Image Feature Extraction

17010826 김성민

SIFT(Scale Invariant Feature Transform)

- 이미지의 크기와 회전에도 변하지 않는 특징을 추출하는 알고리즘
- 서로 다른 두 이미지에서 SIFT 특징을 각각 추출해 비슷한 특징끼리 매칭 => 두 이미지에서 대응되는 특징을 찾을 수 있음
- 장점 : 크기와 방향이 달라도 일치되는 부분을 잘 찾아서

  매칭해줌 => 파노라마에서도 사용됨

SIFT 알고리즘 진행 과정

1. Scale space 만들기
2. Different of Gaussian(DoG) 연산
3. Keypoint 찾기
4. 나쁜 Keypoint 찾기
5. Keypoint에 방향 할당
6. 최종적인 SIFT 특징 산출

Scale Space를 만들기 전에

- Scale Space는 SIFT를 통해 나온 개념이 아닌 기존에 있던 개념
- Scale이란 이미지를 보는 척도를 뜻한다.
    - 가까이서 세세히 보는 것 -> 작은 스케일
    - 멀리서 전체적으로 보는 것 -> 큰 스케일

그림1. 큰 스케일로 보는 것과 작은 스케일로 보는 것. 참고로 이 사진은 크로아티아 두브로브니크 성벽 투어 중 찍은 것이다.

Scale Space란?

- 우리가 어떤 장면을 보고 해석할 때 전체적인 틀 파악(큰 스케일)세부적인 내용을 파악(작은 스케일)하는 것 모두 필요
- 따라서 성공적인 이미지 처리를 위해 “Scale”개념을 도입
- 이 개념을 반영하기 위한 것이 “Scale Space”
- 이미지 스케일을 변화시키는 가장 간단한 방법

  => 이미지 피라미드 : 이미지를 확대, 축소하는 것
- 이미지의 사이즈는 그대로 유지, 이미지를 블러링(blurring) 시킴으로 다른 스케일의 이미지를 얻을 수도 있다.
- 그렇다면 이미지 블러링은 무엇일까?

오른쪽으로 갈수록 디테일이 점점 떨어진다.
(오른쪽으로 갈수록 큰 스케일)

이미지 블러링(Image Blurring)

- 이미지 처리, CV에 사용되는 기본적인 이미지 변형 방법
- 노이즈 제거에 유용하며 이미지를 더 흐리게(부드럽게, 잡음 제거, 고주파 차단) 보이도록 만드는 효과를 낸다.
- 이미지 상에서의 픽셀의 값은 공간적으로 느리게 변함

  즉, 픽셀 간의 상관관계가 크다.
- 노이즈의 경우는 픽셀의 상관관계가 없다.
- 이를 주변 픽셀의 값을 사용해 노이즈의 값을 완화.
- 이미지 내의 물체의 경계도 노이즈처럼 주변 값을 이용해 경계가 흐릿해 짐 (경계도 노이즈처럼 주변 픽셀이 급격하게 변함)

이미지 블러링(Image Blurring)

- 이미지와 필터의 2차원 컨볼루션 계산을 통해 이루어짐
- 대표적인 4가지 필터
  - Averaging : 주변 값을 평균을 내주는 필터 사용
    (ex) 1/9로만 이루어진 3x3 필터)
  - 중간값 : 픽셀 값을 박스 내 픽셀 값들의 중간 값으로 대체
    (박스 내의 픽셀 값을 일렬로 세워서 그 중 중간 값으로 대체)
  - 가우시안 : 필터 내 가중치가 중심일수록 강함
    중심 픽셀 값은 바로 주변 픽셀의 영향을 가장 크게 받음
  - Bilateral : 위와 마찬가지로 노이즈 제거, but edge는 살림
    edge를 살리기 위해 유사성 함수 도입
    (edge : 이미지 내 물체의 경계, 윤곽선)

가우시안 필터 (Gaussian Filter)

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

- σ는 blurring의 정도를 결정, 클수록 더 많이 blur된다.
  이미지가 더 많이 blur되면 scale이 커짐
  $\Rightarrow$ σ를 스케일 파라미터라고 부른다.
- 가장 적합한 필터
- 왜 그럴까?
  - Scale Space Theory의 Heat Diffusion Equation을 만족하는 유일한 해
    뒤의 LoG = DoG?? 슬라이드에서 언급된다.
  - Gaussian Filter는 일반적인 가우시안 함수에서 평균 값이 0이라고 가정한다.
  - 0일 때 최대 값, 옆으로 퍼질수록 급격하게 0에 가까워짐
    ⇒Low Pass Filter의 역할을 수행할 수 있음

1단계 : Scale Space 만들기

- 원본 이미지를 2배, ½배, ¼배로 크기를 늘린다.
- 크기 조정된 이미지들을 blur되게 만든다.
  각 크기당 5개의 blur 이미지 필요(원본 1장 + blur처리한 이미지 4장)
- 점진적으로 blur된 이미지를 얻기 위해 σ를 k배 씩 늘린다.
  첫 σ는 1/√2, k는 √2로 설정
- 결과적으로 같은 사이즈 내 blur 정도가 다른 5장 존재하고 사이즈는 4가지 존재
- 여기서 같은 사이즈의 이미지 그룹을 “Octave” 라고 함
- 우리가 어떤 물체를 볼 때 작고 선명, 작고 흐릿함, 크고 선명, 크고 흐릿함 등의 현상을 모두 잡아내기 위한 과정

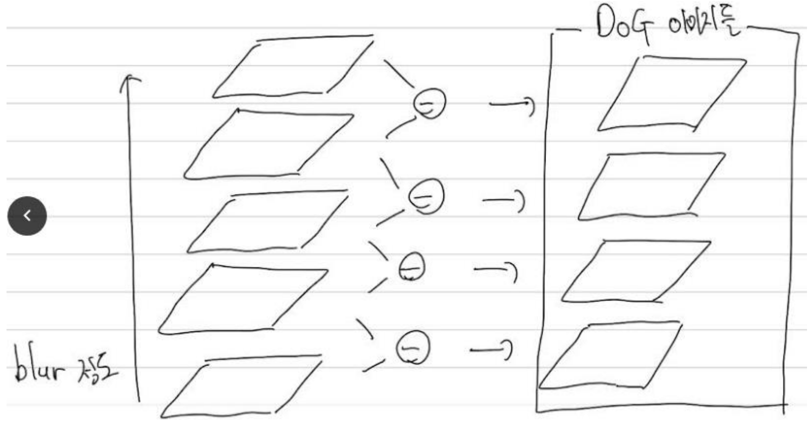
LoG(Laplacion of Gaussian)

- 2단계 DoG연산을 하기 전에 LoG먼저 알아보자.
- DoG와 LoG 모두 이미지 내의 흥미로운 지점, keypoint를 얻기 위한 과정 (엣지, 코너)
- LoG는 Gaussian Filter을 2차 미분한 식으로 얻어진다.

