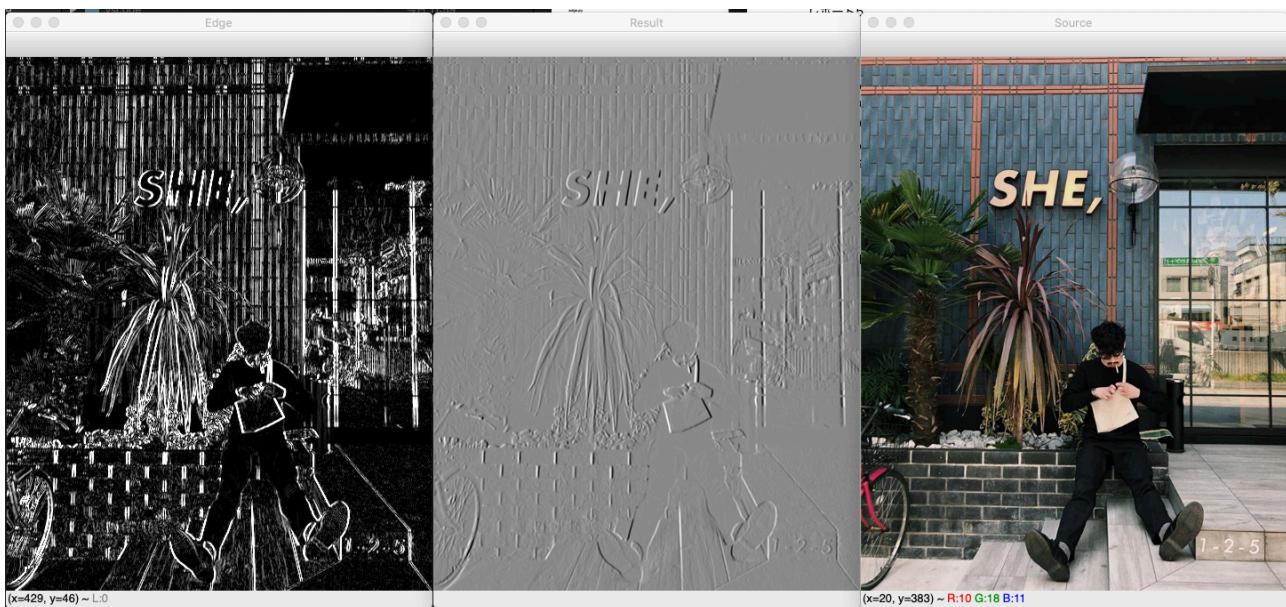


レポート5  
08-172022  
永田 謙

**linear.py**  
実行結果



コード：

```
import cv2
import numpy as np
import time
from grayscale import readImage


def myFilter2D(src, kernel):
    drows, dcols = src.shape[:2]
    dst = np.zeros((drows, dcols), np.float64)
    krows, kcols = kernel.shape[:2]
    ahgt = krows // 2
    awth = kcols // 2
    org = cv2.copyMakeBorder(src, ahgt, ahgt, awth, awth,
cv2.BORDER_DEFAULT)
    for drow in range(0, drows):
        for dcol in range(0, dcols):
            r = 0.0
            for krow in range(0, krows):
                for kcol in range(0, kcols):
```

```
        r += org[drow+krow, dcol+kcol] * kernel[krow,
kcol]
    dst[drow, dcol] = r
return dst
```

```
def main():
    img = readImage()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    m = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
    kernel = np.array(m)
    s = time.time()

    tmp = myFilter2D(gray, kernel)

    print('time:', time.time()-s, 'sec')
    minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(tmp)
    gray = (tmp - minVal) / (maxVal - minVal)
    edge = cv2.convertScaleAbs(tmp)
    cv2.imshow('Source', img)
    cv2.imshow('Result', gray)
    cv2.imshow('Edge', edge)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

