5.5 매칭

- 매칭
  - 어떤 대상을 다른 것과 비교하여 그들이 같은 것인지 알아내는 과정
    - 유사성 혹은 거리를 측정하여 비교
  - 컴퓨터 비전에서 여러 가지 문제를 해결하는 핵심 역할
    - 물체 인식, 물체 추적, 스테레오, 카메라 캘리브레이션 등

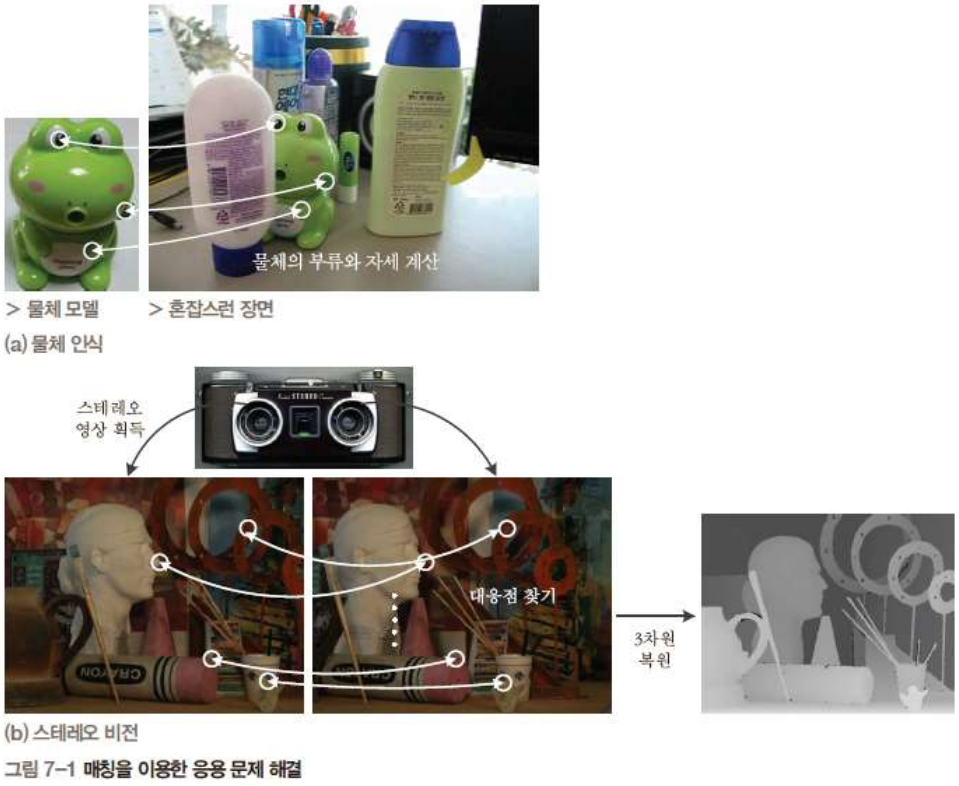


그림 7-1 매칭을 이용한 응용 문제 해결

5.5.1 매칭 전략

문제의 이해

- 두 영상에서 추출한 기술자 집합 $A = \{\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_m\}$와 $B = \{\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_n\}$에서 같은 물체의 같은 곳에서 추출된 $\mathbf{a}_i$와 $\mathbf{b}_j$ 쌍을 모두 찾는 문제
- 매칭을 적용하는 다양한 상황
  - 물체 인식에서는 물체의 모델 영상이 A고 장면 영상이 B
  - 물체 추적이나 스테레오는 두 영상이 동등한 입장
- 가장 쉬운 매칭 방법
  - mn개 쌍 각각에 대해 거리 계산하고 거리가 임계값보다 작은 쌍을 모두 취함
  - [그림 5-12]는 이런 순진한 방법의 한계를 보여줌

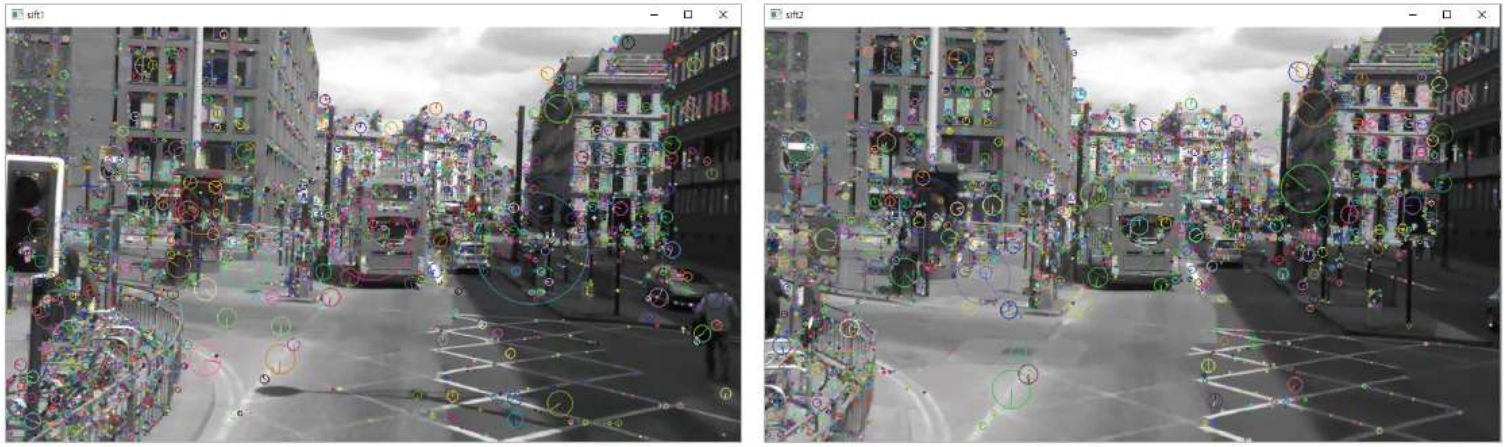


그림 5-12 두 장의 영상에서 추출한 SIFT 특징점을 어떻게 매칭할까?

5.5.1 매칭 전략

■ 두 기술자의 거리 계산

$$d(\mathbf{a},\mathbf{b}) = \sqrt{\sum_{k=1,d}(a_k - b_k)^2} = \|\mathbf{a}-\mathbf{b}\| \qquad (5.12)$$

■ 매칭 전략

- 고정 임계값: 식 (5.13)을 만족하는 모든 쌍을 매칭으로 간주
- 최근접 이웃: $\mathbf{a}_i$는 가장 가까운 $\mathbf{b}_j$를 찾고, $\mathbf{a}_i$와 $\mathbf{b}_j$가 식 (5.13)을 만족하면 매칭 쌍

$$d(\mathbf{a}_i,\mathbf{b}_j) < T \qquad (5.13)$$

- 최근접 이웃 거리 비율 : $\mathbf{a}_i$는 가장 가까운 $\mathbf{b}_j$와 두번째 가까운 $\mathbf{b}_k$를 찾음. $\mathbf{b}_j$와 $\mathbf{b}_k$가 식 (5.14)를 만족하면 $\mathbf{a}_i$와 $\mathbf{b}_j$는 매칭 쌍. 세 가지 전략 중 가장 높은 성능

$$\frac{d(\mathbf{a}_i,\mathbf{b}_j)}{d(\mathbf{a}_i,\mathbf{b}_k)} < T \qquad (5.14)$$

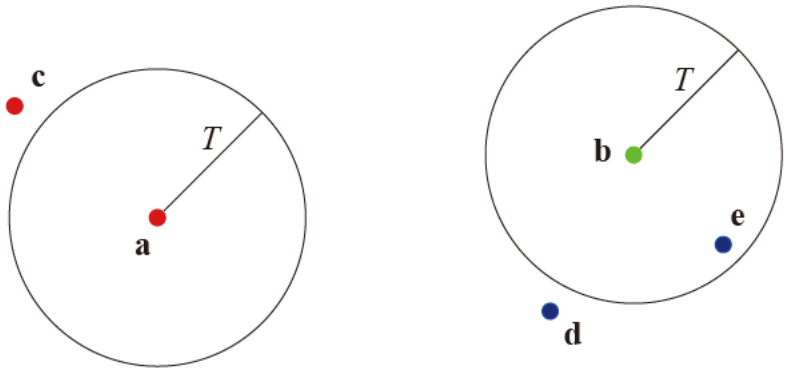


그림 5-13 세 가지 매칭 전략의 비교

- 임계값 T에 따라 거짓 부정false negative과 거짓 긍정false positive 조절

5.5.2 매칭 성능 측정

■ 성능 측정은 컴퓨터 비전에서 아주 중요

- 알고리즘 개선이나 최선의 알고리즘을 선택하는 기준
- 현장 투입 여부 결정하는 기준

■ 정밀도와 재현률

(a) 참 긍정 (b) 거짓 부정 (c) 거짓 긍정 (d) 참 부정

그림 5-14 매칭의 네 가지 경우(같은 색이 진짜 매칭 쌍이고 ----은 매칭 알고리즘이 맺어준 쌍)

