2주차 chap04

연습문제_홀수

Chap4. 에지와 영역

4.1 에지영역

에지

물체 경계에 위치한 점

- 에지 검출 : 에지에 해당 화소 찾기
- 에지 향상 : 에지 더 잘보이도록 하기 위해 에지와 배경 간의 대비 증가
- 에지 추적 : 에지 따라가기
- 에지를 완벽하게 검출해 물체의 경계를 폐곡선으로 따낼 수 있다면 분할 문제 해결
- 특성이 크게 다른 화소에 집중하는 방식
- > 물체의 위치 모양 크기 에 대한 정보 찾기 가능

영역

특성이 비슷한 화소를 묶는 방식

에지 검출 알고리즘 : 물체 내부는 명암이 서서히 변하고 경계는 급격히 변하는 특성 활용

4.1.1 영상의 미분

미분 : 변수의 x 값이 미세하게 증가했을 때 함수 변화량 측정, 에지 추출 방법

- 영상을 (x, y) 변수의 함수로 간주했을 때, 이 함수의 1차 미분(1st derivative) 값이 크게 나타나는 부분을 검출

디지털 영상 미분

- 미분은 기본적으로 연속적인 공간에서 적용, but 영상은 연속공간이 아닌 이산 공간이기에 근사화한 것으로 적용
- 영상 f에 미분 적용해 f' -> 영상에서 가장 작은 단위가 1이며 (-1,1)로 컨볼루션
- 필터 u(에지 연산자)로 컨볼루션하여 구현
- 명암을 미분해 튀어나온 부분이 에지

4.1.2 에지 연산자

- 명암 변화 x = 0
- 명암 변화 o = 3
- 에지 검출은 모두 명암 변화에만 의존, 두 인접한 물체가 비슷한 명암을 가져 명암 변화가 적은 경우 경계에서 에지 발생X

컨볼루션의 값은 f 원래 영상의 부호를 의미한다 !?

물체 경계를 지나면서 명암값이 커지면 미분값 양수, 작아지면 음수

램프에지 : 계단 모양이 아닌, 명암이 몇화소에 걸쳐 변화, 정확한 에지의 위치 찾기 어려움

- --- 에지 검출 과정 ---
  두꺼운 에지에서 위치 찾기 적용

1차미분 : 원래 영상의 변화량이 1차 미분값
에지 발생여부와 에지가 어떤 방향을 향하는지 알 수 있음
봉우리 찾기

2차미분 : 1차 미분값의 변화량이 2차 미분값
영교차 찾기

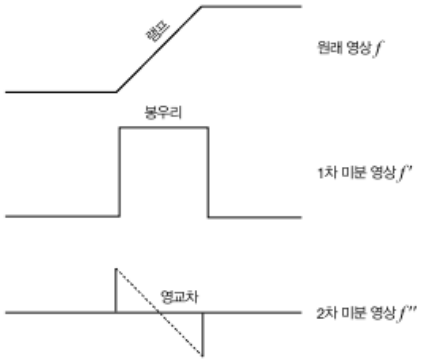


그림 4-5 램프 에지에서 발생하는 봉우리와 영교차

- 봉우리 : 봉우리의 두께가 1이므로 계단 에지만 존재하는 경우 에지 찾기 간단
- 영교차 : 왼쪽과 오른쪽에 부호가 다른 반응 발생, 자신은 0을 갖는 위치

1차 미분에 기반한 에지 연산자

미분에 기반한 2차원 에지 연산자는 크기 확장시 잡음을 흠수해 더 좋은 성능 보임
-> 잡음으로 인해 스무딩이 필요해 2차원 연산저 적용

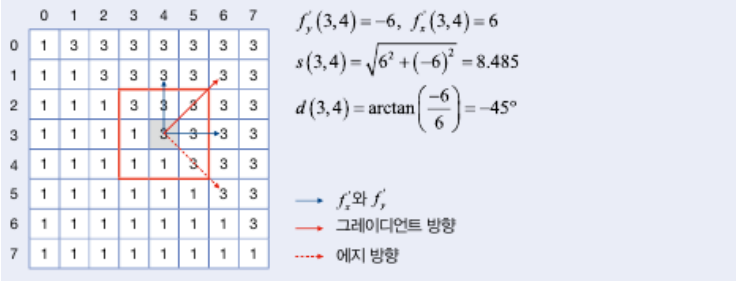
- 프로윗
  장점 : 돌출된 값을 잘 평균화
  단점 : 값이 일렬로 나열되어 있어 대각선보다 수평 수직에 놓인 에지에 민감
- 소벨
  장점: 돌출된 값 잘 평균화, 모든 방향 에지 검출 가능, 잡음에 강함
  단점 : 대각선 방향에 놓인 에지 민감

