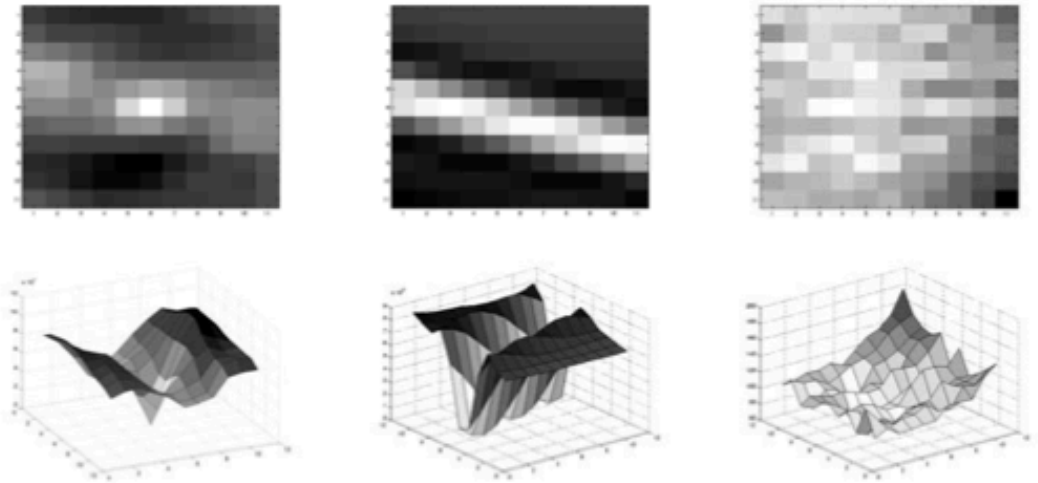
コンピュータビジョン
画像特徴

特徴点

- 位置を正確に決められる点
  - 画像の２次元相似・アフィン変換や究極的には撮影方向に不変

(a)



(b) (c) (d)

Figure 4.5 Three auto-correlation surfaces $E_{\mathrm{AC}}(\Delta \boldsymbol{u})$ shown as both grayscale images and surface plots: (a) The original image is marked with three red crosses to denote where the auto-correlation surfaces were computed; (b) this patch is from the flower bed (good unique minimum); (c) this patch is from the roof edge (one-dimensional aperture problem); and (d) this patch is from the cloud (no good peak). Each grid point in figures b–d is one value of $\Delta \boldsymbol{u}$.

[Szeliski2010]

特徴点

- 局所領域の位置決め精度の高さの尺度
  - u 方向の変動の大きさ

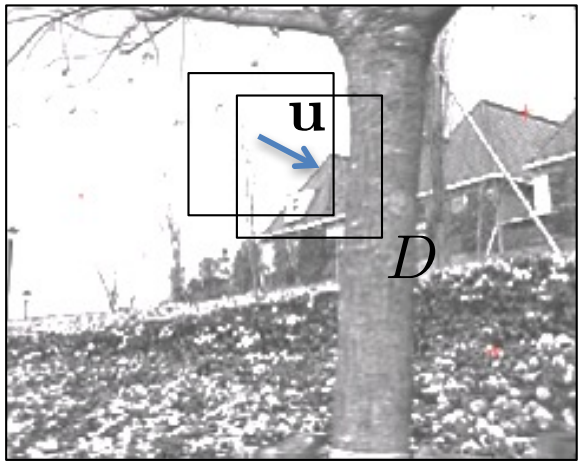
$$f(\mathbf{u}) = \sum_{\mathbf{x}\in D} [I(\mathbf{x}+\mathbf{u}) - I(\mathbf{x})]^2 \qquad \nabla I(\mathbf{x}) = \begin{bmatrix} I_x(\mathbf{x}) \\ I_y(\mathbf{x}) \end{bmatrix}$$

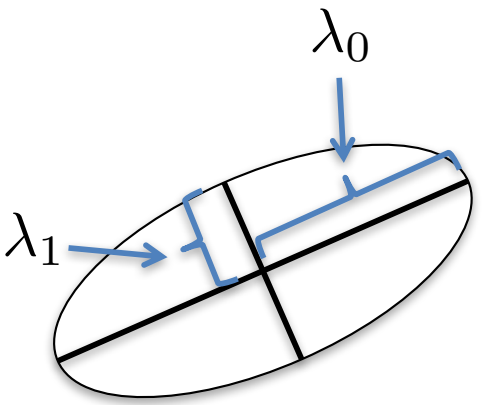
$$\approx \sum_{\mathbf{x}\in D} [I(\mathbf{x}) + \nabla I(\mathbf{x})^\top \mathbf{u} - I(\mathbf{x})]^2$$

$$= \sum_{\mathbf{u}\in D} \mathbf{u}^\top \nabla I(\mathbf{x}) \nabla I(\mathbf{x})^\top \mathbf{u}$$

$$= \mathbf{u}^\top \mathbf{A} \mathbf{u}$$

$$\mathbf{A} = \sum_{\mathbf{x}\in D} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \lambda_0 \mathbf{e}_0 \mathbf{e}_0^\top + \lambda_1 \mathbf{e}_1 \mathbf{e}_1^\top$$