  LoG : $\nabla^2 G$ (G는 Gaussian Filter) = $-\frac{1}{\pi\sigma^4}[1-\frac{x^2+y^2}{2\sigma^2}]e^{-\frac{x^2+y^2}{2\sigma^2}}$
- 1차 미분의 경우 수평 수직 대각선 방향에 놓인 edge에 너무 민감하게 반응 => 2차 미분 사용
- 이 과정을 수행하면 edge 부분은 주변 픽셀과 값의 차이가 크므로 LoG 수행 결과 edge 부분은 비교적 값이 커짐
- 뿐만 아니라 주변에서 톡톡 튀는 값들은 큰 값을 가지게 되며 이 값들을 주목해야 할 부분으로 생각

2단계 : DoG 연산하기

- 그런데 LoG는 너무 복잡… DoG를 사용해보자.
- DoG는 한 octave 내에서 blur정도가 인접한 이미지들끼리 차를 구함으로 얻을 수 있다.
- LoG대신 DoG를 사용해도 성능은 비슷하고 연산량은 훨씬 줄어든다.
- 어떻게 저렇게 간단한 연산만 수행해도 엣지 코너 등의 feature가 추출될까?



⇒ 엣지 코너 등은 주변과의 픽셀 차가 크다. blur는 그 정도를 완화시키는 역할을 수행

⇒ 즉, blur의 정도마다 엣지 코너 등은 값의 차가 생긴다.

⇒엣지가 아닌 부분은 주변과 픽셀 차가 애초에 별로 없으니 blur를 해도 크게 값이 변하지 않고, blur 정도에 따른 값의 차도 미미하다.

LoG = DoG??

- 수학적으로 왜 LoG가 DoG로 대체될 수 있는지 알아보자.
- 열 확산 방정식에 의해 다음이 성립한다. 참고로 이 방정식을 만족하는 식이 가우시안이다.

  $\frac{\partial G}{\partial \sigma} = \sigma\nabla^2 G$ (G는 가우시안 필터, 우변은 LoG에 σ를 곱한 것)

  $\frac{\partial G}{\partial \sigma} \approx \frac{G(x,y,k\sigma)-G(x,y,\sigma)}{k\sigma-\sigma}$ (σ는 스케일 파라미터, k는 앞에서 사용한 blur의 척도를 다르게 하기 위해 사용한 변수)

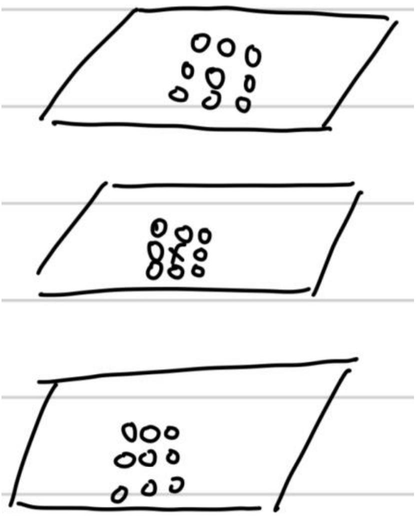
  양 변을 우변의 분모로 곱하면

  $(k-1)\sigma^2\nabla^2 G \approx G(x,y,k\sigma) - G(x,y,\sigma)$
- LoG는 $\nabla^2 G$이지만, 스케일 불변성을 위해 $\sigma^2\nabla^2 G$로 적용해 정규하해서 사용 => scale-normalized LoG라고 함
- 위 식을 통해 DoG는 scale-normalized LoG에 k-1과 같다는 것을 알 수 있음 (k-1은 우리가 뒤에서 찾을 극값에 영향을 주지 않음)

3단계 : Keypoint들 찾기

- 먼저 이미지 내의 극값들의 대략적인 위치를 찾는다.
- 한 픽셀에서 극값들을 결정하는 동안 동일 octave 내의 세 DoG 이미지가 필요하다.

  체크할 이미지 + scale이 한 단계 크고 작은 이미지
- 즉, 한 octave 내에서 2종류의 극값이 표시된 결과물을 얻을 수 있다.
- 체크할 픽셀의 주변 8개와 scale이 한 단계 크고 작은 이미지에서 체크할 픽셀과 가까운 9개씩 총 26개의 픽셀을 체크
- 만약 체크할 픽셀의 값이 가장 크거나 작으면 keypoint로 인정



저게 왜 극값일까?

- DoG 이미지는 blur 이미지들의 스케일 파라미터에 따른 변화량을 나타낸다. 즉, 블러 정도에 대해 이미 미분된 값들
- 우리가 찾은 keypoint는 두 측면에서 최저 혹은 최대 값이다.
- 첫 번째는 해당 이미지에서 주변 값들에 비해 변화량이 최저 혹은 최대이다.
- 두 번째는 서로 다른 blur의 정도를 가진 이미지들 사이에서도 변화량이 최저 혹은 최대이다.
- 우리는 다양한 scale을 참고해 keypoint를 선별해야 하므로 서로 다른 blur의 정도를 가진 이미지를 사용한다.

더 정확한 극값의 위치 찾기

- 이렇게 찾은 극값은 대략적인 것이고, 실제 극값은 픽셀과 픽셀 사이에 있을 가능성이 크다.
- 우리는 이 실제의 극값에 접근할 수 없다.
- 그래서 우리는 이 subpixel의 위치를 수학적으로 찾아야 한다.

  subpixel : 픽셀 사이에 위치한 무언가
- 여기서 우리는 테일러 2차 전개를 사용한다.

  $D(x) = D + \frac{\partial D^T}{\partial x}x + \frac{1}{2}x^T\frac{\partial^2 D}{\partial x^2}x$ (D: DoG 이미지, x는 $(\mathrm{x}, \mathrm{y}, \sigma)^T$)
- 극값 $\hat{x} = -\frac{\partial D}{\partial x}^{-1}\frac{\partial D}{\partial x}$, 극값에서의 DoG : $D(\hat{x}) = D + \frac{1}{2}\frac{\partial D^T}{\partial x}\hat{x}$

잠깐! 이미지 처리에서의 미분이란?

- 미분 가능한 함수는 연속이다
- 영상처리에 있어 데이터들은 sampling된 형태로, 연속 함수로 표현할 수 없다.
- 따라서 불연속(discrete)한 함수에 대해서도 연속을 정의해주어야 한다.
- 만약 1차원에서 생각한다면 미분은 아래와 같이 정의 가능하다.

  f'[n] = f[n+1] – f[n], 즉, [-1, 1] 커널을 사용하면 구할 수 있음
- 하지만 실제로는 중간값 근사가 효율이 더 좋기에 아래와 같이 사용한다.

  f'[n] = (f[n+1] – f[n-1]) / 2, 즉, [-1, 0, 1] 커널을 사용하면 구할 수 있음

잠깐! 이미지 처리에서의 미분이란?

- 이를 이용하면 2차 미분의 경우는 다음과 같아진다.
  $f''[n] = f[n+1] + f[n-1] – 2f[n]$
  ($f'[n] = f[n] – f[n-1]$로 계산 (방향성이 다르게))
- 엄밀하게 ½로 나누어져야 하는 부분이 존재하지만 우리가 이미지 처리에서 미분을 함으로 얻고자 하는 것은 극 값이고, ½는 극 값에 영향을 주지 않으므로 무시한다.
- 2차원으로의 확장
  $\nabla^2 Image = [f(x+1, y) + f(x-1 , y) + f(x, y+1) + f(x, y-1)] – 4f(x, y)$

4단계 : 나쁜 keypoint 제거하기

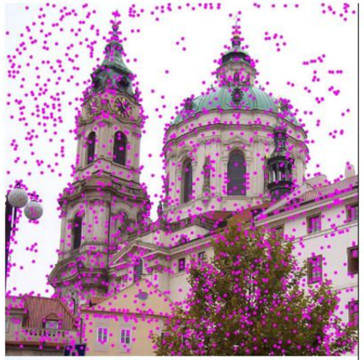
- 앞에서 얻은 keypoint들 중 활용가치가 떨어지는 것들은 제거해주어야 한다.
- 1. 낮은 contrast 값을 가지는 것들 제거

  우리가 구한 극값(keypoint)의 위치 : $\hat{x}$

  극값에서 DoG의 값 : $D(\hat{x}) = D + \frac{1}{2}\frac{\partial D^T}{\partial x}\hat{x}$

  $D(\hat{x})$이 특정값(threshold)보다 작으면 제거

  (제시된 기준은 0.03)



낮은 대비의 점 제거 전

낮은 대비의 점 제거 후

- 왜 이런 결과가?

