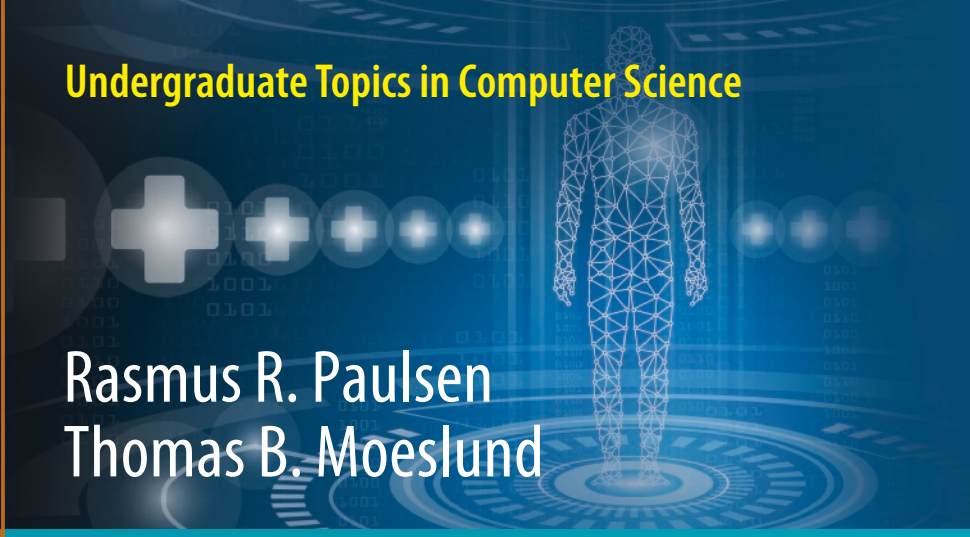
Undergraduate Topics in Computer Science



Rasmus R. Paulsen
Thomas B. Moeslund

Introduction to Medical Image Analysis

UTiCS

Springer

Image Acquisition

2

Before any image processing can commence an image must be captured by a camera and converted into a manageable entity. This is the process known as *image acquisition*. The image acquisition process consists of the steps shown in Fig. 2.1 and can be characterized by three major components; *energy* reflected from the object of interest, an *optical system* which focuses the energy and finally a *sensor* which measures the amount of energy. In Fig. 2.1 the three steps are shown for the case of an ordinary camera with the sun as the energy source. In this chapter, each of these three steps is described in more detail.

2.1 Energy

In order to capture an image, a camera requires some sort of measurable energy. The energy of interest in this context is light or more generally *electromagnetic waves*. An electromagnetic (EM) wave can be described as massless entity, a *photon*, whose electric and magnetic fields vary sinusoidally, hence the name wave. The photon belongs to the group of fundamental particles and can be described in three different ways:

- A photon can be described by its energy E, which is measured in electronvolts [eV]
- A photon can be described by its frequency f, which is measured in Hertz [Hz]. A frequency is the number of cycles or wave-tops in one second
- A photon can be described by its wavelength λ, which is measured in meters [m]. A wavelength is the distance between two wave-tops.

R. R. Paulsen and T. B. Moeslund, *Introduction to Medical Image Analysis*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-39364-9_2

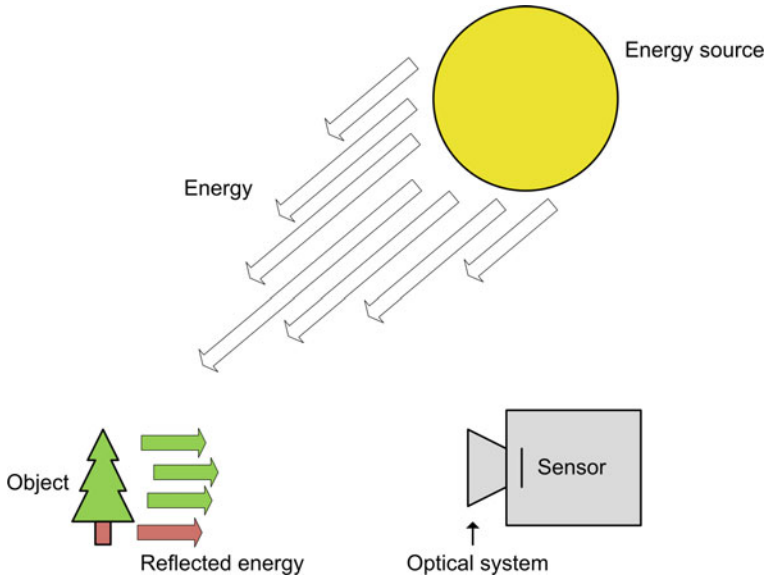


Fig. 2.1 Overview of the typical image acquisition process, with the sun as light source, a tree as object and a digital camera to capture the image. An analog camera would use a film where the digital camera uses a sensor

The three different notations are connected through the speed of light c and Planck's constant h:

$$\lambda = \frac{c}{f} \qquad E = h \cdot f \;\Rightarrow\; E = \frac{h \cdot c}{\lambda} \tag{2.1}$$

An EM wave can have different wavelengths (or different energy levels or different frequencies). When we talk about all possible wavelengths we denote this as the *EM spectrum*, see Fig. 2.2.

In order to make the definitions and equations above more understandable, the EM spectrum is often described using the names of the applications where they are used in practice. For example, when you listen to FM-radio the music is transmitted through the air using EM waves around $100 \cdot 10^6 Hz$, hence, this part of the EM spectrum is often denoted "radio". Other well-known applications are also included in Fig. 2.2.

The range from approximately 400–700 nm (nm = nanometer = 10^{-9}) is denoted the visual spectrum. The EM waves within this range are those your eye (and most cameras) can detect. This means that the light from the sun (or a lamp) in principle is the same as the signal used for transmitting TV, radio or for mobile phones, etc. The only difference, in this context, is the fact that the human eye can sense EM waves in this range and not the waves used, for example, radio. Or in other words, if our eyes were sensitive to EM waves with a frequency around $2 \cdot 10^9$ Hz, then your mobile phone would work as a flashlight, and big antennas would be perceived as

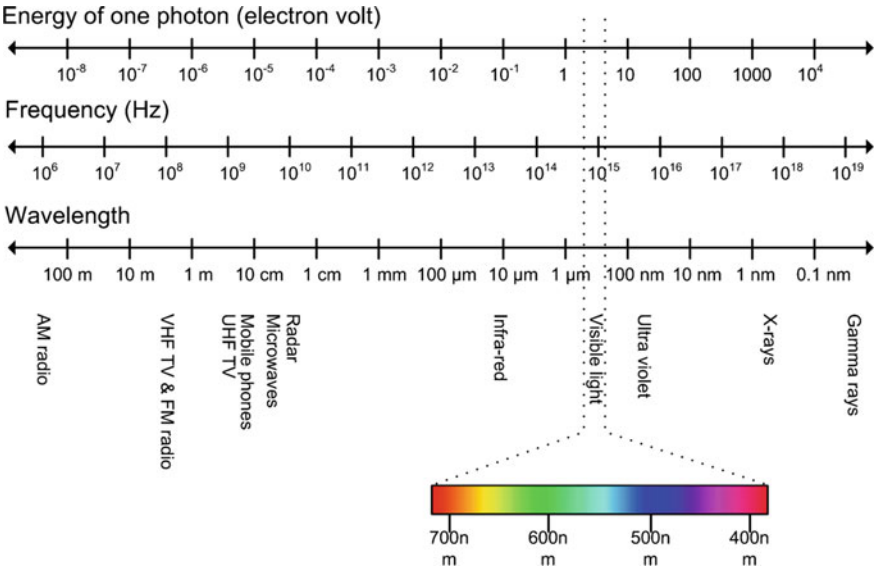


Fig. 2.2 A large part of the electromagnetic spectrum showing the energy of one photon, the frequency, wavelength and typical applications of the different areas of the spectrum

"small suns". Evolution has (of course) not made the human eye sensitive to such frequencies but rather to the frequencies of the waves coming from the sun, hence visible light.

2.1.1 Illumination

To capture an image we need some kind of energy source to illuminate the scene. In Fig. 2.1 the sun acts as the energy source. Most often we apply visual light, but other frequencies can also be applied. For example *x-ray*, which is used to see hard objects within something: bones inside your body or guns in your luggage in airports. Another example is infrared frequencies which are used to see heat, for example, by a rescue helicopter to see people in the dark or by animals hunting at nights. All the concepts, theories, and methods presented throughout this text are general and apply to virtually all frequencies. However, we shall hereafter focus on visual light and denote the energy "light".

If you are processing images captured by others there is nothing much to do about the illumination (although a few methods will be presented in later chapters) which was probably the sun and/or some artificial lighting. When you, however, are in charge of the capturing process yourselves, it is of great importance to carefully think about how the scene should be lit. In fact, for the field of Machine Vision it is a rule-of-thumb that illumination is 2/3 of the entire system design and software only 1/3. To stress this point have a look at Fig. 2.3. The figure shows four images of the

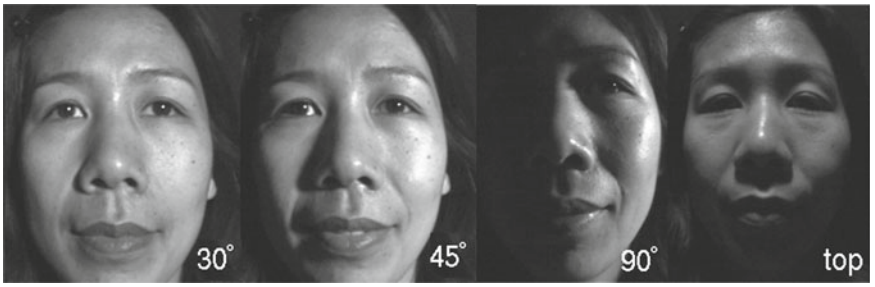


Fig. 2.3 The effect of illuminating a face from four different directions

Fig. 2.4 Backlighting. The light source is behind the object of interest, which makes the object stand out as a black silhouette. Note that the details inside the object are lost

same person facing the camera. The only difference between the four images is the direction of the light source (a lamp) when the images were captured!

Another issue regarding the direction of the illumination is that care must be taken when pointing the illumination directly toward the camera. The reason being that this might result in too bright an image or a nonuniform illumination, e.g., a bright circle in the image. If however the outline of the object is the only information of interest, then this way of illumination—denoted backlighting—can be an optimal solution, see Fig. 2.4. Even when the illumination is not directed toward the camera overly bright spots in the image might still occur. These are known as *highlights* and are often a result of a shiny object surface, which reflects most of the illumination (similar to the effect of a mirror). A solution to such problems is often to use some kind of diffuse illumination either in the form of a high number of less-powerful light sources or by illuminating a rough surface which then reflects the light (randomly) toward the object.

Even though this text is about visual light as the energy form, it should be mentioned that infrared illumination is sometimes useful. For example, when tracking the movements of human body parts, e.g., for use in animations in motion pictures, infrared illumination is often applied. The idea is to add infrared reflecting markers to the human body parts, e.g., in the form of small balls. When the scene is illuminated by infrared light, these markers will stand out and can therefore easily be detected by image processing.

Pages removed ….

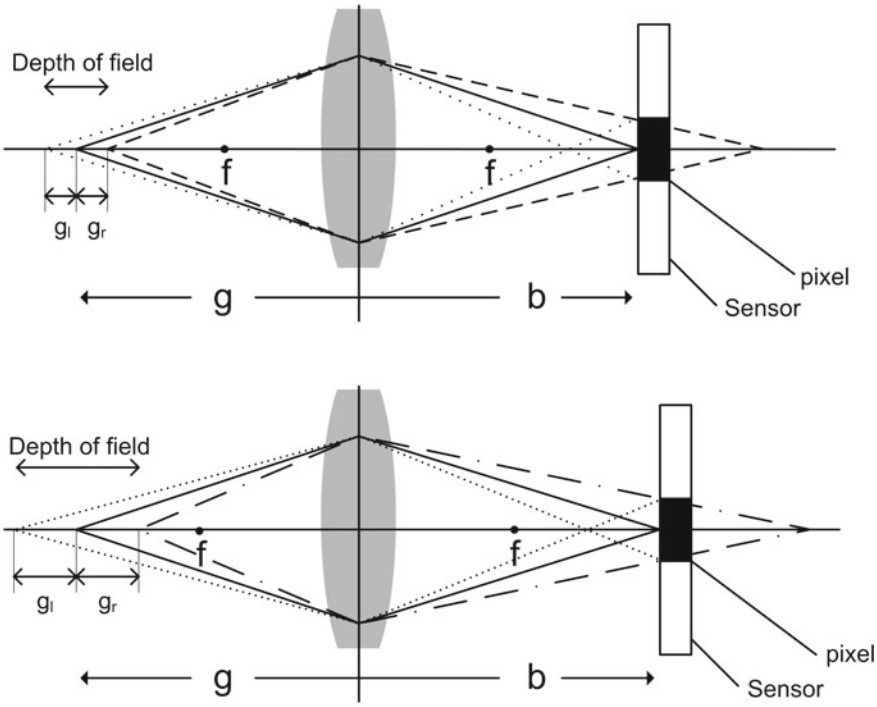


Fig. 2.9 Depth-of-field for two different focal lengths. The solid lines illustrate two light rays from an object (a point) on the optical axis and their paths through the lens and to the sensor where they intersect within the same pixel (illustrated as a black rectangle). The dashed and dotted lines illustrate light rays from two other objects (points) on the optical axis. These objects are characterized by being the most extreme locations where the light rays still enter the same pixel

To sum up, the following interconnected issues must be considered: distance to object, motion of object, zoom, focus, depth-of-field, focal length, shutter, aperture and sensor. In Figs. 2.11 and 2.12 some of these issues are illustrated. With this knowledge, you might be able to appreciate why a professional photographer can capture better images than you can!

