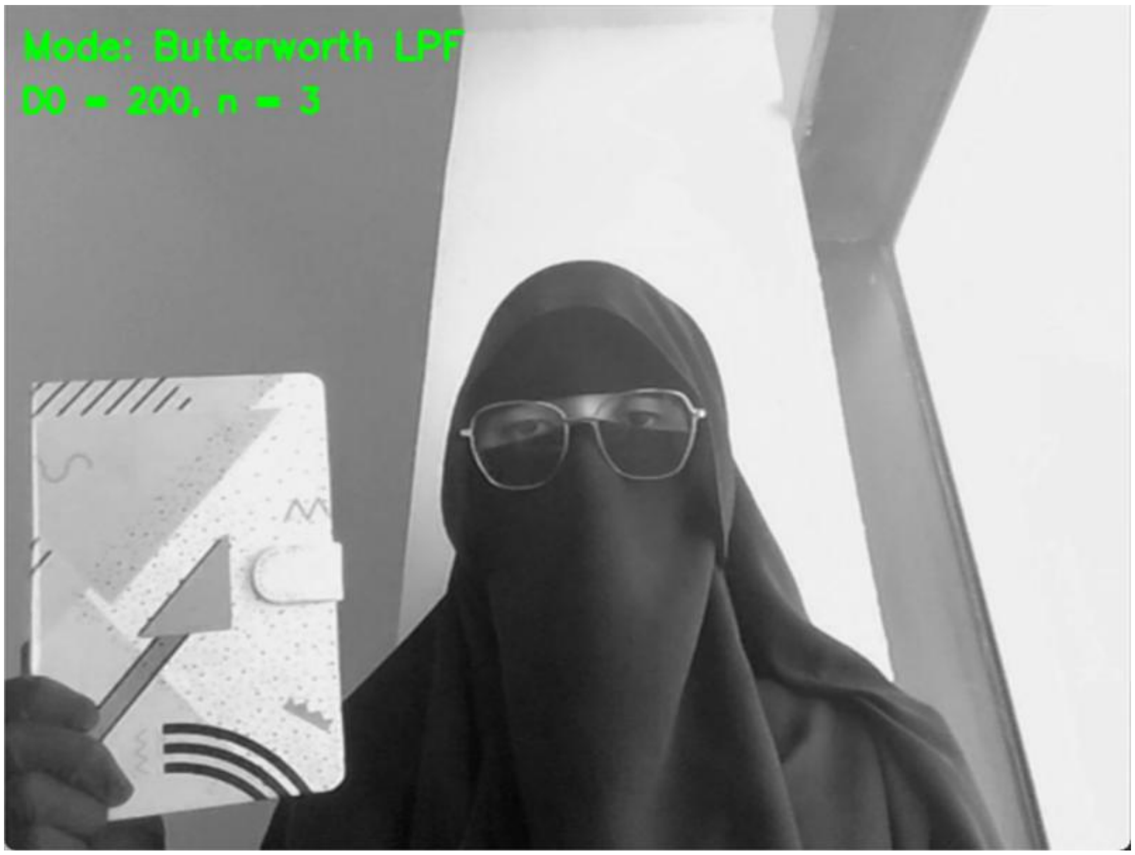


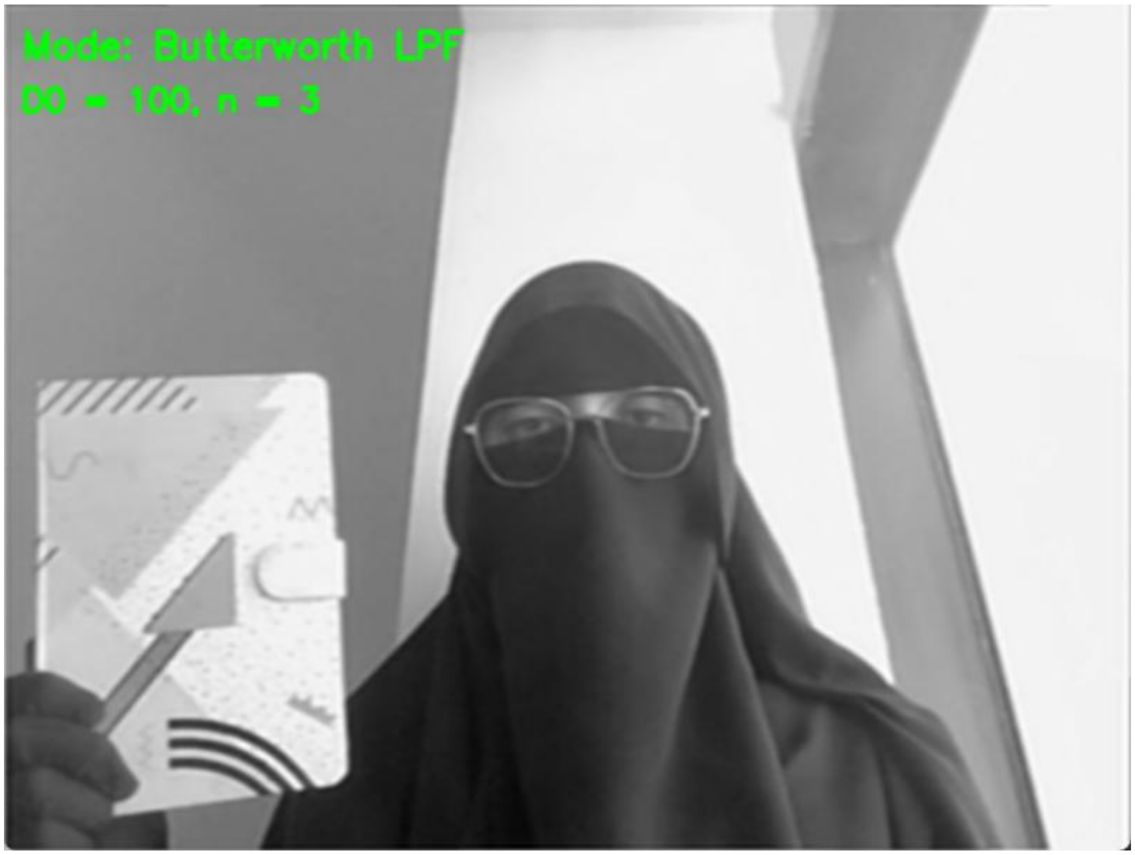
Mode: Butterworth LPF
D0 = 20, n = 1

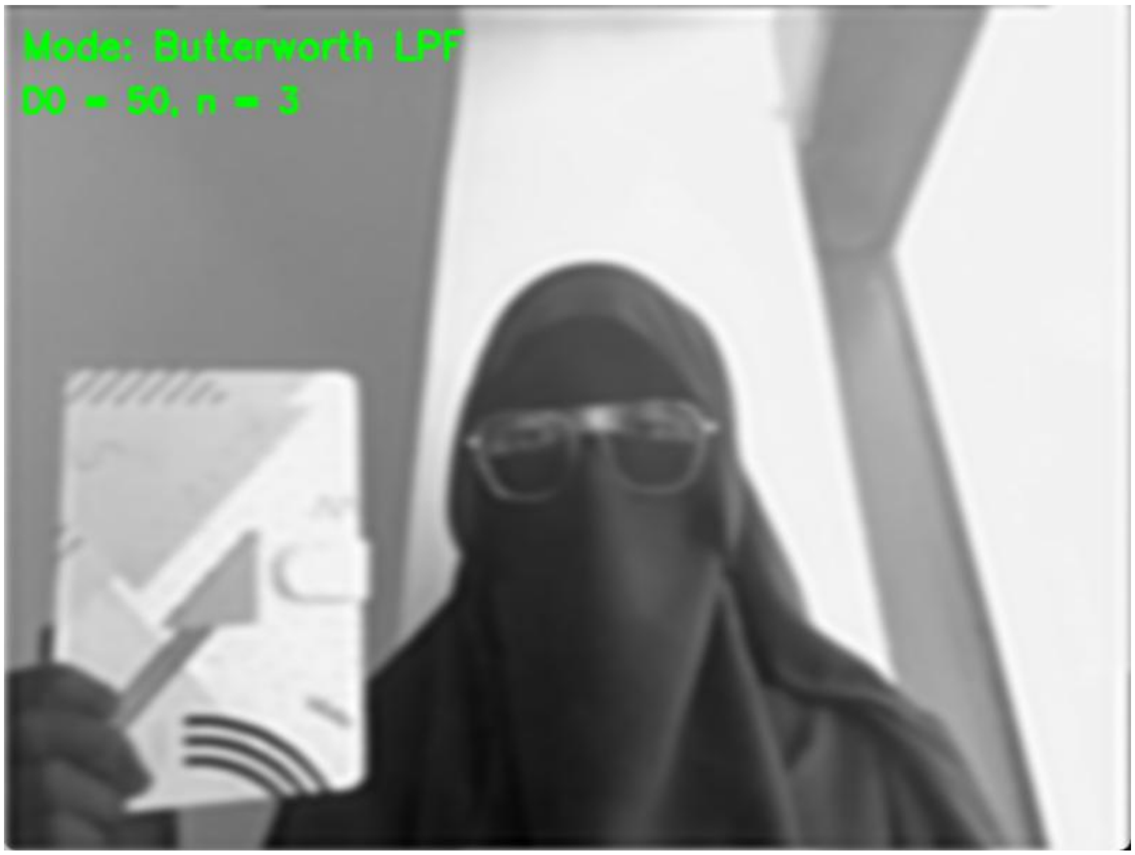


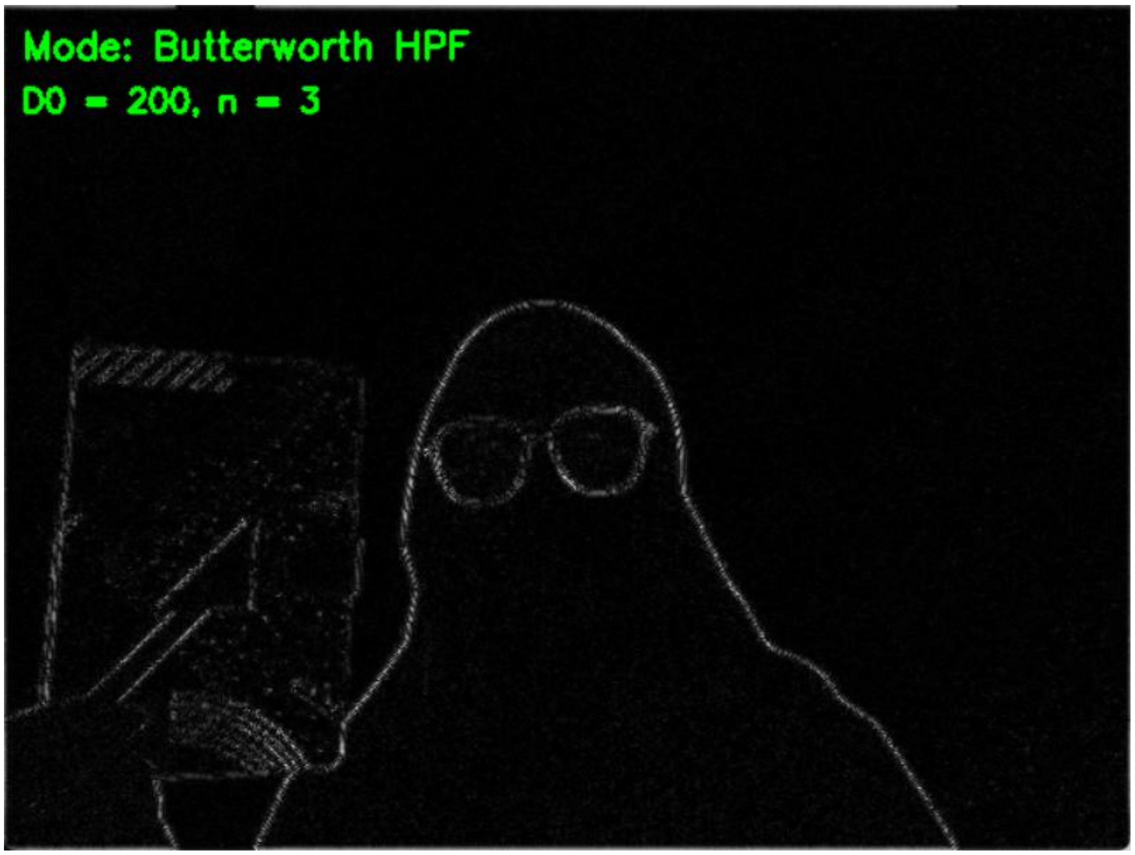