```
if __name__ == '__main__':
    main()
```

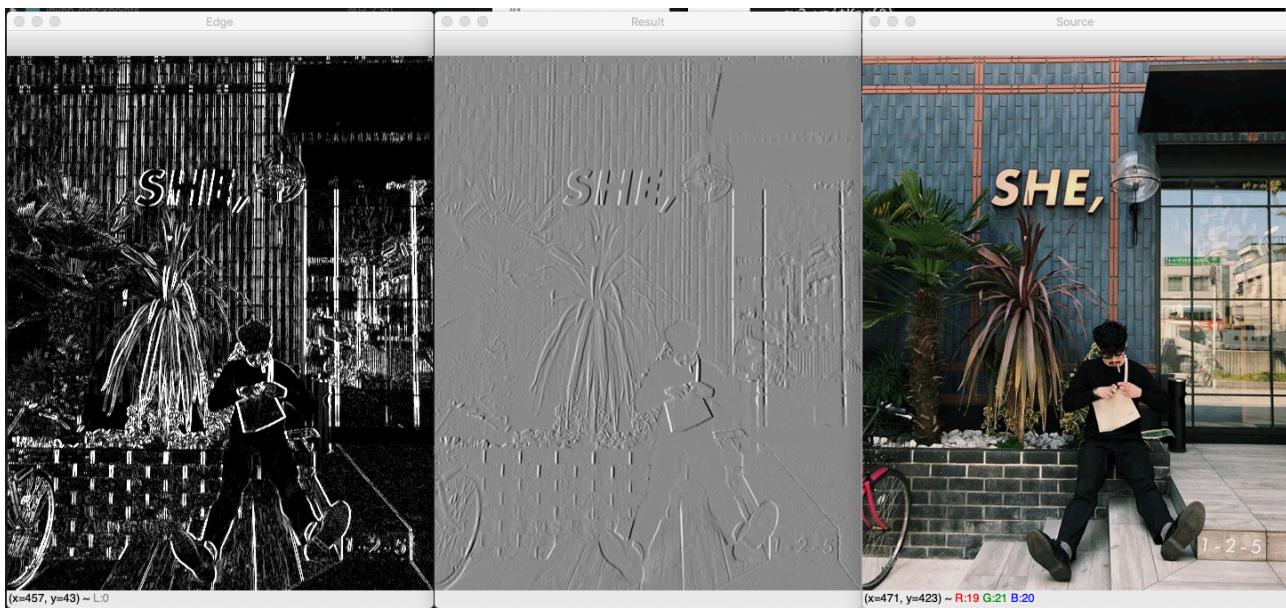
考察：

ソーベルフィルターを素朴に実装している。

ソーベルフィルタは輪郭を抽出するために用いられる空間フィルターで、この実装を見ると。

## linearNP.py

### 実行結果



コード：

```
import cv2
import numpy as np
import time
from grayscale import readImage


def myFilter2D(src, kernel):
    drows, dcols = src.shape[:2]
    dst = np.zeros((drows, dcols), np.float64)
    krows, kcols = kernel.shape[:2]
    ahgt = krows // 2
    awth = kcols // 2
    org = cv2.copyMakeBorder(src, ahgt, ahgt, awth, awth,
cv2.BORDER_DEFAULT)
    org = np.float64(org)
    for krow in range(0, krows):
        for kcol in range(0, kcols):
            dst += org[krow:krow+drows, kcol:kcol+dcols] *
kernel[krow, kcol]
    return dst


def main():
    img = readImage()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    m = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
```

```
kernel = np.array(m)
s = time.time()

tmp = myFilter2D(gray, kernel)

print('time:', time.time()-s, 'sec')
minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(tmp)
gray = (tmp - minVal) / (maxVal - minVal)
edge = cv2.convertScaleAbs(tmp)
cv2.imshow('Source', img)
cv2.imshow('Result', gray)
cv2.imshow('Edge', edge)
cv2.waitKey(0)
cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

考察：

前回と同じであるが、

```
dst += org[krow:krow+drows, kcol:kcol+dcols] * kernel[krow, kcol]
```

ここのループが解消されたことによりパフォーマンスが向上している。

インタプリタ型の言語であるpythonのnumpyでの計算がなぜ早いのか気になって調べてみたら、内部実装はcppであった。openCVも内部はpythonではなく、他の言語とのブリッジが多いいため、他の分野との学際的な関わりの大きな、AIの分野で使われているのだろうかと思った。

