Comprehensive Documentation Report for Real-Time Image Processing Application

N6me: Abdelrahman Abdalla Atta **ID:** 223101276

Under supervised by: Dr. Mahmoud Zaki

Introduction

This document provides a detailed, line-by-line analysis and documentation for the provided Python script, which implements a **Real-Time Image Processing Application** using the `opencv-python` (`cv2`) and `numpy` libraries. The code's primary function is to capture a live video stream from a webcam and allow the user to interactively apply a wide range of image processing filters, covering spatial domain, order statistics, and frequency domain techniques.

The documentation is structured into four main sections, corresponding to the logical blocks in the code, to facilitate easy cross-referencing with screenshots of the source code.

---

1. Libraries, Utilities, and Basic Filters (Lines 2-44)

This section covers the necessary imports, utility functions for image manipulation, and the implementation of fundamental spatial domain filters, including blurring, edge detection, and intensity transformations.

```python
import cv2
import numpy as np

def to_gray(img):
    return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

def overlay(img, text):
    cv2.putText(img, text, (10, 30),
                cv2.FONT_HERSHEY_SIMPLEX, 0.8,
                (255, 255, 255), 2)

# Basic Filters

def blur_filter(img, k):
    return cv2.GaussianBlur(img, (k, k), 0)

def gray_filter(img):
    return to_gray(img)

def canny_filter(img):
    return cv2.Canny(to_gray(img), 100, 200)

def negative_filter(img):
    return 255 - img

def binary_threshold(img):
    gray = to_gray(img)
    _, th = cv2.threshold(gray, 120, 255, cv2.THRESH_BINARY)
    return th

def sobel_filter(img):
    gray = to_gray(img)
    sx = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
    sy = cv2.Sobel(gray, cv2.CV_64F, 0, 1)
    mag = cv2.magnitude(sx, sy)
    return np.uint8(mag / mag.max() * 255)
```

Line(s)	Code Snippet	Explanation
2-3	`import cv2` `import numpy as np`	Imports: Imports the OpenCV library (`cv2`) for image processing and the NumPy library (`np`) for efficient numerical array operations.
9-10	`def to_gray(img):` `return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)`	**`to_gray`** Function: Converts an input image (`img`) from the default `BGR` (Blue, Green, Red) color space used by OpenCV to a single-channel grayscale image.
12-15	`def overlay(img, text):` `cv2.putText(img, text, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 255, 255), 2)`	**`overlay`** Function: Draws status text onto the image. It places the given `text` at coordinates (10, 30), using a simple font, size 0.8, white color (255, 255, 255), and a thickness of 2. This is used to display the active filter mode and parameters.
21-22	`def blur_filter(img, k): return cv2.GaussianBlur(img, (k, k), 0)`	**`blur_filter`** Function: Applies a Gaussi6n Blur to the image. The blur is defined by a kernel size of `k x k`. Gaussian `blur` is a common technique for smoothing images and reducing noise.
24-25	`def gray_filter(img):` `return to_gray(img)`	**`gray_filter`** Function: A simple wrapper that calls the `to_gray` utility function to return the grayscale version of the image.
27-28	`def canny_filter(img):` `return cv2.Canny(to_gray(img), 100, 200)`	**`canny_filter`** Function: Implements the C6nny Edge Detection algorithm. It first converts the image to grayscale and then applies the Canny algorithm using fixed low (100) and high (200) thresholds.

30-31	`def negative_filter(img):` `return 255 - img`	**`negative_filter`** Function: Calculates the Im6ge Neg6tive. For an 8-bit image (0-255), subtracting the pixel value from 255 inverts the intensity.
33-36	`def binary_threshold(img):` `gray = to_gray(img)`	**`binary_threshold`** Function: Converts the image to a Bin6ry Im6ge. It converts to

Line(s)	Code Snippet	Expl6n6tion
	`_, th = cv2.threshold(gray, 120, 255, cv2.THRESH_BINARY)` `return th`	grayscale, then applies a threshold at 120, setting all pixels above 120 to 255 (white) and below to 0 (black).