Mode: Butterworth LPF
D0 = 50, n = 1

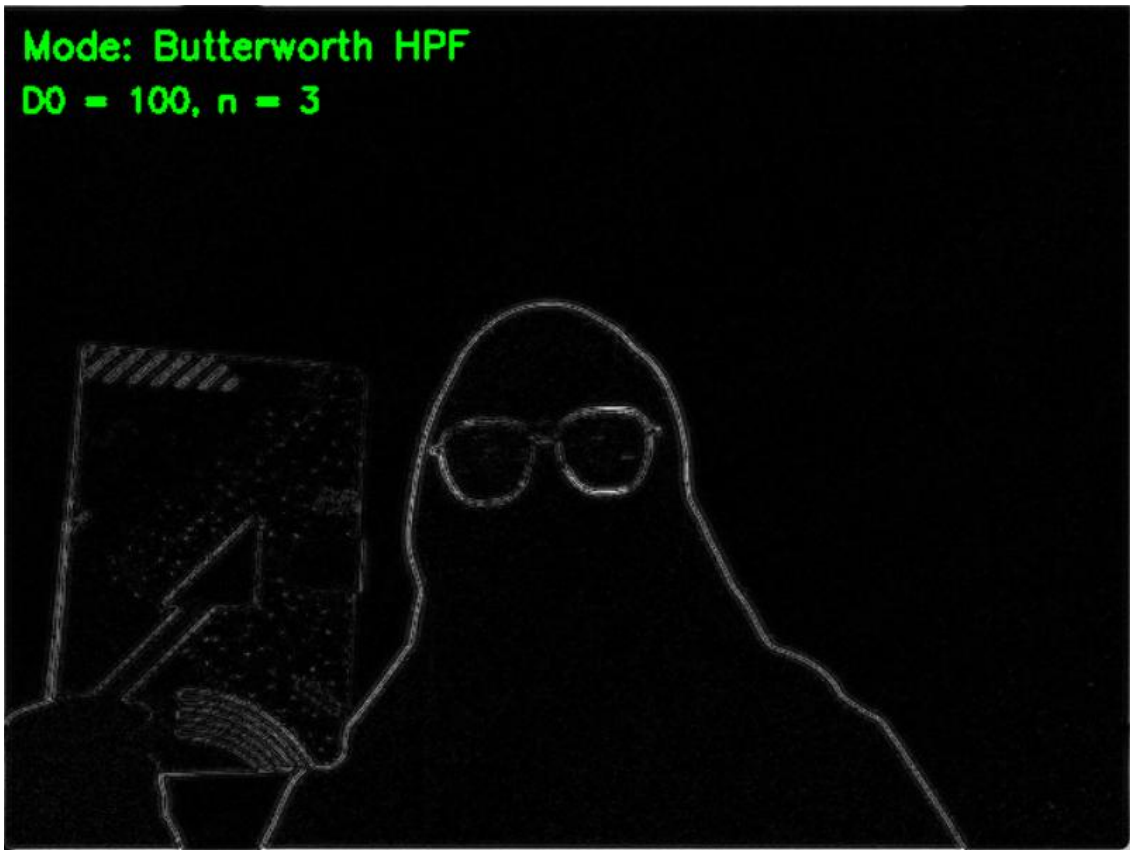

Mode: Butterworth LPF
D0 = 20, n = 3

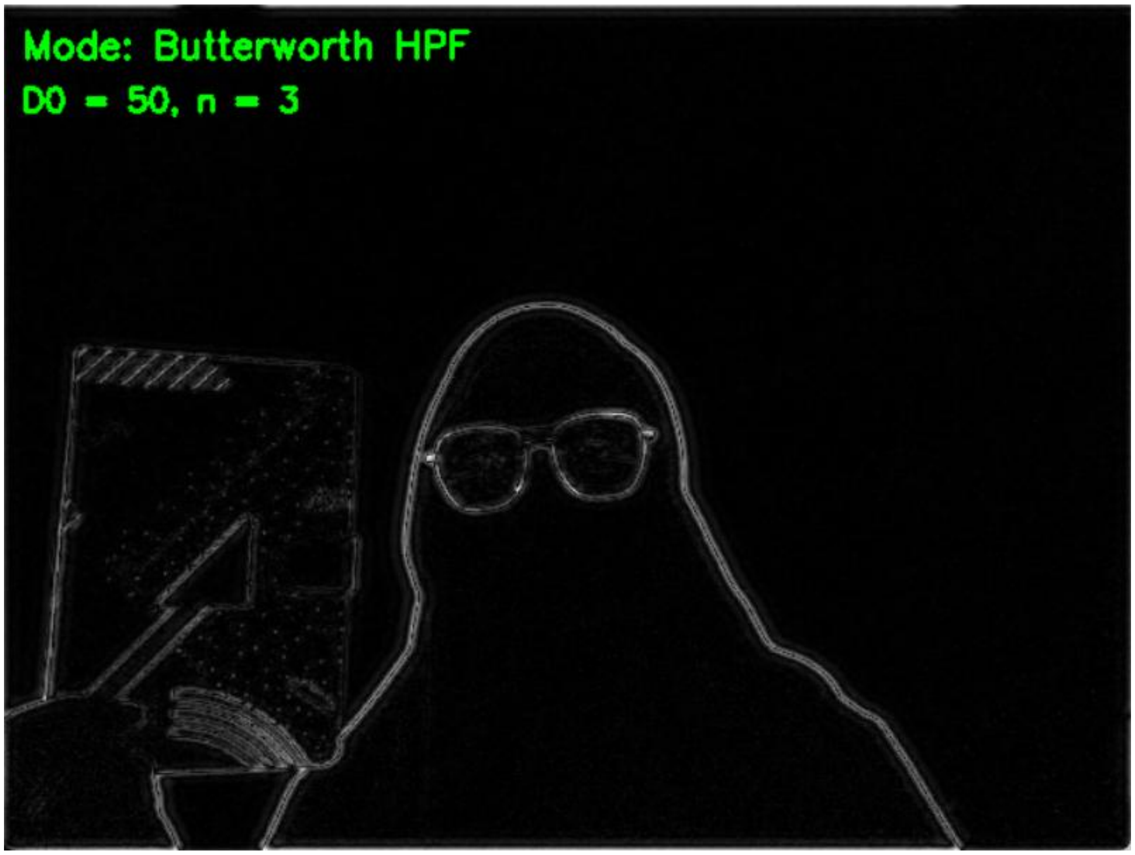

Mode: Butterworth LPF
D0 = 200, n = 3

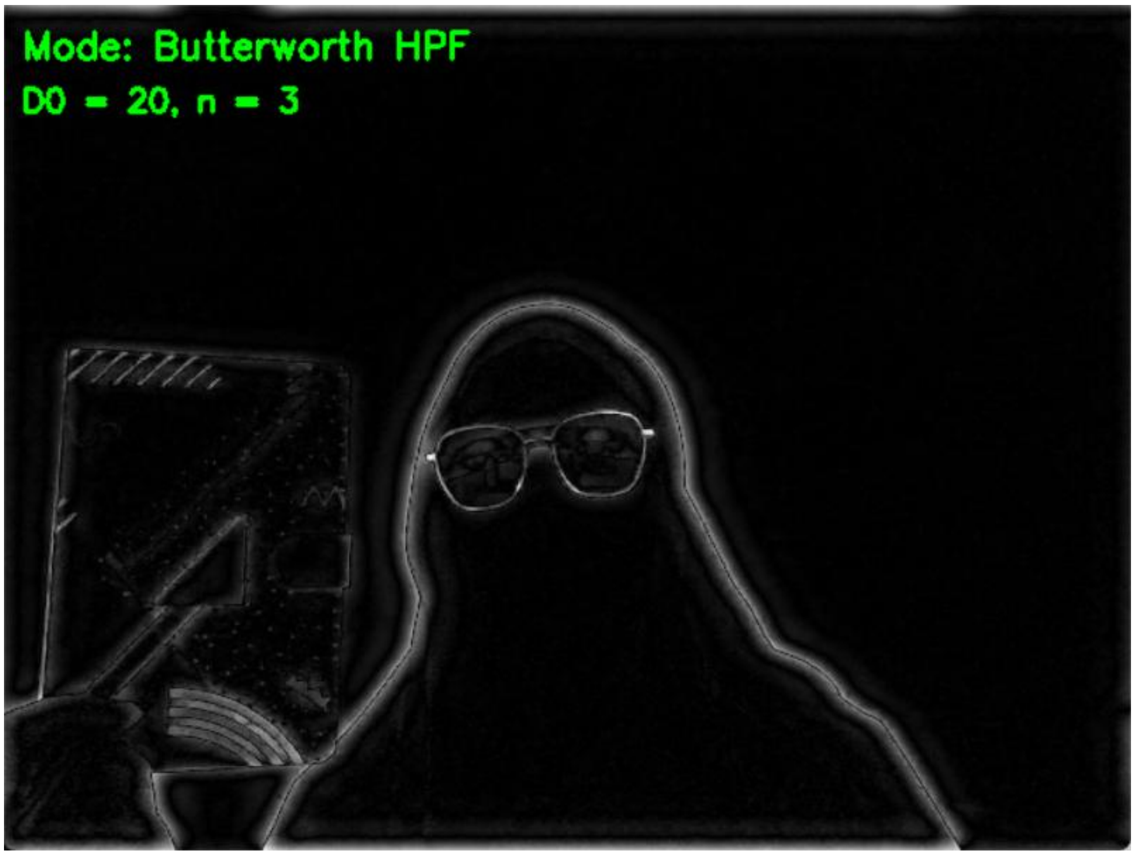

