

Image Feature Extraction

17010826김성민

01 SIFT 알고리즘

01 Scale Space 만들기

02 DoG 연산

03 Keypoint찾기

04 나쁜 Keypoint 제거

05 Keypoint에 방향 할당

06 SIFT descriptor 산출

07 프로그램 순서도

02 Harris Corner

01 기본 원리

02 테일러 확장

03 코너점 판별

04 프로그램 순서도

01 SIFT 알고리즘

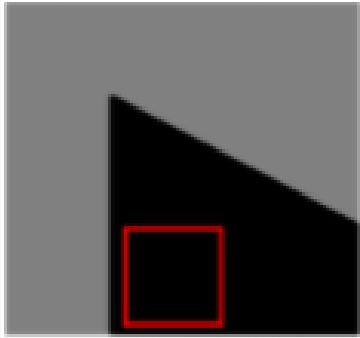
목표 : 이미지의 크기와 회전에 불변하는 특징을 추출하는 알고리즘



두 이미지에서 동일한 부분을
찾아 매칭할 수 있다.



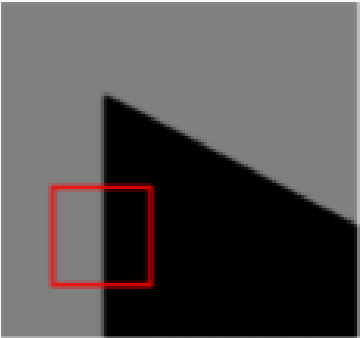
이미지 내의 keypoint와 플랫



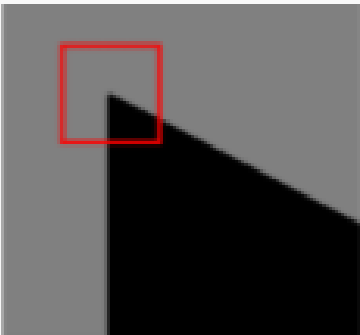
플랫 : 어느 방향으로도 픽셀의 변화량이
저조한 부분

플랫인 지점을 모아도 이미지의 특성을 알기 힘들

→ Keypoint가 될 수 없음



엣지 : 한 방향만 픽셀의 변화량이 큰 부분



코너 : 두 방향 이상에서 픽셀의 변화량이
큰 부분

SIFT 알고리즘

01 Scale Space 만들기

Scale Space 왜 만들까?

이미지 해석 시 전체적인 틀을 파악하는 것과 세부적인 내용을
파악하는 것이 모두 필요

Scale Space 개념 도입

SIFT 알고리즘이 입력 이미지에 대해 스케일 불변성을
가질 수 있게 함

Scale이란?

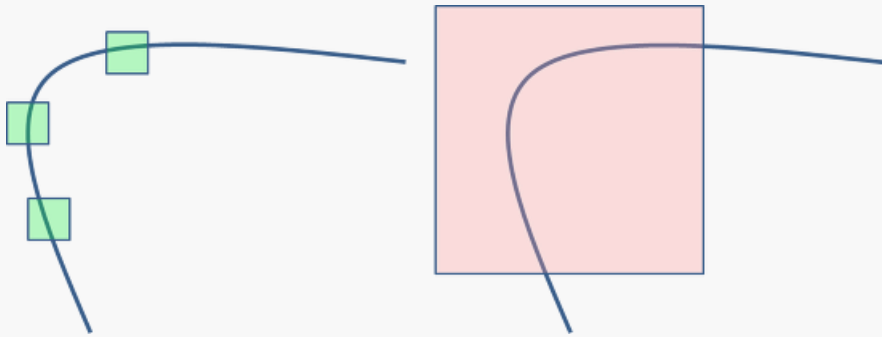
이미지를 보는 척도

가까이서 세세하게 본다

= 작은 스케일로 본다

멀리서 전체적으로 본다

= 큰 스케일로 본다



같은 선을 볼 때

작은 스케일(초록색 박스 크기)로 보면



직선으로 보임



큰 스케일(붉은색 박스 크기)로 보면



곡선으로 보임

같은 이미지를
여러 스케일로 관찰하는 것이 중요

Scale Space란?

Scale Space

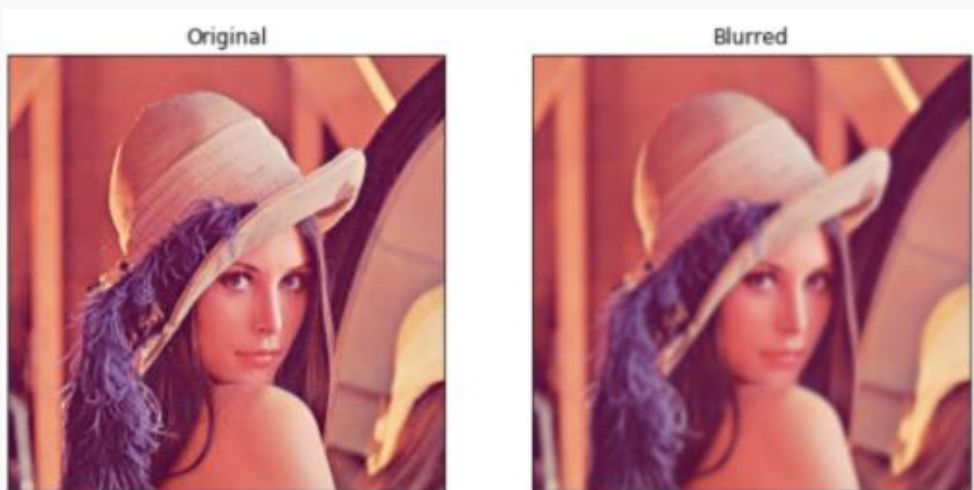
Scale의 개념을 반영하기 위해
하나의 이미지를 다양한 Scale로 저장한 공간

Image pyramid

단순히 입력 이미지의 크기를
단계적으로 변화시켜 만든 이미지의 집합

Image Blurring

이미지 처리에 사용되는
기본적인 이미지 변형 방법으로 노이즈 제거에 유용



이미지와 필터의
2차원 Convolution 계산을 통해 이루어짐



Scale을 변화시키기 위해 확대 축소 or Blurring 사용

Gaussian Filter

Image blurring에서 사용하는 필터 중 하나

$$g_{\sigma}(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

σ 는 blurring의 정도를 결정하며 클수록 더 많이 blur 된다.

더 많이 blur 될수록 scale이 커짐 → σ 를 Scale parameter라고 함

9x9 Gaussian Filter

4	0.0000	0.0000	0.0000	0.0000	0.0001	0.0000	0.0000	0.0000	0.0000
3	0.0000	0.0000	0.0002	0.0011	0.0018	0.0011	0.0002	0.0000	0.0000
2	0.0000	0.0002	0.0029	0.0131	0.0215	0.0131	0.0029	0.0002	0.0000
1	0.0000	0.0011	0.0131	0.0585	0.0965	0.0585	0.0131	0.0011	0.0000
0	0.0001	0.0018	0.0215	0.0965	0.1592	0.0965	0.0215	0.0018	0.0001
-1	0.0000	0.0011	0.0131	0.0585	0.0965	0.0585	0.0131	0.0011	0.0000
-2	0.0000	0.0002	0.0029	0.0131	0.0215	0.0131	0.0029	0.0002	0.0000
-3	0.0000	0.0000	0.0002	0.0011	0.0018	0.0011	0.0002	0.0000	0.0000
-4	0.0000	0.0000	0.0000	0.0000	0.0001	0.0000	0.0000	0.0000	0.0000
	-4	-3	-2	-1	0	1	2	3	4

중간으로 갈수록 높은 가중치 배치

→ 필터링 대상이 주변 픽셀의 영향을 더 많이 받음

Gaussian Filter의 좌표는 좌측 상단이 아닌 가장 가운데 부분이 (0, 0)

→ 수식을 이용해 필터를 구할 때 조심해야 함

SIFT에서 가장 적합한 필터 → 뒤에서 설명

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

L은 blurring 결과, G는 Gaussian Filter, I는 blurring할 이미지

Scale parameter인 σ 에 의해 결과물이 달라짐

Scale Space 만들기

전체적인 컨셉

입력 이미지를
점진적으로 $k^s\sigma$ 만큼 blur시켜
서로 다른 blur 이미지를 얻는다.

1단계

처음엔 입력 이미지를 σ 만큼 blur 시키고,
그 이후엔 σ 의 k 배를 하여 blur시킨다.

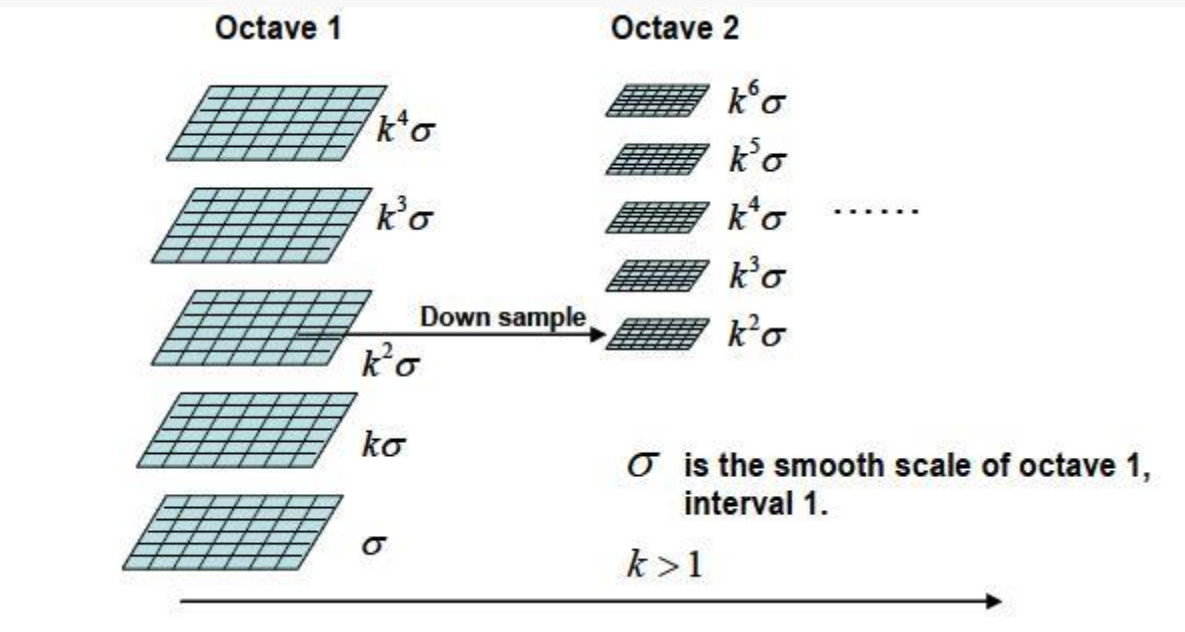
여기까지가 하나의 Octave (그림에서의 Octave 1)

2단계

k 가 2σ 가 되는 지점에서 새로운 Octave 시작
(그림에서 k 는 $\sqrt{2}$)

k 가 2σ 가 되는 이미지를 $\frac{1}{2}$ 로 downsampling
하여 새로운 Octave의 첫 번째 이미지로 사용

1단계 반복



3단계

1단계와 2단계를 반복하여
최종적인 Scale Space를 만든다.

