14장 지역 특징점 검출과 매칭

14.1 코너 검출

14.1.1 해리스 코너 검출 방법

- 템플릿 매칭
  - 입력 영상에서 특정 객체 위치를 찾을 때 유용하게 사용
  - 영상의 크기가 바뀌거나 회전이 되면 제대로 동작하지 않는다는 한계

  → 두 영상 사이에 기하학적 변환이 있어도 효과적으로 사용할 수 있는 지역 특징점 기반 매칭 방법
- **특징**
  - 영상으로부터 추출할 수 있는 유용한 정보

    ex) 평균 밝기, 히스토그램, 에지, 직선 성분, 코너
    - **지역 특징**
      - 에지, 직선 성분, 코너처럼 영상 전체가 아닌 일부 영역에서 추출할 수 있는 특징
      - 코너 : 에지의 방향이 급격하게 변하는 부분
        - 에지나 직선 성분 등의 다른 지역 특징에 비해 분별력이 높고 대체로 영상 전 영역에 골고루 분포하기 때문에 영상을 분석하는 데 유용한 지역 특징으로 사용
- **특징점** (키포인트, 관심점) : 코너처럼 한 점의 형태로 표현할 수 있는 특징
- **해리스 코너 검출 방법**
  - 코너 점 구분을 위한 기본적인 아이디어를 수학적으로 잘 정의하였다는 점에서 큰 의미
  - 영상의 특정 위치 (x, y)에서 Δx와 Δy만큼 떨어진 픽셀과의 밝기 차이를 다음 수식으로 표현

$$E(\Delta x, \Delta y) = \sum_{x,y} w(x,y)\left[I(x+\Delta x, y+\Delta y) - I(x,y)\right]^2$$

    - w(x, y) : 균일한 값을 갖는 사각형 윈도우 또는 가우시안 형태의 가중치를 갖는 윈도우
    - E(Δx, Δy) 함수가 모든 방향으로 값이 크게 나타난다면 점 (x, y)는 코너라고 간주

      → 모든 방향으로 값이 크게 나타나는지 검사 위해

      테일러 급수(Taylor series), 고윳값 분석(eigenvalue analysis) 등의 수학적 기법을 적용하여 **코너 응답 함수 R**을 유도

$$R = \mathrm{Det}(\mathbf{M}) - k \cdot \mathrm{Tr}(\mathbf{M})^2$$

$$\mathbf{M} = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

Det()는 행렬식, Tr()은 대각합

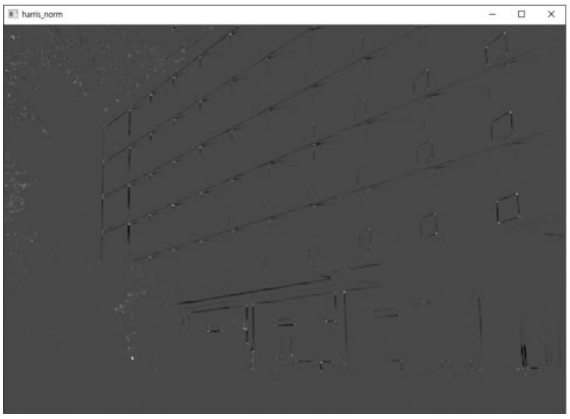
Ix와 Iy는 입력 영상 I를 각각 x축 방향과 y축 방향으로 편미분한 결과