  ⇒극값은 어디서나 존재 가능
  ⇒하지만 그 점에서의 DoG의 값은 천차만별
  ⇒DoG로 얻은 결과물이 엣지/코너에서 값이 크고 밋밋한 부분에서 값이 작음(앞 슬라이드에서 다룸)
  ⇒우리가 얻은 극값에도 마찬가지로 적용, 이미지 내 물체를 더 잘 표현할 수 있는 극값(keypoint)만 사용하자. (일정 값 이상의 극값만 사용하자)

4단계 : 나쁜 keypoint 제거하기

- 2. 엣지 위에 존재하는 keypoint 제거
  - DoG가 엣지를 찾아낼 때 약간의 노이즈에도 반응할 수 있음
  - 즉, 노이즈를 엣지로 반응할 수 있고, 이를 keypoint로 사용하기에는 약간의 위험성이 따름 => 엣지에 있는 keypoint 제거
  - 원리 : 엣지는 수직 or 수평 방향에서의 그래디언트만 큼

    flat한 부분은 어느 방향으로든 그래디언트가 크지 않음

    코너(이상적인 keypoint)는 수직, 수평 모두 그래디언트가 큼

    => 이를 이용해 엣지 부분을 걸러내자!

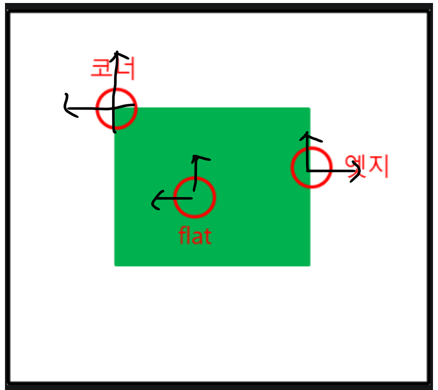


그림 10. flat 지역, 엣지, 코너 설명

- 수학적으로는?

$$D_{xx} = D(x+1, y) + D(x-1, y) - 2D(x, y)$$
$$D_{yy} = D(x, y+1) + D(x, y-1) - 2D(x, y)$$
$$D_{xy} = \frac{D(x-1, y-1) + D(x-1, y) - D(x+1, y-1) - D(x-1, y+1)}{4}$$
$$D_{xx} + D_{yy} = \alpha + \beta, D_{xx}D_{yy} - \left(D_{xy}\right)^2 = \alpha\beta$$

- 여기서 $\frac{\alpha}{\beta}$의 비율이 10보다 작아야 함 그 이상이면 edge 제거
- 근데 이게 무슨 뜻일까?

고유 값과 고유 벡터

- 헤시안 행렬 : 2차 미분 계수로 이루어진 행렬
- 앞에서 사용한 D_{xx}, D_{yy}, D_{xy} 는 그 원소이다. H=$\begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$
- 어떤 NxN 행렬 H는 N개의 고유 값과 고유 벡터를 가지고 있다.
- 만약 H와 고유 벡터를 곱하면, 그 고유 벡터에 해당하는
  고유 값 x 고유벡터가 결과물로 나온다.
  (고유 값이 크기, 고유 벡터가 방향)
- 한편, 행렬의 대각합은 두 고유 값의 합을, 행렬식은 두 고유 값의 곱을 나타낸다.

  => H의 경우 $D_{xx} + D_{yy} = \alpha + \beta, D_{xx}D_{yy} - \left(D_{xy}\right)^2 = \alpha\beta$

우리에게 고유 값과 고유벡터의 의미

- 우리의 경우에 대입하면 H행렬 (2차 미분 계수 행렬) 이 방향, 행렬식과 대각합으로 구할 수 있는 고유 값이 그 크기를 나타낸다.
- 그런데, 한쪽 방향이 너무 크다는 것은? =>엣지를 의미

  두 방향 모두 크다는 것은? => 코너를 의미

  모두 값이 미미하면? => flat을 의미
- 즉, H 행렬의 고유 값의 비율을 비교하는 것이 중요!

어떻게 적용할까?

- 행렬식 $Det(H) = \alpha\beta$, 대각합 $Tr(H) = \alpha + \beta$, (α, β는 고유값)
- 우리가 고유 값을 모든 H마다 구할 수가 없으므로 아래의 식을 이용한다.

$$\frac{Tr(H)^2}{Det(H)} = \frac{\alpha + \beta^2}{\alpha\beta} = \frac{(\gamma + 1)^2}{\gamma}$$

- 여기서 γ는 $\frac{\alpha}{\beta}$를 의미한다. (α는 항상 β보다 크다. 즉, γ는 1보다 크다.)
- 이 때 $\frac{(\gamma+1)^2}{\gamma}$는 γ이 1보다 클 때 증가 함수이며 일대일 함수이다.
- 이를 이용하면 우리가 고유 값을 구하지 않아도 그 비율을 이용해 엣지 위에 있다고 판단되는 keypoint를 제거할 수 있다.
- 일반적으로 우리가 챙겨야 할 keypoint의 γ는 10 미만으로 제시된다.

Edge 제거 전

Edge 제거 후

- 왼쪽은 앞의 방법으로 edge에 있는 keypoint들을 제거하기 전후 이미지이다.
- 생각보다 엣지라고 할 수 있는 부분에 keypoint들이 많이 남아 있는 것을 알 수 있다.
- 이는 왜 엣지에 있는 keypoint를 제거하려 했었는지 생각해보면 어느정도 이유를 알 수 있다.
- 엣지 자체가 문제가 아니라, 노이즈가 엣지처럼 보이는 경우를 우려한 것이기에 보수적으로 keypoint를 제거한 것이라고 생각할 수 있다.

5단계 : keypoint에 방향 할당해주기

- 이전 단계까지 적당한 keypoint를 찾았고 이 keypoint는 scale invariance(스케일 불변성)을 가진다.
- 이제 방향을 할당해주어 rotation invariance(회전 불변성)을 할당해주자.
- 방법 : 각 keypoint 주변의 그레디언트 방향과 크기를 모으자!
  - 1단계 : keypoint 주변에 특정크기의 윈도우 생성 후 keypoint가 blur된 만큼 해당 윈도우도 blur해준다. (가우시안 필터로)
  - 2단계 : 그 다음 모든 픽셀의 그레디언트 방향과 크기를 계산한다.

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

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

- 키 포인트 내의 모든 픽셀의 방향과 크기를 계산할 때 위의 식을 사용한다.
- 크기를 구하는 식, 방향을 구하는 식 모두 어디서 본 듯 하지만 약간 다르다는 것을 금방 알 수 있다.
- 크기를 구하는 식은 좌표평면에서 두 점 사이의 거리를 구하는 식, 방향을 구하는 식은 두 점 사이의 기울기를 구하는 식과 삼각함수를 이용해 x축과의 각도를 구하는 식이다.
- 크기를 구하는 식의 경우 x와 y 모두 자기자신 앞 뒤 값을 이용함으로써 주변 픽셀의 변화량을 -, + 방향에 관계 없이 적극적으로 담아냈다.