38-43	`def sobel_filter(img):`
	`gray = to_gray(img)`
	`sx = cv2.Sobel(gray, cv2.CV_64F, 1, 0)`
	`sy = cv2.Sobel(gray, cv2.CV_64F, 0, 1)`
	`mag = cv2.magnitude(sx, sy)`
	`return np.uint8(mag / mag.max() * 255)`
		`sobel_filter` **Function:** Computes the **Sobel Gr6dient M6gnitude** for edge detection. It calculates the horizontal (`sx`) and vertical (`sy`) derivatives, finds the magnitude of the gradient vector (`mag`), and then normalizes the result to the 0-255 range for display.

2. Mean Filters and Order Statistics Filters (Lines 49-90)

This section details the implementation of various noise reduction filters, including different types of mean filters and order statistics filters (based on sorting pixel values).

```python
# Mean Filters

def arithmetic_mean(gray, k):
    return cv2.blur(gray, (k, k))

def geometric_mean(gray, k):
    gray = gray.astype(np.float32) + 1
    return np.exp(cv2.blur(np.log(gray), (k, k))).astype(np.uint8)

def harmonic_mean(gray, k):
    gray = gray.astype(np.float32) + 1
    return ((k*k) / cv2.blur(1/gray, (k, k))).astype(np.uint8)

def contraharmonic_mean(gray, k, Q=1.5):
    num = cv2.blur(gray**(Q+1), (k, k))
    den = cv2.blur(gray**Q, (k, k)) + 1e-9
    return (num / den).astype(np.uint8)

# Order Statistics Filters

def min_filter(gray, k):
    return cv2.erode(gray, np.ones((k, k), np.uint8))

def max_filter(gray, k):
    return cv2.dilate(gray, np.ones((k, k), np.uint8))

def midpoint_filter(gray, k):
    return ((min_filter(gray, k).astype(np.int16) +
             max_filter(gray, k).astype(np.int16)) // 2).astype(np.uint8)

def alpha_trimmed(gray, k, d=4):
    pad = k // 2
    padded = np.pad(gray, pad, mode='edge')
    out = np.zeros_like(gray)

    for i in range(gray.shape[0]):
        for j in range(gray.shape[1]):
            window = padded[i:i+k, j:j+k].flatten()
            window.sort()
            window = window[d//2: len(window)-d//2]
            out[i, j] = np.mean(window)
    return out.astype(np.uint8)
```

Mean Filters (Lines 49-63)

Line(s)	Code Snippet	Explanation
49-50	`def arithmetic_mean(gray, k):` `return cv2.blur(gray, (k, k))`	**Arithmetic Mean Filter:** Implements the standard arithmetic mean (average) filter over a `k x k` neighborhood using `cv2.blur`. This is effective for reducing Gaussian noise.

52-54	`def geometric_mean(gray, k):`
	`gray = gray.astype(np.float32) + 1`
	`return np.exp(cv2.blur(np.log(gray), (k, k))).astype(np.uint8)`
		Geometric Mean Filter: Calculates the geometric mean. It involves taking the logarithm of the image (after adding 1 to avoid `log(0)`), applying the arithmetic mean (blur), and then taking the exponential. This filter is better at preserving image detail than the arithmetic mean.

56-58	`def harmonic_mean(gray, k):`
	`gray = gray.astype(np.float32) + 1`
	`return ((k*k) / cv2.blur(1/gray, (k, k))).astype(np.uint8)`
		Harmonic Mean Filter: Calculates the harmonic mean. It is effective at reducing "salt" noise (bright outliers) but performs poorly on "pepper" noise (dark outliers).