Harris corner detector

- なるべく両方向に「尖って」いてほしい
  → 両固有値がともに大きくあって欲しい

Harris-Stephens(1988): $\det(\mathbf{A}) - \alpha \mathrm{trace}(\mathbf{A})^2 = \lambda_0\lambda_1 - \alpha(\lambda_0 + \lambda_1)^2$

Brown-Szeliski-Winder(2005): $\frac{\det(\mathbf{A})}{\mathrm{trace}(\mathbf{A})} = \frac{\lambda_0\lambda_1}{\lambda_0 + \lambda_1}$

harris.py

```
while 1:
    stat, image = cap.read(0)

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    gray = numpy.float32(gray)
    resp = cv2.cornerHarris(gray,2,3,0.04)
    resp = cv2.dilate(resp, None)
    image[resp > resp.max()*0.005] = [0,255,0]

    cv2.imshow('Camera', image)
    key = cv2.waitKey(10)
```

SIFT (Scale Invariant Feature Transform)

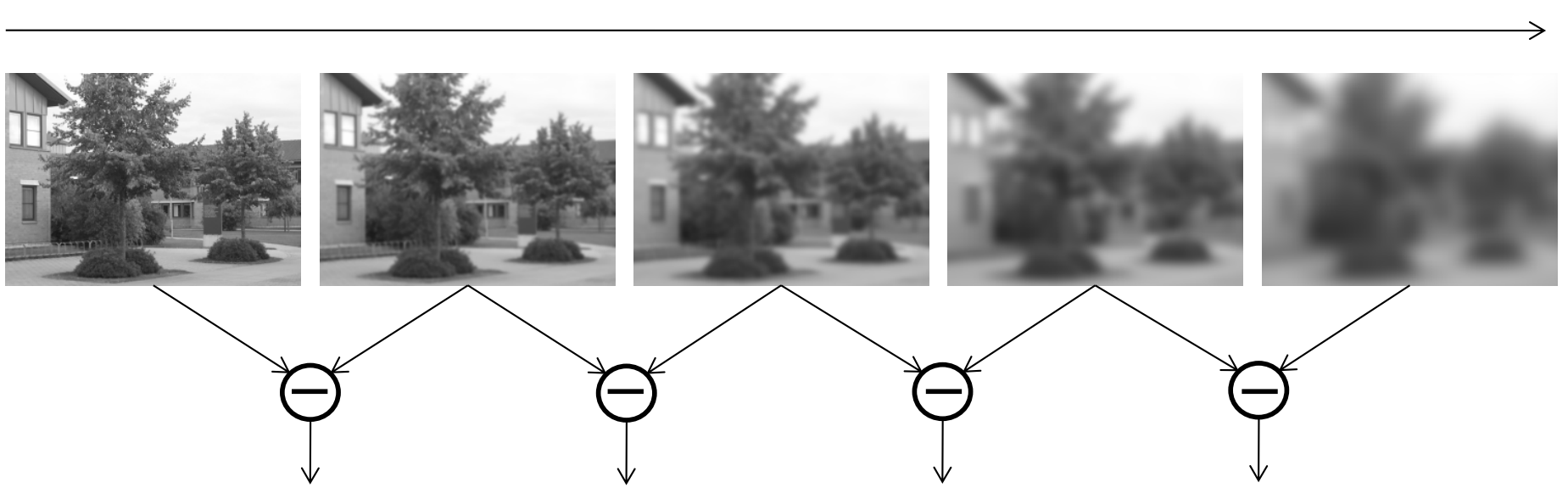
Lowe, Object recognition from local scale-invariant features, ICCV99

- 目的：同じ物体を色々な向き・大きさで撮影したとき，物体表面の特徴点を対応づけたい

- Key point (＝特徴点) detector
  - スケールと回転変動に不変な位置決め
- Descriptor at a key point
  - スケールと回転変動に不変な「特徴量」

SIFT: Keypoints

- キーポイント＝ スケールスペースでの DoG の極点
  - DoG: Difference of Gaussian
  - スケールの自動決定＝スケール不変性の獲得
- スケールスペース
  - ガウスフィルタを入力画像に適用し，画像を得る
  - ガウスフィルタのスケールを変化させたときの画像列



DoG　$f(x, y, s)$

SIFT: Descriptors

- 主方向の決定
  - 36方向のヒストグラムのピーク方向を主方向に選ぶ
  - Rotation不変性の獲得
- 主スケールと主方向で正規化した矩形内で勾配方向ヒストグラム
  - 位置，方向&大きさ，128ビン（8方向 x 4 x 4）