상수 k는 보통 0.04~0.06 사이의 값을 사용

- R이 0보다 충분히 큰 양수이면 코너 점이라고 간주
- R이 0에 가까운 실수이면 평탄한 영역
- 0보다 작은 음수이면 에지라고 판별

```
void corner_harris()
{
  // building.jpg 영상을 그레이스케일 형식으로 불러와 src에 저장
  Mat src = imread("building.jpg", IMREAD_GRAYSCALE);

  if (src.empty()) {
    cerr << "Image load failed!" << endl;
    return;
  }

  // src 영상으로부터 해리스 코너 응답 함수 행렬 harris를 구함
  Mat harris;
  cornerHarris(src, harris, 3, 3, 0.04);

  // harris 행렬 원소 값 범위를 0부터 255로 정규화
  // 타입을 CV_8UC1로 변환하여 harris_norm에 저장
  // 해리스 코너 응답 함수 분포를 영상 형태로 화면에 표시
  Mat harris_norm;
  normalize(harris, harris_norm, 0, 255, NORM_MINMAX, CV_8U);

  // src 영상을 3채널 컬러 영상으로 변환하여 dst에 저장
  Mat dst;
  cvtColor(src, dst, COLOR_GRAY2BGR);

  for (int j = 1; j < harris.rows - 1; j++) {
    for (int i = 1; i < harris.cols - 1; i++) {
      // harris_norm 영상에서 값이 120보다 큰 픽셀을 코너로 간주
      if (harris_norm.at<uchar>(j, i) > 120) {
        // 간단한 비최대 억제 수행
        // (i, j) 위치에서 주변 네 개의 픽셀을 비교하여 지역 최대인 경우에만 dst 영상에 빨간색 원으로 코너를 표시
        if (harris.at<float>(j, i) > harris.at<float>(j - 1, i) &&
          harris.at<float>(j, i) > harris.at<float>(j + 1, i) &&
          harris.at<float>(j, i) > harris.at<float>(j, i - 1) &&
          harris.at<float>(j, i) > harris.at<float>(j, i + 1)) {
```

```
            circle(dst, Point(i, j), 5, Scalar(0, 0, 255), 2);
          }
        }
      }
    }

    imshow("src", src);
    imshow("harris_norm", harris_norm);
    imshow("dst", dst);

    waitKey(0);
    destroyAllWindows();
}
```



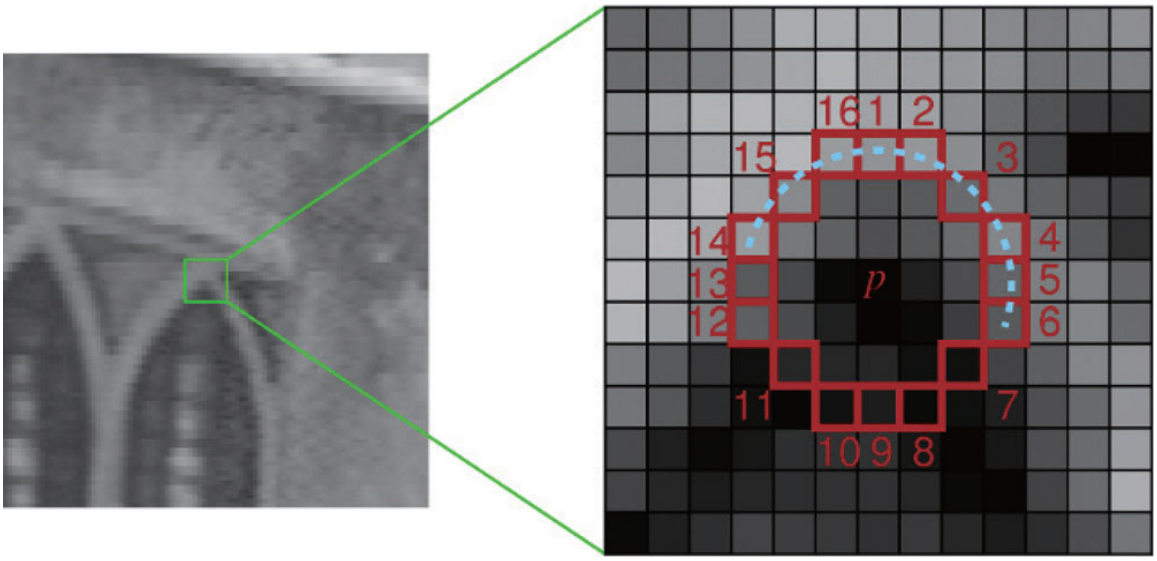
- src는 building.jpg 파일을 그레이스케일 형식으로 불러온 영상
- harris_norm은 해리스 코너 응답 함수 값을 0부터 255 사이로 정규화하여 나타낸 그레이스케일 영상
  - 밝은 회색 점처럼 표현된 부분이 코너 위치
- harris_norm 영상에서 픽셀 값이 120보다 크고, 지역 최대인 지점을 선별하여 빨간색 원으로 표시한 결과를 dst 창에 나타냄

