3

Histograms and Image Statistics

Histograms are used to depict image statistics in an easily interpreted visual format. With a histogram, it is easy to determine certain types of problems in an image, for example, it is simple to conclude if an image is properly exposed by visual inspection of its histogram. In fact, histograms are so useful that modern digital cameras often provide a real-time histogram overlay on the viewfinder (Fig. 3.1) to help prevent taking poorly exposed pictures. It is important to catch errors like this at the image capture stage because poor exposure results in a permanent loss of information, which it is not possible to recover later using image-processing techniques. In addition to their usefulness during image capture, histograms are also used later to improve the visual appearance of an image and as a "forensic" tool for determining what type of processing has previously been applied to an image. The final part of this chapter shows how to calculate simple image statistics from the original image, its histogram, or the so-called integral image.

Fig. 3.1
Digital camera back display showing the associated RGB histograms.

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_3

3.1 What is a Histogram?

Histograms in general are frequency distributions, and histograms of images describe the frequency of the intensity values that occur in an image. This concept can be easily explained by considering an old-fashioned grayscale image like the one shown in Fig. 3.2.

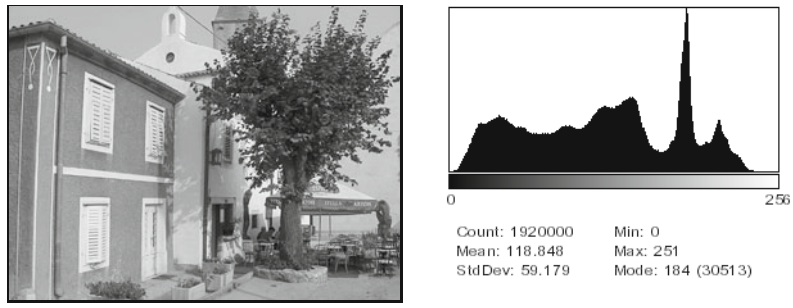


Fig. 3.2 An 8-bit grayscale image and a histogram depicting the frequency distribution of its 256 intensity values.

The histogram h for a grayscale image I with intensity values in the range $I(u,v) \in [0, K{-}1]$ holds exactly K entries, where $K = 2^8 = 256$ for a typical 8-bit grayscale image. Each single histogram entry is defined as

$$\mathsf{h}(i) = \text{the } \textit{number} \text{ of pixels in } I \text{ with the intensity value } i,$$

for all $0 \le i < K$. More formally stated,[1]

$$\mathsf{h}(i) = \text{card}\big\{(u,v) \mid I(u,v) = i\big\}. \tag{3.1}$$

Therefore, $\mathsf{h}(0)$ is the number of pixels with the value 0, $\mathsf{h}(1)$ the number of pixels with the value 1, and so forth. Finally, $\mathsf{h}(255)$ is the number of all white pixels with the maximum intensity value $255 = K{-}1$. The result of the histogram computation is a 1D vector h of length K. Figure 3.3 gives an example for an image with $K = 16$ possible intensity values.

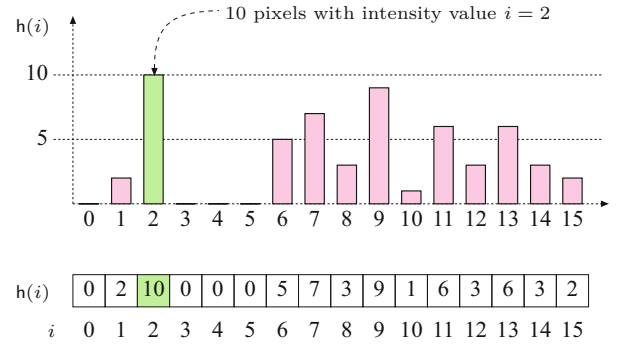


Fig. 3.3 Histogram vector for an image with $K = 16$ possible intensity values. The indices of the vector element $i = 0 \ldots 15$ represent intensity values. The value of 10 at index 2 means that the image contains 10 pixels of intensity value 2.

Since the histogram encodes no information about *where* each of its individual entries originated in the image, it contains no information about the spatial arrangement of pixels in the image. This

[1] card{...} denotes the number of elements ("cardinality") in a set (see also Sec. A.1 in the Appendix).

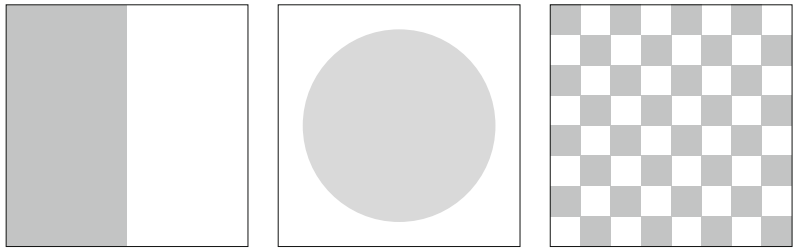


Fig. 3.4
Three very different images with identical histograms.

is intentional, since the main function of a histogram is to provide statistical information, (e.g., the distribution of intensity values) in a compact form. Is it possible to reconstruct an image using only its histogram? That is, can a histogram be somehow "inverted"? Given the loss of spatial information, in all but the most trivial cases, the answer is no. As an example, consider the wide variety of images you could construct using the same number of pixels of a specific value. These images would appear different but have exactly the same histogram (Fig. 3.4).

3.2 Interpreting Histograms

A histogram depicts problems that originate during image acquisition, such as those involving contrast and dynamic range, as well as artifacts resulting from image-processing steps that were applied to the image. Histograms are often used to determine if an image is making effective use of its intensity range (Fig. 3.5) by examining the size and uniformity of the histogram's distribution.