2.3 The Imaging Sensor

The light reflected from the object of interest is focused on some optics and now needs to be recorded by the camera. For this purpose, an image sensor is used. An image sensor consists of a 2D array of cells as shown in Fig. 2.13. Each of these cells is denoted a pixel and is capable of measuring the amount of incident light and convert that into a voltage, which in turn is converted into a digital number.

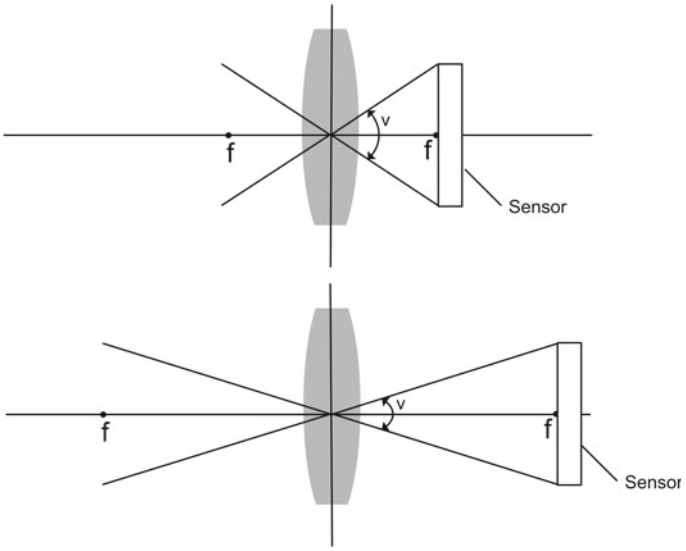


Fig. 2.10 The field-of-view of two cameras with different focal lengths. The field-of-view is an angle, which represents the part of the world observable to the camera. As the focal length increase so does the distance from the lens to the sensor. This in turn results in smaller field-of-view. Note that both a horizontal field-of-view and a vertical field-of-view exist. If the sensor has equal height and width these two fields-of-view are the same, otherwise they are different

Fig. 2.11 Three different camera settings resulting in three different depth-of-fields

The more incident light the higher the voltage and the higher the digital number. Before a camera can capture an image, all cells are emptied, meaning that no charge is present. When the camera is to capture an image, light is allowed to enter and charges start accumulating in each cell. After a certain amount of time, known as the *exposure time*, and controlled by the *shutter*, the incident light is shut out again. If the exposure time is too low or too high the result is an underexposed or overexposed image, respectively, see Fig. 2.14.

Furthermore, if the object of interest is in motion the exposure time in general needs to be low in order to avoid *motion blur*, where light from a certain point on the object will be spread out over more cells.

The accumulated charges are converted into digital form using an *analog-to-digital converter*. This process takes the continuous world outside the camera and

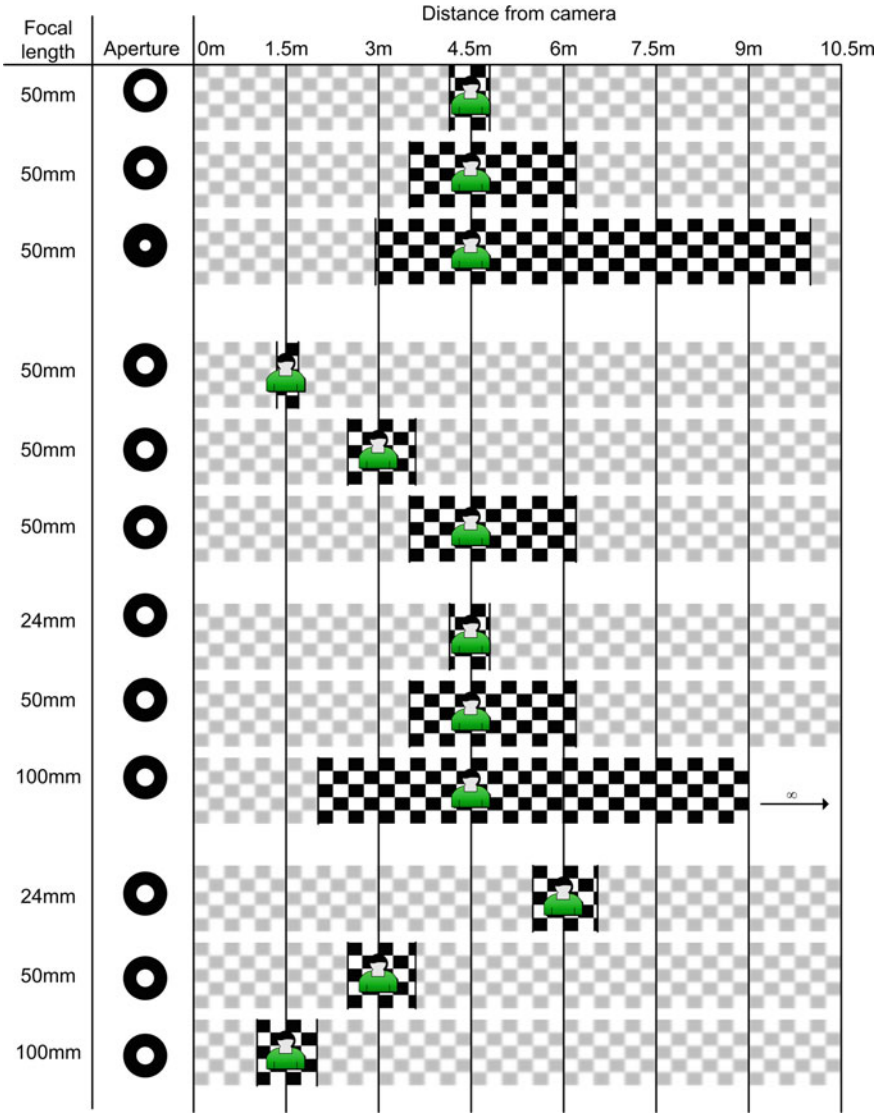


Fig. 2.12 Examples of how different settings for focal length, aperture, and distance to object result in different depth-of-fields. For a given combination of the three settings, the optics are focused so that the object (person) is in focus. The focused checkers then represent the depth-of-field for that particular setting, i.e., the range in which the object will be in focus. The figure is based on a Canon 400D

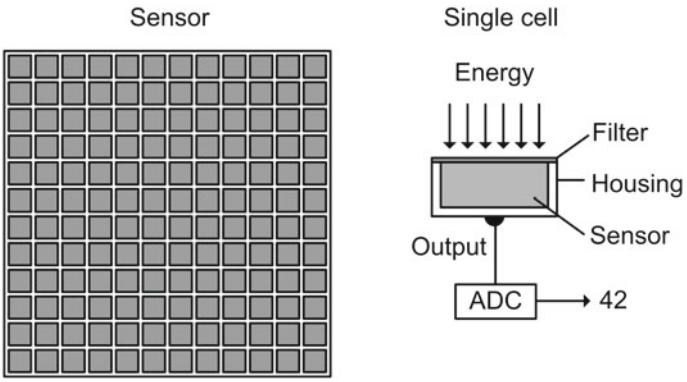


Fig. 2.13 The sensor consists of an array of interconnected cells. Each cell consists of a housing which holds a filter, a sensor, and an output. The filter controls which type of energy is allowed to enter the sensor. The sensor measures the amount of energy as a voltage, which is converted into a digital number through an Analog-To-Digital Converter (ADC)

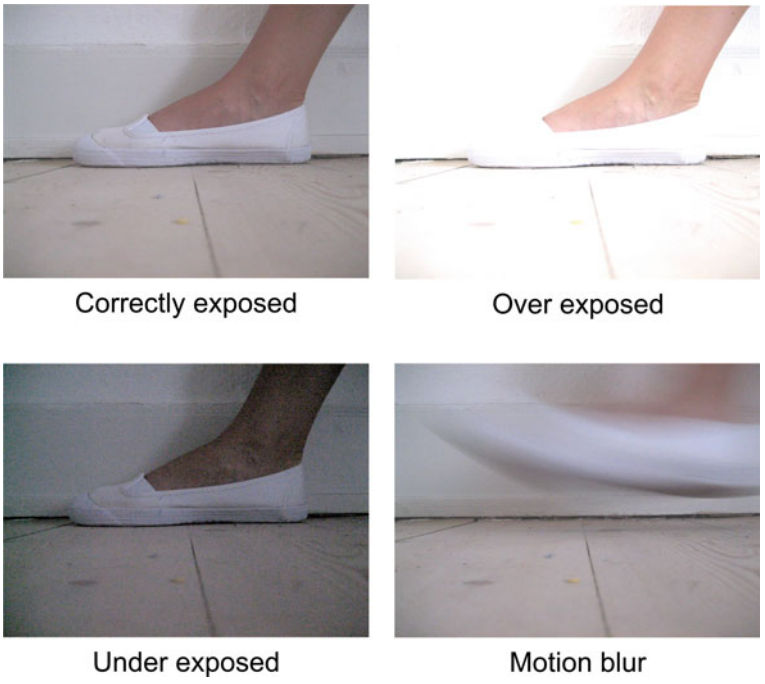


Fig. 2.14 The input image was taken with the correct amount of exposure. The over- and underexposed images are too bright and too dark, respectively, which makes it hard to see details in them. If the object or camera is moved during the exposure time, it produces motion blur as demonstrated in the last image

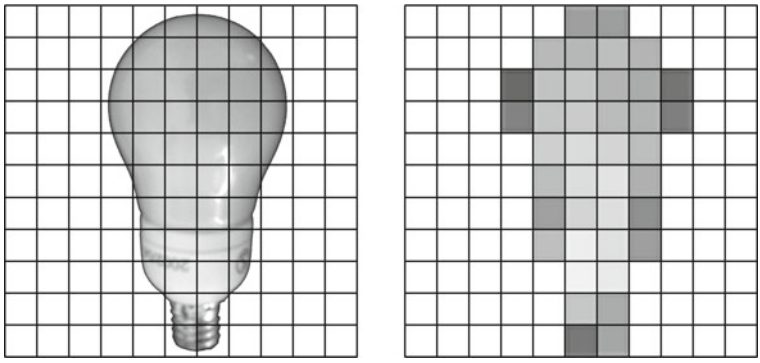


Fig. 2.15 To the left the amount of light which hits each cell is shown. To the right the resulting image of the measured light is shown

converts it into a digital representation, which is required when stored in the computer. Or in other words, this is where the image becomes digital. To fully comprehend the difference, have a look at Fig. 2.15.

To the left we see where the incident light hits the different cells and how many times (the more times the brighter the value). This results in the shape of the object and its intensity. Let us first consider the shape of the object. A cell is sensitive to incident light hitting the cell, but not sensitive to where exactly the light hits the cell. So if the shape should be preserved, the size of the cells should be infinitely small. From this it follows that the image will be infinitely large in both the x- and y-direction. This is not tractable and therefore a cell, of course, has a finite size. This leads to loss of data/precision and this process is termed *spatial quantization*. The effect is the blocky shape of the object in the figure to the right. The number of pixels used to represent an image is also called the *spatial resolution* of the image. A high resolution means that a large number of pixels are used giving fine details in the image. A low resolution means that a relatively low number of pixels is used. Sometimes the words fine and coarse resolution are used. The visual effect of the spatial resolution can be seen in Fig. 2.16. Overall we have a trade-off between memory and shape preservation. It is possible to change the resolution of an image by a process called *image-resampling*. This can be used to create a low-resolution image from a high-resolution image. However, it is normally not possible to create a high-resolution image from a low-resolution image.

A similar situation is present for the representation of the amount of incident light within a cell. The number of photons hitting a cell can be tremendously high requiring an equally high digital number to represent this information. However, since the human eye is not even close to being able to distinguish the exact number of photons, we can quantify the number of photons hitting a cell. Often this quantization results in a representation of one Byte (8 bits), since one byte corresponds to the way memory is organized inside a computer (see Appendix A for an introduction to bits and bytes). In the case of 8-bit quantization, a charge of 0 volt will be quantized to

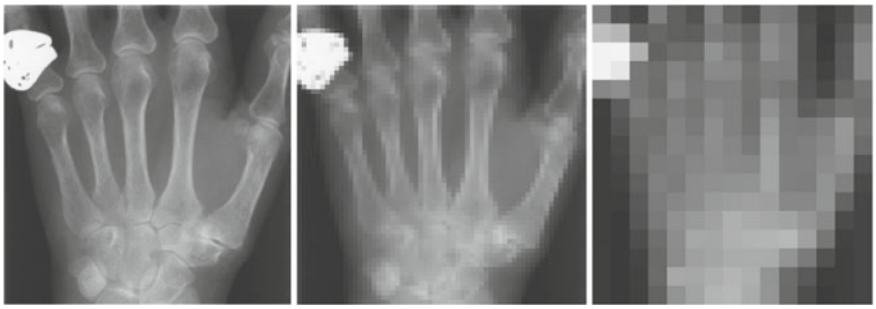


Fig. 2.16 The effect of spatial resolution demonstrated on an X-ray image of a hand (notice the ring). The spatial resolution is from left to right: 256×256, 64×64, and 16×16

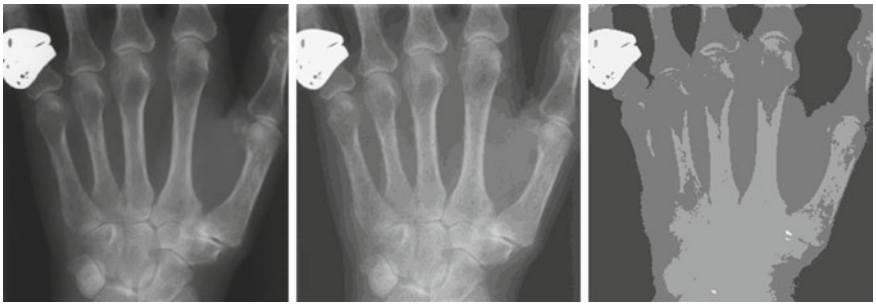


Fig. 2.17 The effect of gray-level resolution demonstrated on an X-ray image of a hand. The gray-level resolution is from left to right: 256, 16, and 4 gray levels

0 and a high charge quantized to 255. Other gray-level quantizations are sometimes used. The effect of changing the gray-level quantization (also called the gray-level resolution) can be seen in Fig. 2.17. With more than 64 gray levels, the visual effect is rather small. Down to 16 gray levels the image will frequently still look realistic, but with a clearly visible quantization effect. Maybe surprisingly the image can still be visually interpreted with only four gray levels. The gray-level resolution is usually specified in number of bits. While, typical gray-level resolutions are 8-, 10-, and 12-bit corresponding to 256, 1024, and 4096 gray levels, 8-bit images are the most common and are the topic of this text (unless otherwise noted).

