# Computer Vision (CSE 6239)

Submited To

Dr. Sk. Mohammad Masudul Ahsan
(Professor)
(Department of Computer Science and Engineering)
(Khulna University of Engineering and Technology (KUET) Khulna 9203 Bangladesh)

Submited By

Md. Shahidul Islam
(Student ID: 1907509)

**Computer Science and Engineering Discipline**
**Khulna University of Engineering and Technology**
Khulna-9208, Bangladesh

# Chapter 1

# Answer to the question No 1

All program can be run online in colab:

https://colab.research.google.com/drive/12ZYc1OvALNBnE8OlXQIfTgfxKPxk31jx

## 1.1 Averaging Kernel (3x3 and 5x5 )

Average (or mean) filtering is a method of 'smoothing' images by reducing the amount of intensity variation between neighbouring pixels. The average filter works by moving through the image pixel by pixel, replacing each value with the average value of neighbouring pixels, including itself. There are some potential problems:

- A single pixel with a very unrepresentative value can significantly affect the average value of all the pixels in its neighbourhood.
- When the filter neighbourhood straddles an edge, the filter will interpolate new values for pixels on the edge and so will blur that edge. This may be a problem if sharp edges are required in the output.

In bellow there;s given the python implementation of Averaging Kernel (3x3 and 5x5 )

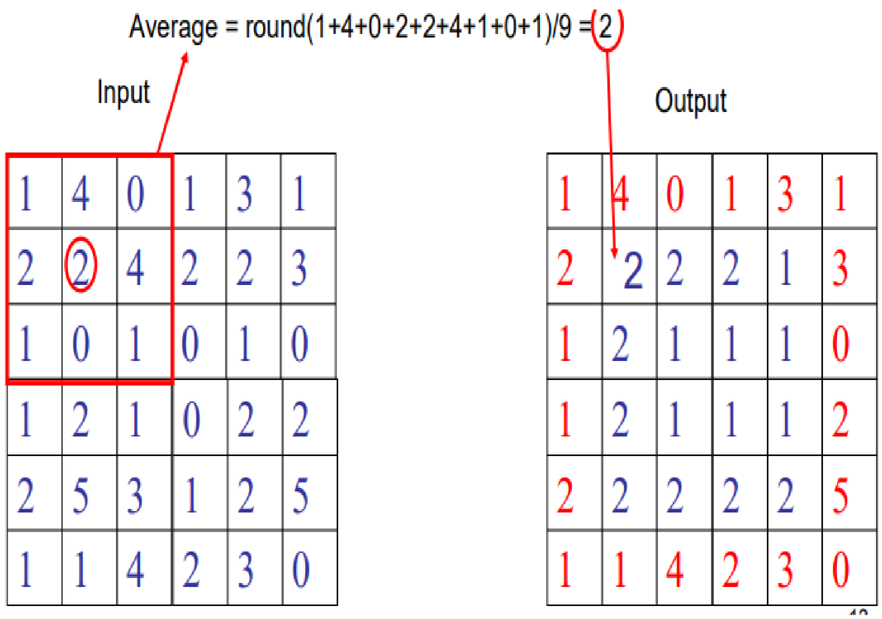


Figure 1.1: average filter

```
#============================ average.py ======================
import numpy as np
import cv2
import sys
import getopt
import matplotlib.pyplot as plt



def readImage(filename):
"""
Read in an image file, errors out if we can't find the file
:param filename: The image filename.
:return: The img object in matrix form.
"""
```

```
img = cv2.imread(filename, 0)
if img is None:
print('Invalid image:' + filename)
return None
else:
print('Image successfully read...')
return img


def integralImage(img):
"""

:param img:
:return:
"""
height = img.shape[0]
width = img.shape[1]
int_image = np.zeros((height, width), np.uint64)
for y in range(height):
for x in range(width):
up = 0 if (y-1 < 0) else int_image.item((y-1, x))
left = 0 if (x-1 < 0) else int_image.item((y, x-1))
diagonal = 0 if (
x-1 < 0 or y-1 < 0) else int_image.item((y-1, x-1))
val = img.item((y, x)) + int(up) + int(left) - int(diagonal)
int_image.itemset((y, x), val)
```

```
return int_image



def adjustEdges(height, width, point):
"""
This handles the edge cases if the box's bounds are outside the image range b
:param height: Height of the image.
:param width: Width of the image.
:param point: The current point.
:return:
"""
newPoint = [point[0], point[1]]
if point[0] >= height:
newPoint[0] = height - 1

if point[1] >= width:
newPoint[1] = width - 1
return tuple(newPoint)



def findArea(int_img, a, b, c, d):
"""
Finds the area for a particular square using the integral image. See summed a
:param int_img: The
:param a: Top left corner.
:param b: Top right corner.
```

```
:param c: Bottom left corner.
:param d: Bottom right corner.
:return: The integral image.
"""
height = int_img.shape[0]
width = int_img.shape[1]
a = adjustEdges(height, width, a)
b = adjustEdges(height, width, b)
c = adjustEdges(height, width, c)
d = adjustEdges(height, width, d)

a = 0 if (a[0] < 0 or a[0] >= height) or (
a[1] < 0 or a[1] >= width) else int_img.item(a[0], a[1])
b = 0 if (b[0] < 0 or b[0] >= height) or (
b[1] < 0 or b[1] >= width) else int_img.item(b[0], b[1])
c = 0 if (c[0] < 0 or c[0] >= height) or (
c[1] < 0 or c[1] >= width) else int_img.item(c[0], c[1])
d = 0 if (d[0] < 0 or d[0] >= height) or (
d[1] < 0 or d[1] >= width) else int_img.item(d[0], d[1])

return a + d - b - c



def boxFilter(img, filterSize):
"""
Runs the subsequent box filtering steps. Prints original image, finds integra
```

```
:param img: An image in matrix form.
:param filterSize: The filter size of the matrix
:return: A final image written as finalimage.png
"""
print("Printing original image...")
print(img)
height = img.shape[0]
width = img.shape[1]
intImg = integralImage(img)
finalImg = np.ones((height, width), np.uint64)
print("Printing integral image...")
print(intImg)
cv2.imwrite("integral_image.png", intImg)
loc = filterSize//2
for y in range(height):
for x in range(width):
finalImg.itemset((y, x), findArea(intImg, (y-loc-1, x-loc-1),
(y-loc-1, x+loc), (y+loc, x-loc-1), (y+loc, x+loc))//(filterSize**2))
print("Printing final image...")
print(finalImg)
plt.imshow(finalImg,cmap='gray')

cv2.imwrite("finalimage.png", finalImg)



def main():
```

```
img = readImage("House1.jpg")
boxFilter(img, 3)



if __name__ == "__main__":
main()
#================================ END of average.py ================
```

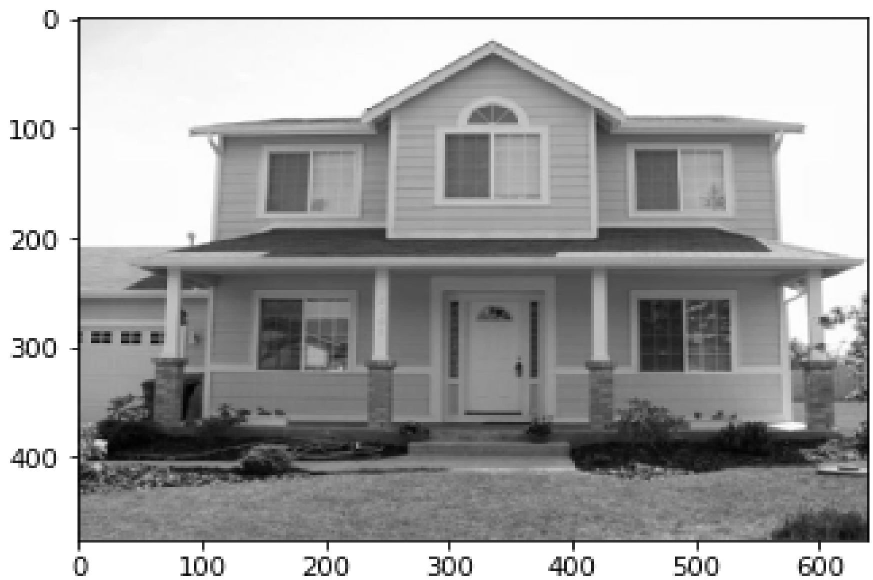



Figure 1.2: Averaging Kernel(3x3)