→ 이미지의 홀수 번째 pixel만 살리고, 짝수 번째 행과 열 제거

Octave 수와 그 안 blur 이미지의 수는 이미지 해상도에 따라 프로그램 작성자가 결정

알고리즘 제안자는 4 Octave와 5 blur 이미지, $\sigma = 1.6$, $k = \sqrt{2}$ 사용

→ 이를 Gaussian pyramid라고 한다.

SIFT 알고리즘

02 DoG 연산

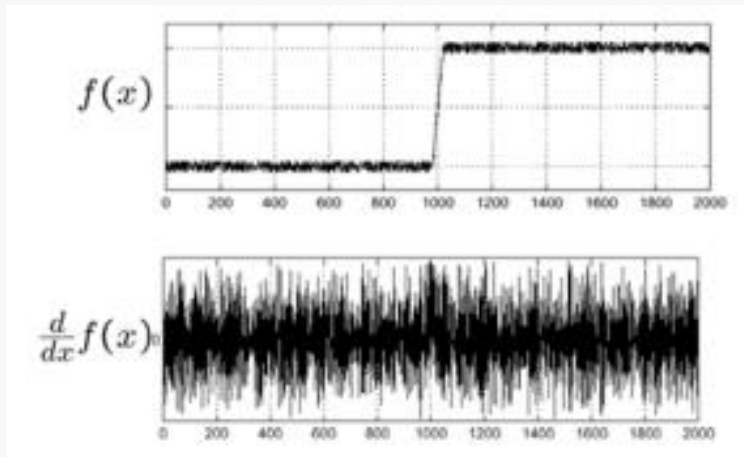
LoG(Laplacian of Gaussian)

LoG는 이미지 내의 keypoint를 얻기 위한 과정

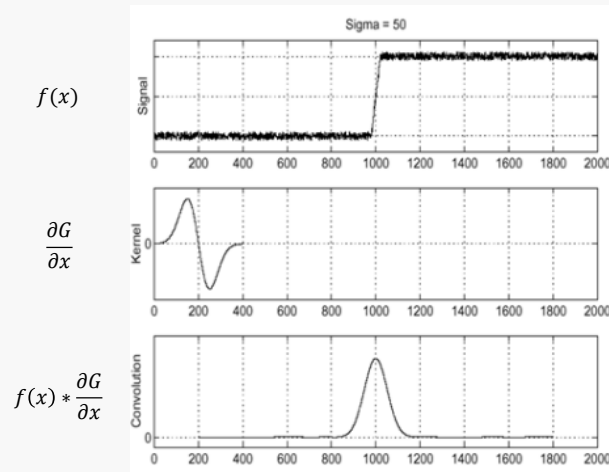
$LOG : \nabla^2 G$ ($G : Gaussian\ filter$)

이미지의 노이즈를 먼저 날리기 위해 먼저 이미지를 Gaussian blur한 다음에 2차 미분한다.

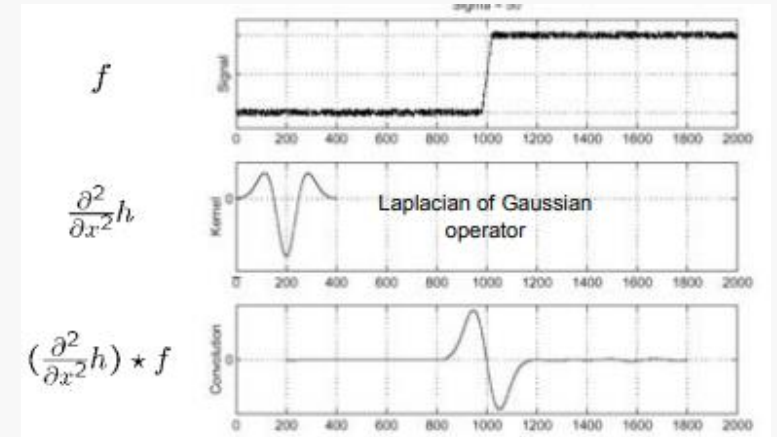
영상의 엣지나 코너를 추출하기 위해서 픽셀의 변화율을 구해야 함 → 이미지를 미분



노이즈 제거를 하지 않고
이미지를 미분할 시 미분 값으로
엣지 판별이 힘들다.



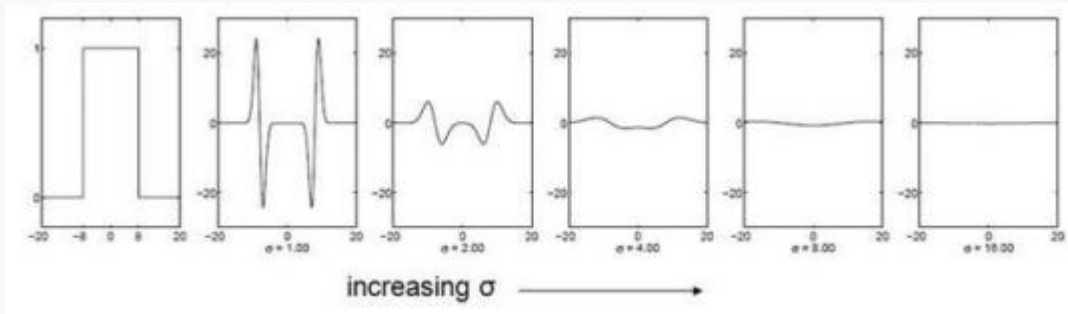
노이즈 제거를 한 경우 엣지 검출이
더 명확하고 정확해졌다.



엣지에 대해 LoG 연산한 식이
반응하는 모습을 볼 수 있다.

LoG(Laplacian of Gaussian)

그런데, σ 값이 다르다면?

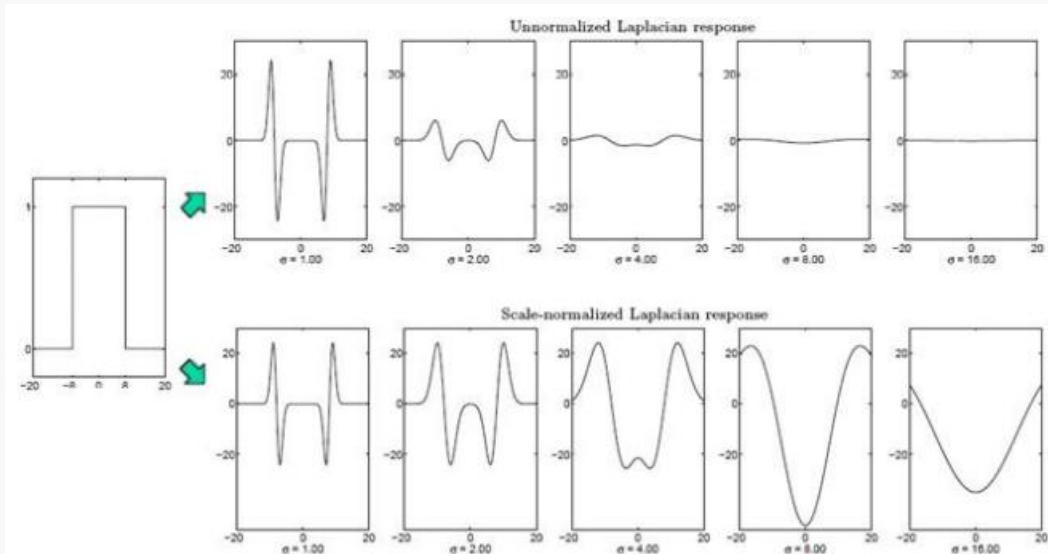


맨 왼쪽이 입력 함수이다.
LoG의 σ 를 키울수록 엣지에 대한 반응이 약해
엣지 검출 함수로 사용하기 힘들어진다.

LoG는 scale이 커지는 경우
결과가 좋지 않은 성질을 가짐

Scale-normalized LoG 사용

LoG에 σ 를 곱하는 방식



같은 입력에 대해, 같은 σ 인 경우
위는 LoG 연산,
아래는 scale-normalized LoG 연산이다.

Scale-normalized LoG를 사용한 경우
 σ 가 커져 제대로 엣지 추출을 못하는 경우를
방지한다.