In the case of an overexposed image, a number of cells might have charges above the maximum measurable charge. These cells are all quantized to 255. There is no way of knowing just how much incident light entered such a cell and we therefore say that the cell is *saturated*. This situation should be avoided by setting the shutter (and/or aperture), and saturated cells should be handled carefully in any image processing system. When a cell is saturated it can affect the neighbor pixels by increasing their changes. This is known as *blooming* and is yet another argument for avoiding saturation.

2.4 Digital Images

To transform the information from the sensor into an image, each cell content is now converted into a pixel value in the range: [0, 255]. Such a value is interpreted as the amount of light hitting a cell during the exposure time. This is denoted the *intensity* of a pixel. It is visualized as a shade of gray denoted a *gray-scale value* or *gray-level value* ranging from black (0) to white (255), see Fig. 2.18.

A gray-scale image (as opposed to a color image, which is the topic of Chap. 8) is a 2D array of pixels (corresponding to the 2D array of cells in Fig. 2.13) each having a number between 0 and 255. A frequent representation of digital images is as matrices or as discrete function values over a rectangular two-dimensional region. An 8-bit gray-scale image is represented as a two-dimensional matrix with M rows and N columns, where each element contains a number between 0 and 255.

2.4.1 Image Coordinate Systems

Unfortunately, several coordinates systems are used for images. In MATLAB where images are considered matrices, pixel coordinates are specified as (row, column) with (1, 1) being placed in the upper left corner. A pixel value will often be written as $v(r, c)$ or just as v, where r is the row number and c is the column number. In some textbooks and systems (x, y) with (0, 0) in the upper left corner is used. Finally, the standard mathematical (x, y) system with (0, 0) in the lower left corner can also be used. The three different systems can be seen in Fig. 2.19.

Systems that uses (0, 0) as origin are said to be 0-based and systems that use (1, 1) are 1-based . An overview of which systems different software packages use can be found in Table 2.1.

In this text, the coordinate 0-based system with (0, 0) in the upper left corner is used, unless otherwise specified. The image is represented as *f(x, y)*, where x is the horizontal position of the pixel and y the vertical position. For the small image in Fig. 2.19 to the left, *f(0, 0) = 23*, *f(3, 1) = 130*, and *f(2, 3) = 7*. It can also be seen that the pixel with value $v = 158$ is placed at $(r, c) = (2, 3)$ in the MATLAB coordinate system, in Photoshop it can be found at $(x, y) = (2, 1)$, and in the mathematical systems at $(x, y) = (2, 2)$.

The conversion between, for example, MATLAB (r, c) and Photoshop coordinates (x, y) is done by $(x, y) = (c - 1, r - 1)$ and $(r, c) = (y + 1, x + 1)$.

So whenever you see a gray-scale image you must remember that what you are actually seeing is a 2D array of numbers as illustrated in Fig. 2.20.

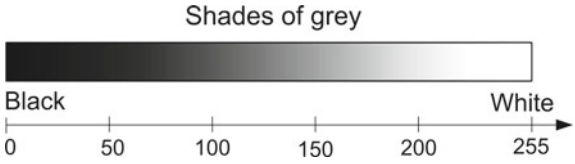


Fig. 2.18 The relationship between the intensity values and the different shades of gray

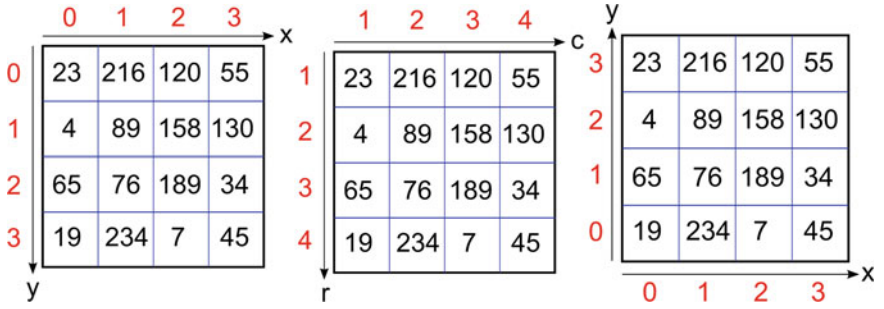


Fig. 2.19 Pixel coordinate systems. The left is the one used by, for example, Photoshop and GIMP. The middle is the standard matrix system used by MATLAB. The right is a standard mathematical coordinate system

Table 2.1 Coordinate systems used by different software packages

Software	Origin	System	Base
MATLAB images	Upper left corner (1, 1)	(r, c)	1-based
MATLAB matrices	Upper left corner (1, 1)	(r, c)	1-based
C and C++	Upper left corner (0, 0)	(x, y)	0-based
VXL (C++)	Upper left corner (0, 0)	(x, y)	0-based
Photoshop	Upper left corner (0, 0)	(x, y)	0-based
GIMP	Upper left corner (0, 0)	(x, y)	0-based
MATLAB plotting	Lower left corner (0, 0)	(x, y)	0-based
Many plotting packages	Lower left corner (0, 0)	(x, y)	0-based

2.4.2 The Region-of-Interest (ROI)

As digital cameras are sold in larger and larger numbers the development within sensor technology has resulted in many new products including larger and larger numbers of pixels within one sensor. This is normally defined as the size of the image that can be captured by a sensor, i.e., the number of pixels in the vertical direction multiplied by the number of pixels in the horizontal direction. Having a large number of pixels can result in high quality images and has made, for example, digital zoom a reality.

When it comes to image processing, a larger image size is not always a benefit. Unless you are interested in tiny details or require very accurate measurements in the image, you are better off using a smaller sized image. The reason being that when we start to process images we have to process each pixel, i.e., perform some math on each pixel. And, due to the large number of pixels, that quickly adds up to quite a large number of mathematical operations, which in turn means a high computational load on your computer.

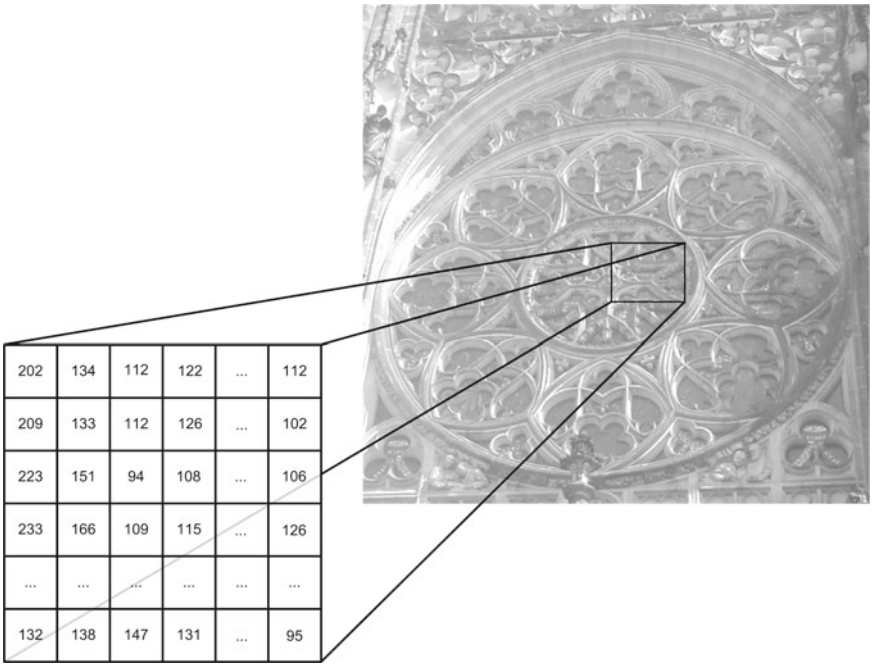


Fig. 2.20 A gray-scale image and part of the image described as a 2D array, where the cells represent pixels and the value in a cell represents the intensity of that pixel

Say you have an image which is 500×500 pixels. That means that you have $500 \cdot 500 = 250.000$ pixels. Now say that you are processing video with 50 images per second. That means that you have to process $50 \cdot 250.000 = 12.500.000$ pixels per second. Say that your algorithm requires 10 mathematical operations per pixel, then in total your computer has to do $10 \cdot 12.500.000 = 125.000.000$ operations per second. That is, quite a number even for today's powerful computers. So when you choose your camera do not make the mistake of thinking that bigger is always better!

Besides picking a camera with a reasonable size you should also consider introducing a *Region-Of-Interest* (ROI). A ROI is simply a region (normally a rectangle) within the image which defines the pixels of interest. Those pixels not included in the region are ignored altogether and less processing is required. An ROI is illustrated in Fig. 2.21.

The ROI can sometimes be defined for a camera, meaning that the camera only captures those pixels within the region, but usually it is something you as a designer define in software. Say that you have put up a camera in your home in order to detect if someone comes through one of the windows while you are on holiday. You could then define an ROI for each window seen in the image and *only* process these pixels. When you start playing around with image processing you will soon realize the need for an ROI.

Fig. 2.21 The rectangle defines a Region-Of-Interest (ROI), i.e., this part of the image is the only one being processed

2.4.3 Binary Images

In a binary image, there are only two possible pixel values. Unless otherwise noted we shall talk about 0-pixels and 1-pixels. Often the set of 0 value pixels are interpreted as the background and the 1 value pixels as the foreground. The foreground will normally consist of a number of connected components (or BLOBS, see Chap. 7) that represent objects. The background is sometimes called the 0-phase and the foreground the 1-phase. Binary images can be displayed using two colors, for example, black and white. Binary images are typically a result of thresholding a gray-level image as will be described in Sect. 4.4. A binary image can be seen in Fig. 2.22 in the middle, where a computed tomography image of the human brain has been thresholded so the bone pixels constitute the foreground object.

2.4.4 Label Images

Occasionally, another image representation is used, where the pixel value does not represent a light intensity but is a number that tells something about the pixel. A typical example is that the pixel value tells which object the pixel belongs to. If an image contains a man and a dog, the pixel values could be 0 for pixels belonging to the background, 1 for pixels that constitute the man, and 2 for dog pixels. Here we say that the pixels have been labeled with three label values and the image is called a label image. Label images are typically the result of image segmentation, BLOB analysis, or pixel classification algorithms. In Fig. 2.22 to the right a label image can be seen. The different tissue types in the human head have been classified in a computed tomography image. The labels are visualized using colors, where the gray pixels are the background, the blue pixels belong to the soft tissue, and the green and the red pixels belong to two different types of bones.

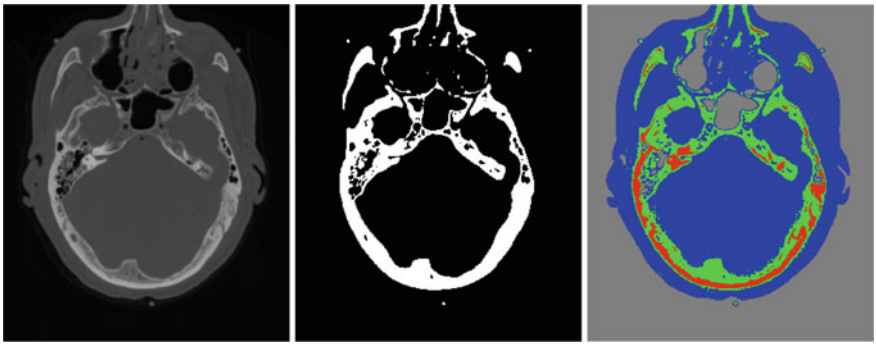


Fig. 2.22 A computed tomography scan of the human head. To the left it is seen as a gray-scale image, in the middle it has been transformed into a binary image, and to the right a label image has been created (gray = background, blue = soft tissue, green and red = bone)

2.4.5 Multi-spectral Images

In a gray-level image there is one measurement (the intensity) per pixel and in color images there are three values per pixel (the red, green, and blue. See Chap. 8). However, it can be useful to measure several intervals of the electromagnetic spectrum individually and in this way assign more than three values per pixel. An example is the addition of near-infrared information to a pixel value. Images containing information captured at multiple spectral wavelengths are called *multi-spectral images* or *multi-channel images*. Images captured by satellites are often multi-spectral. Furthermore, there is extensive research in multi-spectral cameras for diagnostic purposes as, for example, early skin cancer diagnostics.