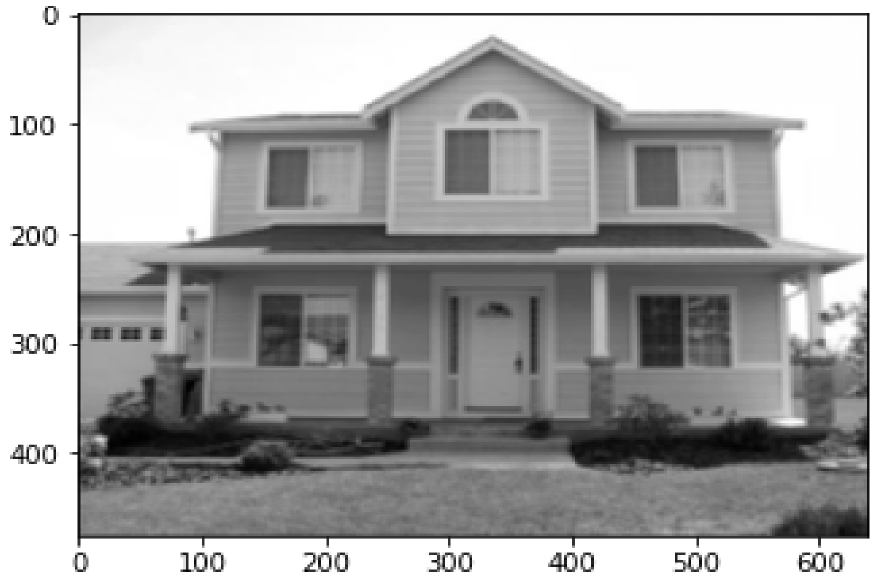



Figure 1.3: Averaging Kernel(5x5)

## 1.2 Gaussian Kernel

When working with images we need to use the two dimensional Gaussian function. This is simply the product of two 1D Gaussian functions (one for each direction) and is given by:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{1.1}$$

A graphical representation of the 2D Gaussian distribution with mean(0,0) and sigma = 1 is shown to the right.

Where sigma is the standard deviation of the distribution. The distribution is assumed to have a mean of zero. We need to discretize the continuous Gaussian functions to store it as discrete pixels.An integer valued 5 by 5 convolution kernel approximating a Gaussian

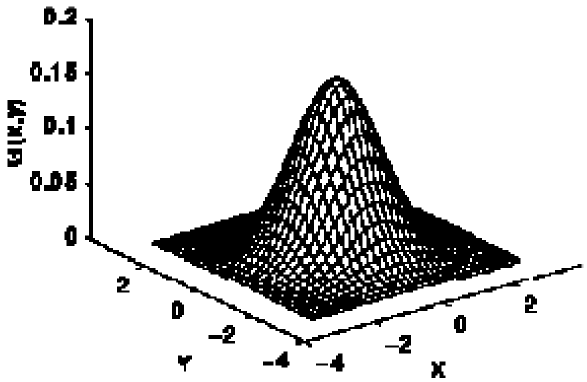



Figure 1.4: Gaussian kernel

with a sigma of 1 is shown to the right,

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \tag{1.2}$$

```
#===================== convolution.py =====================
import numpy as np
import cv2
import matplotlib.pyplot as plt



def convolution(image, kernel, average=False, verbose=False):
if len(image.shape) == 3:
print("Found 3 Channels : {}".format(image.shape))
```

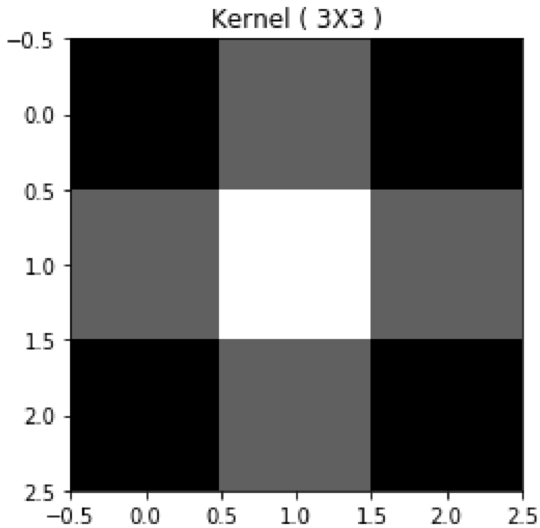
```
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
print("Converted to Gray Channel. Size : {}".format(image.shape))
else:
print("Image Shape : {}".format(image.shape))

print("Kernel Shape : {}".format(kernel.shape))

if verbose:
plt.imshow(image, cmap='gray')
plt.title("Image")
plt.show()

image_row, image_col = image.shape
kernel_row, kernel_col = kernel.shape

output = np.zeros(image.shape)

pad_height = int((kernel_row - 1) / 2)
pad_width = int((kernel_col - 1) / 2)

padded_image = np.zeros(
(image_row + (2 * pad_height), image_col + (2 * pad_width)))

padded_image[pad_height:padded_image.shape[0] - pad_height,
pad_width:padded_image.shape[1] - pad_width] = image
```

```
if verbose:
plt.imshow(padded_image, cmap='gray')
plt.title("Padded Image")
plt.show()

for row in range(image_row):
for col in range(image_col):
output[row, col] = np.sum(
kernel * padded_image[row:row + kernel_row, col:col + kernel_col])
if average:
output[row, col] /= kernel.shape[0] * kernel.shape[1]

print("Output Image size : {}".format(output.shape))

if verbose:
plt.imshow(output, cmap='gray')
plt.title("Output Image using {}X{} Kernel".format(
kernel_row, kernel_col))
plt.show()

return output
#====================== END of convolution.py ==============

#====================== gaussian_smoothing.py ====================
import numpy as np
import cv2
```

```
import argparse
import matplotlib.pyplot as plt
import math
from convolution import convolution



def dnorm(x, mu, sd):
return 1 / (np.sqrt(2 * np.pi) * sd) * np.e ** (-np.power((x - mu) / sd, 2) /



def gaussian_kernel(size, sigma, verbose=False):
kernel_1D = np.linspace(-(size // 2), size // 2, size)
for i in range(size):
kernel_1D[i] = dnorm(kernel_1D[i], 0, sigma)
kernel_2D = np.outer(kernel_1D.T, kernel_1D.T)

kernel_2D *= 1.0 / kernel_2D.max()

if verbose:
plt.imshow(kernel_2D, interpolation='none', cmap='gray')
plt.title("Kernel ( {}X{} )".format(size, size))
plt.show()

return kernel_2D

# sigma=math.sqrt(kernel_size)
```

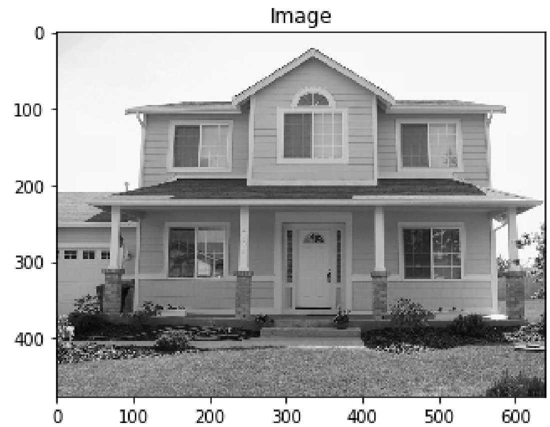
```
def gaussian_blur(image, kernel_size, sigma, verbose=False):
kernel = gaussian_kernel(
kernel_size, sigma, verbose=verbose)
return convolution(image, kernel, average=True, verbose=verbose)


if __name__ == '__main__':
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
default='B1.jpg', help="Path to the image")
ap.add_argument("-s", "--sigma", required=True,
default=1, help="sigma value")

args = vars(ap.parse_args())

image = cv2.imread(args["image"])
sigma = int(args["sigma"])
kernel_size = (2*sigma + 1)

gaussian_blur(image, kernel_size, sigma, verbose=True)
#==================== END of gaussian_smoothing.py ====================
```

