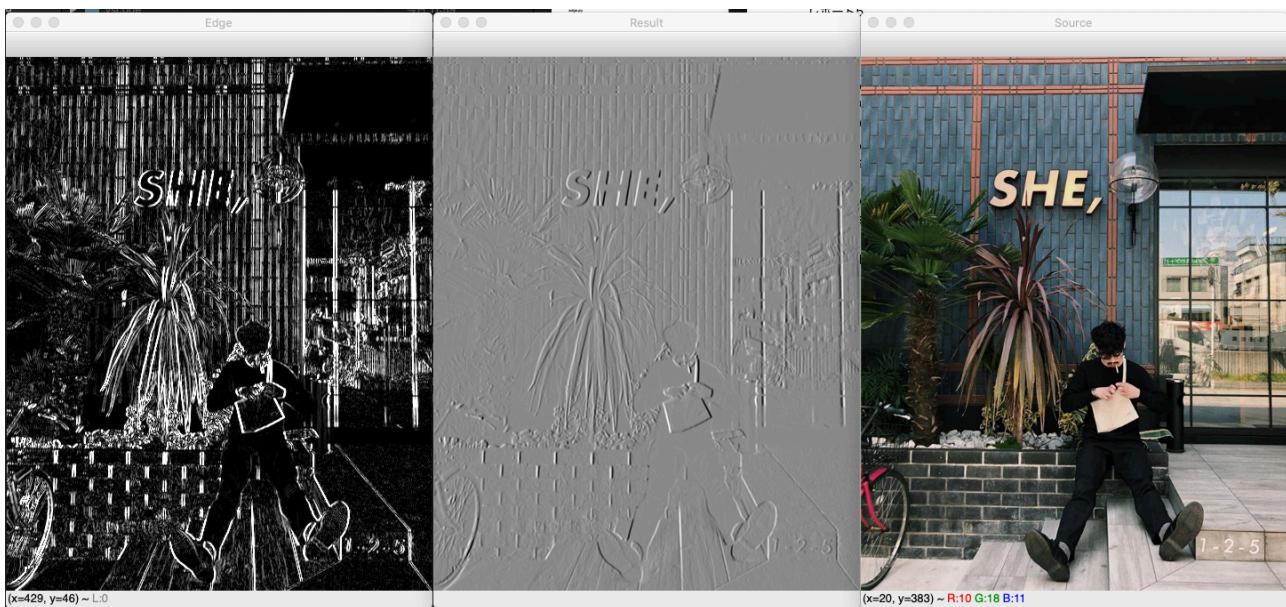


レポート5
08-172022
永田 謙

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()

```

出力

time: 7.9633448123931885 sec

考察 :

ソーベルフィルターを素朴に実装している。実行時に、4回もネストされたループを利用しているので非常に時間がかかっている。

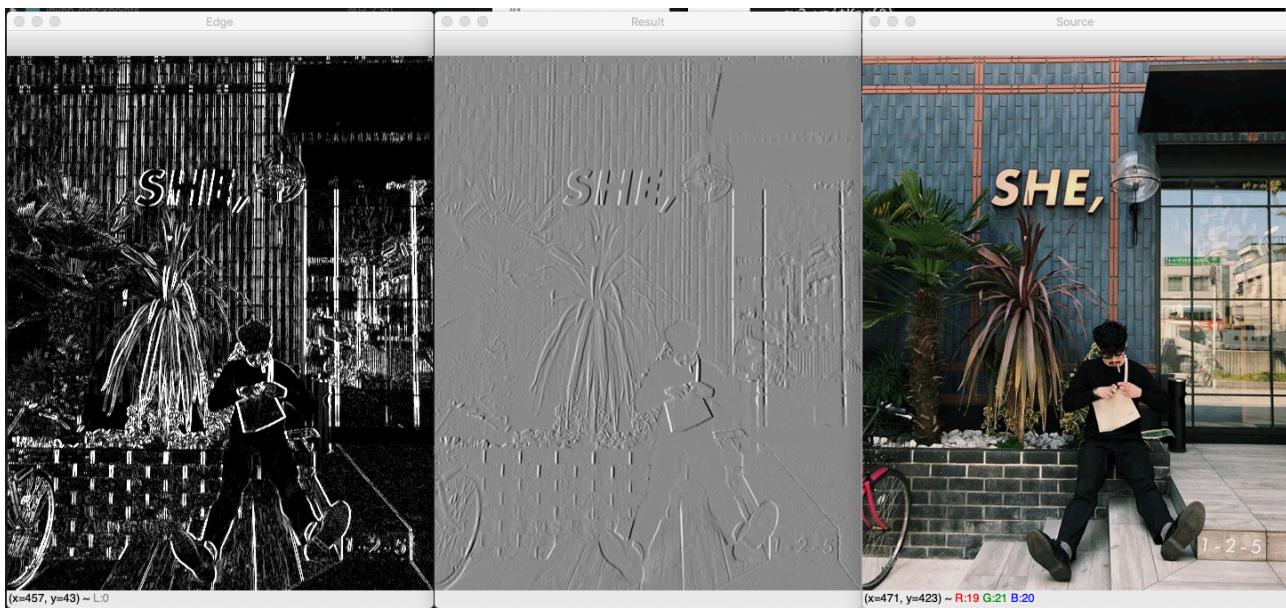
ソーベルフィルタは輪郭を抽出するために用いられる空間フィルター。

ソーベルフィルターには方向性があり、今回は水平方向のフィルターを利用している。

カーネルという変数名で定義された行列がこの方向性を決定している。