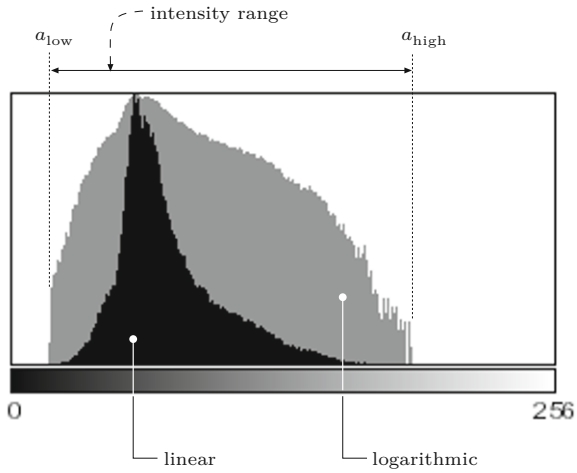
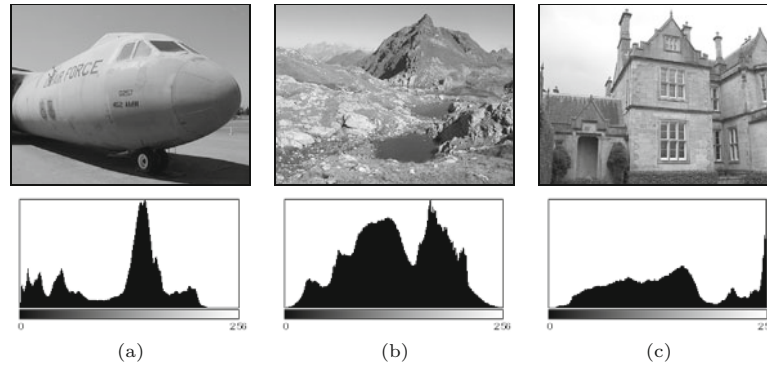


Fig. 3.5
Effective intensity range. The graph depicts the frequencies of pixel values *linearly* (black bars) and *logarithmically* (gray bars). The logarithmic form makes even relatively low occurrences, which can be very important in the image, readily apparent.

3.2.1 Image Acquisition

Histograms make typical exposure problems readily apparent. As an example, a histogram where a large section of the intensity range at one end is largely unused while the other end is crowded with



(a) (b) (c)

Fig. 3.6 Exposure errors are readily apparent in histograms. Underexposed (a), properly exposed (b), and overexposed (c) photographs.

high-value peaks (Fig. 3.6) is representative of an improperly exposed image.

Contrast

Contrast is understood as the range of intensity values *effectively* used within a given image, that is the difference between the image's maximum and minimum pixel values. A full-contrast image makes effective use of the entire range of available intensity values from $a = a_{\min}, \ldots, a_{\max}$ with $a_{\min} = 0$, $a_{\max} = K-1$ (black to white). Using this definition, image contrast can be easily read directly from the histogram. Figure 3.7 illustrates how varying the contrast of an image affects its histogram.

Dynamic range

The dynamic range of an image is, in principle, understood as the number of *distinct* pixel values in an image. In the ideal case, the dynamic range encompasses all K usable pixel values, in which case the value range is completely utilized. When an image has an available range of contrast $a = a_{\text{low}}, \ldots, a_{\text{high}}$, with

$$a_{\min} < a_{\text{low}} \qquad \text{and} \qquad a_{\text{high}} < a_{\max}\,,$$

then the maximum possible dynamic range is achieved when all the intensity values lying in this range are utilized (i.e., appear in the image; Fig. 3.8).

While the contrast of an image can be increased by transforming its existing values so that they utilize more of the underlying value range available, the dynamic range of an image can only be increased by introducing artificial (that is, not originating with the image sensor) values using methods such as interpolation (see Ch. 22). An image with a high dynamic range is desirable because it will suffer less image-quality degradation during image processing and compression. Since it is not possible to increase dynamic range after image acquisition in a practical way, professional cameras and scanners work at depths of more than 8 bits, often 12–14 bits per channel, in order to provide high dynamic range at the acquisition stage. While most output devices, such as monitors and printers, are unable to actually reproduce more than 256 different shades, a high dynamic range is always beneficial for subsequent image processing or archiving.

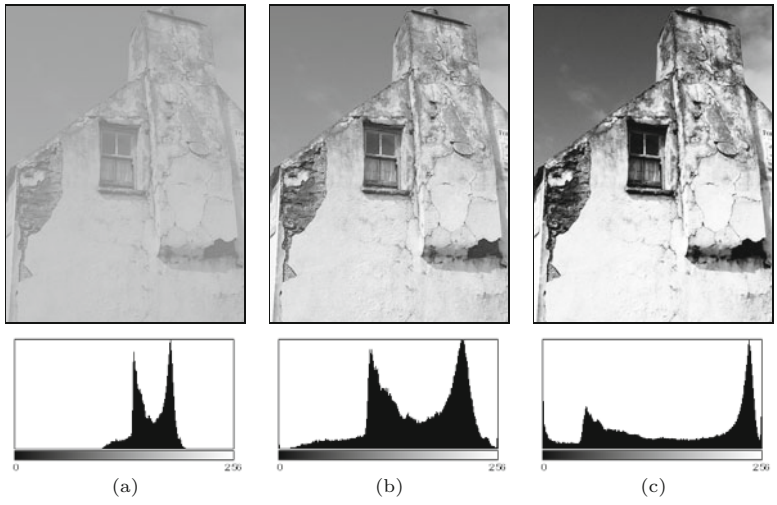


Fig. 3.7
How changes in contrast affect the histogram: low contrast (a), normal contrast (b), high contrast (c).

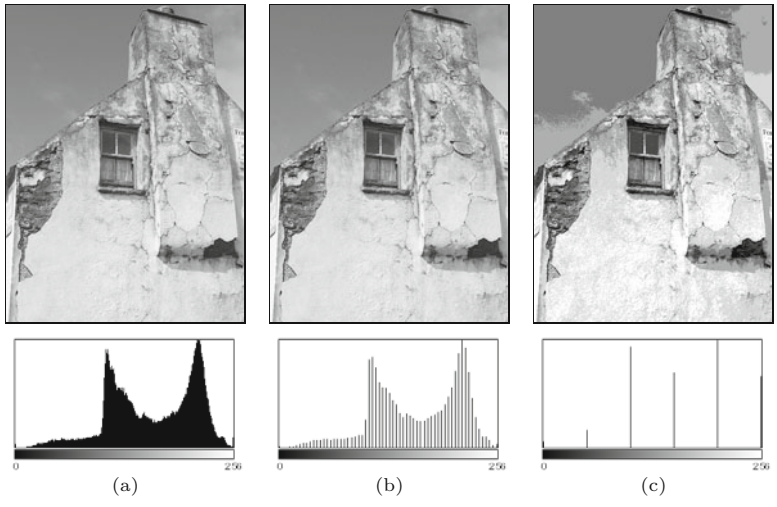


Fig. 3.8
How changes in dynamic range affect the histogram: high dynamic range (a), low dynamic range with 64 intensity values (b), extremely low dynamic range with only 6 intensity values (c).

3.2.2 Image Defects

Histograms can be used to detect a wide range of image defects that originate either during image acquisition or as the result of later image processing. Since histograms always depend on the visual characteristics of the scene captured in the image, no single "ideal" histogram exists. While a given histogram may be optimal for a specific scene, it may be entirely unacceptable for another. As an example, the ideal histogram for an astronomical image would likely be very different from that of a good landscape or portrait photo. Nevertheless, there are some general rules; for example, when taking a landscape image with a digital camera, you can expect the histogram to have evenly distributed intensity values and no isolated spikes.

