

レポート4
08-172022
永田 謙

scale.py

実行結果



コード

```
# todo変更

import cv2
from grayscale import readImage


def main():
    img = readImage()
    cv2.imshow('Original', img)
    scaleX = 2
    scaleY = 3
    rows, cols = img.shape[:2]
    size = (cols * scaleX, rows * scaleY)

    result = cv2.resize(img, size, interpolation=cv2.INTER_NEAREST)
    cv2.imshow('NEAREST', result)
    result = cv2.resize(img, size, interpolation=cv2.INTER_LINEAR)
    cv2.imshow('LINEAR', result)
    result = cv2.resize(img, size, interpolation=cv2.INTER_CUBIC)
    cv2.imshow('CUBIC', result)

    cv2.waitKey(0)
```

```
cv2.destroyAllWindows()

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

考察：

実行結果は左から、Cubic, Linear, Nearestである

特徴が明確なのはNearestで、ピクセルが強調され、ガタガタに見える。これにはニアレストネイバー方が用いられている。一般的に、回転後の画像のある画素に対応する、回転前の画像の画素の座標は、整数値を取らず実数値を取る。よって最近傍の画素を選択し、この濃度値を回転後の画像の対応画素に書き込む方法が、ニアレストネイバー法である。ただし、90°、180°、270°のような90°毎以外の回転では、±0.5画素未満の量子化誤差（実数を整数とみなしたことによる）を伴うため、エッジにはジャギー（ギザギザ）が発生する。

LINEARが行うのはバイリニア補間である。ニアレストネイバー法では最近傍の画素を選択したが、バイリニア補間では変更前と変更後で、座標の近い4個の画素の平均値を採用する。ニアレスト補間よりも計算処理は重いが、画質の劣化を抑えることが出来る。

Cubicが行うのは4x4の近傍領域を利用するバイキュービック補間である。変更前と変更後で、座標に近い16個の画素を利用し、距離に応じて三次関数を使い分け、加重平均を求める。補間に用いる式は、 $\sin(\pi x) / \pi x$ で、理論（サンプリング定理）的には最も完全な濃度補間式である。これを泰イラーフィルタで3次の項で近似し、補間式として用いる。バイリニア補間法よりも計算処理は重いが、もっとも自然な結果が得られる。

affine.py

実行結果



コード

```
1. import cv2
2. import numpy as np
3. from grayscale import readImage
4.
5.
6. def main():
7.     img = readImage()
8.     cv2.imshow('Original', img)
9.     rows, cols = img.shape[:2]
10.    size = (cols, rows)
11.    original = np.float32((0, 0), (cols, 0), (0, rows)))
12.    translate = np.float32((100, 150), (500, 50), (50, 550)))
13.    mat = cv2.getAffineTransform(original, translate)
```

