

# 로봇비전시스템

# Assignment Report

Optical Flow & Mean Shift Segmentation

멀티미디어공학과  
2016112622  
김동연

# CONTENTS

## 1. 개발 환경

## 2. Code Explanation

### A. Algorithm Flow Chart

### B. Detail Process

- i. Theoretical background and operation description

- ii. Code description

## 3. Result Analysis

### A. Evaluate The Accuracy

### B. Code Operation Process

# 1. 개발 환경

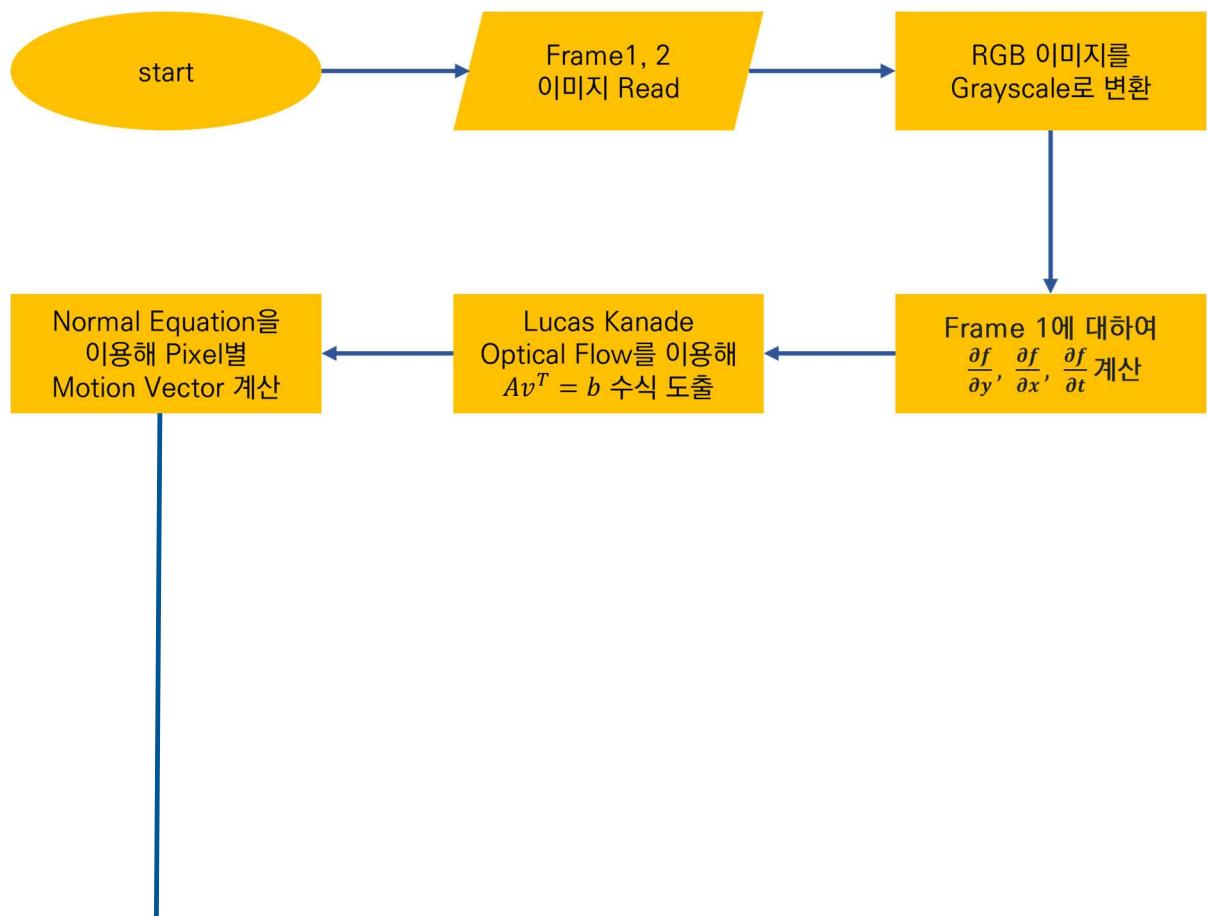
사용 언어 : python

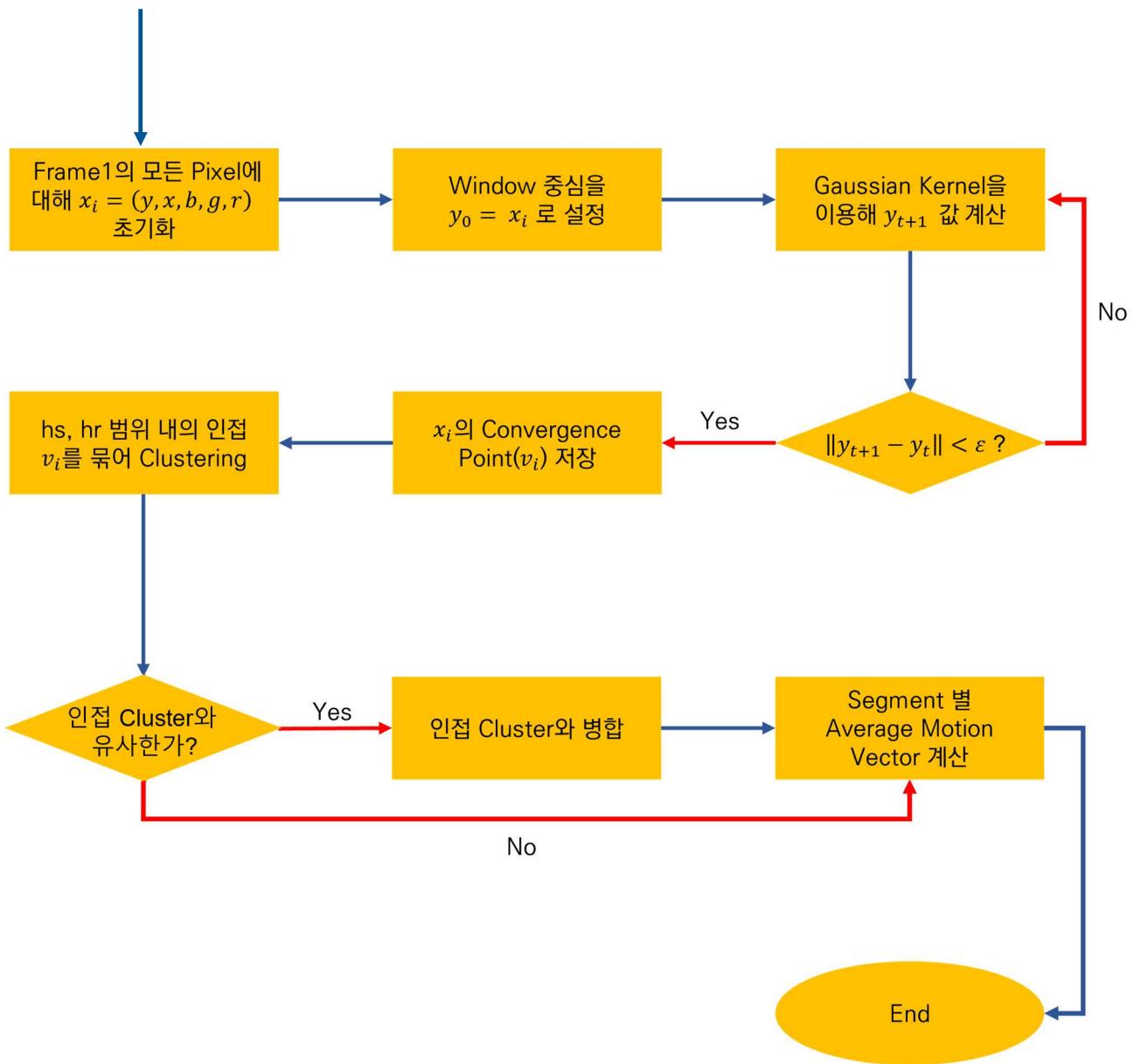
사용 라이브러리 : opencv-python + numpy

개발 IDE : PyCharm Community 2020

## 2. Code Explanation

### A. Algorithm Flow Chart





## B. Detail Process

### i. Theoretical background and operation description

#### ① Optical Flow

- Frame 1에 대하여  $\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial t}$  을 계산

2 개의 연속적인 frame에서 1 차 Taylor series 를 표현한 다음 수식을 통해 이동한 픽셀의 값을 알아낼 수 있다.

$$f(y+dy, x+dx, t+dt) = f(y, x, t) + \frac{\partial f}{\partial y} dy + \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial t} dt + \text{high order terms}$$

이때 두 frame 간의 시간 차이는 매우 짧고, Brightness consistency 를 가정하였으므로 다음 수식이 만족한다.

$$f(y+\Delta y, x+\Delta x, t+\Delta t) = f(y, x, t)$$

따라서 첫번째 수식으로부터 다음  $\frac{\partial f}{\partial y} dy + \frac{\partial f}{\partial x} dx + \frac{\partial f}{\partial t} dt$  값이 0 임을 도출해낼 수 있다.

위 수식의 양변을  $dt$ 로 나누면  $\frac{\partial f}{\partial y} \frac{dy}{dt} + \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial t} = 0$  이 되고  $\frac{dy}{dt}, \frac{dx}{dt}$  를 각각  $v, u$ 로 치환하게 되면  $\frac{\partial f}{\partial y} v + \frac{\partial f}{\partial x} u + \frac{\partial f}{\partial t} = 0$  와 같은 수식 하나를 픽셀마다 구할 수 있다.

하지만 위 수식에서 수식은 1 개이고, 미지수는  $v, u$ 로 2 개이기 때문에 무수히 많은  $v, u$ 가 구해지게 되어 하나의 벡터 값을 도출 할 수 없는 문제가 발생한다.

따라서 Lucas–Kanade Algorithm 을 이용하여 다음 문제를 해결한다.