21행에서 사용한 임계값 120을 낮추면 더 많은 건물 모서리를 코너로 검출할 수 있지만, 나뭇잎 또는 풀밭에서 코너로 검출되는 부분도 함께 늘어날 수 있으니 주의

14.1.2 FAST 코너 검출 방법

- 해리스 코너 검출 방법은 복잡한 연산을 필요로 하기 때문에 연산 속도가 느리다는 단점
- FAST 코너 검출 방법은 단순한 픽셀 값 비교 방법을 통해 코너를 검출

- FAST 코너 검출 방법
  - 영상의 모든 픽셀에서 픽셀을 둘러싸고 있는 16개의 주변 픽셀과 밝기를 비교하여 코너 여부를 판별



  - 영상의 모든 픽셀에서 픽셀을 둘러싸고 있는 16개의 주변 픽셀과 밝기를 비교하여 코너 여부를 판별
  - 주변 16개의 픽셀 중에서 점 p보다 충분히 밝거나 또는 충분히 어두운 픽셀이 아홉 개 이상 연속으로 존재하면 코너로 정의

```
void corner_fast()
{
  // building.jpg 영상을 그레이스케일 형식으로 불러와 src에 저장
  Mat src = imread("building.jpg", IMREAD_GRAYSCALE);

  if (src.empty()) {
    cerr << "Image load failed!" << endl;
    return;
  }

  // src 영상에서 FAST 방법으로 코너 점을 검출
  // 밝기 차이 임계값으로 60을 지정, 비최대 억제를 수행
  // 검출된 모든 코너 점 좌표는 keypoints 변수에 저장
  vector<KeyPoint> keypoints;
  FAST(src, keypoints, 60, true);

  // src 영상을 3채널 컬러 영상으로 변환하여 dst에 저장
  Mat dst;
```

```
  cvtColor(src, dst, COLOR_GRAY2BGR);

  // 검출된 모든 코너 점에 반지름이 5인 빨간색 원을 그림
  for (KeyPoint kp : keypoints) {
    Point pt(cvRound(kp.pt.x), cvRound(kp.pt.y));
    circle(dst, pt, 5, Scalar(0, 0, 255), 2);
  }

  imshow("src", src);
  imshow("dst", dst);

  waitKey(0);
  destroyAllWindows();
}
```

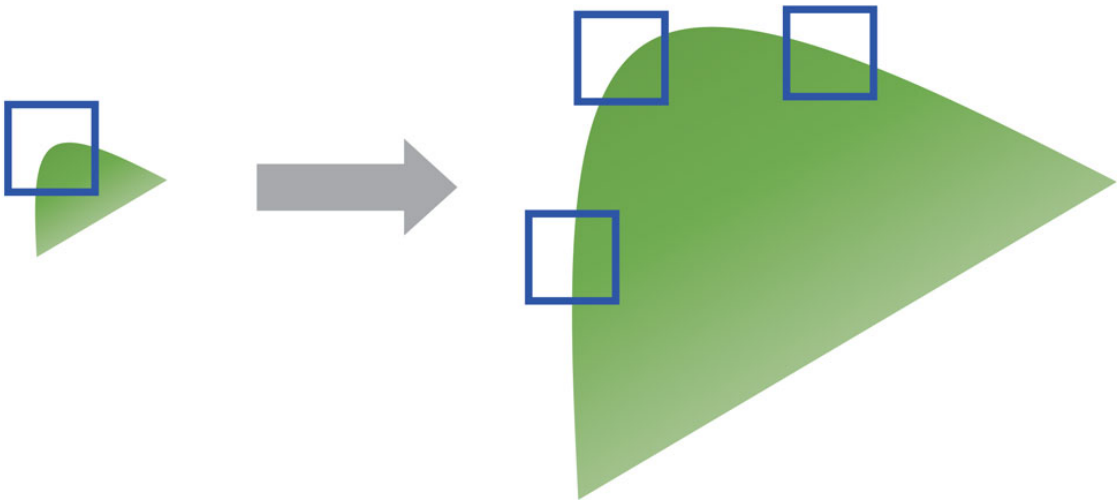
- 대부분의 건물 모서리와 나뭇잎 부분에서 다수의 코너가 검출된 것을 확인 가능