이미지에서의 미분

LoG는 이미지를 2차 미분함 \longrightarrow 미분은 연속함수에서만 가능 \longrightarrow 이미지에서의 미분 정의

연속함수에서의 미분 $f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$

이미지에서의 미분 Δx 를 1로 정의 $f'(x) = f(x+1) - f(x)$

이미지에서의 2차 미분 $f''(x) = f(x+1) + f(x-1) - 2f(x)$

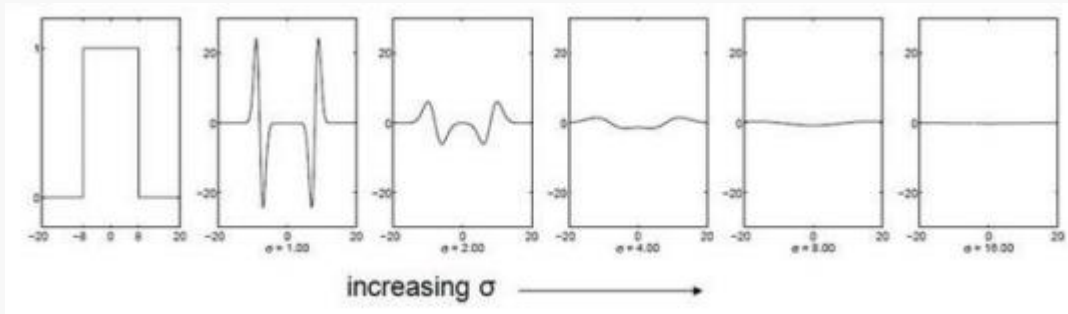
이미지의 2차 미분의 2차원 표현

$$\nabla^2 f = [f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1)] - 4f(x, y)$$

이미지에서의 미분은 옆 픽셀 값과의 차를 의미하며 옆 픽셀과의 기울기를 의미한다.

LoG(Laplacian of Gaussian)

그런데, σ 값이 다르다면?

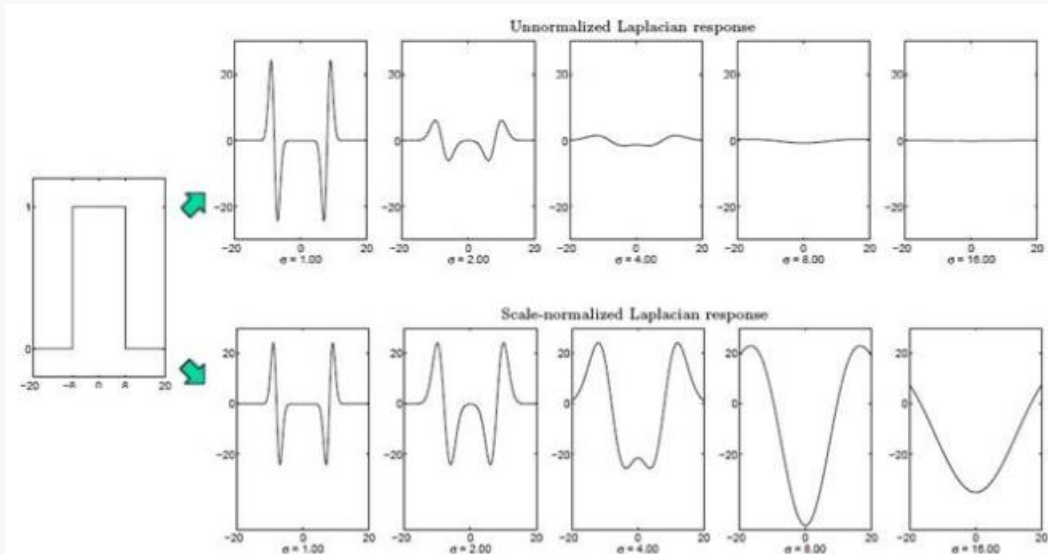


맨 왼쪽이 입력 함수이다.
LoG의 σ 를 키울수록 엣지에 대한 반응이 약해
엣지 검출 함수로 사용하기 힘들어진다.

LoG는 scale이 커지는 경우
결과가 좋지 않은 성질을 가짐

Scale-normalized LoG 사용

LoG에 σ 를 곱하는 방식



같은 입력에 대해, 같은 σ 인 경우
위는 LoG 연산,
아래는 scale-normalized LoG 연산이다.

Scale-normalized LoG를 사용한 경우
 σ 가 커져 제대로 엣지 추출을 못하는 경우를
방지한다.

DoG(Difference of Gaussian)

LoG는 연산이 너무 복잡 \rightarrow DoG로 대체해서 사용

DoG 연산

DoG 이미지 하나를 얻기 위해
한 Octave 내의 σ 값이 인접한 2개의 이미지 필요
두 이미지의 차를 구해 DoG 이미지를 구함
반복 수행 \rightarrow Octave 마다 DoG 이미지 획득

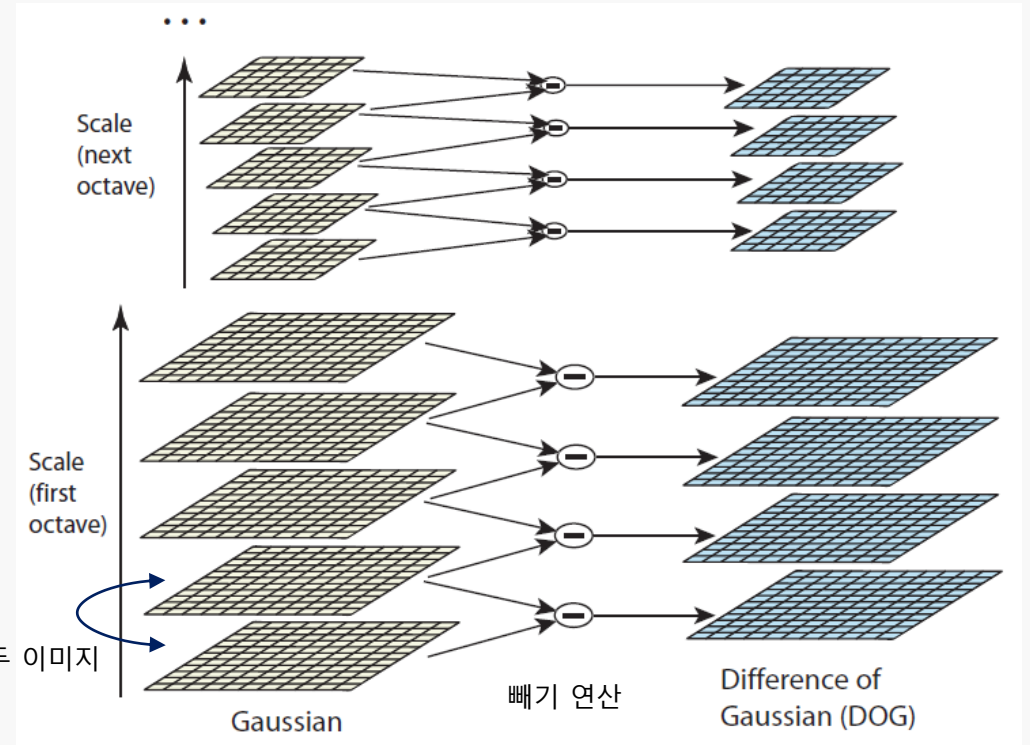
어떻게 DoG를 통해 keypoint를 추출할까?

Image blurring은 노이즈나 엣지, 코너처럼
주변 픽셀에 비해 값이 크게 달라지는 지점을 완화시킴



σ 가 다르면 keypoint 부분이 다른 값으로 표현됨
flat 부분은 주변 픽셀과 값이 비슷하므로 영향 X

\rightarrow 서로 다른 두 σ 를 가진 blur 영상의 차를 구하면 keypoint가 도출됨



LoG=DoG??

DoG가 어떻게 LoG를 대체할 수 있는지 수식을 통해 알아보자.

열 확산 방정식에 의해 다음이 성립한다.

$$\frac{\partial G}{\partial \sigma} = \sigma \nabla^2 G \quad (G : \text{Gaussian filter})$$

이 방정식을 만족시키는 필터가 Gaussian 필터이므로
우리는 앞에서 Gaussian 필터를 사용했다.

좌변의 식은 미분의 성질을 이용하면 아래와 같이 근사된다.

$$\frac{\partial G}{\partial \sigma} \approx \frac{G(x, y, k\sigma) - G(x, y, \sigma)}{k\sigma - \sigma} \quad \text{즉,} \quad \sigma \nabla^2 \approx \frac{G(x, y, k\sigma) - G(x, y, \sigma)}{k\sigma - \sigma}$$

여기서 양변을 우변의 분모로 곱하면 다음과 같다.

$$(k-1)\sigma \nabla^2 G \approx G(x, y, k\sigma) - G(x, y, \sigma)$$

이 때, 좌변은 scale-normalized LoG에 k-1이 곱해진 식을, 우변은 DoG 연산을 수식으로 나타낸 것이다.

우리가 뒤에서 얻고자 하는 keypoint는 이미지의 극값이다.

→ (k-1)은 우리의 목적에 아무 영향을 끼치지 않음
따라서, scale-normalized LoG와 DoG는 비슷한 성능을 보임

SIFT 알고리즘

03 Keypoint 찾기

Keypoint 찾기

먼저 이미지 내의 극값들의 대략적인 위치를 찾는다.

필요한 준비물

대략적인 극값을 찾기 위해서는
인접한 scale를 가진 DoG 이미지 세 장이 필요하다.

그 중 중간 scale를 가진 DoG 이미지에서 극값을 찾는다.

