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:

```
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:

```
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:

```
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.

```
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:

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

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

6.1 Standard compression

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]
                    ])
```

Each of the kernels can be used within the program, using a different parameter given in the parameter "-quantmattype". The results for every of these masks can be seen in the Assignment7 directory.

6.2 Wavelets

The wavelets were approached in a totally different scheme than the proposed one by the book. We used purely matrix operations to achieve our goal.

A wavelet transform of an image A can be displayed by:

$$B = H^T A H$$

where H is the wavelet transform matrix and B is the resulting encoded image.

Constructing H is a little bit complicated. Generally we construct H as having two components. The upper component, which values all sum up to 2 and the lower component, which sums up to 0. Therefore we split the matrix H from the beginning on in two halves and then insert the values for the corresponding wavelet. In the case of Haar wavelets, we used the Kronecker Product:

$$I \otimes h_0 = \begin{pmatrix} \frac{1}{\sqrt{(2)}} & \frac{1}{\sqrt{(2)}} & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & \frac{1}{\sqrt{(2)}} & \frac{1}{\sqrt{(2)}} & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{(2)}} & \frac{1}{\sqrt{(2)}} \end{pmatrix}$$

$$I \otimes h_1 = \begin{pmatrix} \frac{1}{\sqrt{(2)}} & -\frac{1}{\sqrt{(2)}} & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & \frac{1}{\sqrt{(2)}} & -\frac{1}{\sqrt{(2)}} & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{(2)}} & -\frac{1}{\sqrt{(2)}} \end{pmatrix}$$

Afterwards we simply concatenate both halves together (stack them on each other, row wise) and obtain the transformation matrix H.

Calculating the reconstruction is a simple task due to this operation:

$$A = HBH^T$$

A special notice should be given for the larger wavelets. Here Kronecker product does not work, since it would lead to being H of larger dimension than A, which therefore would increase the picture size and fill it with mostly zeros.

For the case that the wavelets having a size larger than 2, we do basically the same procedure like for the Haar wavelets, we insert in each upper half the wavelet h_0 and in the lower h_1. We use hereby a step size of 2, so we get for example the following matrix (a, b, c, d are just place holders for an e.g. daubechies 4-tap, the principle is the same for the 8-tap etc.)

$$H_{up} = \begin{pmatrix} a & b & c & d & 0 & \dots & & \\ 0 & 0 & a & b & c & d & \dots & 0 \\ 0 & 0 & 0 & 0 & \ddots & \ddots & \ddots & 0 \\ d & 0 & 0 & 0 & \dots & a & b & c \\ c & d & 0 & 0 & \dots & 0 & a & b \\ b & c & d & 0 & \dots & 0 & 0 & a \end{pmatrix}$$

As it can be seen the array "rolls" within the matrix. Using this we can easily compute the matrix H for the daubechies, symlet and cohen wavelet.

The calculations for the encoding and decoding still remain the same, and the usage of the inverse function g_0,g_1 are completely avoided, thus having a much more efficient and clean solution for the wavelet problem.

7 Assignment 8

In assignment 8 we implemented the techniques boundaryextraction,holefilling,erosion,diletation,opening,closing and connected components.

Since erosion and diltation are the conjugate of each other, we simply negated the logic operators in erosion to get diletation.

```
def erosion(inputimg, mask):
    maskoffset = len(mask) / 2
    returnimg = np.zeros(inputimg.shape, dtype=bool)
    for i in range(maskoffset, len(inputimg) - maskoffset):
        for j in range(maskoffset, len(inputimg[0]) - maskoffset):
            curmask = inputimg[i - maskoffset:i + maskoffset + 1, j - maskoffset:j + maskoffset + 1]
            if (curmask == mask).all() :
                returnimg[i, j] = 1
    return returnimg
```

Moreover the diletation is computed only changing the values within the returnedimg. Whereas for erosion we start by having a zeroed array, for diletation we start by having a array filled with ones and zero it out.

```
def diletation(inputimg, mask):
    maskoffset = len(mask) / 2
    m, n = inputimg.shape
    returnimg = np.ones((m, n), dtype=bool)
    padimg(returnimg, 0)
    for i in range(maskoffset, len(inputimg) - maskoffset):
        for j in range(maskoffset, len(inputimg[0]) - maskoffset):
            structureelement = inputimg[i - maskoffset:i + maskoffset + 1, j - maskoffset:j + maskof
            if (structureelement == mask).all():
                returnimg[i, j] = 0
    return returnimg
```

Opening and closing are just combinations of the operations above.

Boundary following and hole filling is done using again the same algorithms of above, yet their runtime is very slow, so it is not recommended to run these by anybody again (runtime is at least 30 mins).

8 Assignment 9

8.1 Edgedetection

In this exercise, the maar filter, was firstly used to get some sense of how the filter should work. The implemented filter uses a laplacian mask to firstly get the second derivative of the image. Afterwards the filter finds the zero crossings of a pixels neighbourhood.

The canny filter is therefore by far complexer.

The canny filter has the following 4 steps:

1. Smooth the image by using a Gaussian
2. Calculate the gradient magnitude
3. Apply non-maximum suppression
4. Apply hysteresis thresholding

The steps are done using some help of pythons numpy library. We begin by calculate the gradient magnitude and angles. gx, gy are the 2 dimensional grids for the x, y components respectively, which are calculated by applying a sobel or prewitt operator on the image.

