7 Corner Detection

Corners are prominent structural elements in an image and are therefore useful in a wide variety of applications, including following objects across related images (*tracking*), determining the correspondence between stereo images, serving as reference points for precise geometrical measurements, and calibrating camera systems for machine vision applications. Thus corner points are not only important in human vision but they are also "robust" in the sense that they do not arise accidentally in 3D scenes and, furthermore, can be located quite reliably under a wide range of viewing angles and lighting conditions.

7.1 Points of Interest

Despite being easily recognized by our visual system, accurately and precisely detecting corners automatically is not a trivial task. A good corner detector must satisfy a number of criteria, including distinguishing between true and accidental corners, reliably detecting corners in the presence of realistic image noise, and precisely and accurately determining the locations of corners. Finally, it should also be possible to implement the detector efficiently enough so that it can be utilized in real-time applications such as video tracking.

Numerous methods for finding corners or similar interest points have been proposed and most of them take advantage of the following basic principle. While an *edge* is usually defined as a location in the image at which the gradient is especially high in *one* direction and low in the direction normal to it, a *corner point* is defined as a location that exhibits a strong gradient value in *multiple* directions at the same time.

Most methods take advantage of this observation by examining the first or second derivative of the image in the x and y directions to find corners (e.g., [77, 102, 137, 154]). In the next section, we describe in detail the Harris detector, also known as the "Plessey feature point detector" [102], since it turns out that even though more efficient

© Springer-Verlag London 2016
W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_7

detectors are known (see, e.g., [210, 220]), the Harris detector, and other detectors based on it, are the most widely used in practice.

7.2 Harris Corner Detector

This operator, developed by Harris and Stephens [102], is one of a group of related methods based on the same premise: a corner point exists where the gradient of the image is especially strong in more than one direction at the same time. In addition, locations along edges, where the gradient is strong in only one direction, should not be considered as corners, and the detector should be isotropic, that is, independent of the orientation of the local gradients.

7.2.1 Local Structure Matrix

The Harris corner detector is based on the first partial derivatives (gradient) of the image function $I(u,v)$, that is,

$$I_x(u,v) = \frac{\partial I}{\partial x}(u,v) \qquad \text{and} \qquad I_y(u,v) = \frac{\partial I}{\partial y}(u,v). \tag{7.1}$$

For each image position (u,v), we first calculate the three quantities

$$A(u,v) = I_x^2(u,v), \tag{7.2}$$
$$B(u,v) = I_y^2(u,v), \tag{7.3}$$
$$C(u,v) = I_x(u,v) \cdot I_y(u,v) \tag{7.4}$$

that constitute the elements of the *local structure matrix* $\mathbf{M}(u,v)$:[1]

$$\mathbf{M} = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}. \tag{7.5}$$

Next, each of the three scalar fields $A(u,v)$, $B(u,v)$, $C(u,v)$ is individually smoothed by convolution with a linear Gaussian filter $H^{G,\sigma}$ (see Sec. 5.2.7),

$$\bar{\mathbf{M}} = \begin{pmatrix} A * H_\sigma^G & C * H_\sigma^G \\ C * H_\sigma^G & B * H_\sigma^G \end{pmatrix} = \begin{pmatrix} \bar{A} & \bar{C} \\ \bar{C} & \bar{B} \end{pmatrix}. \tag{7.6}$$

The *eigenvalues*[2] of the matrix $\bar{\mathbf{M}}$, defined as[3]

$$\begin{aligned} \lambda_{1,2} &= \frac{\operatorname{trace}(\bar{\mathbf{M}})}{2} \pm \sqrt{\left(\frac{\operatorname{trace}(\bar{\mathbf{M}})}{2}\right)^2 - \det(\bar{\mathbf{M}})} \\ &= \frac{1}{2} \cdot \left(\bar{A} + \bar{B} \pm \sqrt{\bar{A}^2 - 2 \cdot \bar{A} \cdot \bar{B} + \bar{B}^2 + 4 \cdot \bar{C}^2}\right), \end{aligned} \tag{7.7}$$

[1] For improved legibility, we simplify the notation used in the following by omitting the function coordinates (u,v); for example, the function $I_x(u,v)$ is abbreviated as I_x or $A(u,v)$ is simply denoted A etc.

[2] See also Sec. B.4 in the Appendix.

[3] $\det(\bar{\mathbf{M}})$ denotes the *determinant* and $\operatorname{trace}(\bar{\mathbf{M}})$ denotes the *trace* of the matrix $\bar{\mathbf{M}}$ (see, e.g., [35, pp. 252 and 259]).

are (because the matrix is symmetric) positive and real. They contain essential information about the local image structure. Within an image region that is uniform (that is, appears flat), $\bar{\mathbf{M}} = 0$ and therefore $\lambda_1 = \lambda_2 = 0$. On an ideal ramp, however, the eigenvalues are $\lambda_1 > 0$ and $\lambda_2 = 0$, independent of the orientation of the edge. The eigenvalues thus encode an edge's *strength*, and their associated *eigenvectors* correspond to the local edge *orientation*.

