

Digital and Medical Image Processing

Twan Maintz

Part I, Chapters 1–6
[Inleiding Beeldverwerking]
Part II, Chapters 7–10 (partially)
[Image Processing]
Update 11/2005

Contents

1	Introduction	9
1.1	Image processing: definition and examples	9
1.2	The purpose of image processing	12
1.3	The digital image	12
1.3.1	Mathematical representation of images	14
1.3.2	Display of images	15
1.4	Image processing and related disciplines	15
2	The human visual system	17
2.1	The human eye	17
2.2	Reflectivity and luminance	20
2.3	Some psychophysics	21
3	Digital image acquisition	27
3.1	Digital sampling	27
3.1.1	The Nyquist frequency	30
3.1.2	Aliasing	31
3.1.3	Sampling and convolution	32
3.1.4	Extension to two dimensions	36
3.1.5	The point spread function	37
3.2	Quantization	38
3.3	Color models	39

4 Operations on grey values	47
4.1 Examples of simple grey value remapping	47
4.2 Gamma correction	49
4.3 Contrast stretching	50
4.4 Other remapping operations	52
4.5 The image histogram	56
4.6 Histogram equalization	59
4.7 Multi-image operations	62
4.7.1 Image subtraction	62
4.7.2 Image addition	63
4.7.3 Boolean operations	65
4.7.4 Image compositions	66
4.8 Range considerations	69
4.9 The two-dimensional histogram	70
4.10 Noise	72
5 Simple spatial operations	75
5.1 Discrete convolution	75
5.1.1 Smoothing and low pass filtering	78
5.1.2 Sharpening and high pass filtering	79
5.1.3 Derivatives	82
5.1.4 Edge detection	89
5.1.5 Approximating continuous convolution	96
5.1.6 More on kernels	101
5.2 Simple non-linear filtering	104
5.3 Handling image borders	105
5.4 Interpolation	107
5.4.1 Interpolation by convolution	109
5.5 Geometric transformation	112
5.5.1 Backward mapping	119

6 Mathematical morphology	121
6.1 Complement and operator properties	123
6.2 Relation between sets and images	126
6.3 Erosion and dilation	126
6.3.1 Properties of erosion and dilation	133
6.4 Opening and closing	134
6.4.1 Properties of opening and closing	136
6.5 Geodesic operations and reconstruction	136
6.5.1 Opening by reconstruction of erosion	139
6.5.2 Other openings by reconstruction	139
6.6 Residues	142
6.6.1 Morphological gradient filters	142
6.6.2 Top hat filters	144
6.6.3 The skeleton and ultimate erosion	147
6.6.4 Hit-or-miss transformation	154
6.6.5 Thinning and thickening	155
6.7 Applications	157
6.7.1 Alternating sequential filters	157
6.7.2 Granulometry	158
6.7.3 Toggle mapping	160
7 The Fourier transform	163
7.1 The relation between digital images and sinusoids	163
7.1.1 The Fourier transform	165
7.1.2 Fourier series	172
7.2 Image processing in the frequency domain	172
7.2.1 Filter shapes	177

7.2.2	Removing periodic noise	178
7.3	Properties of the Fourier transform	179
7.3.1	Separability	179
7.3.2	Linearity	180
7.3.3	Convolution property	181
7.3.4	Derivative property	182
7.3.5	Other properties	182
7.4	Inverse filtering	183
7.5	The Discrete Fourier Transform	187
7.6	Other transforms	188
8	Object representation and analysis	193
8.1	Object shape measurements	193
8.1.1	Skeleton based measurements	200
8.1.2	Moment based measurements	206
8.1.3	Grey value moments	213
8.2	Multiple object measurements	215
8.2.1	Aggregating separate object measurements	215
8.2.2	Counting objects	217
8.2.3	Objects intersecting the image boundary	218
8.2.4	Overlapping objects	220
8.2.5	Object distribution	223
8.2.6	The Hough transform	224
8.3	Object representation	228
8.3.1	Boundary representation: chain codes	229
8.3.2	Measurements using chain codes	232
8.3.3	Boundary representation: distance from the center of mass	236
8.3.4	Boundary representation: Fourier descriptors	240
8.3.5	Boundary representation: splines	241
8.3.6	Region representation: run-length codes	242
8.3.7	Region representation: quadtrees	244
8.3.8	Other representations	246

9 Scale space	247
9.1 Scale space	247
9.1.1 Scaled differential operators	251
9.1.2 Scaled differential features	254
9.2 Scale space and diffusion*	262
9.3 The resolution pyramid	267
 10 Segmentation	 273
10.1 Threshold based segmentation	274
10.1.1 Threshold selection	276
10.1.2 Enhancing threshold segmentation	287
10.1.3 Multispectral thresholding	293
10.2 Edge based segmentation	294
10.2.1 Edge linking	296
10.2.2 Watershed segmentation	303
10.2.3 Snakes	305
10.3 Region based segmentation	310
10.3.1 Merging methods	312
10.3.2 Splitting and split & merge methods	315
10.3.3 Pyramid, tree, and scale space methods	316
10.3.4 Texture measures	321
10.4 Clustering techniques	326
10.5 Matching	330

Chapter 1

Introduction

Images are everywhere. No wonder, since we –as human beings– rely on the images we perceive with our eyes more than any other sensory stimulus. Almost all of the information we digest comes to us in the form of an image; whether we look at a photograph, watch television, admire a painting, or read a book, it all makes use of imagery. Images are so natural to us, that we go to great lengths to convert almost any kind of information to images. For example: the TV weather forecast shows the temperature distribution in some geographical area as an image with different colors representing different temperatures, medical scanners can show human metabolism activity as an image where bright spots indicate high activity, etc. Moreover, our vision is usually the most efficient of our senses: consider, for example, a computer keyboard. The function of each key is represented by a small image (a character). We could also have identified each key by a specific relief texture, but it would be far less efficient. We could even try to give each key a specific smell, but it is easy to imagine (!) the trouble we would have typing.

We are also adept at a number of *image processing* tasks. For example, the focusing of our eyes: when we look at something, the first image our eyes send to the brain is probably out of focus. The brain then tries to correct this by adjusting the eye lens, whereupon a new image is sent from the eyes to the brain, and so on. This feedback process is so fast that we aren't even aware of it. Another example is stereo vision: our eyes send two two-dimensional images to the brain, but the brain is able to merge these into one three-dimensional image *virtually instantaneously*.

The science of *image processing* combines this natural way humans use images with the science of mathematics. This provides a unique blend, since images and image processing are described in a rigorous mathematical way without loss of the intuitive character of imagery.

1.1 Image processing: definition and examples

Image processing can be defined as

“The manipulation and analysis of information contained in images”.

This definition is of course very broad, and includes a wide range of natural and artificial processes, from the use of a pair of glasses to automatic analysis of images transmitted by the Hubble telescope. ‘Simple’ forms of image processing can be found all around us; examples include:

- the use of glasses or contact lenses
- brightness, contrast, *etc.* controls on a television or monitor
- taking (and developing) a picture using a photocamera
- natural examples: reflection of scenery on a water surface, distortions of scenery in the mist, a *fata morgana*, *etc.*

Examples of the use of advanced image processing include:

- **forensic science:** enhancement of images of video surveillance cameras, automatic recognition and classification of faces, fingerprints, DNA codes, *etc.* from images
- **industry:** checking of manufactured parts, application to CAD/CAM
- **information processing:** ‘reading’ of handwritten and printed texts (frequently referred to as OCR; optical character recognition), scanning and classification of printed images

A large number of applications of image processing can be found in the medical sciences using one or more medical images of a patient, *e.g.*,

- **Visualization.** For example: before we can make a 3D visualization of a three dimensional object (such as the head in figure 1.1), we often need to extract the object information first from two-dimensional images.
- **Computer aided diagnosis.** For example: in western countries it is common to regularly make breast radiographs of females above a certain age in order to detect breast cancer in an early stage. In practice, the number of images involved is so large that it would be very helpful to do part of the screening by means of automated computer image processing.
- **Image segmentation,** *i.e.*, the division of an image into meaningful structures. For example: the division of a brain image into structures like white brain matter, grey brain matter, cerebrospinal fluid, bone, fat, skin, *etc.* An example can be seen in figure 1.2. Segmentation is useful in many tasks, ranging from improved visualization to the monitoring of tumor growth.
- **Image registration** (also called image matching), *i.e.*, the exact alignment of two or more images of the same patient, which is necessary if the information contained in these images is to be combined in a meaningful new image. An example is shown in figure 1.3.

