Assignment 1 Report

Heinrich Dinkel
ID: 1140339107
E-mail: heinrich.dinkel@sjtu.edu.cn

1 Assignment 2

In this exercise, the task is to write a convolution and use the laplacian operator to calculate the second derivative of the given image.

The Laplacian, which is used is:

$$\mathcal{L} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 * A & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

As it can be seen, the Laplacian sums up to 0, which is one property of the derivative for images in two dimensions. We did overall 3 experiments, which used the following parameters:

$$A \in \{0.7, 1.0, 1.0\}$$

The resulting Pictures can be seen in "Assignment2"

2 Assignment 3

In this exercise the task is to write different filters, for blurring purposes.

The program can be executed by using the following syntax:

```
python __init__.py [inputfile] -k gaussian -sig 10 -o output
```

It offers the following parameters:

- -o Outputs the processed file into the given path. Note that no directories will be generated.
- -s sets the size for the shape of the mask which will be used. The used mask will always be square
- -k sets the kernel type, 'gaussian','butterworth','ideal' are the only parameters, default is gaussian
- -sig the sigma / cutoff parameter for the gaussian/ideal/butterworth filters

The outputfile which is going to be produced will be named as given in "-o".

The output files have the following naming schema:

```
KERNELNAME_sigma_SIGMAVALUE.tif
```

Whereas "Kernelname" stands for the currently used kernel and "SIGMAVALUE" stands for the currently used Sigma value, given by -k. The most significant factor is the parameter sigma, whereas large sigmas lead to a heavy blurring and small sigmas to the contrary.

3 Assignment 4

In this exercise we implemented multiple filter types, which try to filter noise.

Noise was applied on the input image,whereas two types of noise was used: uniformed and gaussian.

In python it was easily implemented:

```python
picture += np.random.uniform(low,high,size=picture.shape)
picture += np.random.normal(mean,var,picture.shape)
```

As an example, we show here the implementation of the harmonic filter:

```python
def harmonicMeanFilter(arr,shape=(3,3)):
    x,y = shape
    xoff = x/2
    yoff = y/2
    transformedimg = np.copy(arr)
    for i in range(xoff,len(arr)-xoff):
        for j in range(yoff,len(arr[0])-yoff):
            average = 0
            for p in range(x):
                for q in range(y):
                    average += 1./arr[i+(p-xoff)][j+(q-yoff)]
            if average<0:
                average = 0
            transformedimg[i][j] = (x*y)/average
    return transformedimg
```

The resulting images can be seen in the directory Assignment 4.

4 Assignment 5

The implementation of the wiener filter was troublesome. First of all, we should say that when applying the blurring filter on an image, one needs to consider the limits of the filter function should be used in the case of $u, v = 0$. In our case we used the following kernel:

```python
def blurringkernel(shape, T, a, b):
    xx, yy = shape
    x, y = np.ogrid[(-xx / 2):(xx / 2), (-yy / 2):yy / 2]
    q = (np.pi * (x * a + y * b))
    q[np.where(q == 0)] = T
    return (T / q) * np.sin(q) * np.exp(-1j * q)
```

Moreover when using the inverse filter, one should set the offset values, if u, v are once again 0, to a fixed small ϵ. The implementation of the wienerkernel was done using only fixed parameters for the SNR and the power spectrum.

```python
def wienerkernel(origimg, blurredkernel, param, k):
    '''
    Calculates the wiener kernel, which is H*/(H^2+param*(S_n/S_f))
    '''
    h = blurredkernel
    return np.conj(h).T / ((abs(h) ** 2) + (param * k))
```

5 Assignment 6

Using the bilinear and nearest neighbour interpolation methods, we achieve different results for the given operations. Generally it can be seen that bilinear interpolation does perform for the given image slightly better, especially when using rotation. When using translation and scaling, the differences between the two methods are not easily seen. The implementations for scaling and translating are both straightforward and result for both methods in a roughly equal picture.

Bilinear interpolation is given as:

$$\begin{aligned} f(x,y) \approx\ & \frac{f(Q_{11})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y_2-y)+ \\ & \frac{f(Q_{21})}{(x_2-x_1)(y_2-y_1)}(x-x_1)(y_2-y)+ \\ & \frac{f(Q_{12})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y-y_1)+ \\ & \frac{f(Q_{22})}{(x_2-x_1)(y_2-y_1)}(x-x_1)(y-y_1) \\ =\ & \frac{1}{(x_2-x_1)(y_2-y_1)}\Big(f(Q_{11})(x_2-x)(y_2-y)+ \\ & \qquad f(Q_{21})(x-x_1)(y_2-y)+ \\ & \qquad f(Q_{12})(x_2-x)(y-y_1)+ \\ & \qquad f(Q_{22})(x-x_1)(y-y_1) \quad \Big) \end{aligned}$$

The rotation was done using the following function:

```
def rotate(x, y, theta, ox, oy):
    """Rotate arrays of coordinates x and y by theta radians about the
    point (ox, oy).
    """
    s, c = np.sin(theta), np.cos(theta)
    x, y = np.asarray(x) - ox, np.asarray(y) - oy
    return x * c - y * s + ox, x * s + y * c + oy
```

Nearest neightbor interpolation is straight forward:

```
def _interpolate(self, sx, sy):
    return np.round(sx).astype(int), np.round(sy).astype(int)
```

Bilinear interpolation is somewhat more complex, the formula used to calculate a new point is:

```
def _interpolate(self, im, x, y):
        x = np.asarray(x)
        y = np.asarray(y)

        x0 = np.floor(x).astype(int)
        x1 = x0 + 1
        y0 = np.floor(y).astype(int)
        y1 = y0 + 1
#           Clip the x0 and x1, so that we dont have to check later if they are in range of the img
        x0 = np.clip(x0, 0, im.shape[1] - 1);
        x1 = np.clip(x1, 0, im.shape[1] - 1);
        y0 = np.clip(y0, 0, im.shape[0] - 1);
        y1 = np.clip(y1, 0, im.shape[0] - 1);
        Ia = im[ y0, x0 ]
        Ib = im[ y1, x0 ]
        Ic = im[ y0, x1 ]
        Id = im[ y1, x1 ]
        wa = (x1 - x) * (y1 - y)
        wb = (x1 - x) * (y - y0)
        wc = (x - x0) * (y1 - y)
        wd = (x - x0) * (y - y0)

        return (wa * Ia + wb * Ib + wc * Ic + wd * Id)
```

6 Assignment 7

The image compression was achieved using four different kernels:

```
zonal = np.array([
    [1, 1, 1, 1, 1, 0, 0, 0],
    [1, 1, 1, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
])
zonal_best = np.array([
    [1, 1, 1, 1, 1, 1, 1, 0],
    [1, 1, 1, 1, 1, 1, 0, 0],
    [1, 1, 1, 1, 1, 0, 0, 0],
    [1, 1, 1, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
])
thresholdmask = np.array([
    [1, 1, 0, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
])
jpegstd = np.array([[16, 11, 10, 16, 24, 40, 51, 61],
                    [12, 12, 14, 19, 26, 58, 60, 55],
                    [14, 13, 16, 24, 40, 57, 69, 56],
                    [14, 17, 22, 29, 51, 87, 80, 62],
                    [18, 22, 37, 56, 68, 109, 103, 77],
                    [24, 35, 55, 64, 81, 104, 113, 92],
                    [49, 64, 78, 87, 103, 121, 120, 101],
                    [72, 92, 95, 98, 112, 100, 103, 99]
                    ])
```

7 Assignment 8

8 Assignment 9

9 Assignment 10