실제영상에서 구한 그래디언트 크기와 방향은 프레윗과 소벨을 적용한 결과 영상

- 에지 강도 : 크기 = 픽셀 값의 차이 정도, 변화량 -> 에지 가능성 나타냄
- 에지 방향 : 방향 = 값이 급격히 증가하는 방향 -> 에지 진행 방향 나타냄

[예시 4-1] 소벨 연산자 적용 과정

[그림 4-7]은 대각선을 기준으로 위쪽은 3, 아래쪽은 1인 가상의 영상에 소벨 에지 연산자를 적용하는 과정을 예시한다. 회색으로 표시한 (3,4) 화소에 대한 자세한 계산 과정을 설명한다.



각각 프로윗 소벨 적용한 값을 기반으로 에지 방향 구함

```
# 4-1 에지 검출
import cv2 as cv

img=cv.imread('soccer.jpg')
gray=cv.cvtColor(img,cv.COLOR_BGR2GRAY) # 명암으로 변화

# 소벨 연산자 적용
grad_x=cv.Sobel(gray,cv.CV_32F,1,0,ksize=3) # 결과 영상 32비트 실수 맵 저장, (1,0) x 방향 연산자, 3*3 크기 사용
grad_y=cv.Sobel(gray,cv.CV_32F,0,1,ksize=3) # y 방향 연산자

# 음수가 포함된 맵에 절대값을 취해 양수 영상으로 변환
sobel_x=cv.convertScaleAbs(grad_x)  # convertScaleAbs : 부호없는 8비트형 맵을 형성해 크기가 0보다 작으면 0, 255넘으면 255
sobel_y=cv.convertScaleAbs(grad_y)

# 에지 강도 계산
edge_strength=cv.addWeighted(sobel_x,0.5,sobel_y,0.5,0) # sobel_x * 0.5 + sobel_y * 0 + 0
# sobel_x 와 sobel_y 가 같은 데이터 형이면 결과영상 같은 데이터 형
# 다른 데이터 형이면 결과영상 에러

# 결과영상 윈도우 디스플레이
cv.imshow('Original',gray)
cv.imshow('sobelx',sobel_x)
cv.imshow('sobely',sobel_y)
cv.imshow('edge strength',edge_strength)

cv.waitKey()
cv.destroyAllWindows()
```

```
cv2.Sobel(src, ddepth, dx, dy, dst=None, ksize=None, scale=None,
delta=None, borderType=None) -> dst
```

- 입력 영상
- 출력영상 데이터 타입으로 -1이면 입력 영상과 같은 데이터 타입 사용
- x방향 미분 차수
- y방향 미분 차수
- 출력영상행렬
- 커널 크기

- 연산결과에 추가적으로 곱한 값으로 기본값 1
- 연산 결과에 추가적으로 더할 값으로 기본값0
- 가장자리 픽셀 확장방식

4.2 캐니 에지

1. 블러링 통한 노이즈 제거 : 5*5 크기의 가우시안 필터 적용해 불필요 잠음 제거
2. 그래디언트 계산 : 기울기의 크기와 방향 계산

3. 비최대치 억제
현재 화소가 이웃하는 화소들보다 크면 보존, 그렇지 않으면 에지가 아닌 것으로 간주 == 최대가 아니면 억제
에지 방향에 수직인 두 이웃 화소의 에지 강도 비교

4. 이력 임계값으로 에지 결정

실제 에지가 아닌데 에지로 검출된 화소인 거짓 긍정을 줄이기 위해 임계값 사용

두개의 임계값 T(high), T(low)를 사용해서 에지의 이력을 추적하여 에지를 결정하는 방법

구별을 하기 위해 임계 값 사용

파랑색 영역은 제거(non-relevant)
주황색, 빨간색 영역을 각각 구분
진한 흰색은 빨간색 영역으로 선명한 에지(Strong), 옅은 회색 영역은 주황색 영역(weak)으로 옅은 에지입니다.