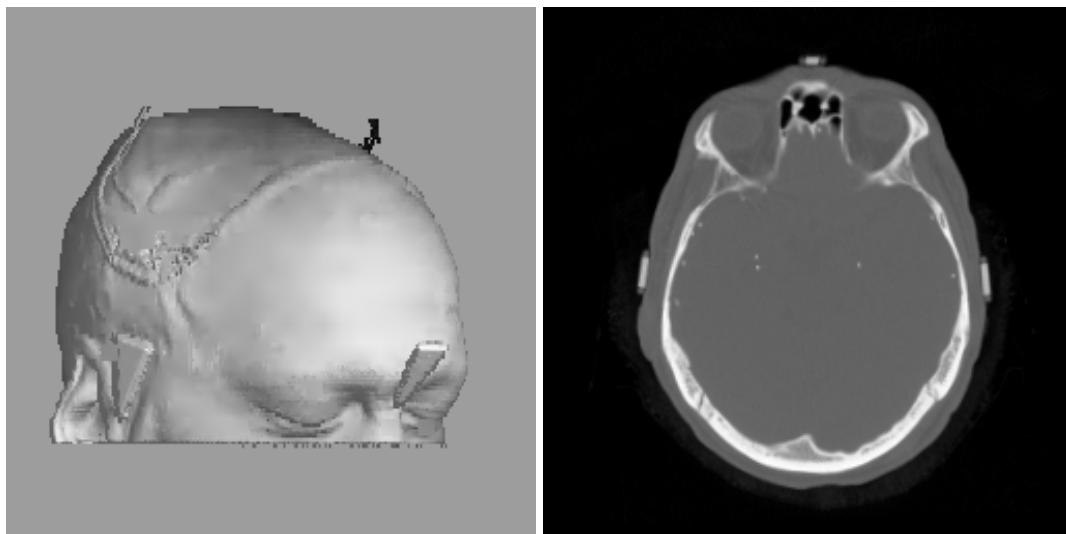


Figure 1.1 Example of extraction of information by image processing: the head visualized in three dimensions on the left was extracted from two dimensional images such as the one on the right.



Figure 1.2 Example of segmentation: the brain on the left was segmented into six structures (indicated by different grey values in the right image) by an automatic algorithm.

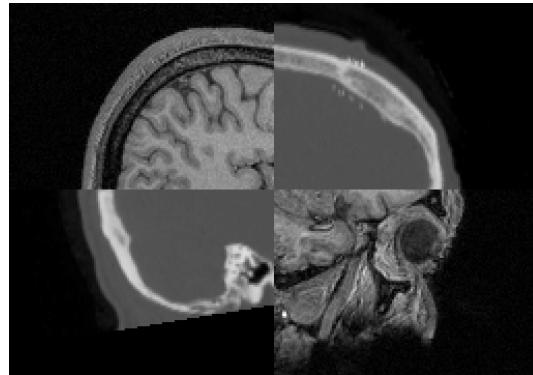


Figure 1.3 Example of two medical images that have been brought into registration: The upper left and lower right quarters belong to the first image, the lower left and upper right belong to the second image. Visualizations like these enable the viewer to make use of the information contained in the separate images simultaneously.

1.2 The purpose of image processing

Image processing applications can have many purposes. Most of the time, the purpose will be in one or more of these categories:

- **Image enhancement**, e.g., the reduction of noise or sharpening of images.
- **Pattern recognition**, e.g., the automatic detection of a certain shape or texture in images.
- **Reduction of data** to information that is more easily handled or interpreted, e.g., the reduction of an image to an “easier” image, to a set of objects or features, or to a set of measurements.
- **Image synthesis**, e.g., reconstructing a three-dimensional scene from two-dimensional photographs.
- **Combination of images.** When images of two different modalities (types) are made from the same scene, combining them involves registration, and, after that, often data reduction and image synthesis.
- **Data compression.** To reduce the size of computer files containing images and speed up image transmission across a network, data compression is often a must.

1.3 The digital image

In this book, we concern ourselves only with *digital* image processing, and not analog processing. The reason for this is that analog processing requires special hardware (electrical circuits), which makes building a special image processing application a difficult

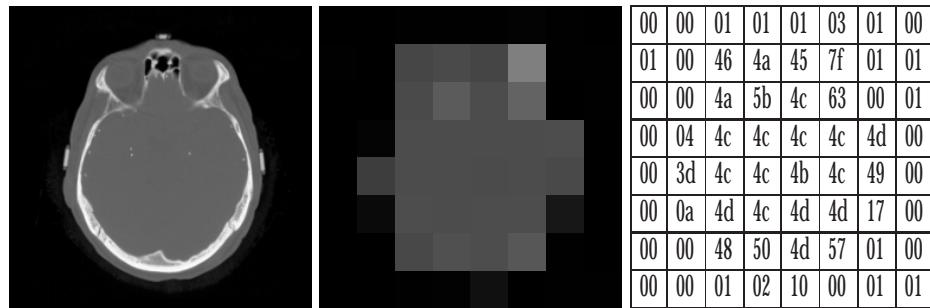


Figure 1.4 Example of sampling and quantization. The left image shows an original scene with continuous intensity values everywhere. After sampling using 8×8 locations (middle image), the image has real intensity values at specific locations only. After quantization (right image), these real values are converted to discrete numbers (here: hexadecimal values).

task. Moreover, the use of analog hardware is rapidly becoming obsolete in many image processing areas, since it can often be replaced by digital hardware (computers) which is much more flexible in its use.

But what exactly is a digital image? Obviously, we start with some kind of imaging device such as a videocamera, a medical scanner, or anything else that can convert a measure of a physical reality to an electrical signal. We assume this imaging device produces a continuous (in space) electrical signal. Since such an analog signal cannot directly be handled by digital circuits or computers, the signal is converted to a discretized form by a digitizer. The resulting image can then directly be used in digital image processing applications.

The digitizer performs two tasks, known as *sampling* and *quantization* (see figure 1.4). In the sampling process, the values of the continuous signal are sampled at specific locations in the image. In the quantization process, the real values are discretized into digital numbers. After quantization we call the result a digital image. So this answers the question at the beginning of this section: a digital image is nothing more than a matrix of numbers. Each matrix element, *i.e.*, a quantized sample, is called a *picture element* or a *pixel*. In the case of three-dimensional images this is named a *volume element* or *voxel*.

We can indicate the location of each pixel in an image by two coordinates (x, y) . By convention, the $(0, 0)$ pixel (the origin) is in the top left corner of the image, the x axis runs from left to right, and the y axis runs from top to bottom (see figure 1.5). This may take a little getting used to, because this differs from the conventional mathematical notation of two-dimensional functions¹, as well as from conventional matrix coordinates².

¹Where the origin is at the bottom left, and the y axis runs from bottom to top.

²Where the origin is the top left, the x axis runs from top to bottom, and the y axis from left to right.

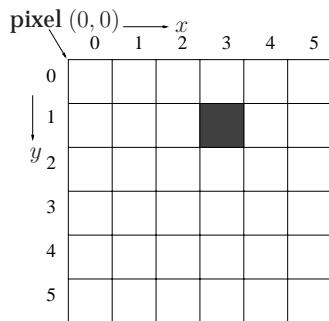


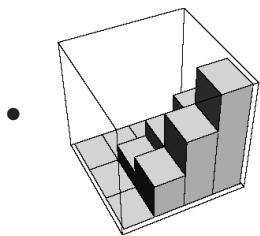
Figure 1.5 Coordinate conventions for images. Pixel (3, 1) is marked in grey.

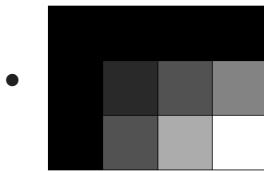
If a digital image is nothing more than a matrix of numbers, a very sceptic person might say that digital image processing is nothing more than a collection of mathematical algorithms that operate on a matrix. Fortunately, reality is not nearly as boring as this sounds, because in practice we will seldomly use the matrix representation shown in figure 1.4. Instead, we work with the middle image from figure 1.4 –which is in fact the same matrix, but with intensity values assigned to each number– but which usually makes much more sense to a human being. Throughout this book, you will find that image processing algorithms will be formulated as mathematical operators working on pixel values or pixel matrices, but the results of these algorithms will also be displayed using images.

1.3.1 Mathematical representation of images

In this book, we will use either functions or matrices to mathematically represent an image. Sometimes, it is also useful to use another graphical representation than the ‘standard’ image where a pixel value is represented by a certain luminance. For example, we may represent pixel values as height, like in the example below. With the mathematical representations, we will always use the image coordinate convention, so the following are equivalent:

- $f(x, y) = xy \text{ for } (x, y) \in \{0, 1, 2, 3\} \times \{0, 1, 2\}$
- $M = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & 2 & 4 & 6 \end{pmatrix}$





Further on in this book, we will come across images with more than two dimensions. The mathematical representations will be logical extensions of the ones above for two-dimensional images. For example, a movie can be regarded as a *time series* of two-dimensional images (frames), so a pixel value at location (x, y) of frame t can be represented by, e.g., $f(x, y, t)$. Another example is given by some medical images that have three *spatial* dimensions, and are therefore represented as, e.g., $f(x, y, z)$. If we want to visualize such three-dimensional images we often use volume renderings or simply display the three-dimensional image one two-dimensional “slice” at a time (see figure 1.1).