2.4.6 16-Bit Images

While the human visual system does not need more than 256 different gray levels in an image, a computer can gain more information by having a large pixel value range available. An example is modern medical scanners, where the pixel value does not represent the intensity of incoming light, but an underlying physical measurement. In Computed Tomography (CT) scanners, the pixel value is typically measured in Hounsfield Units (HU) that describe the attenuation of X-rays in the tissue. The Hounsfield unit is defined so a pixel that describes air has a value of 1000 and a pixel describing water has a value of 0. A typical bone pixel will have a value of 400. Obviously, a single byte with 256 different values cannot be used to store these pixel values (see Appendix A for an introduction to bits and bytes). Instead images where each pixel value is stored as two bytes are used. Two bytes equal 16 bits and this type of images is therefore called 16-bit images. A CT scanner normally creates 16-bit images. Certain modern cameras create 10-bit (1024 gray levels) or 12-bit (4096 gray levels) images, where only 10 or 12 of the 16 bits are used, but the pixel values are still stored as two bytes.

Neighborhood Processing

5

In the previous chapter we saw that a pixel value in the output was set according to a pixel value in the input *at the same position* and a point processing operation. This principle has many useful applications (as we saw), but it cannot be applied to investigate the relationship between *neighboring pixels*. For example, if we look at the pixel values in the small area in Fig. 5.1, we can see that a significant change in intensity values occurs in the lower left corner. This could indicate the boundary of an object and by finding the boundary pixels we have found the object.

In this and the next chapter, we present a number of methods where the neighbor pixels play a role when determining the output value of a pixel. Overall these methods are denoted *neighborhood processing* and the principle is illustrated in Fig. 5.2. The value of a pixel in the output is determined by the value of the pixel at the same position in the input *and* the neighbors together with a neighborhood processing operation. We use the same notation as in the previous chapter, i.e., $f(x, y)$ is the input image and $g(x, y)$ is the output image.[1]

5.1 The Median Filter

If we look at Fig. 5.3, we can see that it has been infected with some kind of noise (the black and white dots). Let us set out to remove this noise. First of all, we zoom in on the image and look closer at particular pixel values. What we can see is that the noise is isolated pixels having a value of either 0 (black) or 255 (white), such

[1] Readers unfamiliar with vectors and matrices are advised to consult Appendix B before reading this chapter.

R. R. Paulsen and T. B. Moeslund, *Introduction to Medical Image Analysis*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-39364-9_5

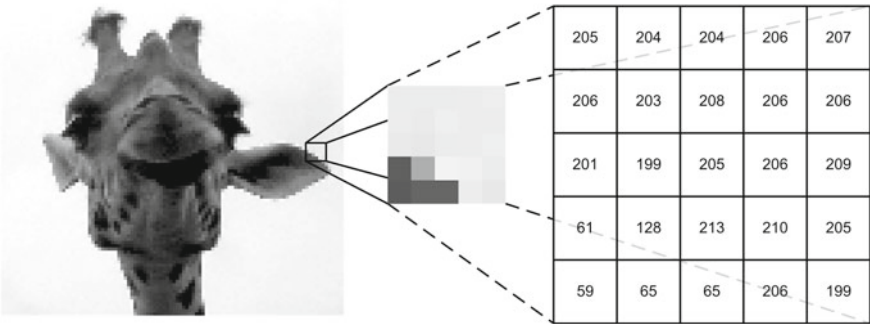


Fig. 5.1 A part of the giraffe image has been enlarged to show the edge which humans easily perceive. Using methods described in this chapter the computer will also be able to tell where the edge is

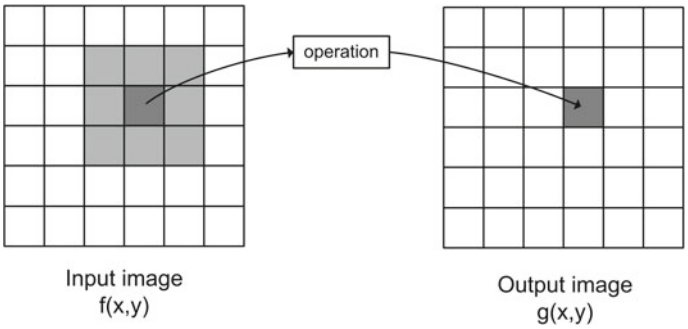


Fig. 5.2 The principle of neighborhood processing. To calculate a pixel in the output image, a pixel from the input image and its neighbors are processed

noise is denoted *salt-and-pepper noise*. By *isolated* we mean that they have a value very different from their neighbors. We need somehow to identify such pixels and replace them by a value which is more similar to the neighbors.

One solution is to replace the noise pixel by the *mean* value of the neighbors. Say we use the eight nearest neighbors for the noise pixel at position (1, 1) in the image patch in Fig. 5.3. To not lose the information, the actual pixel is also included. The mean value is then (using nine pixels in total):

$$\text{Mean value} = \frac{205 + 204 + 204 + 206 + 0 + 208 + 201 + 199 + 205}{9} = 181.3$$

This results in the noise pixel being replaced by 181 (since we convert the value 181.3 to the integer value 181), which is more similar to the neighbors. However, the value still stands out and therefore the *median* is often used instead. The median value of a group of numbers is found by ordering the numbers in increasing order

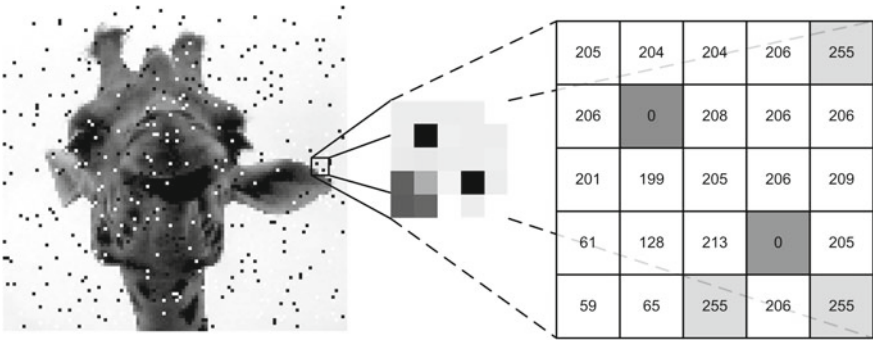


Fig. 5.3 An image infected with salt-and-pepper noise. The noise is easily recognized in both the image and the number representation

and picking the middle value. Say we use the eight nearest neighbors for the first pixel infested by noise in Fig. 5.3 (and also include the pixel itself). The ordering yields

$$\text{Ordering} : [0, 199, 201, 204, \underline{204}, 205, 205, 206, 208] \qquad \text{Median} = 204$$

and the middle value is 204; hence, the median is 204. The noise pixel is now replaced by 204, which does not stand out.

The next question is how to find the noise pixels in order to know where to perform the median operation. For the particular example, we could scan the image pixel-by-pixel and look for isolated values of 0 or 255. When encountered, the median operation could be applied. In general, however, a pixel with a value of say 234 could also be considered noise if it is isolated (stands out from its neighbors). Therefore, the median operation is applied to every single pixel in the image and we call this *filtering the image* using a *median filter*. Filtering the image refers to the process of applying a filter (here the median filter) to the entire image. It is important to note that by filtering the image we apply the filter to each and every pixel.

When filtering the image we of course need to decide which operation to apply but we also need to specify the size of the filter. The filter used in Fig. 5.2 is a 3×3 filter. Since filters are centered on a particular pixel (the center of the filter) the size of the filter is uneven, i.e., 3, 5, 7, etc. Very often filters have equal spatial dimensions, i.e., 3×3, 5×5, 7×7, etc. Sometimes a filter is described by its radius rather than its size. The radius of a 3×3 filter is 1, 2 for a 5×5 filter, 3 for a 7×7 filter, etc. The radius/size of a filter controls the number of neighbors included. The more neighbors included, the more strongly the image is filtered. Whether this is desirable or not depends on the application. Note that the larger the size, the more processing power is required by the computer. Applying a filter to an image is done by scanning through the image pixel-by-pixel from the upper left corner toward the lower right corner, as described in the previous chapter. Figure 5.4 shows how the

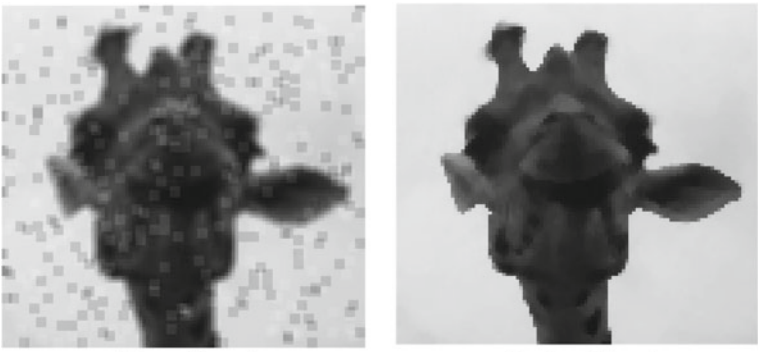


Fig. 5.4 Resulting images of two noise filters. Notice that the mean filter does not remove all the noise and that it blurs the image. The median filter removes all the noise and only blurs the image slightly

image in Fig. 5.3 is being filtered by a 3×3 (radius = 1) mean and median filter, respectively. Note the superiority of the median filter.

In terms of programming, the Median filter can be implemented as illustrated below—here exemplified in C-code[2]:

Implementation of the median filter

```
for (y = Radius; y < (M - Radius); y = y + 1)
{
    for (x = Radius; x < (N - Radius); x = x + 1)
    {
        GetPixelValues(x, y);
        SortPixelValues();
        value = FindMedian();
        SetPixel(output, x, y, value);
    }
}
```

Here `M` is the height of the image, `N` is the width of the image and `Radius` is the radius of the filter. The function `GetPixelValues` receives an array of pixel values in the area defined by `Radius around` the pixel with coordinates `(x,y)`.

What should be noticed both in the figure and in the code is that the output image will be smaller than the input image. The reason is that the filter is defined with a center and a radius, but if the center is a pixel in, for example, the first row, then no

[2]This implementation is just an example. The median filter can be implemented much more efficient.

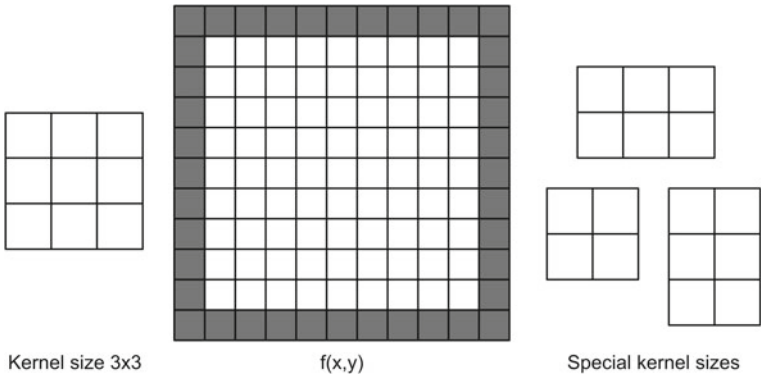


Fig. 5.5 An illustration of the border problem, which occurs when using neighborhood processing algorithms. If a kernel with a size of 3×3 is used, then the border illustrated in $f(x, y)$ cannot be processed. One solution to this is to apply kernels with special sizes on the borders, like the ones showed to the right

neighbors are defined above. This is known as the *border problem*, see Fig. 5.5. If it is unacceptable that the output image is reduced in size (for example, because it is to be added to the input image), then inspiration can be found in one of the following suggestions[3]:

Increase the output image	After the output image has been generated, the pixel values in the last row (if radius = 1) are duplicated and appended to the image. The same for the first row, first column, and last column.
Increase the input image	Before the image is filtered, the pixel values in the last row (if radius = 1) of the input image are duplicated and appended to the input image. The same for the first row, first column, and last column.
Apply special filters at the rim of the image	Special filters with special sizes are defined and applied accordingly, see Fig. 5.5.

[3]Note that this issue is common for all neighborhood processing methods.

5.1.1 Rank Filters

The Median filter belongs to a group of filters known as *Rank Filters*. The only difference between them is the value which is picked after the pixels have been sorted:

The minimum value	This filter will make the image darker.
The maximum value	This filter will make the image brighter.
The difference	This filter outputs the difference between the maximum and minimum value and the result is an image where the transitions between light and dark (and opposite) are enhanced. Such a transition is often denoted an edge in an image. More on this in Sect. 5.2.2.

5.2 Correlation

Correlation is an operation which also works by scanning through the image and applying a filter to each pixel. In correlation, however, the filter is denoted a *kernel* and plays a more active role. First of all the kernel is filled by numbers—denoted *kernel coefficients*. These coefficients weight the pixel value they are covering and the output of the correlation is a sum of weighted pixel values. In Fig. 5.6 some different kernels are shown.