## linearCV.py

実行結果：



コード：

```
import cv2
import numpy as np
import time
from grayscale import readImage


def main():
    img = readImage()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    m = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
    kernel = np.array(m)
    s = time.time()

    tmp = cv2.filter2D(gray, cv2.CV_64F, kernel)

    print('time:', time.time()-s, 'sec')
    minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(tmp)
    gray = (tmp - minVal) / (maxVal - minVal)
    edge = cv2.convertScaleAbs(tmp)
    cv2.imshow('Source', img)
    cv2.imshow('Result', gray)
    cv2.imshow('Edge', edge)
```

```
cv2.waitKey(0)
cv2.destroyAllWindows()

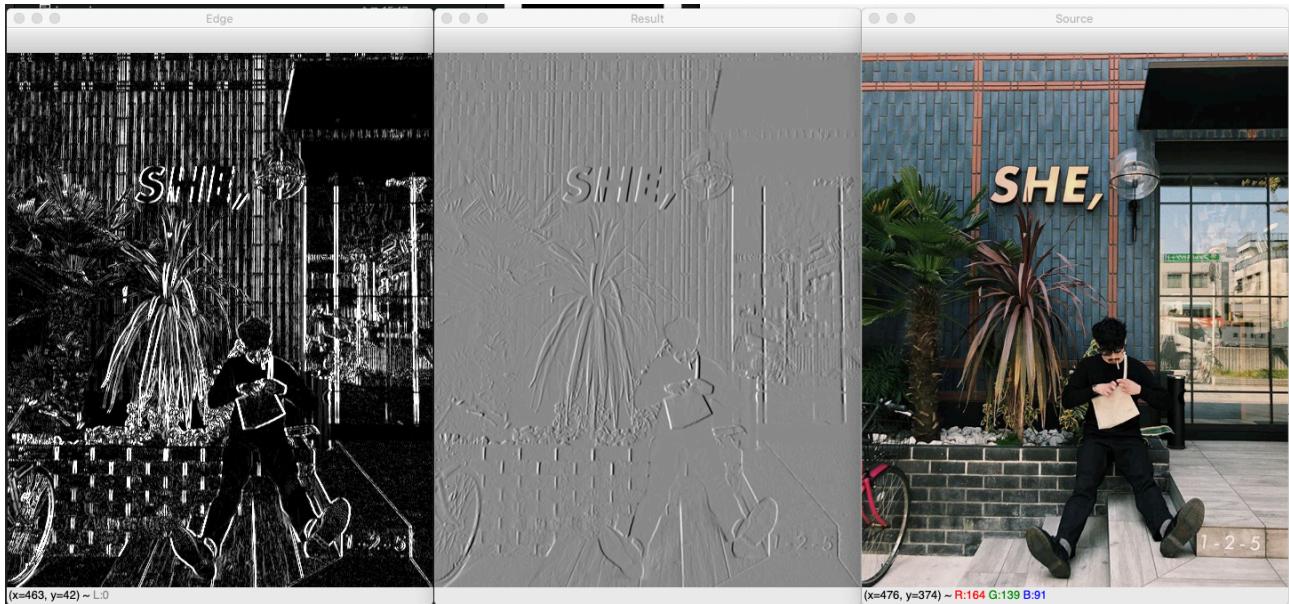
if __name__ == '__main__':
    main()
```

考察：

ここではOpenCVで定義済みのフィルターを使った。

## Sobel.py

### 実行結果



コード：

```
import cv2
import time
from grayscale import readImage

def main():
    img = readImage()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    s = time.time()
    tmp = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
    print('time:', time.time()-s, 'sec')
    minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(tmp)
    gray = (tmp - minVal) / (maxVal - minVal)
    edge = cv2.convertScaleAbs(tmp)
    cv2.imshow('Source', img)
    cv2.imshow('Result', gray)
    cv2.imshow('Edge', edge)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

考察：

OpenCVにはあらかじめ。ソーベルフィルタも存在している。  
いってしまえば、カーネル行列が組み込まれているだけである。

1.11-5は存在しない。

## edge.py

### 実行結果



コード：

```
import cv2
from grayscale import readImage

def main():
    img = readImage()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    cv2.imshow('Source', gray)
    k = 3
    tmp = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=k)
    tmp = cv2.convertScaleAbs(tmp)
    cv2.imshow('SobelX', tmp)
    tmp = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=k)
    tmp = cv2.convertScaleAbs(tmp)
    cv2.imshow('SobelY', tmp)
    tmp = cv2.Laplacian(gray, cv2.CV_64F, ksize=k)
    tmp = cv2.convertScaleAbs(tmp)
    cv2.imshow('Laplacian', tmp)
    lower, upper = 180, 250
    tmp = cv2.Canny(gray, lower, upper)
    tmp = cv2.convertScaleAbs(tmp)
    cv2.imshow('canny', tmp)
```