1.3.2 Display of images

When printing or displaying an image, its pixel values must be converted to grey values on the paper or screen. There are many possible ways to do this, but the most common three are:

- Find the lowest and highest pixel value occurring in the image and represent these as black and white respectively. Map the other pixel values between these extremes in a linear fashion.
- Find the lowest and highest pixel value that the image *type* supports (even if they do not actually occur in the image) and represent these as black and white. Map the other pixel values between these extremes in a linear fashion. For example, if the image type supports integer pixel values in the range $\{0, 1, 2, \dots, 255\}$, then we map 0 to black and 255 to white, even if 0 or 255 do not occur in the image at all.
- Use a *palette*. A palette is a table that explicitly describes the grey value for each pixel value.

1.4 Image processing and related disciplines

There are many disciplines that are closely related or have a large overlap with image processing, such as *computer vision*, *image understanding*, and *scene analysis*. Frankly, there is no real consensus on the borders between them, nor even on their definitions.

You could argue that they are all part of image processing³, given the broad definition we gave to image processing. A -debatable- graph showing relations of image processing and other disciplines can be seen in figure 1.6.

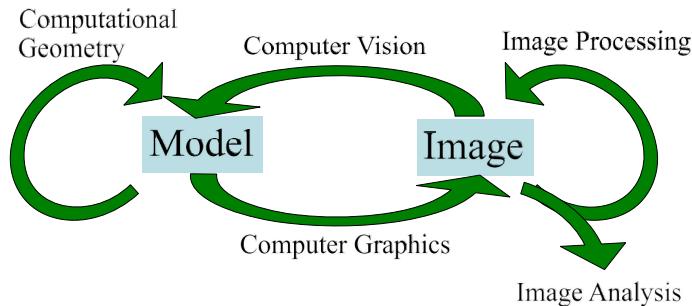


Figure 1.6 The relation of image processing and various connected disciplines.

Here, the various disciplines are defined by their actions regarding *images* and *models*, where the model is usually a compact mathematical structure describing a phenomenon using its essential parameters. For example, the image could be a photograph of a face, and the model a graph of facial feature points such as the corners of the eyes, the tip of the nose, etc, and the model parameters could be the distances between these points. A typical computer vision task would be to find the feature points in the image and match them to the model, e.g., for face recognition. A typical computer graphics task would be to generate an image of a face given the facial model and a set of distances. A typical image processing task would be to produce a new image by somehow enhancing the photograph, e.g., by improving the contrast, 'red eye removal', etc.

Although the disciplines mentioned in the graph all overlap to some extent, such overlap is small in the case of both computational geometry and computer graphics. This is not the case for image processing, computer vision, and image analysis, and attempting to find the borders between these areas often leads to artificial results. In this book, we will therefore not attempt it and simply consider all three to be image processing.

³But this upsets some people.

Chapter 2

The human visual system

It may be surprising to find a chapter on the human visual system in a book on digital image processing, but there are two reasons to include it:

- We need to be aware that there is a large difference between the image we *display* and the image we actually *perceive*. There is, for instance, a difference between the *luminance* of a pixel on a computer screen and the perceived *brightness* of this pixel: if we double the screen luminance, this does not imply that the perceived brightness is doubled too. The brightness also depends on other factors, such as contrast around the pixel and various cognitive processes¹.
- The human visual system can perform a number of image processing tasks in a manner vastly superior to anything we are presently able to do with computers. If we want to mimic such processing, we need to carefully study the way our eyes and brain do this.

2.1 The human eye

The human visual system consists of two functional parts, the eye and (part of the) brain. The brain does all of the complex image processing, while the eye functions as the biological equivalent of a camera.

Figure 2.1 shows a cross section of the human eye and identifies its most important parts. What our eyes perceive of a scene is determined by the *light rays* emitted or reflected from that scene. When these light rays are strong enough (have enough energy),

¹In this book we will use the term *luminance* for the actual physical brightness, and *brightness* for the perceived brightness of an object, pixel, etc.

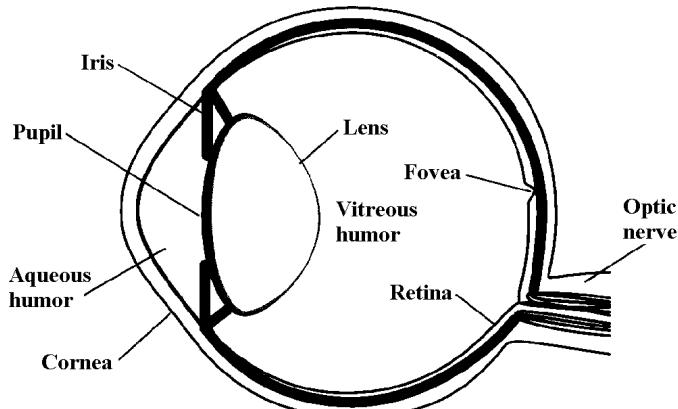


Figure 2.1 Cross section of the human eye.

and are within the right range of the electromagnetic spectrum (about 300 to 700 nm), the healthy eye will react to such a ray by sending an electric signal to the brain through the optic nerve.

When a light ray hits the eye, it will first pass through the *cornea*, then subsequently through the *aqueous humor*, the *iris*, the *lens*, and the *vitreous humor* before finally reaching the *retina*. The cornea is a transparent protective layer, which acts as a lens and refracts the light. The iris forms a round aperture that can vary in size and so determines the amount of light that can pass through. Under dark circumstances the iris is wide open, letting through as much light as possible. In normal daylight, the iris constricts to a small hole. The lens can vary its shape to focus the perceived image onto the retina.

In the retina, the light rays are detected and converted to electrical signals by *photoreceptors*. The eye has two types of photoreceptors: *rods* and *cones*, named after their approximate shape. The rods are abundant, about 100 million in a human eye, and spread evenly about the retina, except at the fovea, where there are almost none. The fovea is the area of the retina where our vision is sharpest. There are much fewer cones, about 6 to 7 million, which are mainly located around the fovea, but can be found in a low density in the entire retina. No photoreceptors are found at the point where the optic nerve attaches to the eye (the so-called *blind spot*), so we cannot perceive anything there. Since rods are more responsive to light than cones we can identify three types of vision, depending on the amount of light that reaches the eye. Under dark circumstances, practically only the rods are active. Since rods cannot discriminate colors, we perceive only shades of grey. We call this *scotopic* or night vision. Under daylight circumstances, the cones are most active, and we experience *photopic* or day vision. In dimly lit circumstances there is an intermediate stage where both rods and cones are active called *mesopic* vision.

We are able to distinguish colors because there are three distinct types of cones, each

sensitive to a different band of the electromagnetic spectrum. The relative sensitivity as a function of wavelength of the cones is shown in figure 2.2. We can see that one type

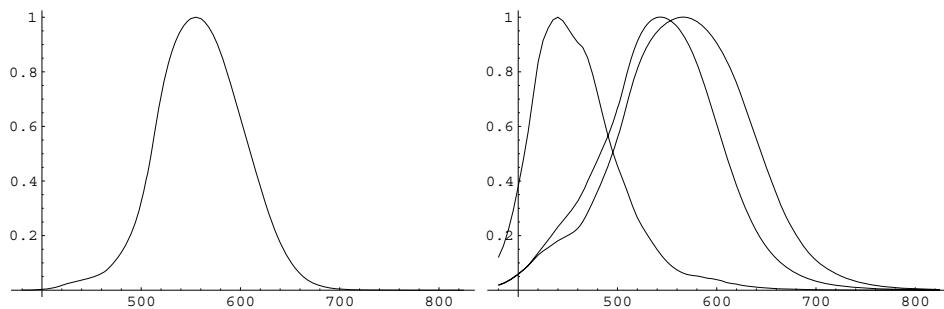


Figure 2.2 Photopic luminous efficiency function of the human visual system (left) and relative sensitivity of the three types of cones (right) as a function of wavelength in nm.

of cone is almost exclusively active in the region of 400 to 500 nm, and hence enables us to perceive colors in the violet-blue part of the spectrum². The other two types have only a slightly different response curve and enable us to see colors in the spectrum from cyan to red. By combining the response of the three types of cones in a small area of the retina we are able to perceive any color. This process is called *trichromacy*³.

Example

If light with a wavelength of 550 nm (i.e., green) is projected onto the retina, the response of the three types of cones (from left to right in figure 2.2) will respectively be 0.1, 0.975, and 0.975 of the maximum responses, which unique combination will be recognized as “green” by the brain.

Vice versa, a response of 0.0, 0.2, and 0.525 indicates the color red (around 640 nm).

Because of the trichromatic way the human eye operates, we also need three distinct colors to *reproduce* a certain color on a medium. For example, if you look up close at a TV screen, you will notice the screen is built up of tiny dots or stripes of either red, blue, or green.

²The visible spectrum of colors ranges from about 380 to 780 nm forming the familiar “rainbow” of colors going from violet to red. The primary colors of blue, green, and red are approximately found at the areas around 450, 550, and 650 nm respectively.

³From the Greek *chroma*, color.