표 5-2 혼동 행렬

		정답(GT)	
		긍정	부정
예측	긍정	참 긍정(TP)	거짓 긍정(FP)
	부정	거짓 부정(FN)	참 부정(TN)

$$\left.\begin{aligned} \text{정밀도} &= \frac{TP}{TP+FP} \\ \text{재현율} &= \frac{TP}{TP+FN} \\ F1 &= \frac{2\times \text{정밀도} \times \text{재현율}}{\text{정밀도} + \text{재현율}} \end{aligned}\right\} \quad (5.15)$$

$$\text{정확율} = \frac{TP+TN}{TP+TN+FP+FN} \quad (5.16)$$

5.5.2 매칭 성능 측정

■ ROC Receiver Operating Characteristic 곡선과 AUC Area Under Curve

- 식 (5.14)에서 T를 작게 하면 거짓 긍정률FPR, False Positive Rate은 작아짐
- T를 크게 하면 거짓 긍정률이 커지는데 참 긍정률TPR, True Positive Rate도 따라 커지는 경향
  - T를 점점 키우면서 측정한 거짓 긍정률과 참 긍정률을 나타낸 그래프가 ROC
  - AUC (Area under Curve: 곡선 아래 면적): 성능을 하나의 수치로 표현할 때 사용

$$\left.\begin{aligned} \text{참 긍정률} &= \frac{TP}{TP+FN} \\ \text{거짓 긍정률} &= \frac{FP}{TN+FP} \end{aligned}\right\} \quad (5.17)$$

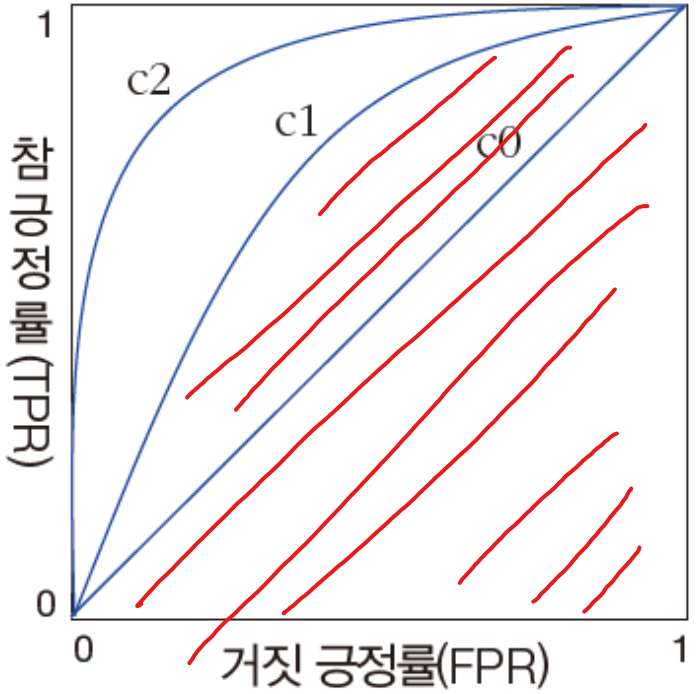


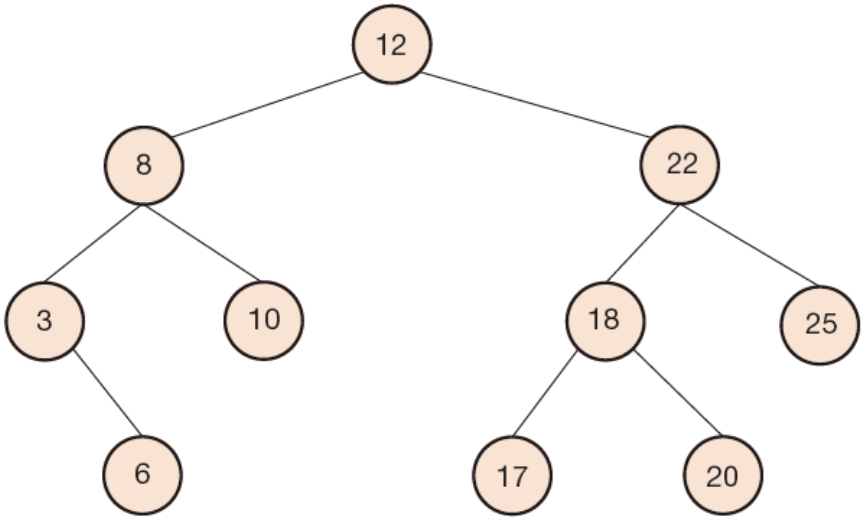
그림 5-15 ROC 곡선

표 5-2 혼동 행렬

		정답(GT)	
		긍정	부정
예측	긍정	참 긍정(TP)	거짓 긍정(FP)
	부정	거짓 부정(FN)	참 부정(TN)

5.5.3 빠른 매칭

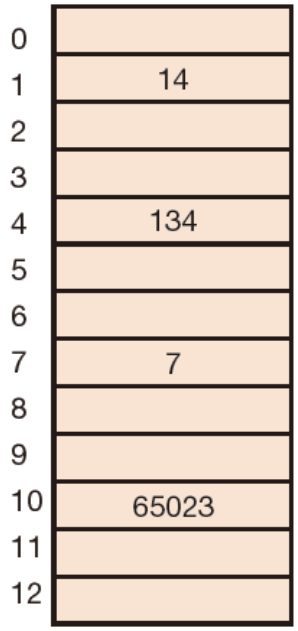
- kd 트리, 위치의존 해싱
  - 레코드가 특징벡터



(a) 이진 탐색 트리

해시 함수

$h(x) = x \% 13$



(b) 해싱

그림 5-16 빠른 탐색을 위한 자료 구조

5.5.3 빠른 매칭

- 순진한 알고리즘
  - 모든 쌍을 일일이 검사 → 시간이 넉넉한 상황에서만 활용 가능

알고리즘 7-1 순진한 매칭 알고리즘

입력 : 첫 번째 영상의 특징 벡터 $\mathbf{a}_i$, $1 \le i \le m$, 두 번째 영상의 특징 벡터 $\mathbf{b}_j$, $1 \le j \le n$, 거리 임계값 T

출력 : 매칭 쌍 리스트 $mlist$

```
1  mlist=∅;
2  for(i=1 to m) {
3    shortest=∞;
4    for(j=1 to n) {
5      dist = d(a_i, b_j);
6      if(dist<shortest) {match=j; shortest = dist;}
7    }
8    if(shortest<T) mlist=mlist ∪(a_i, b_match);
9  }
```

```
import cv2 as cv
import numpy as np
import time
#%%
img1=cv.imread('mot_color70.jpg')[190:350, 440:560]
gray1=cv.cvtColor(img1, cv.COLOR_BGR2GRAY)
img2=cv.imread('mot_color83.jpg'); gray2=cv.cvtColor(img2, cv.COLOR_BGR2GRAY)

cv.imshow("IMG!", img1); cv.imshow("IMG2", img2)
cv.waitKey();
cv.destroyAllWindows()
#%%
sift=cv.SIFT_create()
kp1, des1 =sift.detectAndCompute(gray1, None)
kp2, des2 =sift.detectAndCompute(gray2, None)
print("특징점 개수", len(kp1), len(kp2))

start=time.time()
#bf_matcher=cv.BFMatcher_create(cv.NORM_L2)
bf_matcher=cv.DescriptorMatcher_create(cv.DescriptorMatcher_BRUTEFORCE)
knn_match=bf_matcher.knnMatch(des1, des2, k=2)
```

```
T=0.7; good_match=[]
for nearest1, nearest2 in knn_match:
    if(nearest1.distance/nearest2.distance) < T:
        good_match.append(nearest1)

print("매칭에 걸린 시간", time.time()-start)

img_match=np.empty((max(img1.shape[0], img2.shape[0]),
img1.shape[1]+img2.shape[1], 3), dtype=np.uint8)

cv.drawMatches(img1, kp1, img2, kp2, good_match, img_match,
flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

cv.imshow("Good Matches", img_match)
cv.waitKey()
cv.destroyAllWindows()
```

cv2.DescriptorMatcher_create(matcherType)

: 매칭기 생성자

- matcherType : 생성할 구현 클래스의 알고리즘
- BruteForce : NORM_L2를 사용하는 BFMatcher
- BruteForce-L1 : NORM_L1을 사용하는 BFMatcher
- BruteForce-Hamming : NORM_HAMMING을 사용하는 BRMatcher
- BruteForce-Hamming(2) : NORM_HAMMING2를 사용하는 BFMatcher
- FlannBased : NORM_L2를 사용하는 FlannBasedMatcher
- 파라미터로 구현할 클래스의 알고리즘을 문자열로 전달해줘도 됨

: 생성된 특징 매칭기는 두 개의 디스크립터를 서로 비교하여 매칭 해주는 함수 가짐

: 3개의 함수가 있는데, match(), knnMatch(), radiusMatch()

: 모두 첫 번째 파라미터인 queryDescriptors를 기준으로 두 번째 파라미터인 trainDescriptors에 맞는 매칭을 찾음

- matches : matcher.match(queryDescriptors, trainDescriptors, mask)
  - : 1개의 최적 매칭
  - : queryDescriptors 한 개당 최적의 매칭을 이루는 trainDescriptors를 찾아 결과로 반환

cv2.DescriptorMatcher_create(matcherType)

- matches = matcher.knnMatch(queryDescriptors, trainDescriptors, k, mask, compactResult)
  - : k개의 가장 근접한 매칭
  - : queryDescriptors 한 개당 k개의 최근접 이웃 개수만큼 trainDescriptors에서 찾아 반환
  - : k개의 최근접 이웃 개수만큼이라는 말은 가장 비슷한 k개만큼의 매칭 값을 반환
  - : CompactResult에 default값이 False가 전달되면 매칭 결과를 찾지 못해도 결과에 queryDescriptors의 ID를 보관하는 행을 추가 (True가 전달되면 아무것도 추가하지 않음)
  - - k : 매칭할 근접 이웃 개수
  - - compactResult(optional) : True: 매칭이 없는 경우 매칭 결과에 불포함 (default=False)