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

- 에지 추적은 T_high를 넘는 화소에서 시작, 추적 도중에는 T_low 적용
- 이웃 화소가 추적이력이 있으면 자신은 신뢰도가 낮더라도 에지로 간주
- 추적 시작하지 않은 이웃 화소들을 대상으로 T(low)보디 큰 화소 에지 결정

```
# 4-2 캐니 에지
# import cv2 as cv

img=cv.imread('soccer.jpg') # 영상 읽기
```

```
gray=cv.cvtColor(img,cv.COLOR_BGR2GRAY)


canny1=cv.Canny(gray,50,150)  # Tlow=50, Thigh=150으로 설정
canny2=cv.Canny(gray,100,200) # Tlow=100, Thigh=200으로 설정


cv.imshow('Original',gray)
cv.imshow('Canny1',canny1) # 에지 강도가 작은 화소도 추적 가능하나 잡음 발생
cv.imshow('Canny2',canny2) # 임계값 높으면 에지강도가 큰 화소만 추적해 더 작은 에지 발생


cv.waitKey()
cv.destroyAllWindows()
```

4.3 직선 검출

에지 화소는 1, 에지가 아닌 화소 0**에지를 연결해 경계선으로 변환하고 경계선을 직선으로 변환**

→ **물체 표현이나 인식에 유리**

1) 에지 맵에서 경계선 검출

2) 길이가 임계값 이상인 경계선만 취함

```
# 4-3 직선 검출
import cv2 as cv
import numpy as np


img=cv.imread('soccer.jpg')  # 영상 읽기
gray=cv.cvtColor(img,cv.COLOR_BGR2GRAY)
canny=cv.Canny(gray,100,200)


contour,hierarchy=cv.findContours(canny,cv.RETR_LIST,cv.CHAIN_APPROX_NONE) # 경계선 정보 검출


lcontour=[]
for i in range(len(contour)):
    if contour[i].shape[0]>100: # 길이가 100보다 크면
        lcontour.append(contour[i])

# 외곽선 그리기
cv.drawContours(img,lcontour,-1,(0,255,0),3)


cv.imshow('Original with contours',img)
cv.imshow('Canny',canny)


cv.waitKey()
cv.destroyAllWindows()
```

findCountours 사용

- 경계선 찾을 에지 영상

- 구멍이 있는 경우 바깥쪽 경계선과 그 안에 구멍의 경계선을 계층적으로 찾는 방식 지정
  -> cv.RETR_LIST : 맨 바깥쪽 경계선 찾기
- 경계선 표현 방식 지정
  -> cv.CHAIN_APPROX_NONE : 모든 점 기록 / cv.CHAIN_APPROX_SIMPLE : 직선에 대해 양 끝점만 기록 / cv.CHAIN_APPROX_TC89_L1 & cv.CHAIN_APPROX_TC89_KCOS : Teh-Chin 알고리즘으로 굴곡이 심한 점 찾아 기록
    - contours: 검출된 외곽선 좌표
      • hierarchy: 외곽선 계층 정보

drawCountours 사용

왔다갔다 반복되는 부분이 있어서, 50이상의 경계선을 검출하고 싶다면 한계 100
-> 함수는 시작점부터 끝점까지 추적한 다음 역추적하여 시작점으로 돌아와 경계선 표현하기 때문

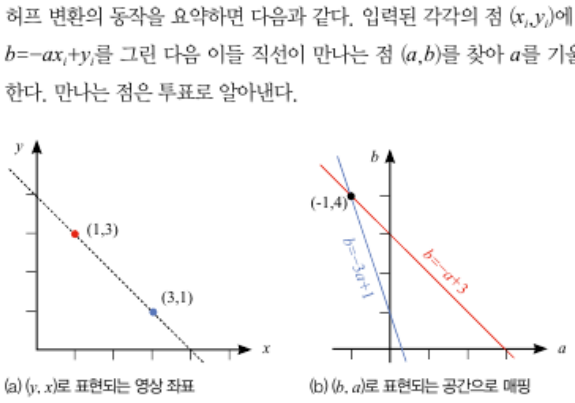
- 경계선 그려넣을 영상
- 경계선
- 모든 경계선 그리기 (-1), 해당 번호에 해당하는 경계선 하나만 그리기 (양수)
- 색
- 두께

허프변환

잡음 발생 : 에지가 자잘하게 끊겨 나타나는 경우 발생 -> **"허프변환"** 적용하면 끊긴 에지 모아 선분 or 원 등 검출

- 직선의 방정식 이용

1. 공간 (x,y)에 위치한 직선 y = ax + b
2. b = -ax + y 로 변경 가능하며 이는 a, b를 변수로 취급, 새로운 공간 (a,b) 생성
3. 1)에서 점이였던 것이 공간이 변하며 직선으로 변경
4. 새로운 공간에서의 두개의 직선이 만난다면 이 만나는 점은 원래 공간에서 두 점을 지나는 직선의 기울기와 y절편