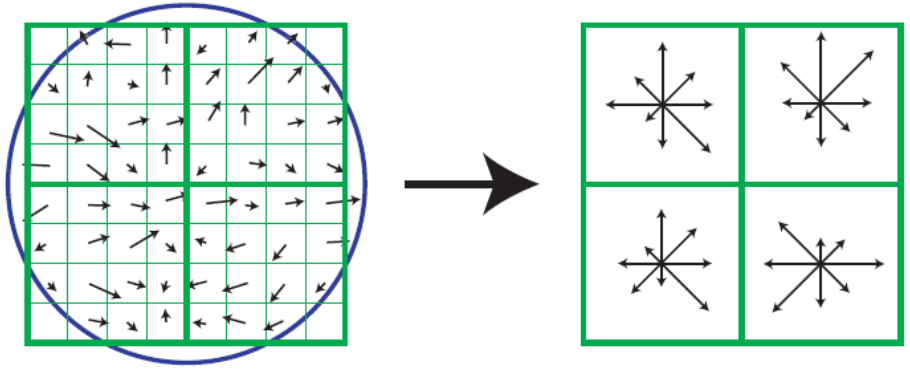


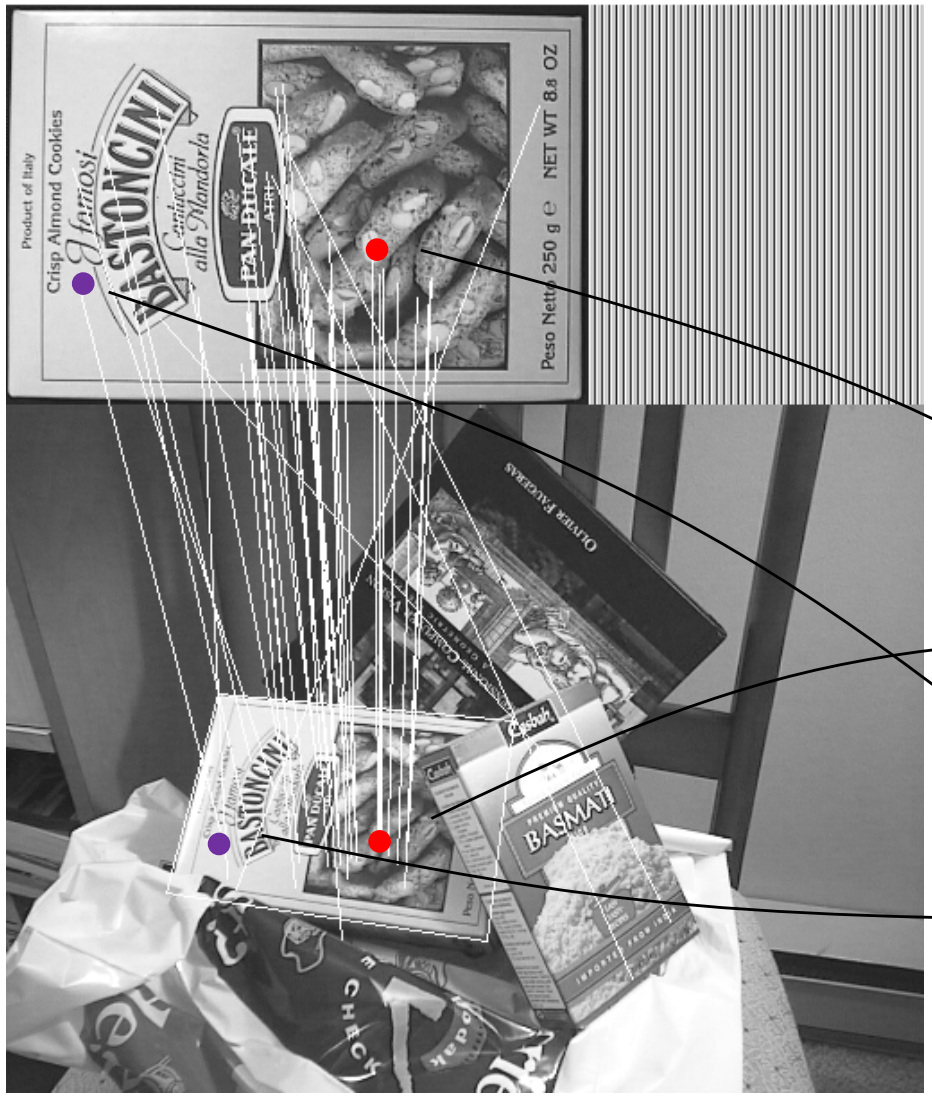
Figure 7: A keypoint descriptor is created by first computing the gradient magnitude and orientation at each image sample point in a region around the keypoint location, as shown on the left. These are weighted by a Gaussian window, indicated by the overlaid circle. These samples are then accumulated into orientation histograms summarizing the contents over 4x4 subregions, as shown on the right, with the length of each arrow corresponding to the sum of the gradient magnitudes near that direction within the region. This figure shows a 2x2 descriptor array computed from an 8x8 set of samples, whereas the experiments in this paper use 4x4 descriptors computed from a 16x16 sample array.