- Lucas–Kanade Algorithm 을 이용해  $A\mathbf{v}^T = \mathbf{b}$  수식 도출

Lucas–Kanade Algorithm 에서는 픽셀  $(y, x)$ 를 중심으로 하는 window 범위 내부의 모든 픽셀의 optical flow 가 같다고 가정을 한다.

즉, window 사이즈가  $3 \times 3$  이면 9 개 픽셀의 optical flow 가 모두 동일하다고 가정하게 된다.

따라서 window 내부에 존재하는 pixel 의 개수만큼 다음과 같은  $v, u$ 에 대한 수식들을  $\frac{\partial f(y_i, x_i)}{\partial y} v + \frac{\partial f(y_i, x_i)}{\partial x} u + \frac{\partial f(y_i, x_i)}{\partial t} = 0, (y_i, x_i) \in N(y, x)$  도출할 수 있다.

위 수식들을 행렬에 대한 연산으로 나타내면 다음과 같다.

$$\mathbf{A} \rightarrow \mathbf{Av}^T = \mathbf{b}$$

$$\mathbf{A} = \begin{pmatrix} \frac{\partial f(y_1, x_1)}{\partial y} & \frac{\partial f(y_1, x_1)}{\partial x} \\ \vdots & \vdots \\ \frac{\partial f(y_n, x_n)}{\partial y} & \frac{\partial f(y_n, x_n)}{\partial x} \end{pmatrix}, \mathbf{v} = (v \ u), \mathbf{b} = \begin{pmatrix} -\frac{\partial f(y_1, x_1)}{\partial t} \\ \vdots \\ -\frac{\partial f(y_n, x_n)}{\partial t} \end{pmatrix}$$

$A$ 는 미지수  $v, u$ 의 계수  $\frac{df}{dy}, \frac{df}{dx}$ ,  $v$ 는  $v, u$ ,  $b$ 는  $\frac{df}{dt}$ 를 행렬로 나타낸 것이다.

- Normal Equation 을 이용해 Pixel 별 Motion Vector 계산

위 행렬식과 같이 미지수를 만족하는 수식이 미지수의 개수보다 많이 존재할 때 모든 수식을 만족하는 해( $v, u$ )는  $\mathbf{v}^T = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}$  Normal Equation 을 통해 도출할 수 있다.

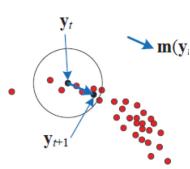
따라서 다음 방정식을 풀어 나온  $v, u$  값이 Lucas-Kanade Algorithm 을 이용하여 구한 해당 픽셀의 모션 벡터가 된다.

## ② Mean Shift Segmentation

- Current pixel & initial point 설정

Image에서 Mean Shift Algorithm 을 적용하게 되면 이미지의 각 pixel 이 하나의 data sample 이 된다. 이 때 Clustering 을 image 의 픽셀의 색상 값만 고려하여 진행하게 되면 k-means clustering 과 유사하기 때문에 pixel 의 색상 값 ( $r, g, b$ )과 위치 값 ( $y, x$ )를 함께 고려하기 위해 이 값들을 하나의 data 를 표현한다.

따라서 data  $x$  는  $x = (y, x, b, g, r)$  와 같이 1x5 vector 형태로 만들어진다.



Mean shift algorithm 에서는 다음 그림과 같이 현재 window 의 중심 좌표가  $y_t$  일 때 window 내 data 들의 mean 값을 계산하여 나온 좌표를  $y_{t+1}$ 로 설정하고 window 를  $y_{t+1}$ 이 중심좌표가 되도록 이동시킨다. 그리고 이러한 과정을 window 가 이동하는 값이 설정한  $\epsilon$ 값 보다 작아져 수렴할 때까지 반복하게 된다.

맨 처음  $t = 0$  일 때에는  $y_t$ 가 현재 픽셀 값인  $x = (y, x, b, g, r)$ 로 초기화된다.

- Gaussian Kernel 을 이용해 거리와 색상을 고려한 Window 의 중심좌표 계산

Window 내 data 들의 mean 값을 계산하기 위해 다음 수식을 이용하게 된다.

$$\mathbf{y}_{t+1} = \frac{\sum_{i=1}^n \mathbf{x}_i k\left(\frac{\mathbf{x}_i - \mathbf{y}_t}{h}\right)}{\sum_{i=1}^n k\left(\frac{\mathbf{x}_i - \mathbf{y}_t}{h}\right)}$$

이 수식을 보면 window 내의  $n$ 개의 data 들의 평균을 구할 때 Gaussian Kernel 함수  $k$  를 계산한 결과값을 각각의 data 에 가중치로 부여해 window 의 평균을 구하게 된다.

이 때 Gaussian Kernel 함수의 정의는 다음과 같다.

$$\text{Gaussian kernel: } k(\mathbf{x}) = \begin{cases} e^{-\frac{\|\mathbf{x}\|^2}{h^2}}, & \|\mathbf{x}\| \leq 1 \\ 0, & \|\mathbf{x}\| > 1 \end{cases}$$

$h$ : scale of kernel  
 $n$ : the number pixels in the current kernel  
 $y_t$ : current mode (mean)  
 $x_i$ : pixel value (feature)

Gaussian Kernel 함수로 전달되는 파라미터  $X$ 값은 현재 window 의 중심 좌표인  $y_t$ 와 현재 픽셀  $x_i$ 와의 거리 값을 나타낸다. 거리 값  $\|\mathbf{x}\| \leq 1$  일 때  $e^{-\frac{\|\mathbf{x}\|^2}{h^2}}$  을 계산하게 되는데, 이는 window 의 중심과 현재 픽셀 사이의 거리가 멀 경우 낮은 가중치를 부여하고, 거리가 가까울 경우 높은 가중치를 부여하여 window 의 mean 값을 계산함을 의미한다.

$$k(\mathbf{x}) = k\left(\frac{\mathbf{x}^s}{h_s}\right)k\left(\frac{\mathbf{x}^r}{h_r}\right)$$

Color Image 에서는 픽셀의 위치 값인  $x^s = (y, x)$ 와 픽셀의 색상 값인  $x^r = (b, g, r)$ 을 함께 고려해야 하기 때문에 Gaussian Kernel 을 위 수식과 같이 정의하여 mean 값을 도출해 낸다.

- Convergence 여부 확인 & Convergence Point 저장

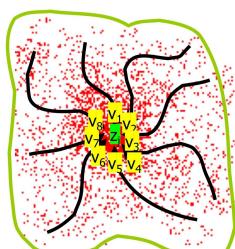
$$\begin{aligned} m(\mathbf{y}_t) &= \mathbf{y}_{t+1} - \mathbf{y}_t \\ &= \frac{\sum_{i=1}^n \mathbf{x}_i k\left(\frac{\mathbf{x}_i - \mathbf{y}_t}{h}\right)}{\sum_{i=1}^n k\left(\frac{\mathbf{x}_i - \mathbf{y}_t}{h}\right)} - \mathbf{y}_t \\ &= \frac{\sum_{i=1}^n (\mathbf{x}_i - \mathbf{y}_t) k\left(\frac{\mathbf{x}_i - \mathbf{y}_t}{h}\right)}{\sum_{i=1}^n k\left(\frac{\mathbf{x}_i - \mathbf{y}_t}{h}\right)} \end{aligned}$$

위에서 Gaussian Kernel 을 이용해 구한 새로운 window 의 중심 좌표  $y_{t+1}$  과 이전 단계 window 의 중심 좌표인  $y_t$ 와의 차이를 구한다.

이 값이  $\varepsilon$ 보다 클 경우  $y_{t+1}$  좌표가  $y_t$ 가 되어 다시 mean 값을 계산하게 되고  $\varepsilon$ 보다 작을 경우 계산을 멈추게 된다.

이후 Clustering 과정에서 사용하기 위해  $x_i$ 의 Convergence Point 인  $y_{t+1}$ 을 따로 저장해둔다.

- Clustering Convergence Points



다음 그림에서 빨간색 point 들이 각각의 data 를 나타낸다.  $v_i$ 값은 pixel 별로 mean 값 계산 후 window 를 이동시는 과정을 반복하여 수렴한 점을 나타낸다.

이제 clustering 을 진행하게 되는데, 거리를 고려한 kernel scale 인  $h_s$ 와 색상을 고려한 kernel scale 인  $h_r$  범위 내에 포함된 Convergence Points  $v_i$ 들을 같은 cluster 로 분류한다. 이후 Cluster Centroid  $z_i$ 를 구한 뒤 해당하는 pixel 에 Cluster 번호를 할당하게 된다.

- Merge Cluster

앞선 Clustering 과정 후, 인접 cluster의 centroid와 거리를 계산하여 일정 기준 이하일 경우 유사한 클러스터로 판단하여 cluster를 merge 한다.

### ③ Average Motion Vector of Segments

- Average Motion Vector of Segments

각 Cluster마다 할당된 pixel들의 motion vector 평균을 구하고 Cluster Centroid에서 시작되는 average motion vector를 표현한다.

## ii. Code description

### ① Optical Flow

- Frame 1에 대하여  $\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial t}$  을 계산

```
# dx, dy, dt 즉 변화량 계산
for x in range(width):
    for y in range(height):
        if x + 1 == width and y + 1 == height:
            dx[y, x] = 0 - frame1[y, x]
            dy[y, x] = 0 - frame1[y, x]
        elif x + 1 == width:
            dx[y, x] = 0 - frame1[y, x]
            dy[y, x] = frame1[y + 1, x] - frame1[y, x]
        elif y + 1 == height:
            dx[y, x] = frame1[y, x + 1] - frame1[y, x]
            dy[y, x] = 0 - frame1[y, x]
        else:
            dx[y, x] = frame1[y, x + 1] - frame1[y, x]
            dy[y, x] = frame1[y + 1, x] - frame1[y, x]
            dt[y, x] = frame2[y, x] - frame1[y, x]
```

padding 하여 미분 값을 계산하였다.

