

PROJECT DESCRIPTION

We have implemented the research paper Weighted Guided Image Filtering with Steering Kernel (attached in the folder) and merged it with 3 other research papers, namely Guided Image Filtering, Kernel Regression and Single Haze Removal for the purpose of application in Image Dehazing and Enhancement using SKWGIF.

We have divided the project into three parts namely:

1. Steering Kernel WGIF Implementation
2. Image Dehazing Using SKWGIF and Histogram Equalization
3. Coloured Image Dehazing Using SKWGIF

• Steering Kernel WGIF Implementation

We have implemented WGIF with a Steering Kernel (SKWGIF). GIF and WGIF suffers from Halo Artifacts. SKWGIF takes advantage of WGIF over GIF while leveraging the edge direction more sufficiently. The code uses Steering Kernel which in turn requires local gradient matrices. The steering kernel helps us to identify the energy dominant directions. We have used different values for the radius of the local window and the regularization parameter. The comparison results clearly show the advantage of using SKWGIF over WGIF and GIF.

Code

```
import cv2
import numpy as np
from math import *

def Grad_func(img):
    """
    This Function returns Local Gradient Matrices.
    Parameters
    -----
    img : Image
        Input Image for which Local Gradient Matrices are to be found.

    Returns
    -----
    Grad_mat : 2D Numpy Matrix
        Matrix containing Local Gradient Matrix for every pixel in input image.
    """
    dummy_mat = np.array([[0.00 for j in range(2)] for i in range(9)])
```

```

Grad_mat = [[dummy_mat for j in range(img.shape[1])] for i in range(img.shape[0])]
for i in range(1, img.shape[0]-1):
    for j in range(1, img.shape[1]-1):
        Gij = []
        roi = img[i-1:i+2, j-1:j+2]
        gx = cv2.Sobel(roi, cv2.CV_64F, 1, 0, ksize = 1)
        gy = cv2.Sobel(roi, cv2.CV_64F, 0, 1, ksize = 1)
        gx = np.reshape(gx, (9, ))
        gy = np.reshape(gy, (9, ))
        Gij.append(gx)
        Gij.append(gy)
        Grad_mat[i][j] = np.transpose(Gij)
return Grad_mat

```

```
def steering_kernel(img):
```

```
'''
```

This Function Returns Steering Kernel Matrices.

Parameters

```
-----
```

img : Image

Returns

```
-----
```

W : 2D Numpy Matrix

Matrix containing Steering Kernel Weight Matrices for each pixel in input image.

```
'''
```

```

Grad_mat = Grad_func(img*255)
dummy_mat = np.array([[0 for j in range(3)] for i in range(3)])
W = [[dummy_mat for j in range(img.shape[1])] for i in range(img.shape[0])]
for i in range(1, img.shape[0]-1):
    for j in range(1, img.shape[1]-1):
        u, s, v = np.linalg.svd(Grad_mat[i][j])
        v2 = v[1]
        if v2[1] == 0:
            theta = pi/2
        else:
            theta = np.arctan(v2[0]/v2[1])
        sigma = (s[0] + 1.0)/(s[1] + 1.0)
        gamma = sqrt(((s[0]*s[1]) + 0.01)/9)
        Rot_mat = np.array([[cos(theta), sin(theta)], [-sin(theta), cos(theta)]])
        El_mat = np.array([[sigma, 0], [0, (1/sigma)]])
        C = gamma*(np.dot(np.dot(Rot_mat, El_mat), np.transpose(Rot_mat)))
        coeff = sqrt(np.linalg.det(C))/(2*pi*(5.76))
        W_i = [[0 for q in range(3)] for p in range(3)]
        for n_i in range(i-1, i+2):
            for n_j in range(j-1, j+2):
                xi = np.array([i, j])
                xk = np.array([n_i, n_j])

```

```

        xik = xi - xk
        wik = coeff*(exp(-(np.dot(np.dot(np.transpose(xik), C), xik))/(11.52))))
        W_i[n_i-i+1][n_j-j+1] = wik
        W[i][j] = W_i
    return W

```

```
def Guided_Image_Filter(im,p,r,eps):
```

```
'''
```

This Function returns the output for
Guided Image Filter applied on Input Image.
Parameters

```
-----
```

im : Guidance Image

p : Input Filter Image

r : Radius of Kernel

eps : Regularization parameter

Returns

```
-----
```

q : Output Image after GIF application

```
'''
```

```

mean_I = cv2.boxFilter(im,cv2.CV_64F,(r,r))
mean_p = cv2.boxFilter(p, cv2.CV_64F,(r,r))
mean_lp = cv2.boxFilter(im*p,cv2.CV_64F,(r,r))
cov_lp = mean_lp - mean_I*mean_p

```

```

mean_II = cv2.boxFilter(im*im,cv2.CV_64F,(r,r))
var_I = mean_II - mean_I*mean_I

```

```

a = cov_lp/(var_I + eps)
b = mean_p - a*mean_I

```

```

mean_a = cv2.boxFilter(a,cv2.CV_64F,(r,r))
mean_b = cv2.boxFilter(b,cv2.CV_64F,(r,r))
cv2.imshow("Edge_GIF", mean_a)
q = mean_a*im + mean_b
return q

```

```
def Weighted_Guided_Image_Filter(im, p, r, r2, eps, lamda, N):
```

```
'''
```

This Function returns the output for Weighted
Guided Image Filter applied on Input Image.
Parameters

```
-----
```

im : Guidance Image

p : Input Filter Image

r : Radius of Kernel

r2 : Radius of Local Window centered at a particular pixel

eps : Regularization parameter

lamda : small constant dependent on dynamic range

N : Number of Pixels in the Input image

Returns

q : Output Image after WGIF application

'''

```
mean_I = cv2.boxFilter(im,cv2.CV_64F,(r,r))
```

```
mean_I2 = cv2.boxFilter(im, cv2.CV_64F,(r2,r2))
```

```
mean_p = cv2.boxFilter(p, cv2.CV_64F,(r,r))
```

```
mean_p2 = cv2.boxFilter(p, cv2.CV_64F, (r2,r2))
```

```
corr_I = cv2.boxFilter(im*im, cv2.CV_64F,(r,r))
```

```
corr_I2 = cv2.boxFilter(im*im,cv2.CV_64F,(r2,r2))
```

```
corr_Ip = cv2.boxFilter(im*p,cv2.CV_64F,(r,r))
```

```
var_I = corr_I - mean_I*mean_I
```

```
var_I2 = corr_I2 - mean_I2*mean_I2
```

```
Psil = ((var_I2+lamda)*np.sum(1/(var_I2 + lamda)))/N
```

```
cov_Ip = corr_Ip - mean_I*mean_p
```

```
a_psi = cov_Ip/(var_I + eps/Psil)
```

```
b_psi = mean_p - (a_psi)*mean_I
```

```
mean_ap = cv2.boxFilter(a_psi,cv2.CV_64F,(r2,r2))
```

```
mean_bp = cv2.boxFilter(b_psi,cv2.CV_64F,(r2,r2))
```

```
cv2.imshow("Edge_WGIF", mean_ap)
```

```
qp = mean_ap*im + mean_bp
```

```
return qp
```

```
def SK_Weighted_Guided_Image_Filter(im,p,r,r2,eps,lamda,N):
```

```
'''
```

This Function returns the output for Steering Kernel
Weighted Guided Image Filter applied on Input Image.