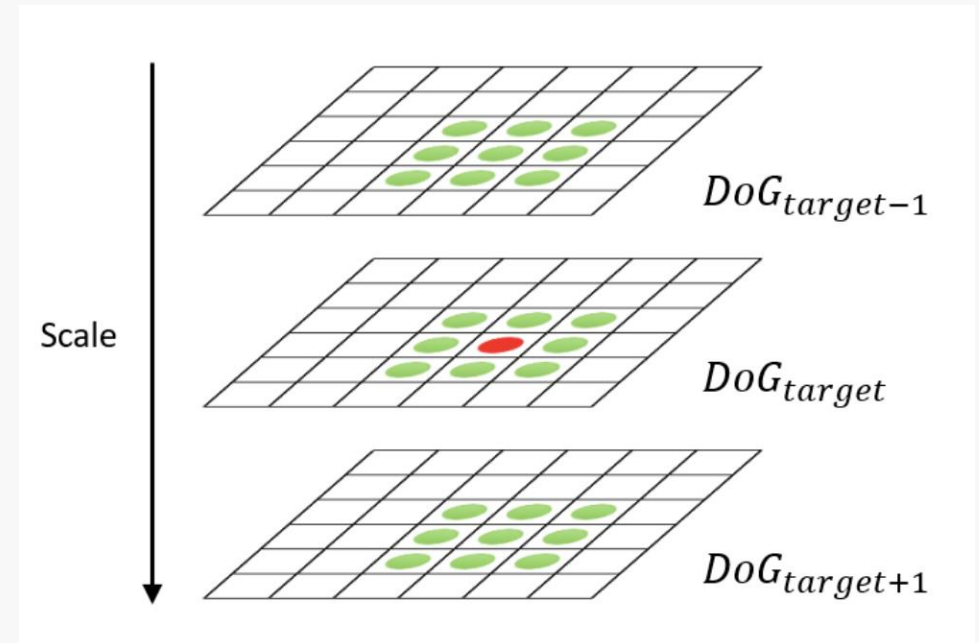
1단계

중간 scale를 가진 DoG 이미지에서
극값인지 판별하고자 하는 픽셀 주변 8개의 값과

Scale이 다른 두 이미지에서
극값인지 판별하고자 하는 픽셀과 가까운 픽셀을 9개씩,
18개를 확인한다.

➡ 총 26개의 픽셀 확인

2단계 극값인지 판별하고자 하는 픽셀이 1단계에서 확인한 픽셀 중
가장 크거나 가장 작으면 극값으로 판정

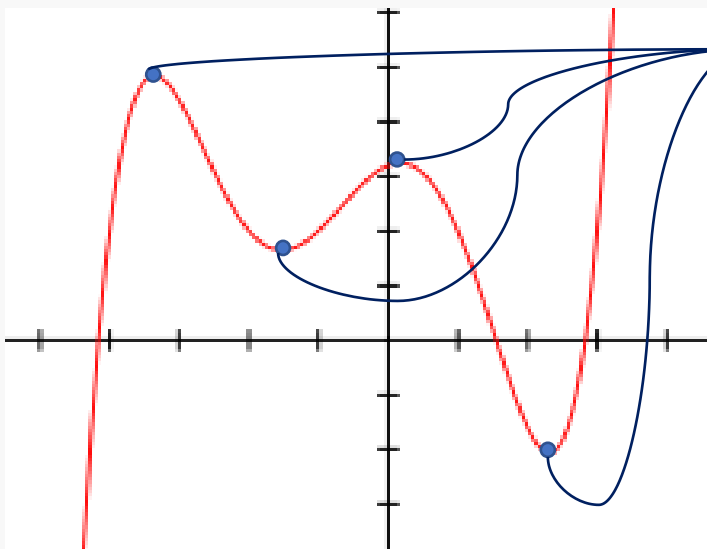


DoG 이미지

빨간 점 : 극값인지 확인하고자 하는 픽셀

초록 점 : 극값을 찾기 위해 확인해야 할 픽셀

왜 주변 값보다 크거나 작으면 극값일까?



5차 함수 그래프

극값

오른쪽 5차 함수의 극값을 살펴보면
주변에 비해 최대 혹은 최소값을 가지는 모습을 볼 수 있다.

극값의 기울기가 0이기 때문에 생기는 특성

→ DoG 이미지에서 극값을 찾을 때 이러한 특성을 활용

하지만 DoG 이미지의 좌표가 불연속이기 때문에
우리가 구한 극값의 위치는 정확한 위치가 아님

→ 테일러 2차 전개를 사용해 더 정확한 극값의 위치 예측

$$D(X) = D + \frac{\partial D^T}{\partial X} X + \frac{1}{2} X^T \frac{\partial^2 D}{\partial X^2} x \quad (D : Dog Image, X : (x, y, \sigma)^T)$$

이 때, 극값이 되는 점은 $\hat{x} = -\frac{\partial D^{-1}}{\partial x} \frac{\partial D}{\partial x'}$, 극값에서의 DoG는 $D(\hat{x}) = D + \frac{1}{2} \frac{\partial D^T}{\partial x} \hat{x}$

SIFT 알고리즘

04 나쁜 keypoint 제거

낮은 contrast 값을 가지는 keypoint 제거

왜 제거할까?

극값은 어디서나 존재 가능 → 실제로는 flat한 지역도
주변값에 비해 최소 최대값이면 극값으로 판정됨



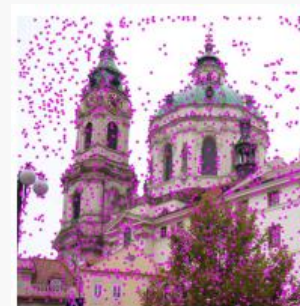
제거

← Flat한 지역은 DoG 값이 작기 때문에
극값에서의 DoG 값이 낮은 부분은 keypoint가 아닌 Flat으로 판단

극값에서의 DoG는 $D(\hat{x}) = D + \frac{1}{2} \frac{\partial D^T}{\partial x} \hat{x}$, (\hat{x} 는 극값)

$D(\hat{x})$ 가 특정 값보다 작으면 제거 (일반적으로 사용하는 기준은 0.03)

오른쪽 이미지를 보면
낮은 contrast 값을 가지는 값을 제거해주었더니
Flat에 위치한 점들이 많이 제거되고,
비교적 물체의 특징을 잘 나타내는 keypoint만 남았다.



낮은 대비의 점 제거 전



낮은 대비의 점 제거 후

엣지 위에 존재하는 keypoint 제거

왜 제거할까?

DoG 필터로 keypoint를 찾을 때 노이즈를 엣지로 판단할 수 있음

- 즉, 모든 엣지를 keypoint로 사용하기에는 약간의 위험성이 따름
- 엣지에 노인 keypoint 제거

엣지를 어떻게 찾을까?

Flat 지역은 어느 방향으로도 그래디언트가 크지 않음
엣지 지역은 한 방향으로만 그래디언트가 큼
코너 지역은 여러 방향에서 그래디언트가 큼



한 방향으로만
그래디언트가 큰 keypoint를 찾자

그래디언트 계산을 위한 헤시안 행렬 사용

헤시안 행렬 : 2차 미분 계수로 이루어진 행렬

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$$

$$D_{xx} = D(x+1, y) + D(x-1, y) - 2D(x, y)$$

$$D_{yy} = D(x, y+1) + D(x, y-1) - 2D(x, y)$$

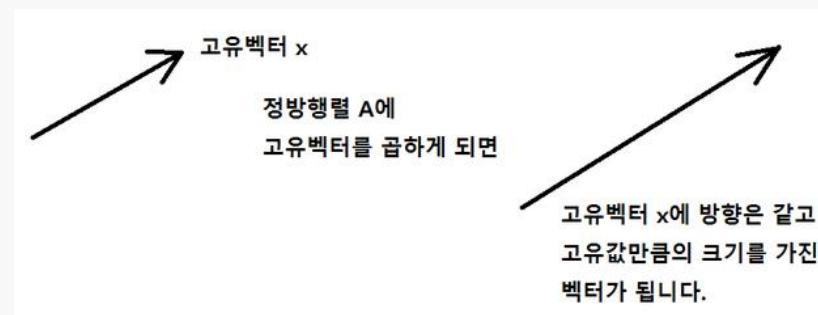
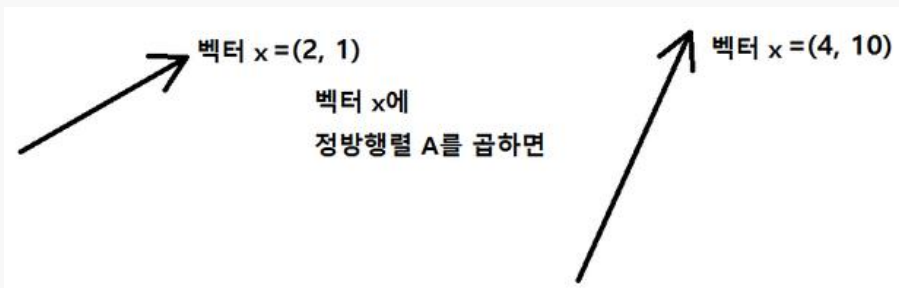
$$D_{xy} = \frac{D(x-1, y-1) + D(x-1, y) - D(x+1, y-1) - D(x-1, y+1)}{4}$$

엣지 위에 존재하는 keypoint 제거

고유값과 고유벡터 개념

일반적으로 정방행렬과 벡터를 곱하면 크기와 방향이 바뀐다.

만약 $N \times N$ 행렬과 고유벡터를 곱하면 크기는 고유값으로 변하고 고유벡터의 방향은 변하지 않는다.



즉, $N \times N$ 행렬이 방향, 고유값이 크기를 나타낸다.

우리의 경우에 헤시안 행렬이 그래디언트의 방향, 헤시안 행렬의 고유값이 그 방향으로의 크기가 된다.

$N \times N$ 행렬은 N 개의 고유값과 고유 벡터를 가진다. \rightarrow 우리는 2개의 고유값과 고유 벡터를 얻을 수 있다.

헤시안 행렬에서

고유값 중 한 값만 크다면? \rightarrow 한 방향으로만 변화가 큼

고유값 두 값이 모두 크다면? \rightarrow 두 방향으로 변화가 큼

고유값 두 값이 모두 작다면? \rightarrow 두 방향 모두 변화가 미미함

\rightarrow 엣지 \rightarrow 두 값의 비율 큼

\rightarrow 코너 \rightarrow 두 값의 비율 작음

\rightarrow 플랫 \rightarrow 두 값의 비율 작음

엣지 위에 존재하는 keypoint 제거

고유값과 고유벡터 적용

$D_{xx} + D_{yy} = \alpha + \beta, D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta$ $\alpha\beta$ 는 고유값으로 α 를 고유값 중 더 큰 값으로 정의한다.