2.2 Reflectivity and luminance

Suppose we have a light source that emits light rays with energy $E(\lambda)$, where λ is the wavelength of the emitted light, then the light I reflected from a certain object can be written as

$$I(\lambda) = \rho(\lambda)E(\lambda),$$

where $\rho(\lambda)$ is the reflectivity of that object. The reflectivity is a function that takes values between 0 and 1. Suppose $\rho(\lambda_1) = 1$, then this means that all of the light with wavelength λ_1 is reflected. Alternatively, if $\rho(\lambda_2) = 0$, this means none of the light at wavelength λ_2 is reflected, i.e., all of the light at this wavelength is absorbed by the object. Effectively, the reflectivity function $\rho(\lambda)$ determines the color of an object. For example: if a certain object reflects only light with wavelengths λ of about 650 nm (i.e., ρ is zero for other values of λ), we call it “red”.

The *luminance* L of an object is defined as

$$L = \int_0^{\infty} I(\lambda)V(\lambda) d\lambda,$$

where $V(\lambda)$ is the *luminous efficiency function* of a visual system. This function $V(\lambda)$ tells us how well a visual system is able to detect light of a certain wavelength. The typical luminous efficiency function of the human visual system is shown in figure 2.2. Obviously, we are best equipped to see light of a wavelength of 550 nm. At this wavelength, a light ray needs only a little energy to trigger a photoreceptor in our visual system. Alternatively, an electromagnetic ray with a wavelength of 1000 nm will not trigger anything, no matter how large its energy is.

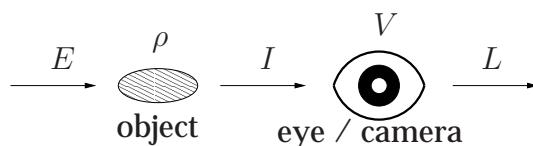


Figure 2.3 Schematic of object luminance.

Figure 2.3 shows a schematic of how object luminance relates to light energy, reflectivity, and luminous efficiency.

Intermezzo

Different visual systems will generally have different luminous efficiency functions. So what is “seen” by a rabbit, a snake, or a camera is generally different

from human vision, amongst others because their visual system reacts differently (either weaker or stronger) to light of certain wavelengths. In biological visual systems, the main cause for this is the variety in cones that can be found. Some insects have been shown to have one or two types (mono- and dichromatic vision). Some turtles have two types, but each type of cone can also contain four different kinds of colored oil, so in effect there are eight types of cone. Pigeons have at least three types of cone, and five oil colors. In the color area we call red, a pigeon has six active forms of cones, and so it can distinguish far more shades of red than humans can.

The difference in cones is only one of many when comparing biological visual systems. If biological vision research has shown anything, it is that almost any species has its own unique visual system in terms of color and luminance perception. Even in species that have the approximate trichromatic vision of humans –like honeybees or bumblebees– there are large differences. For example: bumblebees cannot perceive ordinary white light.

The upshot of this is that we must be careful in taking our perception of an image as a “gold standard”; what an animal, a camera, or, –by extension– a computer perceives in the same image may be something completely different.

Because of various physical and psychophysical reasons, the luminance L of an object is not the same as the perceived brightness. We will show some concepts of psychophysics in the next section. The purely physical reasons include adaptation effects: when we “dark adapt” our eyes (*i.e.*, stay in complete darkness for over an hour), our scotopic vision has a luminous efficiency function that is shifted in comparison to the normal photopic function of figure 2.2. So under these circumstances our eyes will perceive an object as brighter than when using photopic vision. Another adaptation effect is that after seeing a light flash, the cells in the retina need a short period of time to recover. The effect of this recovery is noticeable as the occurrence of so-called “after-images” if we look away from a brightly lit object.

2.3 Some psychophysics

The ability to detect a spot of light does not depend so much on the luminance of the spot itself as on the *difference* in luminance of spot and background, *i.e.*, the *contrast*. Of course, the luminance must be above some minimal value, but it is the contrast of spot and background that must be above a certain threshold before we can detect the spot. We call this threshold the *just noticeable difference*. *Weber’s law* states that the just noticeable difference ΔL is proportional to the background luminance L . In other words, the higher the background luminance, the higher the contrast needs to be before we detect a difference.

In addition to Weber's law there are a number of other contrast effects that influence the perceived brightness, like the *Mach band effect* and the *simultaneous contrast effect*. The Mach band effect is shown in figure 2.4: even though each bar is uniformly grey, our visual system enhances luminance changes. As the figure shows, overshoots and

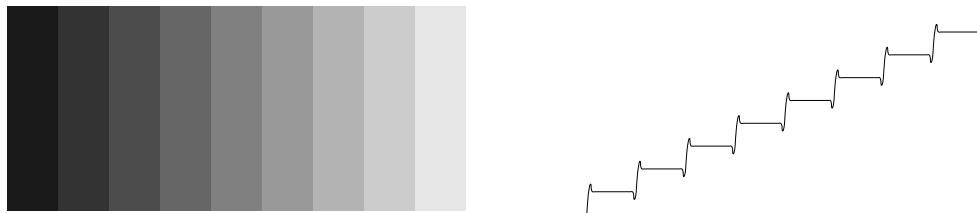


Figure 2.4 The Mach band effect: each bar is uniformly grey, but our visual system enhances each contrast jump. The graph at the right approximately shows the perceived brightness.

undershoots appear in the brightness graph: the bright side of a contrast jump appears to be extra light and the dark side appears extra dark. The Mach band effect shows us that our visual system sharpens the edges of the objects we perceive by adding a little contrast.

The *simultaneous contrast effect* is another effect that shows us that the perceived brightness depends on the contrast, in this case on the contrast with the local background. Figure 2.5 shows four squares of equal luminance, each with a background of different

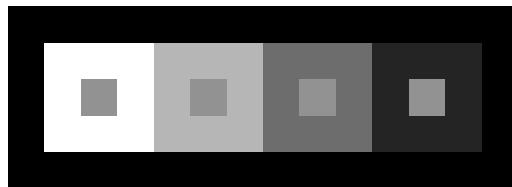


Figure 2.5 Simultaneous contrast effect. Even though the center squares are of equal luminance, they appear to be brighter if the background is darker. the one with the largest contrast to the background appears darkest.

luminance. As the background gets darker, the perceived brightness of the squares increases. Simultaneous contrast is not just a 'low-level' physiological phenomenon, but requires a complex decision of what exactly is "background" in the image. See, e.g., figure 2.6, the so-called *Benussi ring*, where a small change in the image suddenly changes our perception of background for the entire image.

Differences in brightness can even be induced by the mere *suggestion* of object edges, as shown in figure 2.7. Such effects are purely psychological in nature. Another psychological effect is the human tendency to view parts of images that appear to belong

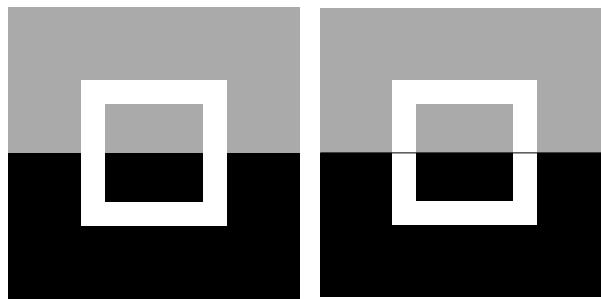


Figure 2.6 The Benussi ring. The brightness of the white ring in the left image appears to be uniform. In fact, the luminance *is* uniform. When we draw a small line as in the right image, the upper and lower parts of the ring are suddenly perceived as belonging to different backgrounds with different contrasts, and the upper and lower brightness is different.

together as a unit. This is a phenomenon known as *Gestalt*: an arrangement of separate elements whose configuration appears to be integrated is viewed as a single unit. The human visual system is uncanny in its ability in finding such units and their integrating patterns. This clustering ability works even in the absence of many object edges, as figure 2.8 shows. The integrating pattern determines what we perceive in such images. Figure 2.9 shows what happens if there are two obvious integrating patterns: we are not able to view both patterns simultaneously. Only with some difficulty are we able to make the *Gestalt switch* between the patterns.

Although explaining such psychological effects is beyond the scope of this book, it can be useful to acknowledge such effects when comparing human vision and computer vision.

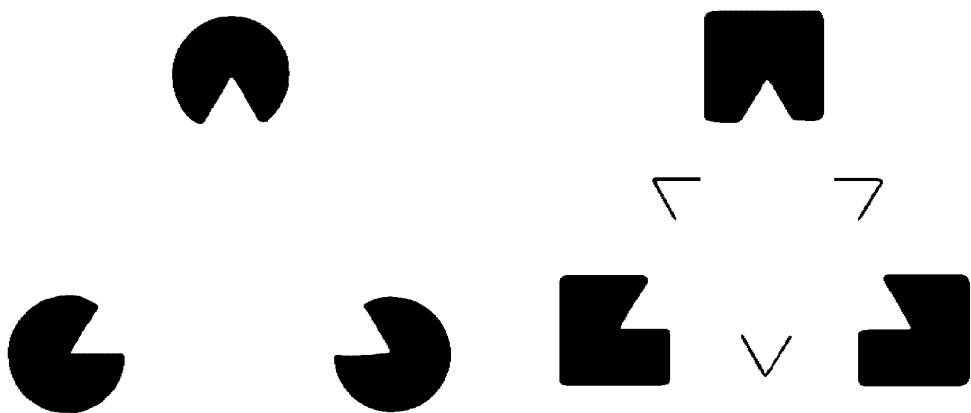


Figure 2.7 Examples of brightness differences induced by suggestion of edges.