: match(), knnMatch(), radiusMatch() 함수의 반환 결과는 DMatch 객체 리스트

DMatch

: 매칭 결과를 표현하는 객체

: DMatch 객체의 queryIdx와 trainIdx로 두 이미지의 어느 지점이 서로 매칭 되었는지 알 수 있음

: distnace로 얼마나 가까운 거리 인지도 알 수 있음

- queryIdx : queryDescriptors의 인덱스
- trainIdx : trainDescriptors의 인덱스
- imgIdx : trainDescriptor의 이미지 인덱스
- distance : 유사도 거리

◆ drawMatches() [1/3]

```
void cv::drawMatches ( InputArray                    img1,
                       const std::vector< KeyPoint > & keypoints1,
                       InputArray                    img2,
                       const std::vector< KeyPoint > & keypoints2,
                       const std::vector< DMatch > &  matches1to2,
                       InputOutputArray              outImg,
                       const Scalar &                matchColor = Scalar::all(-1),
                       const Scalar &                singlePointColor = Scalar::all(-1),
                       const std::vector< char > &   matchesMask = std::vector< char >(),
                       int                           flags = DrawMatchesFlags::DEFAULT
                     )
```

Python:

```
cv.drawMatches( img1, keypoints1, img2, keypoints2, matches1to2, outImg[, matchColor[, singlePointColor[, matchesMask[, flags]]]] ) -> outImg
cv.drawMatches( img1, keypoints1, img2, keypoints2, matches1to2, outImg, matchesThickness[, matchColor[, singlePointColor[, matchesMask[, flags]]]] ) -> outImg
```

Parameters

img1	First source image.
keypoints1	Keypoints from the first source image.
img2	Second source image.
keypoints2	Keypoints from the second source image.
matches1to2	Matches from the first image to the second one, which means that keypoints1[i] has a corresponding point in keypoints2[matches[i]] .
outImg	Output image. Its content depends on the flags value defining what is drawn in the output image. See possible flags bit values below.
matchColor	Color of matches (lines and connected keypoints). If matchColor==**Scalar::all**(-1) , the color is generated randomly.
singlePointColor	Color of single keypoints (circles), which means that keypoints do not have the matches. If singlePointColor==**Scalar::all**(-1) , the color is generated randomly.
matchesMask	Mask determining which matches are drawn. If the mask is empty, all matches are drawn.
flags	Flags setting drawing features. Possible flags bit values are defined by **DrawMatchesFlags**.

5.5.3 빠른 매칭

- 특징 벡터를 미리 인덱싱해 두는 효율적인 알고리즘
  - 두 알고리즘: *kd* 트리와 위치의존 해싱

알고리즘 7-2 빠른 매칭 알고리즘

입력 : 첫 번째 영상의 특징 벡터 $\mathbf{a}_i$, $1 \le i \le m$, 두 번째 영상의 특징 벡터 $\mathbf{b}_j$, $1 \le j \le n$, 거리 임계값 T

출력 : 매칭쌍 리스트 $mlist$

```
1  mlist=∅;
2  for(i=1 to m) {
3    kd트리나 해싱 알고리즘으로 a_i의 최근접 또는 근사 최근접 이웃 b_match를 탐색한다.
4    if(d(a_i, b_match)<T) mlist=mlist ∪ (a_i, b_match);
5  }
```

  - 일반적인 탐색 기법으로 데이터마이닝, 정보 검색, 생물 정보학 등에서도 활용

kd 트리

- 이진검색 트리 (BST)
  - 루트를 기준으로 왼쪽 부분 트리는 루트보다 작은 값, 오른쪽은 큰 값을 갖는 이진 트리 (이 성질이 부분 트리에 재귀적으로 반복 적용), t 는 노드를 나타냄

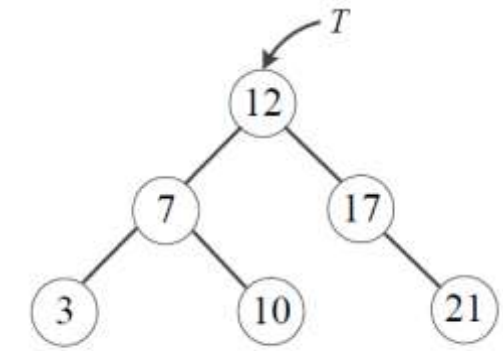


그림 7-5 이진검색 트리

균형이 잡힌 경우
탐색시간 $O(\log_2 n)$

알고리즘 7-3 이진검색 트리의 검색

입력 : 이진검색 트리 T, 검색 키 v

출력 : 검색 결과

```
1   t=search(T, v);
2   if(t=Nil) T 안에 v가 없음을 알린다.
3   else t를 검색 결과로 취한다.
4   function search(t,v) {
5     if(t=Nil or t.key=v) return t;
6     else {
7       if(v<t.key) return search(t.leftchild, v);
8       else return search(t.rightchild, v);
9     }
10  }
```

kd 트리

- BST를 적용할 수 있나?
  - 매칭 문제
    - 검색 키(특징 벡터)가 여러 개의 실수로 구성된 벡터임
    - 동일한 값을 갖는 노드를 찾는 것이 아니라, 최근접 이웃을 찾음

    →BST를 그대로 적용 불가능

- *kd* 트리
  - 두 가지 다른 점을 수용할 수 있게 BST를 확장한 기법 [Bently75]

*k*d 트리

- 표기
  - n개의 벡터를 가지고 kd 트리 구축 $X=\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$: n은 관심점 개수
  - $\mathbf{x}_i$는 d차원 벡터

- kd 트리의 원리
  - 루트 노드는 X를 두 개의 부분 집합 X_{left}와 X_{right} 나눔
  - 이때 분할 기준을 어떻게 선택하나?
    - d개의 차원(축) 중에 어는 것을 쓸 것인가?
      - 분할 효과를 극대화하려면 각 차원의 분산을 계산한 후, 최대 분산을 갖는 축 k를 선택
    - 축을 선택했다면 n개의 샘플 중 어는 것을 기준으로 X를 분할할 것인가?
      - X_{left}와 X_{right}의 크기를 갖게 하여 균형 잡힌 트리를 만들어야 함.
      - X를 차원 k로 정렬하고, 그 결과의 중앙 값을 분할 기준으로 삼는다.
  - X를 X_{left}와 X_{right}로 분할한 후, 각각에 같은 과정을 재귀적으로 반복하면 kd트리 완성

kd 트리

알고리즘 7-4 kd 트리 만들기

입력 : 특징 벡터 집합 $X=\{\mathbf{x}_i, i=1,2,\cdots,n\}$

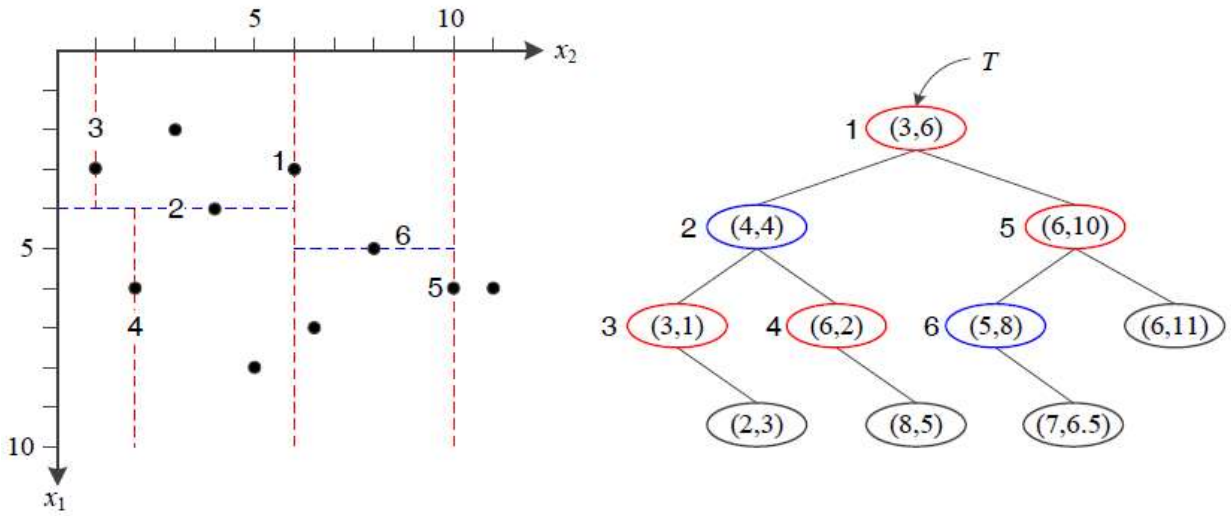
출력 : kd 트리 T

```
1   T=make_kdtree(X);
2   function make_kdtree(X) {
3     if(X=Ø) return Nil;
4     else if(|X|=1) {  // 단말 노드
5       트리 노드 node를 생성한다.
6       node.vector = x_m;  // x_m은 X에 있는 벡터
7       node.leftchild = node.rightchild =Nil;
8       return node;
9     }
10    else {
11      d개의 차원 각각에 대해 X의 분산을 구하고 최대 분산을 갖는 차원을 k라 하자.
12      X를 k차원을 기준으로 정렬하여 리스트 X_sorted를 만든다.
13      X_sorted에서 중앙값을 x_m, 왼쪽 부분집합을 X_left, 오른쪽 부분집합을 X_right라 하자.
14      트리 노드 node를 생성한다.
15      node.dim=k;  // 어느 차원으로 분할하는지
16      node.vector = x_m;  // 어떤 특징 벡터로 분할하는지
17      node.leftchild =make_kdtree(X_left);
18      node.rightchild =make_kdtree(X_right);
19      return node;
20    }
21  }
```

kd 트리

예제 7-3 **kd 트리 만들기**

설명을 쉽게 하기 위해 $d=2$로 한정하고, $X=\{\mathbf{x}_1=(3,1),\ \mathbf{x}_2=(2,3),\ \mathbf{x}_3=(6,2),\ \mathbf{x}_4=(4,4),\ \mathbf{x}_5=(3,6),\ \mathbf{x}_6=(8,5),\ \mathbf{x}_7=(7,6.5),\ \mathbf{x}_8=(5,8),\ \mathbf{x}_9=(6,10),\ \mathbf{x}_{10}=(6,11)\}$이라 하자. [그림 7-6(a)]는 주어진 특징 벡터의 집합을 보여준다.