60-63	`def contraharmonic_mean(gray, k, Q=1.5):`
	`num = cv2.blur(gray**(Q+1), (k, k))`
	`den = cv2.blur(gray**Q, (k, k)) + 1e-9`
	`return (num / den).astype(np.uint8)`
		Contraharmonic Mean Filter: A powerful filter whose behavior is controlled by the order `Q`. If $Q > 0$, it removes salt noise; if $Q < 0$, it removes pepper noise. The formula is $\frac{\sum x^{Q+1}}{\sum x^{Q}}$. A small constant (`1e-9`) is added to the denominator to prevent division by zero.

Order Statistics Filters (Lines 69-90)

Line(s)	Code Snippet	Explanation
69-70	`def min_filter(gray, k): return cv2.erode(gray, np.ones((k, k), np.uint8))`	**Minimum Filter:** Implements the minimum filter, which is equivalent to the **Erosion** morphological operation. It replaces the center pixel with the minimum value in the `k x k` neighborhood. Effective for removing bright noise (salt noise).
72-73	`def max_filter(gray, k): return cv2.dilate(gray, np.ones((k, k), np.uint8))`	**Maximum Filter:** Implements the maximum filter, equivalent to the **Dilation** morphological operation. It replaces the center pixel with the maximum value in the `k x k` neighborhood. Effective for removing dark noise (pepper noise).
75-77	`def midpoint_filter(gray, k): return ((min_filter(gray, k).astype(np.int16) + max_filter(gray, k).astype(np.int16)) // 2).astype(np.uint8)`	**Midpoint Filter:** Calculates the average of the minimum and maximum values in the neighborhood. It first converts to `np.int16` to prevent overflow during addition, then performs integer division by 2, and finally converts back to `np.uint8` .
79-90	`def alpha_trimmed(gray, k, d=4):` *(Loop implementation)*	**Alpha-Trimmed Mean Filter:** This custom implementation iterates through every pixel. For each pixel, it takes the `k x k` neighborhood, sorts the pixel values, and removes the `d/2` smallest and `d/2` largest values (the “alpha trim”). The output pixel is the mean of the remaining values. This is highly effective for mixed noise types.

---

3. Frequency Domain Filters (Lines 96-147)

This section implements various filters in the frequency domain using the Discrete Fourier Transform (DFT). These filters are used for both smoothing (Low-Pass) an sharpening (High-Pass) by manipulating the image's frequency spectrum.

Low-Pass and High-Pass Filters (Lines 113-127)

These filters are used for smoothing (Low-Pass) or sharpening (High-Pass). D_0 is the cutoff frequency.

```python
# Frequency Domain

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

def DFT_process(gray, H):
    dft = cv2.dft(np.float32(gray), flags=cv2.DFT_COMPLEX_OUTPUT)
    dft = np.fft.fftshift(dft)
    dft[:, :, 0] *= H
    dft[:, :, 1] *= H
    img_back = cv2.idft(np.fft.ifftshift(dft))
    img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
    img_back = img_back / img_back.max() * 255
    return img_back.astype(np.uint8)

#  Low / High Pass
# Any point inside circle = 1  and out the circle is 0
def ILPF(gray, D0):
    D = D_uv(*gray.shape)
    return DFT_process(gray, (D <= D0).astype(np.float32))
# gaussian curve
def GLPF(gray, D0):
    D = D_uv(*gray.shape)
    return DFT_process(gray, np.exp(-(D**2)/(2*D0**2)))

def BLPF(gray, D0, n=2): # if  n increase it will be close to ideal and make balance between soft the noise and save the details
    D = D_uv(*gray.shape)
    return DFT_process(gray, 1/(1+(D/D0)**(2*n)))
# high pass it's a reverse of low pass
def IHPF(gray, D0): return 255 - ILPF(gray, D0)
def GHPF(gray, D0): return 255 - GLPF(gray, D0)
def BHPF(gray, D0): return 255 - BLPF(gray, D0)
```

Core Frequency Domain Utilities (Lines 96-109)

Line(s)	Code Snippet	Explanation

96-99	`def D_uv(rows, cols):`
	`crow, ccol = rows // 2, cols // 2`
	`y, x = np.ogrid[:rows, :cols]`
	`return np.sqrt((x - ccol)**2 + (y - crow)**2)`
		`D_uv` **Function:** Calculates the **Euclidean distance** matrix from the center of the frequency spectrum. This distance (`D`) is the key variable used to define the shape of all frequency domain filters.