```
14.     result = cv2.warpAffine(img, mat, size,
15.         flags=cv2.INTER_CUBIC)
16.     cv2.imshow('Result', result)
17.     cv2.waitKey(0)
18.
19.
20.if __name__ == '__main__':
21.     main()
```

考察：

情報工学Vでもやったaffine変換である。

10行目で指定しているサイズは、画像のピクセルサイズのことである。

11行目では、オリジナルの画像の3つの頂点を指定している。

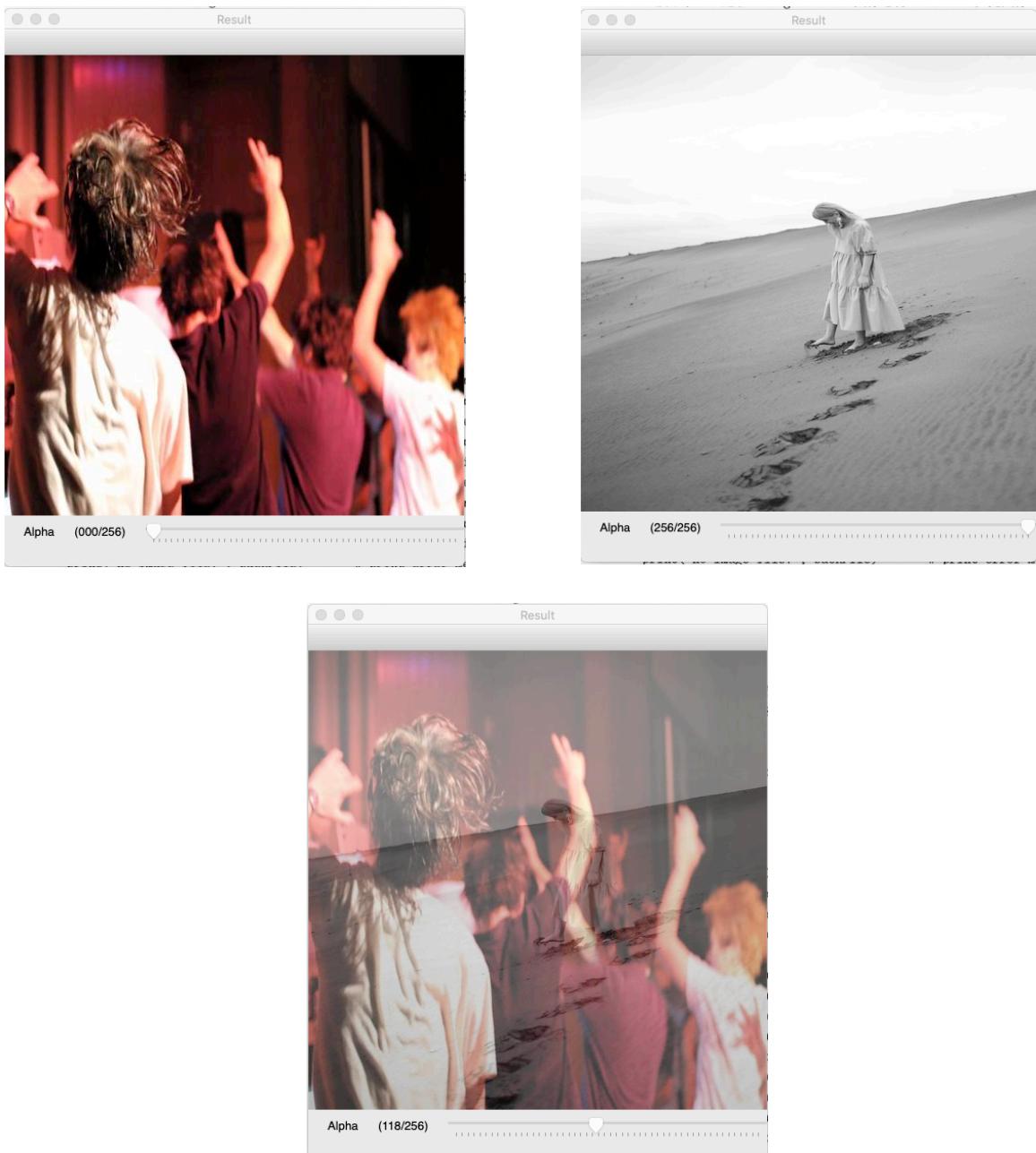
12行目では、アフィン変換後の3つの頂点をしている。

13行目では、openCVに定義済みの関数を使って、アフィン変換用の行列を計算している。

14行目では、13行目で得られた行列を使ってアフィン変換後の画像配列を取得している。

dissolve.py

実行結果



コード：

```
import sys
import cv2
import numpy as np

winResult = 'Result'
trackbar = 'Alpha'
```

```
trackHalf = 128
trackMax = trackHalf * 2

def readImages():
    if len(sys.argv) > 2:
        foreFile = sys.argv[1]
        backFile = sys.argv[2]
    else:
        foreFile = input('foreground image -> ')
        backFile = input('background image-> ')
    fore = cv2.imread(foreFile)
    if fore is None:
        print('no image file:', foreFile)
        sys.exit(1)

    back = cv2.imread(backFile)
    if back is None:
        print('no image file:', backFile)
        sys.exit(1)

    if fore.shape[:2] != back.shape[:2]:
        print('different image sizes: ', foreFile, backFile)
        sys.exit(1)

    return (fore, back)

def onChange(val):
    global fore, back
    alpha = val / trackMax
    result = (fore.astype(np.float64) / 255) * (1-alpha) + \
             (back.astype(np.float64) / 255) * alpha

    cv2.imshow(winResult, result)

def main():
    global fore, back
    fore, back = readImages()
```

```
cv2.namedWindow(winResult)
cv2.createTrackbar(trackbar, winResult, 0, trackMax, onChange)
cv2.setTrackbarPos(trackbar, winResult, trackHalf)
onChange(trackHalf)
cv2.waitKey(0)
cv2.destroyAllWindows()