```
cv2.waitKey(0)
cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

### 考察：

様々なフィルターを使って輪郭抽出をした。

それぞれに特徴があり、

SobelX：ソーベルフィルタをX方向に適応させたもの

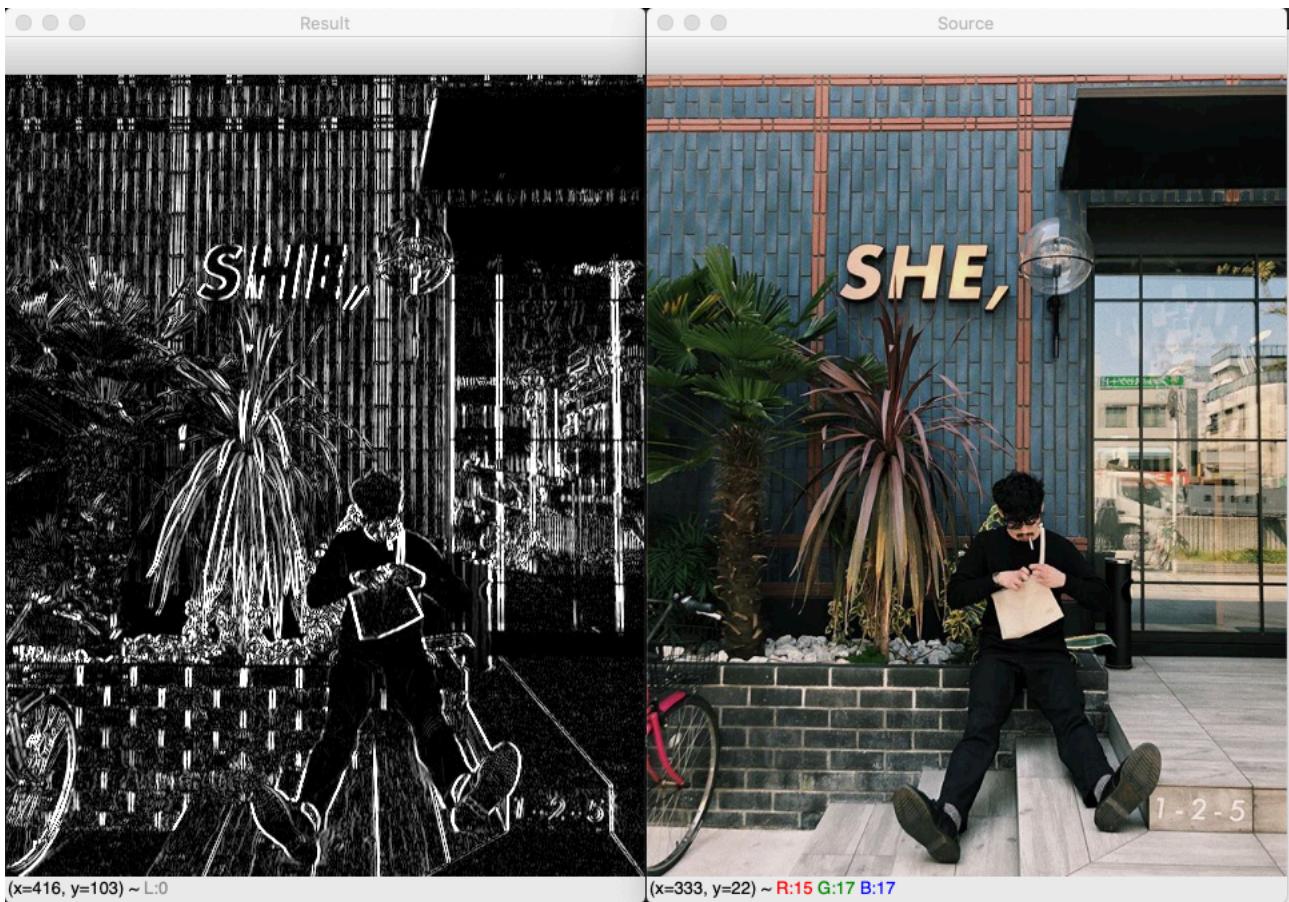
SobelY：ソーベルフィルタをY方向に適応させたもの

Laplacian：ラプラシアンフィルタはソーベルフィルタが1次微分系のフィルタなのに対して、二  
次微分のフィルタである。はっきりと輪郭が取れている。

canny：ノイズを消去したのちにソーベルフィルタで勾配の大きさと方向を求め、勾配方向と大き  
さを元に細線化し、閾値化でエッジを検出している。最初にノイズの消去を行なっているから  
か、無駄な線がほとんどない。

