# HW 03 – REPORT

소속 : 전기컴퓨터공학부 정보컴퓨터공학전공

학번 : 201824469

이름 : 박동진

# 1. 서론

이번 과제에서는 Canny Edge Detection을 구현한다.

Canny Edge Detection을 위해서는 아래와 같은 과정을 거쳐야 한다.

## 1-1. Canny Edge Detection과정

1. 노이즈를 제거하기 위해 이미지에 가우시안 필터를 적용한다.
    - 노이즈를 제거하는 이유는 픽셀 단위로 gradient를 구할 때, 잘못된 gradient를 감지하지 않기 위함이다.
2. 가우시안 필터를 적용한 이미지에 sobel필터를 적용하여 가로, 세로 방향의 gradient를 구한다. 이 gradient의 값이 높다면 해당 방향으로 모서리가 발생한 것을 의미한다. 각 방향의 gradient를 hypot연산하면 원본 이미지(과정1을 마친 이미지)의 테두리를 얻을 수 있다.
3. 2에서 얻은 x, y방향의 gradient를 arctan함수를 이용해 정확한 모서리의 방향을 알아낸다.
4. 모서리의 방향에 따라 최댓값만을 남기고 픽셀을 제거하는 non-maximum supression을 통해 감지된 테두리의 두께를 얇게 변환한다.
5. double-threshold를 통해 분명한 테두리와 희미한 테두리를 구별한다.
6. 분명한 테두리는 테두리로, 희미한 테두리는 분명한 테두리에 연속될 경우 테두리로, 아니면 무시한다.

위의 과정을 하나하나 구현하고 수행하면서 제공된 이미지 iguana.bmp의 테두리를 감지하여 시각화하는 것을 목표로 한다.

정확한 구현에 성공한다면 이미지의 테두리를 감지한 이미지를 확인할 수 있을 것이다.

# 2. 본론

라이브러리를 불러온다.

```
from PIL import Image
import math
import numpy as np
```

## 2-1 Noise Reduction

가우시안필터를 이용해서 노이즈를 제거한다. 노이즈를 제거하지 않으면 정확한 gradient를 계산하기 힘들어진다. HW2에서 사용한 가우시안 필터와 컨벌루션연산을 가져온다.

```
def gauss1d(sigma):
    width: int = int(math.ceil(6 * sigma))  # 6*sigma의 다음 정수로 width를 설정
    if width % 2 == 0:  # 다음 정수가 짝수라면 +1 하여 홀수로 변경
        width += 1

    center = math.floor(width / 2)  # 가운데 index를 획득
    gaussian = np.zeros((1, width)).flatten()  # 반환할 1D Numpy Array 생성
    for i in range(width):
        gaussian[i] = i - center  # Hint:를 토대로 x값을 입력
    for i in range(width):
        # x값에 따라 gaussian값 획득 및 입력
        gaussian[i] = math.exp(-gaussian[i] ** 2 / (2 * sigma ** 2))

    # 아래는 normalize, 전체 합으로 각 엔트리를 나눈다.
    kernal_sum = np.sum(gaussian)
    for i in range(width):
        gaussian[i] /= kernal_sum
    return gaussian

def gauss2d(sigma):
    return np.outer(gauss1d(sigma), gauss1d(sigma))
# outer product로 2d 가우시안 필터를 구한다.

def convolve2d(array, filter):
    padding_width = int(len(filter) / 2)  # filter에 따른 padding 크기
    zero_padding = np.zeros((np.shape(array)[0] + padding_width * 2,
                             np.shape(array)[1] + padding_width * 2))
    # padding한 크기에 맞게 새로운 np.array를 생성

    # convolution연산을 위해서 filter를 축의 방향에 대해서 뒤집는다.
    flipped_filter = np.flip(filter, axis=0)
    flipped_filter = np.flip(flipped_filter, axis=1)
    # 모든 픽셀에 대해서 paading을 제외한 부분에 array를 집어넣어 준다.
    for i in range(padding_width, np.shape(zero_padding)[0] - padding_width):
        for j in range(padding_width, np.shape(zero_padding)[1] - padding_width):
            zero_padding[i][j] = array[i - padding_width][j - padding_width]

    # padding된 이미지의 각 픽셀을 순회하면서 각 픽셀에 컨벌루젼한 값을 입력한다.

    convolved = np.zeros(np.shape(array))

    for i in range(padding_width, np.shape(zero_padding)[0] - padding_width):
        for j in range(padding_width, np.shape(zero_padding)[1] - padding_width):
            subpart = zero_padding[i - padding_width:i + padding_width + 1,
                      j - padding_width:j + padding_width + 1]
            convolved[i-padding_width][j-padding_width] = np.sum(subpart * flipped_filter)
    # 컨벌루젼한 값을 리턴
    return convolved

def gaussconvolve2d(array, sigma):
    return convolve2d(array, gauss2d(sigma))
```