Saturation

Ideally the contrast range of a sensor, such as that used in a camera, should be greater than the range of the intensity of the light that it receives from a scene. In such a case, the resulting histogram will

be smooth at both ends because the light received from the very bright and the very dark parts of the scene will be less than the light received from the other parts of the scene. Unfortunately, this ideal is often not the case in reality, and illumination outside of the sensor's contrast range, arising for example from glossy highlights and especially dark parts of the scene, cannot be captured and is lost. The result is a histogram that is saturated at one or both ends of its range. The illumination values lying outside of the sensor's range are mapped to its minimum or maximum values and appear on the histogram as significant spikes at the tail ends. This typically occurs in an under- or overexposed image and is generally not avoidable when the inherent contrast range of the scene exceeds the range of the system's sensor (Fig. 3.9(a)).

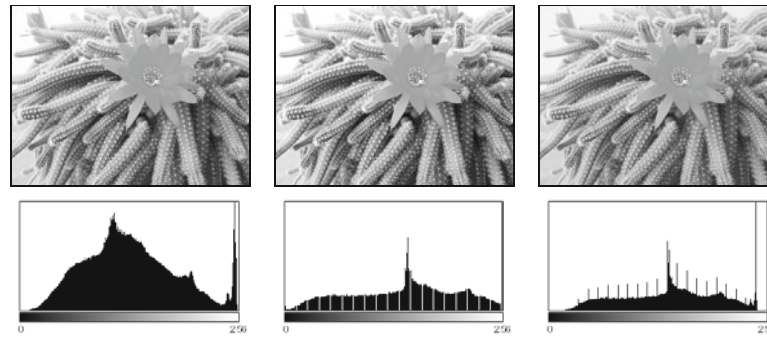


Fig. 3.9
Effect of image capture errors on histograms: saturation of high intensities (a), histogram gaps caused by a slight increase in contrast (b), and histogram spikes resulting from a reduction in contrast (c).

Spikes and gaps

As discussed already, the intensity value distribution for an unprocessed image is generally smooth; that is, it is unlikely that isolated spikes (except for possible saturation effects at the tails) or gaps will appear in its histogram. It is also unlikely that the count of any given intensity value will differ greatly from that of its neighbors (i.e., it is locally smooth). While artifacts like these are observed very rarely in original images, they will often be present after an image has been manipulated, for instance, by changing its contrast. Increasing the contrast (see Ch. 4) causes the histogram lines to separate from each other and, due to the discrete values, gaps are created in the histogram (Fig. 3.9(b)). Decreasing the contrast leads, again because of the discrete values, to the merging of values that were previously distinct. This results in increases in the corresponding histogram entries and ultimately leads to highly visible spikes in the histogram (Fig. 3.9(c)).[2]

Impacts of image compression

Image compression also changes an image in ways that are immediately evident in its histogram. As an example, during GIF compression, an image's dynamic range is reduced to only a few intensities

[2] Unfortunately, these types of errors are also caused by the internal contrast "optimization" routines of some image-capture devices, especially consumer-type scanners.

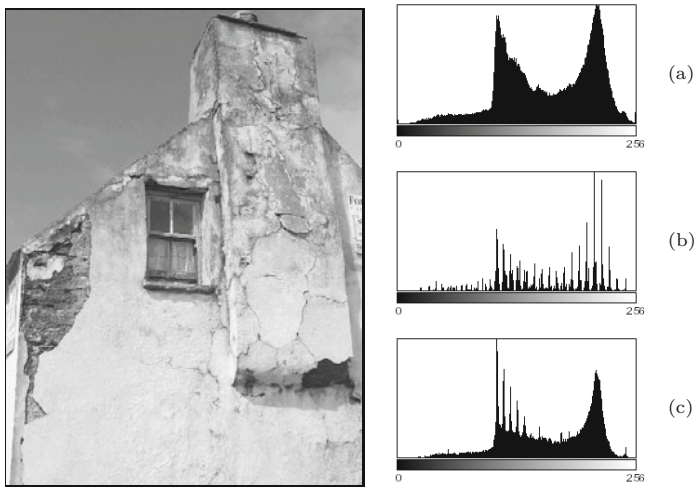


Fig. 3.10
Color quantization effects resulting from GIF conversion. The original image converted to a 256 color GIF image (left). Original histogram (a) and the histogram after GIF conversion (b). When the RGB image is scaled by 50%, some of the lost colors are recreated by interpolation, but the results of the GIF conversion remain clearly visible in the histogram (c).

or colors, resulting in an obvious line structure in the histogram that cannot be removed by subsequent processing (Fig. 3.10). Generally, a histogram can quickly reveal whether an image has ever been subjected to color quantization, such as occurs during conversion to a GIF image, even if the image has subsequently been converted to a full-color format such as TIFF or JPEG.

Figure 3.11 illustrates what occurs when a simple line graphic with only two gray values (128, 255) is subjected to a compression method such as JPEG, that is not designed for line graphics but instead for natural photographs. The histogram of the resulting image clearly shows that it now contains a large number of gray values that were not present in the original image, resulting in a poor-quality image[3] that appears dirty, fuzzy, and blurred.

3.3 Calculating Histograms

Computing the histogram of an 8-bit grayscale image containing intensity values between 0 and 255 is a simple task. All we need is a set of 256 counters, one for each possible intensity value. First, all counters are initialized to zero. Then we iterate through the image I, determining the pixel value p at each location (u, v), and incrementing the corresponding counter by one. At the end, each counter will contain the number of pixels in the image that have the corresponding intensity value.

An image with K possible intensity values requires exactly K counter variables; for example, since an 8-bit grayscale image can contain at most 256 different intensity values, we require 256 counters. While individual counters make sense conceptually, an actual

[3] Using JPEG compression on images like this, for which it was not designed, is one of the most egregious of imaging errors. JPEG is designed for photographs of natural scenes with smooth color transitions, and using it to compress iconic images with large areas of the same color results in strong visual artifacts (see, e.g., Fig. 1.9 on p. 17).