SIFT: コード

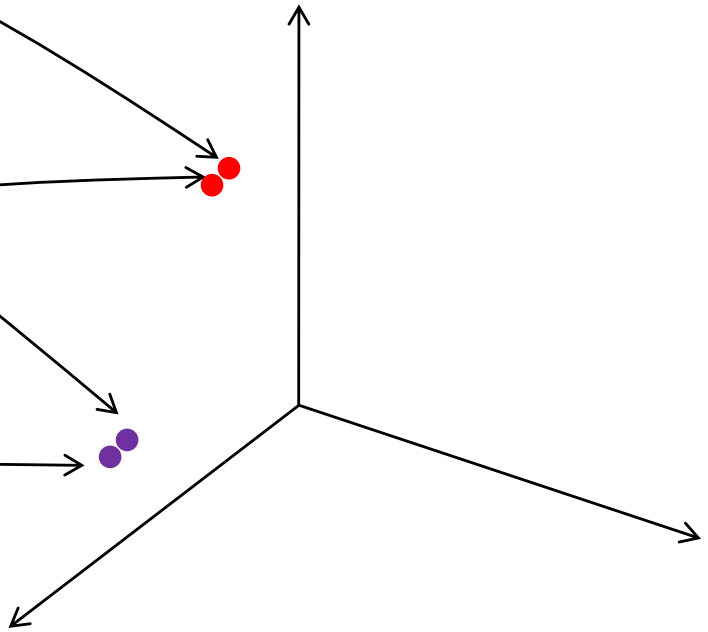
sift.py

```
def drawKeypoints(image, kp, color):
    for p in kp:
        pos = (int(p.pt[0]), int(p.pt[1]))
        pos1 = (int(p.pt[0]+p.size*math.cos(p.angle/180*math.pi)),¥
                int(p.pt[1]+p.size*math.sin(p.angle/180*math.pi)))
        scale = int(p.size)
        cv2.circle(image, pos, scale, color=(0,0,255), thickness=1)
        cv2.line(image, pos, pos1, color=(0,0,255), thickness=2)

sift = cv2.xfeatures2d.SIFT_create(500) # Num of keypoints
#sift = cv2.ORB_create(100)

while 1:
    stat, image = cap.read(0)
    image = cv2.resize(image,
((int)(image.shape[1]/2),(int)(image.shape[0]/2)))

    kp = sift.detect(image)
    kp, des = sift.compute(image, kp)
    drawKeypoints(image, kp, color=(255,0,0))

    cv2.imshow('Camera', image)
    key = cv2.waitKey(10)

    if key == 0x1b: # ESC
        break
```

局所特徴対応付け：最近傍探索



- Descriptorどうしの距離を比較
- Keypoints, Descriptorsともにスケール・回転に不変



descriptorの空間（128D）

局所特徴対応付け：コード(1/2)

sift_bfmatch.py

```
max_pts = 300
sift = cv2.xfeatures2d.SIFT_create(max_pts)

bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True) # Bruteforce matcher
desc0 = None # To store descriptors
color = numpy.random.randint(0, 255, (max_pts, 3)) # Random color for kp

while 1:
    stat, image = cap.read(0)
    image = cv2.resize(image, ((int)(image.shape[1]/2),
(int)(image.shape[0]/2)))

    # Find keypoints and their descriptors
    kp = sift.detect(image)
    kp, desc = sift.compute(image, kp)
    cv2.drawKeypoints(image, kp, image, color=(0,255,0))

    if desc0 != None:
         ...

    cv2.imshow('Camera', image)
    key = cv2.waitKey(10)

    if key == 0x1b: # ESC
        break
    elif key == 0x20:
        desc0 = desc.copy()
        kp0 = kp[:]
        print ('Target updated!')
```

局所特徴対応付け：コード(2/2)

sift_bfmatch.py

```
if desc0 != None:
    # Match the saved and the latest kp's
    matches = bf.match(desc0, desc)

    for i in range(min(max_pts,len(matches))):
        # Matched pair: kp0[m.queryIdx] <--> kp[m.trainIdx]
        m = matches[i]
        #if m.distance > 10.0: # thresholding
        #    continue
        (a, b) = kp[m.trainIdx].pt
        (c, d) = kp0[m.queryIdx].pt
        cv2.circle(image, (int(a), int(b)), 5,¥
                    color[m.queryIdx%max_pts].tolist(), -1)
        cv2.line(image, (int(a), int(b)), (int(c), int(d)),¥
                 color[m.queryIdx%max_pts].tolist(), 2)
```

誤対応への対策

1. descriptor距離に閾値を設定（コード：下）
2. 双方向対応：BFMatcher(crossCheck=True)
3. Ratio-test
   - (1番目に近い点までの距離) / (2番目に近い点までの距離) を閾値処理
4. 画像間幾何を利用したRANSAC
   - エピポーラ幾何や２次元射影変換

sift_bfmatch.py

```
for i in range(min(max_pts,len(matches))):
            # Matched pair: kp0[m.queryIdx] <--> kp[m.trainIdx]
            m = matches[i]
            if m.distance > 0.3: # thresholding
               continue
            (a, b) = kp[m.trainIdx].pt
            (c, d) = kp0[m.queryIdx].pt
```