(a) 특징 벡터의 집합 (b) 완성된 kd 트리

그림 7-6 kd 트리

먼저, 루트 노드로 결정할 만한 분할 기준을 찾아보자. 두 개의 차원은 각각 3, 2, 6, 4, …, 6과 1, 3, 2, 4, …, 11의 값을 가진다. 이들의 분산을 구해 보면 두 번째가 더 크다. 따라서 [알고리즘 7-4]의 11행에서 k는 2가 된다. 두 번째 차원을 기준으로 X를 정렬하면 $X_{sorted}=\{\mathbf{x}_1, \mathbf{x}_3, \mathbf{x}_2, \mathbf{x}_4, \mathbf{x}_6, \mathbf{x}_5, \mathbf{x}_7, \mathbf{x}_8, \mathbf{x}_9, \mathbf{x}_{10}\}$이 된다. 이 리스트의 중앙값 $\mathbf{x}_5$를 기준으로 좌우를 분할하면, $X_{left}=\{\mathbf{x}_1, \mathbf{x}_3, \mathbf{x}_2, \mathbf{x}_4, \mathbf{x}_6\}$, $X_{right}=\{\mathbf{x}_7, \mathbf{x}_8, \mathbf{x}_9, \mathbf{x}_{10}\}$이 된다. 이제 14~16행에서 노드를 하나 할당받아 값을 채운다. 이렇게 만들어진 노드가 [그림 7-6]에서 T가 가리키는 루트 노드이다.

이 루트 노드의 물리적인 의미를 해석해 보자. [그림 7-6(a)]에서 1 옆의 빨간색 선이 이 노드의 역할을 보여준다. 이 노드는 $k=2$에 해당하는 x_2축을 기준으로 공간을 둘로 분할한다. 이때 왼쪽 영역에 있는 점들이 X_{left}가 되고 오른쪽 영역은 X_{right}가 된다. 이제 X_{left}와 X_{right} 각각에 같은 과정을 재귀적으로 반복하면 [그림 7-6(b)]와 같은 kd 트리가 완성된다. 그림에서는 기준이 되는 축을 쉽게 구분할 수 있도록 각각 다른 색으로 표시하였다. x_1축이 기준이라면 파란색, x_2축이 기준이라면 빨간색이다.

*k*d 트리

- *k*d 트리에서 최근접 이웃 탐색
  - 새로운 특징 벡터 **x**가 입력되면 **x**의 최근접 이웃을 어떻게 찾을까?
  - 예) **x**가 =(7, 5.5)
    - 루트가 x_2축을 기준으로 하므로 5.5를 6과 비교하고 작으므로 왼쪽으로 분기
    - (4,4) 노드가 x_1축을 기준으로 하므로 7과 4를 비교하고 크므로 오른쪽으로 분기
    - 반복하면 (8,5) 노드에 도착

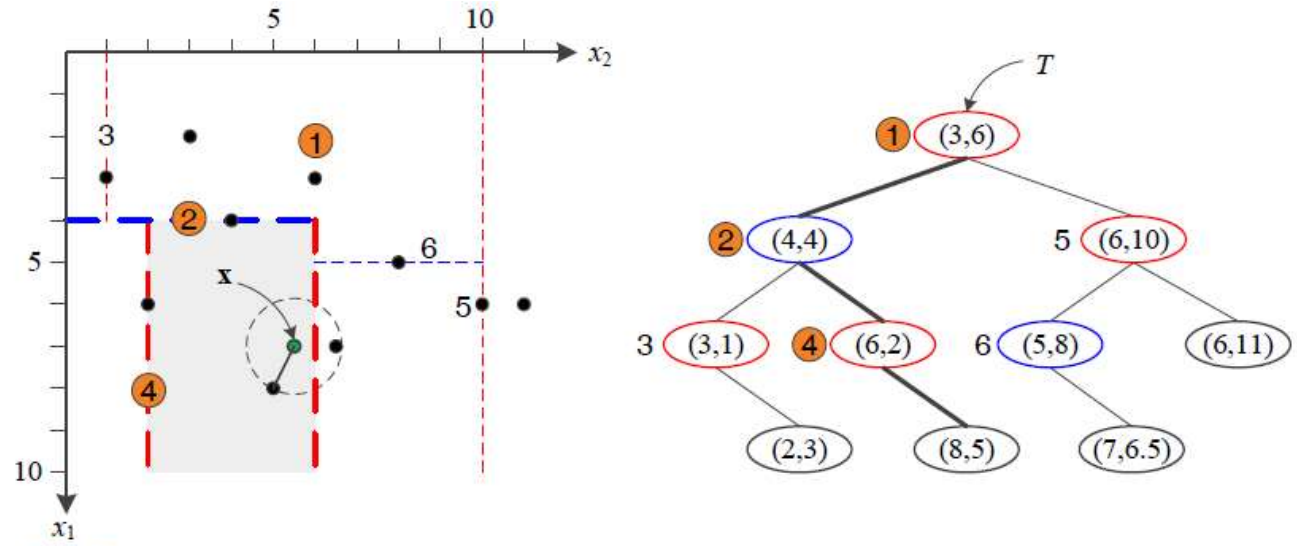


그림 7-7 kd 트리에서 검색하는 예

*k*d 트리

- *k*d 트리를 이용한 최근접 이웃 탐색
  - 리프 노드 (8,5)를 답으로 취하면 될까?
    - 최근접일 가능성이 있지만 반드시 그렇진 않다. 분할 평면의 건너편에 더 가까운 노드가 있을 수 있음
  - 스택을 이용한 백트래킹
    - 한정 분기(branch-and-bound)를 적용하면서 ④→②→①

- *d*=10을 넘으면 순진한 알고리즘과 비슷한 낮은 속도
  - 어떻게 시간 효율을 회복할 수 있을까?

- 대안: 근사 최근접 이웃을 찾는다.(approximate nearest neighbor)
  - 우선순위 큐(힙)를 이용한 백트랙킹
    - 탐색 키와 가까운 순서(우선 순위)로 큐를 생성
    - 우선 순위에 따라 ①→④→② 순으로 처리

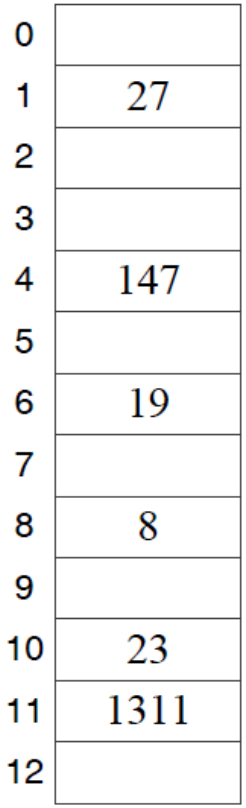
해싱

해싱의 원리

- 해시 함수는 키 값을 해시 테이블의 주소로 변환
  - 배열로 해시테이블 구현, 탐색 시간 복잡도는 O(1)
- 테이블에 골고루 배치할수록 좋은 해시 함수
- 충돌 해결책 필요

x : 탐색 키 => $h(x)$: 해시 주소

해시 함수 $h(x)=x \bmod 13$



← 버켓(bucket)

그림 7-8 해싱의 원리

해싱

- 매칭에 적용
  - 일반 해싱과 다른 점
    - 키는 단일 값이 아니라 실수 벡터임
    - 동일한 요소가 아니라 최근접 이웃을 찾음
  - 가장 크게 다른 점: 일반 해싱과 정반대 목표
    - 일반 해싱은 데이터를 골고루 배치하는 반면, 매칭에서는 가까운 벡터들은 같은 통(bucket)에 담길 확률이 높아야 함

→ 위치의존(locality-sensitive) 해싱(hashing)으로 해결

해싱

- 위치의존 해싱 [Andoni2008]
  - 하나의 해시 함수가 아니라, 해시 함수 집합 H 에서 여러 개를 임의 선택하여 사용
  - H 에 속한 해시 함수 h가 식 (7.8)을 만족하면, H 는 위치의존적

임의의 두점 $\mathbf{a}$와 $\mathbf{b}$에 대해,

$$\begin{aligned} \|\mathbf{a}-\mathbf{b}\| \le R\text{이면, } p(h(\mathbf{a})=h(\mathbf{b})) \ge p_1\text{이고} \\ \|\mathbf{a}-\mathbf{b}\| \ge cR\text{이면, } p(h(\mathbf{a})=h(\mathbf{b})) \le p_2\text{이다.} \\ \text{이때 } c>1,\ p_1>p_2 \end{aligned} \tag{7.8}$$

  - 위치의존의 의미는?