A corner should have a strong edge in the main direction (corresponding to the larger of the two eigenvalues), another edge normal to the first (corresponding to the smaller eigenvalues), and both eigenvalues must be significant. Since $\bar{A}, \bar{B} \geq 0$, we can assume that $\operatorname{trace}(\bar{\mathbf{M}}) > 0$ and thus $|\lambda_1| \geq |\lambda_2|$. Therefore only the smaller of the two eigenvalues, $\lambda_2 = \operatorname{trace}(\bar{\mathbf{M}})/2 - \sqrt{\ldots}$, is relevant when determining a corner.

7.2.2 Corner Response Function (CRF)

From Eqn. (7.7) we see that the difference between the two eigenvalues of the local structure matrix is

$$\lambda_1 - \lambda_2 = 2 \cdot \sqrt{0.25 \cdot \big(\operatorname{trace}(\bar{\mathbf{M}})\big)^2 - \det(\bar{\mathbf{M}})}, \tag{7.8}$$

where the expression under the square root is always non-negative. At a good corner position, the difference between the two eigenvalues λ_1, λ_2 should be as small as possible and thus the expression under the root in Eqn. (7.8) should be a minimum. To avoid the explicit calculation of the eigenvalues (and the square root) the Harris detector defines the function

$$\begin{aligned} Q(u,v) &= \det(\bar{\mathbf{M}}(u,v)) - \alpha \cdot \big(\operatorname{trace}(\bar{\mathbf{M}}(u,v))\big)^2 \qquad (7.9) \\ &= \bar{A}(u,v) \cdot \bar{B}(u,v) - \bar{C}^2(u,v) - \alpha \cdot [\bar{A}(u,v) + \bar{B}(u,v)]^2 \end{aligned}$$

as a measure of "corner strength", where the parameter α determines the sensitivity of the detector. $Q(u,v)$ is called the "corner response function" and returns maximum values at isolated corners. In practice, α is assigned a fixed value in the range of 0.04 to 0.06 (max. $0.25 = \frac{1}{4}$). The larger the value of α, the less sensitive the detector is and the fewer corners detected.

7.2.3 Determining Corner Points

An image location (u,v) is selected as a potential candidate for a corner point if

$$Q(u,v) > \mathsf{t}_H,$$

where the threshold t_H is selected based on image content and typically lies within the range of 10,000 to 1,000,000. Once selected, the corners $\boldsymbol{c}_i = \langle u_i, v_i, q_i \rangle$ are inserted into the sequence

$$\mathcal{C} = (\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_N),$$

which is then sorted in descending order (i.e., $q_i \geq q_{i+1}$) according to *corner strength* $q_i = Q(u_i, v_i)$, as defined in Eqn. (7.9). To suppress

Table 7.1
Harris corner detector—typical parameter settings for Alg. 7.1.

Prefilter (Alg. 7.1, line 2–3): Smoothing with a small xy-separable filter $H_p = H_{px} * H_{py}$, where

$$H_{px} = \frac{1}{9} \cdot \begin{bmatrix} 2 & 5 & 2 \end{bmatrix} \qquad \text{and} \qquad H_{py} = H_{px}^{\intercal} = \frac{1}{9} \cdot \begin{bmatrix} 2 \\ 5 \\ 2 \end{bmatrix}.$$

Gradient filter (Alg. 7.1, line 3): Computing the first partial derivative in the x and y directions with

$$h_{dx} = \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} \qquad \text{and} \qquad h_{dy} = h_{dx}^{\intercal} = \begin{bmatrix} -0.5 \\ 0 \\ 0.5 \end{bmatrix}.$$

Blur filter (Alg. 7.1, line 10): Smoothing the individual components of the structure matrix M with separable Gaussian filters $H_b = H_{bx} * H_{by}$ with

$$h_{bx} = \frac{1}{64} \cdot \begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix} \quad \text{and} \quad h_{by} = h_{bx}^{\intercal} = \frac{1}{64} \cdot \begin{bmatrix} 1 \\ 6 \\ 15 \\ 20 \\ 15 \\ 6 \\ 1 \end{bmatrix}.$$

Control parameter (Alg. 7.1, line 14): $\alpha = 0.04, \ldots, 0.06$ (default 0.05).
Response threshold (Alg. 7.1, line 19): $\mathsf{t}_H = 10\,000, \ldots, 1\,000\,000$ (default 20 000).
Neighborhood radius (Alg. 7.1, line 37): $d_{\min} = 10$ Pixel.

the false corners that tend to arise in densely packed groups around true corners, all except the strongest corner in a specified vicinity are eliminated. To accomplish this, the list $\mathcal{C}$ is traversed from the front to the back, and the weaker corners toward the end of the list, which lie in the surrounding neighborhood of a stronger corner, are deleted.