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()
```

出力

```
time: 0.00738215446472168 sec
```

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非常に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()
```

出力

```
time: 0.0012269020080566406 sec
```

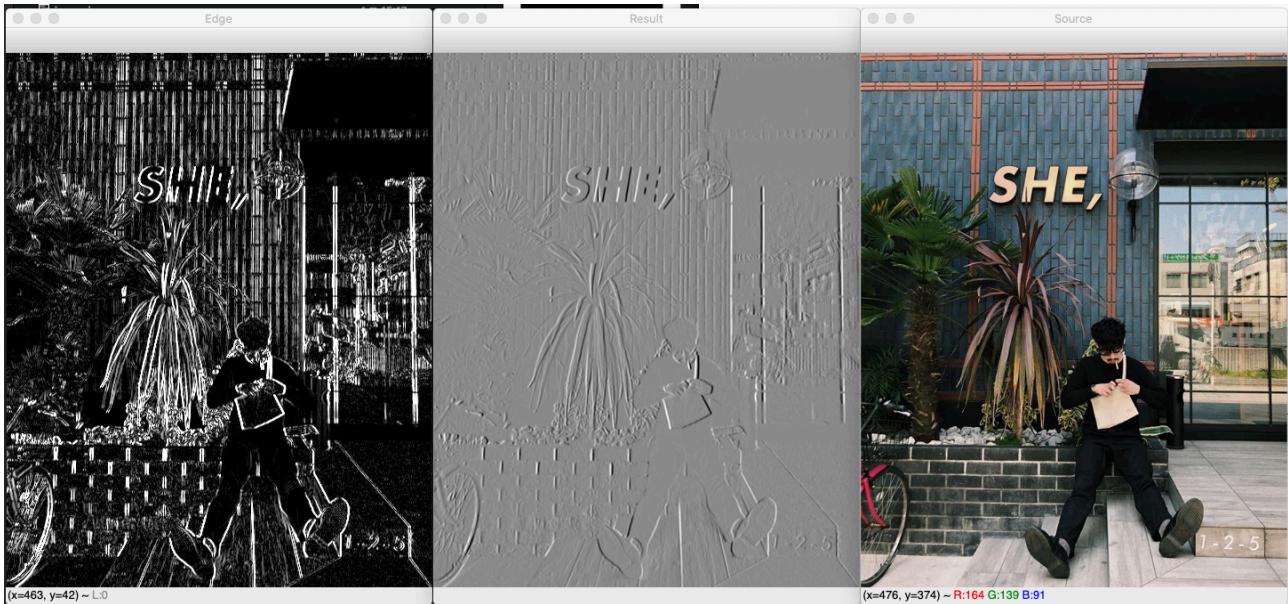
考察：

ここではOpeCVで定義済みのフィルターを使った。

Numpyの計算よりもさらに早かった。

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()
```