Mode: Butterworth LPF
D0 = 100, n = 3

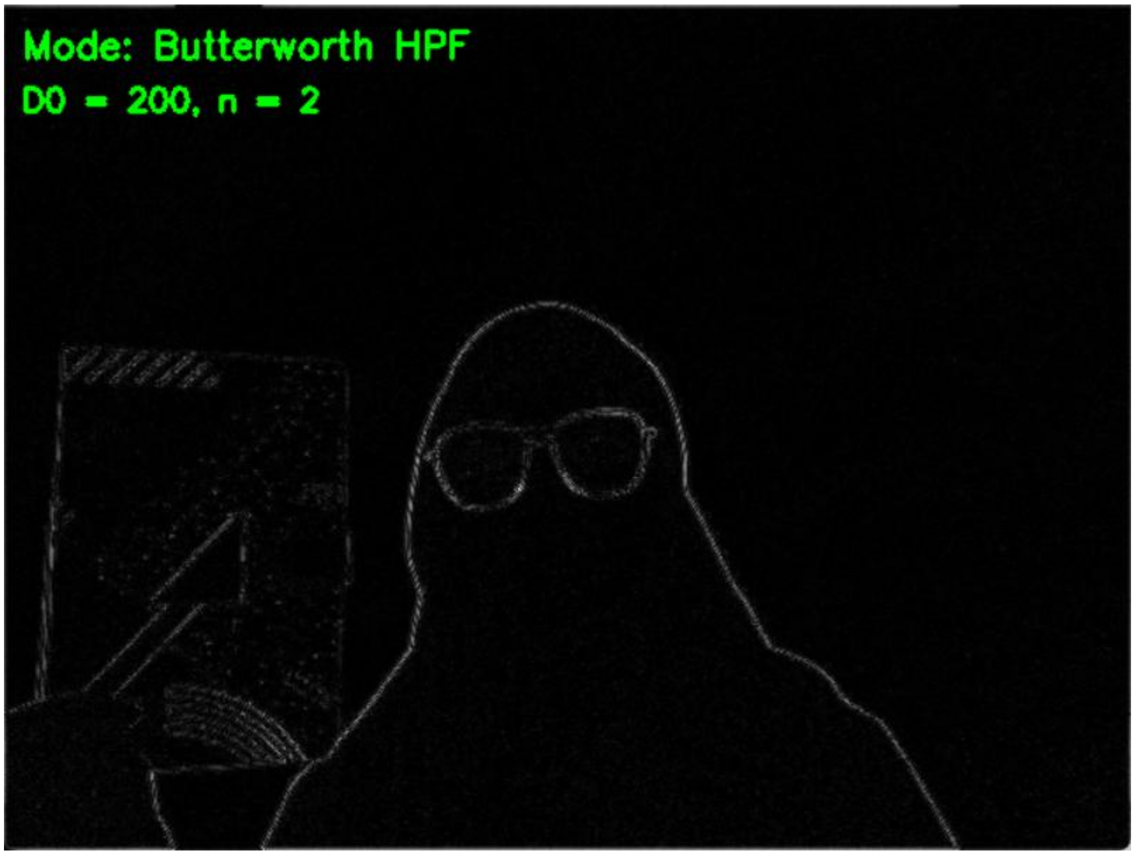

Mode: Butterworth LPF
D0 = 50, n = 3

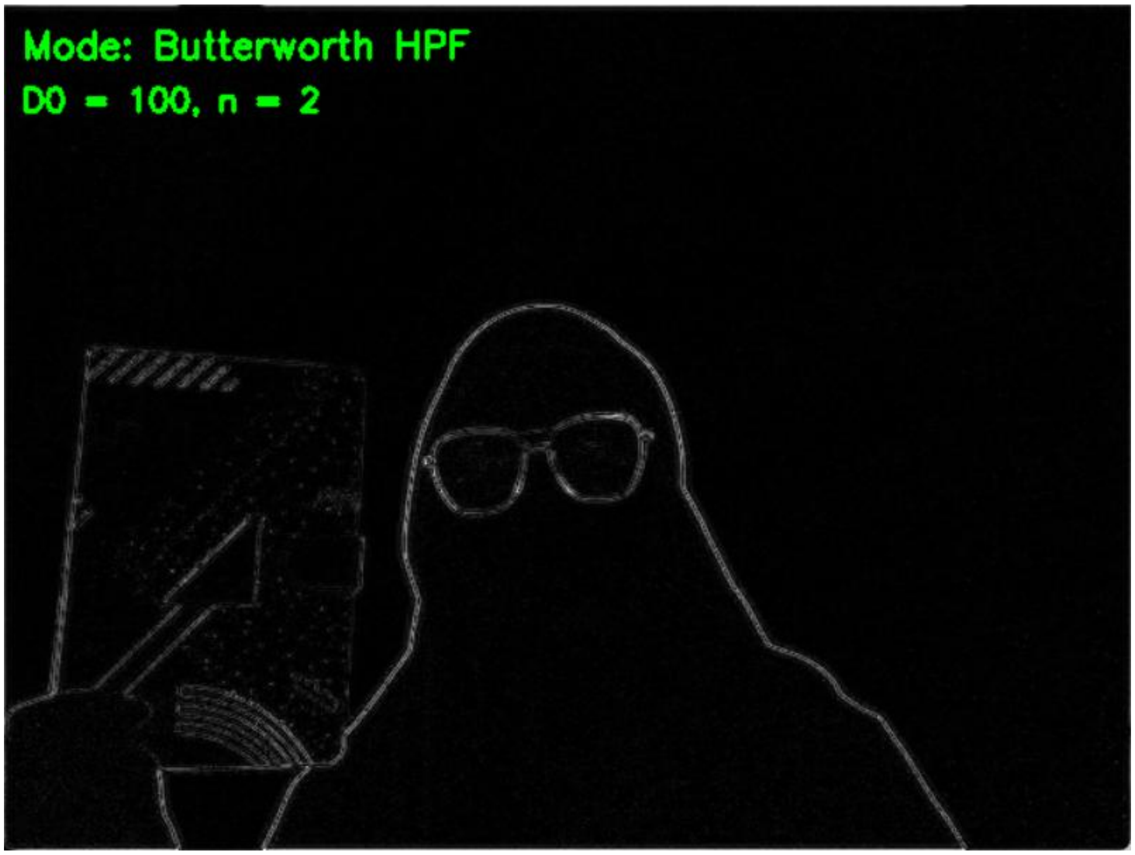

Mode: Butterworth HPF
D0 = 200, n = 3