작성한 함수를 이용해서 이미지를 흑백처리하고 가우시안필터를 적용한다. 가우시안필터에 적용할 시그마값은 과제에서 제공한 1.6이다.

실제 함수의 적용은 main()함수 내부에서 일어난다.

```
# 이구아나 사진을 불러온다.
iguana_img = Image.open('iguana.bmp')
# 사진을 흑백처리한다.
iguana_img_grey = iguana_img.convert('L')
# 이미지를 np 배열형태로 변형한다.
iguana_array_grey = np.asarray(iguana_img_grey)
# 가우시안 필터 효과를 적용한다.
iguana_array_grey_blur = np.uint8(gaussconvolve2d(iguana_array_grey, 1.6))
# 배열을 이미지로 변경하여 보여준다.
Image.fromarray(iguana_array_grey_blur.astype(np.uint8)).show()
```

결과물은 아래와 같다. 조금 흐려지고 흑백이 된 이구아나의 사진이다.

## 2-2 Finding the intensity gradient of the image

소벨필터를 적용하여 위에서 생성한 이미지의 테두리를 얻어내려고 한다. 소벨필터란 x, y축의 변화량을 나타나게 하는 필터로 강의에 따르면 아래와 같다.

$$\frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = S_x \qquad \frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = S_y$$

필터를 구현한다.

```
#sobel연산을 위한 필터를 제작한다. normalization을 위해 8로 나눈다.
X_SOBEL = np.array([1, 0, -1, 2, 0, -2, 1, 0, -1])
Y_SOBEL = np.array([1, 2, 1, 0, 0, 0, -1, -2, -1])
X_SOBEL = X_SOBEL.reshape((3, 3)) / 8.0
Y_SOBEL = Y_SOBEL.reshape((3, 3)) / 8.0
```

앞으로의 작업을 위해 `sobel_filters` 함수를 구현한다. `sobel_filters` 함수는 이미지의 각 방향으로 소벨필터를 적용하여 gradient를 구하고, 가로/세로 방향의 gradient를 hypot연산(제곱하여 합하여 루트씌우기)하여 gradient의 intensity를 구해 반환한다. 반환할 때에는 intensity의 최댓값과 최솟값을 기준으로 0~255에 매핑한다.(G)

또한 `sobel_filters`는 각 픽셀의 gradient와 arctan함수를 이용하여 gradient의 방향을 theta 라는 이름의 배열로 반환한다. 이 theta는 non_maximum suppression에서 테두리를 얇게 만들 때 사용된다. theta의 범위는 -pi/2~pi/2이다.

arctan함수를 이용할 때, 분모가 0(x방향 gradient가 0인 경우) 일 때 발생하는 오류를 방지하기 위해서 아래와 같은 수를 정의한다.

```
# 아래 상수는 divide by zero를 방지하기 위한 적당히 작은 수이다.
QUITELY_SMALL = 0.00000000001
```

구현한 sobel_filters함수는 아래와 같다.

```
def sobel_filters(img):
    # x와 y의 방향에 대해서 각각 sobel 필터를 적용한다.
    x_gradient_img = convolve2d(img, X_SOBEL)
    y_gradient_img = convolve2d(img, Y_SOBEL)

    # G를 계산한다.
    G = np.hypot(x_gradient_img, y_gradient_img)
    # G의 각 원소를 0과 255사이로 매핑한다.
    G = (G / np.max(G)) * 255

    # np.vectorize 함수는 np.array의 모든 원소에 특정 함수를 적용하는 것을 쉽게 만든다.
    vectorized_arctan = np.vectorize(np.arctan)
    # arctan의 분모에 들어갈 x값이 0일 경우 divide by zero가 발생할 수 있으므로 0을 적당히 작은 값으로 치환한다.
    x_gradient_img[x_gradient_img == 0] = QUITELY_SMALL
    # np.arctan를 이용해서 theta 배열을 구한다.
    theta = vectorized_arctan(y_gradient_img / x_gradient_img)
    return (G, theta)
```

```
mapped_sum_gradient_iguana, theta_iguana = sobel_filters(iguana_array_grey_blur)
Image.fromarray(mapped_sum_gradient_iguana.astype(np.uint8)).show()
```