가까운 두 벡터는 같은 통(bucket, bin)에 담길 (해시 함수 값이 같을) 확률이 크고, 먼 벡터는 같은 통에 담길 확률이 작음

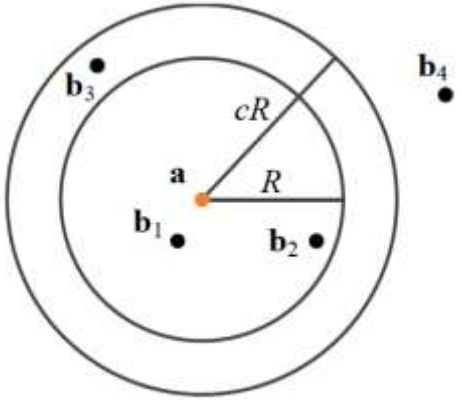


그림 7-9 조건식 (7.8)의 의미

해싱

- *H* (해시함수 집합)를 어떻게 만드나?
  - 여럿 개발되어 있는데, 식 (7.9)는 그 중 하나
  - 난수로 **r**과 *b*를 설정하여 원하는 수만큼 함수 생성 가능

$$h(\mathbf{x}) = \left\lfloor \frac{\mathbf{r} \cdot \mathbf{x} + b}{w} \right\rfloor \qquad (7.9)$$

- 해시 함수 *h* 의 동작
  - *d* 차원 공간을 **r**에 수직인 초평면(hyperplane)으로 분할
  - *w*는 구간의 간격으로서, 작으면 촘촘하게 크면 듬성듬성 분할

해싱

예제 7-4 **위치의존 해시 함수**

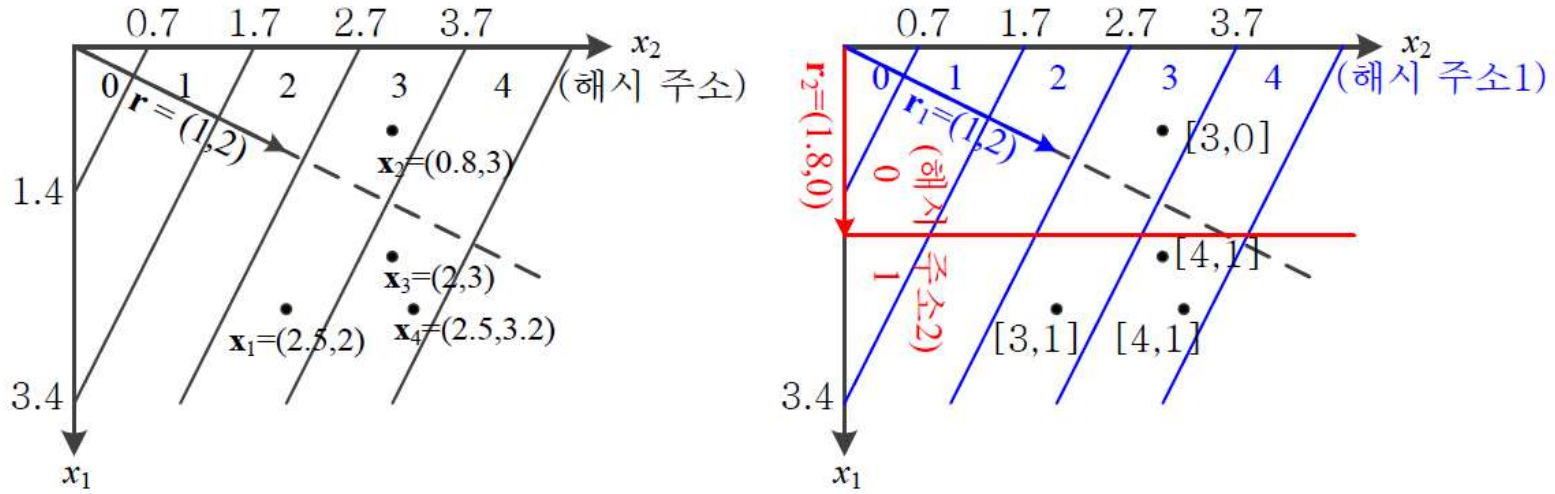
특징 벡터는 $\mathbf{x}=(x_1, x_2)$로 표현되는 2차원이라 가정한다. w는 2로 설정되어 있고, 난수를 생성하여 $\mathbf{r}=(1, 2)$, $b=0.6$을 얻었다고 하자. 식 (7.9)에 따른 해시 함수는 다음과 같다.

$$h(\mathbf{x}) = \left\lfloor \frac{x_1 + 2x_2 + 0.6}{2} \right\rfloor$$

몇 개의 점을 대상으로 이 해시 함수가 특징 벡터를 어떤 주소로 매핑해 주는지 살펴보자. 해시 함수는 2차원 공간을 띠 모양의 영역으로 분할하는데, 원점에서 오른쪽으로 진행하며 0, 1, 2, …라는 주소를 부여한다. [그림 7-10]은 네 개의 특징 벡터 $\mathbf{x}_1=(2.5, 2)$, $\mathbf{x}_2=(0.8, 3)$, $\mathbf{x}_3=(2, 3)$, $\mathbf{x}_4=(2.5, 3.2)$가 어떤 영역으로 매핑되는지 보여준다. [그림 7-10(a)]를 보면 결과적으로 이들은 각각 주소가 3, 3, 4, 4인 통에 담긴다.

$$h(2.5, 2) = \left\lfloor \frac{7.1}{2} \right\rfloor = 3, \quad h(0.8, 3) = \left\lfloor \frac{7.4}{2} \right\rfloor = 3, \quad h(2, 3) = \left\lfloor \frac{8.6}{2} \right\rfloor = 4, \quad h(2.5, 3.2) = \left\lfloor \frac{9.5}{2} \right\rfloor = 4$$

해싱



(a) 해시 함수 한 개 사용

(b) 해시 함수 두 개 사용

그림 7-10 해시 함수의 공간 분할과 주소 매핑

이제 두 개의 해시 함수 h_1과 h_2를 사용하는 [그림 7-10(b)]로 관심을 옮겨 보자. h_1은 이전과 같이 $\mathbf{r}_1=(1,2)$, $b=0.6$으로 정의되고, h_2는 $\mathbf{r}_2=(1.8,0)$, $b=0$으로 정의한다고 하자. 이 상황에서는 값 두 개로 주소가 정해진다. 예를 들어 점 $\mathbf{x}_1=(2.5,2)$는 주소 [3,1]을 가진다. 나머지 점의 주소도 계산해 보면, 그림에 표시된 주소를 갖는다. 이때 해시 함수를 두 개 사용한 효과를 관찰해 보자. 왼쪽 그림에서는 $\mathbf{x}_1$과 $\mathbf{x}_2$가 멀리 떨어져 있음에도 불구하고 주소3에 같이 담겨있다. 하지만 두 개의 해시 함수를 사용하는 오른쪽 그림에서는 이들이 각각 [3,1]과 [3,0]이라는 주소를 가져 다른 통에 담겨있는 것을 확인할 수 있다. 서로 가까운 $\mathbf{x}_3$와 $\mathbf{x}_4$는 여전히 같은 통 [4,1]에 들어 있다.

해싱

- 위치의존을 만족하는 해시 함수 여러 개를 쓰면,
  - 가까운 벡터가 같은 통에 담길 확률이 충분히 높을까? → 그렇지 않다.

- 확률을 높이는 추가적인 방안
  - 해시 테이블을 여러 개 사용
  - 가까운 두 벡터가 여러 테이블 중 하나에라도 같은 통에 있으면 성공

알고리즘 7-7 위치의존 해시 테이블의 구축

입력 : 특징 벡터 집합 $X=\{\mathbf{x}_i, i=1,2,\cdots,n\}$, 해시 테이블이 사용하는 해시 함수의 개수 k, 해시 테이블의 개수 L

출력 : L개의 해시 테이블

```
1  for(j=1 to L) {
2    for(i=1 to k) {
3      가우시안 분포에 따른 난수를 생성하여 (r_i, b_i)를 설정한다.
4      (r_i, b_i)로 해시 함수 h_i를 만든다.
5    }
6    해시 함수 g_j=(h_1, h_2, ···, h_k)를 만든다.
7  }
8  for(i=1 to n)
9    for(j=1 to L) x_i를 g_j로 해싱하여 해당 주소의 통에 담는다.
```

해싱

검색 알고리즘

알고리즘 7-8 위치의존 해시 테이블에서 검색

입력 : L개의 해시 테이블, 특징 벡터 $\mathbf{x}$, 매개변수 R과 N

출력 : 근사 최근접 이웃 $\mathbf{x}_{nearest}$

```
1  Q=∅; // 근사 최근접 이웃을 저장
2  for(j=1 to L) {
3    j번째 해시 테이블에서 주소 g_j(x)인 통을 조사한다.
4    이 통에 있는 점들 중 x와 거리가 R 이내인 것을 Q에 추가한다.
5    Q의 크기가 N을 넘으면 break; // 이 행을 제거하면 R 이내인 모든 점을 Q에 저장
6  }
7  Q에서 거리가 가장 짧은 것을 x_nearest로 취한다.
```