101-109	`def DFT_process(gray, H):` *(DFT, Shift, Multiply, IDFT)*	`DFT_process` **Function:** The core filtering engine. It takes the grayscale image and the filter transfer function (`H`) as input. It performs the following steps: 1. Computes the DFT. 2. Shifts the zero-frequency component to the center (`fftshift`). 3. Multiplies the DFT by the filter function `H` . 4. Performs the Inverse DFT (`idft`). 5. Calculates the magnitude of the result and normalizes it to the 0-255 range.

Line(s)	Code Snippet	Filter Type	
113-115	`def ILPF(gray, D0):`	**Ideal Low-Pass Filter (ILPF):** The filter function is a simple binary mask: 1 inside the circle of radius D_0, and 0 outside. This causes ringing artifacts in the spatial domain.	
117-119	`def GLPF(gray, D0):`	**Gaussian Low-Pass Filter (GLPF):** Uses a smooth Gaussian function, which minimizes ringing artifacts compared to the Ideal filter.	
121-123	`def BLPF(gray, D0, n=2):`	**Butterworth Low-Pass Filter (BLPF):** Provides a transition between the Ideal and Gaussian filters, controlled by the order n.	
125	`def IHPF(gray, D0):`	**Ideal High-Pass Filter (IHPF):** Implemented by inverting the result of the corresponding Low-Pass filter (`255 - ILPF(...)`). High-Pass filters enhance edges and details.	
126	`def GHPF(gray, D0):`	**Gaussian High-Pass Filter (GHPF):** Inversion of GLPF.	
127	`def BHPF(gray, D0):`	**Butterworth High-Pass Filter (BHPF):** Inversion of BLPF.	

Band-Reject and Band-Pass Filters (Lines 131-147)

These filters are used to remove (Reject) or pass (Pass) a specific band of frequencies, typically to eliminate periodic noise. D_0 is the center frequency, and W is the bandwidth.

```python
# Band Reject / Pass



# D0 = desired center of the band
# W = bandwidth
#result: A psychological image of the band's end , but the selected frequencies will
def IBRF(gray, D0, W):
    D = D_uv(*gray.shape)
    H = np.ones(gray.shape)
    H[np.abs(D-D0) <= W/2] = 0
    return DFT_process(gray, H)
# same idea but allowing and blocking is seamless  , it reduces the selected frequen
def GBRF(gray, D0, W):
    D = D_uv(*gray.shape) + 1e-9
    return DFT_process(gray, 1 - np.exp(-((D**2-D0**2)/(D*W))**2))
#   A flexible approach between ideal and gaussian
# n = filter sharpness (higher - like ideal)
def BBRF(gray, D0, W, n=2):
    D = D_uv(*gray.shape) + 1e-9
    return DFT_process(gray, 1/(1+((D*W)/(D**2-D0**2))**(2*n)))
# reverse of reject it only requested band is allowed, and all other frequencies are
def IBPF(gray, D0, W): return 255 - IBRF(gray, D0, W)
def GBPF(gray, D0, W): return 255 - GBRF(gray, D0, W)
def BBPF(gray, D0, W): return 255 - BBRF(gray, D0, W)
```