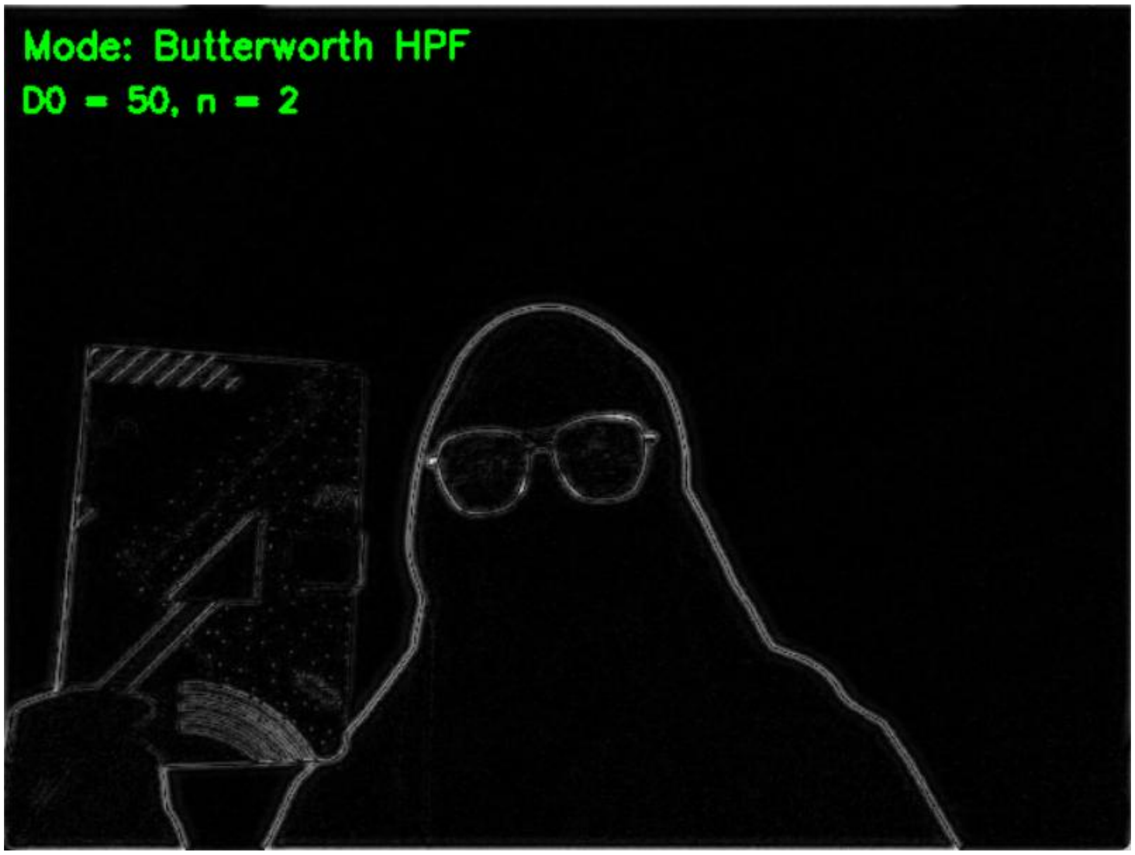

Mode: Butterworth HPF
D0 = 100, n = 3


Mode: Butterworth HPF
D0 = 50, n = 3

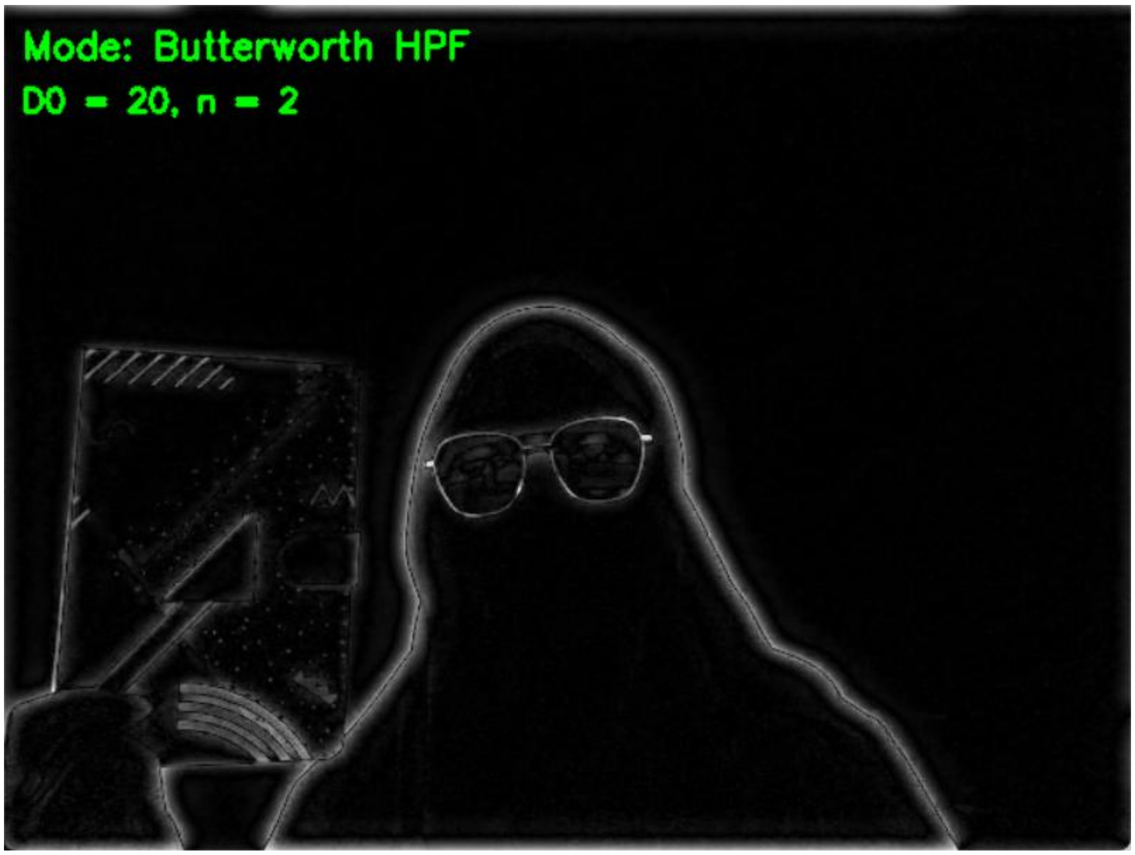

Mode: Butterworth HPF
D0 = 20, n = 3


Mode: Butterworth HPF
