Institute of Information Technology

Jahangirnagar University

Savar, Dhaka-1342

Lab Manual

Course Code: ICT-4202

Course Title: Digital Image Processing Lab

Lab No.: 7

Lab Title: Image Edge Detection Filters and Fourier Transform Filtering

Prepared by

Mehrin Anannya

Assistant Professor

Institute of Information Technology

Jahangirnagar University

Prepared using materials from the internet, especially GeekforGeeks

Lab Title: Using Roberts, Sobel, Scharr, Prewitt, and Farid named Image edge detection filters: To introduce students to image processing edge detection filters named Roberts, Sobel, Scharr, Prewitt, and Farid.

Lab Contents:

- Edge detection using Roberts, Sobel, Scharr, Prewitt, and Farid.
- Edge detection using Canny Edge Detection filter.
- Image Works with Fourier Transform Filter.

Theory with Hands on Practice:

Introduction to Edge Detection Filters:

- **Roberts**

  The idea behind the Roberts cross operator is to approximate the gradient of an image through discrete differentiation which is achieved by computing the sum of the squares of the differences between diagonally adjacent pixels. It highlights regions of high spatial gradient which often correspond to edges.

- **Sobel:**

  Similar to Roberts - calculates gradient of the image. The operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives – one for horizontal changes, and one for vertical.

- **Scharr:**

  Typically used to identify gradients along the x-axis (dx = 1, dy = 0) and y-axis (dx = 0, dy = 1) independently. Performance is quite similar to Sobel filter.

- **Prewitt:**

  The Prewitt operator is based on convolving the image with a small, separable, and integer valued filter in horizontal and vertical directions and is therefore relatively inexpensive in terms of computations like Sobel operator.

- **Farid:**

  Farid and Simoncelli propose to use a pair of kernels, one for interpolation and another for differentiation (similar to Sobel). These kernels, of fixed sizes 5 x 5 and 7 x 7, are optimized so that the Fourier transform approximates their correct derivative relationship.

Edge Detection using Roberts, Sobel, Scharr, Prewitt, and Farid

```
from skimage import io, filters, feature
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
import cv2
import numpy as np
img = cv2.imread('images/sandstone.tif', 0)
```

```
#Edge detection
from skimage.filters import roberts, sobel, scharr, prewitt, farid
roberts_img = roberts(img)
sobel_img = sobel(img)
scharr_img = scharr(img)
prewitt_img = prewitt(img)
farid_img = farid(img)

cv2.imshow("Roberts", roberts_img)
cv2.imshow("Sobel", sobel_img)
cv2.imshow("Scharr", scharr_img)
cv2.imshow("Prewitt", prewitt_img)
cv2.imshow("Farid", farid_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Edge detection using various gradient-based operators on a grayscale image. Here's an explanation of the code:

1. skimage: The scikit-image library, used for image processing tasks.
2. io: A submodule of scikit-image for input and output operations on images.
3. filters: A submodule of scikit-image containing various image filter functions.
4. feature: A submodule of scikit-image for feature detection and extraction.
5. matplotlib.pyplot (plt): A plotting library for creating visualizations in Python.

6. rgb2gray: A function from scikit-image to convert an RGB image to grayscale.
7. cv2: The OpenCV library, providing computer vision functionalities.
8. numpy (np): A library for numerical operations in Python.
9. 'images/sandstone.tif': The path to the image file.
10. img: Variable storing the grayscale image loaded using OpenCV.
11. roberts, sobel, scharr, prewitt, farid**: Various gradient-based edge detection operators available in scikit-image.
12. roberts_img, sobel_img, scharr_img, prewitt_img, farid_img: Variables storing the results of edge detection using different operators.

The gradient-based operators such as Roberts, Sobel, Scharr, Prewitt, and Farid are utilized for edge detection. These operators emphasize regions in the image where pixel intensities change rapidly, highlighting edges and boundaries. The displayed images provide visual representations of the edges detected by each operator, allowing for a comparison of their performance on the given grayscale image.

Introduction to Canny Edge Detection Filter:

Canny edge detection is a renowned image processing algorithm designed to identify edges in an image. Developed by John Canny, this technique involves computing the gradient of the image, highlighting regions where pixel intensities change rapidly. The algorithm includes steps such as gradient calculation, non-maximum suppression, and edge tracking by hysteresis. Through these steps, Canny edge detection robustly identifies significant edges while mitigating the effects of noise.

The output is a binary edge map, offering a clear representation of the most salient boundaries in the image. Known for its accuracy and versatility, the Canny edge detection filter remains a fundamental tool in computer vision, widely used for tasks like object recognition and image segmentation. Adjusting threshold values allows users to tailor the sensitivity of the detector to different image characteristics and noise levels.

The Process of Canny edge detection algorithm can be broken down to 5 different steps:

1. Apply Gaussian filter to smooth the image in order to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges (supplied by the user)
5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

Edge Detection using Canny filtering

Bilateral is slow and not very efficient at salt and pepper.

The provided code example demonstrates the use of bilateral filtering in both OpenCV (cv2.bilateralFilter) and scikit-image (skimage.restoration.denoise_bilateral). Users can adjust the parameters of the bilateral filter based on the specific characteristics of the image and the desired trade-off between smoothing and edge preservation.