Parameters

im : Guidance Image

p : Input Filter Image

r : Radius of Kernel

r2 : Radius of Local Window centered at a particular pixel

eps : Regularization parameter

lamda : small constant dependent on dynamic range

N : Number of Pixels in the Input image

Returns

q : Output Image after SKWGIF application

'''

```
mean_I = cv2.boxFilter(im,cv2.CV_64F,(r,r))
```

```
mean_I2 = cv2.boxFilter(im, cv2.CV_64F,(r2,r2))
```

```
mean_p = cv2.boxFilter(p, cv2.CV_64F,(r,r))
```

```
mean_p2 = cv2.boxFilter(p, cv2.CV_64F, (r2,r2))
```

```
corr_I = cv2.boxFilter(im*im, cv2.CV_64F,(r,r))
```

```
corr_I2 = cv2.boxFilter(im*im,cv2.CV_64F,(r2,r2))
```

```
corr_Ip = cv2.boxFilter(im*p,cv2.CV_64F,(r,r))
```

```
var_I = corr_I - mean_I*mean_I
```

```
var_I2 = corr_I2 - mean_I2*mean_I2
```

```
Psil = ((var_I2+lamda)*np.sum(1/(var_I2 + lamda)))/N
```

```
cov_Ip = corr_Ip - mean_I*mean_p
```

```
a_psi = cov_Ip/(var_I + eps/Psil)
```

```
b_psi = mean_p - (a_psi)*mean_I
```

```
W = steering_kernel(im)
```

```
mean_a = [[0 for j in range(im.shape[1])] for i in range(im.shape[0])]
```

```
mean_b = [[0 for j in range(im.shape[1])] for i in range(im.shape[0])]
```

```
for i in range(1, im.shape[0]-1):
```

```
    for j in range(1, im.shape[1]-1):
```

```
        Wk = W[i][j]
```

```
        roi_a = a_psi[i-1:i+2, j-1:j+2]
```

```
        roi_b = b_psi[i-1:i+2, j-1:j+2]
```

```
        mean_a[i][j] = np.sum(Wk*roi_a)
```

```
        mean_b[i][j] = np.sum(Wk*roi_b)
```

```
mean_a = np.array(mean_a)
```

```

mean_b = np.array(mean_b)
mean_b = b_psi
cv2.imshow("Edge_SKWGIF", mean_a)
q = mean_a*im + mean_b
return q

```

```

def TransmissionRefine(im, et, r, eps, lamda, N):
'''

```

Parameters

im : Guidance Image

et : Input Filter Image

r : Radius of kernel

eps : Regularization Parameter

lamda : small constant dependent on dynamic range

N : Number of Pixels in the Input image

Returns

None.

'''

```

gray = np.float64(im)/255

```

```

rd = 3

```

```

GIF = Guided_Image_Filter(gray, gray, r, eps)

```

```

WGIF = Weighted_Guided_Image_Filter(gray, gray, r, rd, eps, lamda, N)

```

```

SKWGIF = SK_Weighted_Guided_Image_Filter(gray, gray, r, rd, eps, lamda, N)

```

```

cv2.imshow("GIF", GIF)

```

```

cv2.imshow("WGIF", WGIF)

```

```

cv2.imshow("SKWGIF", SKWGIF)

```

```

src = cv2.imread("cat.png")

```

```

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

```

```

cv2.imshow("gray image", gray)

```

```

minimum = np.min(gray)

```

```

maximum = np.max(gray)

```

```

L = (maximum-minimum)

```

```

lamda = (0.001*L)**2

```

```

rows, columns = gray.shape

```

```

N = rows*columns

```

```

r = [2, 4, 8]

```

```

eps = [0.01, 0.04, 0.16]

```

```

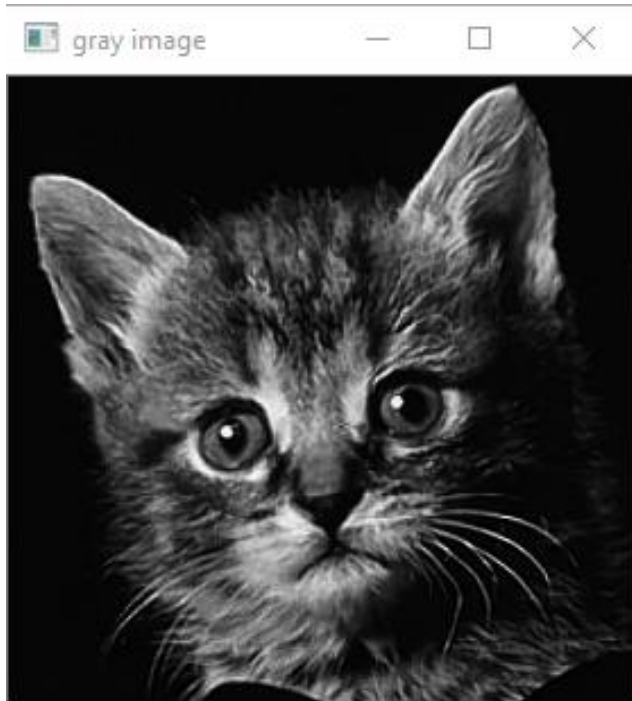
TransmissionRefine(gray, gray, r[0], eps[0], lamda, N)

```

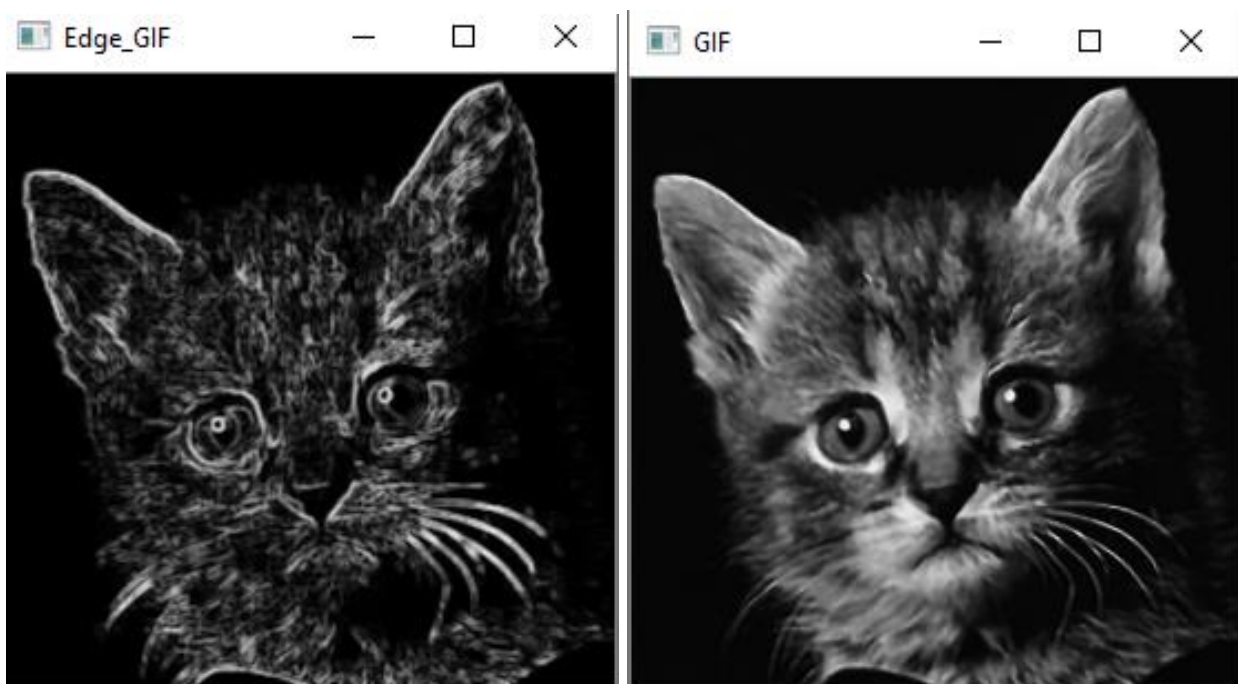
RESULTS AND COMPARISON:

COMPARISON 1:

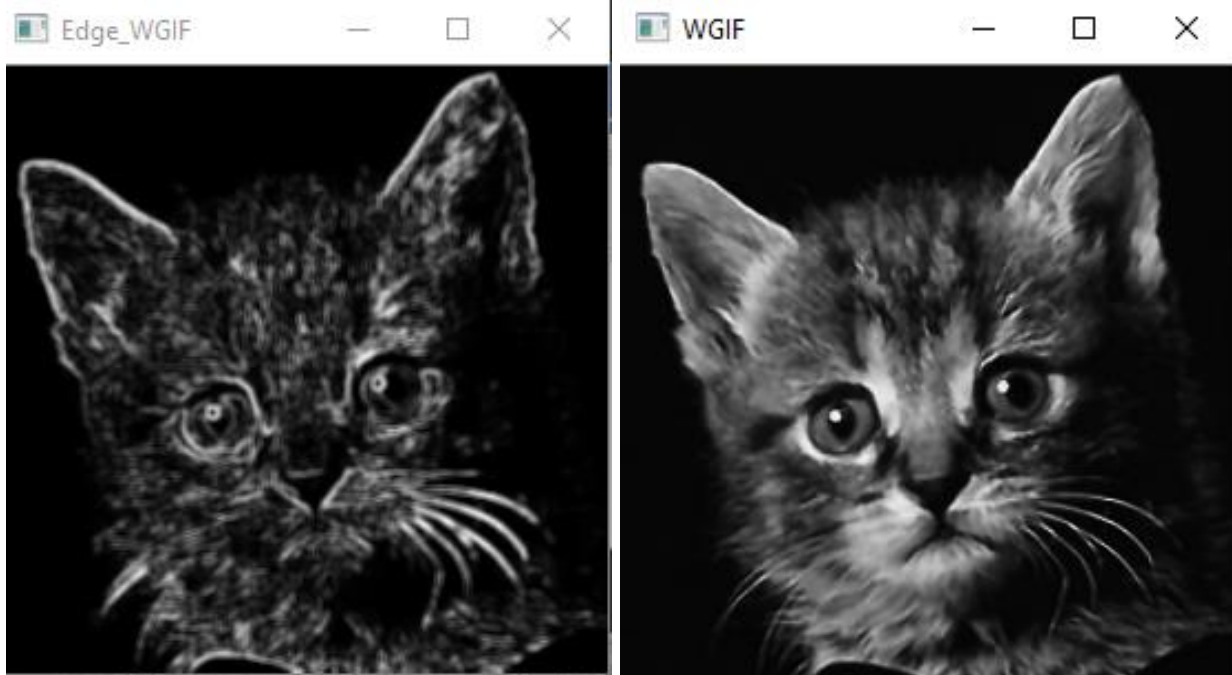
ORIGINAL IMAGE



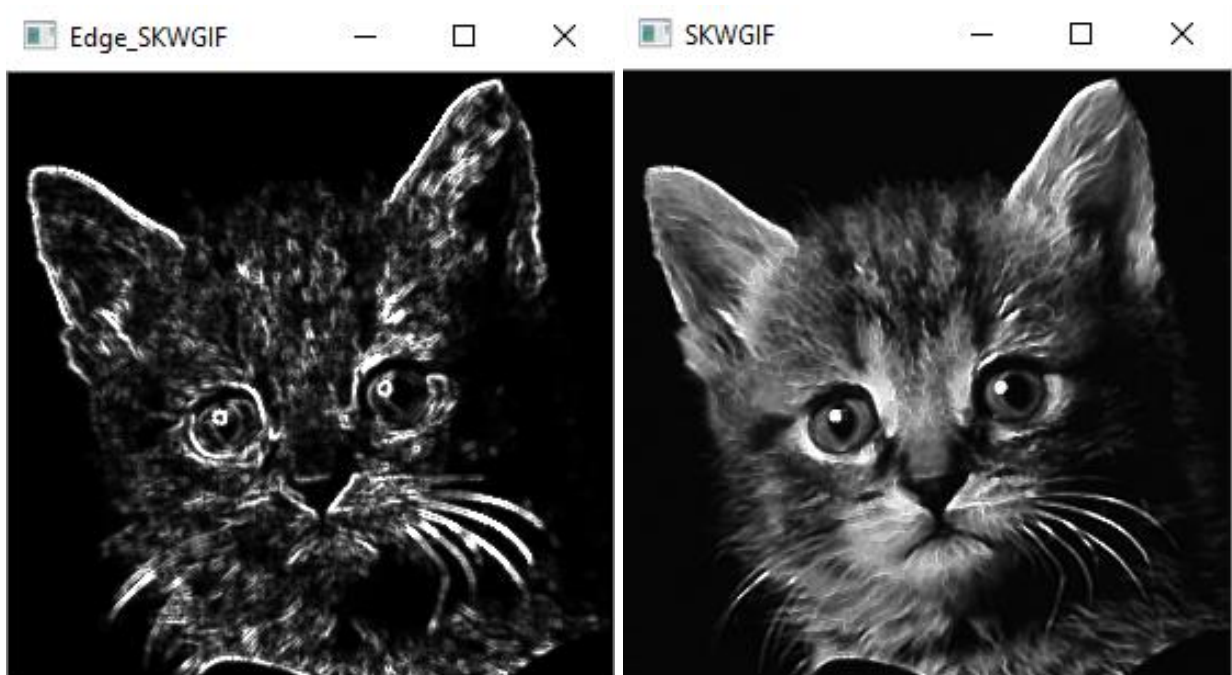
GIF FILTERING OUTPUT



WGIF FILTERING OUTPUT



SKWGIF FILTERING OUTPUT

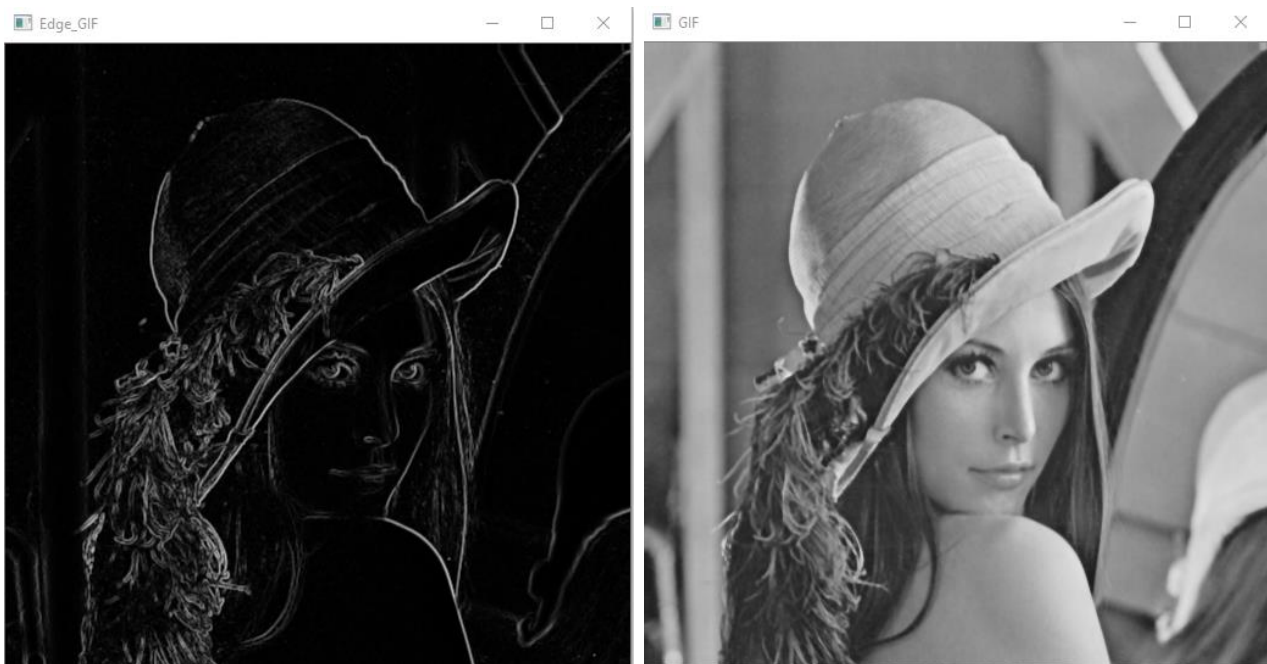


COMPARISON 2:

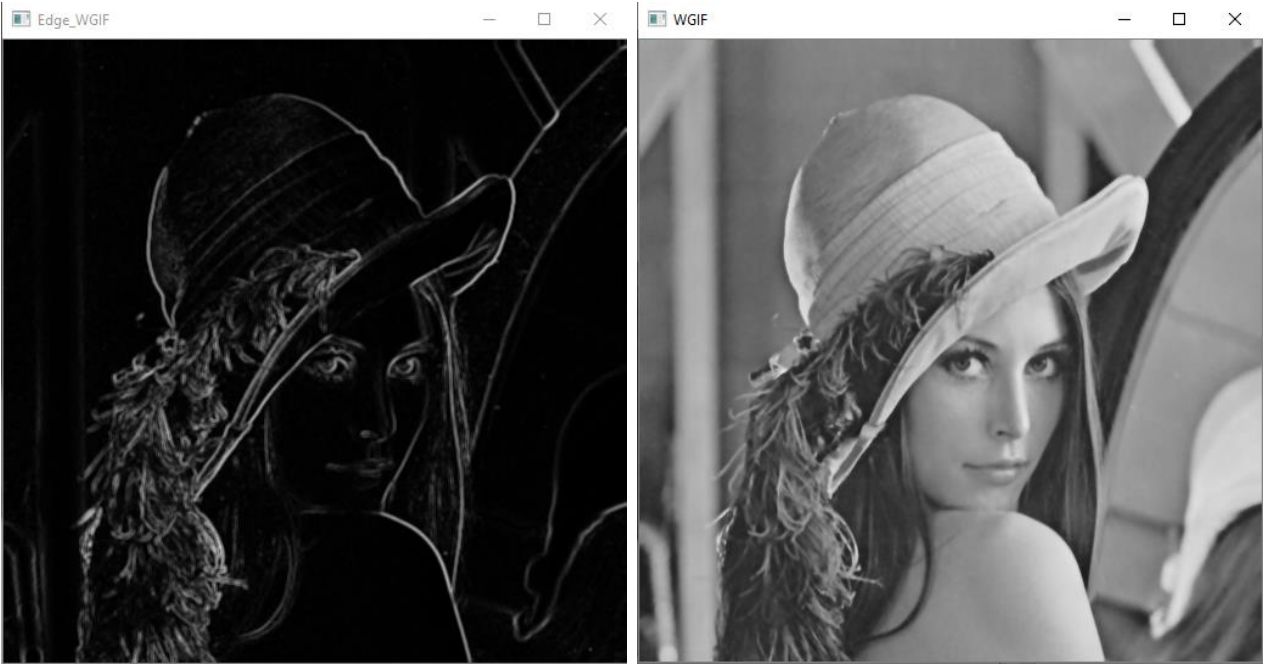
ORIGINAL IMAGE



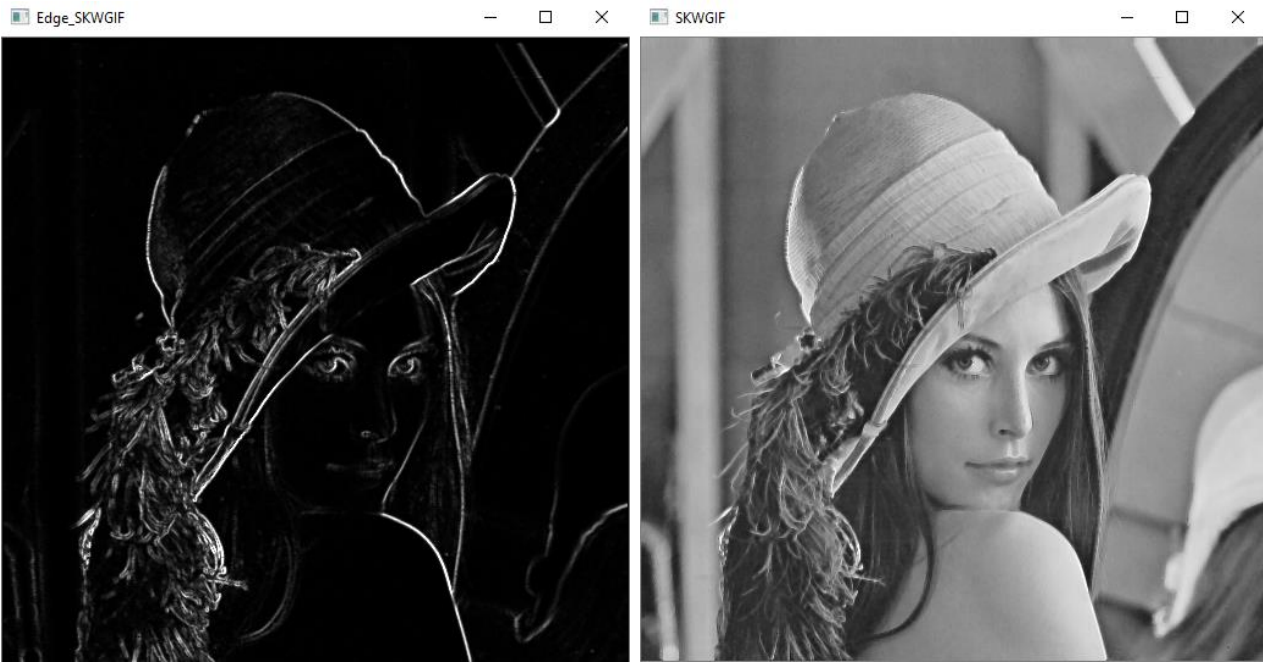
GIF FILTERING OUTPUT



WGIF FILTERING OUTPUT



SKWGIF FILTERING OUTPUT



COMPARISON 3:

ORIGINAL IMAGE



GIF FILTERING OUTPUT



WGIF FILTERING OUTPUT



SKWGIF FILTERING OUTPUT



- **Image Dehazing Using SKWGIF and Histogram Equalization**

We have implemented Image Dehazing for Grayscale Images. First, we applied Histogram Equalization to increase the dynamic range of the image. Then we have used SKWGIF for detail enhancement and edge retention which reduces the contrast of background noise. The Code uses Histogram Equalization function followed by SKWGIF function. The results are presented by varying the values for radius of local window and regularization parameter. SKWGIF gives better results than GIF and WGIF due to supreme edge detection.

Code

```
import cv2
import cv2
import numpy as np
from math import *

#Histogram equalisation
def Histogram_Eq(resized_image):
    """
    This Function returns Histogram Equalised Image for Contrast Enhancement.

    Parameters
    -----
    resized_image : Input Image

    Returns
    -----
    resized_image : Histogram Equalized Image
    """
    r = []
    no_of_pixel = (resized_image.shape[0] * resized_image.shape[1])
    r = np.ravel(resized_image)
    unique, count = np.unique(r, return_counts=True)
    #print(unique)
    counts = np.zeros((256, ))
    val = 0
    for i in range(256):
        if i in unique:
            counts[i] = count[val]
            val += 1;

    sigma_list = []
    sigma_list.append(counts[0])
    for i in range(1, 256):
        cnt = counts[i]
```

```
sigma_list.append(sigma_list[i - 1] + cnt)
```

```
sigma_list = np.array(sigma_list)
```

```
final_list = (sigma_list*255)/(no_of_pixel)#[(x*255)/(no_of_pixel) for x in sigma_list]
```

```
for i in range(resized_image.shape[0]):
```

```
    for j in range(resized_image.shape[1]):
```

```
        temp = resized_image[i][j]
```

```
        resized_image[i][j] = final_list[temp]
```

```
return resized_image
```

```
def Grad_func(img):
```

```
    """
```

```
    This Function returns Local Gradient Matrices.
```

```
    Parameters
```

```
    -----
```

```
    img : Image
```

```
        Input Image for which Local Gradient Matrices are to be found.
```

```
    Returns
```

```
    -----
```

```
    Grad_mat : 2D Numpy Matrix
```

```
        Matrix containing Local Gradient Matrix for every pixel in input image.
```

```
    """
```

```
    dummy_mat = np.array([[0.00 for j in range(2)] for i in range(9)])
```

```
    Grad_mat = [[dummy_mat for j in range(img.shape[1])] for i in range(img.shape[0])]
```

```
    for i in range(1, img.shape[0]-1):
```

```
        for j in range(1, img.shape[1]-1):
```

```
            Gij = []
```

```
            roi = img[i-1:i+2, j-1:j+2]
```

```
            gx = cv2.Sobel(roi,cv2.CV_64F, 1, 0, ksize = 1)
```

```
            gy = cv2.Sobel(roi, cv2.CV_64F, 0, 1, ksize = 1)
```

```
            gx = np.reshape(gx,(9, ))
```

```
            gy = np.reshape(gy,(9, ))
```

```
            Gij.append(gx)
```

```
            Gij.append(gy)
```

```
            Grad_mat[i][j] = np.transpose(Gij)
```

```
    return Grad_mat
```

```
def steering_kernel(img):
```

```
    """
```

```
    This Function Returns Steering Kernel Matrices.
```

```
    Parameters
```

```
    -----
```

```
    img : Image
```

```
    Returns
```

```
    -----
```

```
    W : 2D Numpy Matrix
```

Matrix containing Steering Kernel Weight Matrices for each pixel in input image.