Ratio test：コード

(1番目に近い点までの距離) / (2番目に近い点までの距離) > α

sift_bfrtest.py

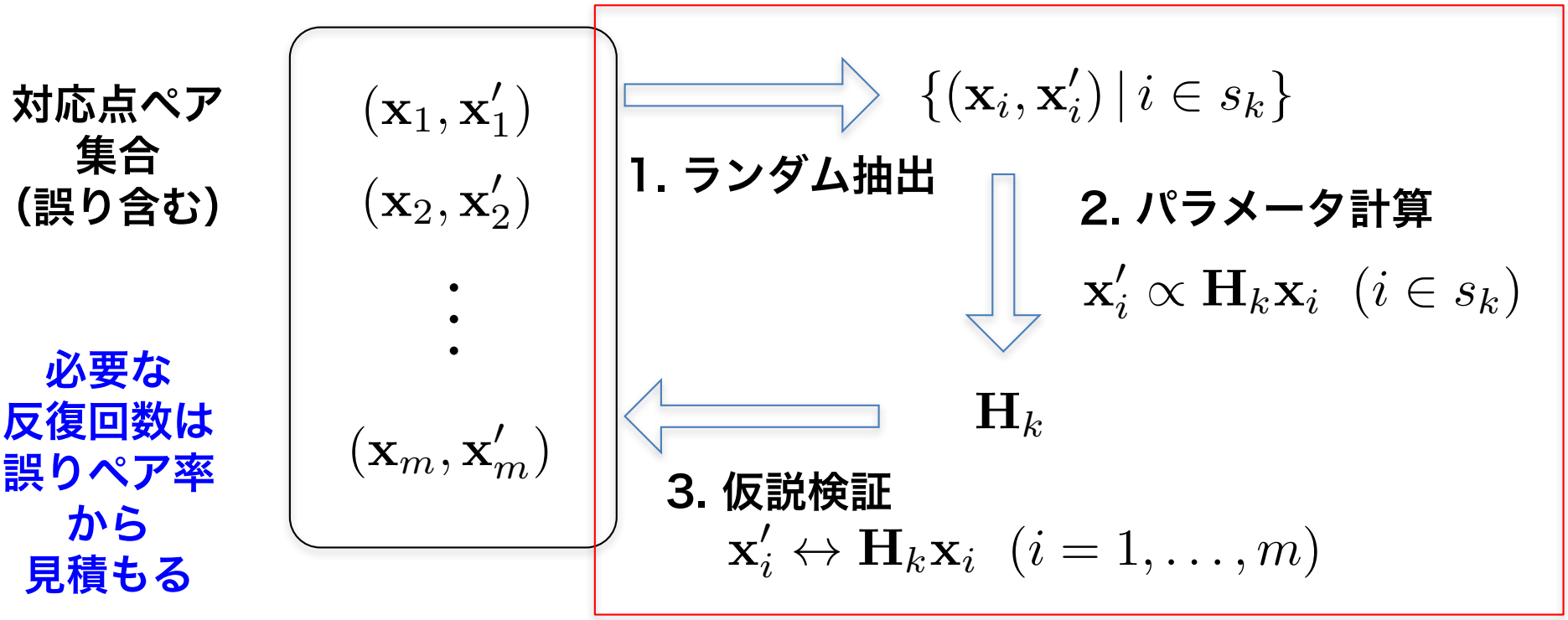
```
    if desc0 != None:
        # Match the saved and the latest kp's
        matches = bf.knnMatch(desc0, desc, 2)

        for i in range(min(max_pts,len(matches))):
            # Matched pair: kp0[m.queryIdx] <--> kp[m.trainIdx]
            m = matches[i]
            if m[0].distance > m[1].distance*0.7:
                continue
            (a, b) = kp[m[0].trainIdx].pt
            (c, d) = kp0[m[0].queryIdx].pt
            cv2.circle(image, (int(a), int(b)), 5,¥
                       color[m[0].queryIdx%max_pts].tolist(), -1)
            cv2.line(image, (int(a), int(b)), (int(c), int(d)),¥
                     color[m[0].queryIdx%max_pts].tolist(), 2)
```

RANSAC

RANdom SAmple Consensus

- 次の３つの手順を何度も繰り返し，最も良いものを選ぶ
  1. 仮説生成：*n* 個の対応点ペアをランダムに抽出する
  2. パラメータ計算：抽出したペアから幾何学モデルのパラメータを計算する
  3. 仮説検証：計算したパラメータを全対応点ペアに当てはめ，当てはまりの良さ（誤差）を評価する