```
from skimage import io, filters, feature
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
```

```
import cv2
import numpy as np
img = cv2.imread('images/sandstone.tif', 0)
#Canny
canny_edge = cv2.Canny(img, 50, 80)  #Supply Thresholds 1 and 2
#Autocanny
sigma = 0.3
median = np.median(img)
# apply automatic Canny edge detection using the computed median
lower = int(max(0, (1.0 - sigma) * median))
#Lower threshold is sigma % lower than median
#If the value is below 0 then take 0 as the value
upper = int(min(255, (1.0 + sigma) * median))
#Upper threshold is sigma% higher than median
#If the value is larger than 255 then take 255 as the value
auto_canny = cv2.Canny(img, lower, upper)
cv2.imshow("Canny", canny_edge)
cv2.imshow("Auto Canny", auto_canny)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Here's an explanation of the code:

1. `skimage`: The scikit-image library, used for image processing tasks.
2. `io`: A submodule of scikit-image for input and output operations on images.
3. `filters, feature`: Submodules of scikit-image containing various image filter and feature detection functions.
4. `matplotlib.pyplot (plt)`: A plotting library for creating visualizations in Python.
5. `rgb2gray`: A function from scikit-image to convert an RGB image to grayscale.
6. `cv2`: The OpenCV library, providing computer vision functionalities.
7. `numpy (np)`: A library for numerical operations in Python.
8. `'images/sandstone.tif'`: The path to the image file.
9. `img`: Variable storing the grayscale image loaded using OpenCV.
10. `cv2.Canny`: The Canny edge detection function in OpenCV. It takes the image and two threshold values as parameters. Any edges with intensity gradient more than maxVal are sure to be edges and those below minVal are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. The edges are detected based on intensity gradients, and the output is a binary image with edges.
11. `canny_edge`: Variable storing the result of Canny edge detection.
12. `sigma`: A parameter used in the computation of automatic Canny thresholds.
13. `median`: The median pixel intensity of the image.
14. `lower, upper`: Computed thresholds for automatic Canny edge detection.
15. `auto_canny`: Variable storing the result of automatic Canny edge detection.
16. The code displays the images obtained from both standard Canny edge detection and the automatic Canny edge detection using OpenCV's `cv2.imshow`.

17. `cv2.waitKey(0)`: Waits for a key event. It waits indefinitely (0) for a key press.
18. `cv2.destroyAllWindows()`: Closes all OpenCV windows.

Canny edge detection is a widely used method for detecting edges in images, and the automatic version allows for adaptive thresholding based on the image's characteristics. The code provides a visual comparison between the standard Canny edge detection and the automatic Canny approach.

Introduction to Fourier Transform:

Spatial Domain: This involves enhancing the image by manipulating individual pixels based on their spatial coordinates at a specific resolution.

Frequency Domain: This approach involves enhancing the image by applying a Fourier Transform to the spatial domain, manipulating pixels in groups and indirectly.