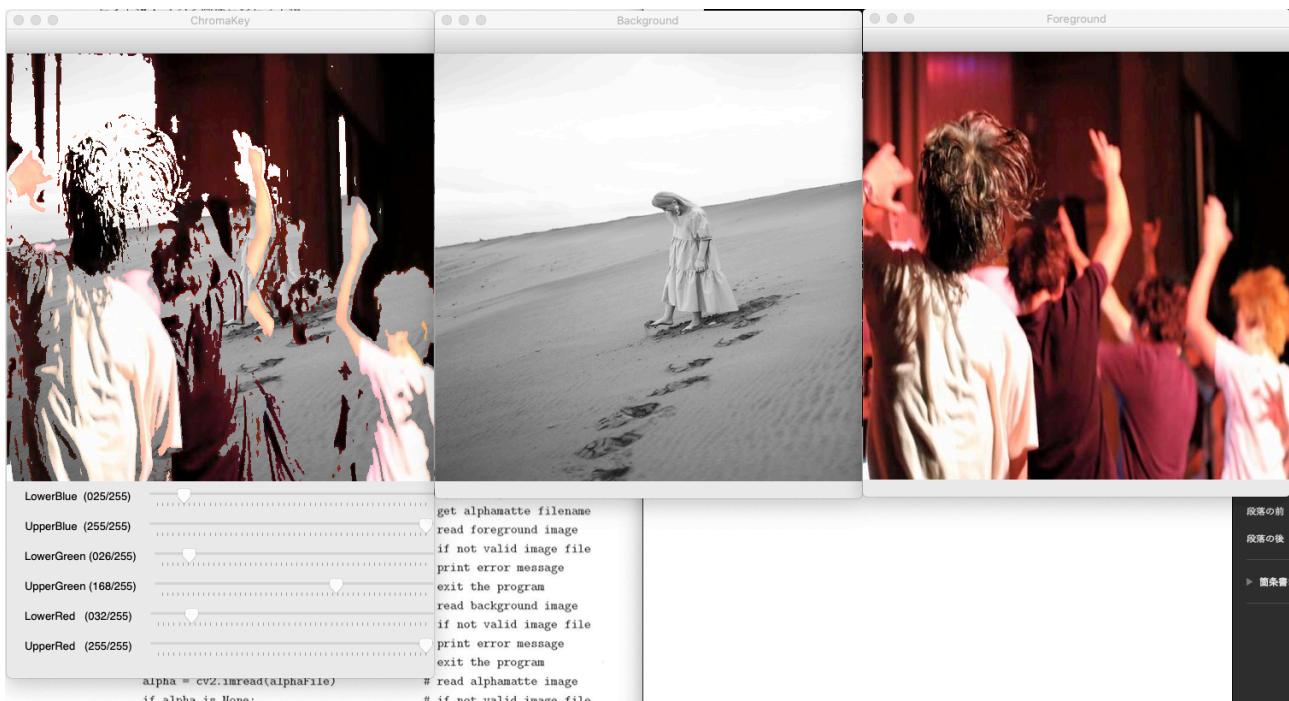
if __name__ == '__main__':
    main()
```

考察：

二つの画像の合計で、不透明度が1になるようになっている。numpyでの計算により非常に簡潔に画像の和を表現できている。

chromaKey.py

実行結果



コード：

```
import cv2
import numpy as np
from dissolve import readImages

winChromaKey = 'ChromaKey'
trackbars = ('LowerBlue', 'UpperBlue', 'LowerGreen',
            'UpperGreen', 'LowerRed', 'UpperRed')
trackMax = 255

def onChange(val):
    global fore, back
    lower, upper = [], []
    for i in range(len(trackbars)):
        if i % 2 == 0:
            lower.append(cv2.getTrackbarPos(
                trackbars[i], winChromaKey) / trackMax)
        else:
            upper.append(cv2.getTrackbarPos(
```

```

        trackbars[i], winChromaKey) / trackMax)

posMask = cv2.inRange(fore, np.array(lower), np.array(upper))
/ trackMax
posMask = cv2.merge((posMask, posMask, posMask))
negMask = np.ones(posMask.shape, np.float64) - posMask

result = fore * negMask + back * posMask
cv2.imshow(winChromaKey, result)

def main():
    global fore, back
    fore, back = readImages()
    fore = fore.astype(np.float64) / trackMax
    cv2.imshow('Foreground', fore)
    back = back.astype(np.float64) / trackMax
    cv2.imshow('Background', back)
    cv2.namedWindow(winChromaKey)

    for i in range(len(trackbars)):
        cv2.createTrackbar(trackbars[i], winChromaKey, 0,
trackMax, onChange)
        cv2.setTrackbarPos(trackbars[i], winChromaKey, (i % 2) *
trackMax)

    onChange(0)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()

```

考察：

よく緑色の背景で行われるクロマキー撮影の実装である。

```

for i in range(len(trackbars)):
    if i % 2 == 0:
        lower.append(cv2.getTrackbarPos(

```