5.5.4 프로그래밍 실습: FLANN을 이용한 특징점 매칭

프로그램 5-3 FLANN 라이브러리를 이용한 SIFT 매칭

```
01  import cv2 as cv
02  import numpy as np
03  import time
04
05  img1=cv.imread('mot_color70.jpg')[190:350,440:560] # 버스를 크롭하여 모델 영상으로 사용
06  gray1=cv.cvtColor(img1,cv.COLOR_BGR2GRAY)
07  img2=cv.imread('mot_color83.jpg')                  # 장면 영상
08  gray2=cv.cvtColor(img2,cv.COLOR_BGR2GRAY)
09
10  sift=cv.SIFT_create()
11  kp1,des1=sift.detectAndCompute(gray1,None)
12  kp2,des2=sift.detectAndCompute(gray2,None)
13  print('특징점 개수:',len(kp1),len(kp2)) ①
14
```

5.5.4 프로그래밍 실습: FLANN을 이용한 특징점 매칭

```
15  start=time.time()
16  flann_matcher=cv.DescriptorMatcher_create(cv.DescriptorMatcher_FLANNBASED)
17  knn_match=flann_matcher.knnMatch(des1,des2,2)
18
19  T=0.7
20  good_match=[]
21  for nearest1,nearest2 in knn_match:
22      if (nearest1.distance/nearest2.distance)<T:
23          good_match.append(nearest1)
24  print('매칭에 걸린 시간:',time.time()-start) ②
25
26  img_match=np.empty((max(img1.shape[0],img2.shape[0]),img1.shape[1]+img2.
    shape[1],3),dtype=np.uint8)
27  cv.drawMatches(img1,kp1,img2,kp2,good_match,img_match,flags=cv.
    DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
28
29  cv.imshow('Good Matches', img_match)
30
31  k=cv.waitKey()
32  cv.destroyAllWindows()
```

식 (5.14)의 최근접 이웃 거리 비율 적용

5.5.4 프로그래밍 실습: FLANN을 이용한 특징점 매칭

아주 빠른 FLANN

```
특징점 개수: 231 4096 ①
매칭에 걸린 시간: 0.03124260902404785 ②
```

5.6 호모그래피 추정

■ SIFT 검출과 기술자 추출 이후에 해야 할 일

- 물체 위치 찾기(아웃라이어 걸러내기)
  - 특징 벡터가 개별적으로 매칭을 수행
    - → 지역정보만 사용한 매칭: 오류 발생
    - → 아웃라이어 매칭 (거짓 긍정) 발생
    - → 광역정보(같은 물체의 다른 대응 쌍은 같은 변환)를 사용해야한다:
    - 기하 정렬을 이용하여 인라이어 집합을 찾아내고, 기하 변환 행렬을 추정해야 함

- 호모그래피$_{\text{homography}}$는 이런 일을 해줌

5.6.1 문제의 이해

■ 3차원 투영

- 3차원 공간에 있는 평면 P의 두 점 p1과 p2 그리고 카메라 A와 B가 있는 상황
- p1과 p2가 카메라의 2차원 영상 평면에 투영 변환됨

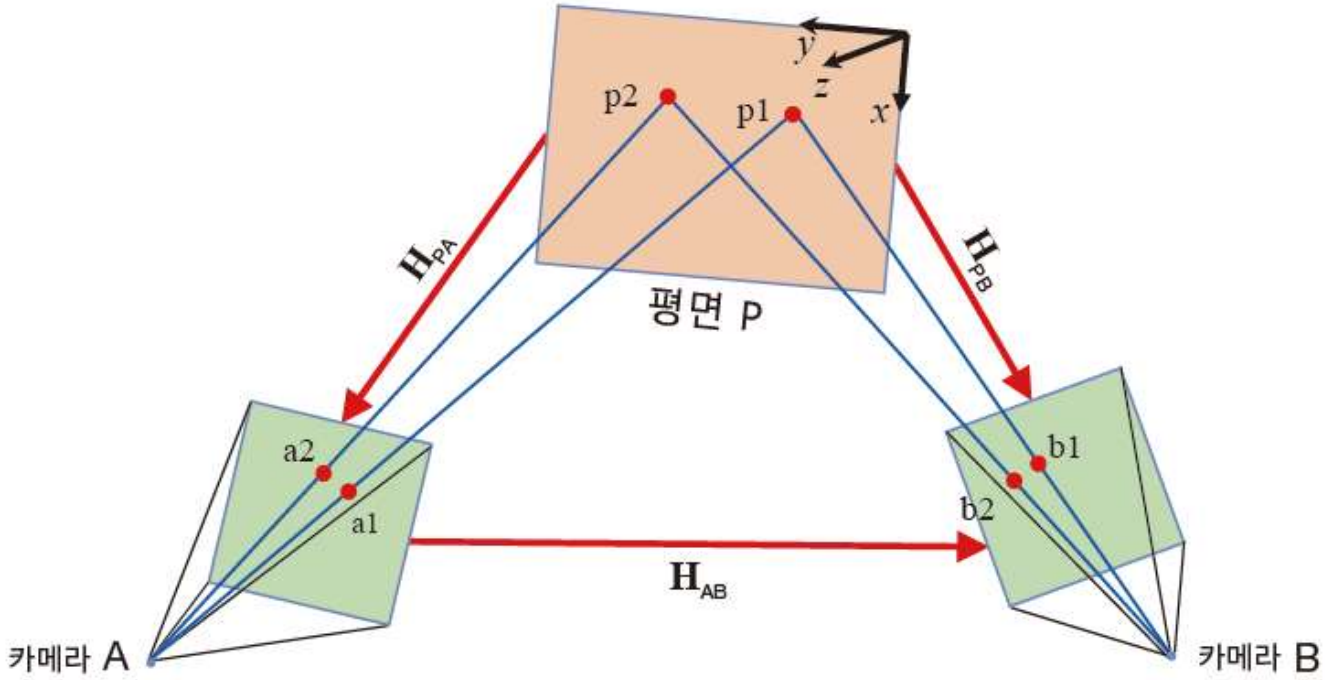


그림 5-18 호모그래피

5.6.1 문제의 이해

■ 투영 변환

- 3차원 점 p를 동차 좌표로 표현하면 (x,y,z,1). 투영 변환 행렬은 4*4
- p1과 p2가 같은 평면 상에 있다고 가정하고 z=0으로 간주하면 3*3 행렬로 표현 가능
- 이런 제한된 상황에서 이루어지는 투영 변환을 평면 호모그래피(줄여 호모그래피)라 부름

■ [그림 5-18]에는 세 개의 평면이 있음

- 물체가 놓인 평면 P, 카메라 A와 B의 영상 평면
- 어떤 평면의 점 **a**를 다른 평면의 점 **b**로 투영하는 변환 행렬을 **H**라 하면,

$$\mathbf{b}^{\mathrm{T}} = \begin{pmatrix} b_x \\ b_y \\ 1 \end{pmatrix} = \begin{pmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & 1 \end{pmatrix} \begin{pmatrix} a_x \\ a_y \\ 1 \end{pmatrix} = \mathbf{H}\mathbf{a}^{\mathrm{T}} \qquad (5.18)$$

풀어쓰면, $b_x = \dfrac{h_{00}a_x + h_{01}a_y + h_{02}}{h_{20}a_x + h_{21}a_y + 1}$, $b_y = \dfrac{h_{10}a_x + h_{11}a_y + h_{12}}{h_{20}a_x + h_{21}a_y + 1}$ (5.19)

5.6.1 문제의 이해

■ 방정식을 풀어 H 구하기

- 매칭 쌍 $(\mathbf{a}_1, \mathbf{b}_1)$, $(\mathbf{a}_2, \mathbf{b}_2)$, $(\mathbf{a}_3, \mathbf{b}_3)$, ...을 가지고 품
- 알아내야 할 값은 8개. 매칭 쌍 하나가 두 방정식을 제공하므로 최소 4개 매칭 쌍이면 됨
- 실제에서는 많은 매칭 쌍을 이용하여 최적의 **H**를 계산

■ 매칭 쌍을 가지고 H 추정

- 매칭 쌍 n개를 $(\mathbf{a}_1, \mathbf{b}_1)$, $(\mathbf{a}_2, \mathbf{b}_2)$, $(\mathbf{a}_3, \mathbf{b}_3)$, ..., $(\mathbf{a}_n, \mathbf{b}_n)$으로 표기하면 식 (5.20)이 성립
  - **B**는 $\mathbf{b}_i$를 i번째 열에 배치한 $3 * n$ 행렬이고 **A**는 $\mathbf{a}_i$를 i번째 열에 배치한 $3 * n$ 행렬