- 가우시안 함수를 적용했기 때문에 크기는 keypoint에 가까울수록 높은 크기를 갖게 된다.
- 아무튼 이제 방향을 할당해주는데, 먼저 히스토그램을 작성한다.
- 360도의 방향을 10도로 나누어 36개의 값을 가진다.
- 그 후 해당 keypoint의 윈도우 내의 모든 픽셀에서 그레디언트 방향의 값을 해당 픽셀에서의 크기만큼 해당하는 부분에 할당한다. 그러면 그레디언트 방향에 대한 히스토그램이 완성된다.
- 가장 높은 값을 갖는 방향이 해당 keypoint의 방향으로 할당된다.
- 만약 가장 높은 값을 갖는 방향의 빈도수의 80%를 충족하는 방향들이 있다면 이들도 새로운 keypoint로써 분리되며 같은 크기 다른 방향을 갖는 keypoint가 된다.

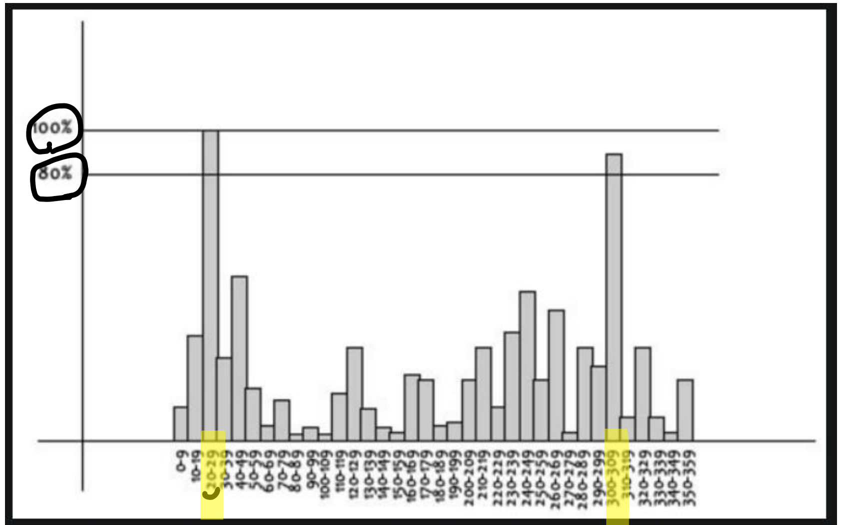
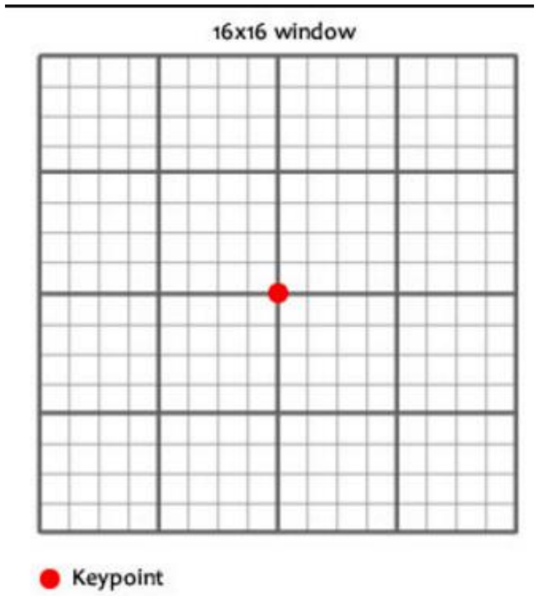


그림 15. keypoint 주변 그레디언트 방향 히스토그램 [1]

6단계 : 최종적인 SIFT 특징 산출

- 우리는 지금까지 keypoint들의 위치와 스케일과 방향을 알고 있으며 keypoint들은 위치와 스케일과 방향을 알고있다.
- 이제 이 keypoint를 식별하기 위해 지문과 같은 특별한 정보를 부여해보자.
- keypoint 주변의 16x16 pixel 윈도우를 만들고 이를 다시 4x4로 나눈다.
- 즉, 윈도우의 크기 : 16x16
- 윈도우를 나눈 구역의 수 : 16
- 윈도우를 또 나눈 윈도우의 크기 : 4x4



- 이전 단계에서 했던 것처럼 각 윈도우(4x4) 내의 모든 픽셀에서 크기와 방향을 모두 구하고, 히스토그램을 완성시킨다.
- 다만 다른 점은 360도를 8개로만 나누어 x축으로 사용한다.
- 우리는 4x4 윈도우가 16개 존재하므로 16개의 히스토그램이 존재한다.
- 각 히스토그램은 8개의 값을 가진다.
- 즉, 우리는 각 keypoint마다 16x8의 값을 가지게 되고 이 128개의 숫자로 이루어진 벡터가 keypoint를 구별하게 해주는 지문이 된다.

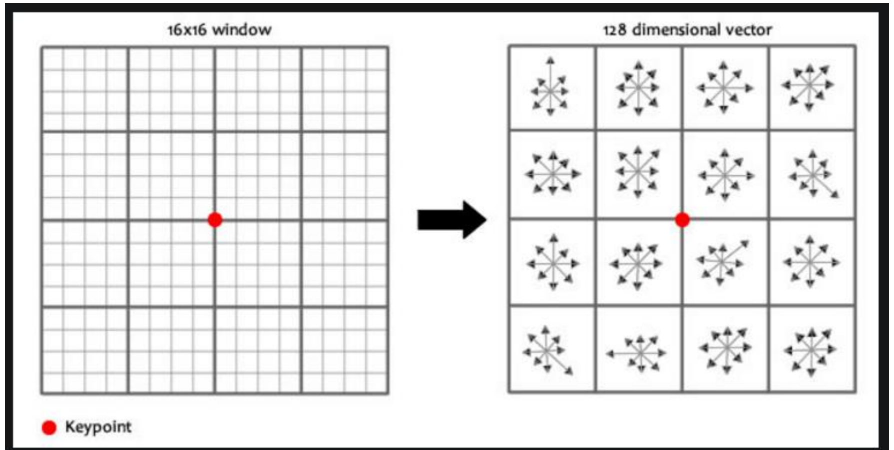


그림 16. 하나의 keypoint의 특징을 나타내는 128개의 숫자를 얻는 과정

- 아직 회전 의존성과 밝기 의존성을 해결해야 한다.
- 128개의 feature vector는 이미지가 회전하면 모든 그레디언트 방향은 변해버리게 된다. 이 문제를 해결하기 위해 keypoint의 방향을 각각의 그레디언트 방향에서 빼준다.
- 그러면 각각의 그레디언트의 방향은 keypoint의 방향에 대해 상대적이게 된다. => 밝기 의존성 해결!
- 밝기 의존성의 경우 정규화를 통해 해결한다.
- 정규화는 범위나 단위가 서로 다른 값들을 평균과 표준편차를 맞춰주는 등의 방법을 통해 그것을 일치시켜주는 과정을 말한다.
- 이렇게 SIFT 특징을 추출하는 과정이 모두 종료되었다. 두 이미지에서 각각의 keypoint를 찾는다면 두 keypoint의 feature vector를 확인해 서로 매칭시킬 수 있다.