Similar to the median filter the kernel is centered above the pixel position whose value we are calculating. We denote this center (0, 0) in the kernel coordinate system and the kernel as $h(x, y)$, see Fig. 5.7. To calculate the output value we take the value of $h(-1, -1)$ and multiply it by the pixel value beneath. Let's say that we are calculating the output value of the pixel at position (2, 2). Then $h(-1, -1)$ will be above the pixel $f(1, 1)$ and the values of these two pixels are multiplied together. The

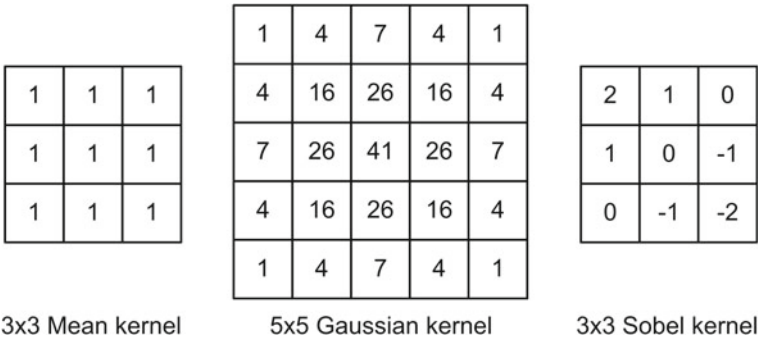


Fig. 5.6 Three different kernels

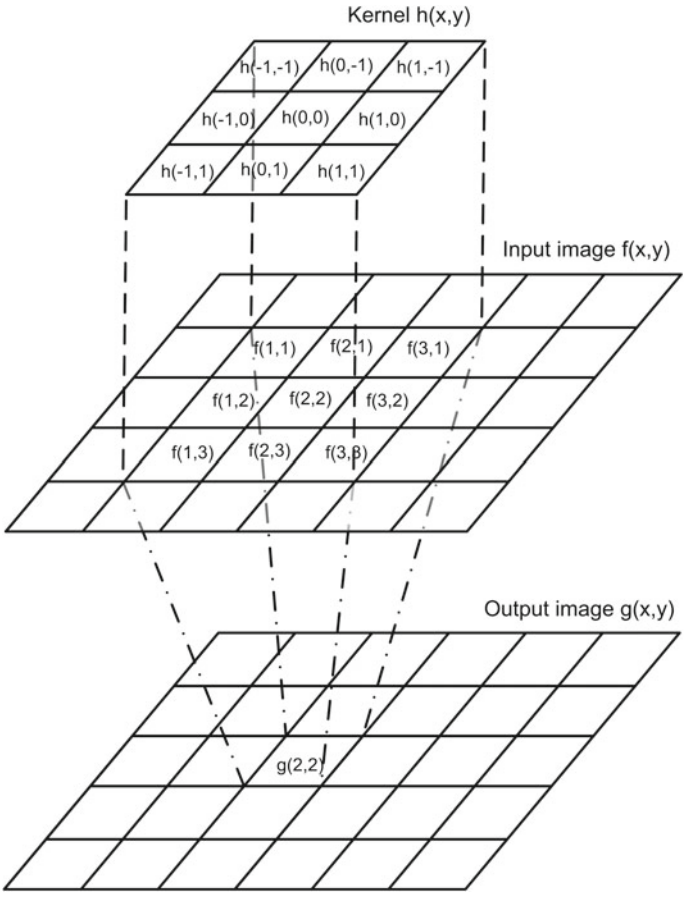


Fig. 5.7 The principle of correlation, illustrated with a 3×3 kernel on a 6×6 image

result is added to the product of the next kernel element $h(0, -1)$ and the pixel value beneath $f(2, 1)$, etc. The final value which will be written into the output image as $g(2, 2)$ is found as

$$\begin{aligned} & g(2,2) = \\ & \quad h(-1,-1) \cdot f(1,1) + h(0,-1) \cdot f(2,1) + h(1,-1) \cdot f(3,1) + \\ & \quad h(-1,0) \cdot f(1,2) + h(0,0) \cdot f(2,2) + h(1,0) \cdot f(3,2) + \\ & \quad h(-1,1) \cdot f(1,3) + h(0,1) \cdot f(2,3) + h(1,1) \cdot f(3,3) \end{aligned} \tag{5.1}$$

The principle is illustrated in Fig. 5.7. We say that we correlate the input image $f(x, y)$ with the kernel $h(x, y)$ and the result is $g(x, y)$. Mathematically this is expressed as $g(x, y) = f(x, y) \circ h(x, y)$ and written as

$$g(x,y) = \sum_{j=-R}^{R} \sum_{i=-R}^{R} h(i,j) \cdot f(x+i, y+j), \tag{5.2}$$

where R is the radius of the kernel.[4] Below, a C-code example of how to implement correlation is shown:

Implementation of correlation

```
for (y = Radius; y < (M - Radius); y = y + 1)
{
    for (x = Radius; x < (N - Radius); x = x + 1)
    {
        temp = 0;
        for (j = -Radius; j < (Radius + 1); j = j + 1)
        {
            for (i = -Radius; i < (Radius+1); i = i + 1)
            {
                temp = temp +
                       h(i,j) * GetPixel(input,x+i,y+j);
            }
        }
        SetPixel(output, x, y, temp);
    }
}
```

When applying correlation, the values in the output can be above 255. If this is the case, then we normalize the kernel coefficients so that the maximum output of the correlation operation is 255. The normalization factor is found as the sum of the kernel coefficients. That is,

$$\sum_x \sum_y h(x, y). \tag{5.3}$$

For the left-most kernel in Fig. 5.6 the normalization factor becomes $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 9$ and the resulting kernel coefficients are $1/9$ as opposed to 1.

Looking back on the previous section, we can now see that the left-most kernel in Fig. 5.6 is exactly the mean filter. The mean filter smoothes or blurs the image which has different applications. In Fig. 5.8 one application is shown where the mean filter is applied within the white box in order to hide the identity of a person. The bigger the kernel, the more the smoothing. Another type of mean filter is when a kernel like the middle one in Fig. 5.6 is applied. This provides higher weights to pixels close to the center of the kernel. This mean filter is known as a *Gaussian filter*, since the kernel coefficients are calculated from the Gaussian distribution (a bell-shaped curve).

[4] The reader is encouraged to play around with this equation in order to fully comprehend it.

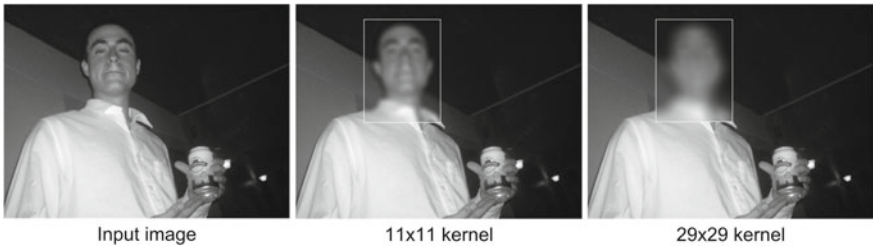


Fig. 5.8 An example of how a mean filter can be used to hide the identity of a person. The size of the mean kernel decides the strength of the filter. Actual image size: 512 × 384

5.2.1 Template Matching

An important application of correlation is *template matching*. Template matching is used to locate an object in an image. When applying template matching the kernel is denoted a *template*. It operates by defining an image of the object we are looking for. This object is now the template (kernel) and by correlating an image with this template, the output image indicates where the object is. Each pixel in the output image now holds a value, which states the similarity between the template and an image patch (with the same size as the template) centered at this particular pixel position. The brighter a value the higher the similarity.

In Fig. 5.9 the correlation-based template matching is illustrated. We can see a bright spot in the center of the upper part of the output corresponding to where the template matches best. Note also that as the template is shifted left and right with respect to this position, a number of bright spots appear. The distances between these spots correspond to the distance between the letters in the text.

Since correlation is based on multiplying the template and the input image, bright areas in the input image tend to produce high values in the output. This is illustrated in Fig. 5.10 where the large white section in the clothing of the child in the middle

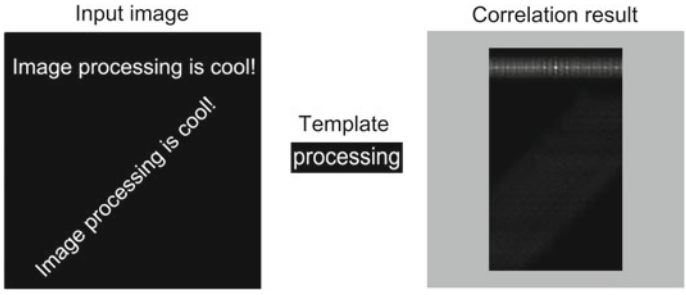


Fig. 5.9 Template matching performed by correlating the input image with a template. The result of template matching is seen to the right. The gray outer region illustrates the pixels that cannot be processed due to the border problem

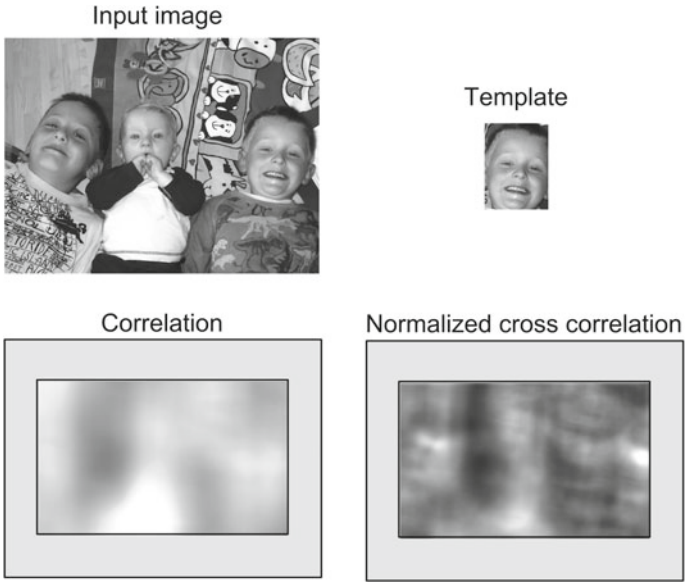


Fig. 5.10 Template matching using correlation and normalized cross correlation. The gray regions illustrate the pixels that cannot be processed due to the border problem

produces the highest values in the output. This problem in general makes it difficult, and in this particular case impossible, to actually find the position of the object by looking at the values in the output image.

To avoid this problem we need to normalize the values in the output so they are independent of the overall level of light in the image. To assist us in doing so we use a small trick. Let us denote the template H and the image patch F. These are both matrices, but by rearranging we can easily convert each matrix into a vector by concatenating each row (or column) in the matrix, i.e., $\vec{H}$ and $\vec{F}$.

If we now look at correlation in terms of this vector representation, we can see that Eq. 5.2 is actually the dot product between the two vectors, see Appendix B. From geometry we know that the dot product between two vectors can be normalized to the interval $[-1, 1]$ using the follow equation:

$$\cos\theta = \frac{\vec{H} \bullet \vec{F}}{|\vec{H}| \cdot |\vec{F}|}, \tag{5.4}$$

where θ is the angle between the two vectors, and $|\vec{H}|$ and $|\vec{F}|$ are the lengths of the two vectors. The normalization of the dot product between the vectors is a fact because $\cos\theta \in [-1, 1]$. The length of $|\vec{H}|$, which is also the "length" of the template, is calculated as

$$\text{Length of template} = \sqrt{\sum_{j=-R}^{R} \sum_{i=-R}^{R} h(i, j) \cdot h(i, j)}, \tag{5.5}$$

where R is the radius of the template and $h(i, j)$ is the coefficient in the template at position (i, j). The length of the image patch is calculated in the same manner.

When using this normalization the bright spots in the output no longer depend on whether the image is bright or not but only on how similar the template and the underlying image patch are. This version of template matching is denoted *Normalized Cross Correlation* (NCC) and calculated for each pixel (x, y) using the following equation:

$$\text{NCC}(x, y) = \frac{\text{Correlation}}{\text{Length of image patch} \cdot \text{Length of template}}$$

that after inserting the relevant equations becomes

$$\text{NCC}(x, y) = \frac{\sum_{j=-R}^{R} \sum_{i=-R}^{R} (H \cdot F)}{\sqrt{\sum_{j=-R}^{R} \sum_{i=-R}^{R} (F \cdot F)} \cdot \sqrt{\sum_{j=-R}^{R} \sum_{i=-R}^{R} (H \cdot H)}}, \tag{5.6}$$

where R is the radius of the template, $H = h(i, j)$ is the template and $F = f(x + i, y + j)$ is the image patch. $\cos\theta \in [-1, 1]$ but since the image patch and the template always contain positive numbers, $\cos\theta \in [0, 1]$, i.e., the output of normalized cross correlation is normalized to the interval [0, 1], where 0 means no similarity and 1 means a complete similarity. In Fig. 5.10 the benefit of applying normalized cross correlation can be seen.

An even more advanced version of template matching exists. Here the mean values of the template and image patch are subtracted from H and F, respectively. This is known as the *zero-mean normalized cross correlation* or the *correlation coefficient*. The output is in the interval $[-1, 1]$ where 1 indicates a maximum similarity (as for NCC) and –1 indicates a maximum *negative* similarity, meaning the same pattern but opposite gray-scale values: 255 instead of 0, 254 instead of 1, etc.

Independent of the type of template matching, the kernel (template) is usually much bigger than the kernels/filters used in other neighborhood operations. Template matching is therefore a time consuming method and can benefit from introducing a region-of-interest, see Sect. 2.4.2.

A general assumption in template matching is that the object we are looking for has roughly the same size and rotation in both the template and the image. If this cannot be ensured, then we need to have multiple scaled and rotated versions of the template and perform template matching using each of these templates. This requires significant resources and the speed of the system is likely to drop.

5.2.2 Edge Detection

Another important application of correlation is *edge detection*. An edge in an image is defined as a position where a significant change in gray-level values occurs. In Fig. 5.11 an image is shown to the left. We now take an image slice defined by the vertical line between the two arrows. This new image will have the same height as the input image, but only be one pixel wide. In the figure, this is illustrated. Note that we have made it wider in order to be able to actually see it. Imagine now that we interpret the intensity values as height values. This gives us a different representation of the image, which is shown in the graph to the right.

What can be seen in the graph is that locations in the original image where we have a significant change in gray-scale value appear as significant changes in height. Such positions are illustrated by circles in the figure. It is these positions where we say we have an edge in an image.

Edges are useful in many applications since they define the contour of an object and are less sensitive to changes in the illumination compared to, for example, thresholding. Moreover, in many industrial applications image processing (or rather machine vision) is used to measure some dimensions of objects. It is therefore of great importance to have a clear definition of where an object starts and ends. Edges are often used for this purpose.