The complete algorithm for the Harris detector is summarized again in Alg. 7.1; the associated parameters are listed in Table 7.1.

7.2.4 Examples

Figure 7.1 uses a simple synthetic image to illustrate the most important steps in corner detection using the Harris detector. The figure shows the result of the gradient computation, the three components of the structure matrix $\mathbf{M}(u, v) = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$, and the values of the *corner response function* $Q(u, v)$ for each image position (u, v). This example was calculated with the standard settings as given in Table 7.1.

The second example (Fig. 7.2) illustrates the detection of corner points in a grayscale representation of a natural scene. It demonstrates how weak corners are eliminated in favor of the strongest corner in a region.

```
1:  HarrisCorners($I, \alpha, \mathsf{t}_H, d_{\min}$)
        Input: $I$, the source image; $\alpha$, sensitivity parameter (typ. 0.05);
        $\mathsf{t}_H$, response threshold (typ. 20 000); $d_{\min}$, minimum distance
        between final corners. Returns a sequence of the strongest corners
        detected in $I$.

        Step 1 – calculate the corner response function:
2:      $I_x \leftarrow (I * h_{px}) * h_{dx}$          ▷ horizontal prefilter and derivative
3:      $I_y \leftarrow (I * h_{py}) * h_{dy}$          ▷ vertical prefilter and derivative

4:      $(M, N) \leftarrow \mathsf{Size}(I)$
5:      Create maps $A, B, C, Q \colon M \times N \mapsto \mathbb{R}$

6:      for all image coordinates $(u, v)$ do
            Compute the local structure matrix $\mathbf{M} = \left(\begin{smallmatrix} A & C \\ C & B \end{smallmatrix}\right)$:
7:          $A(u, v) \leftarrow (I_x(u, v))^2$
8:          $B(u, v) \leftarrow (I_y(u, v))^2$
9:          $C(u, v) \leftarrow I_x(u, v) \cdot I_y(u, v)$

        Blur the components of the local structure matrix ($\bar{\mathbf{M}}$):
10:     $\bar{A} \leftarrow A * H_b$
11:     $\bar{B} \leftarrow B * H_b$
12:     $\bar{C} \leftarrow C * H_b$

13:     for all image coordinates $(u, v)$ do          ▷ calc. corner response:
14:         $Q(u, v) \leftarrow \bar{A}(u, v) \cdot \bar{B}(u, v) - \bar{C}^2(u, v) - \alpha \cdot [\bar{A}(u, v) + \bar{B}(u, v)]^2$

        Step 2 – collect the corner points:
15:     $\mathcal{C} \leftarrow (\,)$                    ▷ start with an empty corner sequence
16:     for all image coordinates $(u, v)$ do
17:         if $Q(u, v) > \mathsf{t}_H \wedge \mathsf{IsLocalMax}(Q, u, v)$ then
18:             $\boldsymbol{c} \leftarrow \langle u, v, Q(u, v) \rangle$          ▷ create a new corner $\boldsymbol{c}$
19:             $\mathcal{C} \leftarrow \mathcal{C} \smile (\boldsymbol{c})$          ▷ add $\boldsymbol{c}$ to corner sequence $\mathcal{C}$

20:     $\mathcal{C}_{\text{clean}} \leftarrow \mathsf{CleanUpCorners}(\mathcal{C}, d_{\min})$

21:     return $\mathcal{C}_{\text{clean}}$

22: IsLocalMax($Q, u, v$)          ▷ determine if $Q(u, v)$ is a local maximum
23:     $\mathcal{N} \leftarrow \mathsf{GetNeighbors}(Q, u, v)$          ▷ se below
24:     return $Q(u, v) > \max(\mathcal{N})$          ▷ true or false

25: GetNeighbors($Q, u, v$)
        Returns the 8 neighboring values around $Q(u, v)$.
26:     $\mathcal{N} \leftarrow (Q(u{+}1, v), Q(u{+}1, v{-}1), Q(u, v{-}1), Q(u-1, v{-}1),$
                 $Q(u{-}1, v), Q(u{-}1, v{+}1), Q(u, v{+}1), Q(u{+}1, v{+}1))$
27:     return $\mathcal{N}$

28: CleanUpCorners($\mathcal{C}, d_{\min}$)
29:     $\mathsf{Sort}(\mathcal{C})$          ▷ sort $\mathcal{C}$ by desc. $q_i$ (strongest corners first)
30:     $\mathcal{C}_{\text{clean}} \leftarrow (\,)$          ▷ empty "clean" corner sequence
31:     while $\mathcal{C}$ is not empty do
32:         $\boldsymbol{c}_0 \leftarrow \mathsf{GetFirst}(\mathcal{C})$          ▷ get the strongest corner from $\mathcal{C}$
33:         $\mathcal{C} \leftarrow \mathsf{Delete}(\boldsymbol{c}_0, \mathcal{C})$          ▷ the 1st element is removed from $\mathcal{C}$
34:         $\mathcal{C}_{\text{clean}} \leftarrow \mathcal{C}_{\text{clean}} \smile (\boldsymbol{c}_0)$          ▷ add $\boldsymbol{c}_0$ to $\mathcal{C}_{\text{clean}}$
35:         for all $\boldsymbol{c}_j$ in $\mathcal{C}$ do
36:             if $\mathsf{Dist}(\boldsymbol{c}_0, \boldsymbol{c}_j) < d_{\min}$ then
37:                 $\mathcal{C} \leftarrow \mathsf{Delete}(\boldsymbol{c}_j, \mathcal{C})$          ▷ remove element $\boldsymbol{c}_j$ from $\mathcal{C}$
38:     return $\mathcal{C}_{\text{clean}}$
```