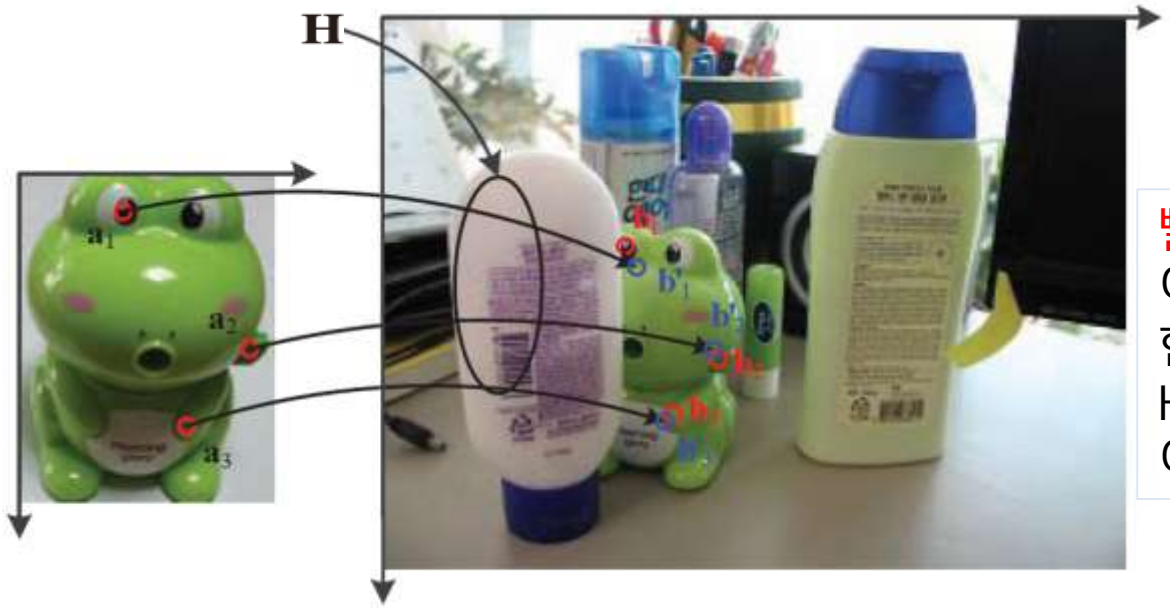
$$\mathbf{B} = \mathbf{HA} \qquad (5.20)$$

최소제곱법과 강인한 추정 기법

- 매칭 문제로 확장
  - 입력은 매칭 쌍 집합 $X=\{(\mathbf{a}_1,\mathbf{b}_1),(\mathbf{a}_2,\mathbf{b}_2),\ldots,(\mathbf{a}_n,\mathbf{b}_n)\}$

$$\mathbf{H}=\begin{pmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & 1 \end{pmatrix}$$

  - 모델은? → 같은 기하 변환 행렬(이동, 크기, 회전 변환 포함)
    - $\boldsymbol{b}_i' = \boldsymbol{H}\boldsymbol{a}_i$



(a) 모델 영상 (b) 장면 영상

빨간색 점은 관찰된 점이고 파란색 점은 추정한 H로 예측한 점이다. H를 호모그래피 행렬이라 부른다.

5.6.2 강인한 호모그래피 추정

■ 최소평균제곱오차 방법

- 식 (5.21)의 E가 최소인 **H**를 찾음
  - numpy의 lstsq 또는 scipy의 leastsq 함수 사용하여 풀 수 있음
  - 모든 매칭 쌍이 같은 자격으로 참여하므로 강인함 없음

$$E = \frac{1}{n}\sum_{i=1,n} \|\mathbf{H}\mathbf{a}_i^{\mathrm{T}} - \mathbf{b}_i\|_2^2 \qquad (5.21)$$

■ 식 (5.21)의 평균 대신 중앙값 사용하면 강인함 확보 가능

- 아웃라이어는 중앙값 계산까지만 영향을 미치고 수렴 여부 결정에서는 빠짐

최소제곱중앙값 : $\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\ \underset{i}{med}\ r_i^2$

최소제곱법과 강인한 추정 기법

- 최소제곱법은 아웃라이어 있으면 오작동: outlier에 민감
  - 예) 아웃라이어 $\mathbf{x}_5$가 포함되면, l_2를 선호
    → 이런 경우에는 강인한 추정 기법 필요

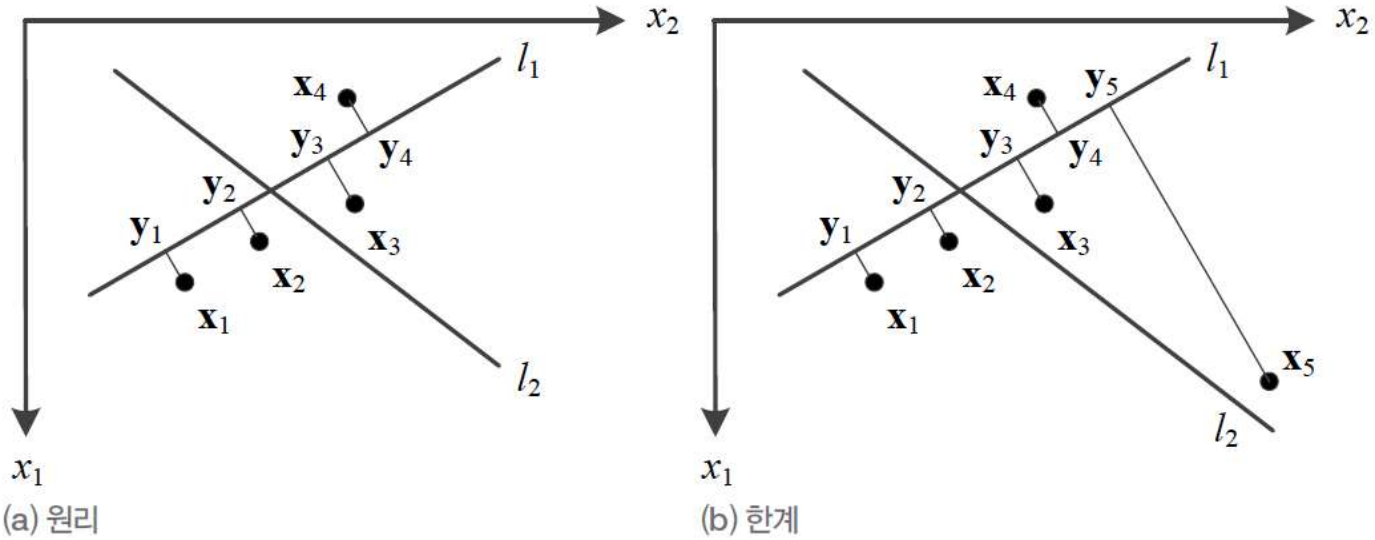


그림 7-12 최소제곱법

RANSAC (Random sample consensus)

- RANSAC은 컨센서스가 최대인, 즉 가장 많은 수의 데이터들로부터 지지를 받는 모델을 선택하는 방법이다.
- 원리
  - 직선 검출하는 3장의 그림 3-31과 같은 원리 (y=ax+b에서 a, b를 추정)

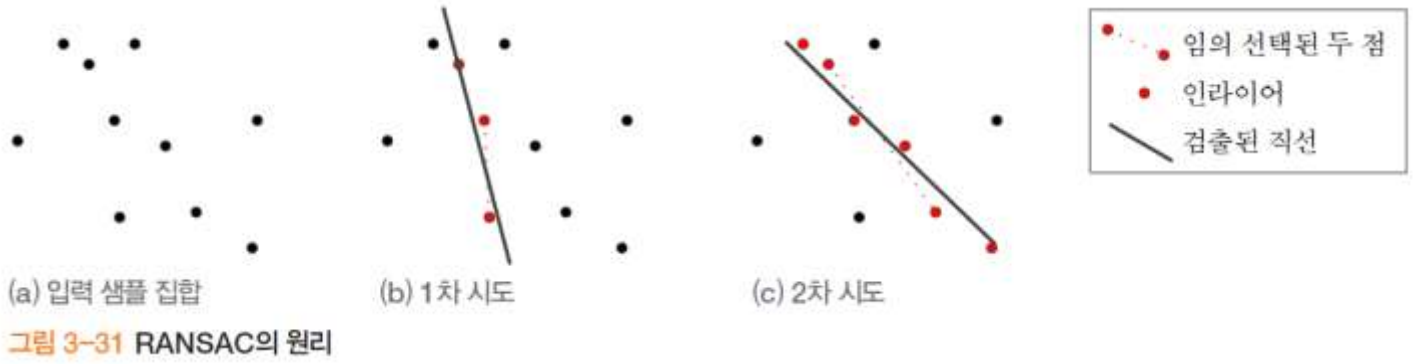


그림 3-31 RANSAC의 원리

- 여기서는,
  - 매칭 쌍 집합 $X=\{(\mathbf{a}_1,\mathbf{b}_1),(\mathbf{a}_2,\mathbf{b}_2),\ldots,(\mathbf{a}_n,\mathbf{b}_n)\}$를 가지고 **H**를 추정할 수 있게 확장

5.6.2 강인한 호모그래피 추정

■ 더욱 강인한 RANSAC

[알고리즘 5-2] 호모그래피 추정을 위한 RANSAC

입력: 매칭 쌍 집합 $X=\{(\mathbf{a}_1,\mathbf{b}_1),(\mathbf{a}_2,\mathbf{b}_2),\cdots,(\mathbf{a}_n,\mathbf{b}_n)\}$,, 반복 횟수 m, 임곗값 t, d, e

출력: 최적 호모그래피 $\hat{\mathbf{H}}$

1. h=[]
2. for j=1 to m
3. X에서 네 쌍을 랜덤하게 선택하고 식 (5.21)을 풀어 호모그래피 행렬 **H**를 추정한다.
4. 이들 네 쌍으로 *inlier* 집합을 초기화한다.
5. for (3행에서 선택한 네 쌍을 제외한 모든 쌍 p에 대해)
6. if (p가 허용 오차 t 이내로 **H**에 적합하면) p를 *inlier*에 삽입한다.
7. if (*inlier*가 d개 이상의 요소를 가지면)
8. *inlier*의 모든 요소를 가지고 호모그래피 행렬 **H**를 다시 추정한다.
9. if (8행에서 적합 오차가 e보다 작으면) **H**를 h에 삽입한다.
10. h에 있는 호모그래피 중에서 가장 좋은 것을 $\hat{\mathbf{H}}$로 취한다.