kernel.png



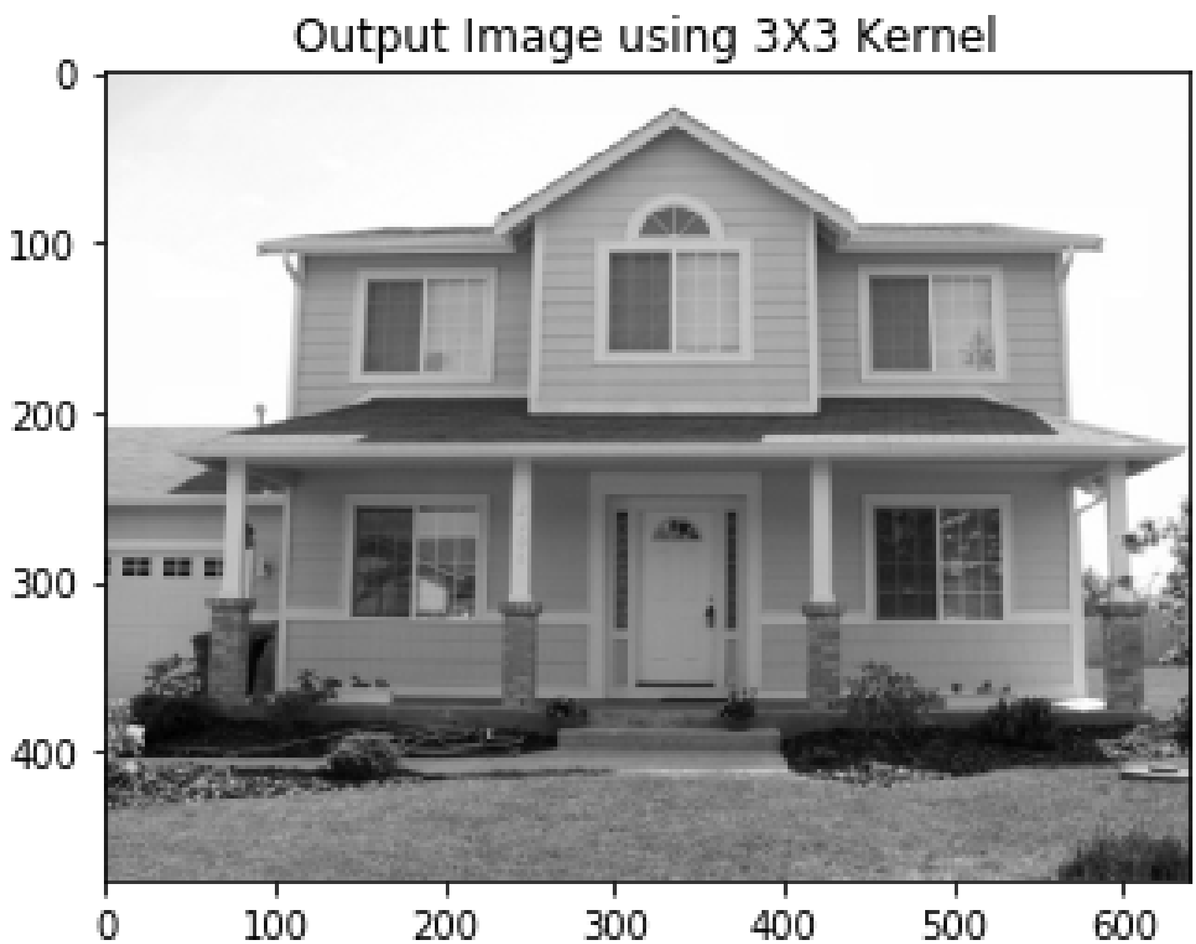

(a) 3x3 kernel

image.png




(b) main image

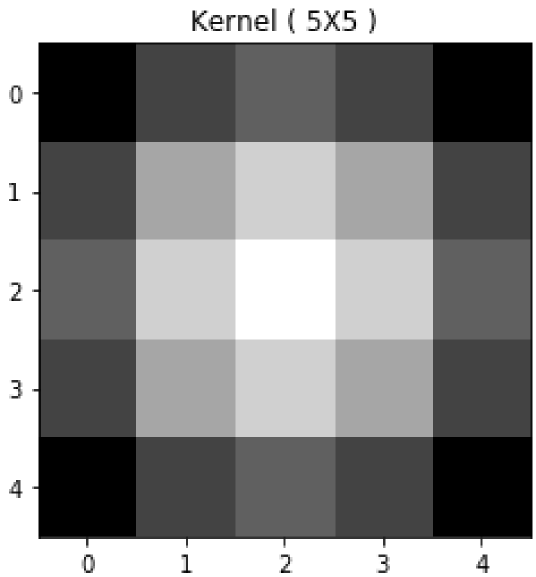
output.png




(c) 3x3 kernel output

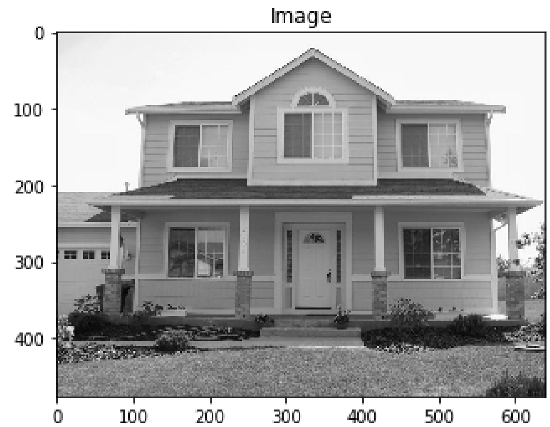
Figure 1.5: 3x3 gaussian kernel output

kernel.png



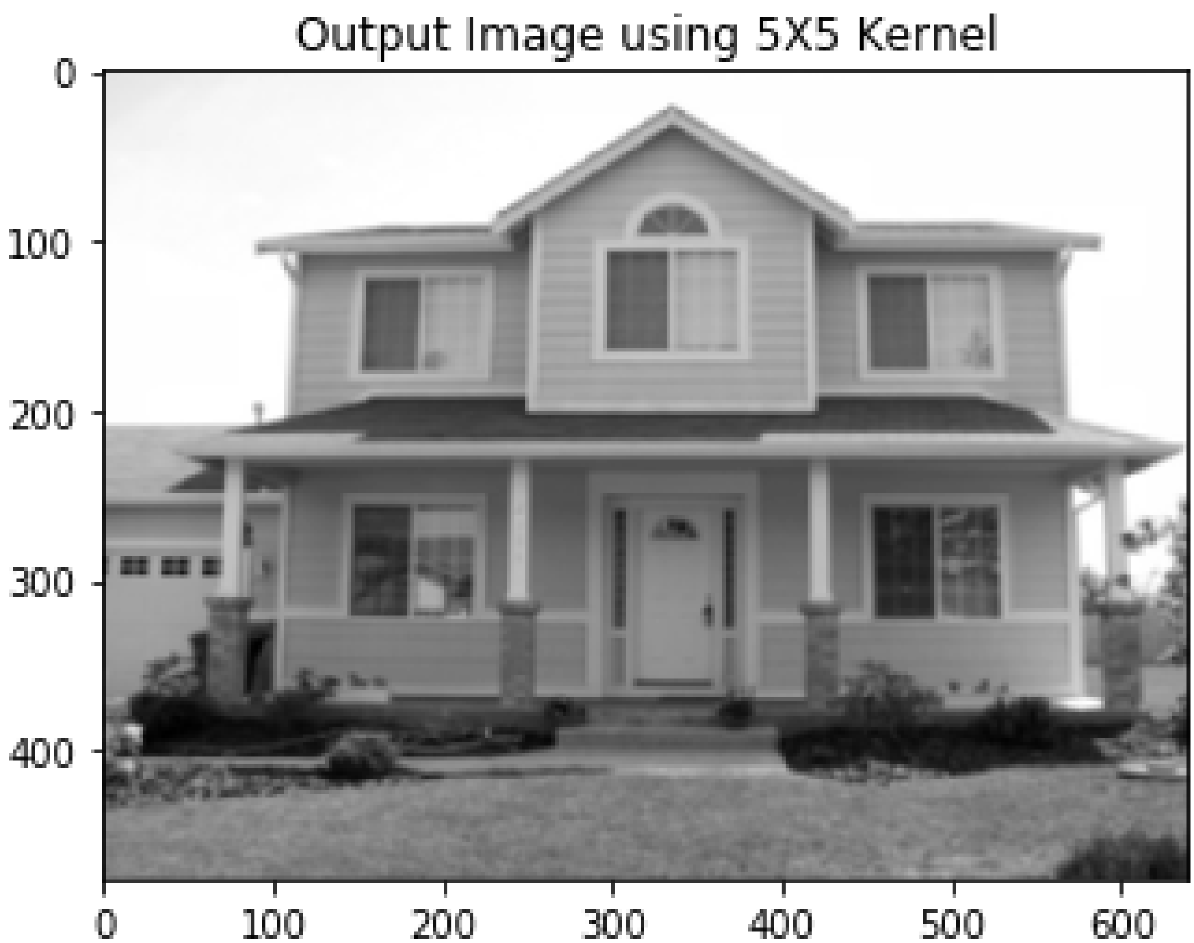

(a) 5x5 kernel

image.png




(b) main image

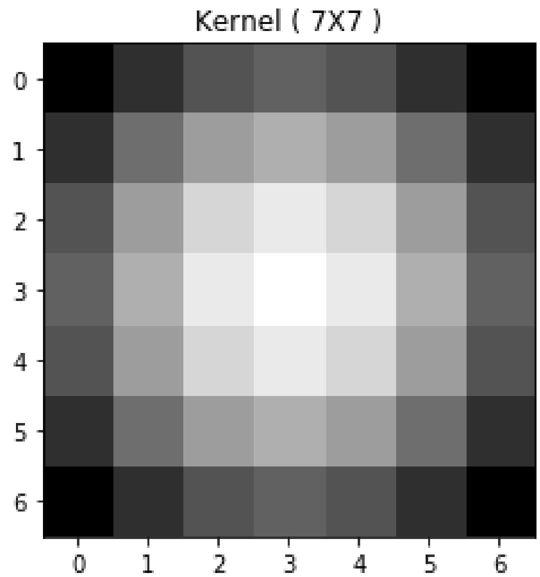
output.png




(c) 5x5 kernel output

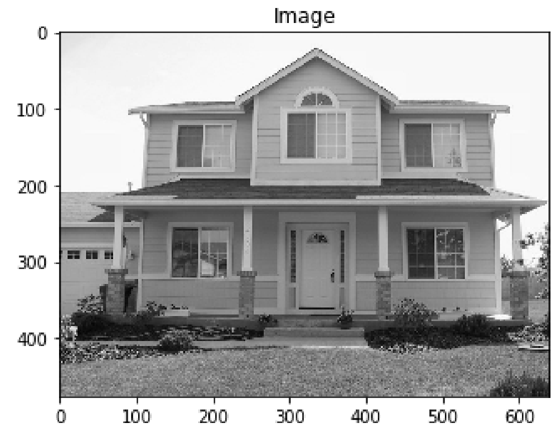
Figure 1.6: 5x5 gaussian kernel output

kernel.png



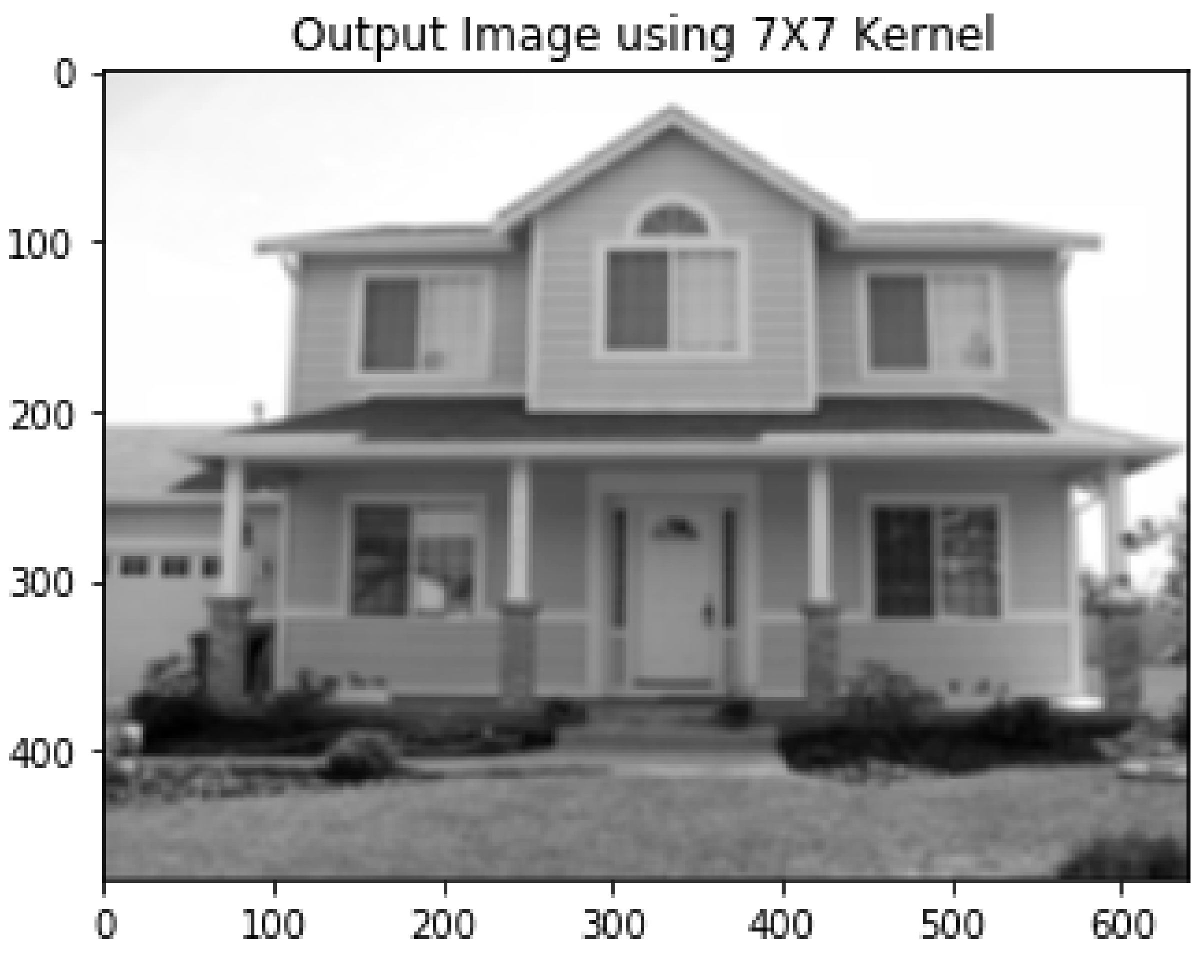

(a) 7x7 kernel

image.png




(b) main image

output.png




(c) 7x7 kernel output

Figure 1.7: 7x7 gaussian kernel output

## 1.3 Sobel Edge Operators

The sobel is one of the most commonly used edge detectors. It is based on convolving the image with a small, separable, and integer valued filter in horizontal and vertical direction and is therefore relatively inexpensive in terms of computations. The Sobel edge enhancement filter has the advantage of providing differentiating (which gives the edge response) and smoothing (which reduces noise) concurrently.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{1.3}$$