'''

```
Grad_mat = Grad_func(img*255)
dummy_mat = np.array([[0 for j in range(3)] for i in range(3)])
W = [[dummy_mat for j in range(img.shape[1])] for i in range(img.shape[0])]
for i in range(1, img.shape[0]-1):
    for j in range(1, img.shape[1]-1):
        u, s, v = np.linalg.svd(Grad_mat[i][j])
        v2 = v[1]
        if v2[1] == 0:
            theta = pi/2
        else:
            theta = np.arctan(v2[0]/v2[1])
        sigma = (s[0] + 1.0)/(s[1] + 1.0)
        gamma = sqrt(((s[0]*s[1]) + 0.01)/9)
        Rot_mat = np.array([[cos(theta), sin(theta)], [-sin(theta), cos(theta)]])
        El_mat = np.array([[sigma, 0], [0, (1/sigma)]])
        C = gamma*(np.dot(np.dot(Rot_mat, El_mat), np.transpose(Rot_mat)))
        coeff = sqrt(np.linalg.det(C))/(2*pi*(5.76))
        W_i = [[0 for q in range(3)] for p in range(3)]
        for n_i in range(i-1, i+2):
            for n_j in range(j-1, j+2):
                xi = np.array([i, j])
                xk = np.array([n_i, n_j])
                xik = xi - xk
                wik = coeff*(exp(-(np.dot(np.dot(np.transpose(xik), C), xik))/(11.52))))
                W_i[n_i-i+1][n_j-j+1] = wik
        W[i][j] = W_i
return W
```

```
def Guided_Image_Filter(im,p,r,eps):
```

'''

This Function returns the output for
Guided Image Filter applied on Input Image.

Parameters

im : Guidance Image

p : Input Filter Image

r : Radius of Kernel

eps : Regularization parameter

Returns

q : Output Image after GIF application

```

'''
mean_I = cv2.boxFilter(im,cv2.CV_64F,(r,r))
mean_p = cv2.boxFilter(p, cv2.CV_64F,(r,r))
mean_Ip = cv2.boxFilter(im*p,cv2.CV_64F,(r,r))
cov_Ip = mean_Ip - mean_I*mean_p

mean_II = cv2.boxFilter(im*im,cv2.CV_64F,(r,r))
var_I = mean_II - mean_I*mean_I

a = cov_Ip/(var_I + eps)
b = mean_p - a*mean_I

mean_a = cv2.boxFilter(a,cv2.CV_64F,(r,r))
mean_b = cv2.boxFilter(b,cv2.CV_64F,(r,r))
cv2.imshow("Edge_GIF", mean_a)
q = mean_a*im + mean_b
return q

def Weighted_Guided_Image_Filter(im, p, r, r2, eps, lamda, N):
'''
    This Function returns the output for Weighted
    Guided Image Filter applied on Input Image.

    Parameters
    -----
    im : Guidance Image

    p : Input Filter Image

    r : Radius of Kernel

    r2 : Radius of Local Window centered at a particular pixel

    eps : Regularization parameter

    lamda : small constant dependent on dynamic range

    N : Number of Pixels in the Input image

    Returns
    -----
    q : Output Image after WGIF application
'''
mean_I = cv2.boxFilter(im,cv2.CV_64F,(r,r))
mean_I2 = cv2.boxFilter(im, cv2.CV_64F,(r2,r2))
mean_p = cv2.boxFilter(p, cv2.CV_64F,(r,r))
mean_p2 = cv2.boxFilter(p, cv2.CV_64F, (r2,r2))

```

```

corr_I = cv2.boxFilter(im*im, cv2.CV_64F,(r,r))
corr_I2 = cv2.boxFilter(im*im,cv2.CV_64F,(r2,r2))
corr_Ip = cv2.boxFilter(im*p,cv2.CV_64F,(r,r))

var_I = corr_I - mean_I*mean_I
var_I2 = corr_I2 - mean_I2*mean_I2

Psil = ((var_I2+lamda)*np.sum(1/(var_I2 + lamda)))/N

cov_Ip = corr_Ip - mean_I*mean_p

a_psi = cov_Ip/(var_I + eps/Psil)
b_psi = mean_p - (a_psi)*mean_I
mean_ap = cv2.boxFilter(a_psi,cv2.CV_64F,(r2,r2))
mean_bp = cv2.boxFilter(b_psi,cv2.CV_64F,(r2,r2))
cv2.imshow("Edge_WGIF", mean_ap)
qp = mean_ap*im + mean_bp
return qp

```

```

def SK_Weighted_Guided_Image_Filter(im,p,r,r2,eps,lamda,N):
'''

```

This Function returns the output for Steering Kernel
Weighted Guided Image Filter applied on Input Image.

Parameters

im : Guidance Image

p : Input Filter Image

r : Radius of Kernel

r2 : Radius of Local Window centered at a particular pixel

eps : Regularization parameter

lamda : small constant dependent on dynamic range

N : Number of Pixels in the Input image

Returns

q : Output Image after SKWGIF application

'''

```

mean_I = cv2.boxFilter(im,cv2.CV_64F,(r,r))
mean_I2 = cv2.boxFilter(im, cv2.CV_64F,(r2,r2))
mean_p = cv2.boxFilter(p, cv2.CV_64F,(r,r))

```



```
mean_p2 = cv2.boxFilter(p, cv2.CV_64F, (r2,r2))
```

```
corr_l = cv2.boxFilter(im*im, cv2.CV_64F,(r,r))
```

```
corr_l2 = cv2.boxFilter(im*im,cv2.CV_64F,(r2,r2))
```

```
corr_lp = cv2.boxFilter(im*p,cv2.CV_64F,(r,r))
```

```
var_l = corr_l - mean_l*mean_l
```

```
var_l2 = corr_l2 - mean_l2*mean_l2
```

```
Psil = ((var_l2+lamda)*np.sum(1/(var_l2 + lamda)))/N
```

```
cov_lp = corr_lp - mean_l*mean_p
```

```
a_psi = cov_lp/(var_l + eps/Psil)
```

```
b_psi = mean_p - (a_psi)*mean_l
```

```
W = steering_kernel(im)
```

```
mean_a = [[0 for j in range(im.shape[1])] for i in range(im.shape[0])]
```

```
mean_b = [[0 for j in range(im.shape[1])] for i in range(im.shape[0])]
```

```
for i in range(1, im.shape[0]-1):
```

```
    for j in range(1, im.shape[1]-1):
```

```
        Wk = W[i][j]
```

```
        roi_a = a_psi[i-1:i+2, j-1:j+2]
```

```
        roi_b = b_psi[i-1:i+2, j-1:j+2]
```

```
        mean_a[i][j] = np.sum(Wk*roi_a)
```

```
        mean_b[i][j] = np.sum(Wk*roi_b)
```

```
mean_a = np.array(mean_a)
```

```
mean_b = np.array(mean_b)
```

```
mean_b = b_psi
```

```
cv2.imshow("Edge_SKWGIF", mean_a)
```

```
q = mean_a*im + mean_b
```

```
return q
```

```
def TransmissionRefine(im, et, r, eps, lamda, N):
```

```
    '''
```

```
    Parameters
```

```
    -----
```

```
    im : Guidance Image
```

```
    et : Input Filter Image
```

```
    r : Radius of kernel
```

```
    eps : Regularization Parameter
```

```
    lamda : small constant dependent on dynamic range
```

N : Number of Pixels in the Input image

Returns

None.