D0 = 200, n = 2

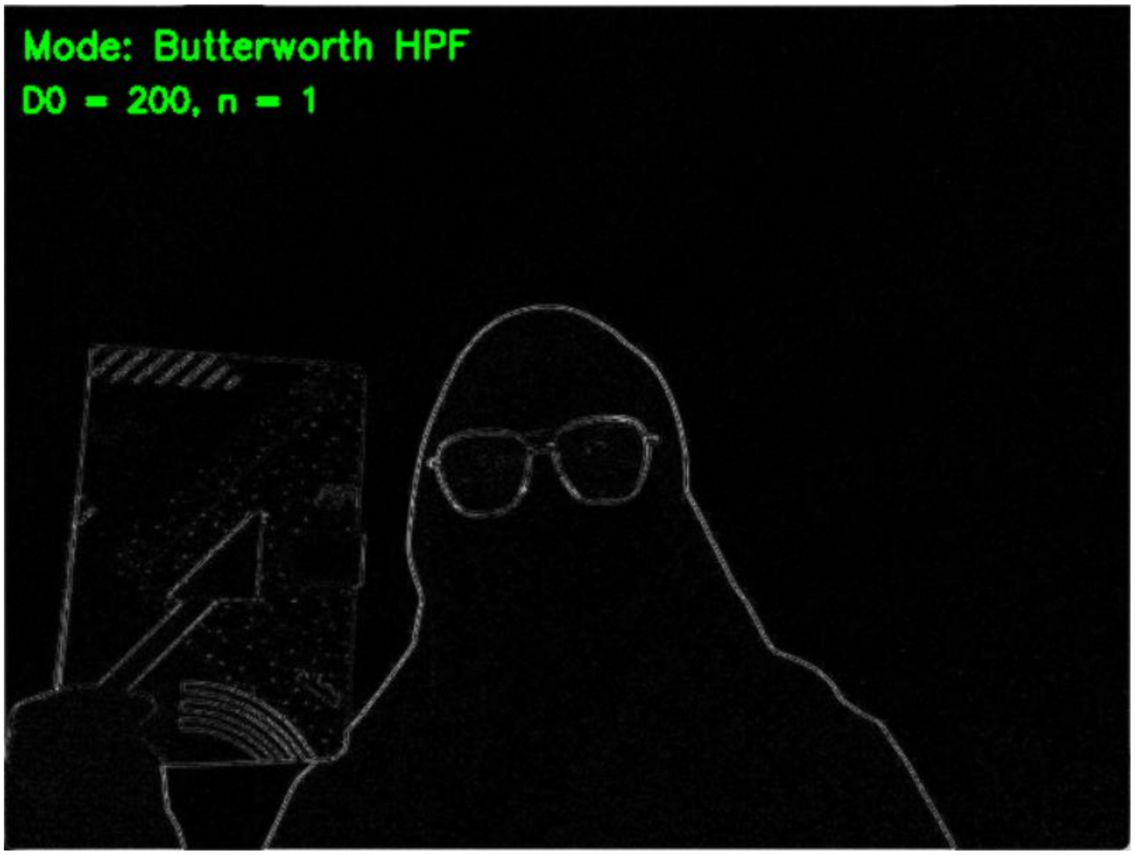

Mode: Butterworth HPF
D0 = 100, n = 2


Mode: Butterworth HPF

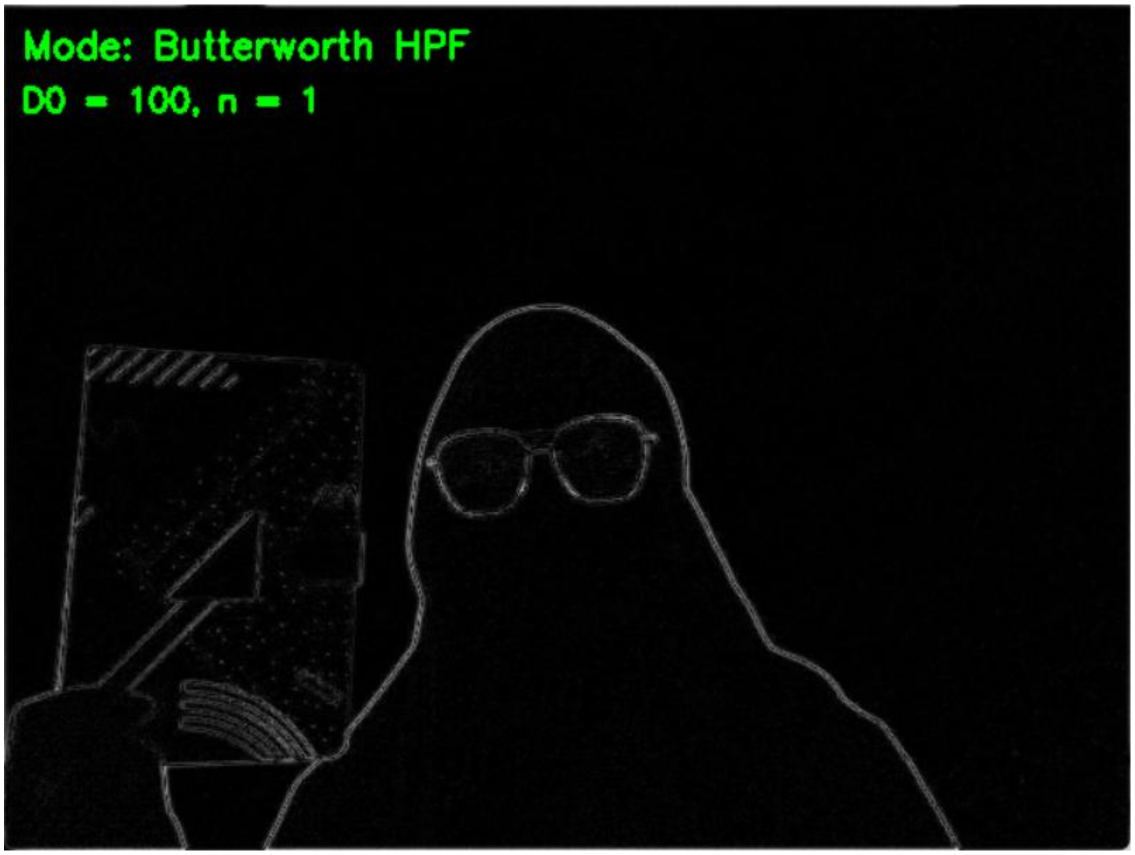
D0 = 50, n = 2


Mode: Butterworth HPF