가우시안 필터가 처리된 이구아나 이미지에 소벨필터를 적용하여 출력한다. 결과물은 아래와 같다. 이구아나의 테두리가 intensity의 형태로 보여진다.

## 2-3 Non-Maximum Suppression

`non_maximum_suppression`함수를 구현한다. 해당 함수는 매개변수로 이미지 G와 gradient각도인 theta를 받는다. `theta`는 라디안으로 계산되어 있기 때문에, 계산을 용이하게 하기 위해서 60분법으로 바꿔야 한다.

라디안을 60분법으로 바꾼 후에는 각도를 0, 45, 90, 135 중에서 가장 가까운 값으로 매핑한다. 22.5를 더하고 45로 나눈 몫을 구하면 5개의 값으로 매핑되는데,

- -2/2의 경우 theta의 값은 -67.5 ~ -90 혹은 90 ~ 67.5이다
- 1의 경우 theta의 값은 67.5 ~ 22.5이다.
- 0의 경우 theta의 값은 22.5 ~ -22.5이다.
- -1의 경우 theta의 값은 -67.5 ~ -22.5이다.

매개변수의 값은 theta의 범위는 -pi/2 ~ pi/2이지만, 매핑을 통해서 적절한 값으로 매핑된다. 이 때 각도는 동경(x축)을 기준으로 한다.

매핑된 `theta`를 이용해서 테두리를 걷어낸다. `remain_max`함수는 angle과 좌표를 이용해서 해당 좌표가 edge의 최댓값이면 해당 intensity를 반환하고, 최대가 아닌 값은 0을 반환한다.

```
def remain_max(array, x, y, angle):
    """
    그림 배열 array의 x(column), y(row)위치에서 angle에 따라 해당 픽셀에 할당되어야 하는 값을 반환한다.
    할당되어야 하는 값이란 굵게 표시된 edge의 최댓값만을 남기기 위해 픽셀에 적용해야 하는 값이다.

    :param array:적용할 이미지 배열
    :param x:x값, 즉 column number
```

```
    :param y:y값, 즉 row number
    :param angle: 해당 픽셀에서의 gradient 방향
    :return: 해당 픽셀에 최고 edge만을 남기기 위해서 적용되어야 하는 값,
    해당 방향으로의 주변픽셀을 확인하여 자신이 최댓값일 경우 남기고 아닐 경우 0으로 치환
    """
    ret = 0
    # -2|2 => 90도, 1 => 45도, 0 => 0도, -1 => 135도
    if angle == -2 or angle == 2:
        if array[y][x] < max(array[y+1][x], array[y][x], array[y-1][x]):
            ret = 0
        else:
            ret =  array[y][x]
    if angle == -1:
        if array[y][x] < max(array[y+1][x-1], array[y][x], array[y-1][x+1]):
            ret =  0
        else:
            ret =  array[y][x]
    if angle == 0:
        if array[y][x] < max(array[y][x+1], array[y][x], array[y][x-1]):
            ret = 0
        else:
            ret = array[y][x]
    if angle == 1:
        if array[y][x] < max(array[y+1][x+1], array[y][x], array[y-1][x-1]):
            ret = 0
        else:
            ret = array[y][x]
    return ret
```