전체 픽셀에 대하여  $\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial t}$  을 계산하는 과정이다.  $\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}$  값의 경우 frame1 이미지에서 해당 픽셀과 인접하는 픽셀과의  $y, x$  변화량을 구하고  $\frac{\partial f}{\partial t}$  값의 경우 frame1 이미지의 픽셀에 대응하는 frame2 이미지의 픽셀과의 변화량을 구하게 된다.

이미지의 범위를 벗어나는 부분은 0 으로

- Lucas-Kanade Algorithm 을 이용해  $A\nu^T = b$  수식 도출

```
# 모든 픽셀에 대하여, motion vector 계산
window_size = 13
border = int(window_size / 2)
A = np.zeros((window_size ** 2, 2), dtype='float64')
b = np.zeros((window_size ** 2, 1), dtype='float64')
motion = np.zeros((height, width, 2), dtype='float64')
for x in range(border, width-border):
    for y in range(border, height-border):
        # window 구성
        # 행렬 A, b 생성
        idx = 0
        for i in range(-border, border+1):
            for j in range(-border, border+1):
                A[idx, 0] = dy[y+i, x+j]
                A[idx, 1] = dx[y+i, x+j]
                b[idx] = -dt[y+i, x+j]
                idx = idx + 1
```

Lucas-Kanade Algorithm 을 이용할 때 window size 를 다양하게 조절하고자 window\_size 변수 선언 했다.

행렬 A, b, motion 를 표현하기 위해 각각 사이즈에 맞춰 배열을 선언하고 0 으로 초기화 했다.

Window 의 영역이 이미지 사이즈를 벗어나는 pixel 은 고려하지 않았으므로

border 변수를 통해 반복문의 범위를 조절 (padding 을 적용하지 않고 motion vector 를 구함)

Window 내의 pixel 들에 대하여 미리 구해 둔  $\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial t}$  값을 행렬 A, b, motion 의 성분으로 저장하였다.

- Normal Equation 을 이용해 Pixel 별 Motion Vector 계산

```
for x in range(border, width-border):
    for y in range(border, height-border):
        # window 구성
        # 행렬 A, b 생성
        idx = 0
        for i in range(-border, border+1):...

        # motion 계산 - Normal Equation
        result = A.T
        result = result.dot(A)
        det = result[0, 0] * result[1, 1] - result[0, 1] * result[1, 0]
        if det != 0:
            result = np.linalg.inv(result)
        else:
            result = np.linalg.pinv(result)
        result = result.dot(A.T)
        result = result.dot(b)
        motion[y, x] = result.T
```

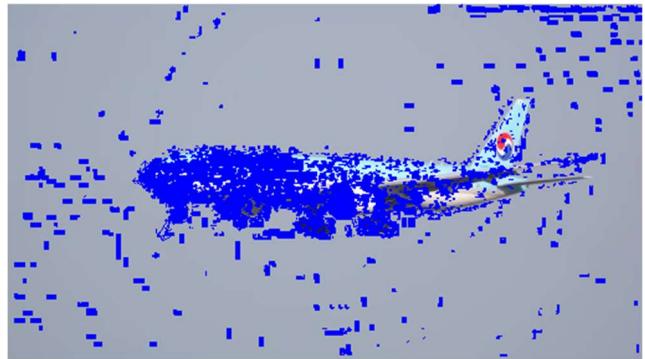
행렬 A 에 대하여  $A^T A$ 를 계산

행렬  $A^T A$  에 대하여 역행렬이 존재하면 역행렬을 바로 구하고 역행렬이 존재하지 않을 경우 (판별식의 분모 값이 0 이 된 경우) 유사역행렬을 통해 역행렬에 가장 인접한 값을 구했다.

모든 픽셀에 대하여 normal equation  $\nu^T = (A^T A)^{-1} A^T b$  계산 결과를 motion 배열에 저장하였다.

- Motion Vector 시각화

```
# motion vector 그리기
norm = np.sqrt(v[y, x, 0]**2 + v[y, x, 1]**2)
if norm > 0:
    cv2.arrowedLine(frame1_img, (x, y), (x + int(round(v[y, x, 0])), y + int(round(v[y, x, 1]))), (255, 0, 0), 1, tipLength=0.3)
```



위 코드처럼 motion vector 가 존재하는 모든 pixel 에 대하여 arrow를 그릴 경우 어느 위치에서 motion vector 가 구해졌는지는 파악할 수 있었으나, motion vector 의 수가 너무 많아 정확한 크기와 방향을 확인하기 어려웠다.

```
# motion vector 시각화
for x in range(3, width-3):
    for y in range(3, height-3):
        if x % 7 == 3 and y % 7 == 3:
            vx = 0
            vy = 0
            # 7*7 윈도우 내 벡터의 합
            for i in range(-3, 4):
                for j in range(-3, 4):
                    vy = vy + v[y + i, x + j, 0]
                    vx = vx + v[y + i, x + j, 1]

            if vy != 0 or vx != 0:
                # normalization
                arrow_size = 5
                norm = np.sqrt(vy ** 2 + vx ** 2)
                ny = int(round(arrow_size * (vy / norm)))
                nx = int(round(arrow_size * (vx / norm)))
                if nx > 0 or ny > 0:
                    cv2.arrowedLine(frame1_img, (x, y), (x + nx, y + ny), (255, 0, 0), 1, tipLength=0.5)

            # resize
            resize = 15
            iy = int(round((vy / 49) * resize))
            ix = int(round((vx / 49) * resize))
            if ix > 0 or iy > 0:
                cv2.arrowedLine(interpolation_img, (x, y), (x + ix, y + iy), (255, 0, 0), 1, tipLength=0.3)
```

따라서  $7 \times 7$  size 의 원도우를 만들어, 원도우 내에 존재하는 pixel 들의 motion vector 의 합을 구해 window 단위로 1 개의 motion vector 를 표현하여 시각화 문제를 개선하였다.

이때 2 가지의 이미지로 motion vector 를 표현하였는데, window 별 motion vector 의 방향을 확인하기 위한 이미지에는 motion vector 를 normalization 하여 동일한 크기로 그렸다.

또한 window 별 motion vector 의 크기 확인을 위한 이미지에는 window 내의 motion vector 들의 평균을 구하여 size 를 재 조정하여 표현하였다.



Motion vector 방향 확인을 위해 normalization 하여 표현한 이미지

Motion vector 크기 확인을 위해 resize 하여 표현

## ② Mean Shift Segmentation

- 기본 Parameter 및 배열 선언

```
# 5차원 벡터 생성
x_vec = np.zeros(5, dtype='float64')
y_vec = np.zeros(5, dtype='float64')
shift_vec = np.zeros(5, dtype='float64')
v = np.zeros((height, width, 5), dtype='float64')

# Scale of Kernel hs, hr
hs = 40
hr = 30

# parameter min_shift
epsilon = 2
```

현재 픽셀  $x = (y, x, b, g, r)$  값을 저장할 벡터  $x\_vec$ 과 window의 중심좌표를 나타내는  $y\_vec$ ,  $y_{t+1}$ 과  $y_t$ 의 차이를 나타내는  $shift\_vec$  그리고 마지막으로 Convergence Point 를 나타내는  $v$ 를 선언하고 0 으로 초기화하였다.

- Current pixel & initial point 설정

```
for x in range(0, width):
    for y in range(0, height):
        # 현재 pixel의 vector x 값 초기화
        x_vec[0] = y
        x_vec[1] = x
        x_vec[2] = frame1[y, x, 0]
        x_vec[3] = frame1[y, x, 1]
        x_vec[4] = frame1[y, x, 2]

        # initial y 설정
        y_vec = x_vec
```

모든 픽셀에 대하여 mean shift 과정을 진행하기에 앞서, 현재 대상 pixel의 위치 좌표  $y$ ,  $x$  와 색상 값  $b$ ,  $g$ ,  $r$  을  $x\_vec$  에 초기화하여  $1 \times 5$  형태의 벡터를 생성하였다.

초기 window 의 중심좌표는 현재 고려중인 픽셀이므로  $y_{t+1} = x_i$ 를 나타내고자  $y\_vec = x\_vec$  으로 initial point 를 설정하였다.

- Gaussian Kernel 을 이용해 거리와 색상을 고려한 Window 의 중심좌표 계산

```
# density가 가장 높은곳 찾기
curr_vec = np.zeros(5, dtype='float64')
while True:
    numerator = np.zeros(5, dtype='float64')
    denominator = 0
    for i in range(0, width):
        for j in range(0, height):
            # calculate distance s, r → kernel 내 pixel일 경우 연산
            s_dist = np.sqrt((j - y_vec[0]) ** 2 + (i - y_vec[1]) ** 2)
            if s_dist < hs:
                r_dist = np.sqrt((frame1[j, i, 0] - y_vec[2]) ** 2 +
                                  (frame1[j, i, 1] - y_vec[3]) ** 2 +
                                  (frame1[j, i, 2] - y_vec[4]) ** 2)
                if r_dist < hr:
```

현재 window 의 중심 좌표  $y_t$  와 다른 pixel 의 위치, 색상 거리를 계산해 window 내에 존재하는 pixel 인지 판단하는 과정이다.

window 의 중심  $y\_vec$  과 비교 대상 픽셀인  $(j, i)$  번째

픽셀과의 위치상 거리  $s\_dist$  와 색상 좌표 거리  $r\_dist$  를 계산하여 각각의 kernel scale  $h_s, h_r$  보다 작을 경우 window 내의 pixel 로 판단하여 다음 과정을 진행한다.

```
# current pixel값 설정
curr_vec[0] = j
curr_vec[1] = i
curr_vec[2] = frame1[j, i, 0]
curr_vec[3] = frame1[j, i, 1]
curr_vec[4] = frame1[j, i, 2]

# ks 값 구하기
ks_param = (curr_vec - y_vec) / hs
l2_norm = np.sqrt(ks_param[0] ** 2 + ks_param[1] ** 2)
if l2_norm <= 1:
    ks = np.exp(-(l2_norm**2))
else:
    ks = 0

# kr 값 구하기
kr_param = (curr_vec - y_vec) / hr
l2_norm = np.sqrt(kr_param[2] ** 2 + kr_param[3] ** 2 + kr_param[4] ** 2)
if l2_norm <= 1:
    kr = np.exp(-(l2_norm**2))
else:
    kr = 0

# k 곱하기
k = ks * kr
numerator = numerator + k * curr_vec
denominator = denominator + k
```

$(j, i)$  번째 pixel 은 현재 window 내의 data 이므로 해당 픽셀을  $1 \times 5$  vector 형태 curr\_vec 로 초기화했다.