出力

time: 0.0013380050659179688 sec

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

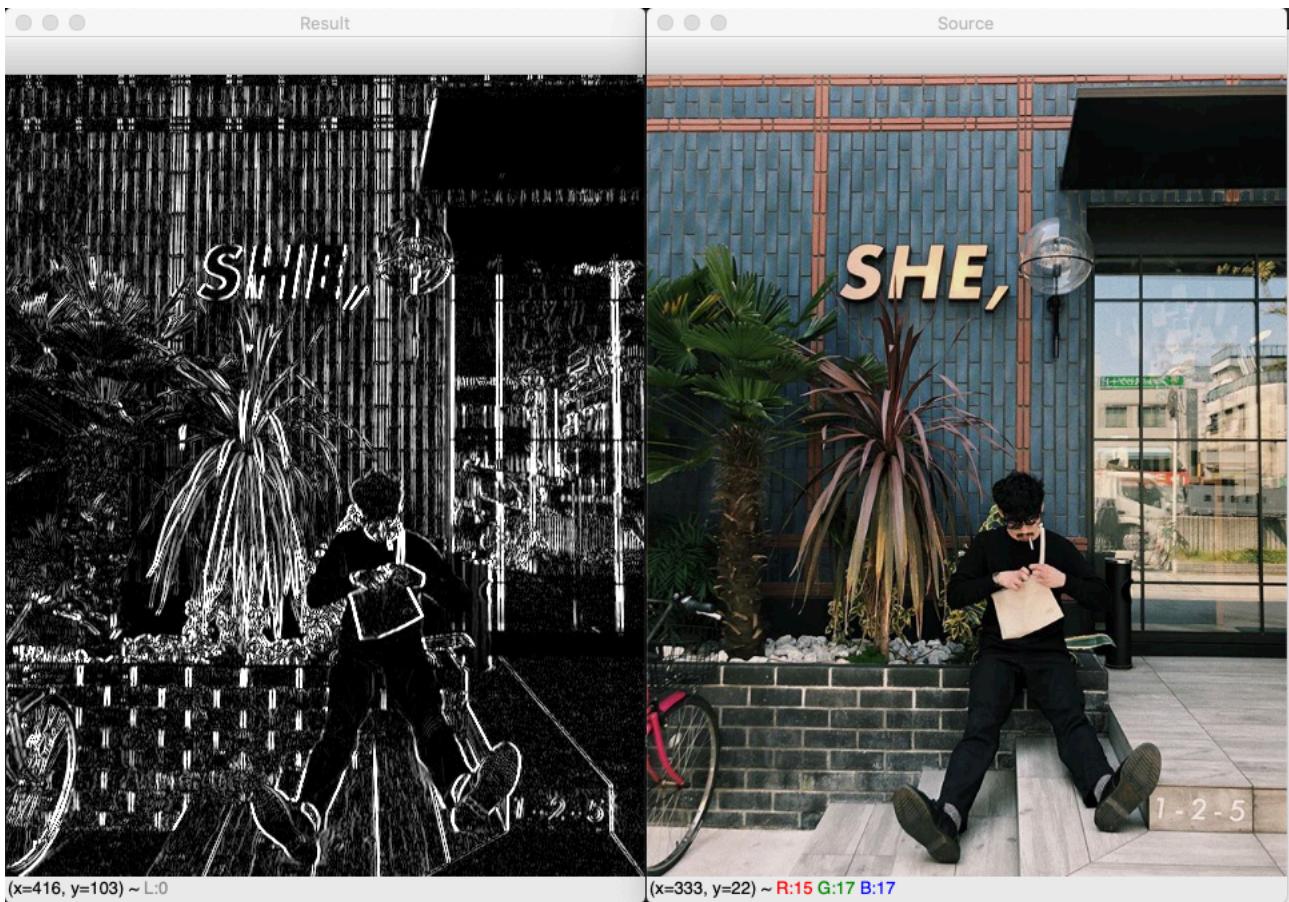
SobelY：ソーベルフィルタをY方向に適応させたもの