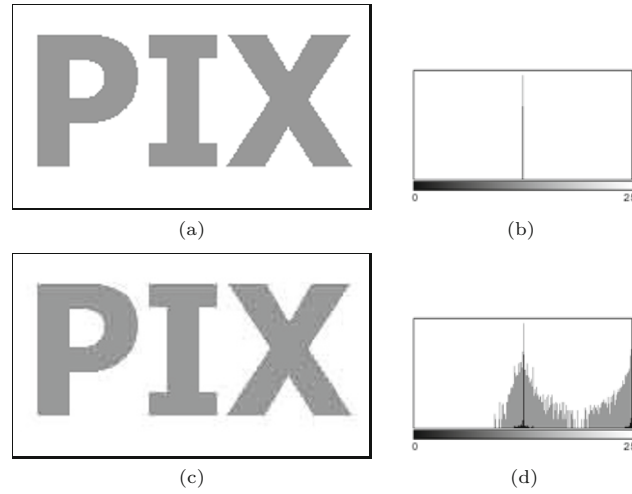


Fig. 3.11 Effects of JPEG compression. The original image (a) contained only two different gray values, as its histogram (b) makes readily apparent. JPEG compression, a poor choice for this type of image, results in numerous additional gray values, which are visible in both the resulting image (c) and its histogram (d). In both histograms, the linear frequency (black bars) and the logarithmic frequency (gray bars) are shown.

Prog. 3.1 ImageJ plugin for computing the histogram of an 8-bit grayscale image. The `setup()` method returns `DOES_8G + NO_CHANGES`, which indicates that this plugin requires an 8-bit grayscale image and will not alter it (line 4). In Java, all elements of a newly instantiated numerical array are automatically initialized to zero (line 8).

```
1  public class Compute_Histogram implements PlugInFilter {
2
3      public int setup(String arg, ImagePlus img) {
4          return DOES_8G + NO_CHANGES;
5      }
6
7      public void run(ImageProcessor ip) {
8          int[] h = new int[256]; // histogram array
9          int w = ip.getWidth();
10         int h = ip.getHeight();
11
12         for (int v = 0; v < h; v++) {
13             for (int u = 0; u < w; u++) {
14                 int i = ip.getPixel(u, v);
15                 h[i] = h[i] + 1;
16             }
17         }
18         // ... histogram h can now be used
19     }
20 }
```

implementation would not use K individual *variables* to represent the counters but instead would use an *array* with K entries (`int[256]` in Java). In this example, the actual implementation as an array is straightforward. Since the intensity values begin at zero (like arrays in Java) and are all positive, they can be used directly as the indices $i \in [0, N-1]$ of the histogram array. Program 3.1 contains the complete Java source code for computing a histogram within the `run()` method of an ImageJ plugin.

At the start of Prog. 3.1, the array `h` of type `int[]` is created (line 8) and its elements are automatically initialized[4] to `0`. It makes no difference, at least in terms of the final result, whether the array is

[4] In Java, arrays of primitives such as `int`, `double` are initialized at creation to `0` in the case of integer types or `0.0` for floating-point types, while arrays of objects are initialized to `null`.

traversed in row or column order, as long as all pixels in the image are visited exactly once. In contrast to Prog. 2.1, in this example we traverse the array in the standard row-first order such that the outer `for` loop iterates over the *vertical* coordinates v and the inner loop over the *horizontal* coordinates u.[5] Once the histogram has been calculated, it is available for further processing steps or for being displayed.

Of course, histogram computation is already implemented in ImageJ and is available via the method `getHistogram()` for objects of the class `ImageProcessor`. If we use this built-in method, the `run()` method of Prog. 3.1 can be simplified to

```
public void run(ImageProcessor ip) {
    int[] h = ip.getHistogram();   // built-in ImageJ method
    ... // histogram h can now be used
}
```

3.4 Histograms of Images with More than 8 Bits

Normally histograms are computed in order to visualize the image's distribution on the screen. This presents no problem when dealing with images having $2^8 = 256$ entries, but when an image uses a larger range of values, for instance 16- and 32-bit or floating-point images (see Table 1.1), then the growing number of necessary histogram entries makes this no longer practical.

3.4.1 Binning

Since it is not possible to represent each intensity value with its own entry in the histogram, we will instead let a given entry in the histogram represent a *range* of intensity values. This technique is often referred to as "binning" since you can visualize it as collecting a range of pixel values in a container such as a bin or bucket. In a binned histogram of size B, each bin $\mathsf{h}(j)$ contains the number of image elements having values within the interval $[a_j, a_{j+1})$, and therefore (analogous to Eqn. (3.1))

$$\mathsf{h}(j) = \text{card}\left\{(u,v) \mid a_j \le I(u,v) < a_{j+1}\right\}, \tag{3.2}$$

for $0 \le j < B$. Typically the range of possible values in B is divided into bins of equal size $k_B = K/B$ such that the starting value of the interval j is

$$a_j = j \cdot \frac{K}{B} = j \cdot k_B \,.$$

3.4.2 Example

In order to create a typical histogram containing $B = 256$ entries from a 14-bit image, one would divide the original value range $j =$

[5] In this way, image elements are traversed in exactly the same way that they are laid out in computer memory, resulting in more efficient memory access and with it the possibility of increased performance, especially when dealing with larger images (see also Appendix F).

$0, \ldots, 2^{14}-1$ into 256 equal intervals, each of length $k_B = 2^{14}/256 = 64$, such that $a_0 = 0$, $a_1 = 64$, $a_2 = 128$, $\ldots$, $a_{255} = 16320$ and $a_{256} = a_B = 2^{14} = 16320 = K$. This gives the following association between pixel values and histogram bins $\mathsf{h}(0), \ldots, \mathsf{h}(255)$:

$$\begin{aligned} 0, \ldots, \quad 63 &\rightarrow \mathsf{h}(0), \\ 64, \ldots, \quad 127 &\rightarrow \mathsf{h}(1), \\ 128, \ldots, \quad 191 &\rightarrow \mathsf{h}(2), \\ \vdots \qquad\quad \vdots \quad &\quad \vdots \\ 16320, \ldots, 16383 &\rightarrow \mathsf{h}(255). \end{aligned}$$

3.4.3 Implementation

If, as in the previous example, the value range $0, \ldots, K-1$ is divided into equal length intervals $k_B = K/B$, there is naturally no need to use a mapping table to find a_j since for a given pixel value $a = I(u, v)$ the correct histogram element j is easily computed. In this case, it is enough to simply divide the pixel value $I(u, v)$ by the interval length k_B; that is,

$$\frac{I(u,v)}{k_B} = \frac{I(u,v)}{K/B} = \frac{I(u,v) \cdot B}{K}\,. \tag{3.3}$$

As an index to the appropriate histogram bin $\mathsf{h}(j)$, we require an integer value

$$j = \left\lfloor \frac{I(u,v) \cdot B}{K} \right\rfloor, \tag{3.4}$$

where $\lfloor \cdot \rfloor$ denotes the *floor* operator.[6] A Java method for computing histograms by "linear binning" is given in Prog. 3.2. Note that all the computations from Eqn. (3.4) are done with integer numbers without using any floating-point operations. Also there is no need to explicitly call the *floor* function because the expression

```
a * B / K
```

in line 11 uses integer division and in Java the fractional result of such an operation is truncated, which is equivalent to applying the floor function (assuming positive arguments).[7] The binning method can also be applied, in a similar way, to floating-point images.