```
def gradientmagnitude(gx, gy):
    return np.hypot(gx,gy)


def directionangle(gx, gy):
    return np.arctan2(gy , gx)
```

After the angles and the magnitudes are calculated, we normalize the angles to go in four different directions: $0, 45, 90, -45$, by simply checking the angle range:

```
angle[(angle < 22.5) | (angle > 337.5) & (angle < 202.5) | (angle > 157.5) ] = 0.
angle[((angle >= 22.5) & (angle <= 67.5)) | ((angle >= 202.5) & (angle <= 247.5))] = 45.
angle[((angle > 67.5) & (angle < 112.5)) | ((angle >= 247.5) & (angle <= 292.5))] = 90.
angle[((angle <= 157.5) & (angle > 112.5)) | ((angle >= 292.5) & (angle <= 337.5))] = -45.
```

Maximum suppression is achieved by setting specific values in the mag matrix to zero:

```
mag_sup = np.copy(mag)
    for x in range(1, width - 1):
        for y in range(1, height - 1):
            if angle[x][y] == 0:
                if (mag[x][y] <= mag[x][y + 1]) or \
                   (mag[x][y] <= mag[x][y - 1]):
                    mag_sup[x][y] = 0
            elif angle[x][y] == 45:
```

```
                if (mag[x][y] <= mag[x - 1][y + 1]) or \
                   (mag[x][y] <= mag[x + 1][y - 1]):
                    mag_sup[x][y] = 0
            elif angle[x][y] == 90:
                if (mag[x][y] <= mag[x + 1][y]) or \
                   (mag[x][y] <= mag[x - 1][y]):
                    mag_sup[x][y] = 0
            else:
                if (mag[x][y] <= mag[x + 1][y + 1]) or \
                   (mag[x][y] <= mag[x - 1][y - 1]):
                    mag_sup[x][y] = 0
```

Edge linking is the process to link all low-thresholded edges to the high ones, whereas the high edges are the ones which are filtered by a larger threshold than the low ones, e.g.:

```
m = np.max(mag_sup)
th = m*0.1
tl = th/2
gnh = np.zeros((width, height),dtype=float)
    gnl = np.zeros((width, height),dtype=float)
    for x in range(1,width-1):
        for y in range(1,height-1):
            if mag_sup[x][y] >= th:
                gnh[x][ y] = mag_sup[x][y]
            if mag_sup[x][y] >= tl:
                gnl[x][ y] = mag_sup[x][y]
gnl = gnl - gnh
def traverse(i, j):
        x = [-1, 0, 1, -1, 1, -1, 0, 1]
        y = [-1, -1, -1, 0, 0, 1, 1, 1]
        for k in range(8):
            if gnh[i + x[k]][j + y[k]] == 0 and gnl[i + x[k]][j + y[k]] != 0:
                gnh[i + x[k]][j + y[k]] = 1
                traverse(i + x[k], j + y[k])
for i in range(1, width - 1):
        for j in range(1, height - 1):
            if gnh[i][j]:
                gnh[i, j] = 1
                traverse(i, j)
```

After traversing the items, the edge linking is done and the array is returned, which contains the picture of the canny edge detection.

9 Assignment 10

The last Assignment is about using boundary following to extract chain codes and using PCA for compression.

9.1 Boundary following

To follow the boundary we used the following algorithm:

1. Put first point , marked as b_0 in a stack and save it's coordinates.
2. Pop the point in the list, which was added last call it b.
3. If that b is the very first popped b, save during this iteration the coordinates of the found point as b_1, else skip
4. Use b and get the corresponding c and start to iterate in clockwise one full round from c on.
5. If another point is found during that iteration , store it in the iteration list and also save the position of the new point's starting-point
6. Do it as long as you reach the point b_1 again.

In our implementation we use a class to store the objects b and c.

```
class Point():
    ''' A container for the given Points, stores 2 variables and returns an iterator and a given ite
        The iterator returns for the current point the circle around it.
    '''
    neighboriter = zip(xstep, ystep)

    def __init__(self, b, c):
        self.b = b
        self.c = c

    def __repr__(self):
        ret = "".join(str(self.b))
        return ret

    def __iter__(self):
        index = self.neighboriter.index(self.c)
#         We need to check both of the slices, we get on one side the left over circle and on the ot
        return iter(self.neighboriter[index:] + self.neighboriter[:index])

    def __getitem__(self, key):
        index = self.neighboriter.index(self.c)
        iterlist = self.neighboriter[index:] + self.neighboriter[:index]
        return iterlist[key]
#     During looping, the in operator can be used to check the b value
    def __eq__(self, other):
        return other == self.b
```

When calculating the first difference, we used the following formula. Given two adjacent elements $c_i, c_{i+1} \in C$, when using b different angles ($b = 8$ in our case), we calculate:

$$s = (b - (c_i - c_{i+1})) \mod b$$

Which gives us the correct results, using the modulo operator as "mod"

9.2 PCA

Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. The number of principal components

is less than or equal to the number of original variables. This transformation is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to (i.e., uncorrelated with) the preceding components. The principal components are orthogonal because they are the eigenvectors of the covariance matrix, which is symmetric. PCA is sensitive to the relative scaling of the original variables.

The results of the PCA compression can be seen in the Assignment 10 folder.