대략적으로 FAST() 코너 검출 방법이 cornerHarris() 방법보다 20배 이상 빠르게 동작함

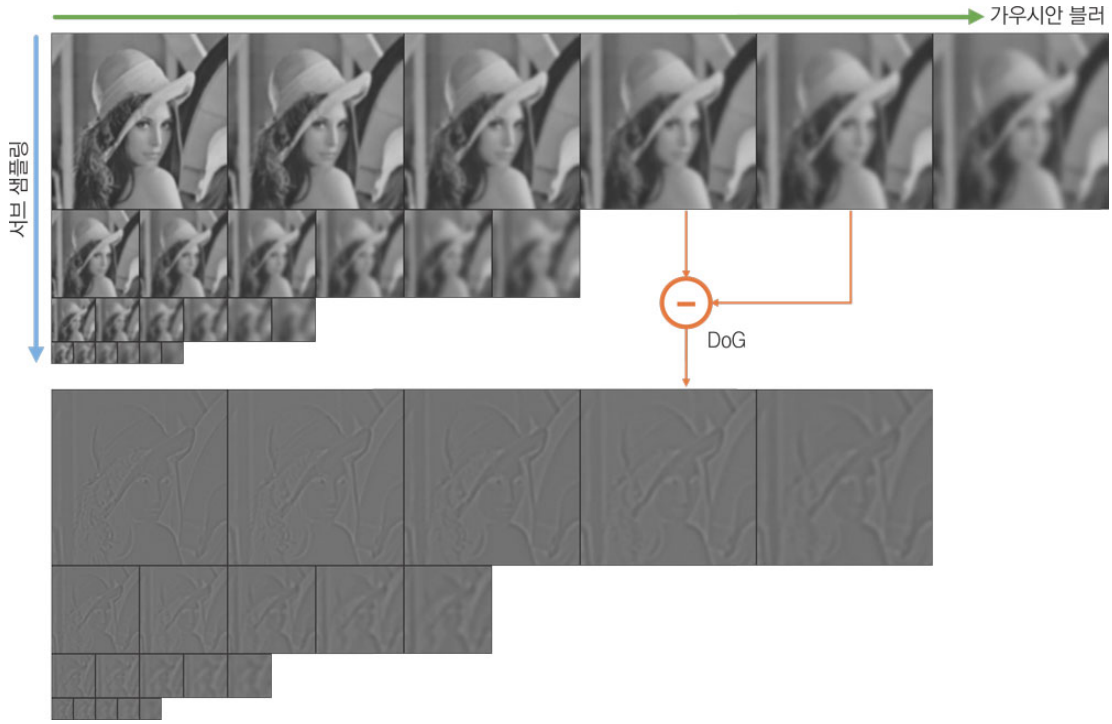
14.2 크기 불변 특징점 검출과 기술

14.2.1 크기 불변 특징점 알고리즘

- 영상의 크기가 변경될 경우 코너는 더 이상 코너로 검출되지 않을 수 있음



- **SIFT 알고리즘**
  - 영상의 크기 변화에 무관하게 특징점을 추출하기 위하여 입력 영상으로부터 **스케일 스페이스** (scale space)를 구성
    - **스케일 스페이스** : 영상에 다양한 표준 편차를 이용한 가우시안 블러링을 적용하여 구성한 영상 집합