Alg. 7.1
Harris corner detector. This algorithm takes an intensity image I and creates a sorted list of detected corner points. $*$ is the convolution operator used for linear filter operations. Details for the parameters H_p, H_{dx}, H_{dy}, H_b, α, and t_H can be found in Table 7.1.

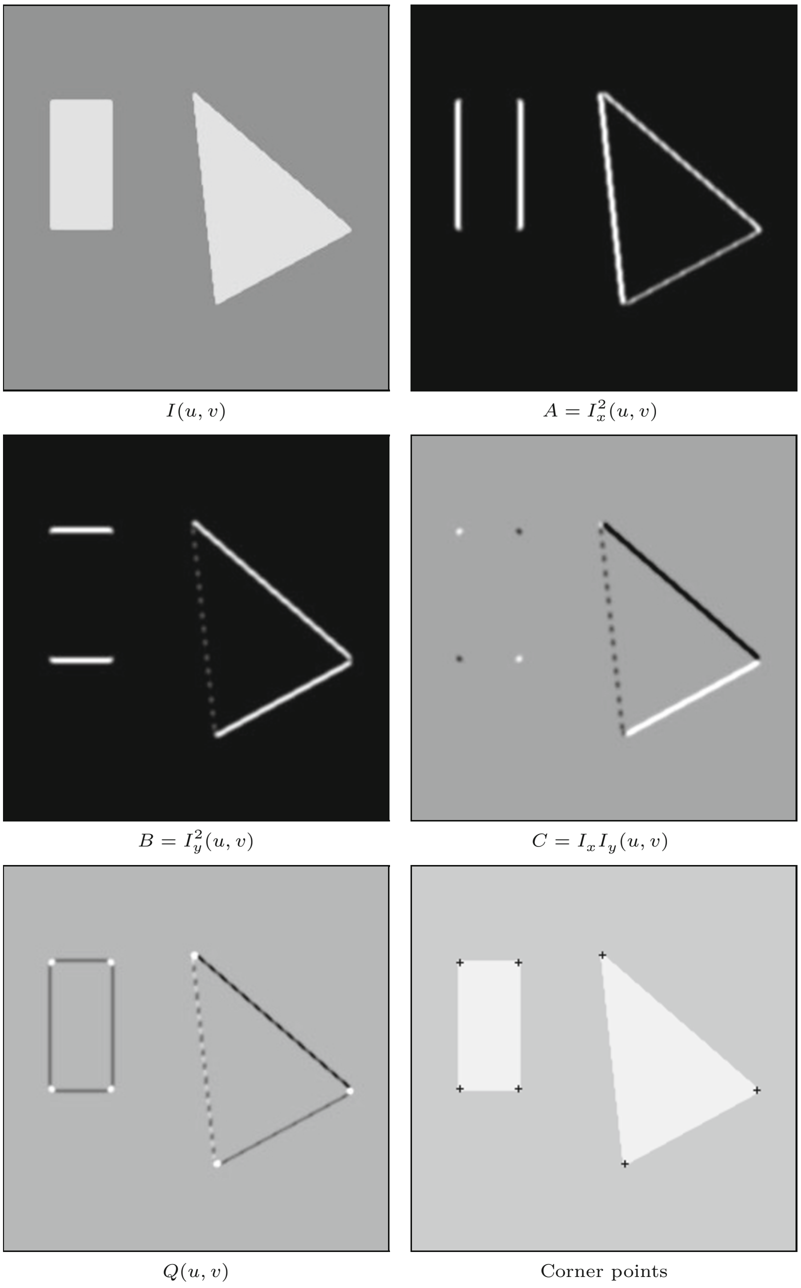
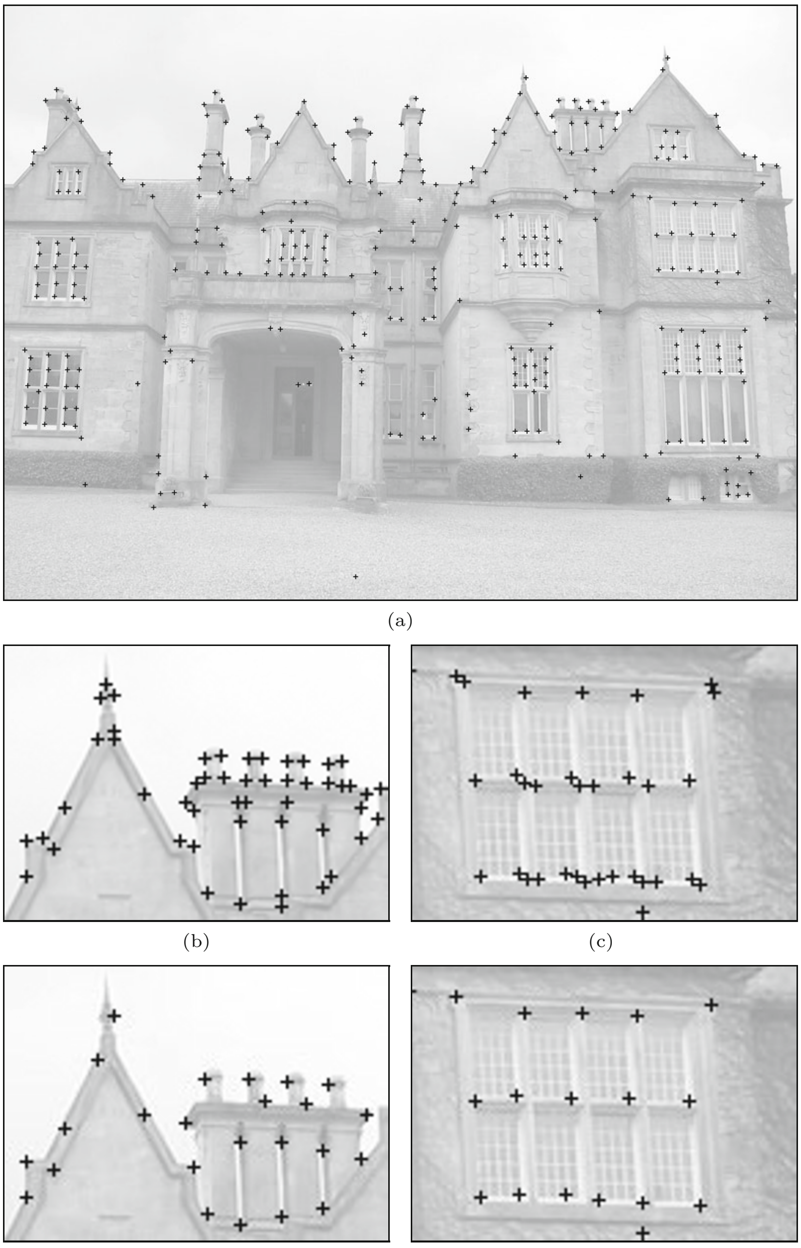