Figure 2.8 Clustering ability of the human visual system. After a few seconds, this image of blobs contains a Dalmatian...



Figure 2.9 Example of images that contain two Gestalt patterns. Although the human visual system is able to detect a certain pattern very rapidly, we are unable to view both patterns simultaneously.

Chapter 3

Digital image acquisition

3.1 Digital sampling

In chapter 1 we already mentioned that when acquiring an image of a real scene it is discretized in two ways: *sampling* and *quantization*. Figure 3.1 shows sampling and quantization of a one-dimensional signal on a uniform grid¹. The signal is sampled at ten positions ($x = 0, \dots, 9$), and each sampled value is then quantized to one of seven levels ($y = 0, \dots, 6$). Sampling and quantization of images is done in exactly the same

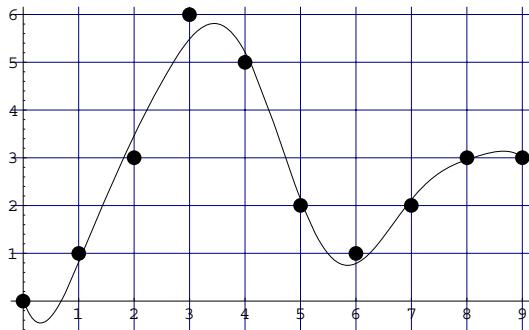


Figure 3.1 Sampling and quantization of a one-dimensional continuous signal.

way, except that sampling is now done in more than one dimension. An example is shown in figure 3.2. Here, we see a continuous signal of two variables (top left), and the corresponding image with signal strength converted to grey values (bottom left). On the right, the signal is shown after sampling on a discrete grid and quantization of the grey values onto five levels.

¹Note: even though this is a book about image processing, we will often use examples taken from one-dimensional signal processing wherever this simplifies matters.

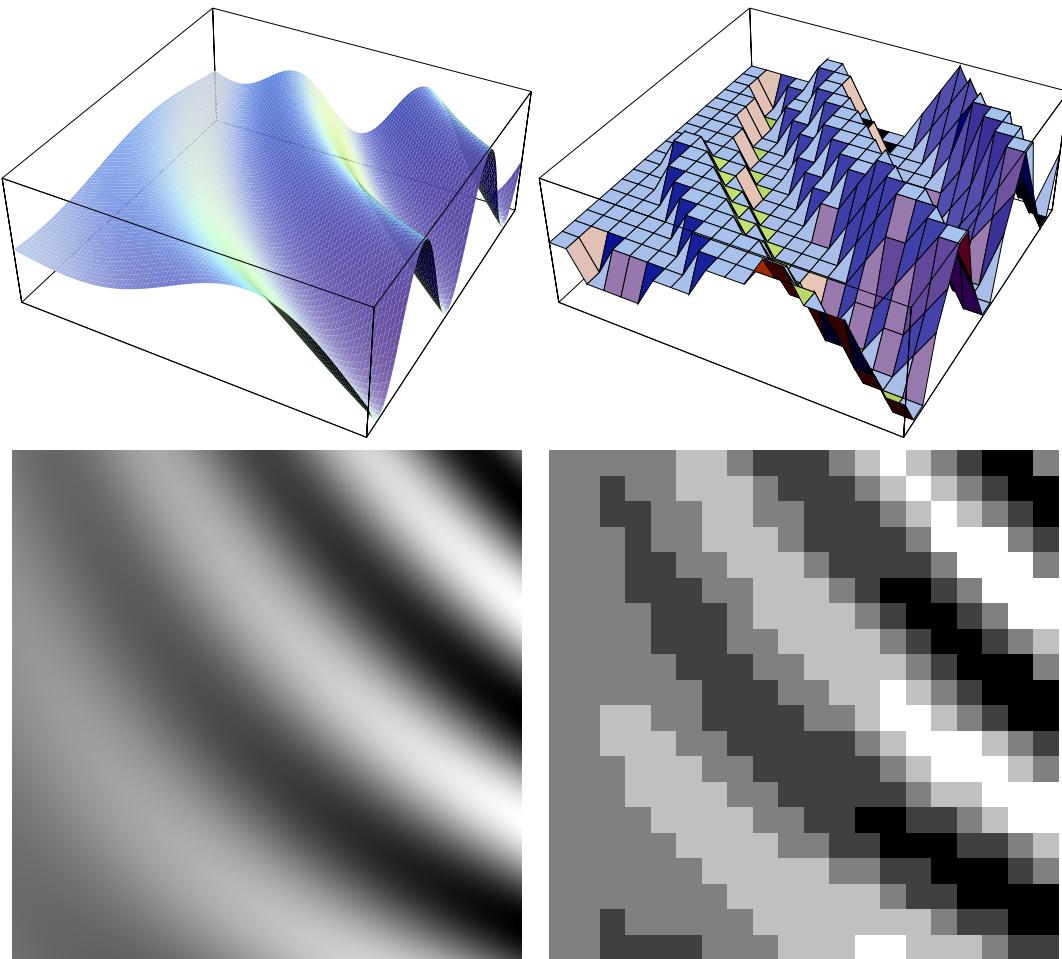


Figure 3.2 Sampling and quantization of a two-dimensional image.

By simply looking at figure 3.2, it is clear that the digital picture is not a very good representation of the original. Obviously the density of the sampling grid and the number of levels we choose in the quantization are important quality factors. These factors are called *resolution*: the *spatial resolution* equals the number of pixels used, and the *intensity resolution* equals the number of grey levels used. In digital images, both types of resolution are finite. The effect of lowering the spatial resolution can be seen in figure 3.3.

There is another kind of resolution called the *optical resolution*, which is the smallest spatial detail a visual system can see. Optical and spatial resolution are often used without their adjectives in texts, and what type of resolution is meant should be determined from the context.

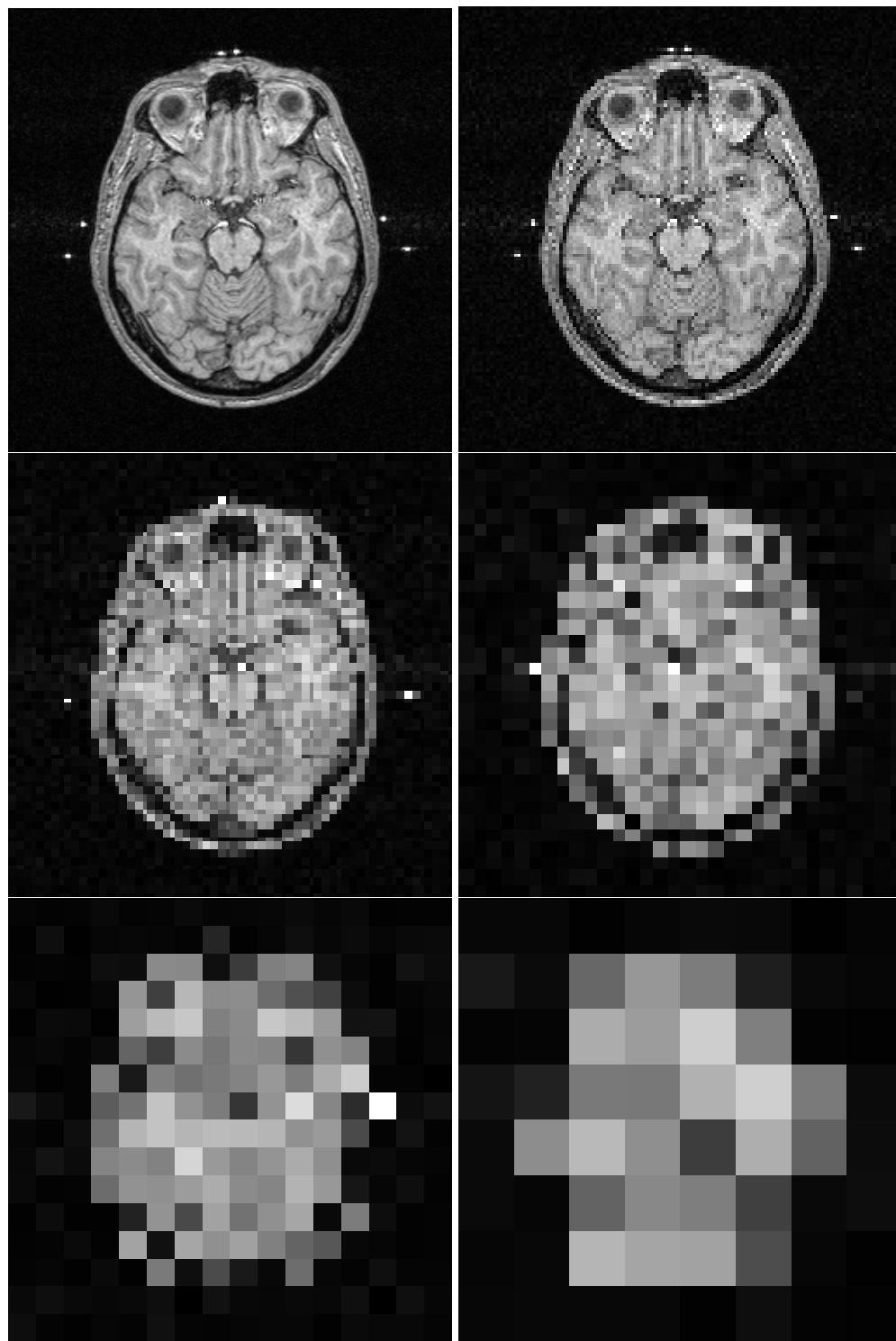


Figure 3.3 Effect of lowering the spatial resolution. From top to bottom and from left to right: original image with a resolution of 256×256 pixels, then resolutions 128×128 , 64×64 , 32×32 , 16×16 , and 8×8 .