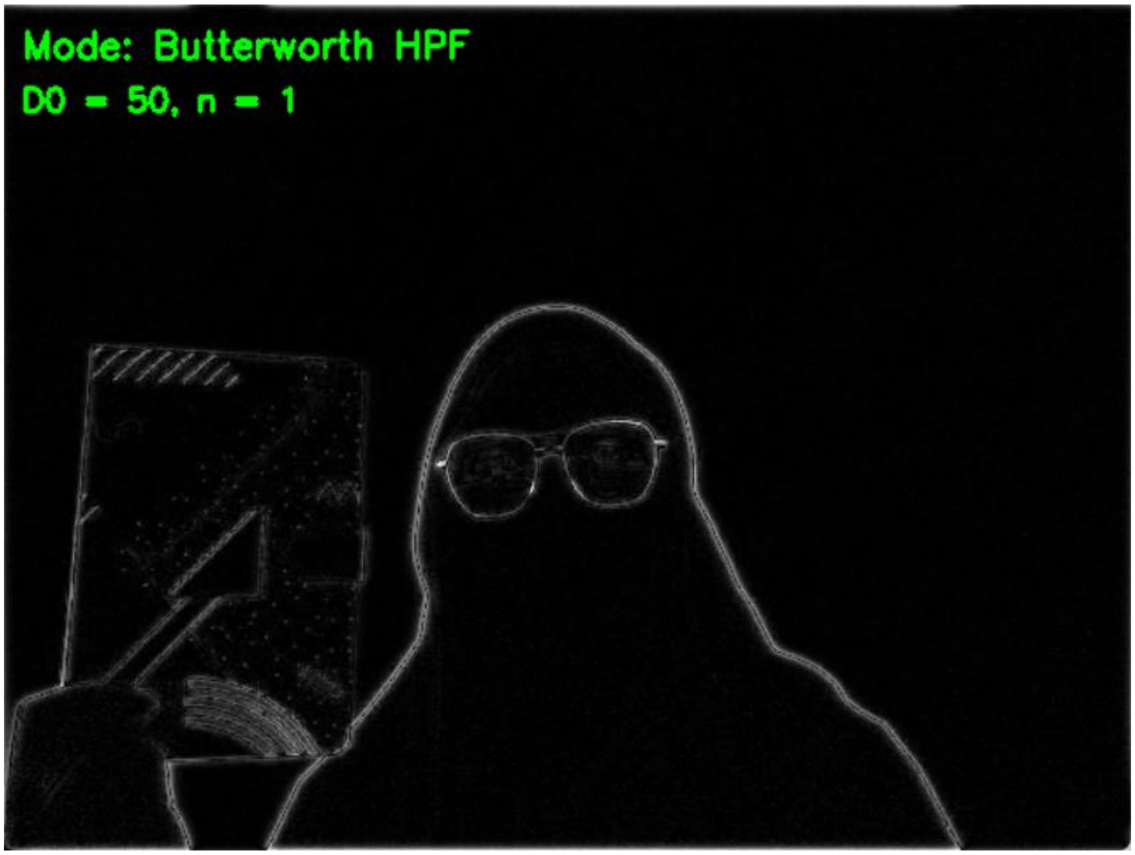
D0 = 20, n = 2

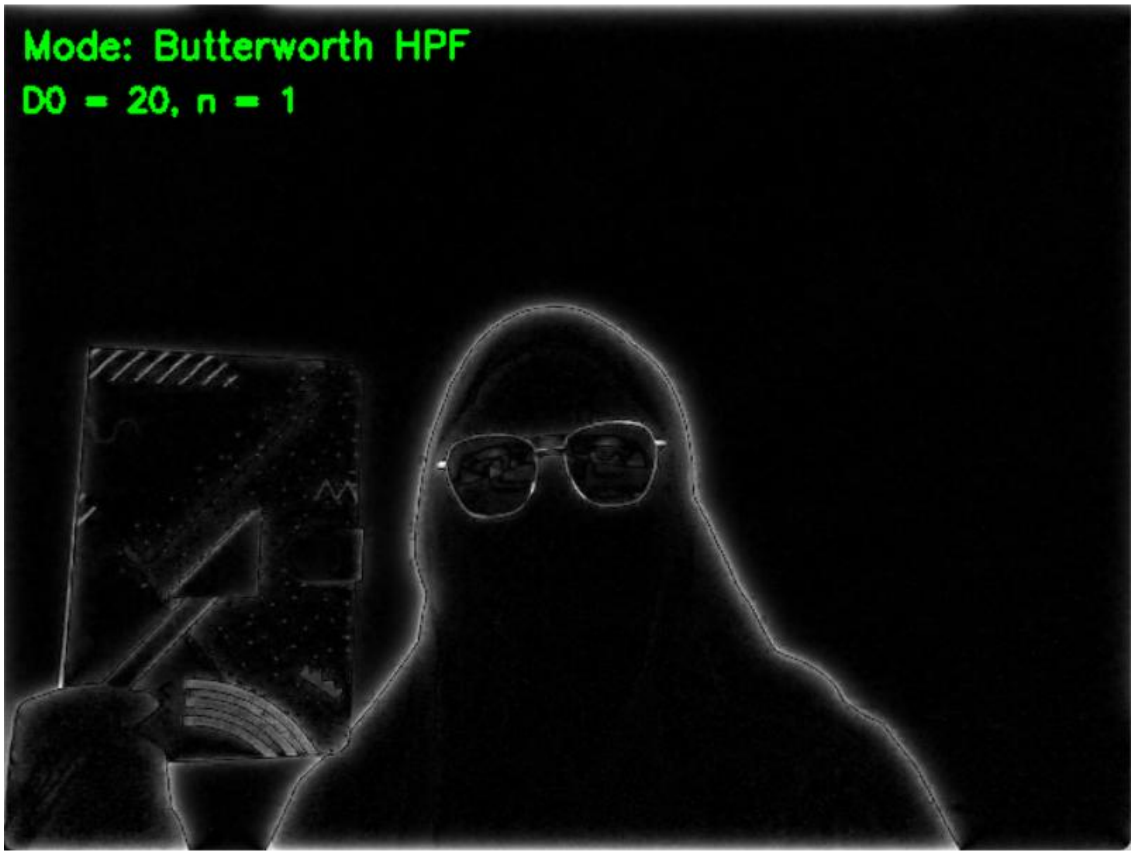

Mode: Butterworth HPF

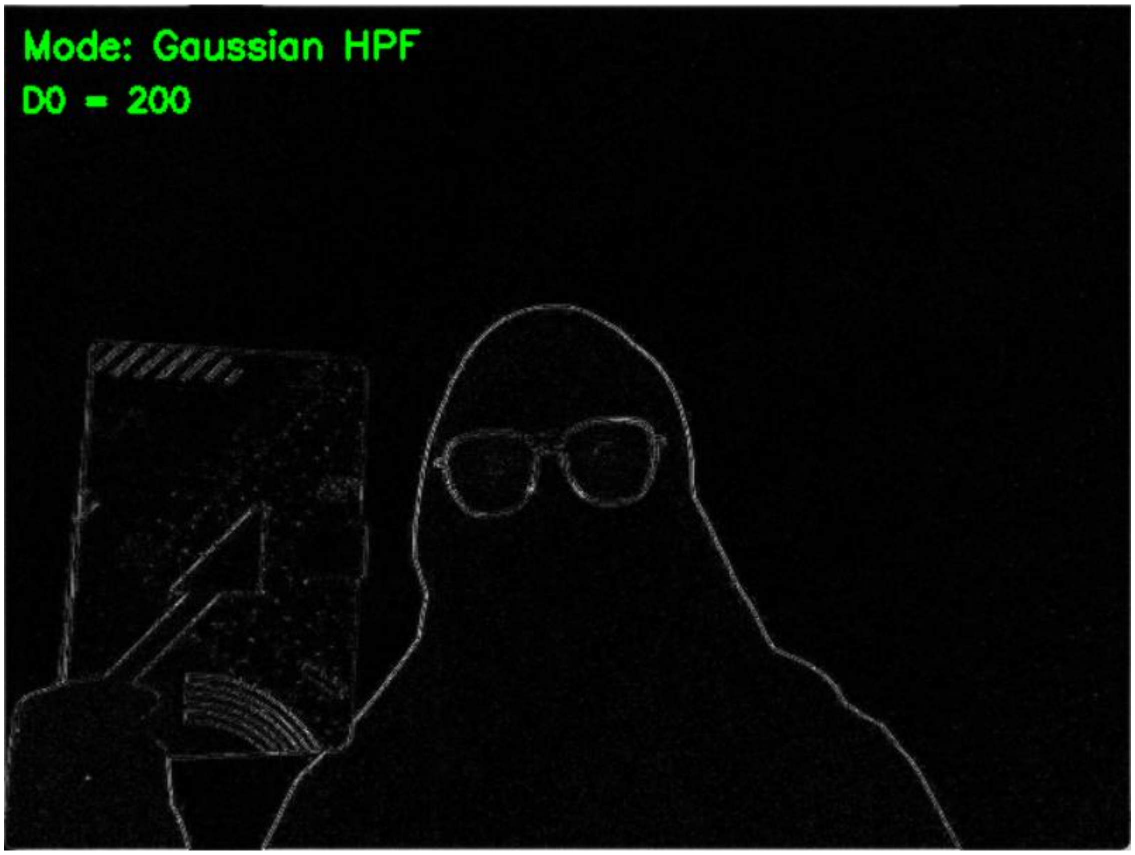
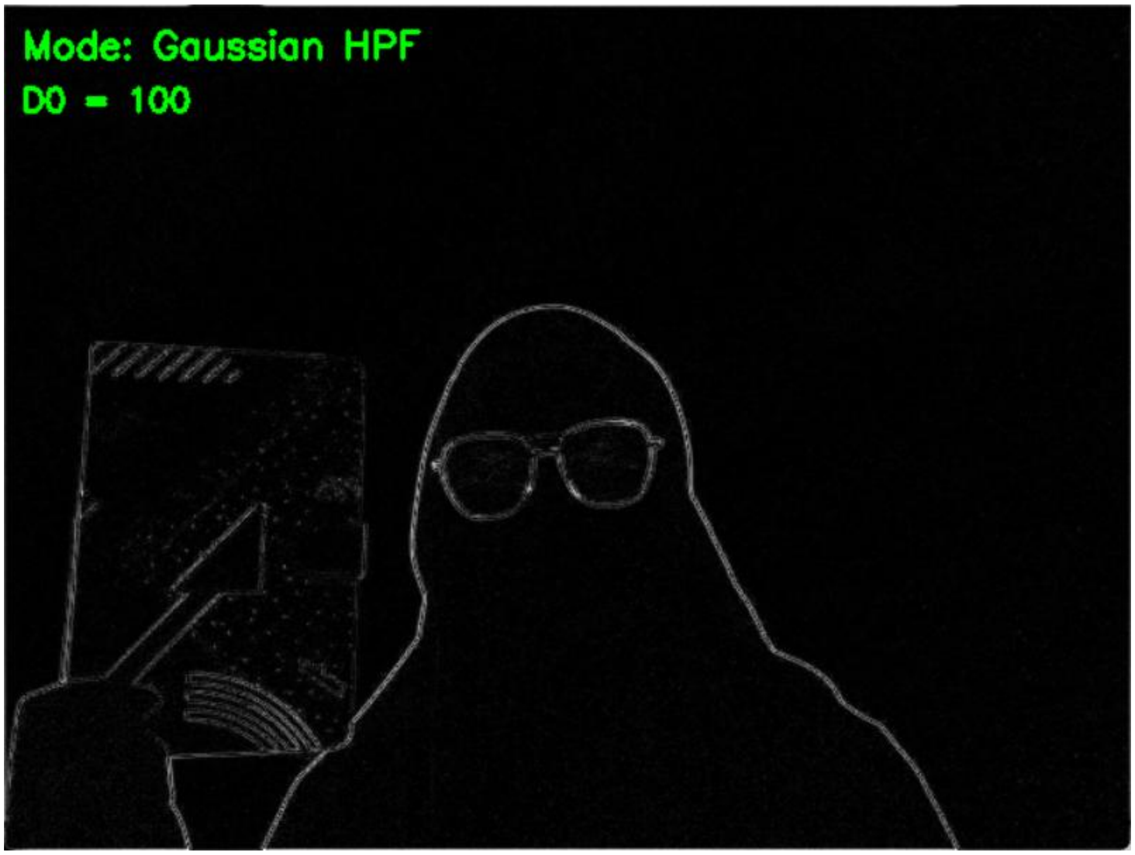
D0 = 200, n = 1