3.5 Histograms of Color Images

When referring to histograms of color images, typically what is meant is a histogram of the image intensity (luminance) or of the individual color channels. Both of these variants are supported by practically every image-processing application and are used to objectively appraise the image quality, especially directly after image acquisition.

[6] $\lfloor x \rfloor$ rounds x down to the next whole number (see Appendix A).

[7] For a more detailed discussion, see the section on integer division in Java in Appendix F (p. 765).

```
1  int[] binnedHistogram(ImageProcessor ip) {
2    int K = 256; // number of intensity values
3    int B = 32; // size of histogram, must be defined
4    int[] H = new int[B]; // histogram array
5    int w = ip.getWidth();
6    int h = ip.getHeight();
7
8    for (int v = 0; v < h; v++) {
9      for (int u = 0; u < w; u++) {
10       int a = ip.getPixel(u, v);
11       int i = a * B / K; // integer operations only!
12       H[i] = H[i] + 1;
13     }
14   }
15   // return binned histogram
16   return H;
17 }
```

Prog. 3.2
Histogram computation using "binning" (Java method). Example of computing a histogram with $B = 32$ bins for an 8-bit grayscale image with $K = 256$ intensity levels. The method `binnedHistogram()` returns the histogram of the image object `ip` passed to it as an `int` array of size B.

3.5.1 Intensity Histograms

The intensity or *luminance* histogram h_{Lum} of a color image is nothing more than the histogram of the corresponding grayscale image, so naturally all aspects of the preceding discussion also apply to this type of histogram. The grayscale image is obtained by computing the luminance of the individual channels of the color image. When computing the luminance, it is not sufficient to simply average the values of each color channel; instead, a weighted sum that takes into account color perception theory should be computed. This process is explained in detail in Chapter 12 (p. 304).

3.5.2 Individual Color Channel Histograms

Even though the luminance histogram takes into account all color channels, image errors appearing in single channels can remain undiscovered. For example, the luminance histogram may appear clean even when one of the color channels is oversaturated. In RGB images, the blue channel contributes only a small amount to the total brightness and so is especially sensitive to this problem.

Component histograms supply additional information about the intensity distribution within the individual color channels. When computing component histograms, each color channel is considered a separate intensity image and each histogram is computed independently of the other channels. Figure 3.12 shows the luminance histogram h_{Lum} and the three component histograms h_{R}, h_{G}, and h_{B} of a typical RGB color image. Notice that saturation problems in all three channels (red in the upper intensity region, green and blue in the lower regions) are obvious in the component histograms but not in the luminance histogram. In this case it is striking, and not at all atypical, that the three component histograms appear completely different from the corresponding luminance histogram h_{Lum} (Fig. 3.12(b)).

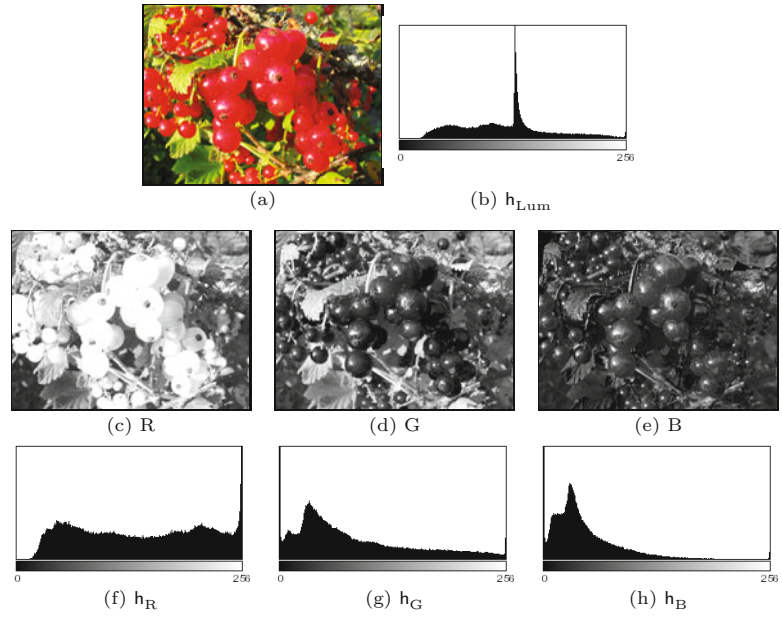


Fig. 3.12 Histograms of an RGB color image: original image (a), luminance histogram h_{Lum} (b), RGB color components as intensity images (c–e), and the associated component histograms h_R, h_G, h_B (f–h). The fact that all three color channels have saturation problems is only apparent in the individual component histograms. The spike in the distribution resulting from this is found in the middle of the luminance histogram (b).

3.5.3 Combined Color Histograms

Luminance histograms and component histograms both provide useful information about the lighting, contrast, dynamic range, and saturation effects relative to the individual color components. It is important to remember that they provide no information about the distribution of the actual *colors* in the image because they are based on the individual color channels and not the combination of the individual channels that forms the color of an individual pixel. Consider, for example, when h_R, the component histogram for the red channel, contains the entry

$$h_R(200) = 24.$$

Then it is only known that the image has 24 pixels that have a red intensity value of 200. The entry does not tell us anything about the green and blue values of those pixels, which could be any valid value $(*)$, that is,

$$(r, g, b) = (200, *, *).$$

Suppose further that the three component histograms included the following entries:

$$h_R(50) = 100, \quad h_G(50) = 100, \quad h_B(50) = 100.$$

Could we conclude from this that the image contains 100 pixels with the color combination

$$(r, g, b) = (50, 50, 50)$$

or that this color occurs at all? In general, no, because there is no way of ascertaining from these data if there exists a pixel in the image in which all three components have the value 50. The only thing we could really say is that the color value $(50, 50, 50)$ can occur at most 100 times in this image.

So, although conventional (intensity or component) histograms of color images depict important properties, they do not really provide any useful information about the composition of the actual colors in an image. In fact, a collection of color images can have very similar component histograms and still contain entirely different colors. This leads to the interesting topic of the *combined* histogram, which uses statistical information about the combined color components in an attempt to determine if two images are roughly similar in their color composition. Features computed from this type of histogram often form the foundation of color-based image retrieval methods. We will return to this topic in Chapter 12, where we will explore color images in greater detail.