첫 식은 헤시안 행렬의 대각합 $Tr(H)$ 이고, 두 번째 식은 헤시안 행렬의 행렬식 $Det(H)$ 이다.

이렇게 행렬의 대각합과 행렬의 행렬식을 이용하면 두 고유값의 합과 곱을 구할 수 있다.

고유값을 직접 구하는 대신 간단한 수식 이용

모든 헤시안 행렬에 대해 고유값을 구하기엔 너무 과정이 복잡해진다.

따라서 아래와 같이 행렬식 $Det(H)$ 과 대각합 $Tr(H)$ 을 이용한다.

$$\frac{Tr(H)^2}{Det(H)} = \frac{\alpha + \beta^2}{\alpha\beta} = \frac{(\gamma + 1)^2}{\gamma}, (\gamma = \frac{\alpha}{\beta}, \alpha > \beta) \quad \alpha > \beta \text{이므로 항상 } \gamma > 1$$

이 때 $\frac{(\gamma+1)^2}{\gamma}$ 는 γ 이 1보다 클 때 증가 함수이며 일대일 함수이다.

→ γ 를 직접 구하지 않아도 γ 제어 가능

우리가 챙겨야 할 keypoint의 γ 은 일반적으로 10 미만으로 제시된다.




SIFT 알고리즘

05 Keypoint에 방향 할당

Keypoint에 방향을 할당하는 방법

전체적인 틀 : Keypoint 주변 픽셀들의 그래디언트 방향과 크기를 모음

1단계 : keypoint 주변에 특정 크기의 윈도우 생성 하고 Gaussian blurring 한다. → Keypoint에 가까운 픽셀일수록 더 높은 가중치



- 논문에서 제시하는 값은 1.5σ 로 blur
- 윈도우의 크기는 일반적으로 16x16 사용

2단계 : 윈도우 내의 모든 픽셀의 그래디언트 방향과 크기를 계산한다.

크기 계산 $m(x, y) = \sqrt{(L(x+1) - L(x-1))^2 + (L(x, y+1) - L(x, y-1))^2}$

방향 계산 $\theta(x, y) = \tan^{-1} \frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)}$

Keypoint에 방향을 할당하는 방법

3단계 : 히스토그램을 만든다. 360도의 방향을 36개의 bin으로 나눈다.

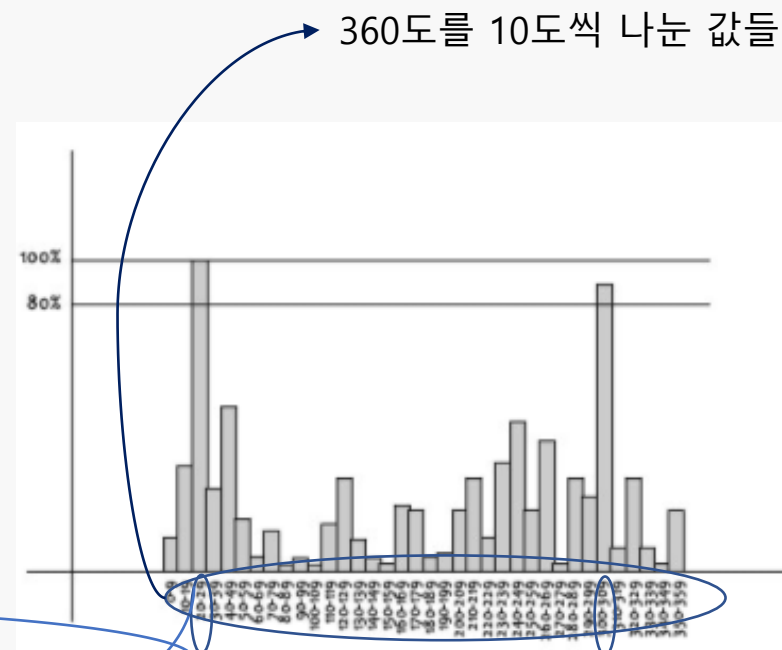
4단계 : 각 픽셀의 방향에 해당하는 bin에 그 픽셀의 크기만큼 할당한다.

여기까지 진행하면 오른쪽 하단과 같은 히스토그램을 얻을 수 있다.

5단계 : 가장 높은 수치를 보이는 값을 해당 keypoint의 방향으로 할당한다.

6단계 : 가장 높은 수치를 보이는 값을 100%라고 한다면 다른 방향 구간에서 80% 이상의 수치를 보인다면 새 keypoint의 방향으로 할당한다.

→ 위치와 크기는 같고 방향이 다른 keypoint가 됨



SIFT 알고리즘

06 SIFT descriptor 산출

Descriptor를 구하는 방법

이미지의 keypoint, 즉 이미지 특징을 구분하기 위한 [지문](#)과 같은 정보

Descriptor의 컨셉

Keypoint 주변 픽셀들의 그래디언트 방향을 벡터로 저장해 Descriptor로 사용

→ 전 단계에서 keypoint의 방향을 할당할 때 사용한 방법과 매우 유사

1단계

Keypoint 주변에 16x16의 윈도우를 만들고 Gaussian Filter로 1.5σ 만큼 blurring

2단계

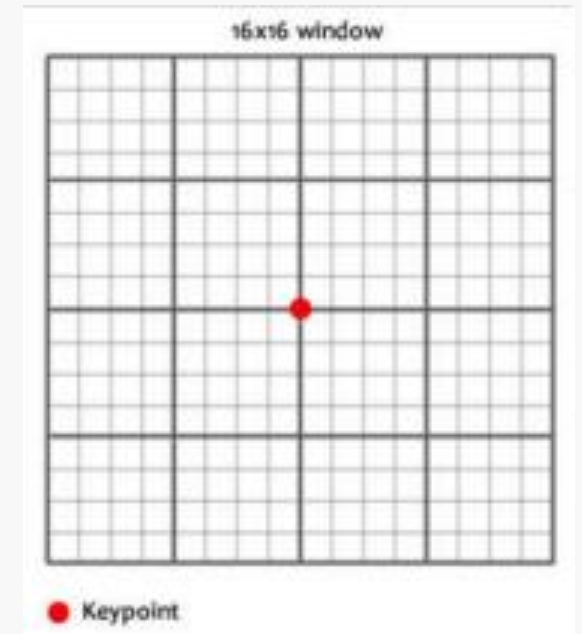
16x16 윈도우를 다시 4x4 윈도우로 나눈다.

3단계

각 4x4 윈도우 내의 모든 픽셀에서 크기와 방향을 구한다.

$$\text{크기 계산} \quad m(x, y) = \sqrt{(L(x+1) - L(x-1))^2 + (L(x, y+1) - L(x, y-1))^2}$$

$$\text{방향 계산} \quad \theta(x, y) = \tan^{-1} \frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)}$$



Keypoint는 픽셀과 픽셀 사이에 위치

Descriptor를 구하는 방법

4단계

각 4x4 윈도우 안의 모든 픽셀의 크기와 방향을 이용해서 4x4 윈도우 마다 히스토그램을 작성한다.

이 때, 360도를 8구간으로 나누어 bin을 설정한다.

→ 16x16에 4x4 윈도우가 16개 존재하므로
16개의 히스토그램 획득 (오른쪽 하단 이미지 형식)

5단계

3단계에서 계산한 픽셀의 방향에 Keypoint에 할당한 방향값을 빼준다.

→ Keypoint에 대해 상대적인 방향값이 되어 회전 불변성 만족

6단계

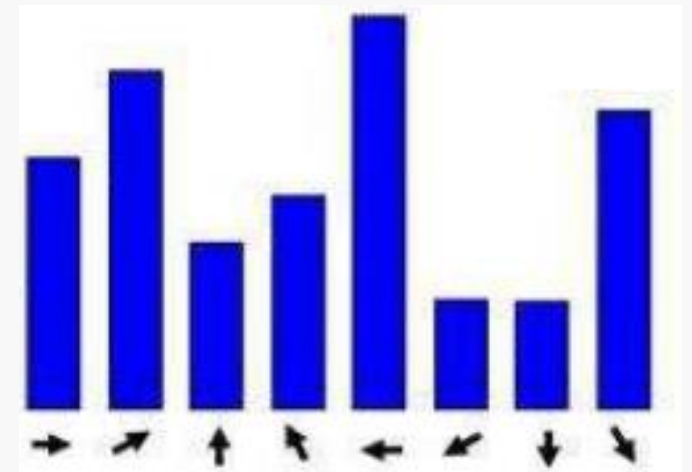
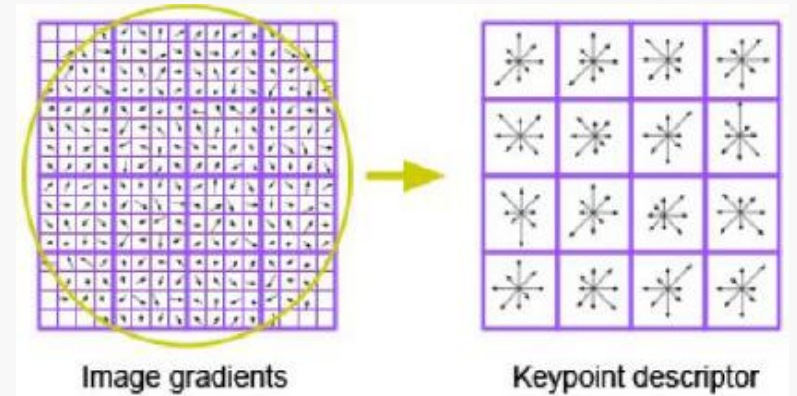
각 픽셀의 방향에 해당하는 bin에 픽셀의 크기만큼 값을 채움