    - 윗 줄에 나타난 여섯 개의 블러링된 영상이 스케일 스페이스를 구성한 결과 : **옥타브**
      - 이후, 입력 영상의 크기를 가로, 세로 반으로 줄여 가면서 여러 옥타브를 구성
    - 아래쪽에 나열한 영상 : 알고리즘에서 크기에 불변한 특징점을 검출할 때에는 인접한 가우시안 블러링 영상끼리의 차 영상을 사용 : **DoG**(Difference of Gaussian) 영상

- SIFT 알고리즘은 인접한 DoG 영상을 고려한 지역 극값 위치를 특징점으로 사용
  - 에지 성분이 강하거나 명암비가 낮은 지점은 특징점에서 제외
- 영상의 크기, 회전 등의 변환뿐만 아니라 촬영 시점 변화에도 충분히 강인하게 동작하며, 잡음의 영향과 조명 변화가 있어도 특징점을 반복적으로 잘 찾아냄

- SIFT 알고리즘은 복잡한 연산을 수행해야 하기 때문에 실시간 응용에서 사용하기 어렵다는 단점

→ **ORB 알고리즘**은 기본적으로 FAST 코너 검출 방법을 이용하여 특징점을 추출

but 기본적인 FAST 알고리즘은 영상의 크기 변화에 취약

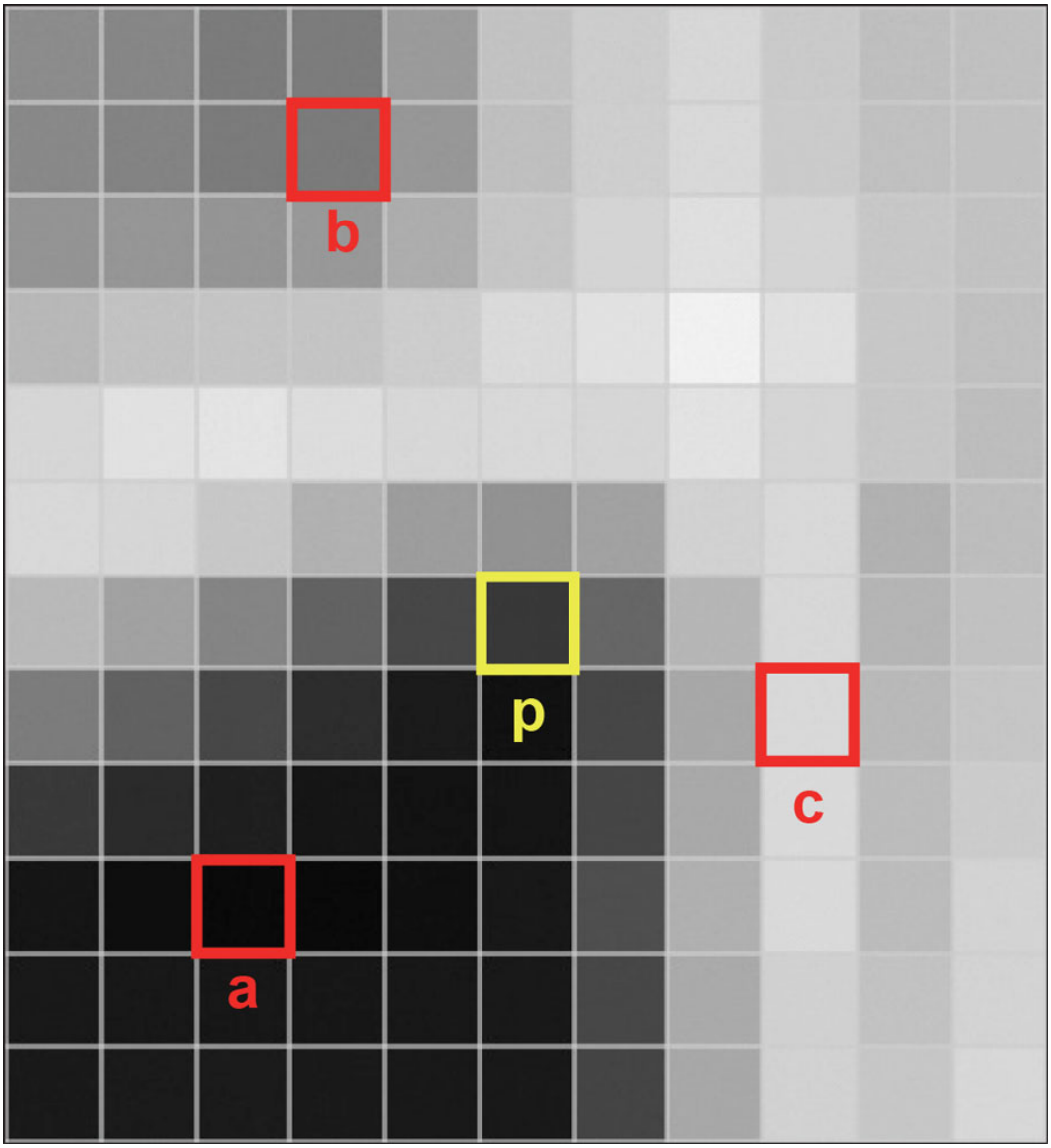
→ **ORB 알고리즘**은 입력 영상의 크기를 점진적으로 축소한 피라미드 영상을 구축하여 특징점을 추출