## sepDeriv.py

実行結果：



コード：

```
import cv2
from grayscale import readImage

def main():
    img = readImage()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ksize = 3
    krow, kcol = cv2.getDerivKernels(1, 0, ksize)
    tmp = cv2.sepFilter2D(gray, cv2.CV_64F, krow, kcol)
    gray = cv2.convertScaleAbs(tmp)
    print('row kernel matrix:n', krow)
    print('column kernel matrix:n', kcol)
    cv2.imshow('Source', img)
    cv2.imshow('Result', gray)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

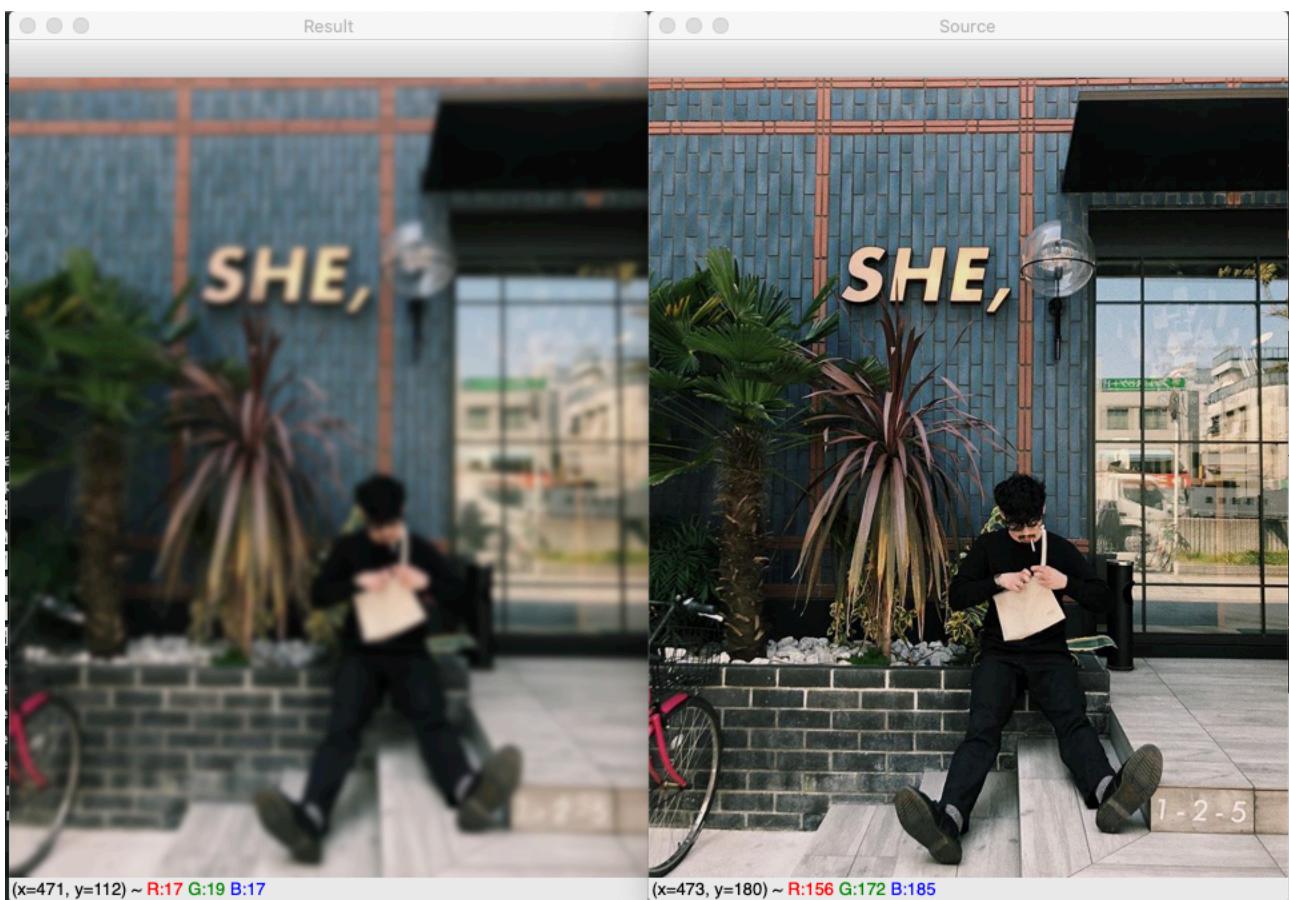
```
if __name__ == '__main__':
    main()
```

考察：

xとy方向に別のフィルタをかけている。

sepGaussian.py

実行結果：



コード：

