# SIT789 – Robotics, Computer Vision and Speech Processing

## Credit Task 2.2: Edge orientation and morphology

## Objectives

The objectives of this lab include:

- Calculating edge orientations of grayscale images
- Applying morphological transforms to document skew estimation

## Tasks

### 1. Calculating edge orientations

In Task 2.1, you have learnt how to calculate horizontal and vertical derivatives of a grayscale image. In week 2, you have also learnt how to calculate edge orientations using horizontal and vertical derivatives (see week 2 handouts). You are now to apply this technique to compute the edge orientations for pixels on the grayscale version of the supplied file "fisherman.jpg" (in OnTrack).

Like Task 2.1, you first need to load the image file "fisherman.jpg" and then convert it to grayscale using the following code,

```python
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt

img = cv.imread('fisherman.jpg') #load image
img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
```

The horizontal and vertical derivatives of img_gray can be calculated using the Sobel filters. Let der_x and der_y be the horizontal and vertical derivative of img_gray respectively. You then need to create a 2D array, namely ori_img_gray, to store the edge orientations of pixels in img_gray.

```python
height, width = img_gray.shape
ori_img_gray = np.zeros((height, width), np.float32) # gradient orientation of img_gray
```

Scan the array ori_img_gray using two for loops (like calculating gradient magnitude in section 2.1 in Task 2.1). At each pixel [i, j], check the following condition. If both der_x[i, j] = 0 and der_y[i, j] = 0, assign ori_img_gray[i, j] = math.inf (i.e., pixel [i, j] does not have gradient and thus its gradient direction is set to infinity). Otherwise, assign ori_img_gray[i, j] = $\tan^{-1}\left(\frac{der_y[i,j]}{der_x[i,j]}\right)$. You can use math.atan2(der_y[i, j], der_x[i, j]) to calculate $\tan^{-1}\left(\frac{der_y[i,j]}{der_x[i,j]}\right)$.

**Note**: Do not forget to import math to use math.atan2, math.inf, math.pi (i.e., $\pi$). atan2(der_y, der_x) will return an angle $\theta$ (in radian) $\in [-\pi, \pi]$. To convert this angle from radian to degree, you can use the following formula:

$$degree = \frac{radian \times 180}{\pi}$$

The angles stored at ori_img_gray[i,j] now vary in $[-180^\circ, 180^\circ]$. Recall that the edge orientations are orthogonal to the gradient directions. Therefore, to obtain the edge orientations for pixels in img_gray, we simply add 90 to the gradient directions of the pixels, i.e., ori_img_gray[i,j] = ori_img_gray[i, j] + 90. This addition transforms ori_img_gray[i,j] from $[-180^\circ, 180^\circ]$ to $[-90^\circ, 270^\circ]$. Note that this step is applied to only pixels whose ori_img_gray[i,j] is not infinity.

Now, given a pixel at [i,j], the edge orientation of that pixel is in $[-90^\circ, 270^\circ]$. Next, you need to represent edge orientations in $[0^\circ, 360^\circ]$. This conversion can be done simply by adding 360 to angles in $[-90^\circ, 0^\circ)$. You can revisit trigonometric rules presented in the Maths supplementary material supplied in CloudDeakin for further details. After doing the conversion for all the edge orientations, you are to create a histogram for the edge orientations using a 1D array including 361 bins (from 0 to 360). Each $k$-th bin in the histogram stores the number of pixels whose edge orientation is $k$. You can revisit Task 1.2C to see how a histogram of an image's intensity levels is calculated. Your task is to calculate such a histogram and plot it in your notebook.

Next, applying the same methodology, you are to calculate an edge orientation histogram for the image in "empire.jpg" (in Task 2.1). Finally, you need compare these two histograms and discussion your findings.

# 2. Applying morphology for document skew estimation

## 2.1. Binary images

We have studied (see week 2 handouts) that a binary image can be obtained by thresholding a grayscale image. In this task, you will be working with binary images. First, you are to download the supplied image file doc.jpg (from OnTrack) and load this image into a variable named "doc" as follows.

```
doc = cv.imread('doc.jpg', 0) # 2nd parameter is set to 0 to read grayscale image
```

You then threshold the image doc with a threshold = 200. Recall that the intensity value for each pixel is integer and ranges in [0, 255] where 0 represents BLACK and 255 represents WHITE. You can use the following code to perform thresholding.

```
threshold = 200
ret, doc_bin = cv.threshold(doc, threshold, 255, cv.THRESH_BINARY)
```

cv.threshold sets pixels whose intensity < threshold as BLACK and those whose intensity >= threshold as WHITE. Therefore, the greater the threshold is, the more black pixels the image has. You need to save the binary version of image doc in a file named "doc_bin.png". You can try with different thresholds and observe the effect of the thresholding method.