3.1.1 The Nyquist frequency

If we sample an image, is it possible to make the image *exactly* the same as the original? And if so, how many samples are necessary? Before we can answer these questions, we need the concept of *spatial frequency*. The frequency of a sinusoid is defined as the

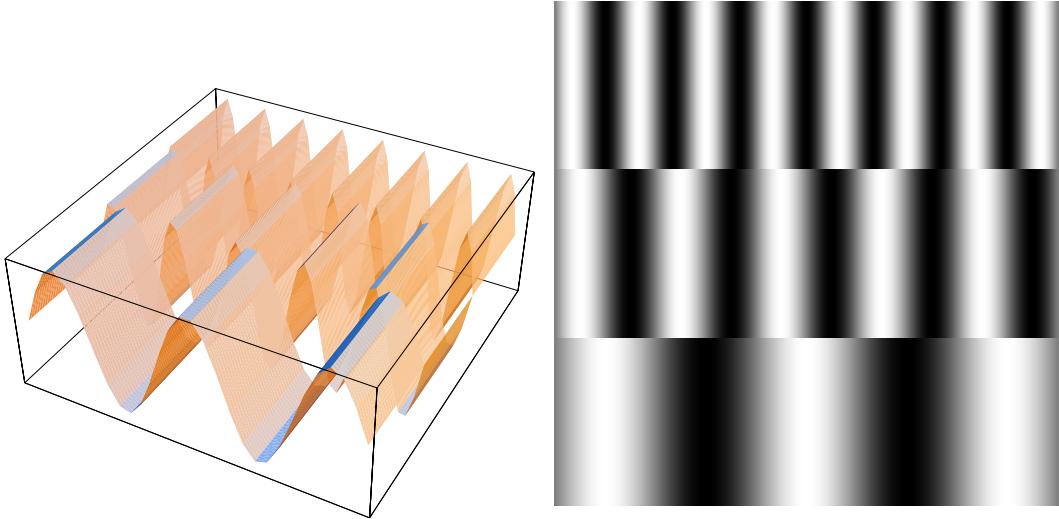


Figure 3.4 Example of three different spatial frequencies.

number of cycles it makes per unit length. For example, the family of functions $f(x) = \sin(ax)$ has frequency $\frac{a}{2\pi}$. Figure 3.4 shows three sinusoids of different frequency and the corresponding image. There is a relationship between spatial frequency and the level of detail in images: a high level of detail corresponds to a high frequency, and a lack of details corresponds to low frequencies. A high level of detail is made up of relatively large contrast, *i.e.*, the image intensity cycles fast from dark to light and vice versa, just like a high frequency sinusoid does. This relationship between frequency and image detail is at the heart of many image processing applications and will be explored further in the chapter on the Fourier transform.

We already noted that, in a digital image, the spatial and intensity resolution are finite. Hence the finest detail that can be represented is limited, and the range of spatial frequencies occurring in the image is also limited². This implies that the number of samples we must take of an image in order to exactly reproduce all the details is also finite. This observation is linked to spatial frequency by the *sampling theorem*:

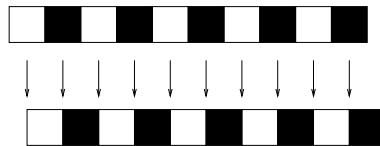
To capture the highest frequencies (*i.e.*, the smallest details) of a continuous image, the sampling rate must be $2\xi_n$ or faster, where ξ_n equals the highest frequency occurring in the original image. $2\xi_n$ is called the *Nyquist frequency*.

²Many books use “band” terms when discussing frequencies. If the frequency has a limited range, it is called *bandlimited*. The range of frequency values itself is called the *bandwidth*.

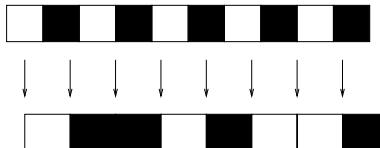
If we sample an image below the Nyquist frequency, details will be lost. Oversampling, on the other hand, will not increase the level of detail: all of the details are already captured when sampling exactly at the Nyquist frequency.

Example

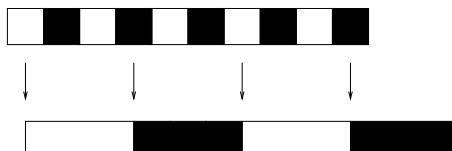
Consider the following line taken from an image:



The pattern has a frequency of $\frac{1}{2}$ per pixel –i.e., there are 2 pixels in a pattern cycle– so the Nyquist frequency is $2 \cdot \frac{1}{2} = 1$ per pixel. In the above, we have sampled the image line with this frequency, and, obviously, the result equals the original. If we lower the sampling frequency to $\frac{4}{5}$ per pixel –i.e., 4 samples in 5 pixels– details are lost:



If we lower the sampling frequency even further to $\frac{1}{3}$ per pixel, a pattern emerges that wasn't even present in the original image:



3.1.2 Aliasing

The phenomenon occurring at the end of the example above is called *aliasing*: if we sample an image at a rate that is below half the Nyquist frequency, high frequency patterns in the original are translated to lower frequency patterns that are absent in the original image. In the example above, we see that a pattern with frequency $\frac{1}{2}$ per pixel is mapped to a pattern with frequency $\frac{1}{6}$ per pixel.

Example

Figure 3.5 shows another example of aliasing. We sampled the function $f(x, y) = \sin(2\pi x(26 - y))$ on the domain $D_f = \{(x, y) \in [0, 1] \times [1, 25]\}$, which has decreasing frequency as we move from the top to the bottom line: frequency 25 per line on the top line, and frequency 1 per line on the bottom line. The function was sampled four times using different sampling frequencies. The aliasing effect at the lowest sampling frequency is so strong, that the high image frequencies at the top of the image get mapped to the same frequencies as the low ones at the bottom of the image.

The examples above are extreme cases of aliasing that are unlikely to occur in practice. Aliasing pops up as an artifact with many imaging applications, however. Telltales are the occurrence of Moiré or repetitive patterns in images. A notorious example is the sampling of printed images, e.g., a picture from a newspaper, where the sampling frequency must be carefully adapted to the frequency used in the printing process, or low frequency artifacts will occur. Aliasing effects may not always be visible to the human eye, but still prove to be disastrous to a computer image processing application. It is therefore wise to check the original and sampling frequencies against possible aliasing.

3.1.3 Sampling and convolution

Until now we have assumed that we can sample an image with infinite precision. In practice this is an impossibility, since practical imaging devices cannot have an infinite precision. For example: if we use a camera to take samples of some infinitely small image location (x, y) , then the camera will gather its sample in a small neighborhood around (x, y) . There is in fact another reason why sampling with infinite precision is impossible: cameras need the energy from incoming light rays to detect these rays. Even if it were possible to point the camera with infinite precision at only the infinitely small point (x, y) , the energy of the incoming light rays would be reduced to zero, and we would detect nothing. There is a tradeoff here: on the one hand we would like our sampling to be as precise as possible, so a small sampling area, but on the other hand the sampling area cannot be too small, or it will not give off enough energy for proper detection.

Example

The diaphragm of a photocamera controls the size of the sampling area on the photosensitive film. The smaller the diaphragm, the smaller the sampling area, and the sharper the picture will be. However, if we make the diaphragm too small,

not enough light will be able to pass through the diaphragm to light the film properly. In practice, the photographer will choose the smallest diaphragm possible considering the ambient light.

To keep the diaphragm as small as possible, a photographer will often try to increase the energy that reaches the film by other means, for instance by extra lighting –using lamps or a flashlight– or increasing the exposure time of the film.