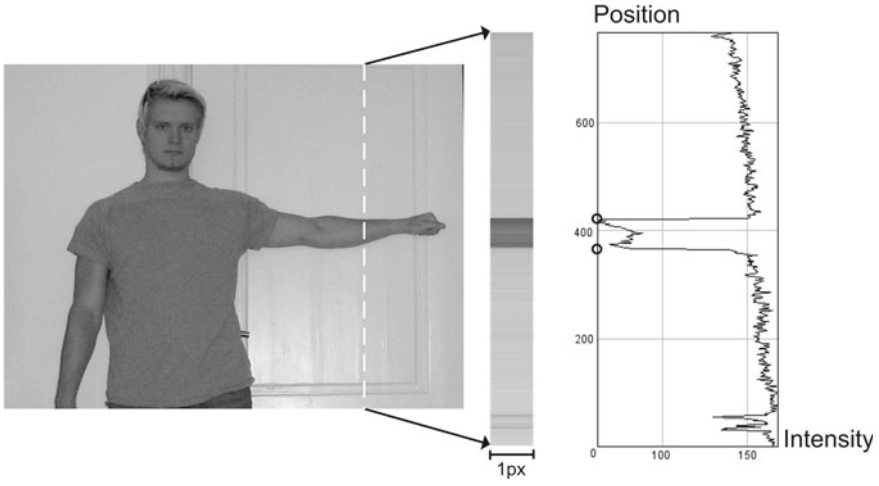


Fig. 5.11 A single column of the image is enlarged and presented in a graph. This graph contains two very significant changes in height, the position of which is marked with circles on the graph. This is how edges are defined in an image

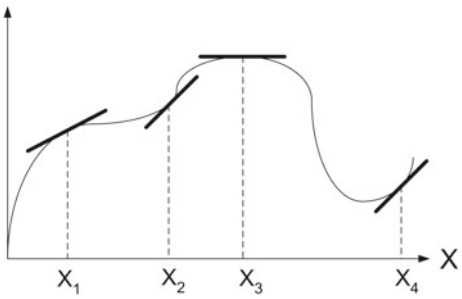


Fig. 5.12 A curve and the tangent at four points

5.2.2.1 Gradients

To enable edge detection we utilize the concept of gradients. We first present gradients for a general curve and then turn to gradients in images. In the 1D case, we can define the gradient of a point as the slope of the curve at this point. Concretely this corresponds to the slope of the tangent at this point. In Fig. 5.12 the tangents of several different points are shown.

If we represent an image by height as opposed to intensity, see Fig. 5.13, then edges correspond to places where we have steep hills. For each point in this image landscape we have two gradients: one in the x-direction and one in the y-direction. Together these two gradients span a plane, known as the *tangent plane*, which intersects the point. The resulting gradient is defined as a vector

$$\nabla f(x, y) = \overrightarrow{G}(g_x, g_y), \tag{5.7}$$

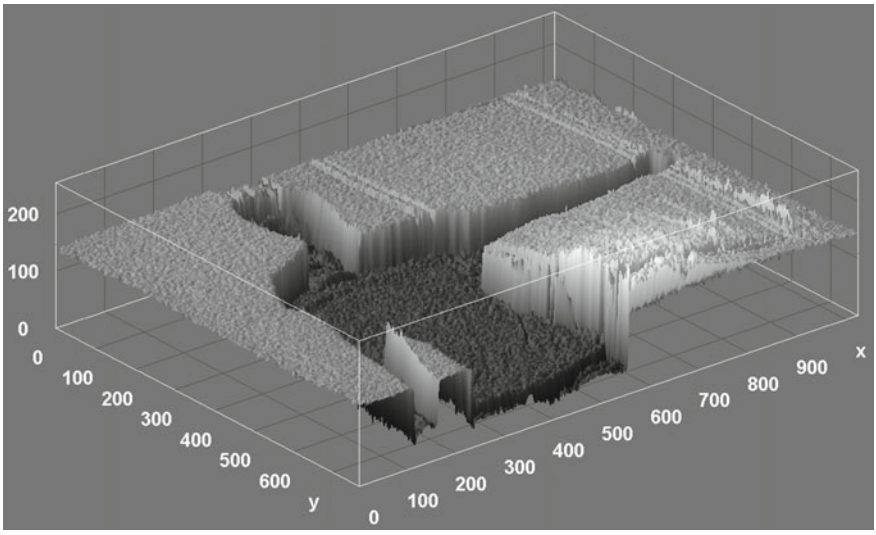


Fig. 5.13 A 3d representation of the image from Fig. 5.11, where the intensity of each pixel is interpreted as a height

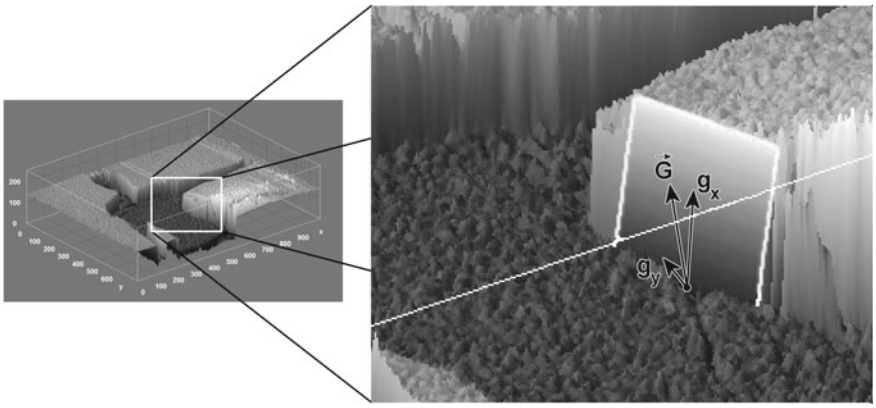


Fig. 5.14 In a 3d representation of an image, a tangent plane is present for each point. Such a plane is defined by two gradient vectors in x- and y-direction, respectively. Here the tangent plane is shown for one pixel

where g_x is the gradient in the x-direction and g_y is the gradient in the y-direction. This resulting gradient lies in the tangent plane, see Fig. 5.14. The symbol, ∇, called *nabla* is often used to describe the gradient.

We can consider $\overrightarrow{G}(g_x, g_y)$ as the direction with the steepest slope (or least steepest slope depending on how we calculate it), or in other words, if you are standing at this position in the landscape, you need to follow the opposite direction of the gradient in order to get down fastest. Or in yet another way, when water falls at this point it will run in the opposite direction of the gradient.

Besides a direction the gradient also has a *magnitude*. The magnitude expresses how steep the landscape is in the direction of the gradient, or how fast the water will run away (if you go skiing you will know that the magnitude of the gradient usually defines the difficulty of the piste). The magnitude is the length of the gradient vector and calculated as

$$\text{Magnitude} = \sqrt{g_x^2 + g_y^2} \tag{5.8}$$

$$\text{Approximated magnitude} = |g_x| + |g_y|, \tag{5.9}$$

where the approximation is introduced to achieve a faster implementation.

5.2.2.2 Image Edges

For the curves shown above, the gradients are found as the first order derivatives. This can only be calculated for continuous curves and since an image has a discrete representation (we only have pixel values at discrete positions: 0, 1, 2, 3, 4, etc.)

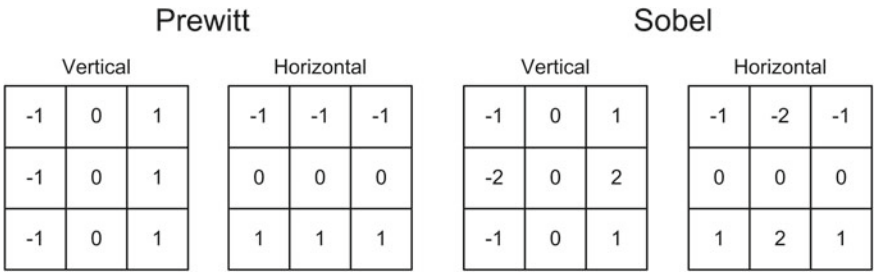


Fig. 5.15 Prewitt and Sobel kernels

we need an approximation. Recalling that the gradient is the slope at a point we can define the gradient as the difference between the previous and next value. Concretely we have the following image gradient approximations:

$$g_x(x, y) \approx f(x + 1, y) - f(x - 1, y) \tag{5.10}$$

$$g_y(x, y) \approx f(x, y + 1) - f(x, y - 1) \tag{5.11}$$

We have included (x, y) in the definition of the gradients to indicate that the gradient values depend on their spatial position. This approximation will produce positive gradient values when the pixels change from dark to bright and negative values when a reversed edge is present. This will of course be opposite if the signs are switched, i.e., $g_x(x, y) \approx f(x - 1, y) - f(x + 1, y)$ and $g_y(x, y) \approx f(x, y - 1) - f(x, y + 1)$. Normally the order does not matter as we will see below.

Equation 5.10 is applied to each pixel in the input image. Concretely this is done using correlation. We correlate the image with a 1×3 kernel containing the following coefficients: –1, 0, 1. Calculating the gradient using this kernel is often too sensitive to noise in the image and the neighbors are therefore often also included into the kernel. The most well-known kernels for edge detection are illustrated in Fig. 5.15: the *Prewitt kernels* and the *Sobel kernels*. The difference is that the Sobel kernels weight the row and column pixels of the center pixel more than the rest.

Correlating the two Sobel kernels with the image in Fig. 5.11 yields the edge images in Fig. 5.16. The image to the left enhances horizontal edges while the image to the right enhances vertical edges. To produce the final edge image we use Eq. 5.8. That is, we first calculate the absolute value[5] of each pixel in the two images and then add them together. The result is the final edge enhanced image. After this, the final task is often to binarize the image, so that edges are white and the rest is black. This is done by a standard thresholding algorithm. In Fig. 5.16 the final edge enhanced image is shown together with binary edge images obtained using different thresholds. The choice of threshold depends on the application.

[5] A definition of the *absolute value* can be found in Appendix B.

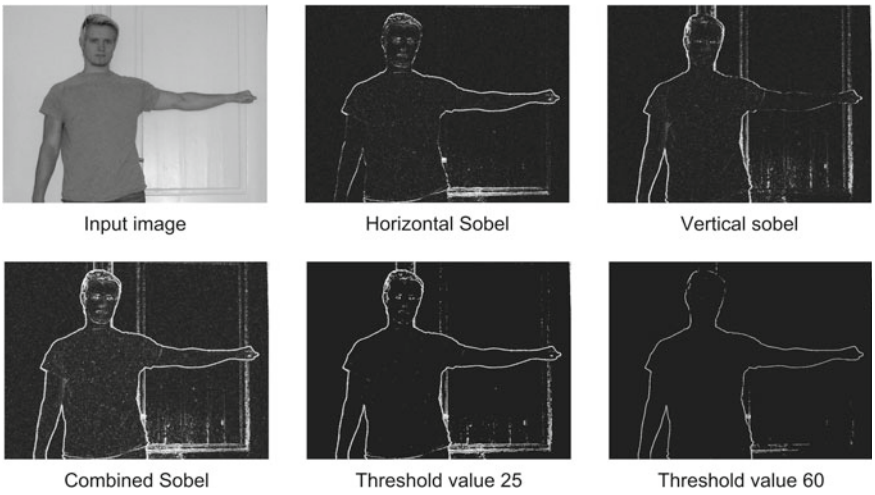


Fig. 5.16 Sobel kernels applied to an image. Each individual kernel finds edges that the other does not find. When they are combined a very nice resulting edge is created. Depending on the application, the threshold value can be manipulated to include or exclude the vaguely defined edges

5.3 Correlation Versus Convolution

The following text is likely to confuse you! One could therefore argue that it should be ignored, but we feel it is important to know the difference between correlation and *convolution* since both names are used throughout the image processing literature.

Convolution is very similar to correlation and only differs by the way the kernel is applied to the image beneath it. Mathematically convolution is defined as

$$g(x, y) = \sum_{j=-R}^{R} \sum_{i=-R}^{R} h(i, j) \cdot f(x - i, y - j) \tag{5.12}$$

Comparing this to the equation for correlation in Eq. 5.2 we can see that the only differences are the two minus signs. The interpretation of these is that the kernel is rotated 180° before doing a correlation. In Fig. 5.17 examples of rotated kernels are shown. What we can see is that symmetric kernels are equal before and after rotation, and hence convolution and correlation produce the same result. Edge detection kernels are not symmetric. However, since we often only are interested in the absolute value of an edge the correlation and convolution again yield the same result.

When applying smoothing filters, finding edges, etc. the process is often denoted convolution even though it is often implemented as correlation! When doing template matching it is virtually always denoted correlation.

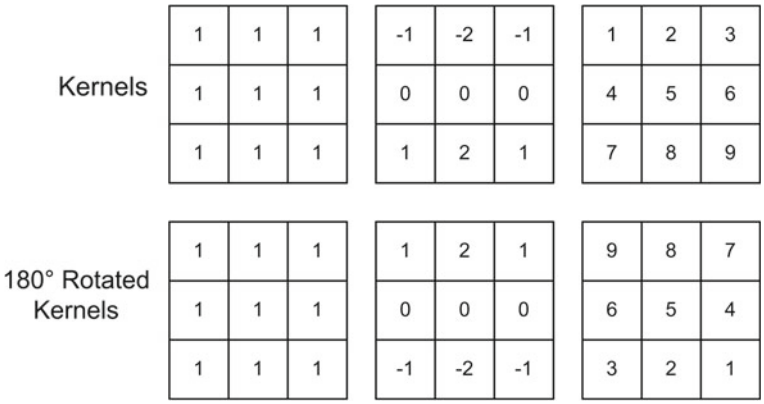


Fig. 5.17 Three kernels and their rotated counterparts

One might rightfully ask why convolution is used in the first place. The answer is that from a general signal processing[6] point of view we actually do convolution, and correlation is convolution done with a rotated kernel. However, since correlation is easier to explain and since it is most often what is done in practice, it has been presented as though it were the other way around in this (and many other) texts. The technical reasons for the definition of convolution are beyond the scope of this text and the interested reader is referred to a general signal processing textbook.