- > (a,b) 공간에서 **두 직선이 만나는 점은 투표**로 알아냄
  -> 직선이 지나지 않음 0, 직선이 지남 1, 두 직선이 만남 2

발생 시 고려사항
조건1) fact, 많은 점들이 있고 점들이 완전히 일직선을 이루진 않음
-> (a,b)공간을 이산화해 해결 : 2차원 누적 배열 v 생성, v를 0으로 초기화한 다음 각각 직선은 자신이 지나는 모든 칸에 1만큼 투표

조건2) 투표가 이뤄진 누적 배열에 잡음 많음
-> 비최대억제로 잡음이 많은 상태에서 한 점 결정

0	1	0	0	0	0	0	0
0	2	2	0	1	3	0	0
0	3	5	3	2	0	0	0
0	2	4	2	6	7	0	0
0	2	3	3	5	8	6	0
0	1	0	0	0	4	5	3

그림 4-11 비최대 억제로 찾은 극점 2개

:: 비최대억제로 극점 3개 남았는데 임계값을 적용해 노란색점 2개가 최종

조건3) y = ax + b, 기울기 a가 무한대인 경우 투표 불가
-> 극좌표에서 직선의 방정식 표현하는 식으로 해결

$$x\sin(\theta)+y\cos(\theta)=\rho$$

- 한계점)
  직선의 양끝점 알려주지 못함
  -> 비최대억제과정에서 극점을 형성한 화소를 찾아 가장 먼 곳에 있는 두 화소 계산하는 추가 과정 필요
  ex) 원 검출하기 위해 원의 방정식 이용

```
cv.HoughCircles(gray,cv.HOUGH_GRADIENT,1,200,param1=150,param2=20,minRadius=50,maxRadius=120) # 원 검출 허프변환
```

- 명암영상에서 원 검출해 중심과 반지름 저장한 리스트 반환
- 여러 변형 알고리즘 중 하나 지정
  -> cv.HOUGH_GRADIENT : 에지 방향 정보 추가 사용
- 누적 배열의 크기 지정
  -> 1 : 입력 영상과 동일 크기 사용
- 원 사이의 최소 거리 지정, 작을수록 많은 원 검출
- 캐니 에지 알고리즘이 사용하는 임계값 T_high
- 비최대 억제 적용할 때 사용하는 임계값
- 원의 최소 반지름
- 원의 최대 반지름

허프변환 문제점

이상치가 섞여있을 수 있는데 허프변환은 모든 점에 동일한 투표기회를 제공하기 때문에 누적 배열에 잡음이 많이 발생하는 원인

최소평균제곱오차 알고리즘

허프변환과 유사함 (이상치에 민감함)
모든 점을 대상으로 오류 계산하고 최소 오류 범하는 직선 찾기

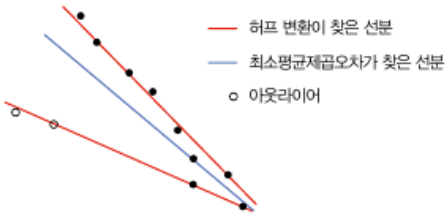


그림 4-12 강인하지 않은 기법의 선분 추정

아웃라이어의 영향으로 실제에서 벗어난 직선 찾음, 모든 샘플이 동등한 자격으로 오류 계산 참여

문제점 해결을 위해 아래 알고리즘 적용

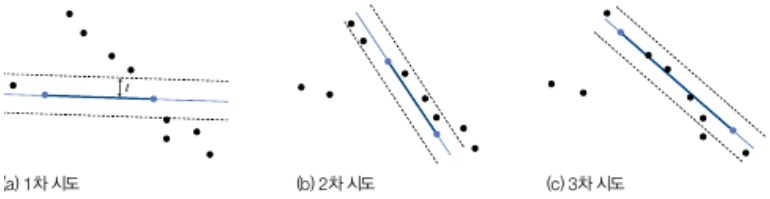
강인한 추정

아웃 라이어를 걸러내는 과정을 가진 추정 기법
=> 중앙값을 이용해 아웃라이어를 배제하여 추정하기 때문

RANSAC

인라이어와 아웃라이어가 섞여 있는 상황에서 인라이어를 찾아 최적 근사하는 기법

1. 랜덤하게 두 점 서택, 두점을 지나는 직선 계산
2. 일정한 양의 오차 t를 허용해 직선에 일치하는 점의 개수 count
3. 개수가 임곗값 d를 넘지 못하면 가능성 X -> 버림
4. 3번 통과시, 해당 점의 개수로 최적 직선 추정하고 추정 오류가 임계값 e보다 작으면 후보군 추가 아니면 out
5. 3번과 4번의 반복으로 후보군에서 최적 찾기



a. d=5라면 해당점이 3개이기에 out
b. d=5라면 해당점이 7이기에 ok, 그러나 7개로 최적 직선 추정하고 추정 오류가 임계값 e보다 작으면 후보군 추가 아니면 out