An important tool in describing the sampling process is mathematical *convolution*:

The convolution $f * g$ of two integrable functions $f, g : \mathbb{R} \rightarrow \mathbb{R}$, is defined as:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(a)g(x - a) da.$$

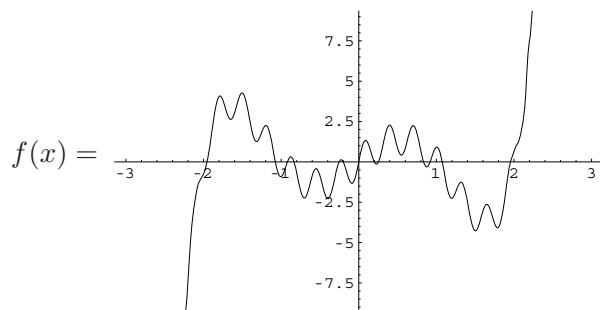
If f and g are two-dimensional images, convolution is defined as

$$(f * g)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(a, b)g(x - a, y - b) da db,$$

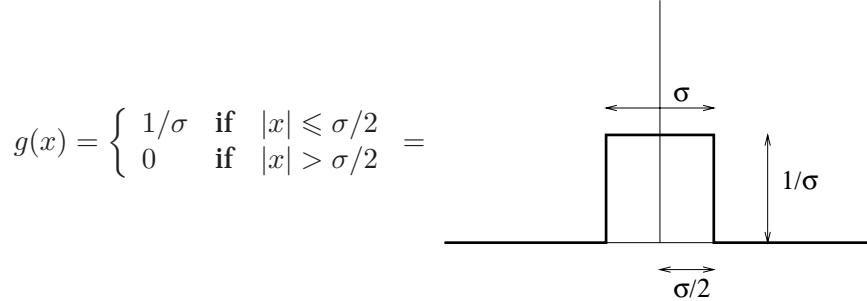
and so on for more dimensions. Convolution is a commutative operator, *i.e.*, $f * g = g * f$.

Example

To understand convolution, we will work out a one-dimensional example. Suppose f is some signal, for instance:



We define $g(x)$ by



In this example, the convolution equals:

$$(g * f)(x) = \int_{-\infty}^{\infty} g(a)f(x-a) da = \frac{1}{\sigma} \int_{-\sigma/2}^{\sigma/2} f(x-a) da. \quad (3.1)$$

The effect of this integral is to take the average of all the values of $f(x)$ in a neighborhood of size σ around x . The effect of this “local averaging” is that the values of f get “smeared out” over a certain region. Figure 3.6 shows $f * g$ for some values of σ . The effect of the convolution is that f gets smoothed: small details of f vanish³. The larger we choose σ the more details of f vanish.

In cases like in the above example, where f is a signal or image, and g is a function with a narrow support (i.e., has significant values only in a small region), we call g the *convolution kernel*. Note that the area under the curve of g in this example is always 1.

If σ tends to zero, g in the example above transforms into a peculiar function that has value zero everywhere, except at $x = 0$, where there is a spike whose value tends to infinity. The area under the curve of g is still 1, however. This special function is known as the *Dirac or delta function* $\delta(x)$:

$$\begin{cases} \delta(x) = 0 & \forall x \in \mathbb{R} \setminus 0 \\ \int_{-\infty}^{\infty} \delta(x) dx = 1. \end{cases}$$

The delta function has a special property, called the *sifting property*:

$$(\delta * f)(x) \stackrel{\text{eq.3.1}}{=} \lim_{\sigma \rightarrow 0} \frac{1}{\sigma} \int_{-\sigma/2}^{\sigma/2} f(x-a) da = f(x),$$

³Since small details correspond to high frequencies, we call this type of convolution a *low pass filter*, because it lets only low frequencies pass, and filters away high frequencies.

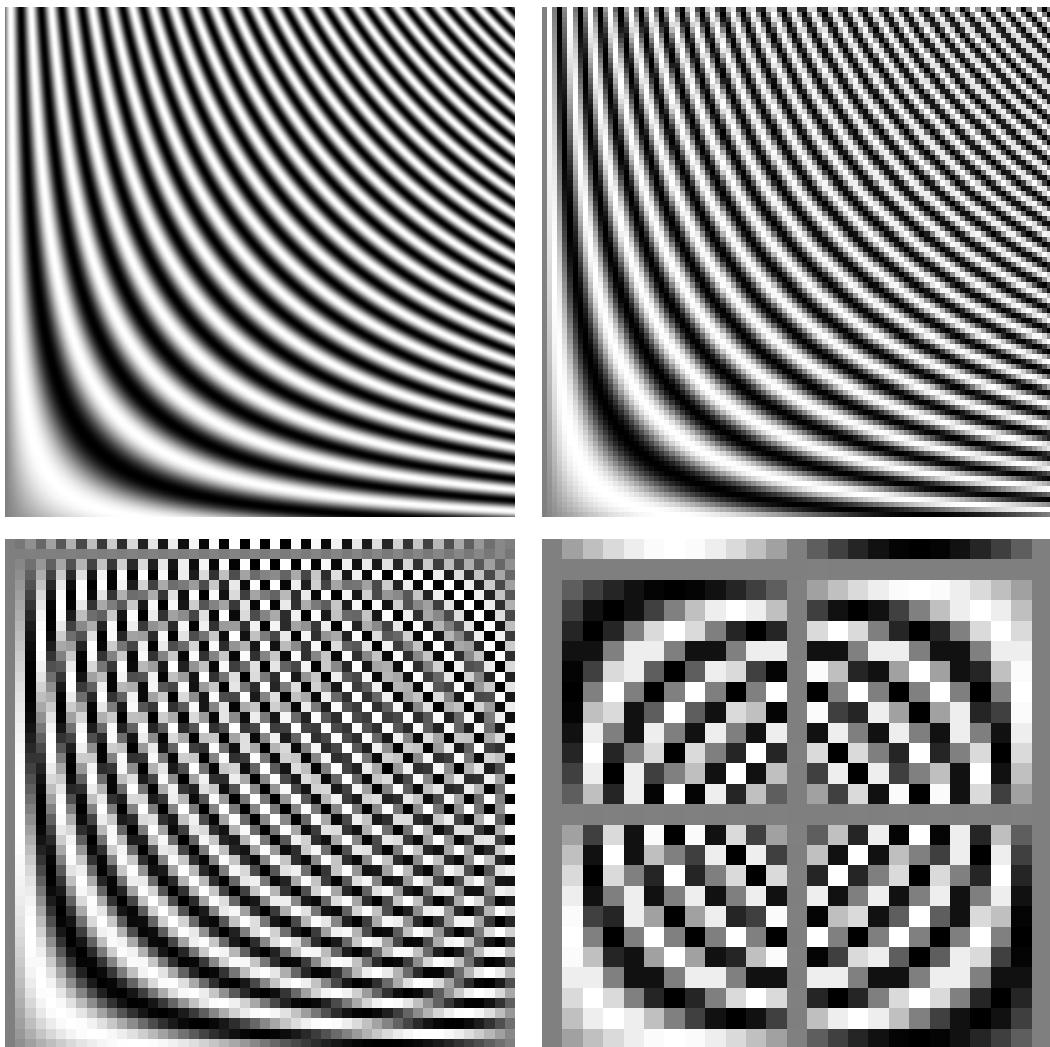


Figure 3.5 Example of aliasing. The same two-dimensional function was sampled four times: using 400×400 points (top left), 100×100 points (top right), 50×50 points (bottom left), and 25×25 points (bottom right).

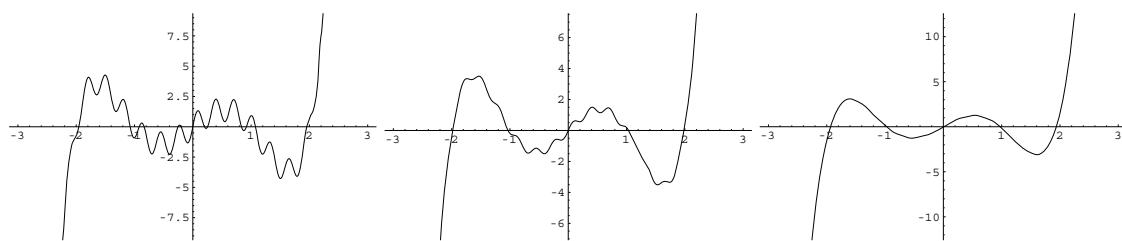


Figure 3.6 Examples of convolution of a signal with a rectangular convolution kernel (see text for details). Left: original signal, middle: convolution with $\sigma = 0.25$, right: convolution with $\sigma = 0.6$.

so convolving with a delta function does not alter the function f . This was to be expected, since we said convolving f with our choice of kernel g results in smoothing of f . The larger we choose σ , the more f gets smoothed. So by the same argument, the *smaller* we choose σ , the more the convolution result resembles f . If σ tends to zero, g transforms to the delta function, and f does not change at all.

We mentioned earlier that imaging devices do not have infinite precision, so sampling a signal at a certain location is done by taking a small neighborhood around that location into account. But this is exactly what convolving f with some convolution kernel g does! We now have all the ingredients we need to describe the sampling process in a mathematical way:

The value of a sample is the result of the convolution of the unknown signal or image f with some convolution kernel g . The kernel g depends only on the imaging device. In the ideal case, the kernel g is infinitely narrow and equals the delta function δ .