7단계

16개의 히스토그램에서 각 8개의 bin이 존재하므로 128개의 숫자 값을 얻음
이를 128차원의 벡터로 만든 후 정규화(벡터의 원소를 벡터 크기로 나눔)

→ 최종적인 SIFT descriptor

→ 밝기 불변성 만족



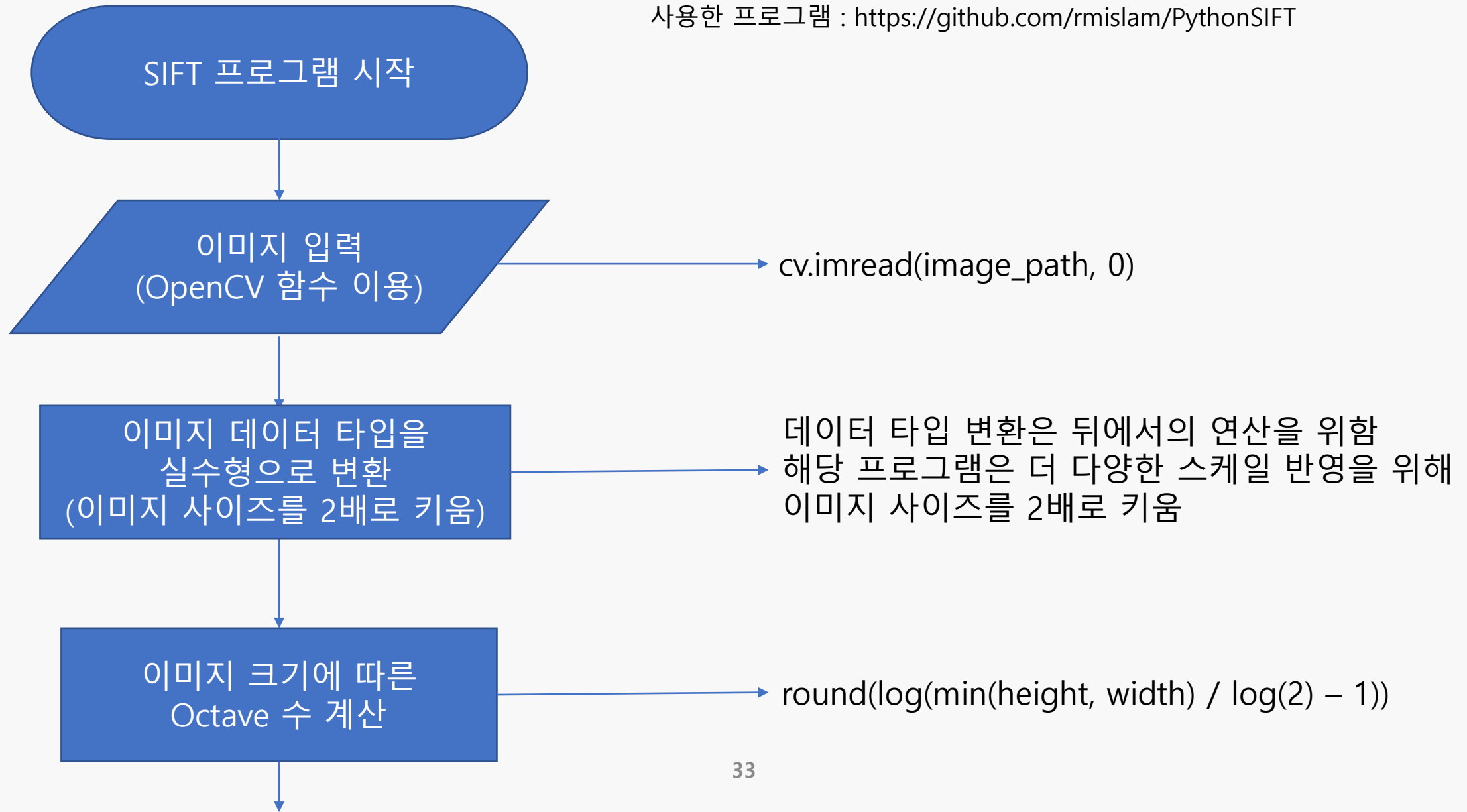
이러한 히스토그램을 16개 얻는다.