$k = 1, \ldots$



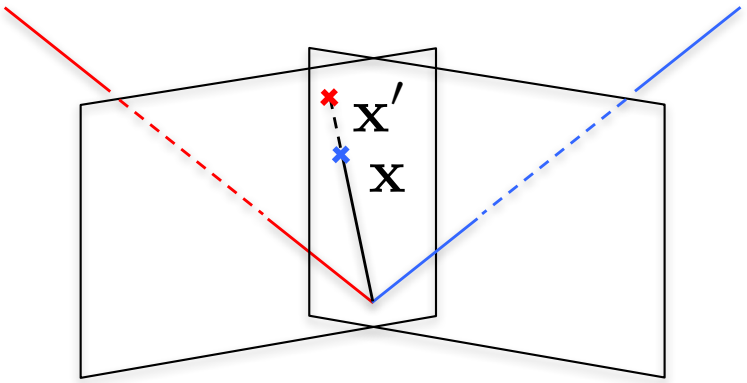
必要な
反復回数は
誤りペア率
から
見積もる

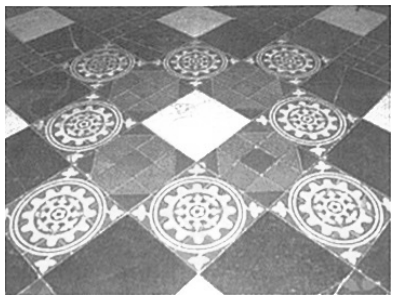
画像間幾何学モデル

- ２枚の画像間の点の幾何学的変換を記述するモデル

２次元射影変換
（その場回転・平面の画像）

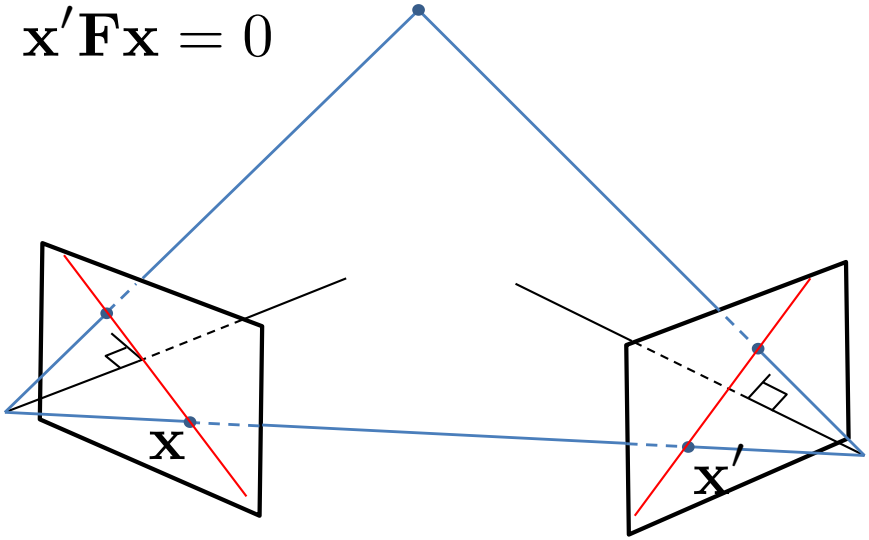
$$\mathbf{x}' \propto \mathbf{H}\mathbf{x}$$