```
import cv2
import time
from grayscale import readImage

def main():
    img = readImage()
    ksize, sigma = 15, 2.5
```

```
kernel = cv2.getGaussianKernel(ksize, sigma, cv2.CV_64F)
blur = cv2.sepFilter2D(img, cv2.CV_8UC3, kernel, kernel)
print('kernel matrix:n', kernel)
cv2.imshow('Source', img)
cv2.imshow('Result', blur)
cv2.waitKey(0)
cv2.destroyAllWindows()

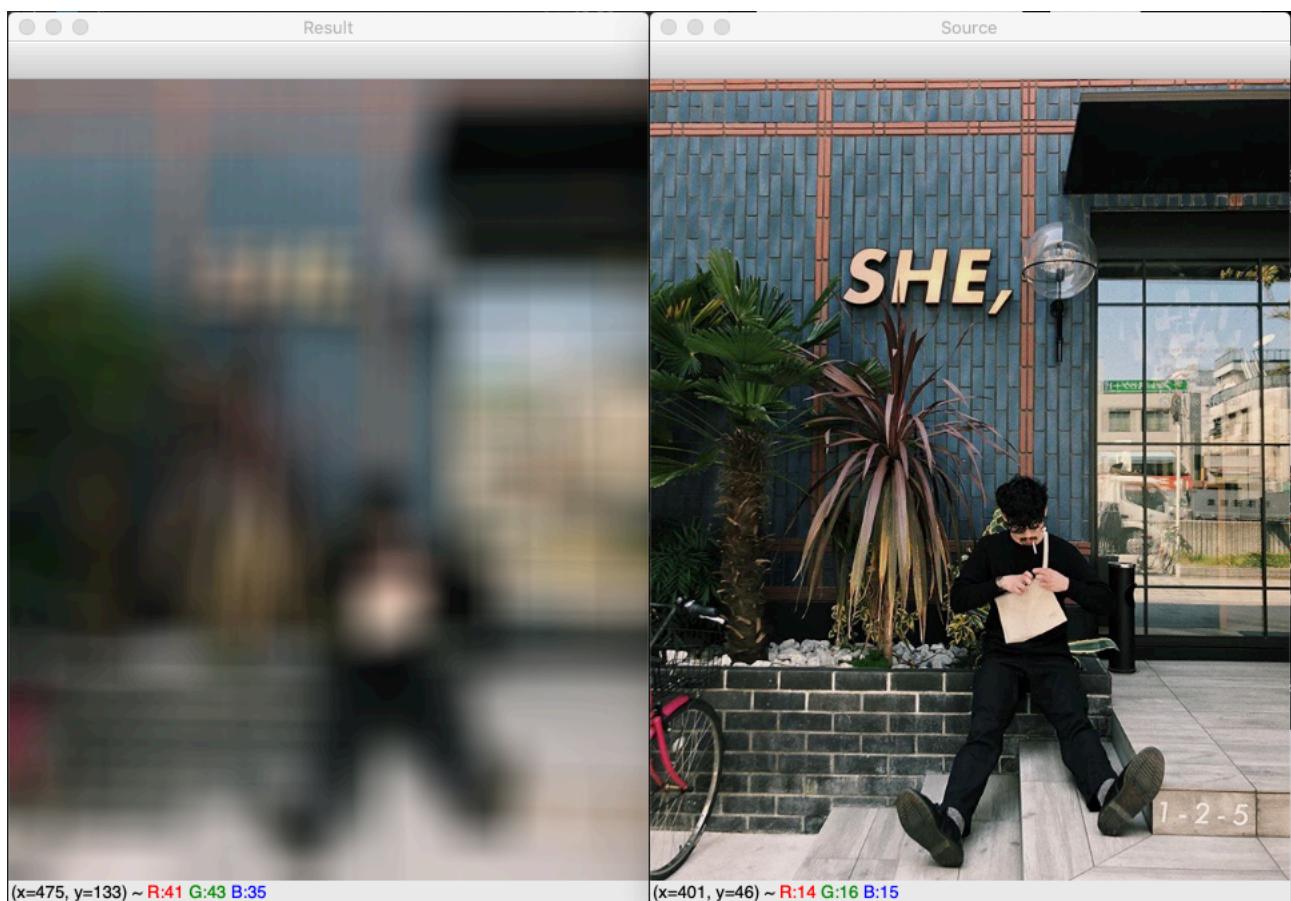
if __name__ == '__main__':
    main()
```

考察：

平滑化をすることで、境目を曖昧にし、モザイクのような表現になっている。

box.py

実行結果：



コード：