## 2.2. Morphological transforms for document skew estimation

In this task, we will re-implement parts of the following paper:

D. T. Nguyen et al., A robust document skew estimation algorithm using mathematical morphology. In proceedings of the IEEE Int. Conf. Tools with Artificial Intelligence, 2007.

The algorithm includes five steps:

**Step 1:** Applying morphological transforms to merge letters in words and words in text lines. To do so, we first apply closing operation with the following structuring element

```
closing_se = np.ones((1, 15), int) # structuring element for closing
```

**Note**: morphological transforms are applied on FOREGROUND pixels. By default, OpenCV considers WHITE pixels (i.e., pixels whose intensity is 255) foreground while the text in our image doc_bin is presented in BLACK. Therefore, prior to applying morphological transforms, we need to get the *negative* image of doc_bin as,

```
doc_bin = 255 - doc_bin # convert black/white to white/black
```

By doing so, we have converted pixels from WHITE to BLACK and vice versa. We then apply a closing operation to doc_bin.

```
closing = cv.morphologyEx(doc_bin, cv.MORPH_CLOSE, closing_se)
plt.imshow(closing, 'gray')
```

The closing operation aims to link letters with the same word and words within the same text line. However, it may also link text from different text lines and/or background noise. To remove such unexpected links, an opening operation is applied to the result of the closing operation subsequently. We first define a structuring element for the opening operation.

```
opening_se = np.ones((8, 8), int) # structuring element for opening
```

Then, call the opening operation with the defined structuring element opening_se

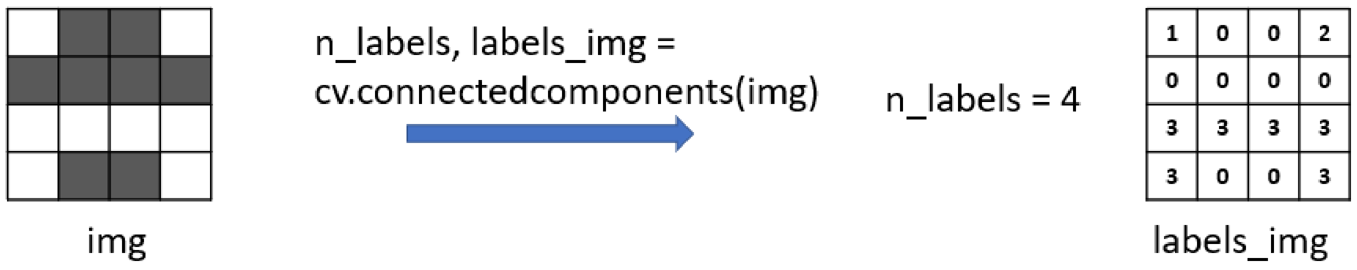
```
opening = cv.morphologyEx(closing, cv.MORPH_OPEN, opening_se)
plt.imshow(opening, 'gray')
```

As shown, the opening operation has cleared up the text lines. Nearby text lines are now well separated. We now move to step 2.

**Step 2**. Extract merged text lines using the connected component labelling technique where each text line corresponds to a connected component.

```
# connected component labelling
num_labels, labels_im = cv.connectedComponents(opening)
```

cv.connectedComponents receives input as a binary image and groups WHITE pixels into connected components. The output of cv.connectedComponents includes the number of connected components and an image containing labels of connected components. In the output labelled image, e.g., labels_im above, each foreground pixel is labelled with the ID of a connected component it belongs to. The following figure illustrates the input and output of cv.connectedComponents.



| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 2 |
| 0 | 0 | 0 | 0 |
| 3 | 3 | 3 | 3 |
| 3 | 0 | 0 | 3 |

For the ease and convenience in following steps, we transform the output of cv.connectedComponents into an array structure where each element is a list of pixels of a connected component. The following function is used for such purpose.