다음은 중심  $y_t$  와 현재 좌표의 거리를 계산해 Gaussian Kernel 의 Parameter 로 넘기는 과정이다.

$$y_{t+1} = \frac{\sum_{i=1}^n \mathbf{x}_i k\left(\frac{\mathbf{x}_i - \mathbf{y}_t}{h}\right)}{\sum_{i=1}^n k\left(\frac{\mathbf{x}_i - \mathbf{y}_t}{h}\right)}$$

Gaussian kernel:  $k(x) = \begin{cases} e^{-\frac{\|x\|^2}{h}}, & \|x\| \leq 1 \\ 0, & \|x\| > 1 \end{cases}$

다음 수식에서 ks\_param, kr\_param 은  $\frac{x_i - y_t}{h}$  를 나타낸 변수이며 이 값의  $L_2 norm$  을 계산하여 1 보다 작거나 같을 경우  $e^{-\|x\|^2}$  를 계산, 1 보다 클 경우 0 으로  $k_s, k_r, k$ , 값을 각각 계산하였다.

마지막으로  $k(x) = k(\frac{x^s}{h_s})k(\frac{x^r}{h_r})$  수식에 따라 ks 와 kr 을 곱해주었으며, 분모와 분자는 window 내 픽셀들에 gaussian 가중치를 곱한 결과의 시그마 값이므로 각각 numerator, denominator 변수에 누적하여 저장하였다.

- Convergence 여부 확인 & Convergence Point 저장

```
next_y_vec = numerator / denominator
shift_vec = next_y_vec - y_vec
l2_norm = np.linalg.norm(shift_vec)
if l2_norm < epsilon:
    v[y, x] = next_y_vec
    break;
else:
    y_vec = next_y_vec
```

window 내에 존재하는 모든 pixel 들에 대해 가중치 값  $k$ 를 계산한 뒤 누적된 분모 분자를 분수로 표현하여  $y_{t+1}$  을 도출한 변수가 next\_y\_vec 이다.

다음으로 이동한 window 와 이전 window 의 중심 값의 차이를 계산한 것이 shift\_vec 이며 shift\_vec 의  $L_2 norm$ 이  $\varepsilon$ 보다 작을 경우 수렴하였다고 판단하여 Convergence point  $v_i$ 에  $y_{t+1}$  값을 저장하였다.

반대로  $\varepsilon$ 보다 클 경우  $y_t = y_{t+1}$  이 되어 window 를 이동한 뒤 다시 while 문 처음 과정부터 반복하게 된다.

- Clustering Convergence Points

```
# clustering vector v
img_size = width * height
assigned_cluster = np.full((height, width), -1, dtype='int64')
c_centroid = []
c_points_num = []
cluster_num = -1
```

각 픽셀별로 할당된 cluster 번호를 저장하고자 assigned\_cluster 를 생성하여 -1 로 초기화하였다.

c\_centroid는 cluster들의 중심점을 나타내는 배열이며 c\_points\_num 은 cluster 별 포함된 pixel 의 개수, cluster\_num 은 현재 clustering 중인 cluster 의 번호를 나타내기 위한 변수로 사용하였다.

```
for i in range(img_size):
    w = int(i / height)
    h = int(i % height)
    # 클러스터링 안된 pixel 찾기 -> 새로운 cluster 생성
    if assigned_cluster[h, w] == -1:
        cluster_num = cluster_num + 1
        assigned_cluster[h, w] = cluster_num
        c_points_num.append(1)
        c_centroid.append(v[h, w])
    # 새로운 cluster에 포함되는 pixel 찾기
```

pixel 을 순차적으로 순회하며 현재 pixel이 clustering 되지 않은 pixel 이면 새로운 cluster 의 centroid 가 되도록 구현하였다.

즉 현재 픽셀의 assigned\_cluster 가 -1 이면 clustering 되지 않은 픽셀이므로 새로운 cluster 를 형성하게 된다. cluster\_num 이 1 증가한 뒤 현재 픽셀이 할당된 cluster 번호를 cluster\_num 으로 설정하고 cluster 에 포함된 현재 픽셀의 개수는 1 개, 중심 좌표는 현재 pixel 의 Convergence Point 로 초기화한다.

이후 0| cluster에 포함된 pixel 들을 찾는 과정을 진행한다.

```
# 새로운 cluster에 포함되는 pixel 찾기
for j in range(i + 1, img_size):
    target_w = int(j / height)
    target_h = int(j % height)
    if assigned_cluster[target_h, target_w] == -1:
        # s, r 거리 계산 → 인접 pixel clustering
        s_dist = np.sqrt((v[target_h, target_w, 0] - v[h, w, 0]) ** 2 +
                         (v[target_h, target_w, 1] - v[h, w, 1]) ** 2)
        if s_dist < hs:
            r_dist = np.sqrt((v[target_h, target_w, 2] - v[h, w, 2]) ** 2 +
                             (v[target_h, target_w, 3] - v[h, w, 3]) ** 2 +
                             (v[target_h, target_w, 4] - v[h, w, 4]) ** 2)
            if r_dist < hr:
                assigned_cluster[target_h, target_w] = c_num
                c_points_num[c_num] = c_points_num[c_num] + 1
                c_centroid[c_num] = c_centroid[c_num] + v[target_h, target_w]

# cluster centroid 계산
c_centroid[c_num] = c_centroid[c_num] / c_points_num[c_num]
```

전 단계에서 새로 생성한 cluster에 포함되는 pixel을 찾는 과정으로 clustering 되지 않은 pixel 들에 대하여 조건 검사를 한다.

새로 생긴 cluster의 cluster\_centeroid 와 현재 픽셀의 위치상 거리 s\_dist 와 color 좌표 거리 r\_dist를 계산하여 인접하는 pixel 일 경우 cluster에 추가하게 된다.

마찬가지로 새로 cluster에 추가되는 pixel에 cluster 번호 c\_num을 할당하고, cluster에 포함된 pixel의 개수를 1개 증가시키며 이후 cluster centroid를 계산하기 위해 각 pixel의 Convergence Point(v)를 누적하여 저장하였다.

```
# cluster centroid 계산
c_centroid[c_num] = c_centroid[c_num] / c_points_num[c_num]
```

마지막으로 해당 cluster에 모든 pixel이 추가되면 누적된 Convergence Point를 cluster 포함된 pixel의 개수로 나누어 cluster centroid를 계산하였다.

### ● Merge Cluster

교안에서는 cluster에 포함된 pixel의 개수가 기준치보다 작을 때 인접 cluster와 병합시켰다. 하지만 cluster centroid를 계산하였을 때 인접 cluster의 centroid와 가까운 cluster들이 존재하는 것을 확인하여 교안의 방법을 조금 변형하였다.

인접 cluster 와의 위치상 거리와 색상 좌표 값 거리가 특정 기준치보다 작을 경우 두개의 cluster를 유사한 cluster로 분류하여 병합하도록 구현해보았다.

```
# Merge Cluster
assigned = np.full(len(c_centroid), -1, dtype='int64')
cluster = []
cluster_num = []
c_num = -1
ds = 30
dr = 30
```

cluster 에 포함된 기존 cluster 의 개수 c\_num 은 merge 중인 cluster 의 번호를 나타낸다.

```
for i in range(len(c_centroid)):
    # 기존 cluster를 인접 cluster와 merge
    if assigned[i] == -1:
        c_num = c_num + 1
        assigned[i] = c_num
        cluster_num.append(i)
        cluster.append(c_centroid[i])
    # 새로운 cluster에 포함되는 cluster 찾기
    for j in range(i + 1, len(c_centroid)):
        if assigned[j] == -1:
            # s, r 거리 계산 → 인접 cluster끼리 clustering
            s_dist = np.sqrt((c_centroid[j][0] - c_centroid[i][0]) ** 2 +
                             (c_centroid[j][1] - c_centroid[i][1]) ** 2)
            if s_dist < ds:
                r_dist = np.sqrt((c_centroid[j][2] - c_centroid[i][2]) ** 2 +
                                 (c_centroid[j][3] - c_centroid[i][3]) ** 2 +
                                 (c_centroid[j][4] - c_centroid[i][4]) ** 2)
                if r_dist < dr:
                    assigned[j] = c_num
                    cluster_num[c_num] = cluster_num[c_num] + 1
                    cluster[c_num] = cluster[c_num] + c_centroid[j]

    # cluster centroid 계산
    cluster[c_num] = cluster[c_num] / cluster_num[c_num]
```

## ● Segments 표현

이미지에서 같은 segment 에 포함되는 pixel 의 값은 해당 segment 의 cluster centroid 값으로 대체하였다.