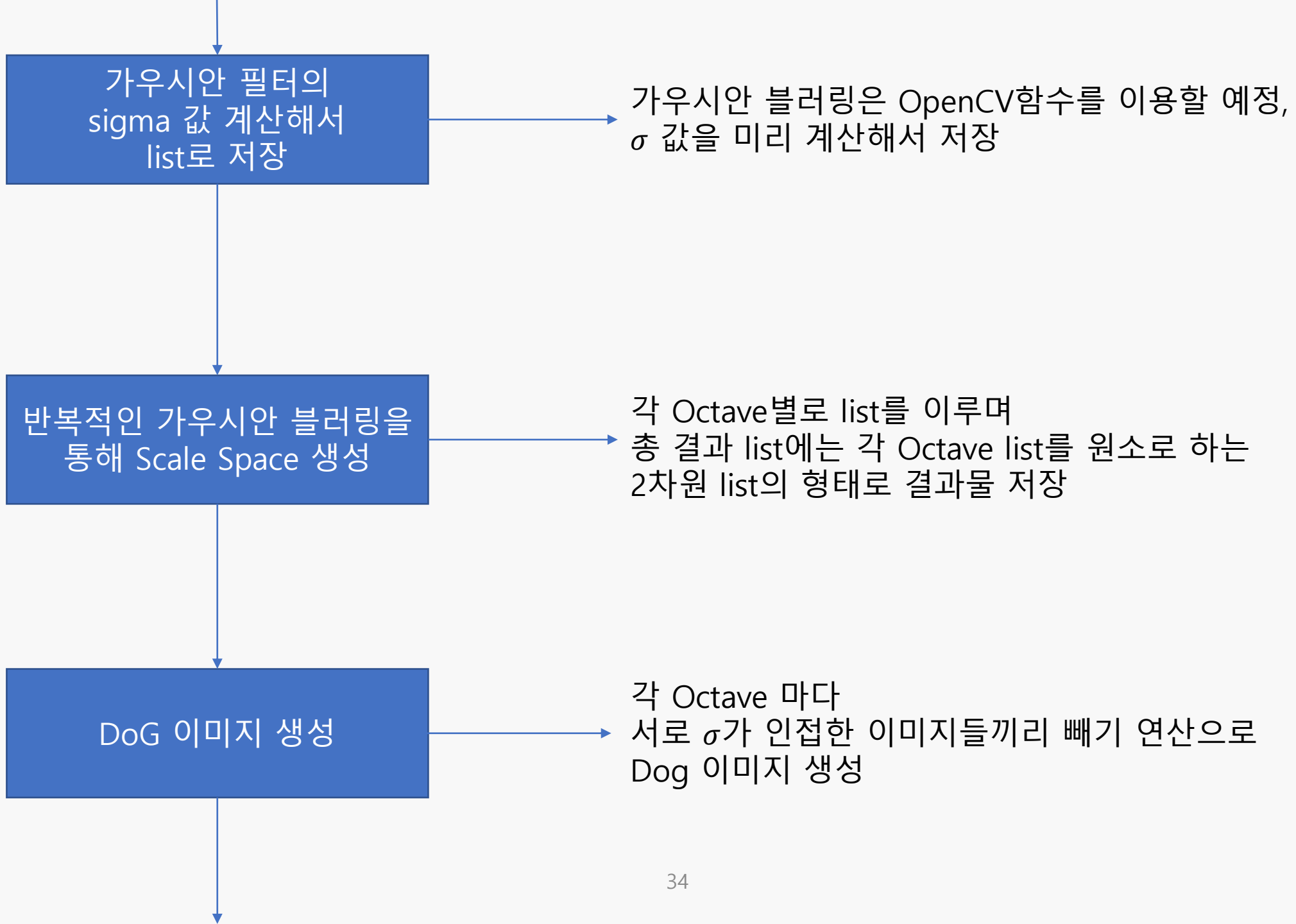
SIFT 알고리즘

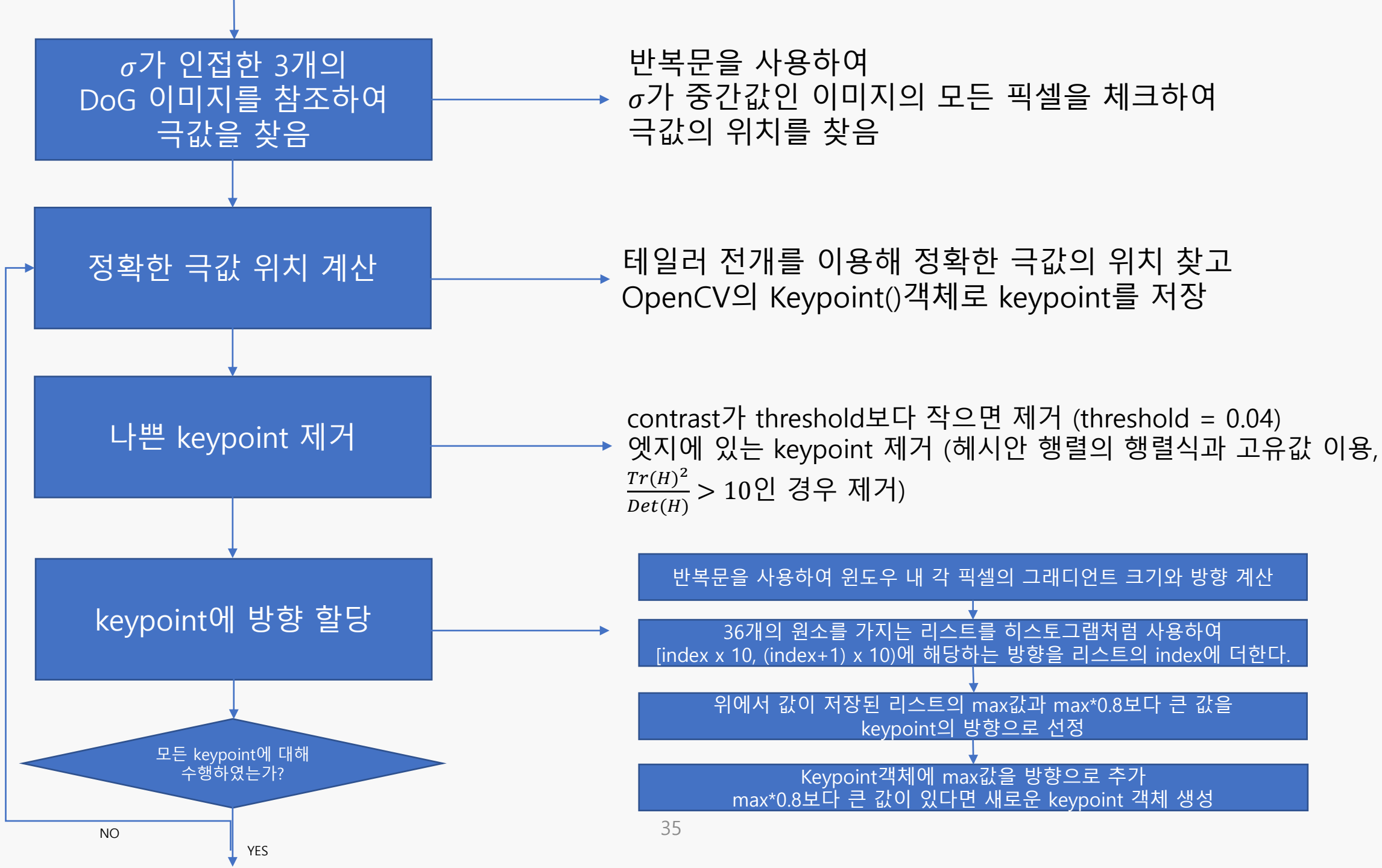
07 프로그램 순서도

프로그램 순서도

사용한 프로그램 : <https://github.com/rmislam/PythonSIFT>







descriptor 계산

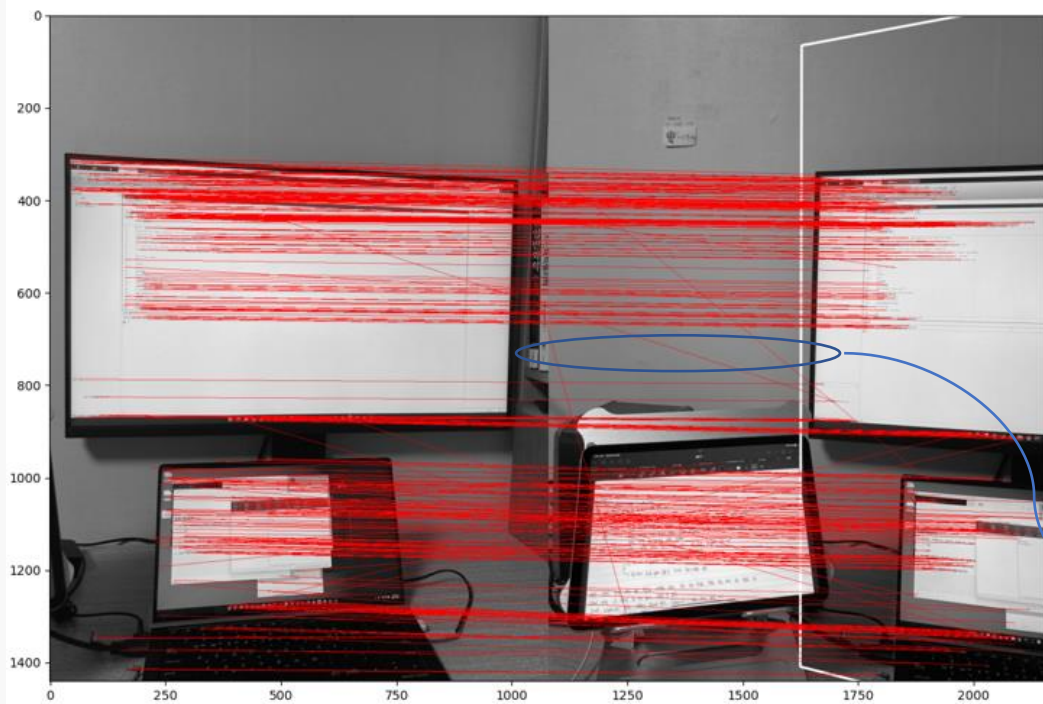
반복문을 사용하여 윈도우 내 각 픽셀의 그래디언트 크기와 방향 계산

윈도우 내의 그래디언트에 keypoint의 방향만큼 빼기 연산

list에 각 윈도우 내의 픽셀의 그래디언트의 크기만큼
해당 방향을 가리키는 index에 저장

이렇게 얻은 list를 descriptor로 사용

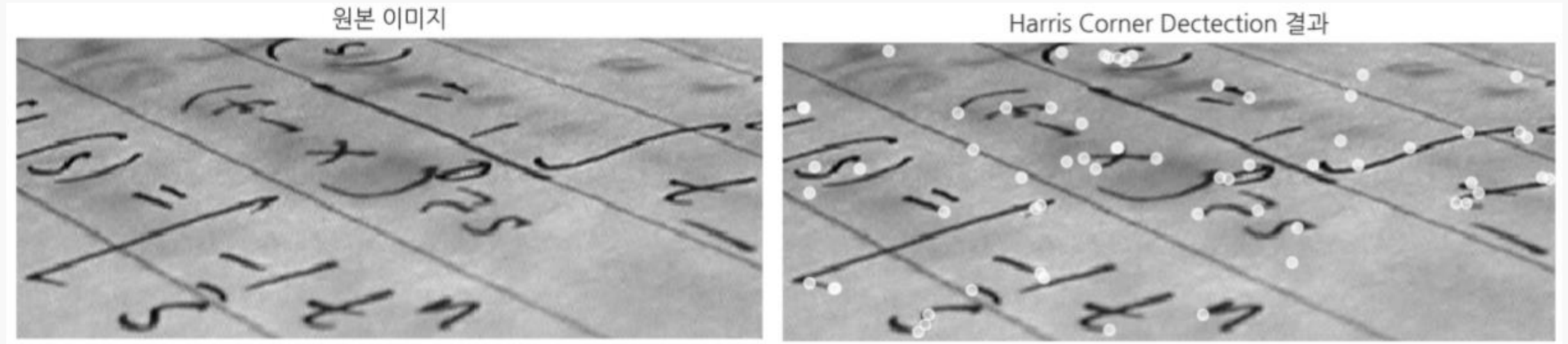
descriptor 벡터를 정규화(해당 벡터의 크기값으로 나눔)



결과물은
대체로 feature matching이 잘 되는 모습이지만,
일부 에러가 존재한다.

02 Harris 코너 검출

목표 : 이미지의 코너점을 검출하는 것

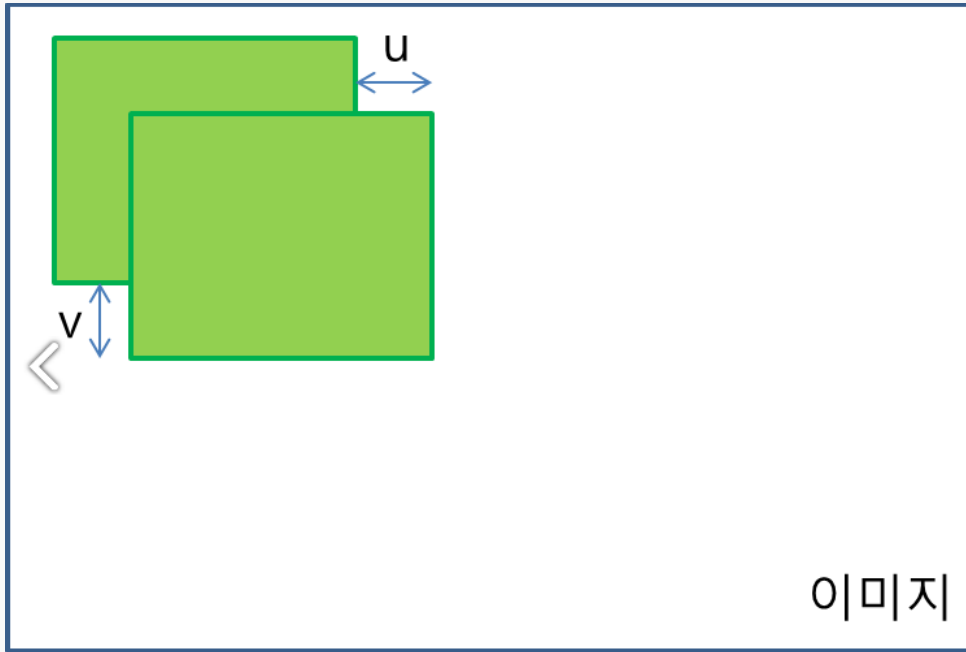


코너점 두 방향 이상에서 픽셀의 변화량이 큰 부분
이미지에서 가장 중요한 정보를 담고 있음

Harris 코너 검출

01 기본 원리

기본 원리



한 픽셀을 중심에 놓고 작은 윈도우를 만듦

이 윈도우를 x축 방향으로 u만큼, y축 방향으로 v만큼 이동시킨다.