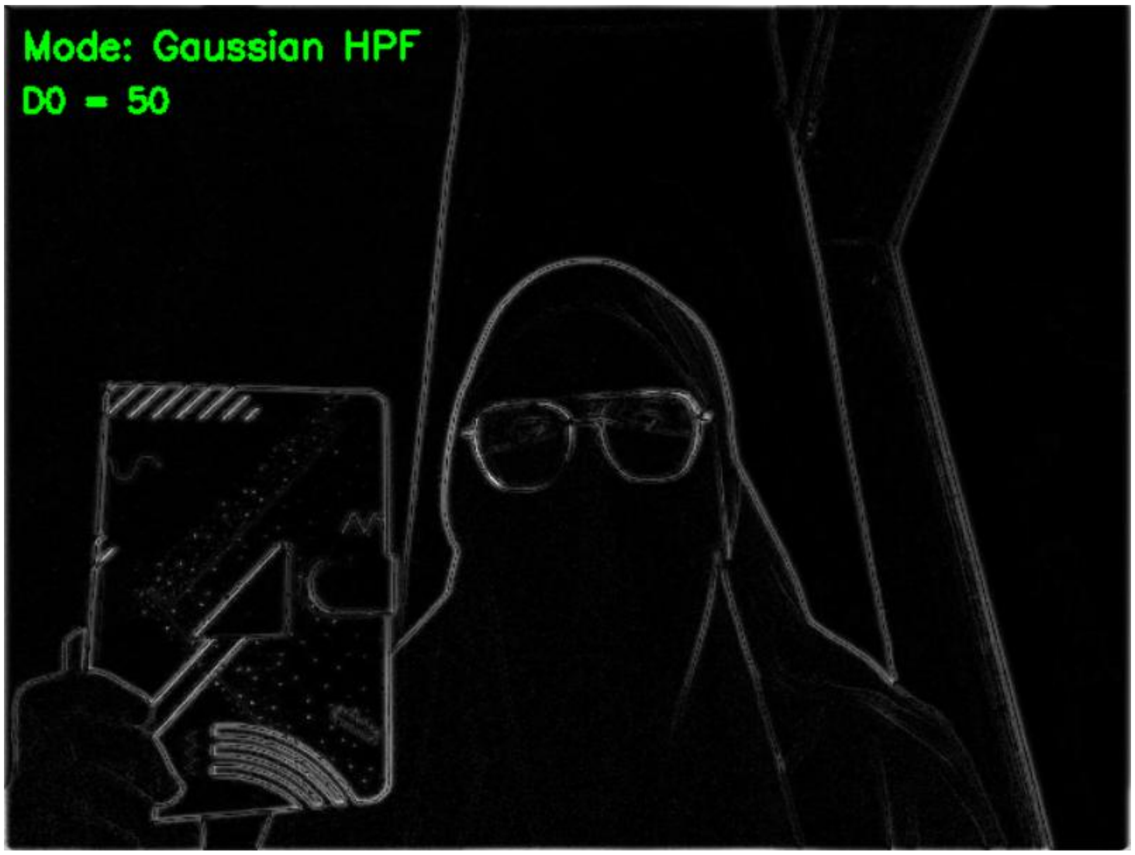
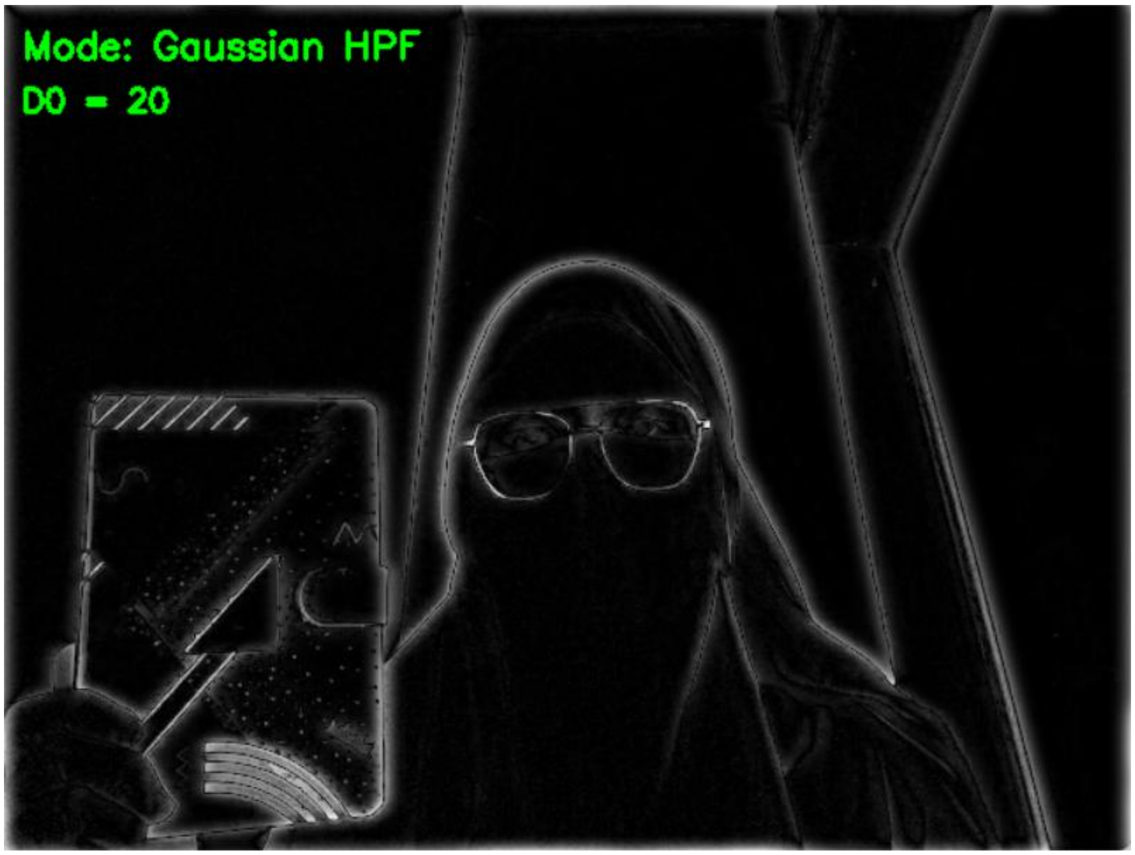