3.6 The Cumulative Histogram

The cumulative histogram, which is derived from the ordinary histogram, is useful when performing certain image operations involving histograms; for instance, histogram equalization (see Sec. 4.5). The cumulative histogram H is defined as

$$\mathsf{H}(i) = \sum_{j=0}^{i} \mathsf{h}(j) \qquad \text{for } 0 \le i < K. \tag{3.5}$$

A particular value $\mathsf{H}(i)$ is thus the sum of all histogram values $\mathsf{h}(j)$, with $j \le i$. Alternatively, we can define H recursively (as implemented in Prog. 4.2 on p. 66):

$$\mathsf{H}(i) = \begin{cases} \mathsf{h}(0) & \text{for } i = 0, \\ \mathsf{H}(i-1) + \mathsf{h}(i) & \text{for } 0 < i < K. \end{cases} \tag{3.6}$$

The cumulative histogram $\mathsf{H}(i)$ is a monotonically increasing function with the maximum value

$$\mathsf{H}(K-1) = \sum_{j=0}^{K-1} \mathsf{h}(j) = M \cdot N, \tag{3.7}$$

that is, the total number of pixels in an image of width M and height N. Figure 3.13 shows a concrete example of a cumulative histogram.

The cumulative histogram is useful not primarily for viewing but as a simple and powerful tool for capturing statistical information from an image. In particular, we will use it in the next chapter to compute the parameters for several common point operations (see Sec. 4.4–4.6).

3.7 Statistical Information from the Histogram

Some common statistical parameters of the image can be conveniently calculated directly from its histogram. For example, the minimum and maximum pixel value of an image I can be obtained by simply

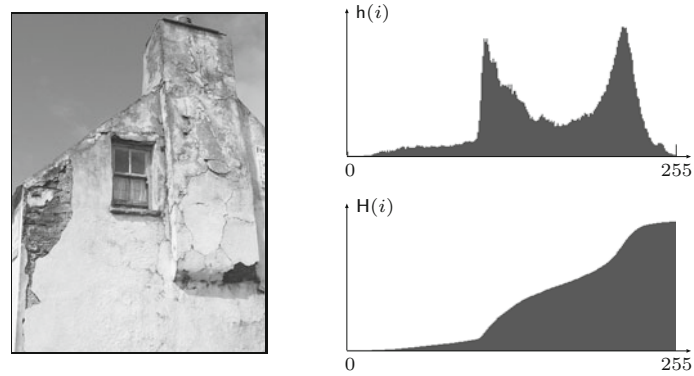


Fig. 3.13 The ordinary histogram $\mathsf{h}(i)$ and its associated cumulative histogram $\mathsf{H}(i)$.

finding the smallest and largest histogram index with nonzero value, i.e.,

$$\begin{aligned} \min(I) &= \min\{\, i \mid \mathsf{h}(i) > 0\}, \\ \max(I) &= \max\{\, i \mid \mathsf{h}(i) > 0\}. \end{aligned} \tag{3.8}$$

If we assume that the histogram is already available, the advantage is that the calculation does not include the entire image but only the relatively small set of histogram elements (typ. 256).

3.7.1 Mean and Variance

The *mean* value μ of an image I (of size $M \times N$) can be calculated as

$$\mu = \frac{1}{MN} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} I(u,v) = \frac{1}{MN} \cdot \sum_{i=0}^{K-1} \mathsf{h}(i) \cdot i, \tag{3.9}$$

i.e., either directly from the pixel values $I(u,v)$ or indirectly from the histogram h (of size K), where $MN = \sum_i \mathsf{h}(i)$ is the total number of pixels.

Analogously we can also calculate the *variance* of the pixel values straight from the histogram as

$$\sigma^2 = \frac{1}{MN} \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} \big[I(u,v) - \mu\big]^2 = \frac{1}{MN} \cdot \sum_{i=0}^{K-1} (i-\mu)^2 \cdot \mathsf{h}(i). \tag{3.10}$$

As we see in the right parts of Eqns. (3.9) and (3.10), there is no need to access the original pixel values.

The formulation of the variance in Eqn. (3.10) assumes that the arithmetic mean μ has already been determined. This is not necessary though, since the mean and the variance can be calculated together in a single iteration over the image pixels or the associated histogram in the form

$$\mu = \frac{1}{MN} \cdot A \qquad \text{and} \tag{3.11}$$

$$\sigma^2 = \frac{1}{MN} \cdot \Big(B - \frac{1}{MN} \cdot A^2\Big), \tag{3.12}$$

with the quantities

$$A = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} I(u,v) \;= \sum_{i=0}^{K-1} i \cdot \mathsf{h}(i), \tag{3.13}$$

$$B = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} I^2(u,v) = \sum_{i=0}^{K-1} i^2 \cdot \mathsf{h}(i). \tag{3.14}$$

The above formulation has the additional numerical advantage that all summations can be performed with integer values, in contrast to Eqn. (3.10) which requires the summation of floating-point values.

3.7.2 Median

The median m of an image is defined as the smallest pixel value that is greater or equal to one half of all pixel values, i.e., lies "in the middle" of the pixel values.[8] The median can also be easily calculated from the image's histogram.

To determine the median of an image I from the associated histogram it is sufficient to find the index i that separates the histogram into two halves, such that the sum of the histogram entries to the left and the right of i are approximately equal. In other words, i is the smallest index where the sum of the histogram entries below (and including) i corresponds to at least half of the image size, that is,

$$m = \min\Big\{ i \mid \sum_{j=0}^{i} \mathsf{h}(j) \geq \frac{MN}{2} \Big\}. \tag{3.15}$$

Since $\sum_{j=0}^{i} \mathsf{h}(j) = \mathsf{H}(i)$ (see Eqn. (3.5)), the median calculation can be formulated even simpler as

$$m = \min\Big\{ i \mid \mathsf{H}(i) \geq \frac{MN}{2} \Big\}, \tag{3.16}$$

given the cumulative histogram H.

3.8 Block Statistics

3.8.1 Integral Images

Integral images (also known as *summed area tables* [58]) provide a simple way for quickly calculating elementary statistics of arbitrary rectangular sub-images. They have found use in several interesting applications, such as fast filtering, adaptive thresholding, image matching, local feature extraction, face detection, and stereo reconstruction [20, 142, 244].