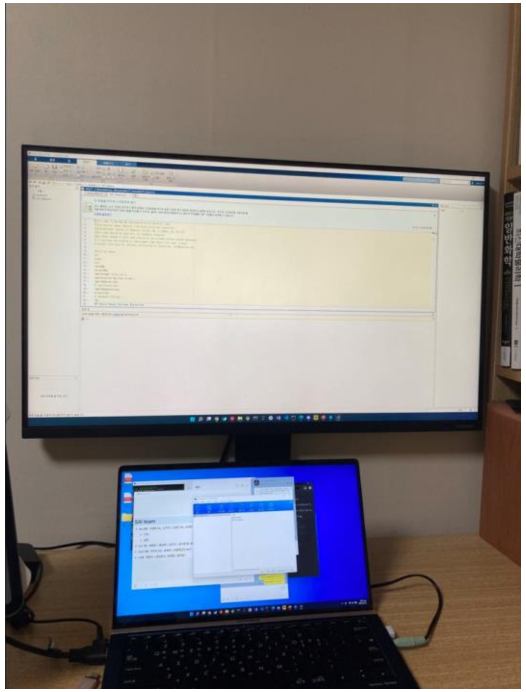
다른 Octave, Blur에서 검출한 keypoint는?

- 다른 Octave, Blur에서 검출된, 즉 다른 Scale에서 검출된 keypoint 역시 앞의 모든 과정을 거쳐서 각 Scale에서 얻은 모든 keypoint 사용해야 한다.
- 서로 다른 Scale에서의 Keypoint를 얻기 위해 Scale Space를 만들었기 때문!
- 만약 중복되는 경우만 사용한다면 다른 Scale에서 얻은 다양한 통찰을 가진 Keypoint를 활용할 수 없음
- Octave를 몇 개로 할 지, 한 Octave 내의 서로 다른 blur된 이미지 수는 몇 개로 할 지는 사용자의 결정

직접 SIFT를 사용해보자

- mathworks.com에 게시된 SIFT 알고리즘을 사용하였다.
- 지금부터 서술될 SIFT의 순서는 해당 m파일을 기준으로 작성된다.

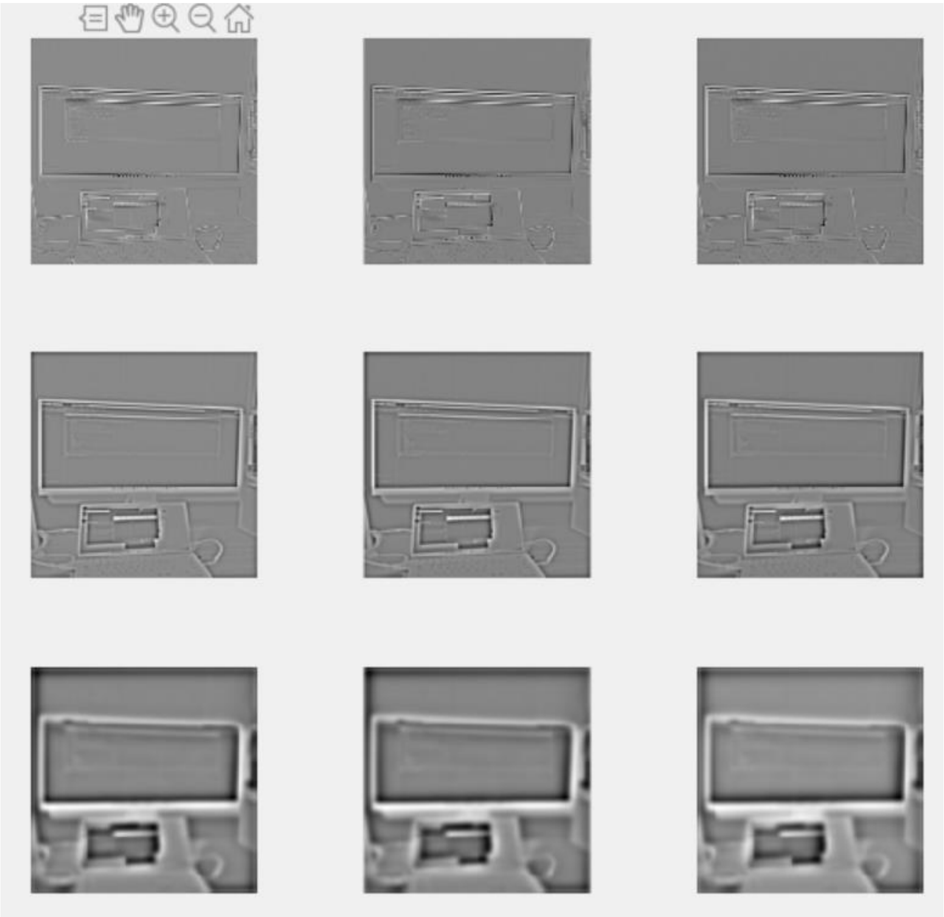
입력 이미지->



전처리

- 이미지를 받은 후 256x256으로 resiz한다.
  =>이미지 크기에 따른 연산량 증가를 방지하는 것으로 보임.
- 이미지를 흑백 이미지로 바꿈
  =>밝기 신호 외에는 크게 필요가 없음
- 이미지를 double로 읽어들임
  =>뒤의 연산 과정에서 실수 데이터가 필요
- octave를 3개로 설정
- 가우시안 필터의 초기값은 $\sqrt{2}$