Line(s)	Code Snippet	Filter Type	
131-135	`def IBRF(gray, D0, W):`	**Ideal Band-Reject Filter (IBRF):** Creates a filter that is 0 (rejects) within the band $[D_0 - W/2, D_0 + W/2]$ and 1 (passes) elsewhere.	
137-139	`def GBRF(gray, D0, W):`	**Gaussian Band-Reject Filter (GBRF):** Uses a Gaussian function to smoothly reject the frequency band, reducing artifacts.	
141-143	`def BBRF(gray, D0, W, n=2):`	**Butterworth Band-Reject Filter (BBRF):** Uses the Butterworth function to reject the band.	
145	`def IBPF(gray, D0, W):`	**Ideal Band-Pass Filter (IBPF):** Implemented by inverting the result of the corresponding Band-Reject filter (`255 - IBRF(...)`). Band-Pass filters isolate the frequencies within the band.	
146	`def GBPF(gray, D0, W):`	**Gaussian Band-Pass Filter (GBPF):** Inversion of GBRF.	
147	`def BBPF(gray, D0, W):`	**Butterworth Band-Pass Filter (BBPF):** Inversion of BBRF.	

---

4. Camera Loop and Main Execution (Lines 153-244)

This section contains the main execution logic, which initializes the webcam, defines parameters, handles user input, and continuously processes and displays the video stream.

```
# Web_cam

cap = cv2.VideoCapture(0)
mode = None
k = 3
D0 = 30
W = 10

while True:
    ret, frame = cap.read()
    if not ret:
        break

    gray = to_gray(frame)
    out = frame.copy()
    key = cv2.waitKey(1) & 0xFF

    if key == ord('q'): break
    elif key == ord('n'): mode = None
    elif key == ord('+'): k = min(k+2, 31)
    elif key == ord('-'): k = max(k-2, 1)
    elif key == ord(']'): D0 += 5
    elif key == ord('['): D0 = max(5, D0-5)
    elif key == ord('}'): W += 5
    elif key == ord('{'): W = max(5, W-5)

    elif key == ord('b'): mode='blur'
    elif key == ord('r'): mode='gray'
    elif key == ord('e'): mode='canny'
    elif key == ord('u'): mode='negative'
    elif key == ord('t'): mode='binary'
    elif key == ord('k'): mode='sobel'

    elif key == ord('a'): mode='arith'
    elif key == ord('g'): mode='geo'
    elif key == ord('h'): mode='harm'
    elif key == ord('m'): mode='contra'

    elif key == ord('z'): mode='median'
    elif key == ord('x'): mode='min'
    elif key == ord('y'): mode='max'
    elif key == ord('p'): mode='mid'
    elif key == ord('l'): mode='alpha'

    elif key == ord('1'): mode='ilpf'
    elif key == ord('2'): mode='glpf'
    elif key == ord('3'): mode='blpf'
    elif key == ord('4'): mode='ihpf'
    elif key == ord('5'): mode='ghpf'
    elif key == ord('6'): mode='bhpf'
    elif key == ord('7'): mode='ibrf'
    elif key == ord('8'): mode='gbrf'
    elif key == ord('9'): mode='bbrf'
    elif key == ord('o'): mode='ibpf'
    elif key == ord('i'): mode='gbpf'
    elif key == ord('j'): mode='bbpf'
```

Line(s)	Code Snippet	Explanation
153	`cap = cv2.VideoCapture(0)`	**Webcam Initialization:** Creates a `VideoCapture` object to connect to the default camera (index 0).

154-157	`mode = None`
	`k = 3`
	`D0 = 30`
	`W = 10`
		Parameter Initialization: Sets the initial state: `mode` (no filter), `k` (kernel size for spatial filters), `D0` (cutoff frequency for frequency filters), and `W` (bandwidth for band filters).

159	`while True:`	**Main Loop Start:** Begins the infinite loop for continuous video processing.
160-162	`ret, frame = cap.read()` `if not ret: break`	**Frame Reading:** Reads a frame from the camera. If `ret` is `False` (e.g., camera disconnected), the loop breaks.
164-165	`gray = to_gray(frame)` `out = frame.copy()`	**Preprocessing:** Converts the frame to grayscale and creates a copy of the original color frame (`out`) to be used as the base for filtering.
166	`key = cv2.waitKey(1) & 0xFF`	**Key Input:** Waits for 1 millisecond for a key press. The `& 0xFF` ensures the key code is correctly captured.