Fig. 7.1
Harris corner detector—Example 1. Starting with the original image $I(u, v)$, the first derivative is computed, and then from it the components of the structure matrix $\mathbf{M}(u, v)$, with $A(u, v) = I_x^2(u, v)$, $B = I_y^2(u, v)$, $C = I_x(u, v) \cdot I_y(u, v)$. $A(u, v)$ and $B(u, v)$ represent, respectively, the strength of the horizontal and vertical edges. In $C(u, v)$, the values are strongly positive (white) or strongly negative (black) only where the edges are strong in both directions (null values are shown in gray). The corner response function, $Q(u, v)$, exhibits noticeable positive peaks at the corner positions.

7.3 Implementation

Since the Harris detector algorithm is more complex than the algorithms we presented earlier, in the following sections we explain its implementation in greater detail. While reading the following you may wish to refer to the complete source code for the class `HarrisCornerDetector`, which is available online as part of the `imagingbook` library.[4]

[4] Package `imagingbook.pub.corners`.



(a) (b) (c) (d) (e)

Fig. 7.2
Harris corner detector—Example 2. A complete result with the final corner points marked (a). After selecting the strongest corner points within a 10-pixel radius, only 335 of the original 615 candidate corners remain. Details *before* (b, c) and *after* selection (d, e).