장점 : 반복횟수가 많아 직선 찾을 가능성 up
단점 : 시간이 더 걸려 적절한 값 설정 필요

4.4 영역분할

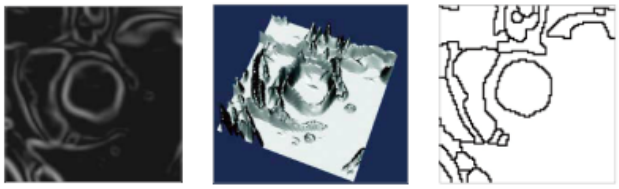
영역 분할 : 물체가 점유한 영역을 구분하는 작업

의미분할 : 의미 있는 단위로 분할하는 방식

이진화 알고리즘, 군집화 알고리즘 를 적용해 좋은 분할 성능을 얻을 수 있음

워터셰드

비가 오면 오목한 곳에 웅덩이가 생기는 현상 모방하는 연산



(a) 에지 강도 맵 (b) 지형으로 간주 (c) 워터셰드

그림 4-14 워터셰드 분할 알고리즘[Cousty2007]

(b)에서 낮은 곳부터 물을 채우는 연산을 반복하면 (c)와 같이 서로 다른 **웅덩이**를 찾을 수 있음

→ 이러한 웅덩이를 **영역으로 간주**

SLIC 알고리즘

슈퍼 화소 알고리즘 (영상을 작은 영역으로 분할해 다른 알고리즘 입력으로 사용하며, 이 영역은 화소보다 크지만 물체보다 작아 슈퍼화소라 불림) 중 하나

- 화소 5차원 벡터표현 (R, G, B, x, y) : 색상 나타내는 3개의 값 + 위치 나타내는 2개의 값

1. 입력영상에서 k개 화소를 군집 중심으로 지정

   등간격 패치로 분할한 다음 패치 중심을 군집 중심으로 간주

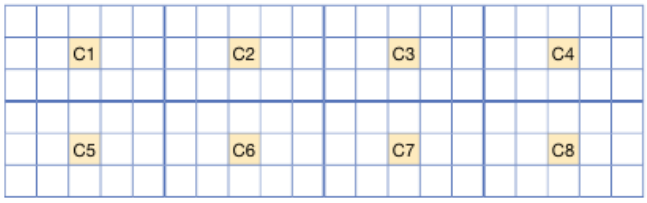


그림 4-15 SLIC 알고리즘의 초기 군집 중심

2. 군집 중심이 물체 경계에 놓이는 일 방지하기 위해 그레이디언트가 가장 낮은 이웃 화소로 이동
3. 화소를 가장 가까운 군집 중심에 할당하는 단계와 군집 중심을 갱신하는 단계 반복
   - 화소 할당 : 화소 각각에 대해 주위 4개 군집 중심과 자신까지 거리를 계산해 가장 유사한 군집 중심 할당
   - 중심 갱신 : 각 군집 중심은 자신에게 할당된 화소를 평균해 군집 중심 갱신
4. 모든 군집 중심의 이동량의 평균을 구하고 평균이 임계치보다 작으면 수렴했다고 판단.

   → 알고리즘 stop

```
# 4-4
import skimage # 입력 영상을 슈퍼화소로 분할하기에 slic 함수가 사용하기 편리
import numpy as np
import cv2 as cv


img = skimage.data.coffee() # 내부 coffee 영상 읽어 객체 저장
cv.imshow('Coffee image',cv.cvtColor(img,cv.COLOR_RGB2BGR))


slic1 = skimage.segmentation.slic(img,compactness=20,n_segments=600) # 슈퍼 화소 분할 수행
sp_img1 = skimage.segmentation.mark_boundaries(img,slic1) # 객체 분할 정보 img 영상에 표시, 결과를 sp_img1 객체에 저장
sp_img1 = np.uint8(sp_img1*255.0) # 표현 type을 0-255사이로 변환하고 unit8형으로 변환

# 위와 동일, 파라미터만 수정
slic2=skimage.segmentation.slic(img,compactness=40,n_segments=600)
sp_img2=skimage.segmentation.mark_boundaries(img,slic2)
sp_img2=np.uint8(sp_img2*255.0)


cv.imshow('Super pixels (compact 20)',cv.cvtColor(sp_img1,cv.COLOR_RGB2BGR))
cv.imshow('Super pixels (compact 40)',cv.cvtColor(sp_img2,cv.COLOR_RGB2BGR))


cv.waitKey()
cv.destroyAllWindows()
```