→ 각 특징점에서 주된 방향 성분을 계산, 방향을 고려한 **BRIEF 알고리즘**으로 이진 기술자를 계산

- **BRIEF 알고리즘**
  - 특징점 기술자만을 생성하는 알고리즘

$$\tau(\mathbf{x},\mathbf{y}) = \begin{cases} 1 & I(\mathbf{x}) < I(\mathbf{y}) \text{일 때} \\ 0 & \text{그 외} \end{cases}$$

→ 특징점 주변의 픽셀 쌍을 미리 정하고, 해당 픽셀 값 크기를 비교하여 0 또는 1로 특징을 기술



1. 특징점 p 주변에 a, b, c 점을 미리 정의
2. τ(a, b), τ(b, c), τ(c, a)를 구하면 이진수 110(2)을 얻을 수 있음

   → b 점이 a보다 밝고, c 점이 b보다 밝고, a 점은 c 점보다 어둡다는 정보를 표현

   이진 기술자 : 특징점 주변 정보를 이진수 형태로 표현하는 기술자

- **ORB 알고리즘**
  1. FAST 기반의 방법으로 특징점을 구함
  2. 각 특징점에서 픽셀 밝기 값 분포를 고려한 코너 방향 성분을 계산
  3. BRIEF 계산에 필요한 점들의 위치를 보정

     → 회전에 불변한 BRIEF 기술자를 계산

     (기본적으로 256개의 크기 비교 픽셀 쌍을 사용하여 이진 기술자를 구성, 하나의 특징점을 256비트로 표현)

참고 : 이진 기술자로 표현된 특징점 사이의 거리는 해밍 거리 방법 사용

즉, 둘 중 하나의 문자열에서 **몇개의 문자를 바꿔야 두 문자열이 같아지느냐** 입니다.