그 다음 윈도우 내의 픽셀 값들의 차이의 제곱합을 구한다.

$$E(u, v) = \sum_{(x_k, y_k) \in W} [I(x_k + u, y_k + v) - I(x_k, y_k)]^2$$

픽셀 값이 얼마나 변화했는지를 계산
코너점이라면 x축과 y축 방향 모두 많이 변화했을 것 → E 값이 크면 코너점이라고 생각

Harris 코너 검출

02 테일러 확장

테일러 확장

기본 원리에서 구한 $E(u, v)$ 에 테일러 확장을 적용하면 아래와 같다.

$$\begin{aligned} E(u, v) &= \sum_{(x_k, y_k) \in W} [I(x_k + u, y_k + v) - I(x_k, y_k)]^2 \\ &\approx \sum_{(x_k, y_k) \in W} \left(\left(\frac{\partial I}{\partial x} \right)^2 u^2 + \left(\frac{\partial I}{\partial y} \right)^2 v^2 + 2 \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} uv \right) \end{aligned}$$

그리고 이것을 행렬식으로 나타내면 아래와 같다.

$$E(u, v) = \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} \sum \left(\frac{\partial I}{\partial x} \right)^2 & \sum \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} \\ \sum \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} & \sum \left(\frac{\partial I}{\partial y} \right)^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

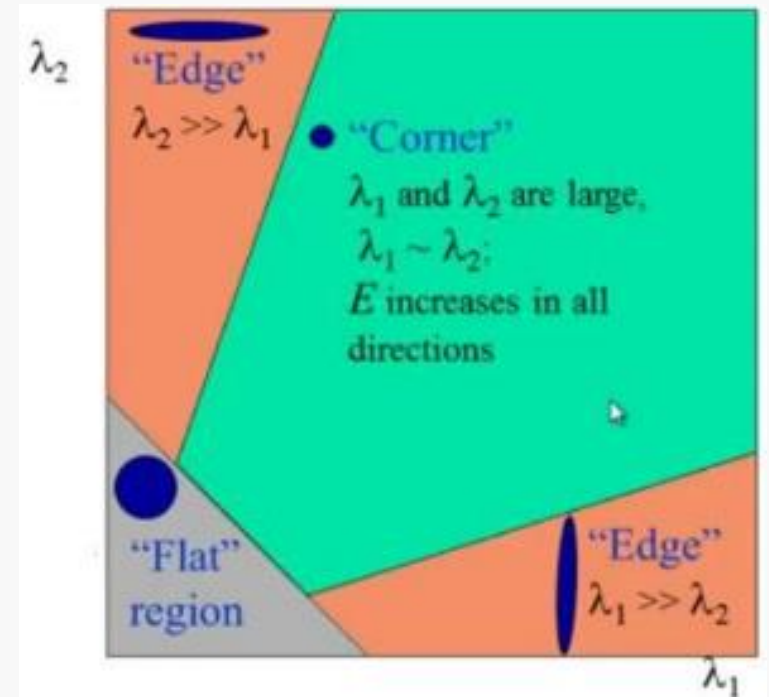
테일러 확장

$$E(u, v) = [u \quad v] \begin{bmatrix} \sum \left(\frac{\partial I}{\partial x} \right)^2 & \sum \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} \\ \sum \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} & \sum \left(\frac{\partial I}{\partial y} \right)^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

위의 식에서 $E(u, v)$ 가 크기 위해서는 가운데 행렬의 크기가 커야 한다
(u 와 v 는 윈도우의 이동값으로 값이 정해져 있음)

따라서 가운데 행렬을 고유값 분해해서 2개의 고유값을 얻을 수 있다.
앞 부분 SIFT에서와 같이 고유값은 그래디언트 방향의 크기를 나타낸다.

→ 두 고유값이 모두 커야 코너점이다.



그래프의 축을 이루는 λ_1, λ_2 는 각 고유값을 나타내며 두 값이 모두 작으면 Flat, 둘 중 하나만 크면 Edge, 두 값이 모두 커야 코너라는 것을 그림을 통해 나타낸다.

Harris 코너 검출

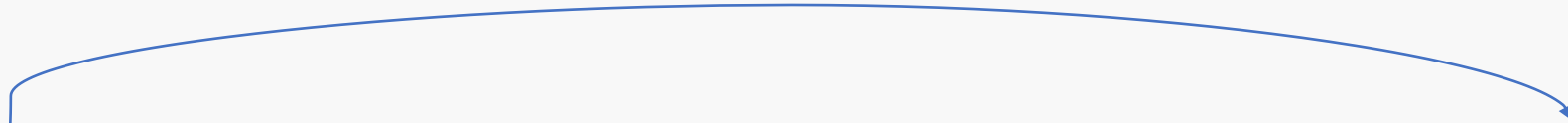
03 코너점 판별

고유값을 이용한 코너점 판별

$$R = \det(M) - k(\text{trace}(M))^2$$

고유값을 모든 경우에 대해 구하기엔 연산이 복잡하니 대체 방법으로 위 수식을 이용한다.

M은 앞에서 다룬 식의 가운데 2x2 행렬을 뜻하고,
k는 입력 파라미터이며 값이 작을수록 더 많은 픽셀을 코너점으로 인식한다.


$$\begin{bmatrix} \sum \left(\frac{\partial I}{\partial x}\right)^2 & \sum \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} \\ \sum \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} & \sum \left(\frac{\partial I}{\partial y}\right)^2 \end{bmatrix}$$

R이 특정 기준점(threshold)보다 큰 경우 코너로 판별한다.

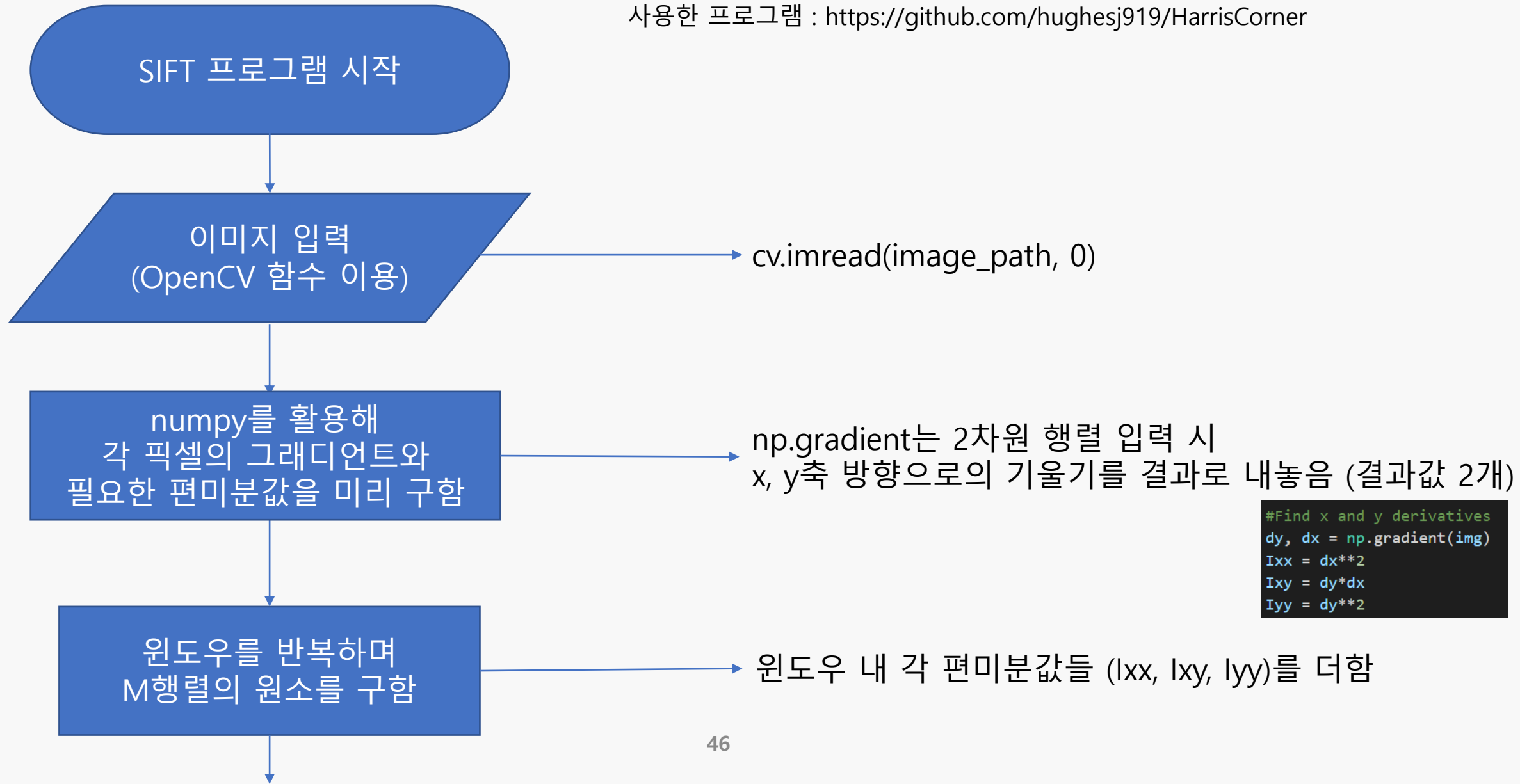
한편, 행렬식은 두 고유값의 곱과 같고, 대각합은 두 고유값의 합과 같다.

Harris 코너 검출

04 프로그램 순서도

프로그램 순서도

사용한 프로그램 : <https://github.com/hughesj919/HarrisCorner>



고유값을 이용한
코너점 판별

대각합과 행렬식을 이용해
 $R = \det(M) - k(\text{trace}(M))^2$ 계산해 threshold보다 크면
코너점으로 판별한다.
k는 0.2, threshold는 10000으로 초기화해주었다.



feature들이 주로 코너 부분이 찍혀있지만,
글자나 아이콘을 이루는 부분은 코너점으로 대부분 덮여있다.

배경색과의 밝기 차이가 커서 그런 것으로 예상할 수 있지만
같은 색차이가 큰 지역인 모니터와 배경은 코너점이 검출되지
않은 것은 의문이다.

책상 위의 코너점들은 책상의 울퉁불퉁한 무늬를 따라
검출된 것으로 예상된다.

THANK YOU –
감사합니다.