[6]Image processing is a subset of signal processing.

Morphology

6

One important branch of neighborhood processing is *mathematical morphology*—or simply *morphology*. It is applicable to both gray-scale images as well as binary images, but in this text only operations related to binary images are covered. Morphology on binary images has a number of applications and in Fig. 6.1 three typical ones are illustrated. The first two illustrate how to remove the noise that very often is a side effect of thresholding. Remember that thresholding is a global operation meaning that all pixels, independent of position, are compared to the same threshold value. But if the light in the scene is not uniform or the objects have different colors (due to, for example, clothing), then we cannot define one perfect threshold value and the result is under-segmentation in some regions and over-segmentation in other regions. The left-most figure illustrates over-segmentation in the form of the small objects in the image. Under-segmentation is illustrated in the middle figure as holes inside the object. The problems associated with thresholding were also mentioned in Chap. 4 where it could be seen as the *problematic histogram* in Fig. 4.18.

The right-most example in Fig. 6.1 illustrates a problem which is related to the next chapter, where we will start to analyze individual objects. To this end we need to ensure that the objects are separated from each other.

Morphology operates like the other neighborhood processing methods by applying a kernel to each pixel in the input. In morphology, the kernel is denoted a *structuring element* and contains "0"s and "1"s. You can design the structuring element as you please, but normally the pattern of "1"s form a box or a disc. In Fig. 6.2 different sized structuring elements are visualized. Which type and size to use is up to the designer, but in general a box-shaped structuring element tends to preserve sharp object corners, whereas a disc-shaped structuring element tends to round the corners of the objects.

A structuring element is *not* applied in the same way as we saw in the previous chapter for the kernels. Instead of using multiplications and additions in the calcula-

R. R. Paulsen and T. B. Moeslund, *Introduction to Medical Image Analysis*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-39364-9_6

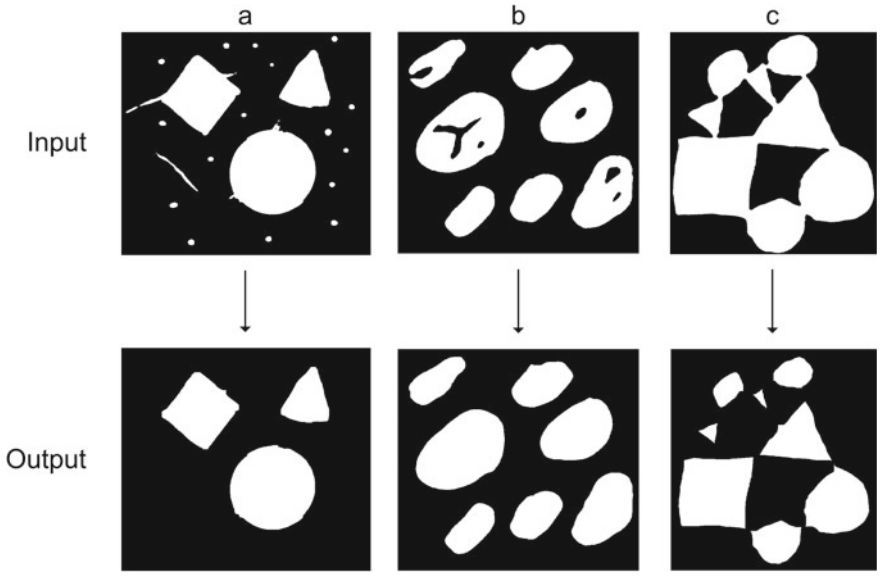


Fig. 6.1 Three examples of the uses of morphology. **a** Removing small objects. **b** Filling holes. **c** Isolating objects

tions, a structuring element is applied using either a *Hit* or a *Fit* operation. Applying one of these operations to each pixel in an image is denoted *Dilation* and *Erosion*, respectively. Combining these two methods can result in powerful image processing tools known as *Compound Operations*. We can say that there exist three levels of operation, see Fig. 6.3, and in the following, these three levels will be described one at a time. Note that for simplicity, we will in this chapter represent white as 1 instead of 255.

6.1 Level 1: Hit and Fit

The structuring element is placed on top of the image as was the case for the kernels in the previous chapter. The center of the structuring element is placed at the position of the pixel in focus and it is the value of this pixel that will be calculated by applying the structuring element. After having placed the structuring element we can apply one of two methods: Hit or Fit.

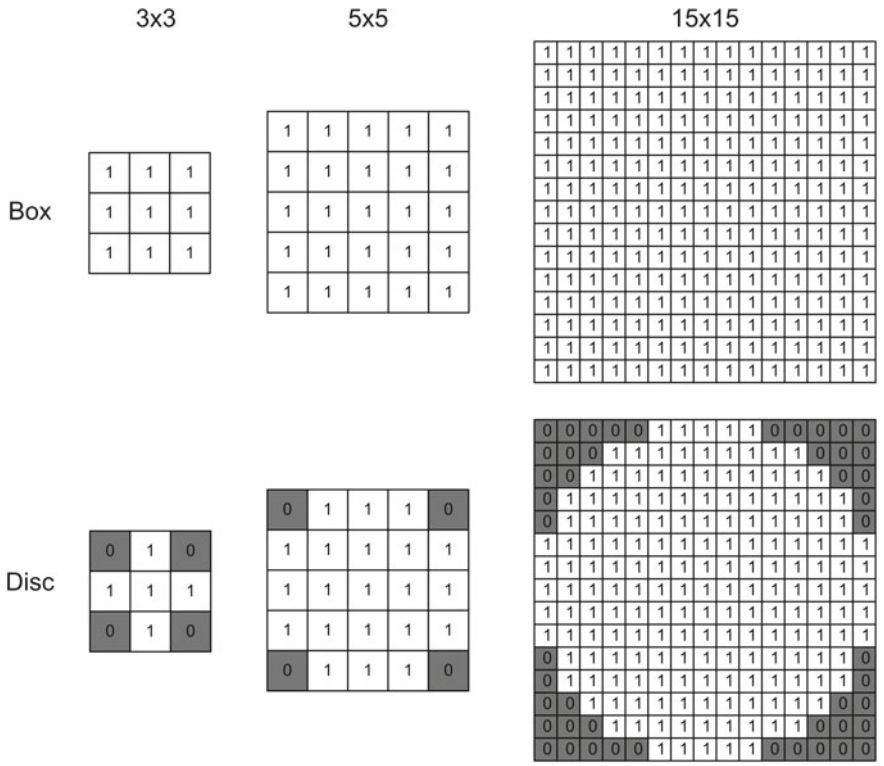
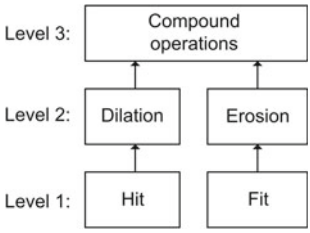


Fig. 6.2 Two types of structuring elements at different sizes

Fig. 6.3 The three levels of operation involved in morphology



6.1.1 Hit

For each "1" in the structuring element we investigate whether the pixel at the same position in the image is also a "1". If this is the case for just one of the "1"s in the structuring element, we say that the structuring element *hits* the image at the pixel position in question (the one on which the structuring element is centered). This pixel is therefore set to "1" in the output image. Otherwise it is set to "0". In Fig. 6.4 and Table 6.1 the hit operation is illustrated with two different structuring elements.

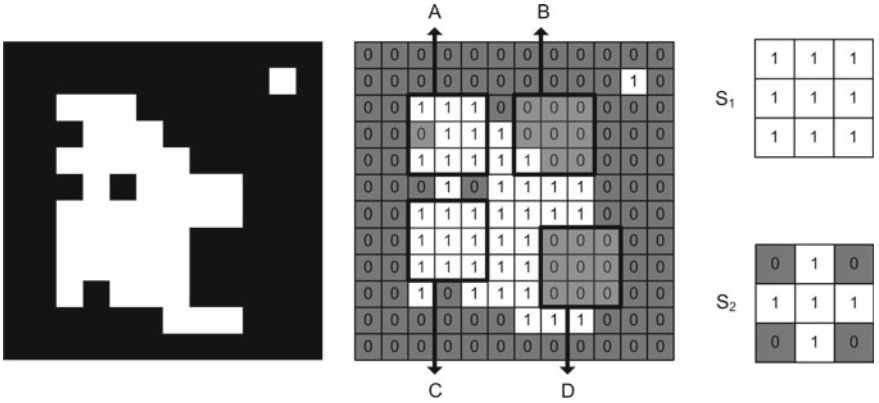


Fig. 6.4 A binary image illustrated both by colors (black and white) and numbers (0 and 1). A, B, C, and D illustrate four 3 × 3 image regions centered at: A: f(3, 3), B: f(7, 3), C: f(3, 7), and D: f(8, 8). Lastly, two different 3 × 3 structuring elements are illustrated

Table 6.1 Results of applying the two Structuring Elements (SE) in Fig. 6.4 to the input image in Fig. 6.4 at four positions: A, B, C, and D

Position	SE	Fit	Hit
A	S_1	No	Yes
A	S_2	No	Yes
B	S_1	No	Yes
B	S_2	No	No
C	S_1	Yes	Yes
C	S_2	Yes	Yes
D	S_1	No	No
D	S_2	No	No

6.1.2 Fit

For each "1" in the structuring element we investigate whether the pixel at the same position in the image is also a "1". If this is the case for *all* the "1"s in the structuring element, we say that the structuring element *fits* the image at the pixel position in question (the one on which the structuring element is centered). This pixel is therefore set to "1" in the output image. Otherwise it is set to "0". In Fig. 6.4 and Table 6.1 the fit operation is illustrated with two different structuring elements. Below we show C-code for the fit operation using a 3 × 3 box-shaped structuring element:

Implementation of the fit operation

```
Temp = 0;
for (j = y-1; j < (y+2); j = j+1)
{
    for (i = x-1; i< (x+2); i = i+1)
    {
        if (GetPixel(input, i, j) == 1)
            Temp = Temp + 1;
    }
}
if(Temp == 9)
    SetPixel(output, x, y, 1);
else
    SetPixel(output, x, y, 0);
```

Here `(x,y)` is the position of the pixel being processed.

6.2 Level 2: Dilation and Erosion

At the next level Hit or Fit is applied to every single pixel by scanning through the image as shown in Fig. 4.24. The size of the structuring element in these operations has the same importance as the kernel size did in the previous chapter. The bigger the structuring element, the bigger the effect in the image. As described in the previous chapter, we also have the border problem present here and solution strategies similar to those listed in Sect. 5.1 can be followed. For simplicity, we will ignore the border problem in this chapter.

6.2.1 Dilation

Applying Hit to an entire image is denoted *Dilation* and is written as

$$g(x, y) = f(x, y) \oplus SE \ . \tag{6.1}$$

The term *dilation* refers to the fact that the object in the binary image is increased in size. In general, dilating an image results in objects becoming bigger, small holes being filled, and objects being merged. How big the effect is depends on the size of the structuring element. It should be noticed that a large structuring element can be implemented by iteratively applying a smaller structuring element. This makes

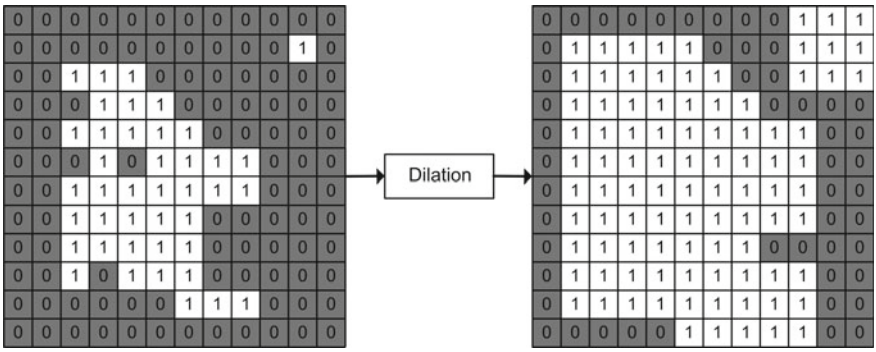


Fig. 6.5 Dilation of the binary image in Fig. 6.4 using S_1

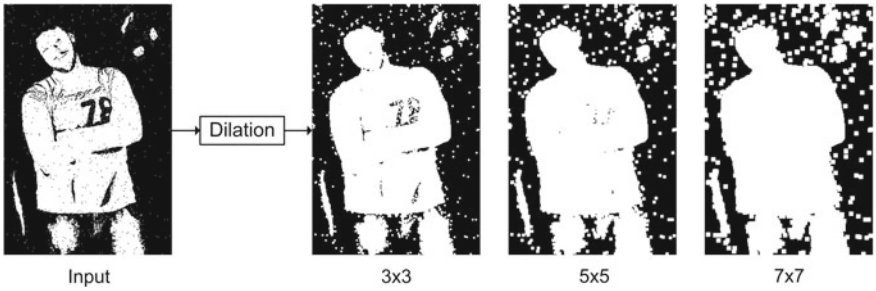


Fig. 6.6 Dilation with different sized structuring elements

sense since Eq. 6.2 holds. The equation states that dilating twice with SE_1 is similar to dilating one time with SE_2, where SE_2 is the same type but has twice the radius of SE_1. For example, if SE_2 is a 5×5 structuring element, then SE_1 is a 3×3, etc.

$$f(x, y) \oplus SE_2 \approx (f(x, y) \oplus SE_1) \oplus SE_1 \ . \tag{6.2}$$

In Fig. 6.5 the binary image in Fig. 6.4 is dilated using the structuring element S_1. First of all we can see that the object gets bigger. Secondly, we can observe that the hole and the convex parts of the object are filled, which makes the object more compact.