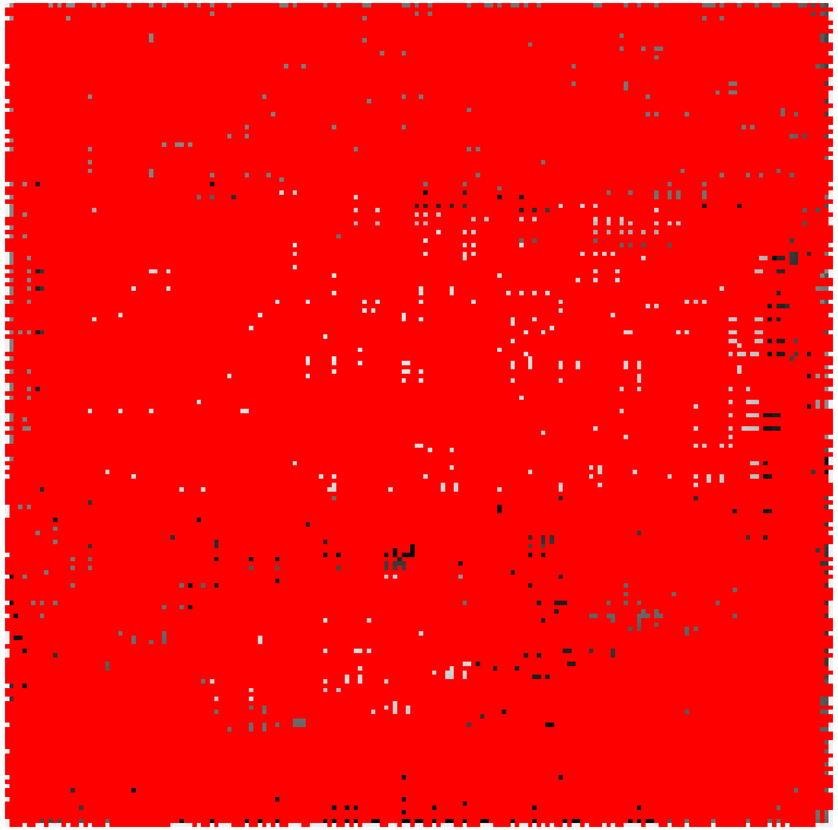
DoG 이미지 출력



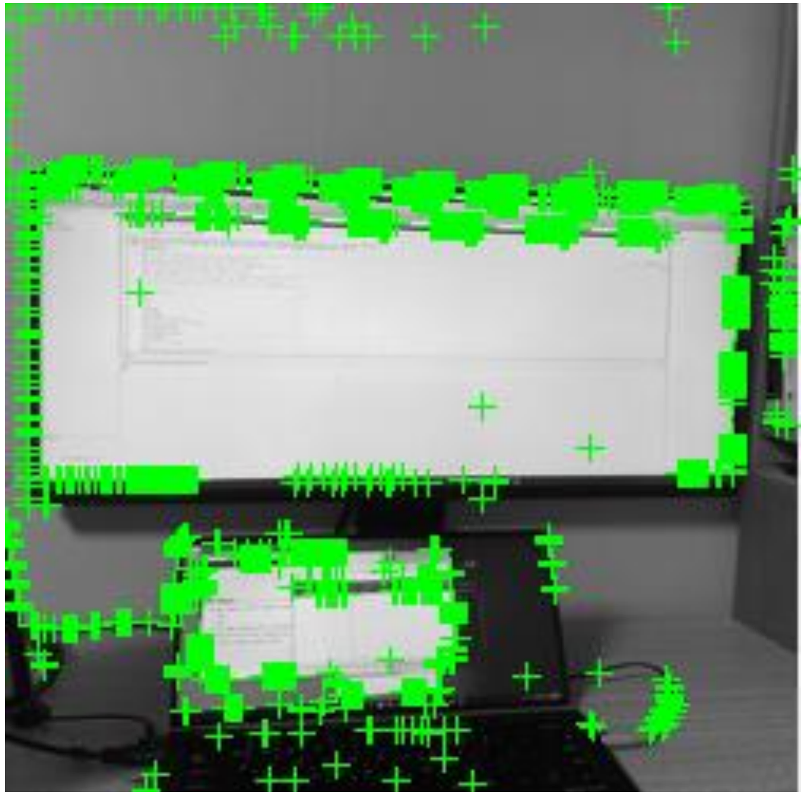
- 각 옥타브 별로 DoG 이미지를 출력한 결과이다. edge가 잘 검출되었음을 알 수 있다.
- 가로줄에 있는 것이 같은 octave이다.
- 3개의 Octave를 사용했으며 그 안에서 4 개의 다른 blur 이미지를 가지고 있었음을 알 수 있다.
- 엣지가 더 진하게 표시되는 것은 서로 다른 사이즈를 같은 사이즈 출력으로 맞추려다 보니 edge가 작은 사이즈일 때 더 두껍게 출력된 것으로 추측된다.
- 각 octave 내에서 DoG 이미지는 매우 비슷한 것은 스케일 파라미터를 일정한 k배씩 증가시키며 사용했기 때문으로 추측된다.

나쁜 keypoint를 제거하기 전의 keypoint

- 사방이 keypoint이다...
- 이 keypoint들은 모든 Octave에서 얻은 keypoint를 나타낸 것이다.
- keypoint들이 무분별하게 검출되어 왜 나쁜 keypoint를 없애야 하는지 납득이 가능하다.

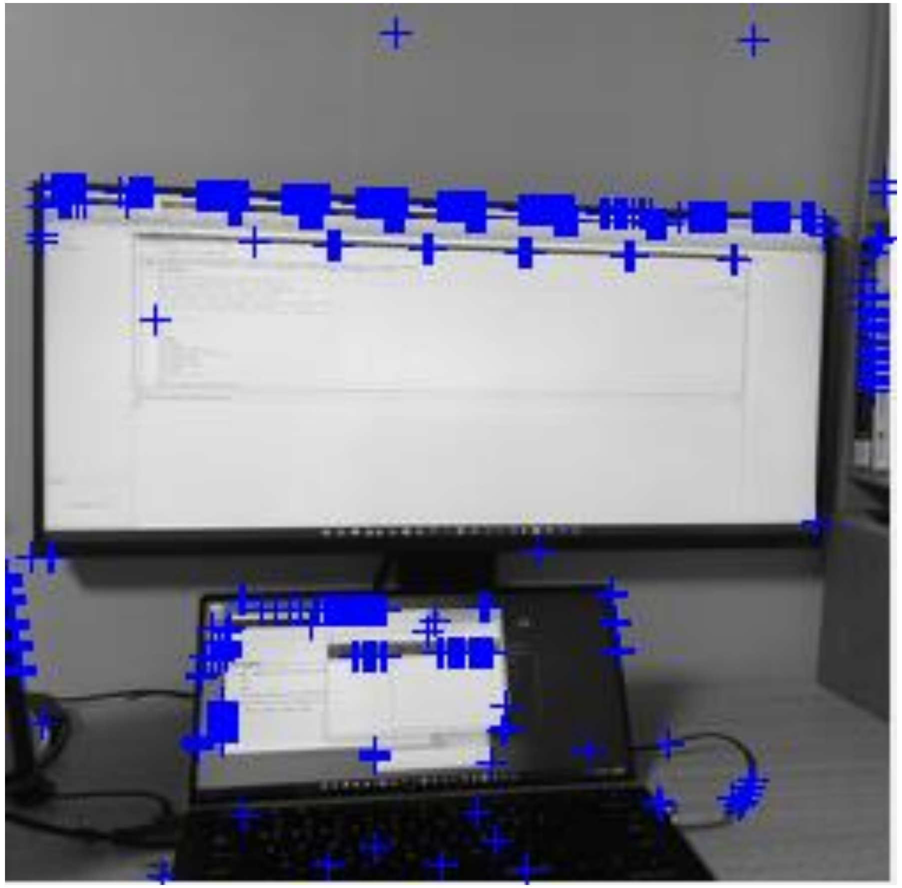


낮은 contrast를 가지는 keypoint 제거



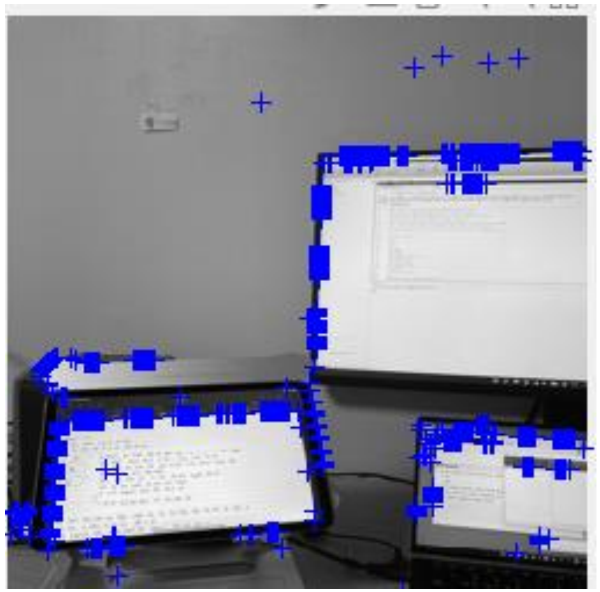
- constrast가 0.1보다 작은 점들을 모두 제거한 모습이다.
- 앞에서 이론에서 제시된 기준은 0.03이었지만 해당 프로그램은 0.1보다 작은 점들을 모두 제거하였다.
- 값이 높아질수록 keypoint의 수는 작아지는 대신, 더 많은 노이즈를 제거할 수 있을 것이다.