In image processing, the most common way to represent pixel location is in the spatial domain by column (x), row (y), and z (value). But sometimes image processing routines may be slow or inefficient in the spatial domain, requiring a transformation to a different domain that offers compression benefits.

A common transformation is from the spatial to the frequency (or Fourier) domain. The frequency domain is the basis for many image filters used to remove noise, sharpen an image, analyze repeating patterns, or extract features. In the frequency domain, pixel location is represented by its x- and y-frequencies and its value is represented by amplitude.

The Fast Fourier Transform (FFT) is commonly used to transform an image between the spatial and frequency domain. Unlike other domains such as Hough and Radon, the FFT

method preserves all original data. Plus, FFT fully transforms images into the frequency domain, unlike time-frequency or wavelet transforms. The FFT decomposes an image into sines and cosines of varying amplitudes and phases, which reveals repeating patterns within the image.

Low frequencies represent gradual variations in the image; they contain the most information because they determine the overall shape or pattern in the image. High frequencies correspond to abrupt variations in the image; they provide more detail in the image, but they contain more noise. One way to filter out background noise is to apply a mask. See Use FFT to Reduce Background Noise for an example.

The filtering process involves convolution or multiplication in either the spatial or frequency domain. Convolution in the spatial domain corresponds to element-wise multiplication in the frequency domain. After filtering, the inverse Fourier Transform is applied to return the image to the spatial domain.

This frequency domain approach to filtering finds applications in diverse areas, such as image enhancement, noise reduction, and edge detection. It provides a powerful framework for tailoring image processing techniques to specific objectives, leveraging the intrinsic frequency information embedded in digital images.

Image Works using Fourier Transform

```
import cv2
from matplotlib import pyplot as plt
import numpy as np
```

```
#Generate a 2D sine wave image
x = np.arange(256)  # generate values from 0 to 255 (our image size)
y = np.sin(2 * np.pi * x / 3)  #calculate sine of x values
#Divide by a smaller number above to increase the frequency.
y += max(y) # offset sine wave by the max value to go out of negative range of sine

#Generate a 256x256 image (2D array of the sine wave)
img = np.array([[y[j]*127 for j in range(256)] for i in range(256)], dtype=np.uint8) #
create 2-D array of sine-wave

#plt.imshow(img)
#img = np.rot90(img)  #Rotate img by 90 degrees

img = cv2.imread('images/sandstone.tif', 0) # load an image
dft = cv2.dft(np.float32(img), flags=cv2.DFT_COMPLEX_OUTPUT)

#Shift DFT. First check the output without the shift
#Without shifting the data would be centered around origin at the top left
#Shifting it moves the origin to the center of the image.
dft_shift = np.fft.fftshift(dft)

#Calculate magnitude spectrum from the DFT (Real part and imaginary part)
#Added 1 as we may see 0 values and log of 0 is indeterminate
```

So, [:, :, 0] is selecting all elements in the first and second dimensions, but only the elements with index 0 in the third dimension. This is often used in multidimensional arrays or images to select a specific channel or layer.

```
magnitude_spectrum = 20 * np.log((cv2.magnitude(dft_shift[:, :, 0], dft_shift[:, :, 1]))+1)
#As the spatial frequency increases (bars closer),
#the peaks in the DFT amplitude spectrum move farther away from the origin
#Center represents low frequency and the corners high frequency (with DFT shift).
#To build high pass filter block center corresponding to low frequencies and let
#high frequencies go through. This is nothing but an edge filter.

fig = plt.figure(figsize=(12, 12))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(img)
ax1.title.set_text('Input Image')
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(magnitude_spectrum)
ax2.title.set_text('FFT of image')
plt.show()
```

The provided code generates a 2D sine wave image and performs a Fourier Transform on another image using the OpenCV library. Here's an explanation of the code:

1. cv2: The OpenCV library, providing computer vision functionalities.
2. matplotlib.pyplot (plt): A plotting library for creating visualizations in Python.
3. numpy (np): A library for numerical operations in Python.