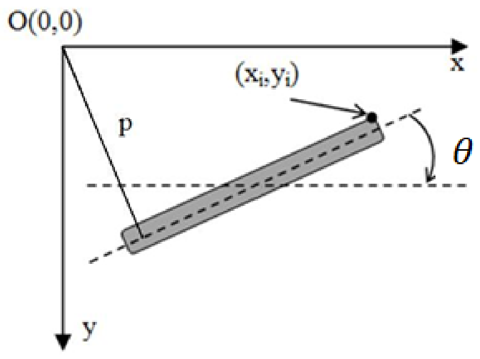
```python
def ExtractConnectedComponents(num_labels, labels_im):
    connected_components = [[] for i in range(0, num_labels)]

    height, width = labels_im.shape
    for i in range(0, height):
        for j in range(0, width):
            if labels_im[i, j] >= 0:
                connected_components[labels_im[i, j]].append((j, i))

    return connected_components
```

We then call the above ExtractConnectedComponets to store components in an array structure.

```python
connected_components = ExtractConnectedComponents(num_labels, labels_im)
```

**Step 3**. Find the orientation for connected components (i.e., text lines) using least squares regression. In particular, for each connected component, we assume that there exists a line representing the connected component (see the figure below). That line is formulated by the following equation.

$$p = x \sin\theta + y \cos\theta$$



A pair of $(p, \theta)$ determines a unique line. Let cc be a connected component including *N* pixels $\{p_i = (x_i, y_i)\}_1^N$. The fitness of a line determined by $(p, \theta)$ with cc is defined as,

$$E = \sum_{i=1}^{N} (x_i \sin\theta + y_i \cos\theta - p)^2$$

The best fitted line of cc will minimise *E*. Therefore, to find the orientation $\hat{\theta}$ of the best fitted line of cc, we get the partial derivative of *E* w.r.t. $\theta$ and solve the following equation.

$$\frac{\partial E}{\partial \theta} = 0$$

Solving the above equation, we get,

$$\hat{\theta} = -\frac{1}{2}\tan^{-1}\left(\frac{2\mu_{xy}}{\mu_{xx} - \mu_{yy}}\right)$$

where

$$\mu_{xy} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y)$$

$$\mu_{xx} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)^2$$

$$\mu_{yy} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mu_y)^2$$

where

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N}x_i$$

$$\mu_y = \frac{1}{N}\sum_{i=1}^{N}y_i$$

This technique is called least squares regression. The following function calculates the orientation of a connected component (cc) using least squares regression.

```python
import math
def FindOrientation(cc):
    mx = 0
    my = 0
    mxx = 0
    myy = 0
    mxy = 0

    for i in range(0, len(cc)):
        mx += cc[i][0] # cc[i][0] is used to store the x coordinate of pixel cc[i]
        my += cc[i][1] # cc[i][1] is used to store the y coordinate of pixel cc[i]
    mx /= len(cc)
    my /= len(cc)

    for i in range(0, len(cc)):
        dx = cc[i][0] - mx
        dy = cc[i][1] - my
        mxx += (dx * dx)
        myy += (dy * dy)
        mxy += (dx * dy)
    mxx /= len(cc)
    myy /= len(cc)
    mxy /= len(cc)

    theta = - math.atan2(2 * mxy, mxx - myy) / 2
    return theta
```

We now call FindOrientation for all connected components and stores resulting orientations in a array named "orientations" for further processing

```python
orientations = np.zeros(num_labels, np.float32)
for i in range(0, num_labels):
    orientations[i] = FindOrientation(connected_components[i])
```

**Step 4**. The orientation of the entire document is computed as the median of the orientations of all text lines. We will call median method from statistics to get this median.

```
import statistics
orientation = statistics.median(orientations)
```

**Step 5**. We now deskew the image in doc by rotating it with an angle of – orientation.

```
# rotate image
height, width = doc.shape
c_x = (width - 1) / 2.0 # column index varies in [0, width-1]
c_y = (height - 1) / 2.0 # row index varies in [0, height-1]
c = (c_x, c_y) # A point is defined by x and y coordinate

M = cv.getRotationMatrix2D(c, -orientation * 180 / math.pi, 1)
doc_deskewed = cv.warpAffine(doc, M, (width, height))
plt.imshow(doc_deskewed, 'gray')
```

Applying the same methodology, deskew the document image in doc_1.jpg (supplied in task resources in Ontrack). Does the method successfully work in this case? If not, what could be the reason? Observe the result and discuss your observations.

## Submission instructions

1. Complete the supplied answer sheet with required results and discussions.
2. Submit the answer sheet (in pdf format) to OnTrack.
3. Submit your code (in .py format) to OnTrack.