Cluster centroid 의 값으로만 segment 를 구분할 경우 유사한 색상의 다른 segment 를 확인하기 모호한 부분이 있어 segment 별 edge 를 확인할 수 있도록 추가하였다.

```
# 클러스터 centroid값으로 세그멘테이션 표현
for i in range(width):
    for j in range(height):
        for k in range(3):
            frame1_img[j,i,k] = cluster[assigned[assigned_cluster[j,i]]][k+2]
```

다음은 각 pixel 별로 할당된 cluster 의 centroid 값으로 픽셀 값을 대체하는 과정이다.

```
#edge 표현
for i in range(1, width-1):
    for j in range(1, height-1):
        if (assigned[assigned_cluster[j, i]] != assigned[assigned_cluster[j - 1, i]]) or \
            (assigned[assigned_cluster[j, i]] != assigned[assigned_cluster[j, i - 1]]):
                frame1_img[j, i, 0] = 0
                frame1_img[j, i, 1] = 0
                frame1_img[j, i, 2] = 0
```

0 으로 설정하여 edge 를 나타내는 과정이다.

assigned 는 기존 cluster 가 병합되어 새롭게 할당될 cluster 의 번호를 나타내고, cluster 배열은 병합된 cluster 의 centroid, cluster\_num 은

다음은 앞에서 Convergence Point 를 clustering 했던 flow 와 유사하다.

즉 인접 cluster 와의 거리를 계산하여 기준치보다 작을 경우 merge 하여 새로운 cluster 를 형성하는 것이다.

다음은 추가적으로 인접 픽셀이 다른 cluster 에 포함될 때 해당 pixel 값을

### ③ Average Motion Vector of Segments

- Average Motion Vector of Segments

```
# segment별 motion vector 표현
length = len(cluster)
segment_motion = np.zeros((length,2), dtype='float32')
pixel_num = np.zeros(length, dtype='float32')
for x in range(width):
    for y in range(height):
        number = assigned[assigned_cluster[y,x]]
        segment_motion[number] = segment_motion[number] + motion[y, x]
        pixel_num[number] = pixel_num[number] + 1
```

length 는 cluster 개수, segment\_motion 은 segment 별 motion vector 의 누적 값, pixel\_num 은 segment 별 할당된 픽셀의 수를 나타낸다.

전체 픽셀을 순회하면서 현재 픽셀이 할당된 cluster 의 번호를 반환하고, 현재 픽셀의 motion vector 값을 해당하는 segment\_motion 배열에 누적시킨다. 이후 해당 cluster 에 할당된 pixel\_num 을 +1 해주었다.

```
for i in range(length):
    center_y = int(cluster[i][0])
    center_x = int(cluster[i][1])
    segment_motion[i] = segment_motion[i] / pixel_num[i]
    dy = int(segment_motion[i][0] * 10)
    dx = int(segment_motion[i][1] * 10)
    cv2.arrowedLine(frame1_img, (center_x, center_y), (center_x + dx, center_y + dy), (255, 0, 0), 1, tipLength=0.5)
```

모든 픽셀을 순회하여 클러스터 별로 motion vector 누적합을 구하는 과정이 끝난 뒤 segment 별 average motion vector 를 도출하는 과정이다.

모든 cluster를 순회하며 cluster별 누적 motion vector를 cluster에 포함된 pixel의 개수로 나누어 average motion vector 를 도출하였다.

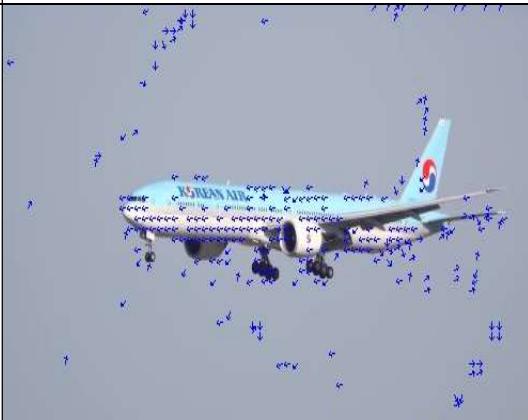
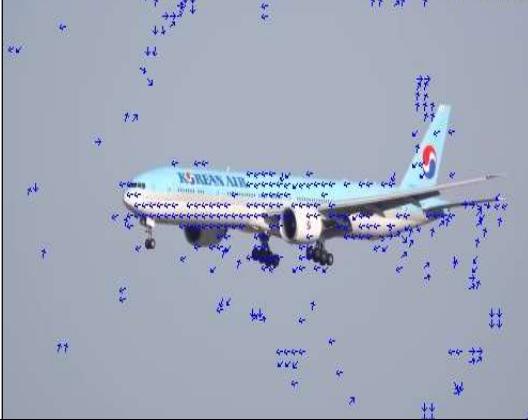
이후 arrowedLine 함수를 통해 motion vector 를 시각화 하였다.

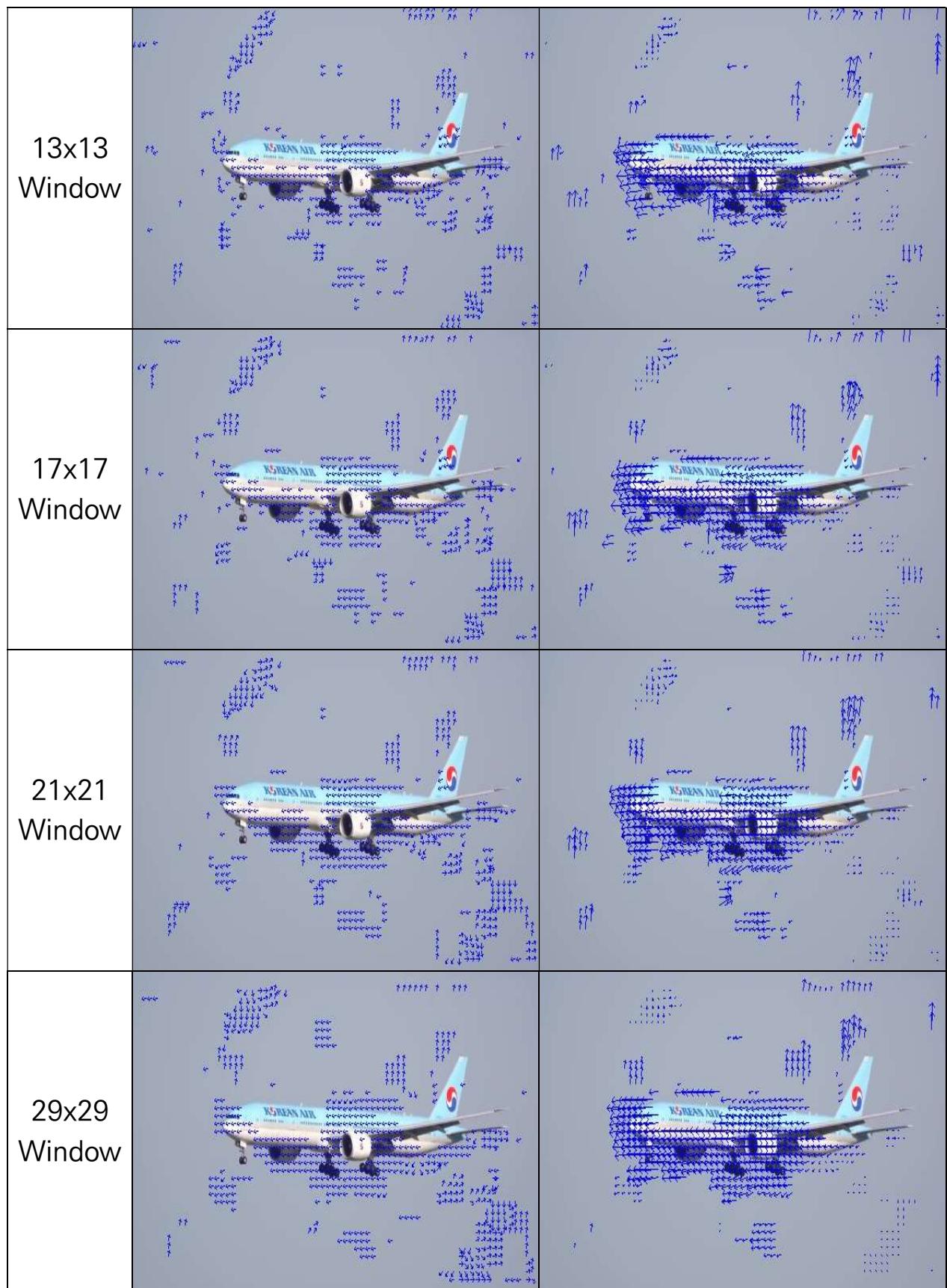
## 3. Result Analysis

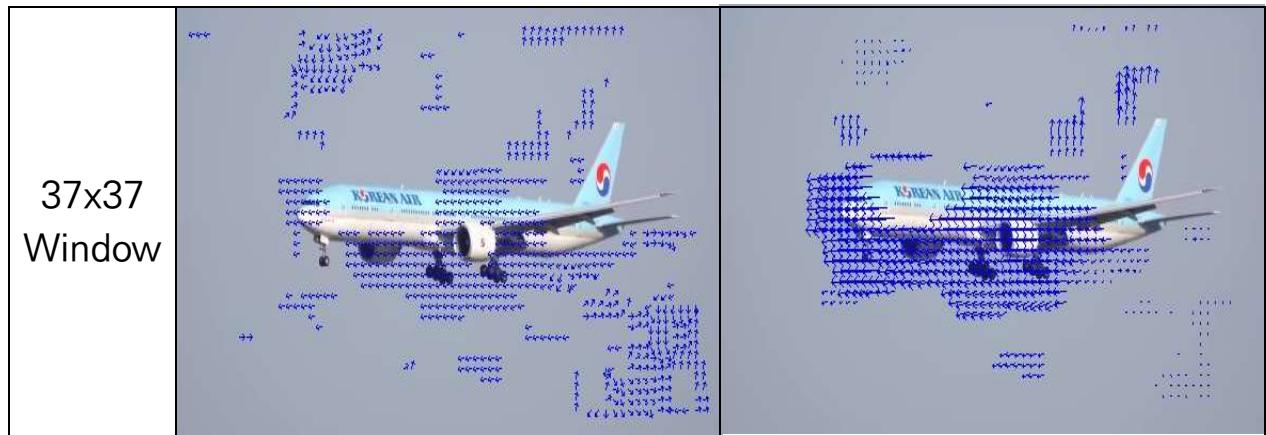
### A. Evaluate Accuracy

#### ① Optical Flow

Optical Flow 부분에서는 Lucas-Kanade Algorithm 을 적용할 때 window size 의 크기 값을 나타내는 parameter 를 다양하게 조절해 분석하고 표로 정리하였다.