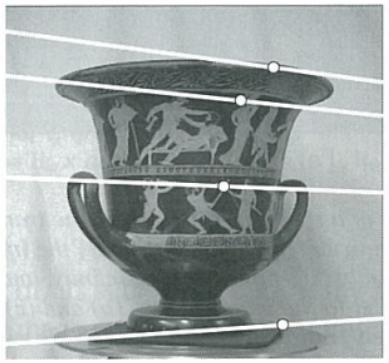


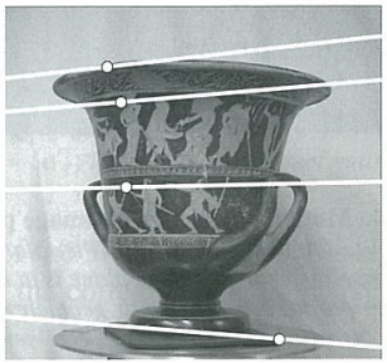


エピポーラ幾何
（Fundamental/Essential matrix）

$$\mathbf{x}'\mathbf{F}\mathbf{x} = 0$$







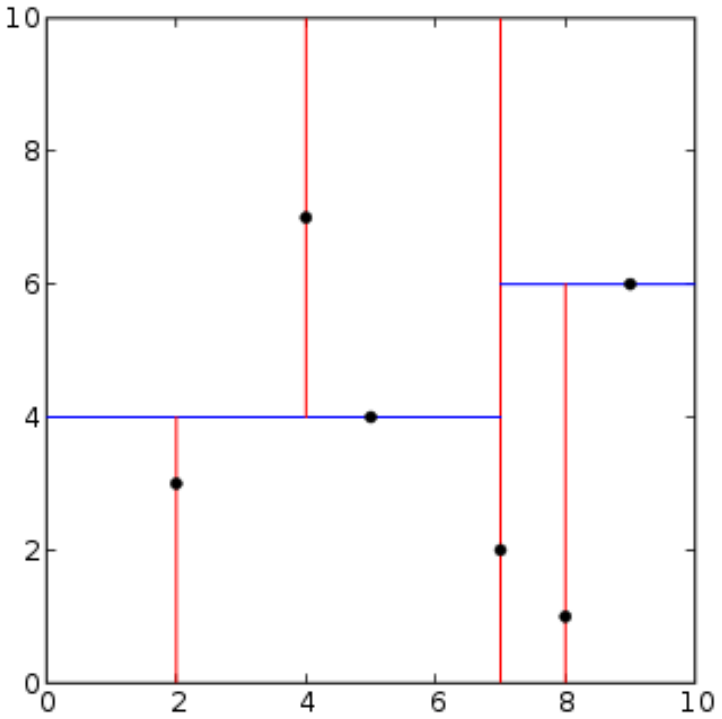
RANSAC＋2次元射影変換 → 外れ値除去

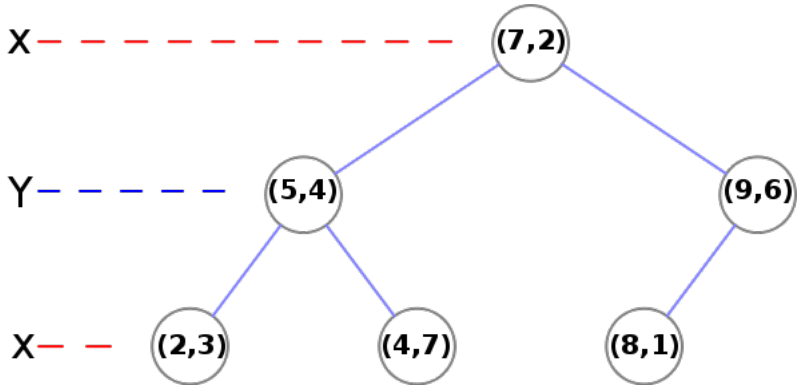
sift_bfransac.py

```
if desc0 != None:
        # Match the saved and the latest kp's
        matches = bf.match(desc0, desc)
        src_pts = numpy.float32([ kp0[m.queryIdx].pt ¥
                                for m in matches ]).reshape(-1,1,2)
        dst_pts = numpy.float32([ kp[m.trainIdx].pt ¥
                                for m in matches ]).reshape(-1,1,2)
        H, mask = cv2.findHomography(src_pts, dst_pts, ¥
                                cv2.RANSAC, 5.0)

        for i in range(min(max_pts,len(matches))):
            # Matched pair: kp0[m.queryIdx] <--> kp[m.trainIdx]
            m = matches[i]
            if mask[i] == 0:
                continue;
            (a, b) = kp[m.trainIdx].pt
            (c, d) = kp0[m.queryIdx].pt
            cv2.circle(image, (int(a), int(b)), 5,¥
                        color[m.queryIdx%max_pts].tolist(), -1)
            cv2.line(image, (int(a), int(b)), (int(c), int(d)),¥
                        color[m.queryIdx%max_pts].tolist(), 2)
```

高速な最近傍探索

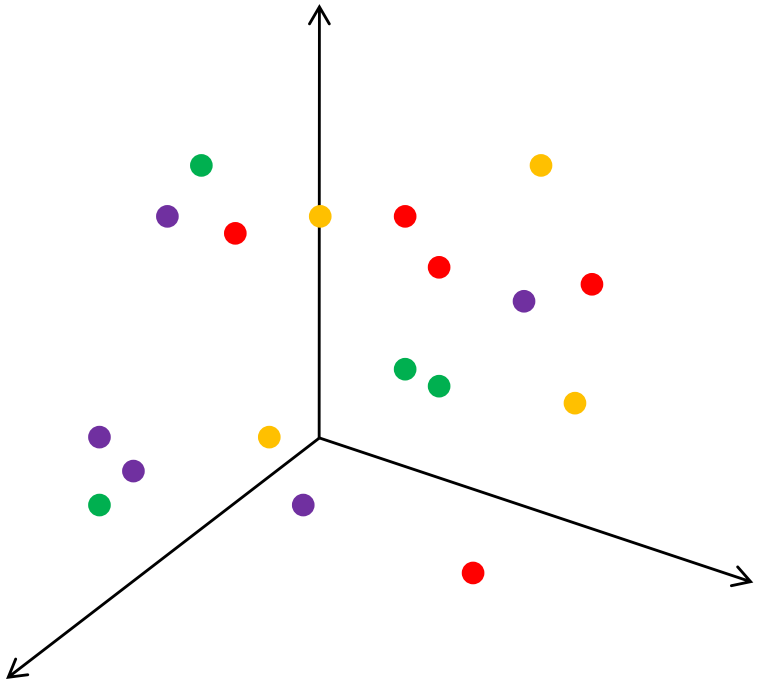
- 全数探索：最近傍探索のコスト＝O(N)
- kd木（kd-tree）：O(logN)
- 近似最近傍探索（ANN: Approximate Nearest Neighbor）