In Fig. 6.6 a real image is dilated with different sized box-shaped structuring elements. Again we can see that the object is becoming bigger and that holes inside the person are filled. What is however also apparent is that the noisy small objects are also enlarged. Below we will return to this problem.

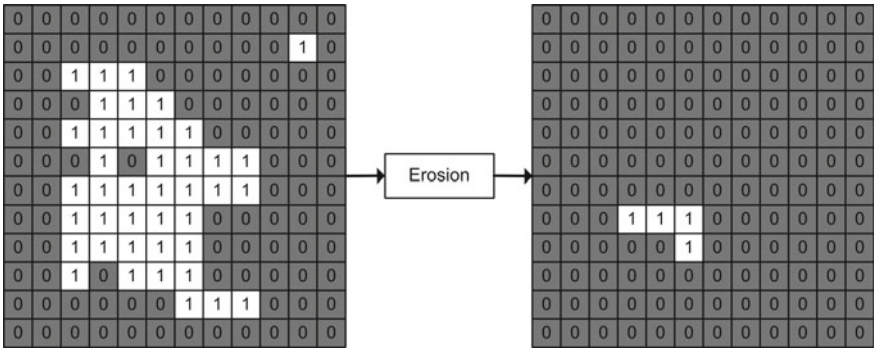


Fig. 6.7 Erosion of the binary image in Fig. 6.4 using S_1

6.2.2 Erosion

Applying Fit to an entire image is denoted Erosion and is written as

$$g(x, y) = f(x, y) \ominus SE \ . \tag{6.3}$$

The term *erosion* refers to the fact that the object in the binary image is decreased in size. In general, erosion of an image results in objects becoming smaller, small objects disappearing, and larger objects splitting into smaller objects. As for dilation the effect depends on the size of the structuring element and large structuring elements can be implemented using an equation similar to Eq. 6.2.

In Fig. 6.7 the binary image in Fig. 6.4 is eroded using the structuring element S_1. First of all, we can see that the main object gets smaller and the small objects disappear. Secondly, we can observe that the fractured parts of the main object are removed and only the "core" of the object remains. The size of this core obviously depends on the size (and shape) of the structuring element.

In Fig. 6.8 a real image is eroded with different sized box-shaped structuring elements. Again we can see that the object becomes smaller and the small (noisy)

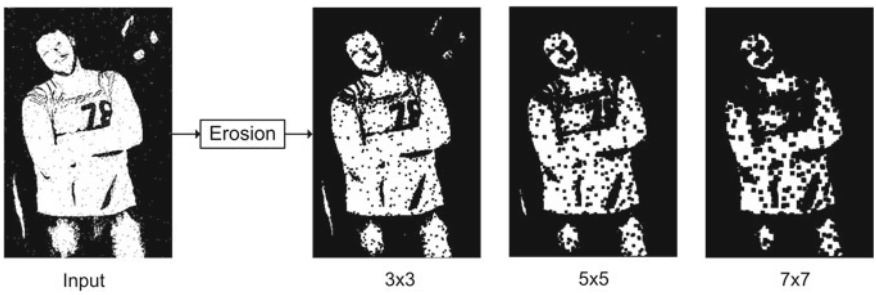


Fig. 6.8 Erosion with different sized structuring elements

objects disappear. So the price we pay for deleting the small noisy objects is that the object of interest becomes smaller and fractured. Below we will return to this problem.

6.3 Level 3: Compound Operations

Combining dilation and erosion in different ways results in a number of different image processing tools. These are denoted *compound operations*. Here we present three of the most common compound operations, namely, *Opening*, *Closing*, and *Boundary Detection*.

6.3.1 Closing

Closing deals with the problem associated with dilation, namely, that the objects increase in size when we use dilation to fill the holes in objects. This is a problem in situations where, for example, the size of the object (number of pixels) matters. The solution to this problem is luckily straight forward: we simply shrink the object by following the Dilation by an Erosion. This operation is denoted *Closing* and is written as

$$g(x, y) = f(x, y) \bullet SE = (f(x, y) \oplus SE) \ominus SE \ , \tag{6.4}$$

where SE is the structuring element. It is essential that the structuring elements applied are exactly the same in terms of size and shape. The closing operation is said to be *idempotent*, meaning that it can only be applied one time (with the same structuring element). If applied again it has no effect whatsoever except for of course a reduced size of $g(x, y)$ due to the border problem. In Fig. 6.9, closing is illustrated for the binary image in Fig. 6.4. Closing is done with structuring element S_1. We can see that the holes and convex structures are filled; hence, the object is more compact. Moreover, the object preserves its original size.

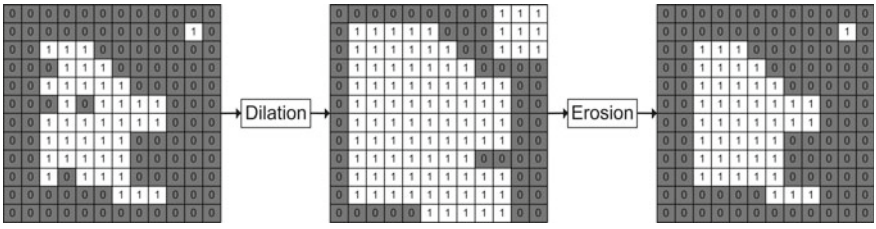


Fig. 6.9 Closing of the binary image in Fig. 6.4 using S_1

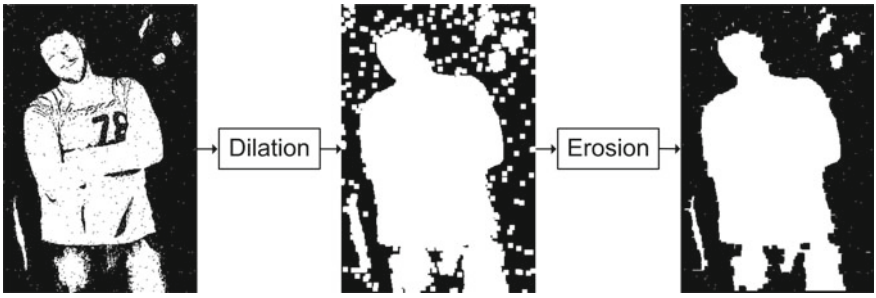


Fig. 6.10 Closing performed using 7 × 7 box-shaped structuring elements

In Fig. 6.10 the closing operation is applied to a real image. We can see that most internal holes are filled while the human object preserves its original size. The noisy objects in the background have not been deleted. This can be done either by the operation described just below or by finding and deleting small objects, which will be described in the next chapter.

6.3.2 Opening

Opening deals with the problem associated with erosion, namely, that the objects decrease when we use erosion to erase small noisy objects or fractured parts of bigger objects. The decreasing object size is a problem in situations where, for example, the size of the object (number of pixels) matters. The solution to this problem is luckily straight forward; we simply enlarge the object by following the erosion by dilation. This operation is denoted *Opening* and is written as

$$g(x, y) = f(x, y) \circ SE = (f(x, y) \ominus SE) \oplus SE \ , \tag{6.5}$$

where SE is the same structuring element. This operation is also idempotent as is the case for the closing operation. In Fig. 6.11 opening is illustrated for the binary image in Fig. 6.4. Opening is done with structuring element S_1. We can see that only a compact version of the object remains.

In Fig. 6.12 opening is applied to a real image using a 7 × 7 box-shaped structuring element. We can see that most noisy objects are removed while the object preserves its original size.

6.3.3 Combining Opening and Closing

In some situations we need to apply both opening and closing to an image. For example, in cases where we both have holes inside the main object *and* small noisy objects. An example is provided in Fig. 6.13. Note that the structuring elements used in the opening and the closing operations need not be the same. In Fig. 6.13

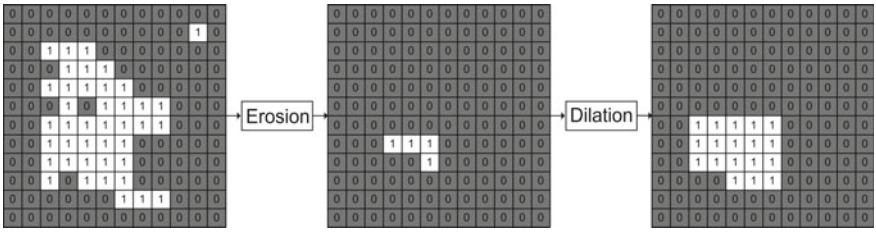


Fig. 6.11 Opening of the binary image in Fig. 6.4 using S_1

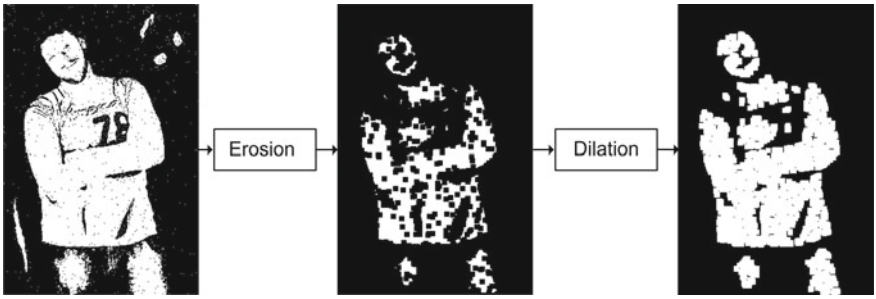


Fig. 6.12 Opening performed using a 7×7 box-shaped structuring element

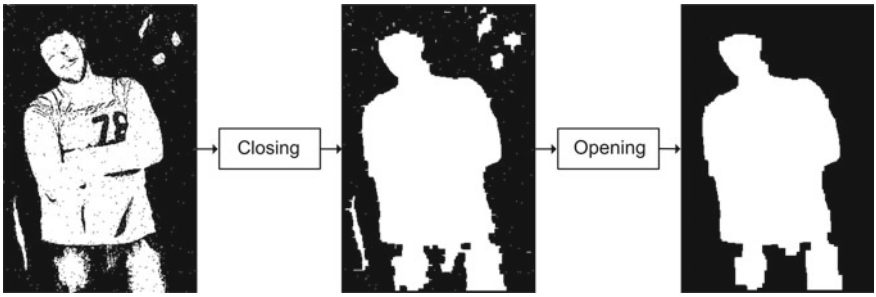


Fig. 6.13 Filtering a binary image where both holes and small noisy objects are present

the closing was performed using a 7×7 box-shaped structuring element while the opening was performed using a 15×15 box-shaped structuring element.

6.3.4 Boundary Detection

Doing edge detection in binary images is normally referred to as *boundary detection* and can be performed as described in the previous chapter. Morphology offers an alternative approach for binary images. The idea is to use erosion to make a smaller

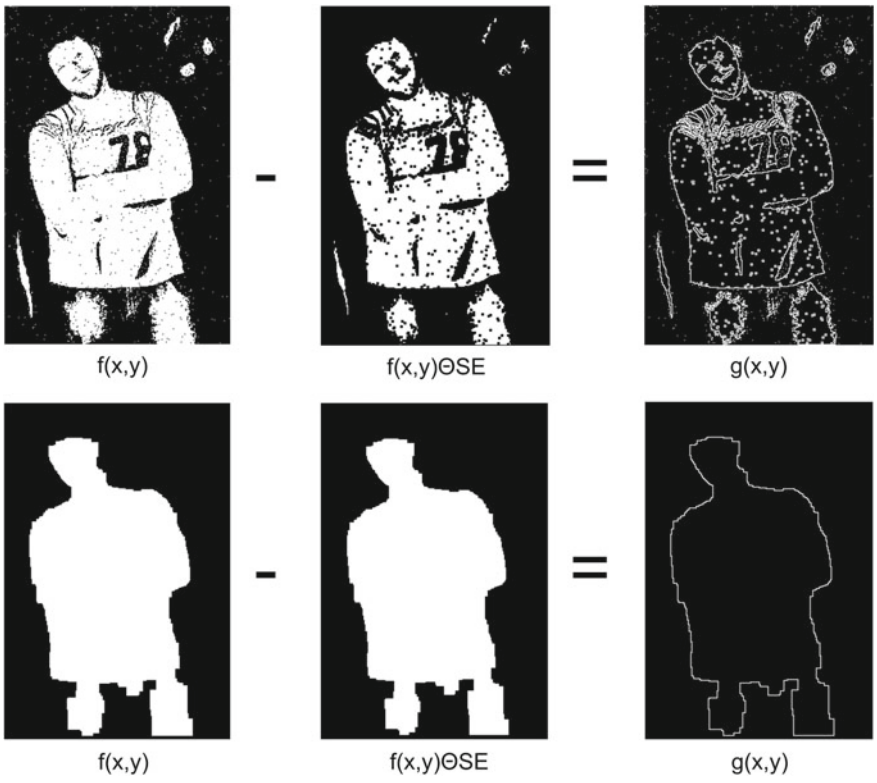


Fig. 6.14 Boundary detection

version of the object. By subtracting this from the input image only the difference stands out, namely, the boundary:

$$g(x, y) = f(x, y) - (f(x, y) \ominus SE) \ . \tag{6.6}$$

If the task is only to locate the outer boundary, then the internal holes should first be filled using dilation or closing. In Fig. 6.14 examples of boundary detection are shown.