7.3.1 Step 1: Calculating the Corner Response Function

To handle the range of the positive and negative values generated by the filters used in this step, we will need to use floating-point images to store the intermediate results, which also assures sufficient range and precision for small values. The kernels of the required filters, that is, the presmoothing filter H_p, the gradient filters H_{dx}, H_{dy}, and the smoothing filter for the structure matrix H_b, are defined as 1D `float` arrays:

```
float[] hp = {2f/9, 5f/9, 2f/9};
```

```
float[] hd = {0.5f, 0, -0.5f};
float[] hb =
    {1f/64, 6f/64, 15f/64, 20f/64, 15f/64, 6f/64, 1f/64};
```

From the original 8-bit image (of type `ByteProcessor`), we first create two copies, `Ix` and `Iy`, of type `FloatProcessor`:

```
  FloatProcessor Ix = I.convertToFloatProcessor();
  FloatProcessor Iy = I.convertToFloatProcessor();
```

The first processing step is to presmooth the image with the 1D filter kernel `hp` ($= h_{px} = h_{py}^{\mathsf{T}}$, see Alg. 7.1, line 2). Subsequently the 1D gradient filter `hd` ($= h_{dx} = h_{dy}^{\mathsf{T}}$) is used to calculate the horizontal and vertical derivatives (see Alg. 7.1, line 3). To perform the convolution with the corresponding 1D kernels we use the (static) methods `convolveX()` and `convolveY()` defined in class `Filter`:[5]

```
  Filter.convolveX(Ix, hp);        // $I_x \leftarrow I_x * h_{px}$
  Filter.convolveX(Ix, hd);        // $I_x \leftarrow I_x * h_{dx}$
  Filter.convolveY(Iy, hp);        // $I_y \leftarrow I_y * h_{py}$
  Filter.convolveY(Iy, hd);        // $I_y \leftarrow I_y * h_{dy}$
```

Now the components $A(u,v)$, $B(u,v)$, $C(u,v)$ of the structure matrix $\mathbf{M}$ are calculated for all image positions (u,v):

```
  A = ImageMath.sqr(Ix);           // $A(u,v) \leftarrow I_x^2(u,v)$
  B = ImageMath.sqr(Iy);           // $B(u,v) \leftarrow I_y^2(u,v)$
  C = ImageMath.mult(Ix, Iy);      // $C(u,v) \leftarrow I_x(u,v) \cdot I_y(u,v)$

```

The components of the structure matrix are then smoothed with a separable filter kernel $H_b = h_{bx} * h_{by}$:

```
  Filter.convolveXY(A, hb);        // $A \leftarrow (A * h_{bx}) * h_{by}$
  Filter.convolveXY(B, hb);        // $B \leftarrow (B * h_{bx}) * h_{by}$
  Filter.convolveXY(C, hb);        // $C \leftarrow (C * h_{bx}) * h_{by}$
```

The variables `A`, `B`, `C` of type `FloatProcessor` are declared in the class `HarrisCornerDetector`. `sqr()` and `mult()` are static methods of class `ImageMath` for squaring an image and multiplying two images, respectively. The method `convolveXY(I, h)` is used to apply a x/y-separable 2D convolution with the 1D kernel `h` to the image `I`.

Finally, the corner response function (Alg. 7.1, line 14) is calculated by the method `makeCrf()` and a new image (of type `FloatProcessor`) is created:

```
private FloatProcessor  makeCrf(float alpha) {
  FloatProcessor Q = new FloatProcessor(M, N);
  final float[] pA = (float[]) A.getPixels();
  final float[] pB = (float[]) B.getPixels();
  final float[] pC = (float[]) C.getPixels();
  final float[] pQ = (float[]) Q.getPixels();
  for (int i = 0; i < M * N; i++) {
    float a = pA[i], b = pB[i], c = pC[i];
    float det = a * b - c * c;   // $\det(\bar{\mathbf{M}})$
    float trace = a + b;         // $\operatorname{trace}(\bar{\mathbf{M}})$
    pQ[i] = det - alpha * (trace * trace);
```

[5] Package `imagingbook.lib.image`.

```
29   }
30   return Q;
31 }
```

7.3.2 Step 2: Selecting "Good" Corner Points

The result of the first stage of Alg. 7.1 is the corner response function $Q(u, v)$, which in our implementation is stored as a floating-point image (`FloatProcessor`). In the second stage, the dominant corner points are selected from Q. For this we need (a) an object type to describe the corners and (b) a flexible container, in which to store these objects. In this case, the container should be a dynamic data structure since the number of objects to be stored is not known beforehand.

The `Corner` class

Next we define a new class `Corner`[6] to represent individual corner points $\boldsymbol{c} = \langle x, y, q \rangle$ and a single constructor (in line 35) with `float` parameters x, y for the position and corner strength q:

```
32 public class Corner implements Comparable<Corner> {
33   final float x, y, q;
34
35   public Corner (float x, float y, float q) {
36     this.x = x;
37     this.y = y;
38     this.q = q;
39   }
40
41   public int compareTo (Corner c2) {
42     if (this.q > c2.q) return -1;
43     if (this.q < c2.q) return 1;
44     else return 0;
45   }
46   ...
47 }
```

The class `Corner` implements Java's `Comparable` interface, such that objects of type `Corner` can be compared with each other and thereby sorted into an ordered sequence. The `compareTo()` method required by the `Comparable` interface is defined (in line 41) to sort corners by descending `q` values.