`skimage` 은 numpy 배열로 영상 표현하며 , RGB 순서로 저장하기에 `cvtColor` 함수로 BGR로 변환해 출력

```
skimage.segmentation.slic(img,compactness=20,n_segments=600)
```

- 슈퍼 화소 분할할 영상
- 슈퍼화소의 모양 조절 : **값이 클수록 네모에 가까운 모양** 형성 but 슈퍼 화소의 모양을 조절
- 슈퍼 화소의 개수 지정 → 알고리즘에서 사용한 **k**

위와 같이 값이 클수록 네모에 가깝게 유지되지만 슢화소의 색상 균일성 down

최적화 분할

영상을 그래프로 표현하고 분활을 최적화 문제로 풀이

→ 지역적 명암 변화를 보며 전역적 정보를 같이 고려

- 영상을 그래프로 표현할 때 슈퍼화소를 노드로 취하면 노드 개수를 효율적으로 감소할 수 있음
- 두 노드 v_p, v_q 를 연결하는 에지 가중치로는 **유사도 s_pq** 사용
- f(v)는 v에 해당하는 화소의 색상과 위치를 결합한 벡터
- “v_q 가 neighbor(v_p)에 속한다면” :: v_q와 v_p가 8-이웃을 이루거나 둘 사이의 거리가 사용자가 지정한 값 r 이내면 참

$$\text{거리}\begin{cases} d_{pq} = \|f(v_p) - f(v_q)\|, \text{ 만일 } v_q \in neighbor(v_p) \\ \infty, \text{ 그렇지 않으면} \end{cases}$$
$$\text{유사도}\begin{cases} s_{pq} = D - d_{pq} \text{ 또는 } \frac{1}{e^{d_{pq}}}, \text{ 만일 } v_q \in neighbor(v_p) \\ 0, \text{ 그렇지 않으면} \end{cases} \quad (4.8)$$

정규화 절단 알고리즘

정규화 절단 : 화소를 노드로 취하고, f(v)로 5차원 벡터 사용, 유사도를 에지 가중치로 사용

$$cut(C_1, C_2) = \sum_{v_p \in C_1, v_q \in C_2} s_{pq} \quad (4.9)$$

원래의 영역을 두개로 분할할 때 *cut* 적용, 영역 분할의 좋은 정도 측정해주는 목적함수

→ 두 영역이 클수록 둘 사이에 에지가 많아 덩달이 증가

$$ncut(C_1, C_2) = \frac{cut(C_1, C_2)}{cut(C_1, C)} + \frac{cut(C_1, C_2)}{cut(C_2, C)} \qquad (4.10)$$

cut 을 정규화해 영역의 크기에 **중립**이 되도록 함

→ ncut이 작을수록 좋은 분할이므로 최소화 문제

```
import skimage
import numpy as np
import cv2 as cv
import time


coffee=skimage.data.coffee()


start=time.time() # 분할하는 데 걸리는 시간 측정
slic=skimage.segmentation.slic(coffee,compactness=20,n_segments=600,start_label=1) # 슈퍼화소로 분할

g=skimage.future.graph.rag_mean_color(coffee,slic,mode='similarity')
ncut=skimage.future.graph.cut_normalized(slic,g)  # 정규화 절단
print(coffee.shape,' Coffee 영상을 분할하는데 ',time.time()-start,'초 소요') # 시간 측정


marking=skimage.segmentation.mark_boundaries(coffee,ncut)
ncut_coffee=np.uint8(marking*255.0)


cv.imshow('Normalized cut',cv.cvtColor(ncut_coffee,cv.COLOR_RGB2BGR))


cv.waitKey()
cv.destroyAllWindows()
```