`non_max_suppression`함수는 `remain_max`함수를 모든 픽셀에 적용한다. 이로 인해서 테두리부분이 더 다듬어진다.

```
def non_max_suppression(G, theta):
    # 라디안으로 들어온 theta를 60분법으로 변환시킨다.
    theta_degree = theta * 180 / np.pi
    # 변환된 각도를 0, 45, 90, 135중에서 가장 가까운 각도에 mapping시킨다.
    # 22.5를 더하는 것은
    theta_mapped = (theta_degree + 22.5) // 45
    """
    theta_mapped의 각 값을 해석하면 아래와 같다.
    -2|2 => 90도, 1 => 45도, 0 => 0도, -1 => 135도
    해석에 따라 주변 픽셀을 비교하여 non_max_suppression을 수행한다.
    """
    # 각도에 따라서 각 픽셀에서 남길 만한 값을 제외하고는 제거한다.
    res = np.zeros(np.shape(G))
    for row in range(1, len(G)-1):
        for col in range(1, len(G[1])-1):
            res[row][col] = remain_max(G, col, row, theta_mapped[row][col])
    # 결과를 반환한다.
    return res
```

작성한 `non_max_suppression`함수를 적용한다.

```
non_max_suppressed_iguana = non_max_suppression(mapped_sum_gradient_iguana, theta_iguana)
Image.fromarray(non_max_suppressed_iguana.astype(np.uint8)).show()
```

결과물은 아래와 같다. 테두리가 더 날카로워졌다.

## 2-4 Double threshold

`double_thresholding` 함수를 작성한다. 이 함수는 non-max-suppression을 통해서 날카롭게 변한 테두리들 중에서 진짜 테두리를 추리는 과정이다. 각 gradient intensity를 비교해서 strong edge, weak edge, no-edge를 판단한다. 과제에서 판단의 기준이 되는 역치점은 주어졌다.

strong_edge와 weak_edge를 표현할 intensity역시 과제에서 주어졌다. 상수로 선언한다.

```
STRONG_EDGE_INTENSE = 255
WEAK_EDGE_INTENSE = 80
```

각 픽셀에 대해서 적용할 것이므로, 함수를 만들어 vectorize를 이용하려고 한다. 역치점을 적용하는 함수를 작성한다.

```
def apply_double_threshold(entry, T_high, T_low):
    """
    :param entry: Threshold 적용할 값
    :param T_high: Strong Edge의 Threshold
    :param T_low: Weak Edge의 Threshold
    :return: Entry의 값을 보고, T_high보다 높으면 STRONG_EDGE_INTENSE,
     T_low와 T_high사이이면 WEAK_EDGE_INTENSE,
     T_low보다 작으면 0.
    """
    if entry >= T_high:
        return STRONG_EDGE_INTENSE
    elif T_high > entry >= T_low:
        return WEAK_EDGE_INTENSE
    else:
        return 0
```

모든 픽셀에 위의 함수를 적용하는 함수 double_thresholding은 아래와 같다. 역치점은 과제에서 주어진 식을 기반으로 한다.

$$diff = \max(image) - \min(image)$$
$$T_{high} = \min(image) + diff * 0.15$$
$$T_{low} = \min(image) + diff * 0.03$$

```
def double_thresholding(img):
    # diff와 문제에서 제시된 threshold를 구한다.
    diff = img.max()
    T_high = img.min() + diff * 0.15
    T_low = img.min() + diff * 0.03

    # 결과를 기록하고 반환할 res 배열을 img에서 복사한다.
    res = img[:]

    # 각 픽셀에 threshold를 적용한 값을 입력하기 위해 vectorize 생성 후 threshold 적용
    vectorized_threshold = np.vectorize(apply_double_threshold)
    res = vectorized_threshold(img, T_high, T_low)

    # 완성된 결과를 반환
    return res
```

구현된 함수를 적용한다.

```
double_threshold_iguana = double_thresholding(non_max_suppressed_iguana)
Image.fromarray(double_threshold_iguana.astype(np.uint8)).show()
```

적용된 모습은 아래와 같다. 각 테두리의 강도에 따라서 사라지거나 strong, weak edge로 치환되었다.

## 2-5 Edge Tracking by hysteresis

위에서 구한 이미지를 통해서 테두리를 판별한다. 테두리를 판별할 때에는 다음과 같은 규칙을 따른다.

- 강한 테두리는 테두리이다.
- 약한 테두리는 강한 테두리의 연장일 경우 테두리다.
- 그 외에는 테두리가 아니다.