Choosing a suitable container

In Alg. 7.1, we used the notion of a *sequence* or *lists* to organize and manipulate the collections of potential corner points generated at various stages. One solution would be to utilize *arrays*, but since the size of arrays must be declared before they are used, we would have to allocate memory for extremely large arrays in order to store all the possible corner points that might be identified. Instead, we make use of the `ArrayList` class, which is one of many dynamic data structures conveniently provided by Java's *Collections Framework*.[7]

[6] Package `imagingbook.pub.corners`.

[7] Package `java.util`.

The `collectCorners()` method

The method `collectCorners()` outlined here selects the dominant corner points from the corner response function $Q(u, v)$. The parameter *border* specifies the width of the image's border, within which corner points should be ignored.

```java
48 List<Corner> collectCorners(FloatProcessor Q, float tH, int
        border) {
49   List<Corner> C = new ArrayList<Corner>();
50   for (int v = border; v < N - border; v++) {
51     for (int u = border; u < M - border; u++) {
52       float q = Q.getf(u, v);
53       if (q > tH && isLocalMax(Q, u, v)) {
54         Corner c = new Corner(u, v, q);
55         C.add(c);
56       }
57     }
58   }
59   return C;
60 }
```

First (in line 49), a new instance of `ArrayList`[8] is created and assigned to the variable `C`. Then the CRF image `Q` is traversed, and when a potential corner point is located, a new `Corner` is instantiated (line 54) and added to `C` (line 55). The Boolean method `isLocalMax()` (defined in class `HarrisCornerDetector`) determines if the 2D function `Q` is a local maximum at the given position `u`, `v`:

```java
61 boolean isLocalMax (FloatProcessor Q, int u, int v) {
62   if (u <= 0 || u >= M - 1 || v <= 0 || v >= N - 1) {
63     return false;
64   }
65   else {
66     float[] q = (float[]) Q.getPixels();
67     int i0 = (v - 1) * M + u;
68     int i1 = v * M + u;
69     int i2 = (v + 1) * M + u;
70     float q0 = q[i1];
71     return   // check 8 neighbors of q0:
72       q0 >= q[i0 - 1] && q0 >= q[i0] && q0 >= q[i0 + 1] &&
73       q0 >= q[i1 - 1] &&                    q0 >= q[i1 + 1]
        &&
74       q0 >= q[i2 - 1] && q0 >= q[i2] && q0 >= q[i2 + 1] ;
75   }
76 }
```

7.3.3 Step 3: Cleaning up

The final step is to remove the weakest corners in a limited area where the size of this area is specified by the radius $d_{\min}$ (Alg. 7.1, lines 29–38). This process is outlined in Fig. 7.3 and implemented by the following method `cleanupCorners()`.

[8] The specification `ArrayList<Corner>` indicates that the list `C` may only contain objects of type `Corner`.

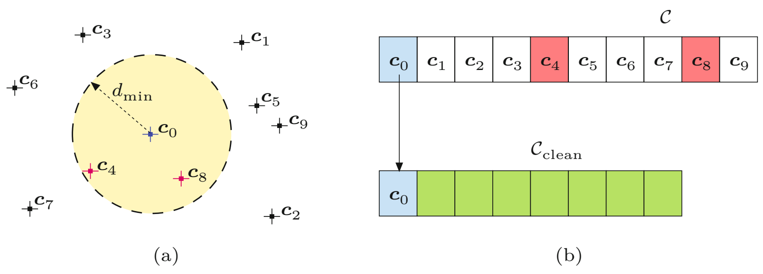

c_3
c_1
c_6
$d_{\min}$
c_0
c_5
c_9
c_4
c_8
c_7
c_2
(a)
$\mathcal{C}$
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9
$\mathcal{C}_{\text{clean}}$
c_0
(b)