	Normalization Image	Resize Image
3x3 Window		
5x5 Window		
9x9 Window		





우선 Window size 가 작을 때에는 해당 pixel 의 motion vector 를 구하는 과정에서 고려하게 되는 주변 픽셀수가 적기 때문에 전체적으로 이미지에 도출된 모션벡터의 수가 적으며, Window size 가 증가함에 따라 window내에 motion이 발생할 확률이 높아지기 때문에 전체적으로 도출되는 모션벡터의 수가 증가함을 확인할 수 있었다.

두번째로 window size 가 작을 때에는 고려되는 범위가 작으므로 motion vector 의 방향이 부정확한 부분들이 많았지만, window size 가 증가하면서 motion vector 의 방향이 정확해지고 인접 pixel 의 motion vector 와의 방향도 균일해지는 모습이 확인되었다.

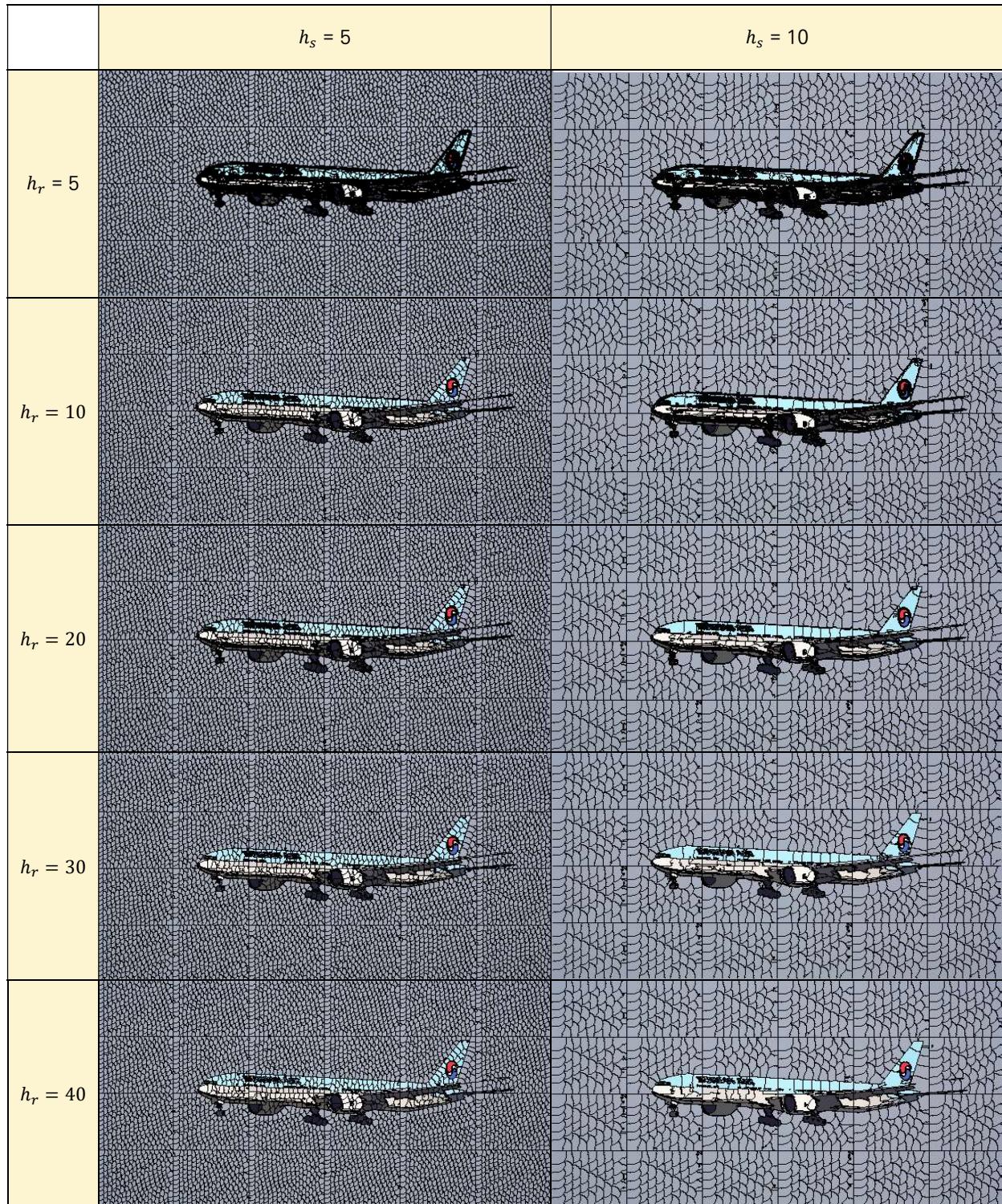
마지막으로 window size 가 작을 때에는 motion vector 의 크기가 큰 pixel 들이 많았지만 window size 가 커질수록 크기가 대체적으로 작아졌으며, 비행기를 제외한 배경 부분에서 미세하게 움직였던 구름의 motion vector 도 잘 도출되었음을 확인했다.

## ② Mean Shift Segmentation

Window 가 Shift 한 이동량의 convergence 여부를 판단하는 parameter 인  $\varepsilon$ 은 2 로 고정하고  $h_s$ 와  $h_r$  parameter 를 다양하게 조절하며 현재 input image 에 적합한 파라미터를 찾아보았다.

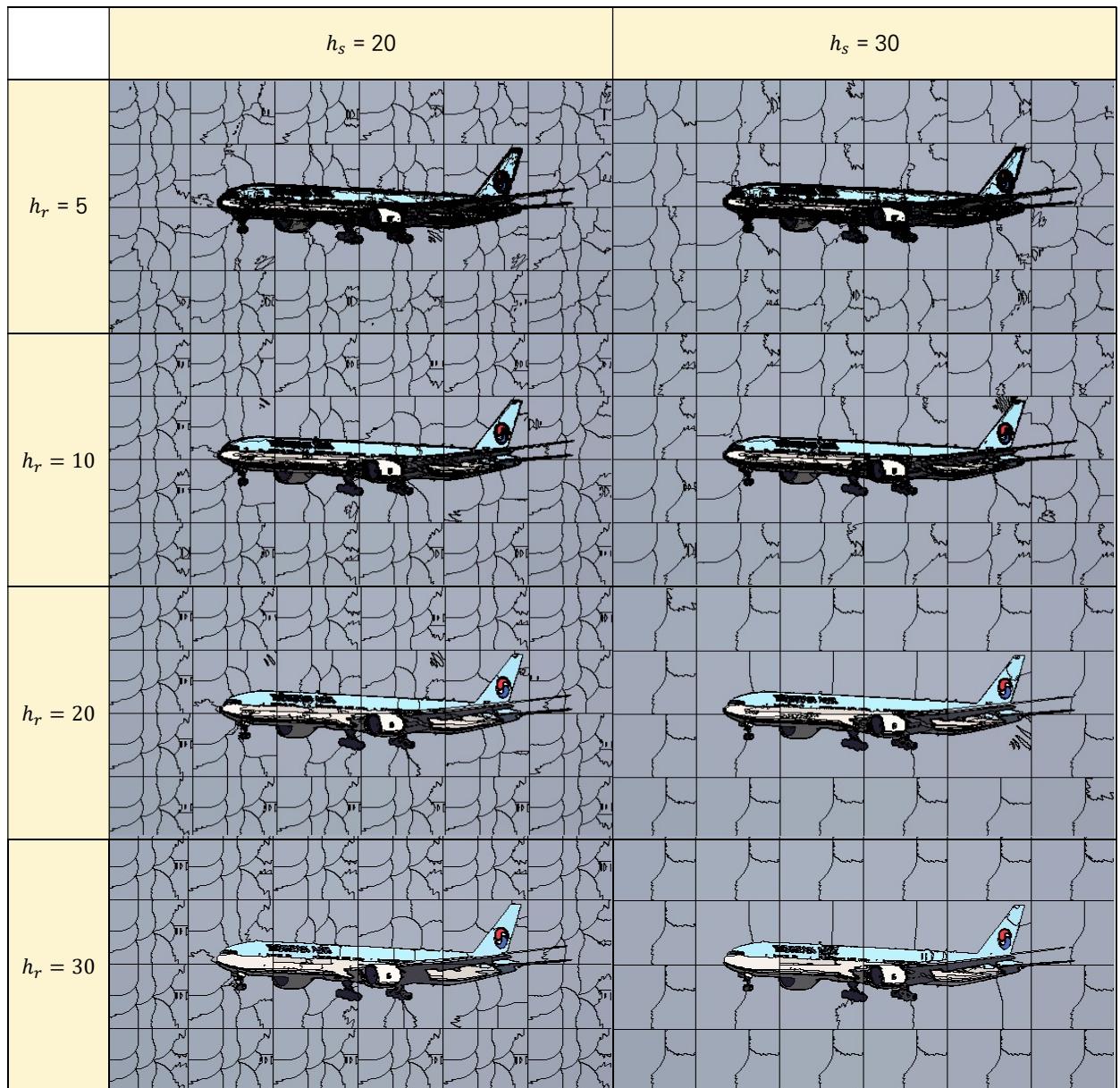
이때 한번에 전체 이미지를 고려하기에는 시간이 오래 걸려서, 파라미터를 찾는 시간을 조금이나마 절약하고자 이미지를 24 개의 box 로 분할하여 test 하였다.