Here is a python implementation of Sobel operator:

```
#====================== sobel.py ====================
    import numpy as np
    from PIL import Image
    import matplotlib.pyplot as plt

    # Open the image
    img = np.array(Image.open('B1.jpg')).astype(np.uint8)

    # Sobel Operator
    h, w, d = img.shape

    # define filters
    horizontal = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    vertical = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
```

```
# define images with 0s
newgradientImage = np.zeros((h, w, d))

# offset by 1
for channel in range(d):
for i in range(1, h - 1):
for j in range(1, w - 1):
horizontalGrad = (horizontal[0, 0] * img[i - 1, j - 1, channel]) + \
(horizontal[0, 1] * img[i - 1, j, channel]) + \
(horizontal[0, 2] * img[i - 1, j + 1, channel]) + \
(horizontal[1, 0] * img[i, j - 1, channel]) + \
(horizontal[1, 1] * img[i, j, channel]) + \
(horizontal[1, 2] * img[i, j + 1, channel]) + \
(horizontal[2, 0] * img[i + 1, j - 1, channel]) + \
(horizontal[2, 1] * img[i + 1, j, channel]) + \
(horizontal[2, 2] * img[i + 1, j + 1, channel])

verticalGrad = (vertical[0, 0] * img[i - 1, j - 1, channel]) + \
(vertical[0, 1] * img[i - 1, j, channel]) + \
(vertical[0, 2] * img[i - 1, j + 1, channel]) + \
(vertical[1, 0] * img[i, j - 1, channel]) + \
(vertical[1, 1] * img[i, j, channel]) + \
(vertical[1, 2] * img[i, j + 1, channel]) + \
(vertical[2, 0] * img[i + 1, j - 1, channel]) + \
(vertical[2, 1] * img[i + 1, j, channel]) + \
(vertical[2, 2] * img[i + 1, j + 1, channel])
```

```
# Edge Magnitude
mag = np.sqrt(pow(horizontalGrad, 2.0) + pow(verticalGrad, 2.0))
# Avoid underflow: clip result
newgradientImage[i - 1, j - 1, channel] = mag

# now add the images r g and b
rgb_edge = newgradientImage[:,:,0] + newgradientImage[:,:,1]
 + newgradientImage[:,:,2]

plt.figure()
plt.title('dancing-spider-sobel-rgb.png')
plt.imsave('dancing-spider-sobel-rgb.png', rgb_edge, cmap='gray', for
plt.imshow(rgb_edge, cmap='gray')
plt.show()
#==================== END of sobel.py ====================
```

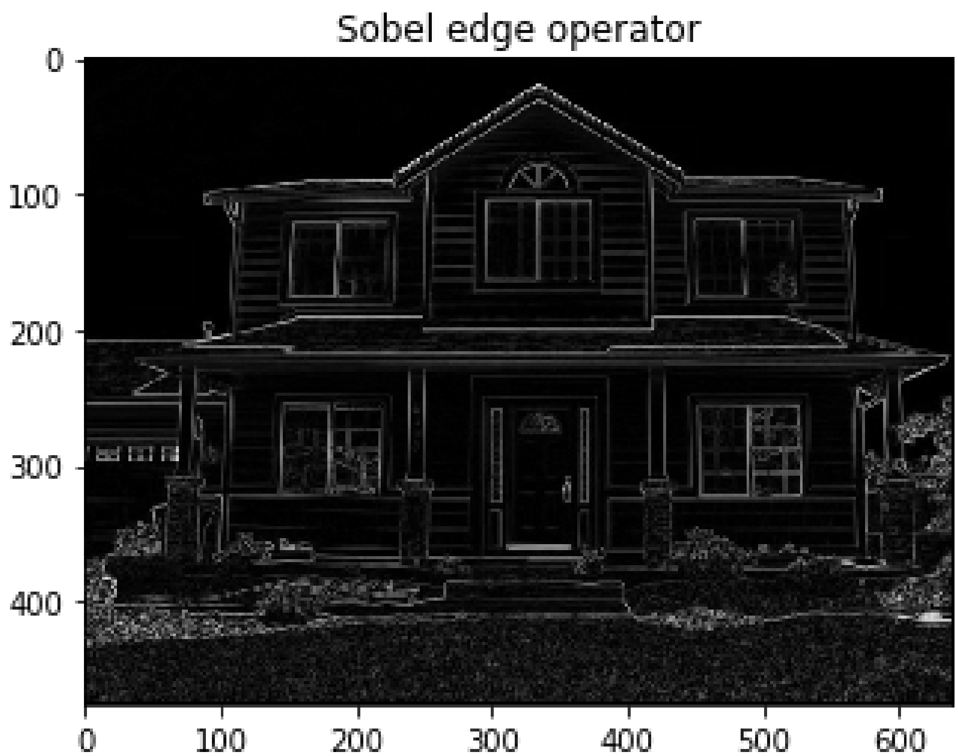



Figure 1.8: Sobel operator

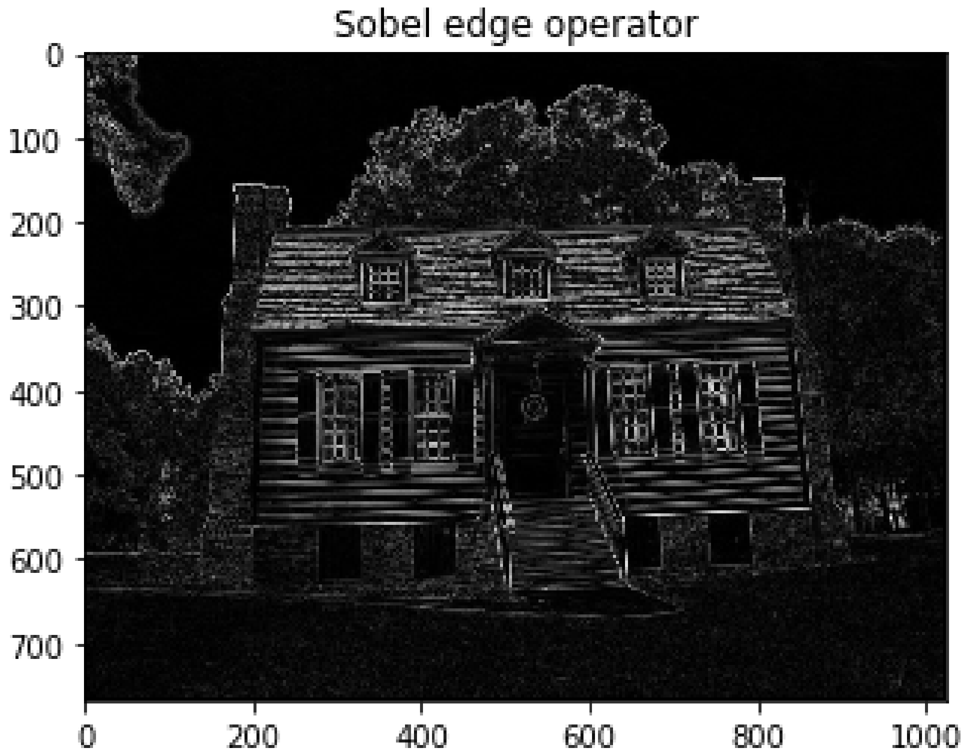



Figure 1.9: Sobel operator

## 1.4 Prewitt Edge Operators.

Prewitt operator is similar to the Sobel operator and is used for detecting vertical and horizontal edges in images. However, unlike the Sobel, this operator does not place any emphasis on the pixels that are closer to the center of the mask.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{1.4}$$

Here is a python implementation of Prewitt operator:

```
#==================== prewitt.py ====================
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Open the image
img = np.array(Image.open('dancing-spider.jpg')).astype(np.uint8)

# Prewitt Operator
h, w, d = img.shape

# define filters
horizontal = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
vertical = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])

# define images with 0s
newgradientImage = np.zeros((h, w, d))
```

```
# offset by 1
for channel in range(d):
for i in range(1, h - 1):
for j in range(1, w - 1):
horizontalGrad = (horizontal[0, 0] * img[i - 1, j - 1, channel]) + \
(horizontal[0, 1] * img[i - 1, j, channel]) + \
(horizontal[0, 2] * img[i - 1, j + 1, channel]) + \
(horizontal[1, 0] * img[i, j - 1, channel]) + \
(horizontal[1, 1] * img[i, j, channel]) + \
(horizontal[1, 2] * img[i, j + 1, channel]) + \
(horizontal[2, 0] * img[i + 1, j - 1, channel]) + \
(horizontal[2, 1] * img[i + 1, j, channel]) + \
(horizontal[2, 2] * img[i + 1, j + 1, channel])

verticalGrad = (vertical[0, 0] * img[i - 1, j - 1, channel]) + \
(vertical[0, 1] * img[i - 1, j, channel]) + \
(vertical[0, 2] * img[i - 1, j + 1, channel]) + \
(vertical[1, 0] * img[i, j - 1, channel]) + \
(vertical[1, 1] * img[i, j, channel]) + \
(vertical[1, 2] * img[i, j + 1, channel]) + \
(vertical[2, 0] * img[i + 1, j - 1, channel]) + \
(vertical[2, 1] * img[i + 1, j, channel]) + \
(vertical[2, 2] * img[i + 1, j + 1, channel])

# Edge Magnitude
```

```
mag = np.sqrt(pow(horizontalGrad, 2.0) + pow(verticalGrad, 2.0))
# Avoid underflow: clip result
newgradientImage[i - 1, j - 1, channel] = mag

# now add the images r g and b
rgb_edge = newgradientImage[:, :, 0] + \
newgradientImage[:, :, 1] + newgradientImage[:, :, 2]

plt.figure()
plt.title('dancing-spider-sobel-rgb.png')
plt.imsave('dancing-spider-sobel-rgb.png', rgb_edge, cmap='gray', format
plt.imshow(rgb_edge, cmap='gray')
plt.show()
#==================== END of prewitt.py ====================
```