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()
```

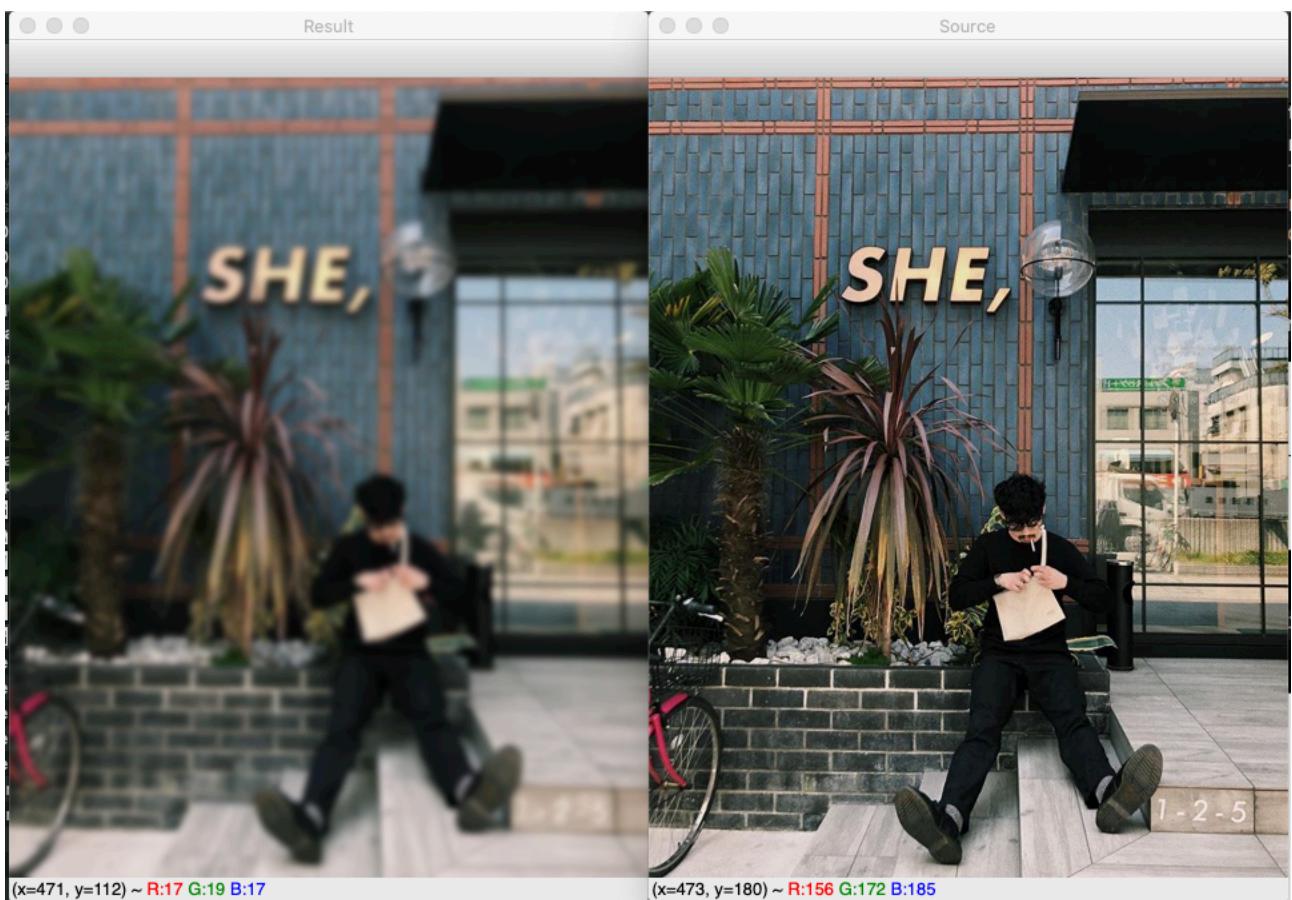
```
if __name__ == '__main__':
    main()