```

import cv2
import numpy as np
import time
from grayscale import readImage


def myboxFilter(img, size):
    rows, cols = img.shape[:2]
    krows, kcols = size
    ahgt = krows // 2
    awth = kcols // 2
    org = cv2.copyMakeBorder(img, ahgt, ahgt, awth, awth,
cv2.BORDER_DEFAULT)
    intimg = cv2.integral(org, cv2.CV_64F)
    res = intimg[0:rows, 0:cols] - \
        intimg[krows:krows+rows, 0:cols] - \
        intimg[0:rows, kcols:kcols+cols] + \
        intimg[krows:krows+rows, kcols:kcols+cols]
    return np.uint8(res / (krows * kcols))

def main():
    img = readImage()
    cv2.imshow('Source', img)
    kmin, kmax = 3, 41
    for kszie in range(kmin, kmax+1, 2):
        print('kernel size =', kszie, 'x', kszie)
        kernel = np.ones((kszie, kszie), np.float64) /
(kszie*kszie)
        s = time.time()
        res = cv2.filter2D(img, cv2.CV_8UC3, kernel)
        print('filter2D:{:6.2f}'.format((time.time()-s)*1000),
end='mst')
        kernel = np.ones(kszie, np.float64) / kszie
        s = time.time()
        res = cv2.filter2D(img, cv2.CV_8UC3, kernel, kernel)
        print('sepFilter:{:6.2f}'.format((time.time()-s)*1000),
end='mst')
        s = time.time()
        res = cv2.boxFilter(img, cv2.CV_8UC3, (kszie, kszie))

```

```

    print('boxFilter:{:6.2f}'.format((time.time()-s)*1000),
end='mst')
    s = time.time()
    res = myboxFilter(img, (ksize, ksize))
    print('myBoxFilter:{:6.2f}'.format((time.time()-s)*1000),
end='mst')
    print('')
    cv2.imshow('Result', res)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()

```

考察 :

平均値をかけることで境界線が曖昧になっている。

**median.py**

実行結果 :



コード :

```

import cv2
from grayscale import readImage


def main():
    img = readImage()
    cv2.imshow('Source', img)

```

```

kmin, kmax = 9, 17
for ksize in range(kmin, kmax+1, 2):
    res = cv2.GaussianBlur(img, (ksize, ksize), -1)
    cv2.imshow('Gaussian', res)
    res = cv2.boxFilter(img, cv2.CV_8UC3, (ksize, ksize))
    cv2.imshow('Box', res)
    res = cv2.medianBlur(img, ksize)
    cv2.imshow('Median', res)
    cv2.waitKey(0)
cv2.destroyAllWindows()

if __name__ == '__main__':
    main()