Fig. 7.3
Selecting the strongest corners within a given spatial distance. (a) Sample corner positions in the 2D plane. (b) The original list of corners ($\mathcal{C}$) is sorted by "corner strength" (q) in descending order; that is, $\boldsymbol{c}_0$ is the strongest corner. First, corner $\boldsymbol{c}_0$ is added to a new list $\mathcal{C}_{\text{clean}}$, while the weaker corners $\boldsymbol{c}_4$ and $\boldsymbol{c}_8$ (which are both within distance $d_{\min}$ from $\boldsymbol{c}_0$) are removed from the original list $\mathcal{C}$. The following corners $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots$ are treated similarly until no more elements remain in $\mathcal{C}$. None of the corners in the resulting list $\mathcal{C}_{\text{clean}}$ is closer to another corner than $d_{\min}$.

```java
77 List<Corner> cleanupCorners(List<Corner> C, double dmin) {
78   double dmin2 = dmin * dmin;
79   // sort corners by descending q-value:
80   Collections.sort(C);
81   // we use an array of corners for efficiency reasons:
82   Corner[] Ca = C.toArray(new Corner[C.size()]);
83   List<Corner> Cclean = new ArrayList<Corner>(C.size());
84   for (int i = 0; i < Ca.length; i++) {
85     Corner c0 = Ca[i];       // get next strongest corner
86     if (c0 != null) {
87       Cclean.add(c0);
88       // delete all remaining corners cj too close to c0:
89       for (int j = i + 1; j < Ca.length; j++) {
90         Corner cj = Ca[j];
91         if (cj != null && c0.dist2(cj) < dmin2)
92           Ca[j] = null;    //delete corner cj from Ca
93       }
94     }
95   }
96   return Cclean;
97 }
```

Initially (in line 80) the corner list `C` is sorted by decreasing corner strenth q by calling the static method `sort()`.[9] The sorted sequence is then converted to an array (line 82) which is traversed from start to end (line 84–95). For each selected corner (`c0`), all subsequent corners (`cj`) with a distance $d_{\min}$ are deleted from the sequence (line 92). The "surviving" corners are then transferred to the final corner sequence `Cclean`.

Note that the call `c0.dist2(cj)` in line 91 returns the *squared* Euclidean distance between the corner points $\boldsymbol{c}_0$ and $\boldsymbol{c}_j$, that is, the quantity $d^2 = (x_0 - x_j)^2 + (y_0 - y_j)^2$. Since the square of the distance suffices for the comparison, we do not need to compute the actual distance, and consequently we avoid calling the expensive square root function. This is a common trick when comparing distances.

7.3.4 Summary

Most of the implementation steps we have just described are initiated through calls from the method `findCorners()` in class `HarrisCornerDetector`:

```java
98 public List<Corner> findCorners() {
```

[9] Defined in class `java.util.Collections`.

```
  FloatProcessor Q = makeCrf((float)params.alpha);
  List<Corner> corners =
      collectCorners(Q, (float)params.tH, params.border);
  if (params.doCleanUp) {
    corners = cleanupCorners(corners, params.dmin);
  }
  return corners;
}
```

An example of how to use the class `HarrisCornerDetector` is shown by the associated ImageJ plugin `Find_Corners` whose `run()` consists of only a few lines of code. This method simply creates a new object of the class `HarrisCornerDetector`, calls the `findCorners()` method, and finally displays the results in a new image (`R`):

```
public class Find_Corners implements PlugInFilter {

  public void run(ImageProcessor ip) {
    HarrisCornerDetector cd = new HarrisCornerDetector(ip);
    List<Corner> corners = cd.findCorners();
    ColorProcessor R = ip.convertToColorProcessor();
    drawCorners(R, corners);
    (new ImagePlus("Result", R)).show();
  }

  void drawCorners(ImageProcessor ip,
                       List<Corner> corners) {
    ip.setColor(cornerColor);
    for (Corner c : corners) {
      drawCorner(ip, c);
    }
  }

  void drawCorner(ImageProcessor ip, Corner c) {
    int size = cornerSize;
    int x = Math.round(c.getX());
    int y = Math.round(c.getY());
    ip.drawLine(x - size, y, x + size, y);
    ip.drawLine(x, y - size, x, y + size);
  }
}
```

For completeness, the definition of the `drawCorners()` method has been included here; the complete source code can be found online. Again, when writing this code, the focus is on understandability and not necessarily speed and memory usage. Many elements of the code can be optimized with relatively little effort (perhaps as an exercise?) if efficiency becomes important.

7.4 Exercises

Exercise 7.1. Adapt the `draw()` method in the class `Corner` (see p. 155) so that the strength (q-value) of the corner points can also be visualized. This could be done, for example, by manipulating the size, color, or intensity of the markers drawn in relation to the strength of the corner.

Exercise 7.2. Conduct a series of experiments to determine how image contrast affects the performance of the Harris detector, and then develop an idea for how you might automatically determine the parameter $\mathtt{t}_H$ depending on image content.

Exercise 7.3. Explore how rotation and distortion of the image affect the performance of the Harris corner detector. Based on your experiments, is the operator truly isotropic?

Exercise 7.4. Determine how image noise affects the performance of the Harris detector in terms of the positional accuracy of the detected corners and the omission of actual corners. Remark: ImageJ's menu command Process ▷ Noise ▷ Add Specified Noise... can be used to easily add certain types of random noise to a given image.