5.6.2 강인한 호모그래피 추정

- PROSAC [Chum2005] RANSAC 보다 좋음
  - 행 3에서 대응 쌍의 품질(거리에 기반)에 따라 선택 확률을 결정하여 성능 향상 꾀함

3 | 매칭 점수가 높을수록 선택 확률이 높은 방식에 따라, X에서 세 쌍을 선택한다.

5.6.3 프로그래밍 실습: 호모그래피 추정

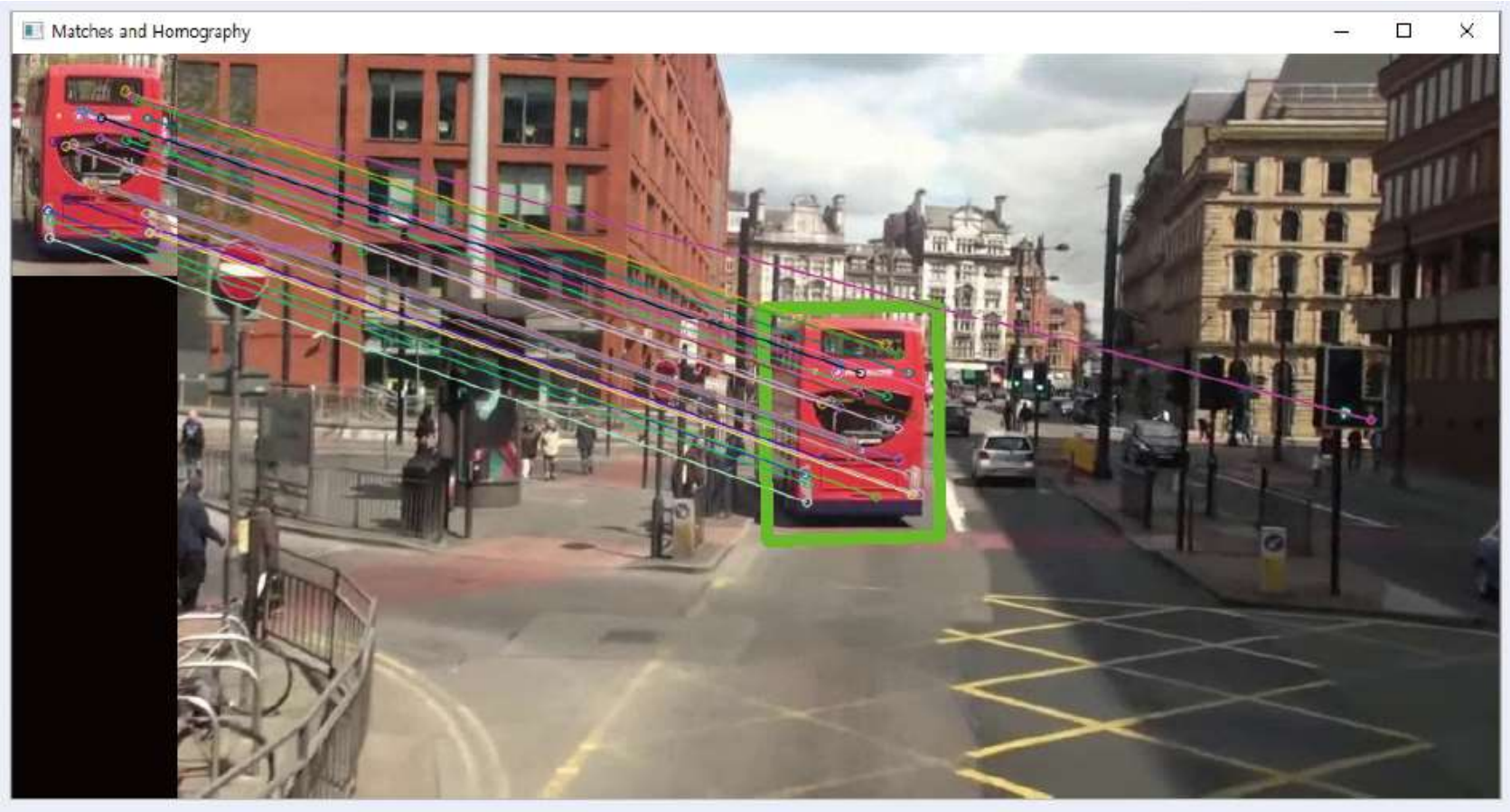
프로그램 5-4 RANSAC을 이용해 호모그래피 추정하기

```
01  import cv2 as cv
02  import numpy as np
03
04  img1=cv.imread('mot_color70.jpg')[190:350,440:560] # 버스를 크롭하여 모델 영상으로 사용
05  gray1=cv.cvtColor(img1,cv.COLOR_BGR2GRAY)
06  img2=cv.imread('mot_color83.jpg')                  # 장면 영상
07  gray2=cv.cvtColor(img2,cv.COLOR_BGR2GRAY)
08
09  sift=cv.SIFT_create()
10  kp1,des1=sift.detectAndCompute(gray1,None)
11  kp2,des2=sift.detectAndCompute(gray2,None)
12
13  flann_matcher=cv.DescriptorMatcher_create(cv.DescriptorMatcher_FLANNBASED)
14  knn_match=flann_matcher.knnMatch(des1,des2,2)      # 최근접 2개
15
16  T=0.7
17  good_match=[]
18  for nearest1,nearest2 in knn_match:
19      if (nearest1.distance/nearest2.distance)<T:
20          good_match.append(nearest1)
21
22  points1=np.float32([kp1[gm.queryIdx].pt for gm in good_match])
23  points2=np.float32([kp2[gm.trainIdx].pt for gm in good_match])
24
```

5.6.3 프로그래밍 실습: 호모그래피 추정

```
25  H,_=cv.findHomography(points1,points2,cv.RANSAC)
26
27  h1,w1=img1.shape[0],img1.shape[1]                    # 첫 번째 영상의 크기
28  h2,w2=img2.shape[0],img2.shape[1]                    # 두 번째 영상의 크기
29
30  box1=np.float32([[0,0],[0,h1-1],[w1-1,h1-1],[w1-1,0]]).reshape(4,1,2)
31  box2=cv.perspectiveTransform(box1,H)
32
33  img2=cv.polylines(img2,[np.int32(box2)],True,(0,255,0),8)
34
35  img_match=np.empty((max(h1,h2),w1+w2,3),dtype=np.uint8)
36  cv.drawMatches(img1,kp1,img2,kp2,good_match,img_match,flags=cv.
    DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
37
38  cv.imshow('Matches and Homography',img_match)
39
40  k=cv.waitKey()
41  cv.destroyAllWindows()
```

5.6.3 프로그래밍 실습: 호모그래피 추정



findHomography() [1/2]

```
Mat cv::findHomography ( InputArray   srcPoints,
                         InputArray   dstPoints,
                         int          method = 0,
                         double       ransacReprojThreshold = 3,
                         OutputArray  mask = noArray(),
                         const int    maxIters = 2000,
                         const double confidence = 0.995
                       )
```

Python:

```
cv.findHomography( srcPoints, dstPoints[, method[, ransacReprojThreshold[, mask[, maxIters[, confidence]]]]] ) -> retval, mask
```

Finds a perspective transformation between two planes.

Parameters

srcPoints Coordinates of the points in the original plane, a matrix of the type CV_32FC2 or vector<Point2f> .

dstPoints Coordinates of the points in the target plane, a matrix of the type CV_32FC2 or a vector<Point2f> .

method Method used to compute a homography matrix. The following methods are possible:

- **0** - a regular method using all the points, i.e., the least squares method
- **RANSAC** - RANSAC-based robust method
- **LMEDS** - Least-Median robust method
- **RHO** - PROSAC-based robust method

ransacReprojThreshold Maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC and RHO methods only). That is, if

$$\|\mathtt{dstPoints}_i - \mathtt{convertPointsHomogeneous}(\mathtt{H} \cdot \mathtt{srcPoints}_i)\|_2 > \mathtt{ransacReprojThreshold}$$

then the point i is considered as an outlier. If srcPoints and dstPoints are measured in pixels, it usually makes sense to set this parameter somewhere in the range of 1 to 10.

mask Optional output mask set by a robust method (RANSAC or LMeDS). Note that the input mask values are ignored.

perspectiveTransform()

```
void cv::perspectiveTransform ( InputArray  src,
                                OutputArray dst,
                                InputArray  m
                              )
```

Python:

```
cv.perspectiveTransform( src, m[, dst] ) -> dst
```

Performs the perspective matrix transformation of vectors.

The function **cv::perspectiveTransform** transforms every element of src by treating it as a 2D or 3D vector, in the following way:

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

Parameters

src input two-channel or three-channel floating-point array; each element is a 2D/3D vector to be transformed.

dst output array of the same size and type as src.

m 3x3 or 4x4 floating-point transformation matrix.