局所特徴を使った「特定物体認識」

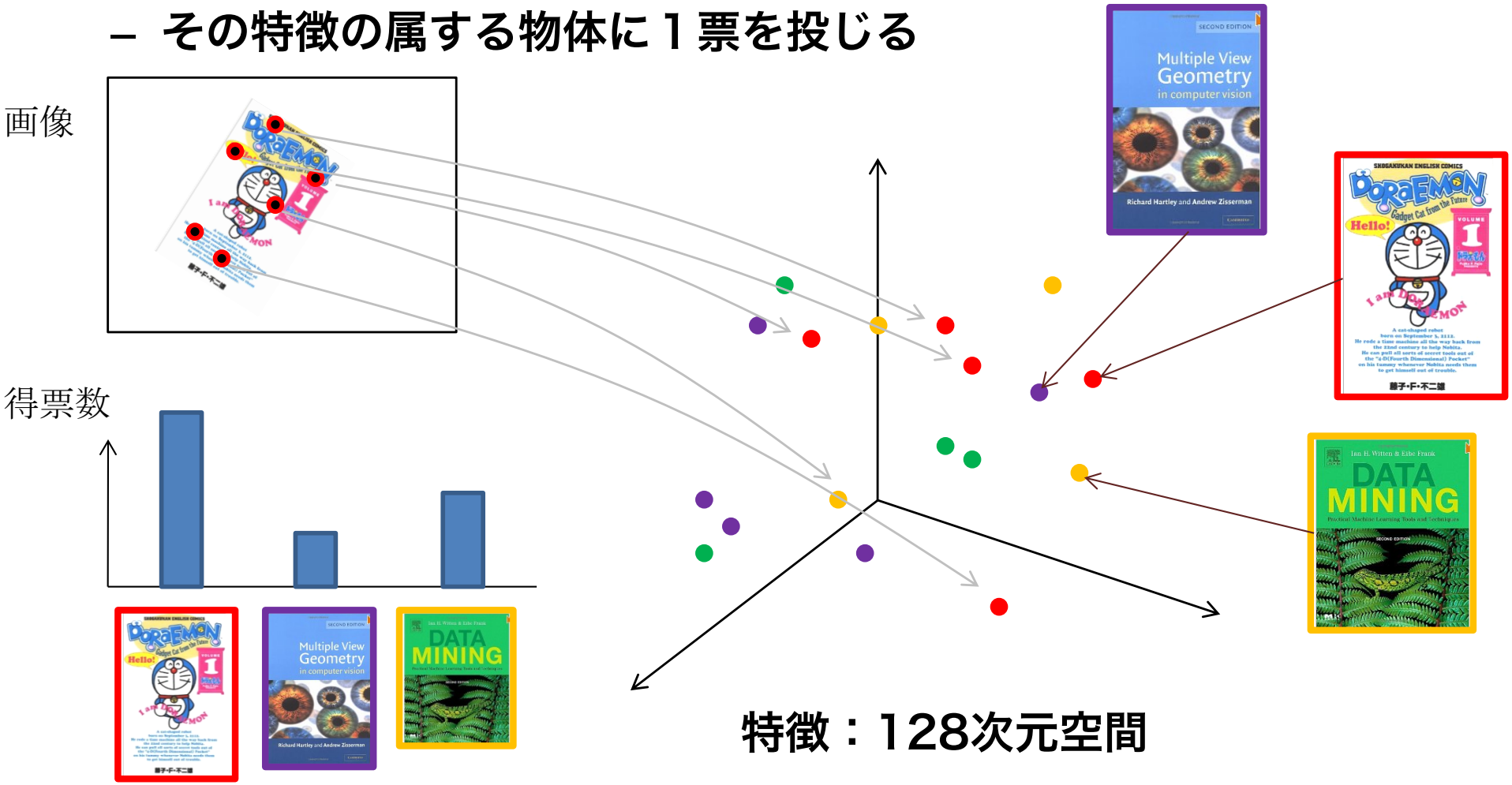
- 同一物体のキーポイントの特徴を特徴空間に記憶



特徴：128次元空間

局所特徴を使った「特定物体認識」

- 画像内の各キーポイントに対し，
  - 記憶した特徴のうち，その特徴と最も近いものを探す
  - その特徴の属する物体に１票を投じる



特定物体認識：コード

surf_recog.py

```
x, y, w, h = 320-150, 240-150, 300, 300
num_objs, desc = 0, None
kp_lbls, target_imgs = [], []

while 1:
    stat, image = cap.read(0)

    # Find keypoints and their descriptors
    kp1 = sift.detect(image, None)
    kp1, des1 = sift.compute(image, kp)
    disp = cv2.drawKeypoints(image, kp1, color=(0,255,0))
    cv2.rectangle(disp, (x, y), (x+w, y+h), 255, 2)

    if desc != None:
        matches = flann.knnMatch(desc1, desc, 2)
        #matches = bf.knnMatch(desc1, desc, 2)

        hist = [0] * num_objs
        for m in matches:
            if m[0].distance > m[1].distance*0.7:
                continue
            hist[kp_lbls[m[0].trainIdx]] += 1
```

特定物体認識：コード

surf_recog.py

```
        print 'histogram = ', hist
        pred = max(enumerate(hist),key=lambda x: x[1])[0]
        cv2.imshow('Reconized', target_imgs[pred]);

    cv2.imshow('Camera', disp)
    key = cv2.waitKey(50)
    if key == 0x1b: # ESC
        break
    elif key == 0x20:
        kp_lbls = kp_lbls + [num_objs]*len(desc1)
        if desc != None:
            desc = numpy.concatenate((desc, desc1))
            target_imgs += [image]
        else:
            desc = desc1.copy()
            target_imgs = [image]
        flann.add(desc)
        flann.train()
        num_objs = num_objs + 1
        print 'Target updated!'
```