두 번째 규칙을 적용하기 위해서는 강한 테두리 주변이 있는 약한 테두리를 테두리로 치환해야 한다. 이 때 탐색 알고리즘이 이용된다. 우리는 과제에서 주어진 dfs를 이용할 것이다.

```
def dfs(img, res, i, j, visited=[]):
    # 호출된 시점의 시작점 (i, j)은 최초 호출이 아닌 이상
    # strong 과 연결된 weak 포인트이므로 res에 strong 값을 준다
    res[i, j] = 255

    # 이미 방문했음을 표시한다
    visited.append((i, j))

    # (i, j)에 연결된 8가지 방향을 모두 검사하여 weak 포인트가 있다면 재귀적으로 호출
    for ii in range(i-1, i+2) :
        for jj in range(j-1, j+2) :
            if (img[ii, jj] == 80) and ((ii, jj) not in visited) :
                dfs(img, res, ii, jj, visited)

def hysteresis(img):
    # 반환할 배열을 생성
    res = np.zeros(np.shape(img))
    # 방문지 기록
    visited = []
    # 순회 인덱스를 찾기 위한 width, height 구하기
    width = len(img)
    height = len(img[0])

    # 모든 픽셀(모서리는 out of range를 방어하기 위해 제외)을 순회하며
    # strong edge에 대해서 dfs 수행
    for i in range(1, width-1):
        for j in range(1, height-1):
            if img[i][j] == 255:
                dfs(img, res, i, j, visited)
    return res
```

함수를 적용한다.

```
hysteresised = hysteresis(double_threshold_iguana)
Image.fromarray(hysteresised.astype(np.uint8)).show()
```

적용한 결과는 아래와 같다. 테두리라고 판별된 부분만이 하얗게 남는다.

구현한 함수의 실행은 `main`에서 이루어진다.

```
def main():
    # 1. noise reduction
    # 이구아나 사진을 불러온다.
    iguana_img = Image.open('iguana.bmp')
    # 사진을 흑백처리한다.
    iguana_img_grey = iguana_img.convert('L')
    # 이미지를 np 배열형태로 변형한다.
    iguana_array_grey = np.asarray(iguana_img_grey)
    # 가우시안 필터 효과를 적용한다.
    iguana_array_grey_blur = np.uint8(gaussconvolve2d(iguana_array_grey, 1.6))
    # 배열을 이미지로 변경하여 보여준다.
    Image.fromarray(iguana_array_grey_blur.astype(np.uint8)).show()

    # 2. Finding the intensity gradient of image
    mapped_sum_gradient_iguana, theta_iguana = sobel_filters(iguana_array_grey_blur)
    Image.fromarray(mapped_sum_gradient_iguana.astype(np.uint8)).show()

    # 3. Non-Maximum Suppression
    non_max_suppressed_iguana = non_max_suppression(mapped_sum_gradient_iguana, theta_iguana)
    Image.fromarray(non_max_suppressed_iguana.astype(np.uint8)).show()

    # 4. Double threshold
    double_threshold_iguana = double_thresholding(non_max_suppressed_iguana)
    Image.fromarray(double_threshold_iguana.astype(np.uint8)).show()

    # 5. Edge Tracking by hysteresis
    hysteresised = hysteresis(double_threshold_iguana)
    Image.fromarray(hysteresised.astype(np.uint8)).show()

main()
```

# 3.결론

Canny Edge Detection을 구현해 보았다. 모든 과정을 거치면 선명한 테두리만이 남은 이구아나 이미지를 구할 수 있다. 최종적으로 얻은 이미지는 아래와 같다.

사용되었던 상수의 값들 (threshold계수, 가우시안효과 시그마)을 조정하면 다른 이미지를 구할 수 있을 것이다.

가우시안효과에 사용되는 시그마값을 높게 하면 이미지가 전체적으로 흐릿해져 더 적은 테두리를 찾아낼 것이고 낮게 하면 더 많은 테두리를 찾아낼 것이다. 심지어는 노이즈마저 테두리로 인식할 수도 있다.

Threshold값을 더 엄격하게, Threshold값을 올리면 더 적은 테두리를 찾아낼 것이고, 낮추면 더 많은 테두리를 찾아낼 것이다. 테두리가 아닌 것도 테두리로 인식할 수도 있다.

이에 따라 적절한 매개변수를 설정하는 것에 대한 이슈가 발생할 수 있다.