```

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()

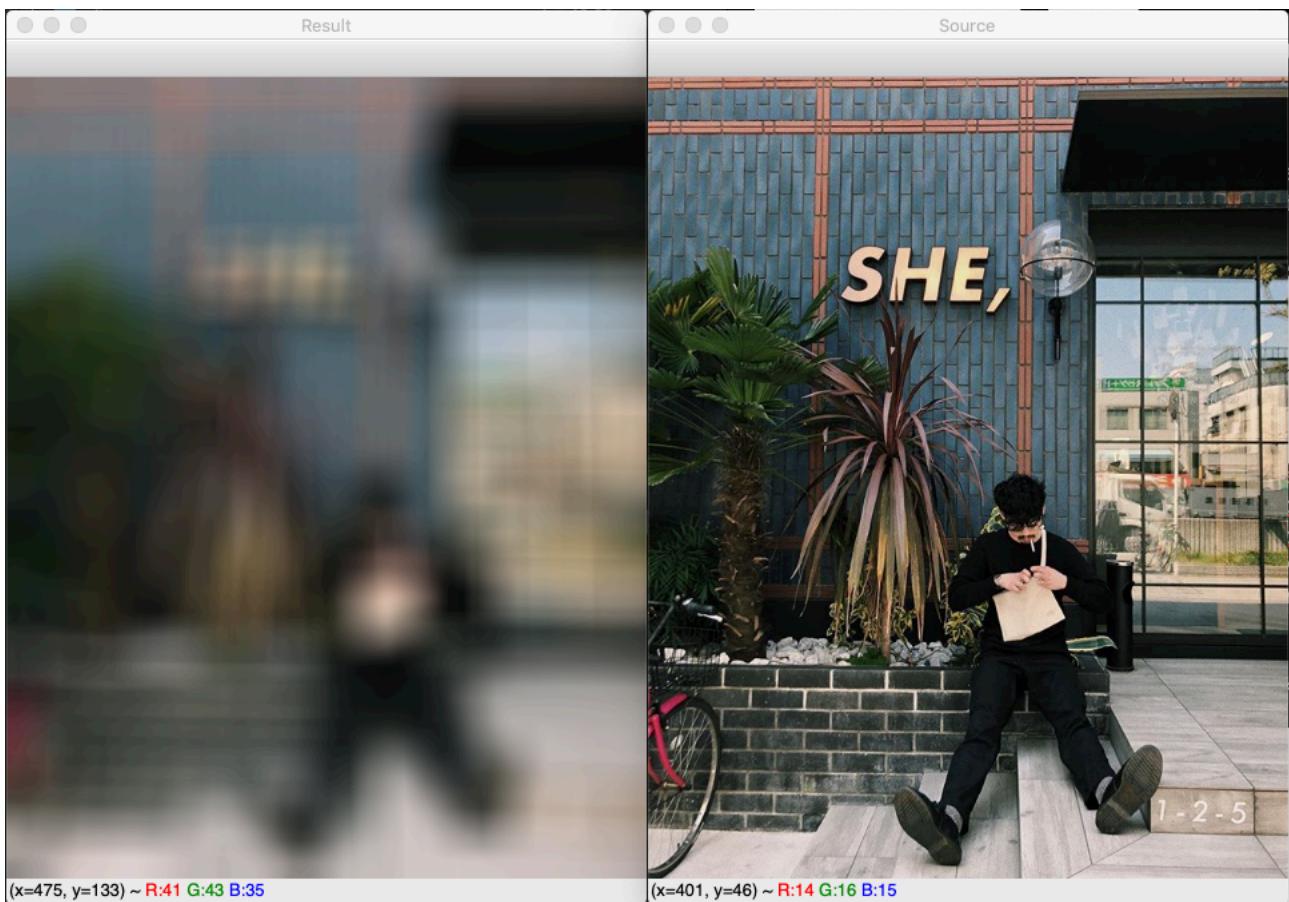
if __name__ == '__main__':
    main()
```