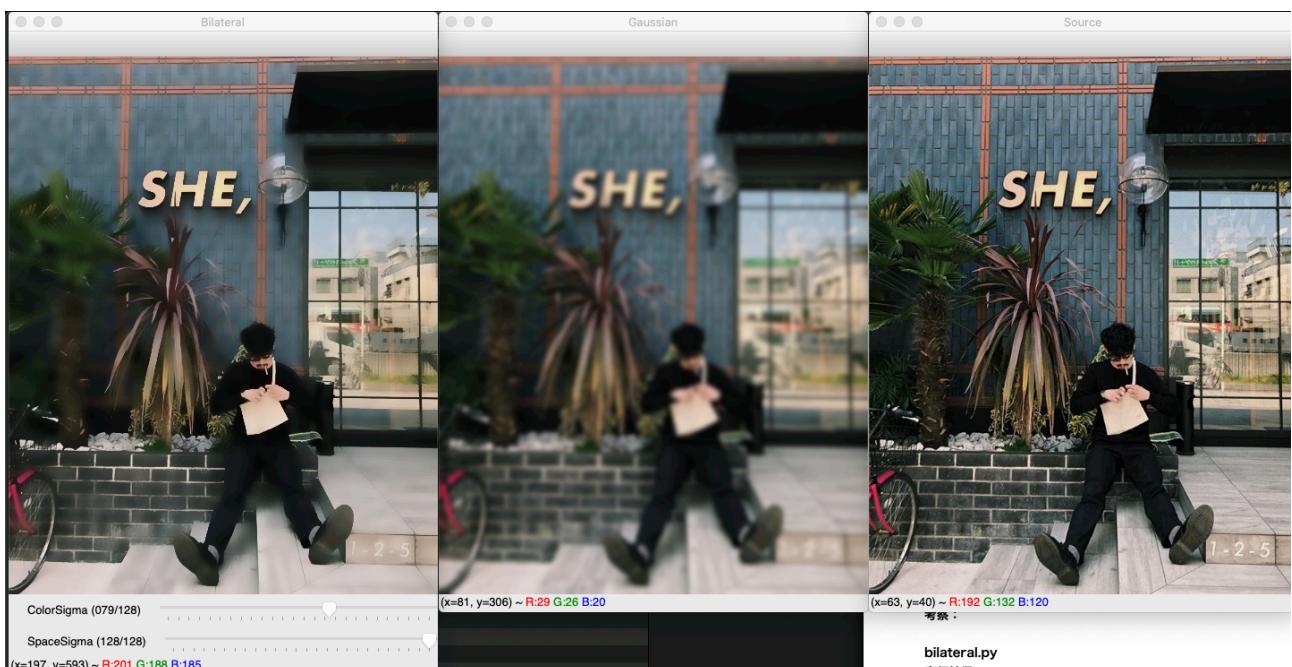
```

考察 :

ミディアンフィルターでの平滑化はノイズを除去するための平滑化なので、輪郭を保持する必要があり、結果的に絵具で塗ったような雰囲気が出ている。

bilateral.py

実行結果 :



```

cv2
from grayscale import readImage

```

```
ksize = 15
winResult = 'Bilateral'
trackColor = 'ColorSigma'
trackSpace = 'SpaceSigma'

def onChange(val):
    global img
    color = cv2.getTrackbarPos(trackColor, winResult)
    sigmaColor = 2**((color/12))
    space = cv2.getTrackbarPos(trackSpace, winResult)
    sigmaSpace = space*space/512
    result = cv2.bilateralFilter(img, ksize, sigmaColor,
sigmaSpace)
    cv2.imshow(winResult, result)

def main():
    global img
    img = readImage()
    cv2.imshow('Source', img)
    res = cv2.GaussianBlur(img, (ksize, ksize), -1)
    cv2.imshow('Gaussian', res)
    trackMax = 128
    cv2.namedWindow(winResult)
    cv2.createTrackbar(trackColor, winResult, 0, trackMax,
onChange)
    cv2.setTrackbarPos(trackColor, winResult, trackMax)
    cv2.createTrackbar(trackSpace, winResult, 0, trackMax,
onChange)
    cv2.setTrackbarPos(trackSpace, winResult, trackMax)
    onChange(0)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

**考察：**

こちらもミディアンフィルタと同様に、エッジを保持する目的の平滑化フィルタであるが、ミディアンよりもエッジを綺麗に保てている。

## 鮮鋭画像

### 実行結果



コード：

```
import cv2
import numpy as np
from grayscale import readImage


def main():
    img = readImage()
    k = 3
    tmp = cv2.Laplacian(img, cv2.CV_64F, ksize=k)
    temp = getSharpImage(img, tmp, 0.2)
    cv2.imshow('Original', img)
    cv2.imshow('Sharp', temp)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

def getSharpImage(image, laplacian, alpha):
```

```
print(laplacian)
temp = (image - alpha * laplacian) / 255.0
return temp

if __name__ == '__main__':
    main()
```

考察：

Laplacianフィルタを用いて、画像を鮮鋭化するプログラムを書いた。

輪郭をラプラシアンフィルタによって取得し、輪郭部分を強調することで、ものの間をはっきりと記述することができた。

numpyの配列として計算することで高速に計算することができた。