두 문자열 => Hamming distance

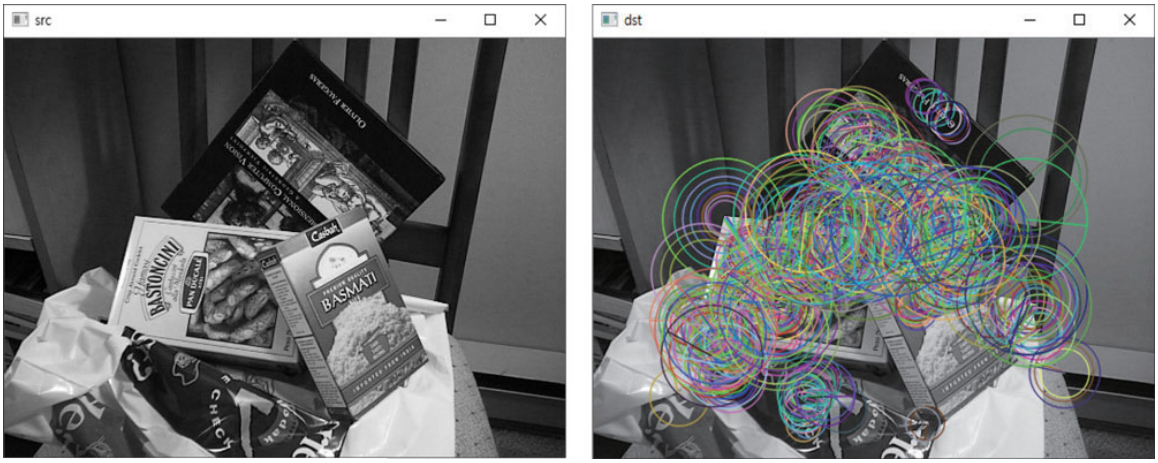
1011 and **01**11 => 2

ac**dd**f and **b**c**dxy** => 3

→ 이진수로 표현된 두 기술자에서 서로 값이 다른 비트의 개수를 세는 방식으로 계산

(두 기술자의 비트 단위 배타적 논리합(XOR) 연산 후, 비트 값이 1인 개수를 세는 방식으로 빠르게 계산 가능)

```
void detect_keypoints()
{
  Mat src = imread("box_in_scene.png", IMREAD_GRAYSCALE);

  if (src.empty()) {
    cerr << "Image load failed!" << endl;
    return;
  }

  // ORB 클래스 객체를 생성하여 feature 스마트 포인터에 저장
  // 스마트 포인터 : 사용이 끝난 메모리를 자동으로 delete() 해주는 포인터
  Ptr<Feature2D> feature = ORB::create();

  // ORB 키포인트를 검출하여 keypoints 벡터에 저장
  vector<KeyPoint> keypoints;
  feature->detect(src, keypoints);

  // ORB 키포인트 기술자를 계산하여 desc 행렬에 저장
  Mat desc;
  feature->compute(src, keypoints, desc);

  // keypoints에 저장된 키포인트 개수와 desc 행렬 크기를 콘솔 창에 출력
  cout << "keypoints.size(): " << keypoints.size() << endl;
  cout << "desc.size(): " << desc.size() << endl;

  // 입력 영상 src에 키포인트를 그린 결과를 dst에 저장
  // DrawMatchesFlags:: DRAW_RICH_KEYPOINTS로 지정하여 키포인트 위치, 크기, 방향 정보를 함께 나타내도록 설정
  Mat dst;
  drawKeypoints(src, keypoints, dst, Scalar::all(-1), DrawMatchesFlags::DRAW_RICH_KEYPOINTS);

  imshow("src", src);
  imshow("dst", dst);

  waitKey();
  destroyAllWindows();
}
```



→ 각 특징점 위치를 중심으로 다수의 원이 그려짐

원의 크기는 특징점 검출 시 고려한 이웃 영역의 크기

원의 중심에서 뻗어 나간 직선을 특징점 근방에서 추출된 주된 방향

> 💡 **keypoints.size(): 500**
> **desc.size(): [32 x 500]**

가 출력되는데, 이는 ORB 알고리즘으로 영상에서 500개의 특징점이 검출되었고,

특징점을 표현하는 기술자 행렬이 500행, 32열로 구성되었음을 나타내는 것