考察 :

ガウシアンフィルターは、自分に対して遠いピクセルの影響がより少なくなるようにガウス関数を利用したフィルタ。全体の境界が総じてぼんやりしたような印象がある。

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()
```

出力

kernel size = 3 x 3			
filter2D: 1.20ms	sepFilter: 0.65ms	boxFilter: 0.43ms	myBoxFilter: 11.13ms
kernel size = 5 x 5			
filter2D: 2.08ms	sepFilter: 0.58ms	boxFilter: 0.52ms	myBoxFilter: 6.31ms
kernel size = 7 x 7			
filter2D: 3.98ms	sepFilter: 0.75ms	boxFilter: 0.69ms	myBoxFilter: 6.31ms
kernel size = 9 x 9			
filter2D: 6.23ms	sepFilter: 0.96ms	boxFilter: 0.88ms	myBoxFilter: 8.21ms
kernel size = 11 x 11			
filter2D: 9.55ms	sepFilter: 0.96ms	boxFilter: 0.66ms	myBoxFilter: 7.20ms
kernel size = 13 x 13			
filter2D: 11.49ms	sepFilter: 1.10ms	boxFilter: 0.64ms	myBoxFilter: 6.55ms
kernel size = 15 x 15			
filter2D: 9.94ms	sepFilter: 1.25ms	boxFilter: 0.70ms	myBoxFilter: 7.52ms
kernel size = 17 x 17			
filter2D: 9.13ms	sepFilter: 1.37ms	boxFilter: 0.98ms	myBoxFilter: 6.84ms
kernel size = 19 x 19			
filter2D: 13.61ms	sepFilter: 2.02ms	boxFilter: 1.10ms	myBoxFilter: 7.98ms
kernel size = 21 x 21			
filter2D: 13.58ms	sepFilter: 1.69ms	boxFilter: 0.68ms	myBoxFilter: 6.70ms
kernel size = 23 x 23			
filter2D: 13.52ms	sepFilter: 1.85ms	boxFilter: 0.81ms	myBoxFilter: 10.28ms
kernel size = 25 x 25			
filter2D: 15.57ms	sepFilter: 2.08ms	boxFilter: 0.72ms	myBoxFilter: 6.93ms
kernel size = 27 x 27			
filter2D: 14.46ms	sepFilter: 2.36ms	boxFilter: 0.78ms	myBoxFilter: 7.52ms
kernel size = 29 x 29			
filter2D: 13.42ms	sepFilter: 2.29ms	boxFilter: 0.72ms	myBoxFilter: 6.88ms
kernel size = 31 x 31			
filter2D: 13.33ms	sepFilter: 2.79ms	boxFilter: 0.90ms	myBoxFilter: 8.10ms
kernel size = 33 x 33			
filter2D: 13.63ms	sepFilter: 2.59ms	boxFilter: 0.74ms	myBoxFilter: 6.97ms
kernel size = 35 x 35			
filter2D: 15.10ms	sepFilter: 3.63ms	boxFilter: 1.05ms	myBoxFilter: 7.86ms
kernel size = 37 x 37			
filter2D: 14.25ms	sepFilter: 3.01ms	boxFilter: 0.79ms	myBoxFilter: 6.65ms
kernel size = 39 x 39			
filter2D: 13.43ms	sepFilter: 3.07ms	boxFilter: 0.86ms	myBoxFilter: 6.54ms
kernel size = 41 x 41			
filter2D: 13.33ms	sepFilter: 3.55ms	boxFilter: 1.16ms	myBoxFilter: 6.39ms

考察：

平均値をかけることで境界線が曖昧になっている。

実行時間に関しては、sepFilterとboxFilterは非常に高速だった。

そのほか、カーネルサイズが大きくなるにつれ、処理時間は長くなる傾向があることがわかった。

積分画像を用いたmyBoxFilterと、filter2Dを比較すると、積分画像を生成する時間分だけ、カーネルサイズが小さいうちはmyBoxFilterを用いた方が実行速度は劣るが、カーネルサイズが大きくなると、myBoxFilterの方が早く処理を終えている。

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()