'''

```
gray = np.float64(im)/255
```

```
rd = 3
```

```
GIF = Guided_Image_Filter(gray, gray, r, eps)
```

```
WGIF = Weighted_Guided_Image_Filter(gray, gray, r, rd, eps, lamda, N)
```

```
SKWGIF = SK_Weighted_Guided_Image_Filter(gray, gray, r, rd, eps, lamda, N)
```

```
cv2.imshow("GIF", GIF)
```

```
cv2.imshow("WGIF", WGIF)
```

```
cv2.imshow("SKWGIF", SKWGIF)
```

```
original_image = cv2.imread('Input Images Used\Building with Haze.png', cv2.IMREAD_GRAYSCALE)
```

```
cv2.imshow("original_image", original_image)
```

```
histogram_equalised_image = Histogram_Eq(original_image)
```

```
gray = histogram_equalised_image
```

```
minimum = np.min(gray)
```

```
maximum = np.max(gray)
```

```
L = (maximum-minimum)
```

```
lamda = (0.001*L)**2
```

```
rows, columns = gray.shape
```

```
N = rows*columns
```

```
r = 2
```

```
eps = 0.01
```

```
TransmissionRefine(gray, gray, r, eps, lamda, N)
```

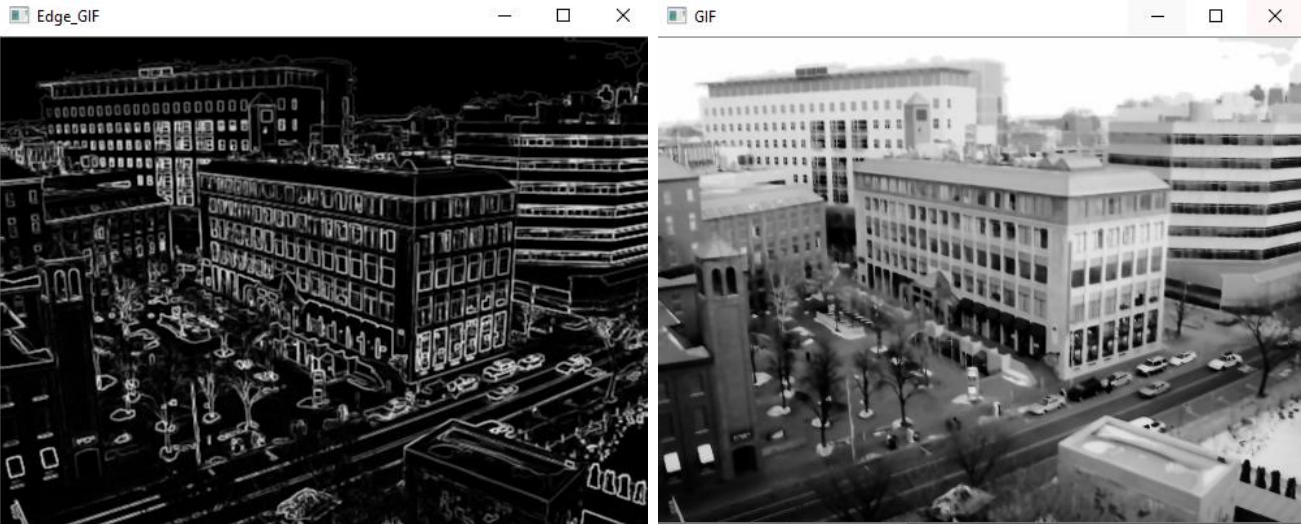
RESULT AND ANALYSIS

COMPARISON 1:

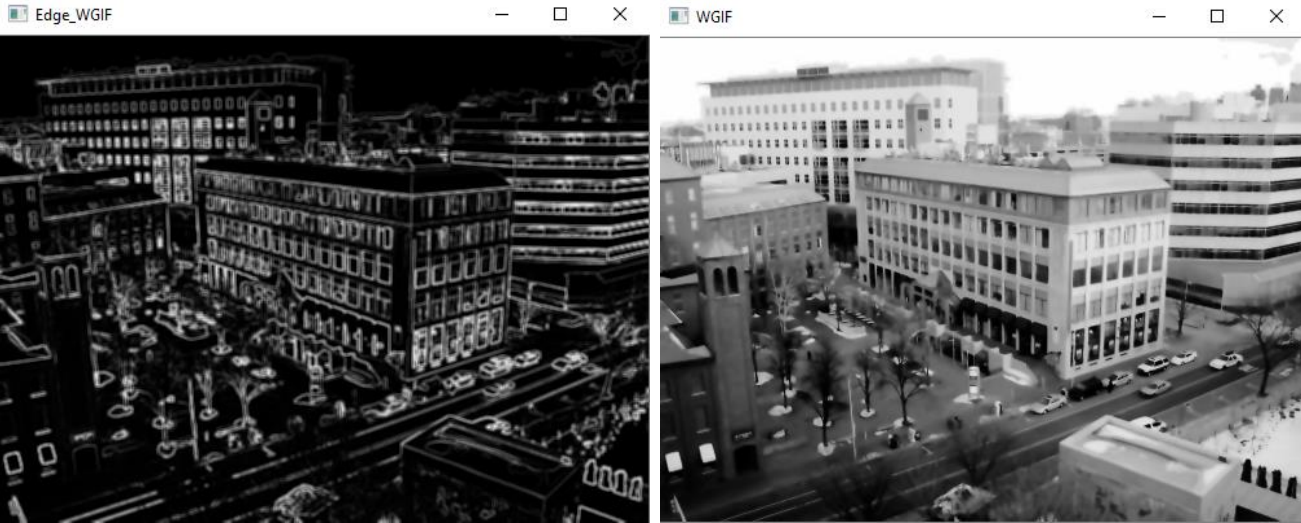
ORIGINAL IMAGE



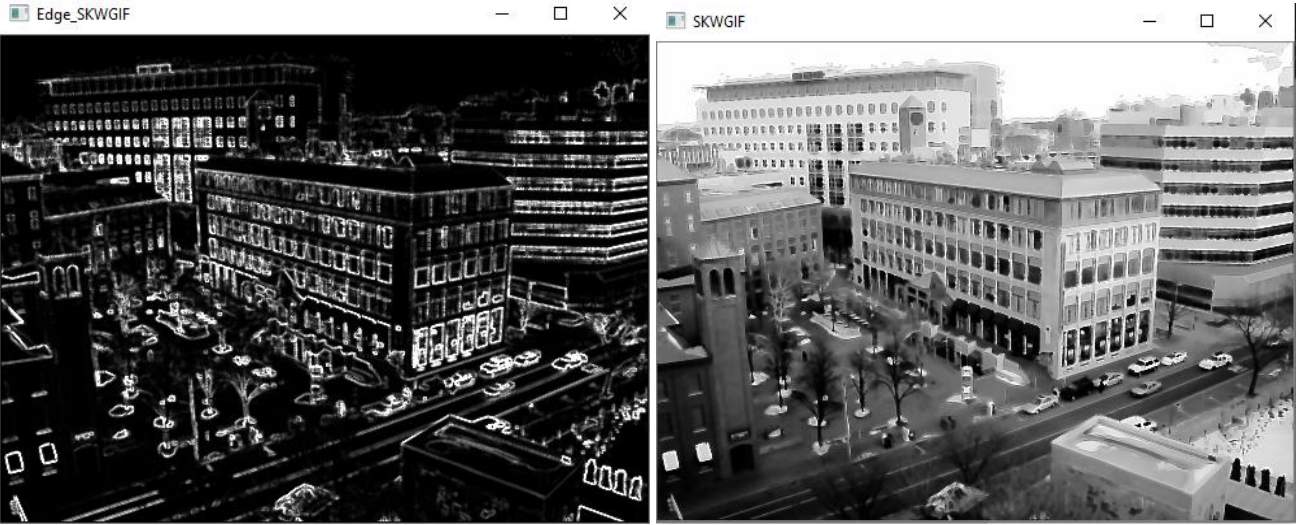
GIF FILTERING OUTPUT



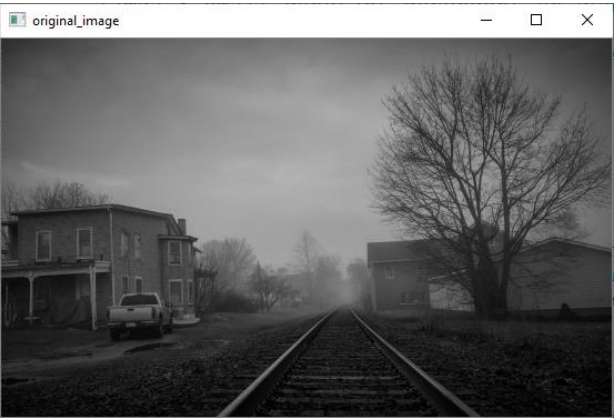
WGIF FILTERING OUTPUT



SKWGIF FILTERING OUTPUT



COMPARISON 2: ORIGINAL IMAGE



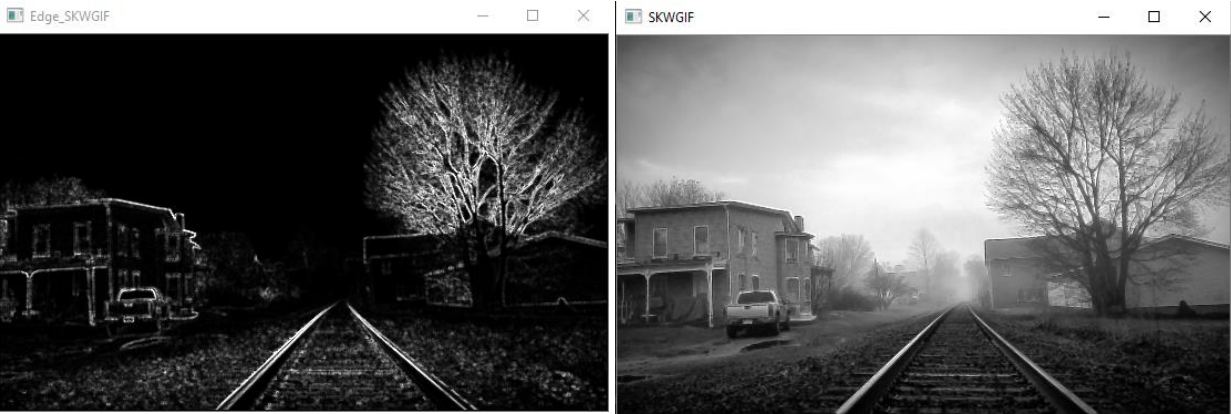
GIF FILTERING OUTPUT



WGIF FILTERING OUTPUT



SKWGIF FILTERING OUTPUT



● Coloured Image Dehazing Using SKWGIF

We have implemented image dehazing for coloured images using SKWGIF and Dark Channel Prior. Instead of using Laplacian matrix to refine the haze transmission map, we simply use SKWGIF to filter the raw transmission map under the guidance of the hazy image. The code uses Dark Channel Function, AtmLight Function, Transmission Refine and Recovery function along with SKWGIF function for guided image filtering. The results come out with a large amount of haze removed and visually similar to the original method while being a simpler algorithm and with less overhead.

Code

```
import cv2
from math import *
import numpy as np;

def DarkChannel(im,sz):
    """ This Function is used to get the pixel values
    whose intensity is very low in the given colour channel """
    b,g,r = cv2.split(im)
    dc = cv2.min(cv2.min(r,g),b);
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT,(sz,sz))
    dark = cv2.erode(dc,kernel)
    return dark

def AtmLight(im,dark):
    """ This Function is used to estimate
    the atmospheric light in image"""
    [h,w] = im.shape[:2]
    imsz = h*w
    numpx = int(max(floor(imsz/1000),1))
    darkvec = dark.reshape(imsz,1);
    imvec = im.reshape(imsz,3);

    indices = darkvec.argsort();
    indices = indices[imsz-numpx::]

    atmsum = np.zeros([1,3])
    for ind in range(1,numpx):
        atmsum = atmsum + imvec[indices[ind]]

    A = atmsum / numpx;
    return A
```



```

def TransmissionEstimate(im,A,sz):
    """ This Function deletes the preliminary transmittal
    based on latent area obtained from dark channel"""
    omega = 0.95;
    im3 = np.empty(im.shape,im.dtype);

    for ind in range(0,3):
        im3[:, :,ind] = im[:, :,ind]/A[0,ind]

    transmission = 1 - omega*DarkChannel(im3,sz);
    return transmission

def Grad_func(img):
    """
    This Function returns Local Gradient Matrices.
    Parameters
    -----
    img : Image
        Input Image for which Local Gradient Matrices are to be found.

    Returns
    -----
    Grad_mat : 2D Numpy Matrix
        Matrix containing Local Gradient Matrix for every pixel in input image.
    """
    dummy_mat = np.array([[0.00 for j in range(2)] for i in range(9)])
    Grad_mat = [[dummy_mat for j in range(img.shape[1])] for i in range(img.shape[0])]
    for i in range(1, img.shape[0]-1):
        for j in range(1, img.shape[1]-1):
            Gij = []
            roi = img[i-1:i+2, j-1:j+2]
            gx = cv2.Sobel(roi,cv2.CV_64F, 1, 0, ksize = 1)
            gy = cv2.Sobel(roi, cv2.CV_64F, 0, 1, ksize = 1)
            gx = np.reshape(gx,(9, ))
            gy = np.reshape(gy,(9, ))
            Gij.append(gx)
            Gij.append(gy)
            Grad_mat[i][j] = np.transpose(Gij)
    return Grad_mat

def steering_kernel(img):
    """
    This Function Returns Steering Kernel Matrices.
    Parameters
    -----
    img : Image

    Returns
    """

```

W : 2D Numpy Matrix

Matrix containing Steering Kernel Weight Matrices for each pixel in input image.

'''

```
Grad_mat = Grad_func(img*255)
dummy_mat = np.array([[0 for j in range(3)] for i in range(3)])
W = [[dummy_mat for j in range(img.shape[1])] for i in range(img.shape[0])]
for i in range(1, img.shape[0]-1):
    for j in range(1, img.shape[1]-1):
        u, s, v = np.linalg.svd(Grad_mat[i][j])
        v2 = v[1]
        if v2[1] == 0:
            theta = pi/2
        else:
            theta = np.arctan(v2[0]/v2[1])
        sigma = (s[0] + 1.0)/(s[1] + 1.0)
        gamma = sqrt(((s[0]*s[1]) + 0.01)/9)
        Rot_mat = np.array([[cos(theta), sin(theta)], [-sin(theta), cos(theta)]])
        El_mat = np.array([[sigma, 0], [0, (1/sigma)]])
        C = gamma*(np.dot(np.dot(Rot_mat, El_mat), np.transpose(Rot_mat)))
        coeff = sqrt(np.linalg.det(C))/(2*pi*(5.76))
        W_i = [[0 for q in range(3)] for p in range(3)]
        for n_i in range(i-1, i+2):
            for n_j in range(j-1, j+2):
                xi = np.array([i, j])
                xk = np.array([n_i, n_j])
                xik = xi - xk
                wik = coeff*(exp(-(np.dot(np.dot(np.transpose(xik), C), xik))/(11.52))))
                W_i[n_i-i+1][n_j-j+1] = wik
        W[i][j] = W_i
return W
```

def SK_Weighted_Guided_Image_Filter(im,p,r,r2,eps,lamda,N):

'''

This Function returns the output for Steering Kernel
Weighted Guided Image Filter applied on Input Image.

Parameters

im : Guidance Image

p : Input Filter Image

r : Radius of Kernel

r2 : Radius of Local Window centered at a particular pixel

eps : Regularization parameter

lamda : small constant dependent on dynamic range

N : Number of Pixels in the Input image

Returns

q : Output Image after SKWGIF application

'''

```
mean_I = cv2.boxFilter(im,cv2.CV_64F,(r,r))
```

```
mean_I2 = cv2.boxFilter(im, cv2.CV_64F,(r2,r2))
```

```
mean_p = cv2.boxFilter(p, cv2.CV_64F,(r,r))
```

```
mean_p2 = cv2.boxFilter(p, cv2.CV_64F, (r2,r2))
```

```
corr_I = cv2.boxFilter(im*im, cv2.CV_64F,(r,r))
```

```
corr_I2 = cv2.boxFilter(im*im,cv2.CV_64F,(r2,r2))
```

```
corr_Ip = cv2.boxFilter(im*p,cv2.CV_64F,(r,r))
```

```
var_I = corr_I - mean_I*mean_I
```

```
var_I2 = corr_I2 - mean_I2*mean_I2
```

```
Psil = ((var_I2+lamda)*np.sum(1/(var_I2 + lamda)))/N
```

```
cov_Ip = corr_Ip - mean_I*mean_p
```

```
a_psi = cov_Ip/(var_I + eps/Psil)
```

```
b_psi = mean_p - (a_psi)*mean_I
```

```
W = steering_kernel(im)
```

```
mean_a = [[0 for j in range(im.shape[1])] for i in range(im.shape[0])]
```

```
mean_b = [[0 for j in range(im.shape[1])] for i in range(im.shape[0])]
```

```
for i in range(1, im.shape[0]-1):
```

```
    for j in range(1, im.shape[1]-1):
```

```
        Wk = W[i][j]
```

```
        roi_a = a_psi[i-1:i+2, j-1:j+2]
```

```
        roi_b = b_psi[i-1:i+2, j-1:j+2]
```

```
        mean_a[i][j] = np.sum(Wk*roi_a)
```

```
        mean_b[i][j] = np.sum(Wk*roi_b)
```

```
mean_a = np.array(mean_a)
```

```
mean_b = np.array(mean_b)
```

```
mean_b = b_psi
```

```
#cv2.imshow("Edge_SKWGIF", mean_a)
```

```
q = mean_a*im + mean_b
```

```
return q
```

```
def TransmissionRefine(im,et):
```

```
'''
```