Given a scalar-valued (grayscale) image $I\colon M \times N \mapsto \mathbb{R}$ the associated *first-order* integral image is defined as

$$\Sigma_1(u,v) = \sum_{i=0}^{u} \sum_{j=0}^{v} I(i,j). \tag{3.17}$$

[8] See Sec. 5.4.2 for an alternative definition of the median.

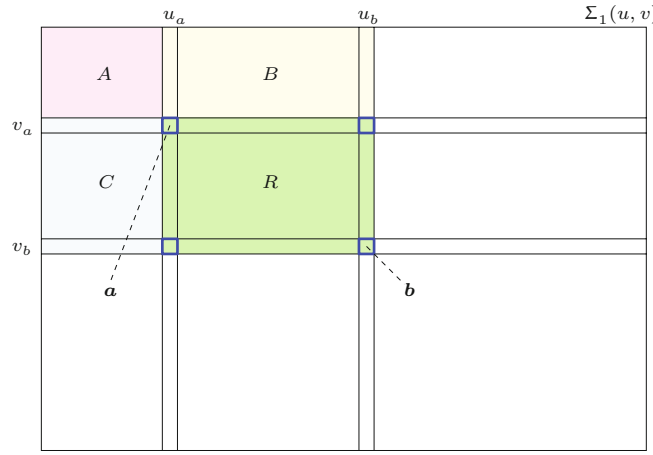


Fig. 3.14 Block-based calculations with integral images. Only four samples from the integral image Σ_1 are required to calculate the sum of the pixels inside the (green) rectangle $R = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$, defined by the corner coordinates $\boldsymbol{a} = (u_a, v_a)$ and $\boldsymbol{b} = (u_b, v_b)$.

Thus a value in Σ_1 is the sum of all pixel values in the original image I located to the left and above the given position (u, v), inclusively. The integral image can be calculated efficiently with a single pass over the image I by using the recurrence relation

$$\Sigma_1(u,v) = \begin{cases} 0 & \text{for } u < 0 \text{ or } v < 0, \\ \Sigma_1(u-1,v) + \Sigma_1(u,v-1) - & \\ \quad \Sigma_1(u-1,v-1) + I(u,v) & \text{for } u, v \geq 0, \end{cases} \tag{3.18}$$

for positions $u = 0, \ldots, M-1$ and $v = 0, \ldots, N-1$ (see Alg. 3.1).

Suppose now that we wanted to calculate the sum of the pixel values in a given rectangular region R, defined by the corner positions $\boldsymbol{a} = (u_a, v_a)$, $\boldsymbol{b} = (u_b, v_b)$, that is, the *first-order block sum*

$$S_1(R) = \sum_{i=u_a}^{u_b} \sum_{j=v_a}^{v_b} I(i,j), \tag{3.19}$$

from the integral image Σ_1. As shown in Fig. 3.14, the quantity $\Sigma_1(u_a-1, v_a-1)$ corresponds to the pixel sum within rectangle A, and $\Sigma_1(u_b, v_b)$ is the pixel sum over all four rectangles A, B, C and R, that is,

$$\begin{aligned} \Sigma_1(u_a-1, v_a-1) &= S_1(A), \\ \Sigma_1(u_b, v_a-1) &= S_1(A) + S_1(B), \\ \Sigma_1(u_a-1, v_b) &= S_1(A) + S_1(C), \\ \Sigma_1(u_b, v_b) &= S_1(A) + S_1(B) + S_1(C) + S_1(R). \end{aligned} \tag{3.20}$$

Thus $S_1(R)$ can be calculated as

$$\begin{aligned} S_1(R) &= \underbrace{S_1(A)+S_1(B)+S_1(C)+S_1(R)}_{\Sigma_1(u_b,v_b)} + \underbrace{S_1(A)}_{\Sigma_1(u_a-1,v_a-1)} \\ &\quad - \underbrace{[S_1(A)+S_1(B)]}_{\Sigma_1(u_b,v_a-1)} - \underbrace{[S_1(A)+S_1(C)]}_{\Sigma_1(u_a-1,v_b)} \\ &= \Sigma_1(u_b, v_b) + \Sigma_1(u_a-1, v_a-1) - \Sigma_1(u_b, v_a-1) - \Sigma_1(u_a-1, v_b), \end{aligned} \tag{3.21}$$

that is, by taking only *four* samples from the integral image Σ_1.

3.8.2 Mean Intensity

Given the region size N_R and the sum of the pixel values $S_1(R)$, the average intensity value (*mean*) inside the rectangle R can now easily be found as

$$\mu_R = \frac{1}{N_R} \cdot S_1(R), \tag{3.22}$$

with $S_1(R)$ as defined in Eqn. (3.21) and the region size

$$N_R = |R| = (u_b - u_a + 1) \cdot (v_b - v_a + 1). \tag{3.23}$$

3.8.3 Variance

Calculating the *variance* inside a rectangular region R requires the summation of squared intensity values, that is, tabulating

$$\Sigma_2(u, v) = \sum_{i=0}^{u} \sum_{j=0}^{v} I^2(i, j), \tag{3.24}$$

which can be performed analogously to Eqn. (3.18) in the form

$$\Sigma_2(u, v) = \begin{cases} 0 & \text{for } u < 0 \text{ or } v < 0, \\ \Sigma_2(u-1, v) + \Sigma_2(u, v-1) - & \\ \quad \Sigma_2(u-1, v-1) + I^2(u, v) & \text{for } u, v \geq 0. \end{cases} \tag{3.25}$$

As in Eqns. (3.19)–(3.21), the sum of the *squared* values inside a given rectangle R (i.e., the *second-order block sum*) can be obtained as

$$\begin{aligned} S_2(R) &= \sum_{i=u_0}^{u_1} \sum_{j=v_0}^{v_1} I^2(i, j) \\ &= \Sigma_2(u_b, v_b) + \Sigma_2(u_a-1, v_a-1) - \Sigma_2(u_b, v_a-1) - \Sigma_2(u_a-1, v_b). \end{aligned} \tag{3.26}$$

From this, the variance inside the rectangular region R is finally calculated as

$$\sigma_R^2 = \frac{1}{N_R}\big[S_2(R) - \frac{1}{N_R} \cdot (S_1(R))^2\big], \tag{3.27}$$

with N_R as defined in Eqn. (3.23). In addition, certain higher-order statistics can be efficiently calculated with summation tables in a similar fashion.

3.8.4 Practical Calculation of Integral Images

Algorithm 3.1 shows how Σ_1 and Σ_2 can be calculated in a single iteration over the original image I. Note that the accumulated values in the integral images Σ_1, Σ_2 tend to become quite large. Even with pictures of medium size and 8-bit intensity values, the range of 32-bit integers is quickly exhausted (particularly when calculating Σ_2). The use of 64-bit integers (type `long` in Java) or larger is recommended to avoid arithmetic overflow. A basic implementation of integral images is available as part of the `imagingbook` library.[9]

[9] Class `imagingbook.lib.image.IntegralImage`.