```
    trackbars[i], winChromaKey) / trackMax)
else:
    upper.append(cv2.getTrackbarPos(
        trackbars[i], winChromaKey) / trackMax)

posMask = cv2.inRange(fore, np.array(lower), np.array(upper))
/ trackMax
posMask = cv2.merge((posMask, posMask, posMask))
negMask = np.ones(posMask.shape, np.float64) - posMask

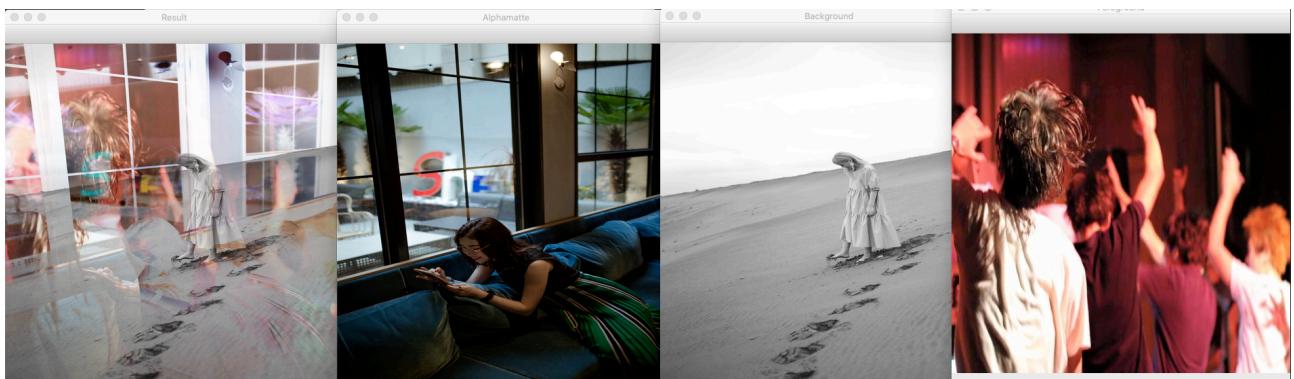
result = fore * negMask + back * posMask
cv2.imshow(winChromaKey, result)
```

ここで行なっているのは

閾値を設定し、その閾値との関係によって、binaryで表されるピクセル毎のマスクを作成しており、1ならば後ろの画像を0ならば前の画像をと言う形で簡単な行列計算に落とし込んでいる。単純で面白い。

alphamatte.py

実行結果



コード

```
import sys
import cv2
import numpy as np

def main():
    if len(sys.argv) > 3:
        foreFile = sys.argv[1]
        backFile = sys.argv[2]
        alphaFile = sys.argv[3]
    else:
        foreFile = input('foreground image -> ')
        backFile = input('background image -> ')
        alphaFile = input('alphamatte -> ')

    fore = cv2.imread(foreFile)
    if fore is None:
        print('no image file:', foreFile)
        sys.exit(1)

    back = cv2.imread(backFile)
    if back is None:
        print('no image file:', backFile)
        sys.exit(1)

    alpha = cv2.imread(alphaFile)
```

```
if alpha is None:
    print('no image file:', alphaFile)
    sys.exit(1)

if fore.shape[:2] != back.shape[:2]:
    print('different image sizes:', foreFile, backFile)
    sys.exit()

if fore.shape[:2] != alpha.shape[:2]:
    print('different image sizes:', foreFile, alphaFile)
    sys.exit()

fore = fore.astype(np.float64) / 255.0
cv2.imshow('Foreground', fore)
back = back.astype(np.float64) / 255.0
cv2.imshow('Background', back)
alpha = alpha.astype(np.float64) / 255.0
cv2.imshow('Alphamatte', alpha)
result = fore * alpha + back * (1.0 - alpha)

cv2.imshow("Result", result)
cv2.waitKey(0)
cv2.destroyAllWindows()