```

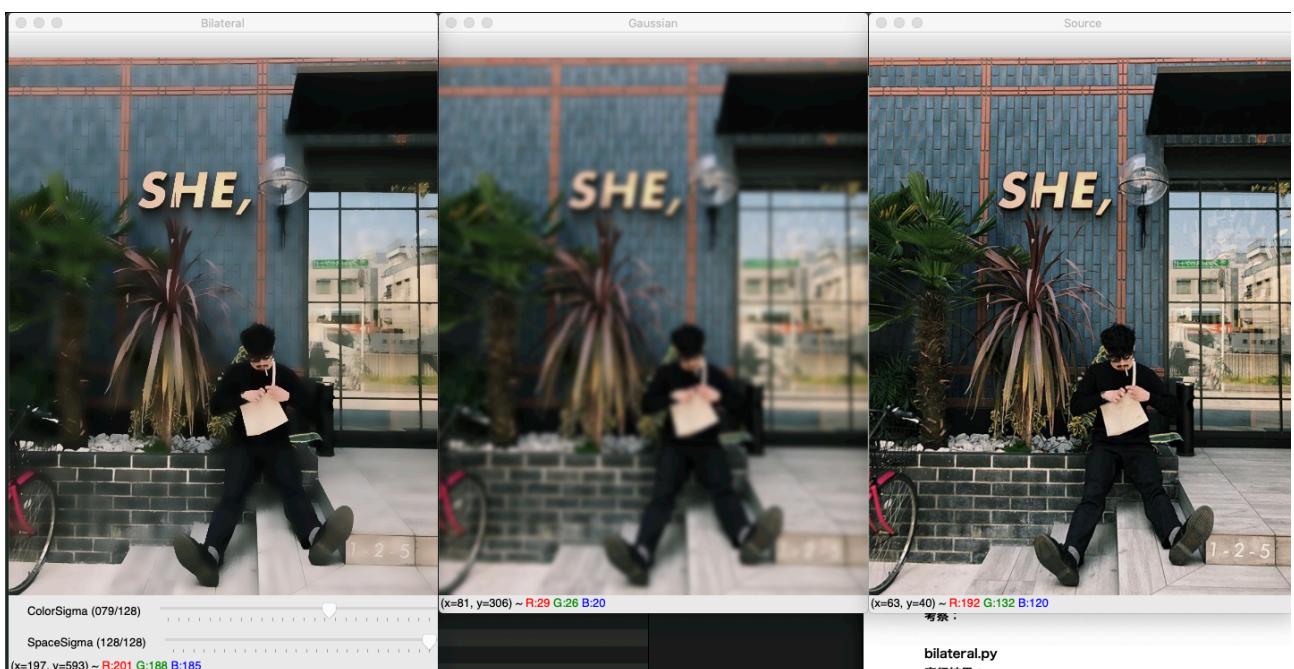
考察 :

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。

メディアンフィルタでは、ある画素を周りの画素の中央値にすることでエッジを保ったままの平滑化を実現している。つまり、あまりにも小さなピクセル単位のノイズは消えてしまうということである。写真を見れば分かるとおり、SHEという文字は変形してしまっている。