The code can be divided into two main parts:

1. Generation of 2D Sine Wave Image

```python
x = np.arange(256)  # generate values from 0 to 255 (image size)
y = np.sin(2 * np.pi * x / 3)  # calculate sine of x values
y += max(y)  # offset sine wave by the max value to go out of the negative range of sine

# Generate a 256x256 image (2D array of the sine wave)
img = np.array([[y[j]*127 for j in range(256)] for i in range(256)], dtype=np.uint8)
```

This part creates a 2D array representing a sine wave and stores it in the variable `img`. It is then visualized using matplotlib, but the visualization is commented out.

2. Fourier Transform of an Image

```python
img = cv2.imread('images/sandstone.tif', 0)  # load an image
dft = cv2.dft(np.float32(img), flags=cv2.DFT_COMPLEX_OUTPUT)
# Shift DFT to center the data around the origin
dft_shift = np.fft.fftshift(dft)

# Calculate the magnitude spectrum from the DFT (Real part and imaginary part)
```

magnitude_spectrum = 20 * np.log((cv2.magnitude(dft_shift[:, :, 0], dft_shift[:, :, 1]))+1)

```

This part reads an image ('sandstone.tif' in this case), applies the Fourier Transform to it, shifts the data to center it around the origin, and calculates the magnitude spectrum of the shifted DFT. Finally, it visualizes both the input image and the magnitude spectrum using matplotlib.

The code provides insight into the frequency content of the image using the Fourier Transform, showcasing the spatial frequencies present in the image.

## Image Works using Fourier Transform Filter:

```
import cv2
from matplotlib import pyplot as plt
import numpy as np

img = cv2.imread('images/sandstone.tif', 0) # load an image

#Output is a 2D complex array. 1st channel real and 2nd imaginary
#For fft in opencv input image needs to be converted to float32
dft = cv2.dft(np.float32(img), flags=cv2.DFT_COMPLEX_OUTPUT)

#Rearranges a Fourier transform X by shifting the zero-frequency
#component to the center of the array.
#Otherwise it starts at the tope left corner of the image (array)
dft_shift = np.fft.fftshift(dft)

##Magnitude of the function is 20.log(abs(f))
```
```

```
#For values that are 0 we may end up with indeterminate values for log.
#So we can add 1 to the array to avoid seeing a warning.
magnitude_spectrum = 20 * np.log(cv2.magnitude(dft_shift[:, :, 0], dft_shift[:, :, 1]))



# Circular HPF mask, center circle is 0, remaining all ones
#Can be used for edge detection because low frequencies at center are blocked
#and only high frequencies are allowed. Edges are high frequency components.
#Amplifies noise.

rows, cols = img.shape
crow, ccol = int(rows / 2), int(cols / 2)

mask = np.ones((rows, cols, 2), np.uint8)
r = 80
center = [crow, ccol]
x, y = np.ogrid[:rows, :cols]
mask_area = (x - center[0]) ** 2 + (y - center[1]) ** 2 <= r*r
mask[mask_area] = 0



# Circular LPF mask, center circle is 1, remaining all zeros
# Only allows low frequency components - smooth regions
#Can smooth out noise but blurs edges.
#
"""
rows, cols = img.shape
crow, ccol = int(rows / 2), int(cols / 2)
```

```
mask = np.zeros((rows, cols, 2), np.uint8)
r = 100
center = [crow, ccol]
x, y = np.ogrid[:rows, :cols]
mask_area = (x - center[0]) ** 2 + (y - center[1]) ** 2 <= r*r
mask[mask_area] = 1

# Band Pass Filter - Concentric circle mask, only the points living in concentric circle are ones
rows, cols = img.shape
crow, ccol = int(rows / 2), int(cols / 2)

mask = np.zeros((rows, cols, 2), np.uint8)
r_out = 80
r_in = 10
center = [crow, ccol]
x, y = np.ogrid[:rows, :cols]
mask_area = np.logical_and(((x - center[0]) ** 2 + (y - center[1]) ** 2 >= r_in ** 2),
                           ((x - center[0]) ** 2 + (y - center[1]) ** 2 <= r_out ** 2))
mask[mask_area] = 1
"""



# apply mask and inverse DFT: Multiply fourier transformed image (values)
#with the mask values.
fshift = dft_shift * mask

#Get the magnitude spectrum (only for plotting purposes)
epsilon = 1e-8  # Small constant to avoid zero in logarithm
```

```
fshift_mask_mag = 20 * np.log(cv2.magnitude(fshift[:, :, 0], fshift[:, :, 1]) + epsilon)


#Inverse shift to shift origin back to top left.
f_ishift = np.fft.ifftshift(fshift)

#Inverse DFT to convert back to image domain from the frequency domain.
#Will be complex numbers
img_back = cv2.idft(f_ishift)

#Magnitude spectrum of the image domain
img_back = cv2.magnitude(img_back[:, :, 0], img_back[:, :, 1])
fig = plt.figure(figsize=(12, 12))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(img, cmap='gray')
ax1.title.set_text('Input Image')
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(magnitude_spectrum, cmap='gray')
ax2.title.set_text('FFT of image')
ax3 = fig.add_subplot(2,2,3)
ax3.imshow(fshift_mask_mag, cmap='gray')
ax3.title.set_text('FFT + Mask')
ax4 = fig.add_subplot(2,2,4)
ax4.imshow(img_back, cmap='gray')
ax4.title.set_text('After inverse FFT')
plt.show()
```