168-175	`if key == ord('q'): break`
	`elif key == ord('n'): mode = None`
	`elif key == ord('+'): k = min(k+2, 31)`
	`elif key == ord('-'): k = max(k-2, 1)`
	(and other parameter controls)
		Parameter Control: Handles key presses for quitting (`q`), resetting the filter (`n`), and adjusting the parameters `k` , `D0` , and `W` . `k` is constrained to be an odd number between 1 and 31.

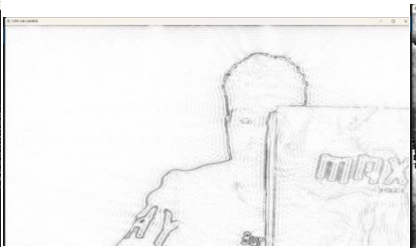
177-206	`elif key == ord('b'): mode='blur'` *(and all other filter mode selections)*	**Filter Selection:** A large block of `elif` statements maps single-character key presses (e.g., 'b', 'a', '1') to the corresponding filter `mode` string.
209-238	`if mode == 'blur': out = blur_filter(frame, k)`	**Processing Block:** This block applies the function corresponding to the currently selected `mode` . Note that most filters return a grayscale image,

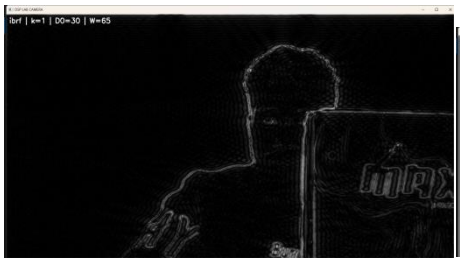
Line(s)	Code Snippet	Explanation
	(and all other filter applications)	which is then converted back to a 3-channel BGR image using `cv2.cvtColor(..., cv2.COLOR_GRAY2BGR)` so it can be displayed alongside the color frame.
240	`overlay(out, f"{mode} | k={k} | D0={D0} | W={W}")`	**Overlay Status:** Calls the `overlay` utility function to display the current filter mode and parameter values on the processed image.
241	`cv2.imshow("DSP LAB CAMERA", out)`	**Display:** Displays the final processed image in a window titled “DSP LAB CAMERA”.
243	`cap.release()`	**Cleanup:** Releases the webcam resource when the loop is exited (e.g., by pressing ‘q’).
244	`cv2.destroyAllWindows()`	**Cleanup:** Closes all OpenCV windows.

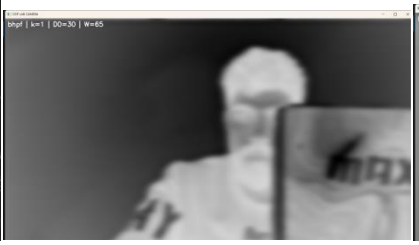
Conclusion

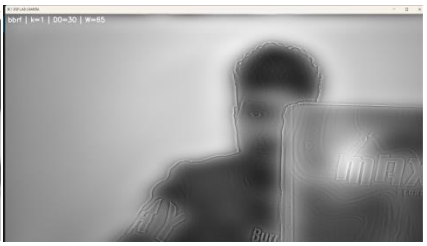
The script is a comprehensive, interactive tool for digital image processing education and experimentation. Its modular design, separating utility functions, filter implementations, and the main loop, makes it highly readable and maintainable. The extensive set of key bindings allows for immediate, real-time visualization of the effects of various spatial and frequency domain filters, making it an excellent resource for understanding image processing concepts.


canny | k=3 | D0=30 | W=10




ibrf | k=1 | D0=30 | W=65


bhpf | k=1 | D0=30 | W=65


lbrf | k=1 | D0=30 | W=65