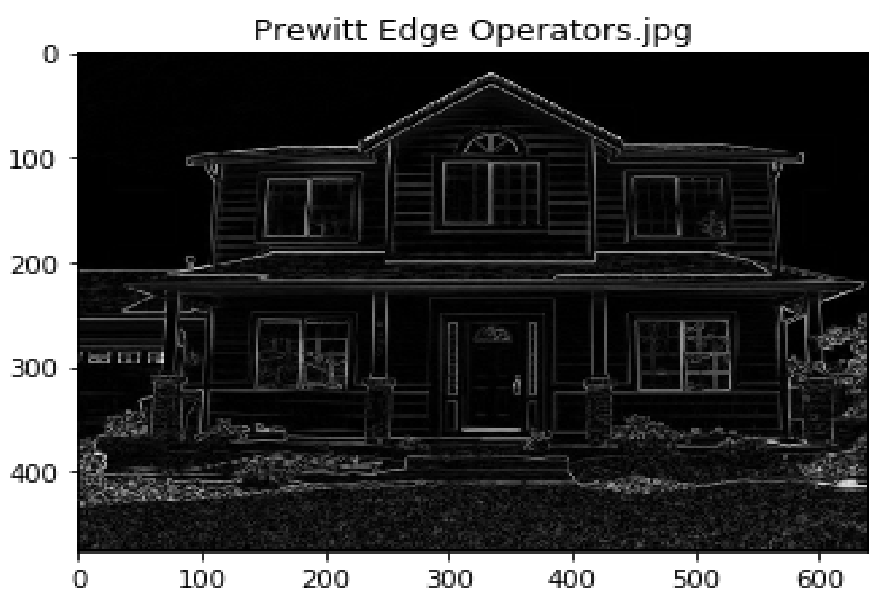



Figure 1.10: prewitt edge operator

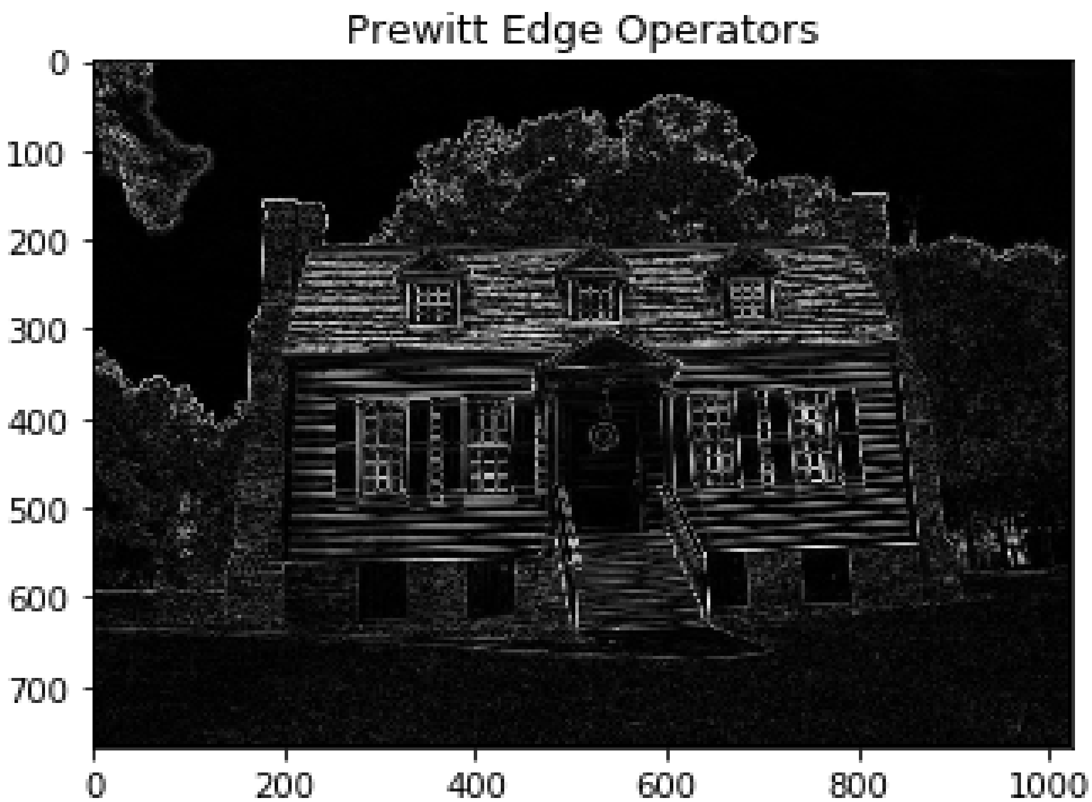



Figure 1.11: prewitt edge operator

# Chapter 2

# Answer to the question No 2

## 2.1 5 by 5 Averaging filter

5 by 5 Averaging filter

```
#=================== average.py ===================


import numpy as np
import cv2
import sys
import getopt
import matplotlib.pyplot as plt


def readImage(filename):
"""
Read in an image file, errors out if we can't find the file
```

```
:param filename: The image filename.
:return: The img object in matrix form.
"""
img = cv2.imread(filename, 0)
if img is None:
print('Invalid image:' + filename)
return None
else:
print('Image successfully read...')
return img


def integralImage(img):
"""


:param img:
:return:
"""
height = img.shape[0]
width = img.shape[1]
int_image = np.zeros((height, width), np.uint64)
for y in range(height):
for x in range(width):
up = 0 if (y-1 < 0) else int_image.item((y-1, x))
left = 0 if (x-1 < 0) else int_image.item((y, x-1))
diagonal = 0 if (
```

```
x−1 < 0 or y−1 < 0) else int_image.item((y−1, x−1))
val = img.item((y, x)) + int(up) + int(left) − int(diagonal)
int_image.itemset((y, x), val)
return int_image



def adjustEdges(height, width, point):
"""
This handles the edge cases if the box's bounds are outside the image ran
:param height: Height of the image.
:param width: Width of the image.
:param point: The current point.
:return:
"""
newPoint = [point[0], point[1]]
if point[0] >= height:
newPoint[0] = height − 1

if point[1] >= width:
newPoint[1] = width − 1
return tuple(newPoint)



def findArea(int_img, a, b, c, d):
"""
Finds the area for a particular square using the integral image. See summ
```

```
:param int_img: The
:param a: Top left corner.
:param b: Top right corner.
:param c: Bottom left corner.
:param d: Bottom right corner.
:return: The integral image.
"""
height = int_img.shape[0]
width = int_img.shape[1]
a = adjustEdges(height, width, a)
b = adjustEdges(height, width, b)
c = adjustEdges(height, width, c)
d = adjustEdges(height, width, d)

a = 0 if (a[0] < 0 or a[0] >= height) or (
a[1] < 0 or a[1] >= width) else int_img.item(a[0], a[1])
b = 0 if (b[0] < 0 or b[0] >= height) or (
b[1] < 0 or b[1] >= width) else int_img.item(b[0], b[1])
c = 0 if (c[0] < 0 or c[0] >= height) or (
c[1] < 0 or c[1] >= width) else int_img.item(c[0], c[1])
d = 0 if (d[0] < 0 or d[0] >= height) or (
d[1] < 0 or d[1] >= width) else int_img.item(d[0], d[1])

return a + d - b - c
```

```
def boxFilter(img, filterSize):
"""
Runs the subsequent box filtering steps. Prints original image, finds int
:param img: An image in matrix form.
:param filterSize: The filter size of the matrix
:return: A final image written as finalimage.png
"""
print("Printing original image...")
print(img)
height = img.shape[0]
width = img.shape[1]
intImg = integralImage(img)
finalImg = np.ones((height, width), np.uint64)
print("Printing integral image...")
print(intImg)
cv2.imwrite("integral_image.png", intImg)
loc = filterSize//2
for y in range(height):
for x in range(width):
finalImg.itemset((y, x), findArea(intImg, (y-loc-1, x-loc-1),
(y-loc-1, x+loc), (y+loc, x-loc-1), (y+loc, x+loc))//(filterSize**2))
print("Printing final image...")
print(finalImg)
plt.imshow(finalImg)

cv2.imwrite("finalimage.png", finalImg)
```

```
def main():
"""
Reads in image and handles argument parsing
:return: None
"""
args, img_name = getopt.getopt(sys.argv[1:], '', ['filter_size='])
args = dict(args)
filter_size = args.get('--filter_size')

print("Image Name: " + str(img_name[0]))
print("Filter Size: " + str(filter_size))

img = readImage(img_name[0])
if img is not None:
print("Shape: " + str(img.shape))
print("Size: " + str(img.size))
print("Type: " + str(img.dtype))
boxFilter(img, int(filter_size))



if __name__ == "__main__":
main()

#==================== END of average.py ====================
```