Alg. 3.1 Joint calculation of the integral images Σ_1 and Σ_2 for a scalar-valued image I.

1: **IntegralImage**(I)
Input: I, a scalar-valued input image with $I(u,v) \in \mathbb{R}$.
Returns the first and second order integral images of I.
2: $(M, N) \leftarrow \mathsf{Size}(I)$
3: Create maps $\Sigma_1, \Sigma_2 \colon M \times N \mapsto \mathbb{R}$
Process the first image line ($v = 0$):
4: $\Sigma_1(0,0) \leftarrow I(0,0)$
5: $\Sigma_2(0,0) \leftarrow I^2(0,0)$
6: **for** $u \leftarrow 1, \ldots, M-1$ **do**
7: $\Sigma_1(u,0) \leftarrow \Sigma_1(u-1,0) + I(u,0)$
8: $\Sigma_2(u,0) \leftarrow \Sigma_2(u-1,0) + I^2(u,0)$
Process the remaining image lines ($v > 0$):
9: **for** $v \leftarrow 1, \ldots, N-1$ **do**
10: $\Sigma_1(0,v) \leftarrow \Sigma_1(0,v-1) + I(0,v)$
11: $\Sigma_2(0,v) \leftarrow \Sigma_2(0,v-1) + I^2(0,v)$
12: **for** $u \leftarrow 1, \ldots, M-1$ **do**
13: $\Sigma_1(u,v) \leftarrow \Sigma_1(u-1,v) + \Sigma_1(u,v-1) - \Sigma_1(u-1,v-1) + I(u,v)$
14: $\Sigma_2(u,v) \leftarrow \Sigma_2(u-1,v) + \Sigma_2(u,v-1) - \Sigma_2(u-1,v-1) + I^2(u,v)$
15: **return** (Σ_1, Σ_2)

3.9 Exercises

Exercise 3.1. In Prog. 3.2, `B` and `K` are constants. Consider if there would be an advantage to computing the value of `B/K` outside of the loop, and explain your reasoning.

Exercise 3.2. Develop an ImageJ plugin that computes the cumulative histogram of an 8-bit grayscale image and displays it as a new image, similar to $\mathsf{H}(i)$ in Fig. 3.13. *Hint:* Use the `ImageProcessor` method `int[] getHistogram()` to retrieve the original image's histogram values and then compute the cumulative histogram "in place" according to Eqn. (3.6). Create a new (blank) image of appropriate size (e.g., 256×150) and draw the scaled histogram data as black vertical bars such that the maximum entry spans the full height of the image. Program 3.3 shows how this plugin could be set up and how a new image is created and displayed.

Exercise 3.3. Develop a technique for nonlinear binning that uses a table of interval limits a_j (Eqn. (3.2)).

Exercise 3.4. Develop an ImageJ plugin that uses the Java methods `Math.random()` or `Random.nextInt(int n)` to create an image with random pixel values that are uniformly distributed in the range $[0, 255]$. Analyze the image's histogram to determine how equally distributed the pixel values truly are.

Exercise 3.5. Develop an ImageJ plugin that creates a random image with a Gaussian (normal) distribution with mean value $\mu = 128$ and standard deviation $\sigma = 50$. Use the standard Java method `double Random.nextGaussian()` to produce normally-distributed

random numbers (with $\mu = 0$ and $\sigma = 1$) and scale them appropriately to pixel values. Analyze the resulting image histogram to see if it shows a Gaussian distribution too.

Exercise 3.6. Implement the calculation of the arithmetic *mean* μ and the *variance* σ^2 of a given grayscale image from its histogram `h` (see Sec. 3.7.1). Compare your results to those returned by ImageJ's Analyze ▷ Measure tool (they should match *exactly*).

Exercise 3.7. Implement the first-order integral image (Σ_1) calculation described in Eqn. (3.18) and calculate the sum of pixel values $S_1(R)$ inside a given rectangle R using Eqn. (3.21). Verify numerically that the results are the same as with the naive formulation in Eqn. (3.19).

Exercise 3.8. Values of integral images tend to become quite large. Assume that 32-bit signed integers (`int`) are used to calculate the integral of the squared pixel values, that is, Σ_2 (see Eqn. (3.24)), for an 8-bit grayscale image. What is the maximum image size that is guaranteed not to cause an arithmetic overflow? Perform the same analysis for 64-bit signed integers (`long`).

Exercise 3.9. Calculate the integral image Σ_1 for a given image I, convert it to a floating-point iamge (`FloatProcessor`) and display the result. You will realize that integral images are without any apparent structure and they all look more or less the same. Come up with an efficient method for reconstructing the original image I from Σ_1.

Prog. 3.3 Creating and displaying a new image (ImageJ plugin). First, we create a `ByteProcessor` object (`histIp`, line 20) that is subsequently filled. At this point, `histIp` has no screen representation and is thus not visible. Then, an associated `ImagePlus` object is created (line 33) and displayed by applying the `show()` method (line 34). Notice how the title (`String`) is retrieved from the original image inside the `setup()` method (line 10) and used to compose the new image's title (lines 30 and 33). If `histIp` is changed *after* calling `show()`, then the method `updateAndDraw()` could be used to redisplay the associated image again (line 34).

```
1  import ij.ImagePlus;
2  import ij.plugin.filter.PlugInFilter;
3  import ij.process.ByteProcessor;
4  import ij.process.ImageProcessor;
5
6  public class Create_New_Image implements PlugInFilter {
7    ImagePlus im;
8
9    public int setup(String arg, ImagePlus im) {
10     this.im = im;
11     return DOES_8G + NO_CHANGES;
12   }
13
14   public void run(ImageProcessor ip) {
15     // obtain the histogram of ip:
16     int[] hist = ip.getHistogram();
17     int K = hist.length;
18
19     // create the histogram image:
20     ImageProcessor hip = new ByteProcessor(K, 100);
21     hip.setValue(255);   // white = 255
22     hip.fill();
23
24     // draw the histogram values as black bars in hip here,
25     // for example, using hip.putpixel(u, v, 0)
26     // ...
27
28     // compose a nice title:
29     String imTitle = im.getShortTitle();
30     String histTitle = "Histogram of " + imTitle;
31
32     // display the histogram image:
33     ImagePlus him = new ImagePlus(title, hip);
34     him.show();
35   }
36 }
```