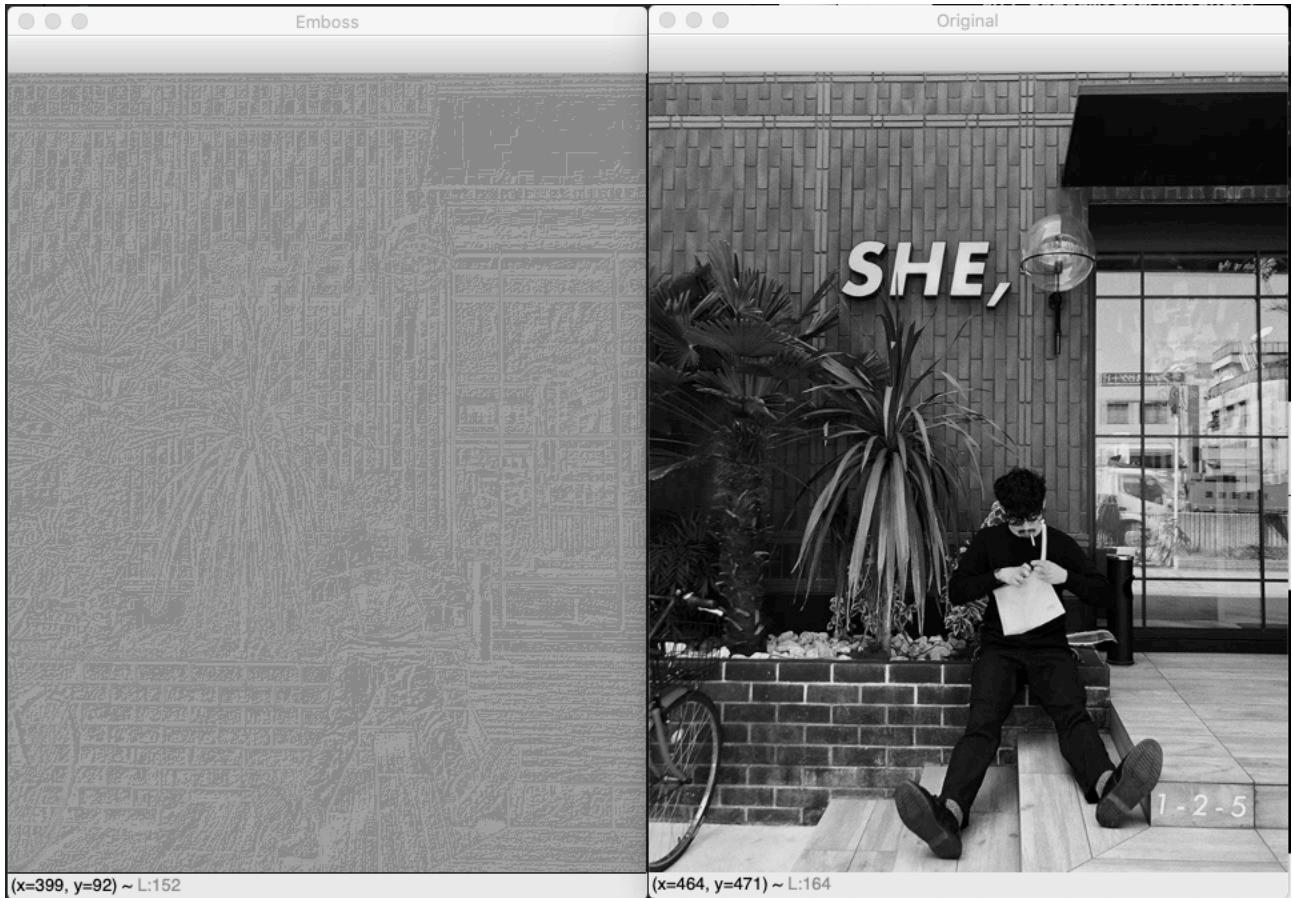
if __name__ == '__main__':
    main()
```

考察：

alpha値は固定であったが、今回はalpha値に画像を用いることで、ピクセルごとに異なる値のalpha値を指定し、結果的にぼんやりとalpha値の算出に用いた画像が見えるようになっている。

emboss.py

実行結果



コード：

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

def main():
    img = readImage()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    emboss = getEmboss(gray, 0.1, 128)
    cv2.imshow('Original', gray)
    cv2.imshow('Emboss', emboss)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

def getEmboss(image, alpha, staticColor):
```

```
....  
imageは二次元配列  
....  
  
res = np.zeros(image.shape, np.uint8)  
res[len(image)-1, len(image[0])-1] = staticColor  
for row in range(len(image)-1):  
    for col in range(len(image[row])-1):  
        temp = alpha * (image[row, col] -  
                         image[row + 1, col + 1]) + staticColor  
        if temp <= 0:  
            res[row, col] = 0  
        elif 255 <= temp:  
            res[row, col] = 255  
        else:  
            res[row, col] = int(temp)  
return res  
  
  
if __name__ == '__main__':  
    main()
```

考察：

今回の問題の意図からすれば、画像を並行に移動して、色を引くという方法をとることを求められていたと思うが、numpyの内部実装がいまいちわからなかつたので、直接行列をいじってみた。全てのピクセルを素朴に計算することで、比較対象のピクセルとの色差を抽出し、ものの境目を強調した。並列処理などを行えていないのでfor文で回すのではなく、numpyの関数を使えばよりパフォーマンスを上げることができるはずである。