Figure 2.1: Noise removing with average filter

Figure 2.2: Noise removing with average filter

## 2.2 Median filter

The median filter is normally used to reduce noise in an image, somewhat like the mean filter. However, it often does a better job than the mean filter of preserving useful detail in the image.Like the mean filter, the median filter considers each pixel in the image in turn and looks at its nearby neighbors to decide whether or not it is representative of its surroundings. Instead of simply replacing the pixel value with the mean of neighboring pixel values, it replaces it with the median of those values. The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value. (If the neighborhood under consideration contains an even number of pixels, the average of the two middle pixel values is used.) Figure 2.1 illustrates an example calculation.

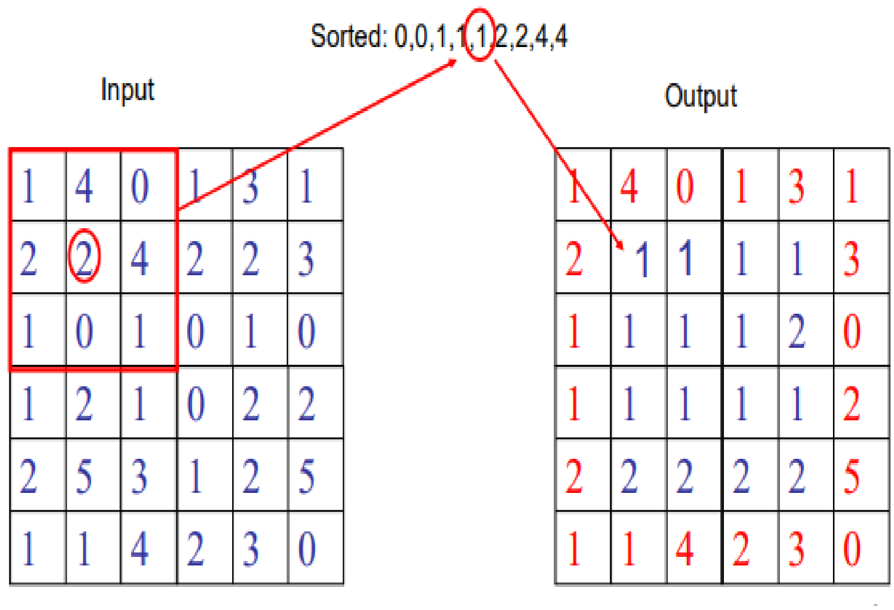



Figure 2.3: Median filter

Here is a python implementation of median filter for noise removal:

```
#====================== median.py ====================

import numpy

from PIL import Image



def median_filter(data, filter_size):

temp = []

indexer = filter_size // 2

data_final = []

data_final = numpy.zeros((len(data), len(data[0])))

for i in range(len(data)):

for j in range(len(data[0])):
```

```
for z in range(filter_size):
if i + z - indexer < 0 or i + z - indexer > len(data) - 1:
for c in range(filter_size):
temp.append(0)
else:
if j + z - indexer < 0 or j + indexer > len(data[0]) - 1:
temp.append(0)
else:
for k in range(filter_size):
temp.append(data[i + z - indexer][j + k - indexer])

temp.sort()
data_final[i][j] = temp[len(temp) // 2]
temp = []
return data_final



def main():
img = Image.open("Noisyimage1.jpg").convert(
"L")
arr = numpy.array(img)
removed_noise = median_filter(arr, 3)
img = Image.fromarray(removed_noise)
img.show()
```

```
main()
#======================== END of median.py ========================
```