Mode: Butterworth HPF

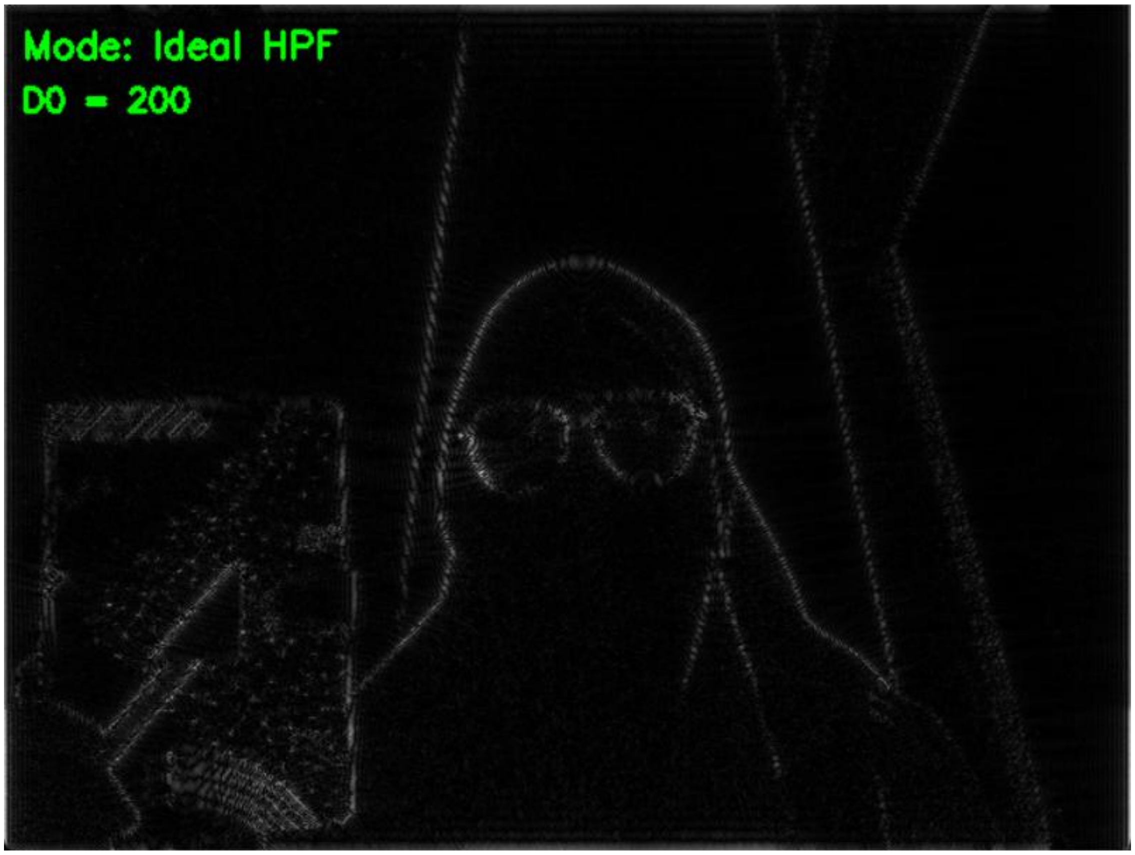
D0 = 100, n = 1

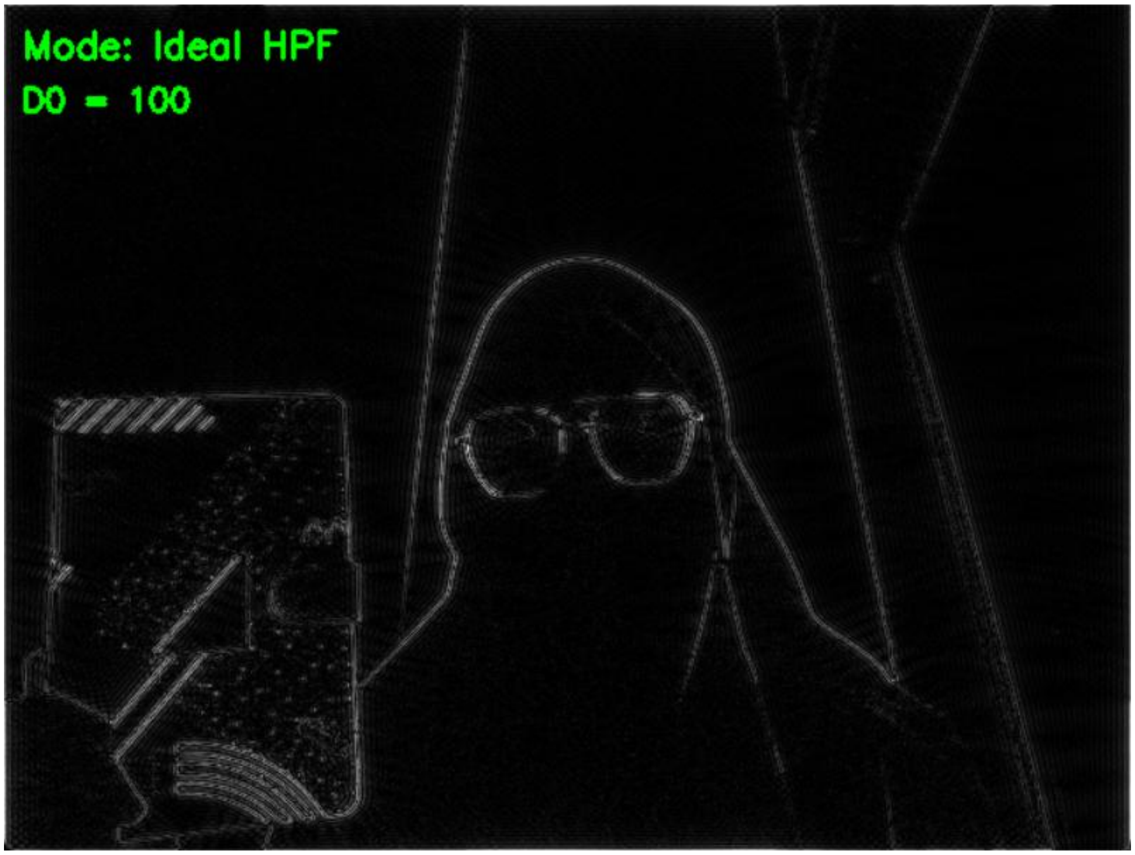

Mode: Butterworth HPF
D0 = 50, n = 1

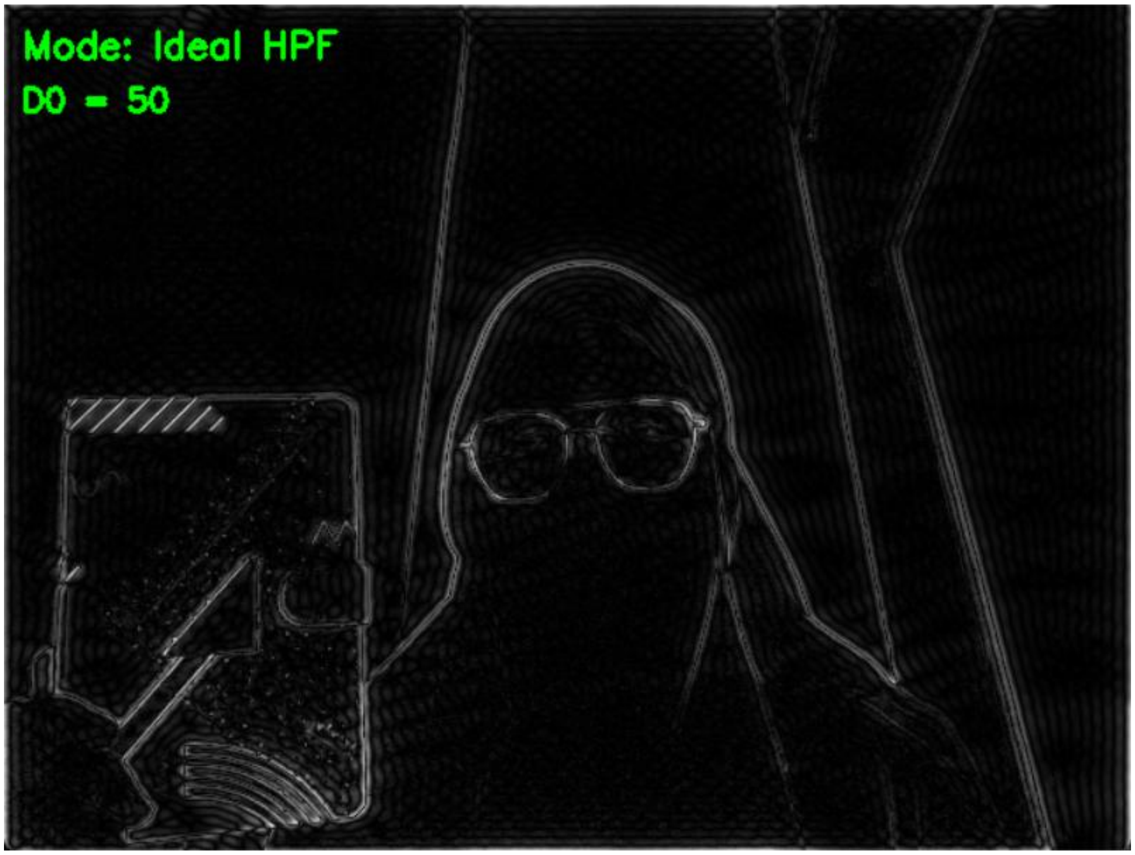

Mode: Butterworth HPF
D0 = 20, n = 1

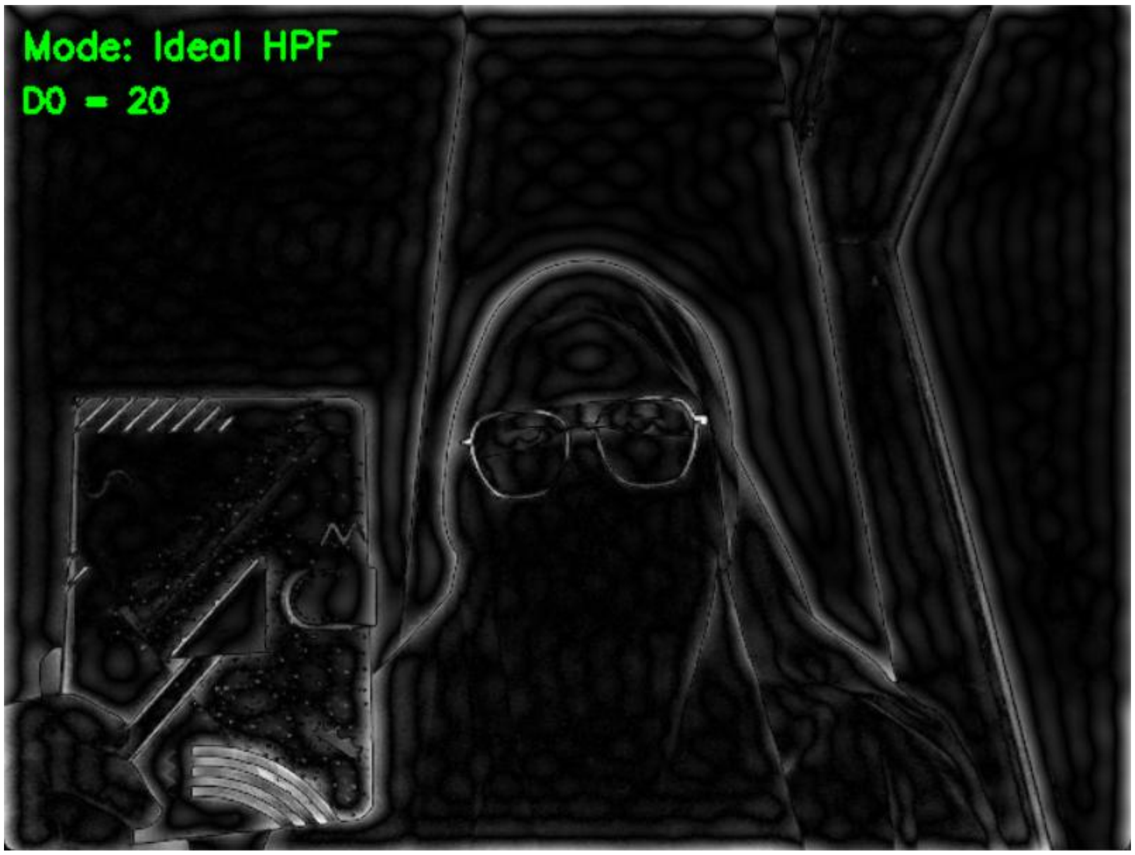

Mode: Gaussian HPF
D0 = 200

Mode: Gaussian HPF
D0 = 100


Mode: Gaussian HPF
D0 = 50

Mode: Gaussian HPF
D0 = 20


Mode: Ideal HPF
D0 = 200


Mode: Ideal HPF
D0 = 100


Mode: Ideal HPF
D0 = 50


Mode: Ideal HPF
D0 = 20