Let's break down the provided code step by step:

The code demonstrates the application of a high-pass filter in the frequency domain using the Discrete Fourier Transform (DFT) and Inverse DFT in OpenCV. Here's a detailed explanation of each part:

1. Load the Image:

```
img = cv2.imread('images/sandstone.tif', 0)
```

- Uses OpenCV's `cv2.imread` function to load the grayscale image 'sandstone.tif'.

2. Discrete Fourier Transform (DFT):

```
dft = cv2.dft(np.float32(img), flags=cv2.DFT_COMPLEX_OUTPUT)
```

- Applies the DFT to the input image using `cv2.dft`. The result is a 2D complex array with the real part in the first channel and the imaginary part in the second channel.

3. Shift Zero Frequency Component to Center:

```
dft_shift = np.fft.fftshift(dft)
```

- Shifts the zero-frequency component to the center using `np.fft.fftshift`.

4. Magnitude Spectrum Calculation:

```
magnitude_spectrum = 20 * np.log(cv2.magnitude(dft_shift[:, :, 0], dft_shift[:, :, 1]) + 1)
```

- Calculates the magnitude spectrum of the shifted DFT. The `cv2.magnitude` function computes the magnitude of a complex number, and `np.log` is applied to it. Adding 1 is to avoid issues when taking the logarithm of zero.

5. Circular High-Pass Filter Design:

```
mask = np.ones((rows, cols, 2), np.uint8)
r = 80
center = [crow, ccol]
x, y = np.ogrid[:rows, :cols]
mask_area = (x - center[0]) ** 2 + (y - center[1]) ** 2 <= r*r
mask[mask_area] = 0
```

- Designs a circular high-pass filter by creating a binary mask with ones everywhere and zeros in the circular region. This filter blocks low-frequency components and allows high-frequency components.

6. Apply the Filter and Inverse DFT:

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

- Applies the high-pass filter in the frequency domain by element-wise multiplication (`*`). Then, the `np.fft.ifftshift` function is used to inverse shift the zero frequency component back to the top-left. Finally, the Inverse DFT (`cv2.idft`) is applied to obtain the image in the spatial domain.

7. Display the Results:

- Displays a 2x2 subplot figure:
  - The original input image.
  - The magnitude spectrum of the image in the frequency domain.
  - The frequency spectrum after applying the high-pass filter.
  - The resulting image after applying the Inverse DFT.

This code visually demonstrates the impact of a circular high-pass filter on the frequency components of an image, emphasizing edges and fine details while suppressing low-frequency components. The subplots allow for a side-by-side comparison of each stage in the process.

Tasks:

1. Take two images and compare between any two edge detection techniques and write necessary comments on it mentioning which filtering techniques works well on the images.
2. Apply Fourier transform on those images.
3. Apply three pass filters on them and write you comment mentioning which filtering process is more feasible to extract the actual image.