Parameters

im : Guidance Image

et : Input Filter Image

Returns

t : Final image

'''

```
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY);
gray = np.float64(gray)/255;
r = 32;
eps = 0.01;
minimum = np.min(gray)
maximum = np.max(gray)
L = (maximum-minimum)
lamda = (0.001*L)**2
rows, columns = gray.shape
N = rows*columns
t = SK_Weighted_Guided_Image_Filter(gray, et, r, 3, eps, lamda, N);

return t;
```

def Recover(im,t,A,tx = 0.1):

""" This function recovers a haze-free image determined by three surface colour values as well as transmission value at every pixel."""

```
res = np.empty(im.shape,im.dtype);
t = cv2.max(t,tx);
```

for ind in range(0,3):

```
    res[:, :, ind] = (im[:, :, ind]-A[0, ind])/t + A[0, ind]
```

return res

fn = "Input Images Used\Building with Haze Color.jpg"

src = cv2.imread(fn);

I = src.astype('float64')/255;

dark = DarkChannel(I,15);

A = AtmLight(I,dark);

te = TransmissionEstimate(I,A,15);

t = TransmissionRefine(src,te);

J = Recover(I,t,A,0.1);

cv2.imshow("dark",dark);

cv2.imshow("Transmission Image",t);

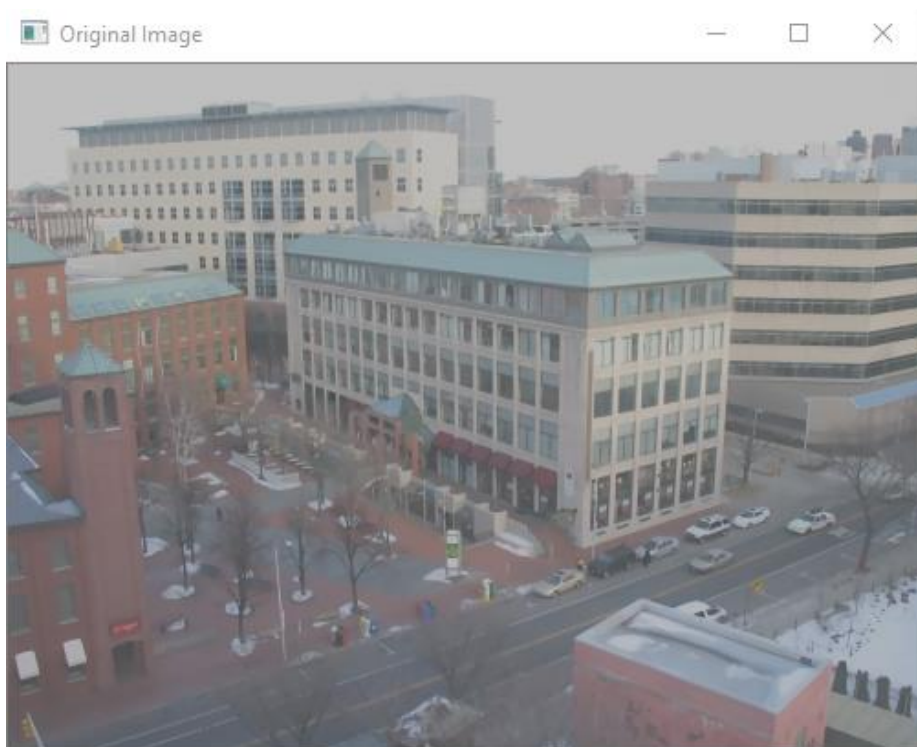
cv2.imshow('Original Image',src);

cv2.imshow('Final Image',J);

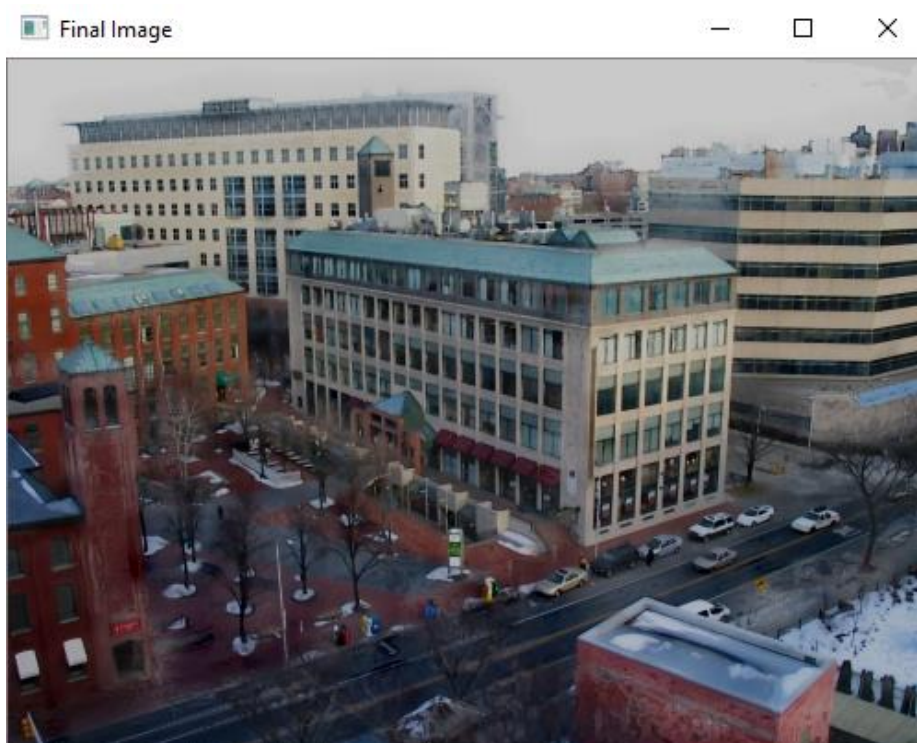
RESULT AND ANALYSIS

RESULT 1:

ORIGINAL IMAGE

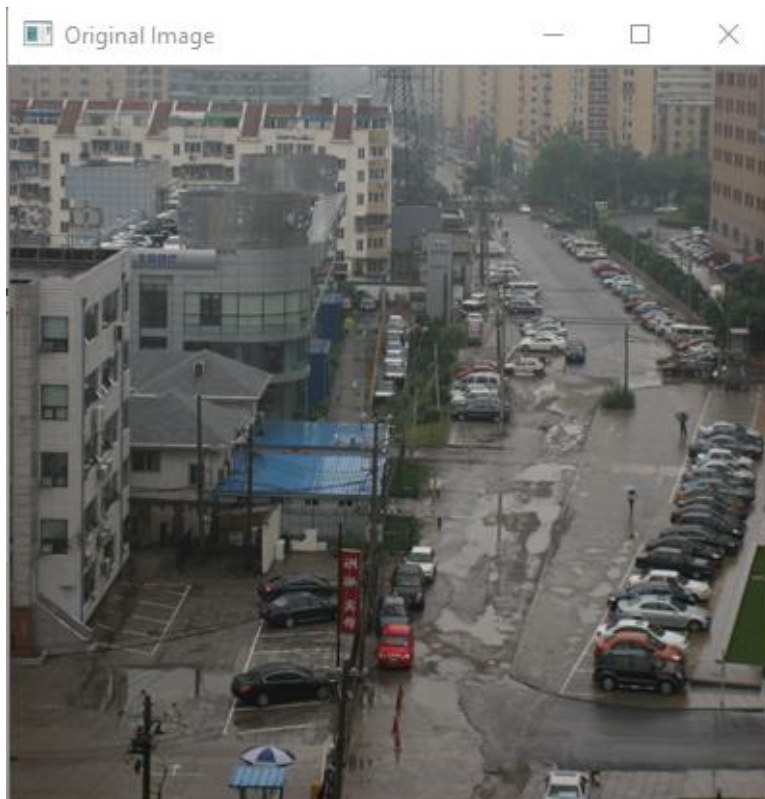


FINAL IMAGE AFTER HAZE REMOVAL



RESULT 2:

ORIGINAL IMAGE



FINAL IMAGE AFTER HAZE REMOVAL