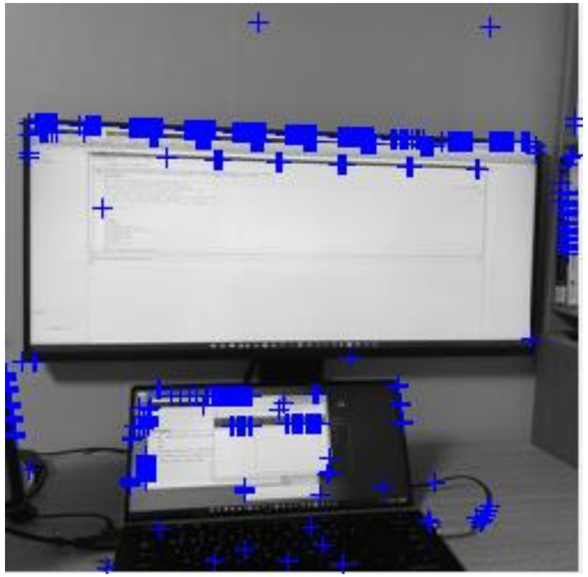
edge에 있는 keypoint 제거



- 앞 슬라이드의 경우 어떤 물체 위의 엣지가 아닌 좌측 상단의, 노이즈라고 할 수 있는 점이 상당히 많이 있었다.
- edge에 있는 keypoint를 제거함으로써 이 러한 노이즈도 많이 제거되었음을 확인할 수 있다.
- 다만 모니터의 엣지에 해당하는 부분이 많이 없어진 모습은 아쉽다.
- 원본 이미지에서도 검정-하양 부분이라 한 방향에서의 기울기가 상당히 급격했기에 많은 keypoint들이 없어진 것으로 보인다.
- 다만, 좌측 edge가 많이 없어진건 이해가 되는데 상단 edge의 keypoint들이 많이 살아 있는 것은 의문이다.

잘 매칭될까?





- 겹치는 부분을 눈대중으로 살펴볼 때 꽤 많은 부분이 겹친다는 느낌이 든다.
- 하지만 일부 포인트의 경우 한 이미지에서만 나타난다.
- 특히 모니터 좌측 모서리는 왼쪽 이미지는 keypoint 검출이 많이 된 것에 비해 오른쪽 이미지는 거의 검출되지 않았다.
- 남아있는 같은 keypoint를 어떻게 매칭시켜야 하는지와 이를 어떻게 연결시키는지가 중요할 것으로 보인다.

Harris 코너 검출기

- 다른 특징 추출 방법 중 하나인 Harris 코너 검출기에 대해 알아보자.
- Harris 코너 검출기는 특징 추출에 사용하고, descriptor는 SIFT의 방법을 사용해야 한다.
- 다시 한 번 엣지와 코너에 대해 짚고 넘어가면,
  - 엣지는 한 방향에서 변화가 급격한 점
  - 코너는 두 방향이상에서 변화가 급격한 점
  - 평탄한 점은 이도 저도 아닌 점들을 뜻한다.
- 코너가 중요한 이유는 이미지에서 가장 중요한 정보를 담기 때문이며 이미지에서 코너점들만 남겨놔도 이미지 내의 물체들의 형상을 대충 알 수 있다.

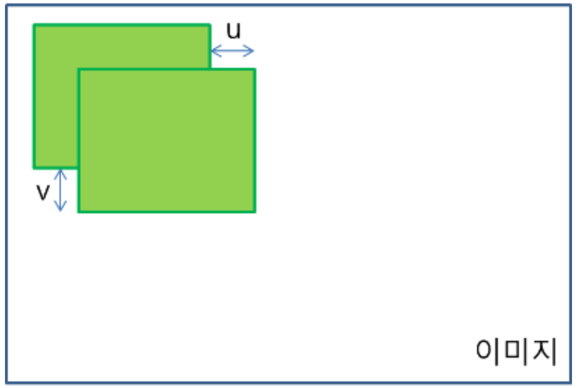
Harris 코너 검출기의 기본 원리

- 한 픽셀을 중심에 놓고 작은 윈도우를 설정한 다음, x축 방향으로 u만큼, y축 방향으로 v만큼 이동시킨다.
- 그 다음 윈도우 내의 픽셀 값들의 차이의 제곱의 합을 구해준다.

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

- 윈도우를 이동시키기 전과 후가 얼마나 변화했는지 계산해주는 것이다.
- 코너점이라면 x축, y축 방향 모두 많이 변화했을 것

=> 일단 E값이 크면 코너점이라고 본다.



테일러 확장

- 그런데 앞의 식은 테일러 확장에 의해 다음과 같이 근사된다.

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

$$\approx \sum_{(x_k,y_k)\in W} (\frac{\partial I}{\partial x}u)^2 + (\frac{\partial I}{\partial y}v)^2 + 2\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}uv$$

이를 행렬로 나타내면

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

- 두 슬라이드 전, E값이 크면 코너점이라고 했는데, 그러려면 중간에 위치한 2x2 행렬의 값이 커야 한다. 이 행렬을 M이라고 하자. (이 행렬을 structure tensor라고 부르기도 한다.)
- 이 값을 고유 값 분해해서 SIFT에서 얻은 논리를 그대로 사용한다.
- 두 방향의 변화가 크면 고유 값들은 모두 충분히 큰 값을 가지고 하나는 크고 하나는 작으면 엣지, 둘 다 작으면 flat이다.
- 그런데 모든 윈도우마다 고유값을 계산하면 너무 복잡하므로 이번에도 공식을 대체한다. 이번에는 약간 다른 공식을 사용한다.

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

둘 다 고유값이 모두 크면 R은 0보다 크게 되고 둘 다 작으면 0에 가깝게, 한쪽 값만 크면 –값으로 출력된다. (k는 파라미터)

고유 값을 이용한 식이 다 다르다?

- SIFT에서 사용한 $\frac{Tr(H)^2}{Det(H)} = \frac{\alpha+\beta^2}{\alpha\beta} = \frac{(\gamma+1)^2}{\gamma}$
- Harris 코너에서 사용한 $R = \det(M) - k(trace(M))^2$
- 그런데 교수님 자료에서 제시한 Harris Operator는 다음과 같다.