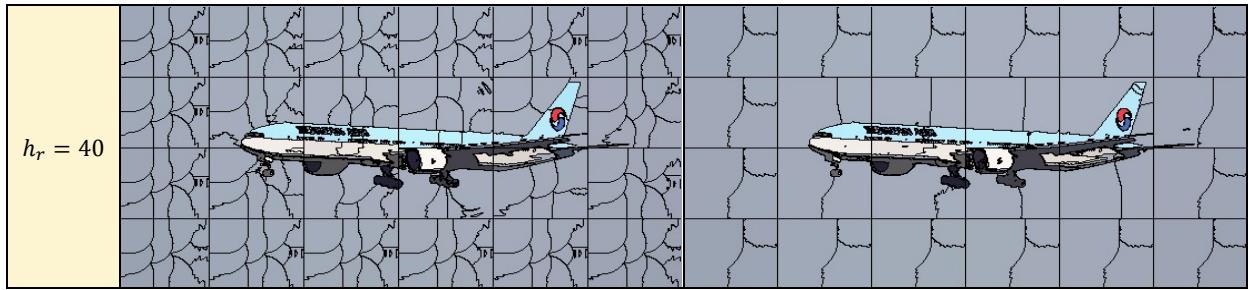
각 box 와 전체적인 이미지에서 파라미터 증가에 따른 clustering 정확도 여부를 어느정도 판단한 뒤 적절한 파라미터를 찾아내고 전체 이미지에 대해 clustering 을 진행해 보았다.



우선  $h_s = 5, 10$  일 때는 이미지 사이즈에 비해 거리상 너무 작은 범위의 pixel 들을 고려하다 보니 cluster 로 묶일 수 있는 범위가 너무 작았다. 따라서 매우 많은 수의 cluster 가 형성되었기 때문에 clustering 이 정확하게 이루어지지 않음을 확인할 수 있었다.

또한 24 개로 분할한 박스 내에서도 정확히 clustering 이 이루어지지 않음을 확인할 수 있었다.



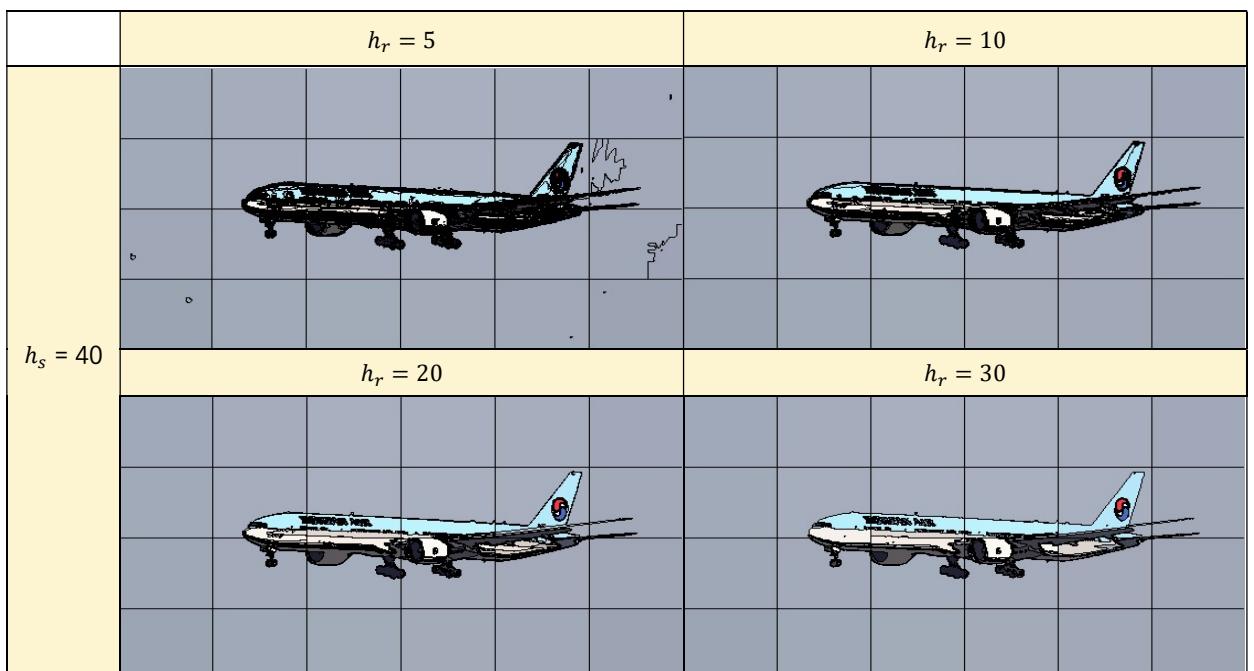


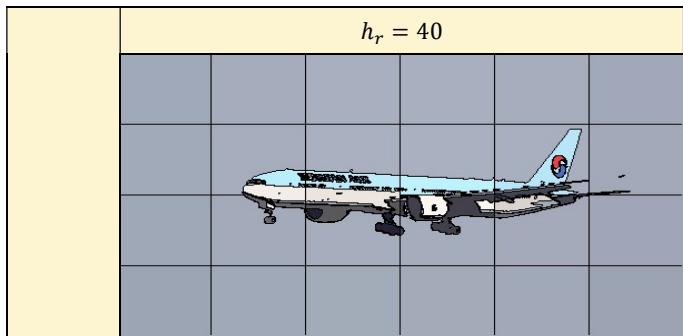
다음으로  $h_s = 20, 30$ 로 설정하였을 때의 결과이다. 이전  $h_s = 5, 10$  일 때와는 확실히 배경 부분에 대해서 점점 더 큰 범위로 clustering 되는 것을 확인할 수 있었다.

$h_s = 30$  으로 설정하였을 때 배경부분을 살펴보면 24 개의 박스 내에서 일반적으로 3 개의 cluster 로 나뉘어짐을 확인할 수 있었다. 하지만 여전히 전체 이미지를 clustering 하는 경우를 고려하면  $h_s = 30$  은 부족하였다.

$h_s$  값이 증가함에 따라 시각적으로 비행기 객체가 잘 보이게 되어  $h_r$  parameter 에 대해서는 적절한 파라미터를 찾을 수 있었다.  $h_r = 5, 10, 20$  일 때는 비행기 객체 내부에서 수많은 cluster 로 나뉘어 졌음을 확인했다. 이에 비해  $h_r = 30$ 인 경우 비행기 객체 내에서 clustering 이 대체적으로 잘 이뤄졌음을 확인할 수 있었다. 하지만  $h_r = 40$  까지 증가시키자 비행기의 큰 날개부분이 배경과 같은 cluster 로 분류되어 중간에 잘리는 모습을 확인할 수 있었다.

따라서 현재 이미지에 대한 적절한  $h_r$  parameter 값을 30~40 정도로 결정할 수 있었다.



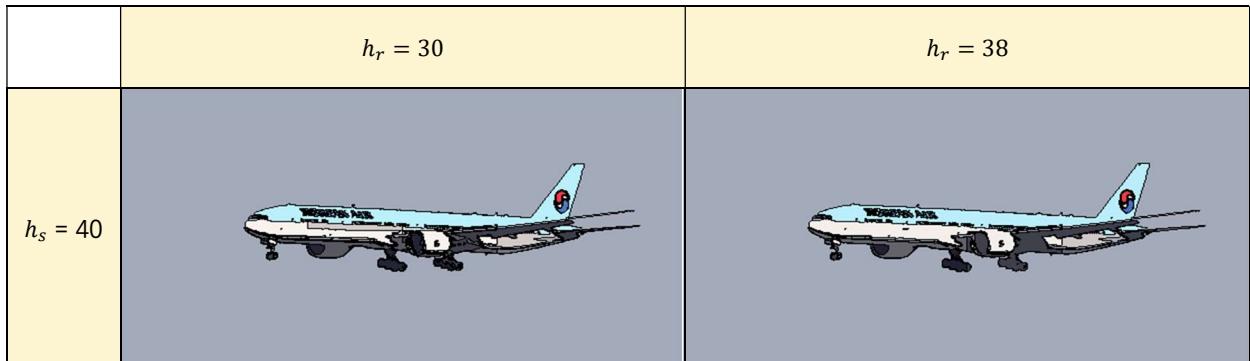


마지막으로  $h_s$  parameter 를 40 으로 설정했을 때의 결과이다. 전체적으로 분할했던 박스 내부에 배경이 하나의 cluster 로 clustering 되었음을 확인할 수 있었다.

또한  $h_s=30$ ,  $h_r=30$  이미지에서 비행기 상단의 Korean air 글씨 부분에서 정확하게 clustering 되지 않는 모습을 확인할 수 있었는데  $h_s=40$ ,  $h_r=30$  일 때의 이미지를 보면 Korean air 글씨 부분도 대체적으로 잘 clustering 된 것을 확인하였다.

마찬가지로  $h_r=40$  을 넘어가면 비행기의 날개부분이 하늘과 같은 cluster 로 분류되었다.

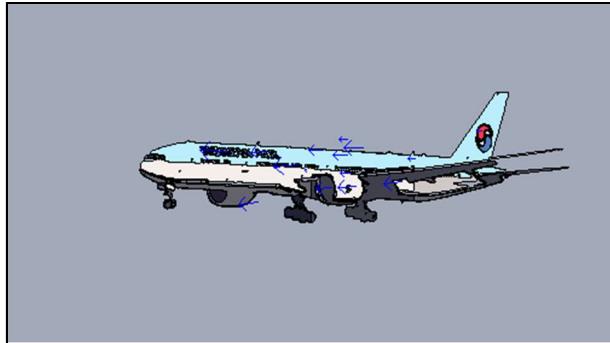
따라서 전체 이미지에 대하여  $h_s$ 는 40 이상으로,  $h_r$ 은 30~40 정도로 설정하여 clustering 하도록 결정하였다.