Figure 2.4: Noise removing with median filter

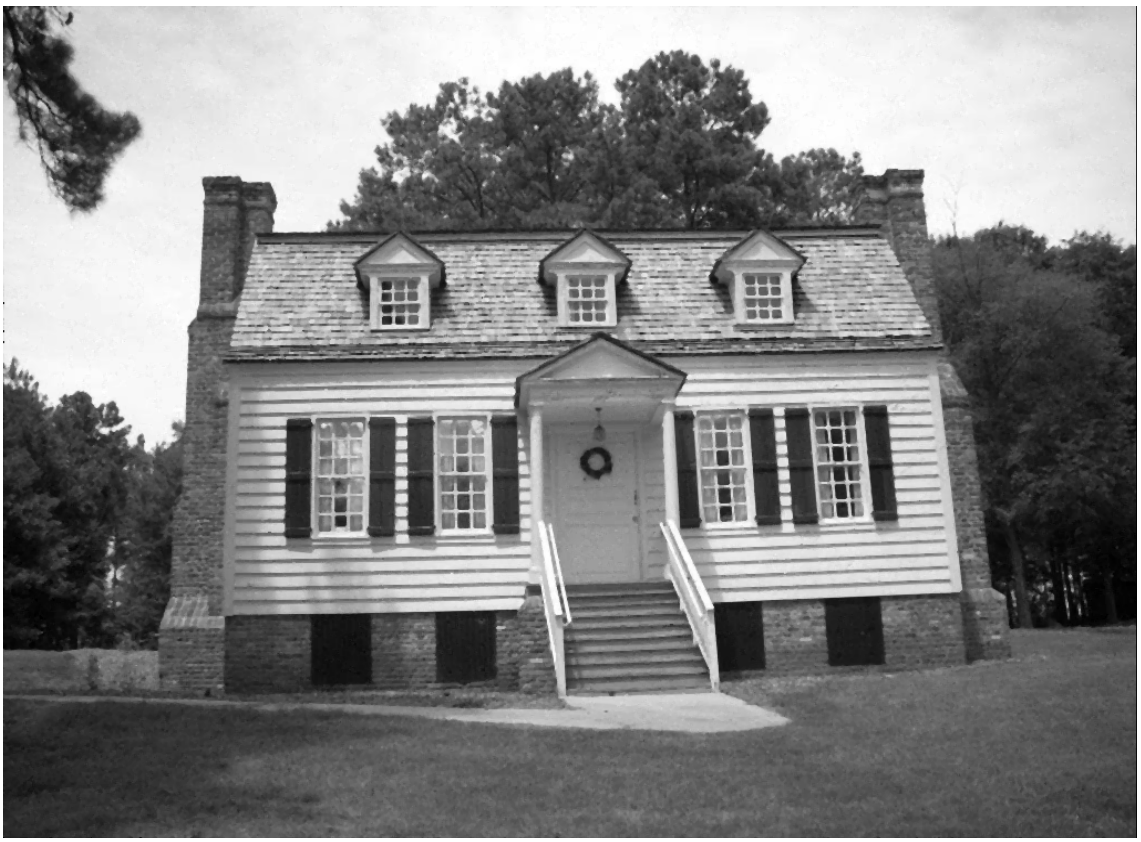


Figure 2.5: Noise removing with median filter

# Chapter 3

# Answer to the question No 3

## 3.1 Computation of gradient magnitude and gradient orientation

Here is a python implementation of gradient magnitude and gradient orientation:

```
import numpy as np
from numpy import arctan2, fliplr, flipud
import matplotlib.pyplot as plt
import cv2



def gradient(image, same_size=False):
""" Computes the Gradients of the image separated pixel difference

Gradient of X is computed using the filter
[-1, 0, 1]
Gradient of X is computed using the filter
```

```
[[1,
0,
-1]]
Parameters
----------
image: image of shape (imy, imx)
same_size: boolean, optional, default is True
If True, boundaries are duplicated so that the gradients
has the same size as the original image.
Otherwise, the gradients will have shape (imy-2, imx-2)

Returns
-------
(Gradient X, Gradient Y), two numpy array with the same shape as imag
(if same_size=True)
"""
sy, sx = image.shape
if same_size:
gx = np.zeros(image.shape)
gx[:, 1:-1] = -image[:, :-2] + image[:, 2:]
gx[:, 0] = -image[:, 0] + image[:, 1]
gx[:, -1] = -image[:, -2] + image[:, -1]

gy = np.zeros(image.shape)
gy[1:-1, :] = image[:-2, :] - image[2:, :]
gy[0, :] = image[0, :] - image[1, :]
```

```
gy[-1, :] = image[-2, :] - image[-1, :]

else:
gx = np.zeros((sy-2, sx-2))
gx[:, :] = -image[1:-1, :-2] + image[1:-1, 2:]

gy = np.zeros((sy-2, sx-2))
gy[:, :] = image[:-2, 1:-1] - image[2:, 1:-1]

return gx, gy



def magnitude_orientation(gx, gy):
""" Computes the magnitude and orientation matrices from the gradient
Parameters
----------
gx: gradient following the x axis of the image
gy: gradient following the y axis of the image

Returns
-------
(magnitude, orientation)

Warning
-------
The orientation is in degree, NOT radian!!
```

```
"""

magnitude = np.sqrt(gx**2 + gy**2)
orientation = (arctan2(gy, gx) * 180 / np.pi) % 360

return magnitude, orientation



if __name__ == '__main__':
image = cv2.cvtColor(cv2.imread("Q_3.jpg"), cv2.COLOR_BGR2GRAY)

gx, gy = gradient(image)
# print('====================value gx========')
# # print(gx)
# plt.hist(gx)
# plt.show()
# print('====================value gy========')
# # print(gy)
# plt.title('')
# plt.hist(gy)
# plt.show()

magnitude, orientation = magnitude_orientation(gx, gy)
print('====================magnitude========')
print(magnitude)
plt.title('Magnitude')
```

```
plt.hist(magnitude.ravel(), bins=50)
plt.hist(magnitude)
plt.show()
print('==================orienftation========')
print(orientation)
plt.title('orientation')
plt.hist(orientation)
plt.show()
```

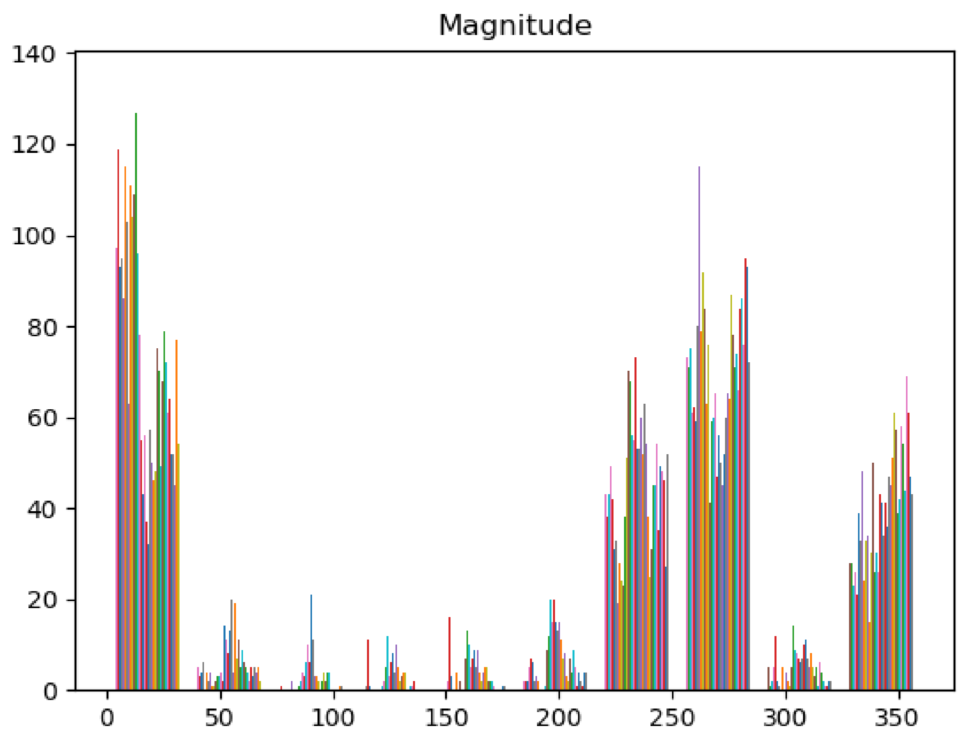



Figure 3.1: histrogram of magnitude

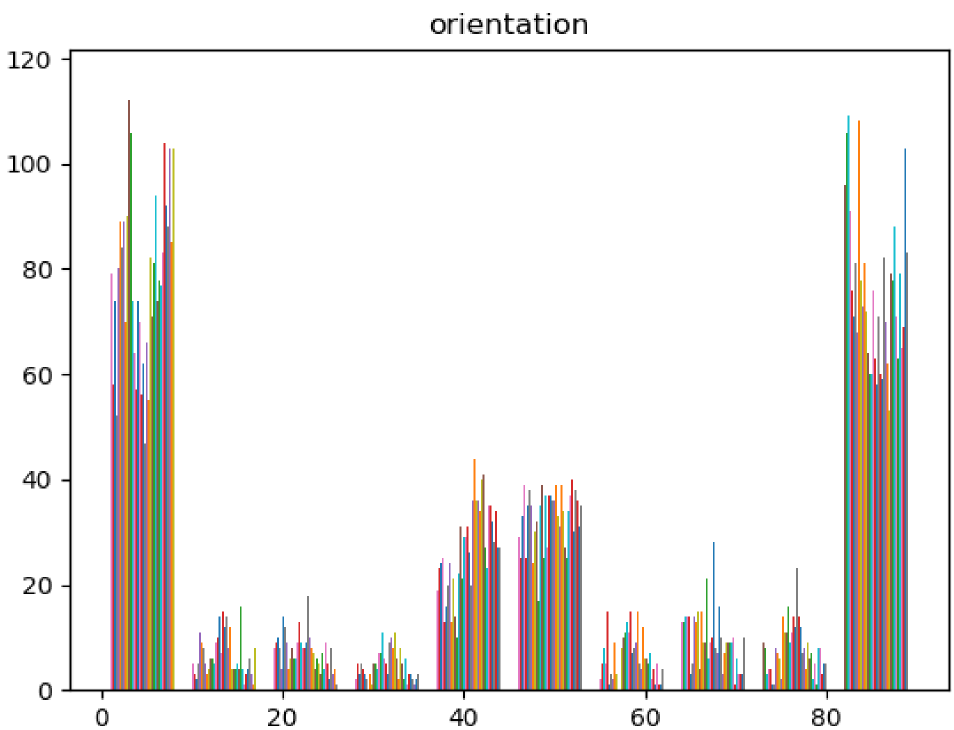



Figure 3.2: histrogram of orientation

# Chapter 4

# Answer to the question No 4

## 4.1 Image subtract

```
import cv2
img1 = cv2.imread("walk_1.jpg")
img2 = cv2.imread("walk_2.jpg")
subtraction = img1-img2
sub2 = cv2.subtract(img1, img2)
cv2.imshow("Subtraction", subtraction)
cv2.imshow("Sub 2", sub2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The ans is: by subtracting the two picture we will get the background free image ass the same thing vanish by subtraction so we get the only pedastian on the result image.

Figure 4.1: result of image subtract