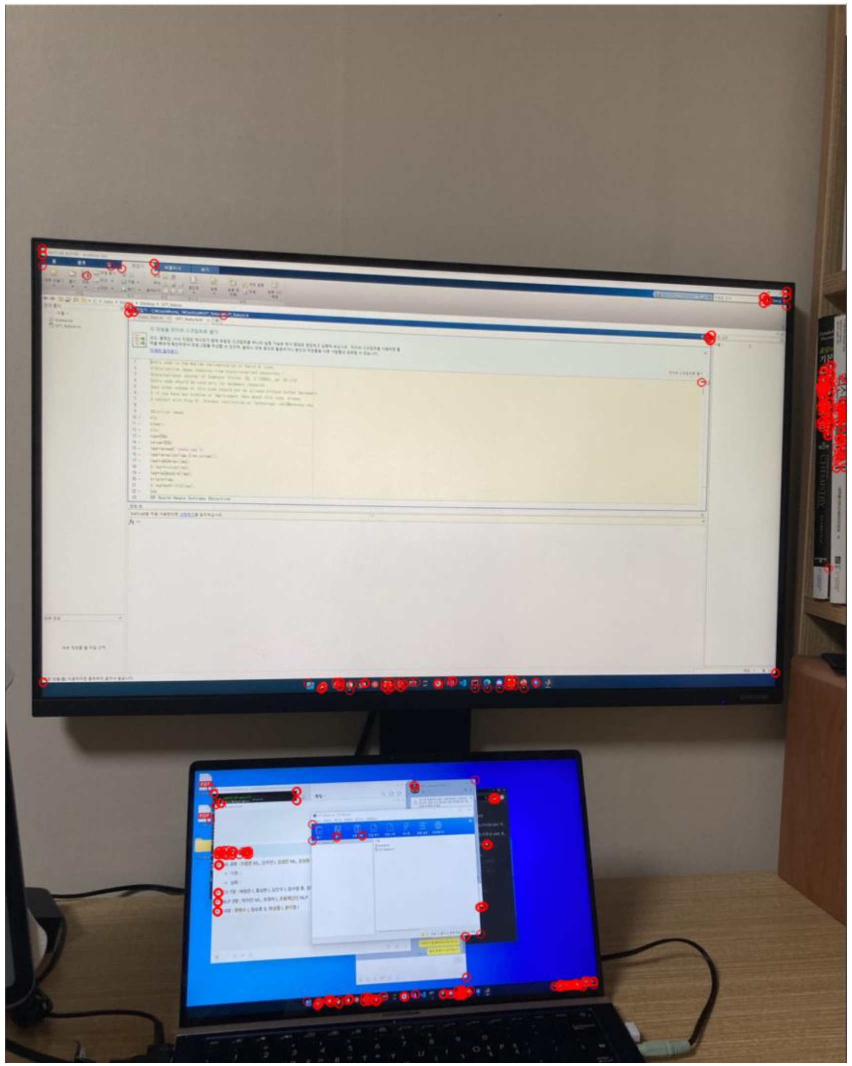
$$\frac{\det(A^T A)}{trace(A^T A)}$$

개인적인 생각으로는, 고유 값을 왜 사용하는지와 구하기 번거로우니 그 비율을 간단한 식을 통해 구해서 사용하자는 같은 아이디어로 보이며 구체적인 식만 다른 것으로 보인다. 즉, 논리는 같다.

가우시안 필터를 쓸까 말까?

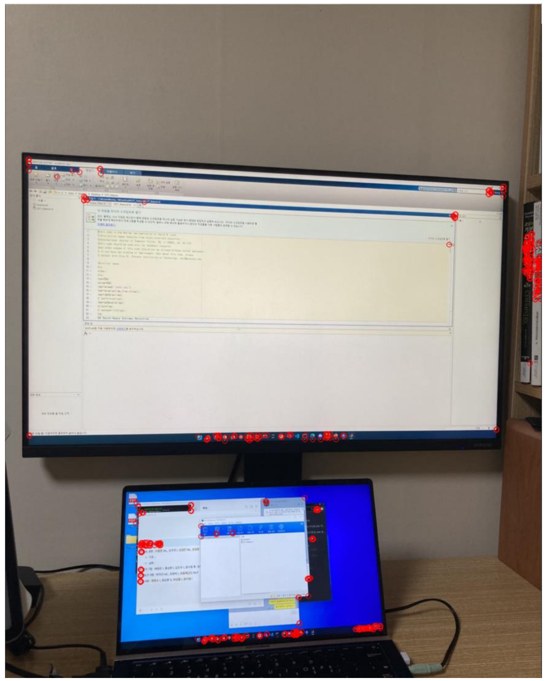
- 이 부분 역시 직접 찾아본 글에서는 가우시안 필터를 사용하지 않았지만, 교수님 자료에서는 가우시안 필터를 사용하는 경우가 존재했다.
- SIFT는 가우시안 필터가 Size는 같지만 Scale을 다르게 하기위해 사용하는 것이 주 목적이었다. 즉, 무조건 사용!
- 해리스 코너 검출에서 가우시안 필터를 사용한다면 노이즈 제거를 위함이다.
- 해리스 코너 검출은 Scale 조정 필요x, 가우시안 필터는 blur에도 사용되고, LPF의 역할도 하므로 노이즈 제거에 유용

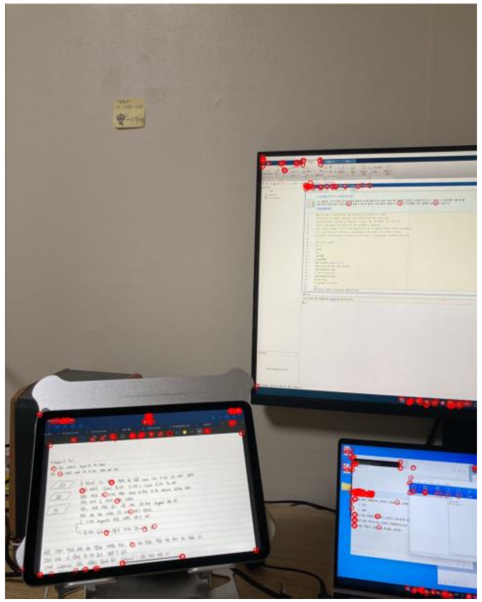
Python openCV를 이용한 해리스 코너 검출



- 빨갛게 표시된 부분이 코너(Feature)로 검출된 부분이다.
- 목적에 맞게 코너 위주로 검출이 되었으며 엣지 부분에는 거의 검출되지 않았다.
- 의도치 않게 매우 디테일한 영역에도 검출되었는데, pc 화면 내의 아이콘에서도 몇 개의 코너가 검출되었으며 책장의 책의 글자에도 코너가 검출되었다.
- 책의 경우에는 검은 바탕에 흰 글씨, 흰 바탕에 검은 글씨로 이루어져 그래디언트가 큰 값을 가지며 코너로 검출된 것으로 보인다.
- 노트북, 모니터의 큰 부분에서 값이 출력되지 않은 것은 의문이다.
- 주로 짙은색 – 하얀색으로 경계가 이루어진 코너가 잘 검출되었고 검은색 – 파란색으로 이루어진 경계 등은 잘 검출되지 않았다.

잘 매칭 될까?





- 이번에도 짙은색 – 하얀색으로 경계가 이루어진 코너가 잘 검출되었고 검은색 – 파란색으로 이루어진 경계 등은 잘 검출되는 등 앞의 이미지와 여러모로 같은 경향
- 겹치는 부분을 보면, 노트북 화면 내부에서 추출된 점들이 꽤 잘 매칭될 것으로 보인다.
- SIFT의 경우 매칭이 아예 안될 것으로 보이는 feature들이 많았는데, 해리스 코너의 경우 feature는 별로 없지만, 남아도는 feature의 수도 거의 없다.

정리

- SIFT, Harris 코너는 이미지를 잘 표현할 수 있는 특징을 추출하기 위한 알고리즘이며 SIFT는 이 특징들에 지문까지 부여해준다.
- SIFT 알고리즘은 그래디언트의 크기, 방향 정보를 사용해서 descriptor(설명자, 앞 문장의 지문)을 생성한다.
- Harris 코너 검출의 경우 descriptor를 부여하기 위해서는 SIFT 등의 다른 알고리즘의 힘을 빌려야 한다.
- 한 이미지의 헤시안 행렬에서 고유 값은 고유 벡터의 크기를 나타내며 주변 픽셀과 현재 픽셀의 변화도를 나타낸다.