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()
```

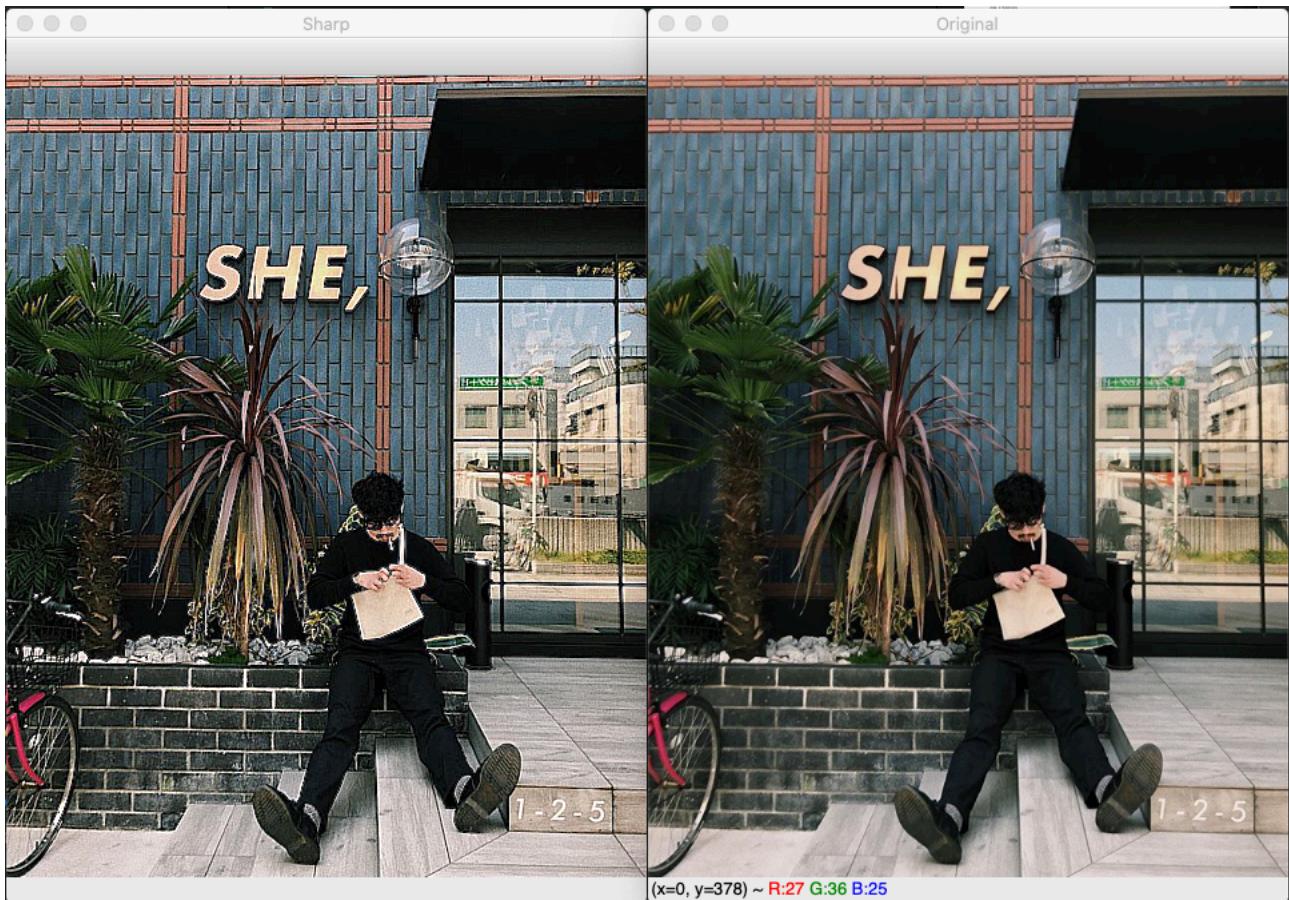
考察 :

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。

バイラテラルフィルタは、正規分布の重みがついたガウシアンフィルターと考えることができ、画素値の値が大きく違うピクセルの影響を小さくするように設計されている。メディアンでは、自分の値との比較がなかったが、バイラテラルフィルタでは自分の値を考慮するため、より良い精度でノイズの除去が行えている。

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。