`rag_mean_color` 슈퍼화소를 노드로 사용하고 'similarity' 를 에지 가중치로 사용한 그래프를 구성

`mark_boundaries` 원래 영상에 화소에 영역의 번호를 부여한 맵을 이용해 영역 경계를 표시

4.5 대화식 분할

능동 외곽선 알고리즘

사용자가 물체 내부에 초기 곡선을 지정하면 곡선을 점점 확장하며 물체 외곽선 접근

- 곡선이 꿈틀대며 에너지가 최소인 상태를 찾아가기에 스네이크라는 별명 생성

  → 곡선 구현 방법 필요

$$g(l) = (y(l), x(l))$$

g(l) : 2차원 상의 곡선이므로 위와 같이 표현

l : 매개변수, [0,1] 범위의 실수이지만 디지털 공간은 이산공간이기에 *0, 1, 2 .., n* 으로 표현

$$E(g)=\sum_{l=0}^{n}\left(e_{image}\left(g(l)\right)+e_{internal}\left(g(l)\right)+e_{domain}\left(g(l)\right)\right)$$

에너지를 최소로 하는 최적의 곡선을 찾는 최적화 문제

1. 사용자가 입력 받아 초기 곡선 설정
2. 자신과 8-이웃을 포함한 9개의 점에 대해 곡선 에너지 E 도출
3. 에너지가 최소점으로 이동하고 이동량 증가
4. 곡선의 이동량이 임계치보다 작으면 수렴했다고 간주하고 최적 곡선 도출

[알고리즘 4–1] 스네이크로 물체 분할

입력: 명암 영상, 임곗값 T

출력: 최적 곡선 $\hat{g}$

1. 사용자 입력을 받아 초기 곡선 g를 설정한다.
2. while TRUE
3. $moved$=0
4. for i=0 to n−1
5. for $g(i)$의 9개 이웃점 각각에 대해 // 자신과 8−이웃을 포함한 9개 점
6. $g(i)$를 이웃점으로 이동한 곡선의 에너지 E를 식 (4.11)로 구한다.
7. if 에너지가 최소인 점이 $g(i)$와 다르면
8. $g(i)$를 최소점으로 이동하고 $moved$를 1 증가시킨다.
9. if $moved < T$ // 곡선의 이동량이 임계치보다 작으면 수렴했다고 간주하고 탈출
10. break

GrabCut

사용자가 붓칠한 정보를 이용해 물체 분할

```
import cv2 as cv
import numpy as np


img=cv.imread('soccer.jpg') # 영상 읽기
img_show=np.copy(img)   # 붓 칠을 디스플레이할 목적의 영상

# 사용자가 붓칠에 따라 물체인지 배경인지에 대한 정보를 기록할 배열 생성
mask=np.zeros((img.shape[0],img.shape[1]),np.uint8)
mask[:,:]=cv.GC_PR_BGD    # 모든 화소를 배경일 것 같음으로 초기화


BrushSiz=9        # 붓의 크기
LColor,RColor=(255,0,0),(0,0,255) # 파란색(물체)과 빨간색(배경)


def painting(event,x,y,flags,param):
    if event==cv.EVENT_LBUTTONDOWN:
        cv.circle(img_show,(x,y),BrushSiz,LColor,-1)  # 왼쪽 버튼 클릭하면 파란색
        cv.circle(mask,(x,y),BrushSiz,cv.GC_FGD,-1)

    elif event==cv.EVENT_RBUTTONDOWN:
        cv.circle(img_show,(x,y),BrushSiz,RColor,-1)  # 오른쪽 버튼 클릭하면 빨간색
        cv.circle(mask,(x,y),BrushSiz,cv.GC_BGD,-1)
```

```
    elif event==cv.EVENT_MOUSEMOVE and flags==cv.EVENT_FLAG_LBUTTON:
        cv.circle(img_show,(x,y),BrushSiz,LColor,-1)# 왼쪽 버튼 클릭하고 이동하면 파란색
        cv.circle(mask,(x,y),BrushSiz,cv.GC_FGD,-1)

    elif event==cv.EVENT_MOUSEMOVE and flags==cv.EVENT_FLAG_RBUTTON:
        cv.circle(img_show,(x,y),BrushSiz,RColor,-1)  # 오른쪽 버튼 클릭하고 이동하면 빨간색
        cv.circle(mask,(x,y),BrushSiz,cv.GC_BGD,-1)

    cv.imshow('Painting',img_show)


cv.namedWindow('Painting')
cv.setMouseCallback('Painting',painting)


while(True):        # 붓 칠을 끝내려면 'q' 키를 누름
    if cv.waitKey(1)==ord('q'):
        break

# ----------------- GrabCut 적용하는 코드 -----------------
background=np.zeros((1,65),np.float64)  # 배경 히스토그램 0으로 초기화
foreground=np.zeros((1,65),np.float64)  # 물체 히스토그램 0으로 초기화

# 실제 분할시도
cv.grabCut(img,mask,None,background,foreground,5,cv.GC_INIT_WITH_MASK)
mask2=np.where((mask==cv.GC_BGD)|(mask==cv.GC_PR_BGD),0,1).astype('uint8')
grab=img*mask2[:,:,np.newaxis]
cv.imshow('Grab cut image',grab)


cv.waitKey()
cv.destroyAllWindows()
```

```
cv.grabCut(img,mask,None,background,foreground,5,cv.GC_INIT_WITH_MASK)
```