마지막으로 전체 이미지에 대해  $h_s = 40$ ,  $h_r$ 은 각각 30, 38 을 적용한 뒤 merge cluster 까지 적용한 결과 이미지이다.  $h_r = 38$ 일 때 가장 잘 clustering 되었다.

전체 이미지에 대해 clustering 을 진행하였을 때 비행기를 제외한 하늘부분이 몇개의 cluster 로 나뉘어 졌지만 merge clustering 과정에서 고려하는 거리 값을 늘리자 하나로 clustering 되었다.

### ③ Average Motion Vector of Segments



다음 사진이 최종 clustering 된 이미지에 cluster centroid 별로 평균 모션 벡터를 표현한 사진이다. 하늘 부분 segment에는 대체적으로 motion 이 없고, 비행기 object에 대부분의 motion 이 있었기 때문에 비행기 부분에 있는 segments에서 대부분의 motion vector 가 도출됨을 확인할 수 있었다.

## B. Code Operation Process

구체적인 코드 내부의 동작 과정 및 의미는 이미 앞에서 설명하였기 때문에 코드의 구체적인 설명은 하지 않고 전체 코드의 흐름을 파악할 수 있도록 코드 이미지를 캡처하여 첨부했다.

### ① Optical Flow

- Image gradient 계산 & Motion Vector 추출

```
# dx, dy, dt 즉 변화량 계산
for x in range(width):
    for y in range(height):
        if x + 1 == width and y + 1 == height:
            dx[y, x] = 0 - frame1[y, x]
            dy[y, x] = 0 - frame1[y, x]
        elif x + 1 == width:
            dx[y, x] = 0 - frame1[y, x]
            dy[y, x] = frame1[y + 1, x] - frame1[y, x]
        elif y + 1 == height:
            dx[y, x] = frame1[y, x + 1] - frame1[y, x]
            dy[y, x] = 0 - frame1[y, x]
        else:
            dx[y, x] = frame1[y, x + 1] - frame1[y, x]
            dy[y, x] = frame1[y + 1, x] - frame1[y, x]
            dt[y, x] = frame2[y, x] - frame1[y, x]

# 모든 픽셀에 대하여, motion vector 계산
window_size = 13
border = int(window_size / 2)
A = np.zeros((window_size ** 2, 2), dtype='float64')
b = np.zeros((window_size ** 2, 1), dtype='float64')
motion = np.zeros((height, width, 2), dtype='float64')
for x in range(border, width-border):
    for y in range(border, height-border):
        # window 구성
        # 행렬 A, b 생성
        idx = 0
        for i in range(-border, border+1):
            for j in range(-border, border+1):
                A[idx, 0] = dy[y+i, x+j]
                A[idx, 1] = dx[y+i, x+j]
                b[idx] = -dt[y+i, x+j]
                idx = idx + 1

        # motion 계산 - Normal Equation
        result = A.T
        result = result.dot(A)
        det = result[0, 0] * result[1, 1] - result[0, 1] * result[1, 0]
        if det != 0:
            result = np.linalg.inv(result)
        else:
            result = np.linalg.pinv(result)
        result = result.dot(A.T)
        result = result.dot(b)
        motion[y, x] = result.T
```

## ② Mean Shift Segmentation

### ● Mean Shift

```
# Model Seeking
for x in range(0, width):
    for y in range(0, height):
        # 현재 pixel의 vector x 값 초기화
        x_vec[0] = y
        x_vec[1] = x
        x_vec[2] = frame1[y, x, 0]
        x_vec[3] = frame1[y, x, 1]
        x_vec[4] = frame1[y, x, 2]

        # initial y 설정
        y_vec = x_vec

        # density가 가장 높은곳 찾기
        curr_vec = np.zeros(5, dtype='float64')
        while True:
            numerator = np.zeros(5, dtype='float64')
            denominator = 0
            for i in range(0, width):
                for j in range(0, height):
                    # calculate distance s, r -> kernel  $\frac{1}{s}$  pixel일 경우 연산
                    s_dist = np.sqrt((j - y_vec[0]) ** 2 + (i - y_vec[1]) ** 2)
                    if s_dist < hs:
                        r_dist = np.sqrt((frame1[j, i, 0] - y_vec[2]) ** 2 +
                                         (frame1[j, i, 1] - y_vec[3]) ** 2 +
                                         (frame1[j, i, 2] - y_vec[4]) ** 2)
                        if r_dist < hr:
                            # current pixel값 설정
                            curr_vec[0] = j
                            curr_vec[1] = i
                            curr_vec[2] = frame1[j, i, 0]
                            curr_vec[3] = frame1[j, i, 1]
                            curr_vec[4] = frame1[j, i, 2]

                            # ks 값 구하기
                            ks_param = (curr_vec - y_vec) / hs
                            l2_norm = np.sqrt(ks_param[0] ** 2 +
                                              ks_param[1] ** 2)

                            if l2_norm <= 1:
                                ks = np.exp(-(l2_norm**2))
                            else:
                                ks = 0

                            # kr 값 구하기
                            kr_param = (curr_vec - y_vec) / hr
                            l2_norm = np.sqrt(kr_param[2] ** 2 +
                                              kr_param[3] ** 2 +
                                              kr_param[4] ** 2)

                            if l2_norm <= 1:
                                kr = np.exp(-(l2_norm**2))
                            else:
                                kr = 0

                            # k 곱하기
                            k = ks * kr
                            numerator = numerator + k * curr_vec
                            denominator = denominator + k

            # check convergence
            next_y_vec = numerator / denominator
            shift_vec = next_y_vec - y_vec
            l2_norm = np.linalg.norm(shift_vec)
            if l2_norm < epsilon:
                v[y, x] = next_y_vec
                break
            else:
                y_vec = next_y_vec
```

## ● Clustering Convergence Vector

```
# clustering vector v
img_size = width * height
assigned_cluster = np.full((height, width), -1, dtype='int64')
c_centroid = []
c_points_num = []
c_num = -1
for i in range(img_size):
    w = int(i / height)
    h = int(i % height)
    # 클러스터링 안된 pixel 찾기 -> 새로운 cluster 생성
    if assigned_cluster[h, w] == -1:
        c_num = c_num + 1
        assigned_cluster[h, w] = c_num
        c_points_num.append(1)
        c_centroid.append(v[h, w])
    # 새로운 cluster에 포함되는 pixel 찾기
    for j in range(i + 1, img_size):
        target_w = int(j / height)
        target_h = int(j % height)
        if assigned_cluster[target_h, target_w] == -1:
            # s, r 거리 계산 -> 인접 pixel clustering
            s_dist = np.sqrt((v[target_h, target_w, 0] - v[h, w, 0]) ** 2 +
                             (v[target_h, target_w, 1] - v[h, w, 1]) ** 2)
            if s_dist < hs:
                r_dist = np.sqrt((v[target_h, target_w, 2] - v[h, w, 2]) ** 2 +
                                 (v[target_h, target_w, 3] - v[h, w, 3]) ** 2 +
                                 (v[target_h, target_w, 4] - v[h, w, 4]) ** 2)
                if r_dist < hr:
                    assigned_cluster[target_h, target_w] = c_num
                    c_points_num[c_num] = c_points_num[c_num] + 1
                    c_centroid[c_num] = c_centroid[c_num] + v[target_h, target_w]

    # cluster centroid 계산
    c_centroid[c_num] = c_centroid[c_num] / c_points_num[c_num]
```

## ● Merge Cluster

```
# Merge Cluster
assigned = np.full(len(c_centroid), -1, dtype='int64')
cluster = []
cluster_num = []
c_num = -1
ds = 500
dr = 30
for i in range(len(c_centroid)):
    # 기존 cluster를 인접 cluster와 merge
    if assigned[i] == -1:
        c_num = c_num + 1
        assigned[i] = c_num
        cluster_num.append(1)
        cluster.append(c_centroid[i])
    # 새로운 cluster에 포함되는 cluster 찾기
    for j in range(i + 1, len(c_centroid)):
        if assigned[j] == -1:
            # s, r 거리 계산 -> 인접 cluster끼리 clustering
            s_dist = np.sqrt((c_centroid[j][0] - c_centroid[i][0]) ** 2 +
                             (c_centroid[j][1] - c_centroid[i][1]) ** 2)
            if s_dist < ds:
                r_dist = np.sqrt((c_centroid[j][2] - c_centroid[i][2]) ** 2 +
                                 (c_centroid[j][3] - c_centroid[i][3]) ** 2 +
                                 (c_centroid[j][4] - c_centroid[i][4]) ** 2)
                if r_dist < dr:
                    assigned[j] = c_num
                    cluster_num[c_num] = cluster_num[c_num] + 1
                    cluster[c_num] = cluster[c_num] + c_centroid[j]

    # cluster centroid 계산
    cluster[c_num] = cluster[c_num] / cluster_num[c_num]
```

### ③ Average Motion Vector of Segments

```
# segment별 motion vector 표현
length = len(cluster)
segment_motion = np.zeros((length,2), dtype='float32')
pixel_num = np.zeros(length, dtype='float32')
for x in range(width):
    for y in range(height):
        number = assigned[assigned_cluster[y,x]]
        segment_motion[number] = segment_motion[number] + motion[y, x]
        pixel_num[number] = pixel_num[number] + 1

for i in range(length):
    center_y = int(cluster[i][0])
    center_x = int(cluster[i][1])
    segment_motion[i] = segment_motion[i] / pixel_num[i]
    dy = int(segment_motion[i][0] * 20)
    dx = int(segment_motion[i][1] * 20)
    cv2.arrowedLine(frame1_copy_img, (center_x, center_y), (center_x + dx, center_y + dy), (255, 0, 0), 1, tipLength=0.5)
```