- 원본 영상
- 사용자가 지정한 물체와 배경정보를 가진 맵
- 관심영역을 지정하는 ROI : None - 전체 영상을 대상으로 하라고 지시
- 배경
- 물체 히스토그램
- n 번 반복
- 배경과 물체를 표시한 맵 사용

→ 여러번 반복해 정교하게 물체 오려내는 실험 go

4.6 영역 특징

불변성 : 변환을 해도 특징의 값이 변하지 않음

↔ 등변성 : 특징이 어떤 변환에 대해 따라 변화함

회전과 축소에 불변인 특징을 사용해야 구분해낼 수 있음

영역 R의 모멘트 정의

$$m_{qp}(R)=\sum_{(y,x)\in R} y^q x^p \qquad (4.13)$$

(y,x) : 영역 R에 속하는 화소

$$\left.\begin{array}{l} \text{면적: } a=m_{00} \\ \text{중점: } (\dot{y},\dot{x})=\left(\dfrac{m_{10}}{a},\dfrac{m_{01}}{a}\right) \end{array}\right\} \qquad (4.14)$$

$$\mu_{qp}=\sum_{(y,x)\in R}(y-\dot{y})^q(x-\dot{x})^p \qquad (4.15)$$

(4.15) 중심 모멘트로부터 열분산, 행분산, 열행분산 모두 구할 수 있고, 주축 방향 도출 가능

(4.17)에 해당하는 **크기불변** 도출 가능 → 영역의 둘레, 둥근 정도 측정 가능

$$\eta_{qp}=\frac{\mu_{qp}}{\mu_{00}^{\left(\frac{q+p}{2}+1\right)}} \qquad (4.17)$$

텍스처

일정한 패턴의 반복

텍스처가 세밀하면 많은 에지 발생, 거칠면 적게 발생하는 성질 이용

busy는 에지 화소 수를 전체 화소 수로 나눠 세밀함 측정

LBP Local Binary Patern : 중심 화소와 주위 화소의 명암값을 비교해 텍스처 구함

→ 작은 명암 변화에 민감

LTP Local Ternary Patern : LBP 단점 보완

이진영역의 특징을 추출하는 함수 사용

```
import skimage
import numpy as np
import cv2 as cv


orig=skimage.data.horse()
img=255-np.uint8(orig)*255 # 말 영역은 255, 배경은 0
cv.imshow('Horse',img)

# 물체 경계선 추출
contours,hierarchy=cv.findContours(img,cv.RETR_EXTERNAL,cv.CHAIN_APPROX_NONE)


img2=cv.cvtColor(img,cv.COLOR_GRAY2BGR)   # 컬러 디스플레이용 영상
```

```
cv.drawContours(img2,contours,-1,(255,0,255),2) # 경계선 표시 (영상,경계선,경계선 모두 표시, 색깔, 선 두께)
cv.imshow('Horse with contour',img2)


contour=contours[0]


m=cv.moments(contour)       # 모멘트 추출해 저장
area=cv.contourArea(contour) # 경계선으로 둘러싸인 영역의 면적 계산
cx,cy=m['m10']/m['m00'],m['m01']/m['m00'] # 중점
perimeter=cv.arcLength(contour,True) # 둘레의 길이 계산
roundness=(4.0*np.pi*area)/(perimeter*perimeter) # 둥근정도
print('면적=',area,'\n중점=(',cx,',',cy,')','\n둘레=',perimeter,'\n둥근 정도=',roundness)



img3=cv.cvtColor(img,cv.COLOR_GRAY2BGR)   # 컬러 디스플레이용 영상



contour_approx=cv.approxPolyDP(contour,8,True)  # 직선 근사
cv.drawContours(img3,[contour_approx],-1,(0,255,0),2)


hull=cv.convexHull(contour)     # 볼록 헐
hull=hull.reshape(1,hull.shape[0],hull.shape[2])
cv.drawContours(img3,hull,-1,(0,0,255),2)


cv.imshow('Horse with line segments and convex hull',img3)


cv.waitKey()
cv.destroyAllWindows()
```