In practice, we are usually able to determine the specific kernel g of imaging devices, so we are able to determine its behavior when sampling a variety of input images f .

The convolution effect of the rectangular kernel g in our example is that the image f gets smoothed; each value $f(x)$ was replaced by the local average of values in a neighborhood of size σ . It is often easy to predict the convolution result from the shape of the kernel, see table 3.1 for examples.

The last two kernels in table 3.1 have a realistic appeal: like the rectangular kernel they compute local averages of $f(x)$, but in a weighted way: values close to x have a higher weight than values further away. This is what we expect of imaging systems like cameras: the sampled value is an average of values in a small region, but the center of this region has a larger impact on the result than the edges have.

The last kernel in the table is called the *Gaussian* function, $g(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{x}{\sigma})^2}$, which is especially useful in modeling realistic kernels for many imaging devices. It also has a number of interesting theoretical properties that we will come back to later in this book.

3.1.4 Extension to two dimensions

In the previous section we have seen examples of one-dimensional kernels and their effect on one-dimensional signals. We will now show a few examples of two-dimensional kernels, *i.e.*, kernels working on two-dimensional images. In the one-dimensional case we found that the area under the kernel equals one. This also holds in the two-dimensional case, or rather, the *volume* under the kernel equals one. In general, integrating

the kernel over all of its variables should result in one. Often, two-dimensional kernels are *symmetrical* in their variables, *i.e.*, swapping the variables will not alter the kernel formula. Many imaging systems have this property.

For example, the symmetrical two-dimensional version of the rectangular kernel (first kernel in table 3.1) would be

$$g(x, y) = \begin{cases} 1/\sigma^2 & \text{if } |x| \leq \sigma/2 \text{ and } |y| \leq \sigma/2 \\ 0 & \text{otherwise} \end{cases}$$

Further examples: the equivalent of the ideal sampling function $\delta(x)$ in two dimensions is $g(x, y) = \delta(x)\delta(y)$, and the equivalent of the Gaussian kernel $\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2}$ is $g(x, y) = \frac{1}{\sigma^2 2\pi} e^{-\frac{1}{2}\frac{x^2+y^2}{\sigma^2}}$

In all cases, it can be verified that the volume under the kernel equals one:

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(x, y) dx dy = 1.$$

Figure 3.7 shows an example of a two-dimensional Gaussian kernel.

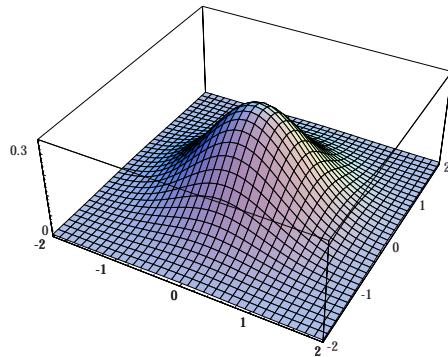


Figure 3.7 Example of a two-dimensional Gaussian kernel with $\sigma = 0.7$.

3.1.5 The point spread function

To find out what the convolution kernel of an imaging system looks like, we often examine the *point spread function* (PSF) of the system⁴. The PSF is defined as the image produced by the system if we present it with a delta function (a “point”). In terms of

⁴We can often assume the PSF and the convolution kernel of the system to be identical.

images, this means that we expose our camera to an image that is fully dark everywhere, except at one infinitely small spot, where the luminance is infinite. In practice, we cannot produce such an image, so we use the next best thing, which is an image that is dark everywhere, but has the highest luminance we can achieve at a very small spot. Figure 3.8 illustrates this.

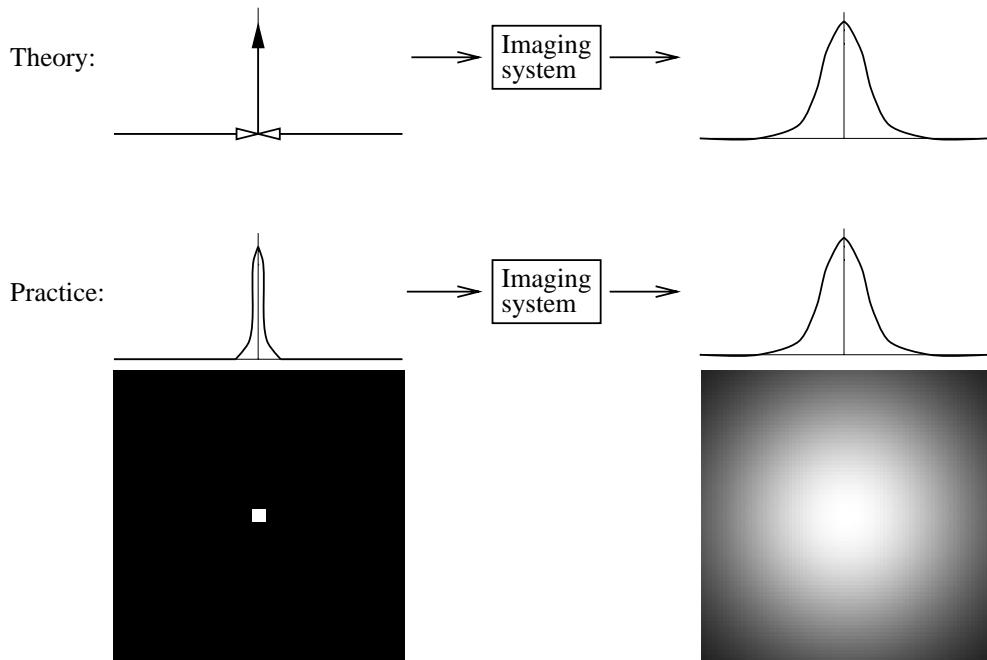


Figure 3.8 Establishing the PSF of an imaging system. Left column: the delta function or its practical approximation. Right column: the resultant PSF.

The broader the PSF, the more the original image will be smoothed by the imaging system. Figure 3.9 shows an example of images that have been smoothed using a Gaussian convolution kernel.

3.2 Quantization

After sampling, the image is quantized using a finite number of grey values. With most imaging systems, the quantizer is a digital circuit that converts the sampled value to an n -bit word. Since an n -bit word can take 2^n different values, the number of possible grey values equals 2^n . Figure 3.10 illustrates this. This figure also shows the effect of using different values of n on the grey scale: the larger n is, the smoother the grey value transitions are. Figure 3.11 shows the effects of reducing n on a real image. In practice, n

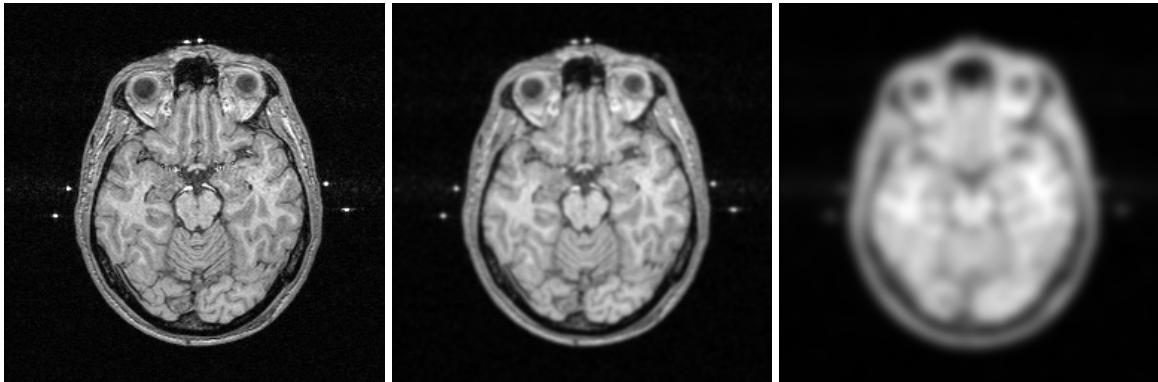


Figure 3.9 Example of smoothing of an image with a Gaussian PSF. Original image with a resolution of 256×256 pixels (left), after convolution with a Gaussian with $\sigma = 1$ pixel (middle), and $\sigma = 4$ pixels (right).

varies from 1 to about 16. For reproducing grey valued images intended for humans, a value of $n = 8$ (256 grey values) is usually sufficient. Medical scanners like a CT scanner have 12 or more bits of quantization.

Although a large n will give us a smooth grey scale, it is not always useful to make n larger. For one thing, a larger n means that quantizing a sample will take more time. In many cases, the effects of a slow quantization with a large n are worse than using a fast quantization with a smaller n . Another reason is that increasing n only makes sense if the original signal has enough detail to merit this. For example: suppose we have a grey valued image with 256 grey values (*i.e.*, $n = 8$), then it would be pointless to scan this image with $n > 8$. In practice, there is always an upper bound for n above which it is pointless to acquire an image.

The number of bits n used in the quantization can only be used to determine the number of grey levels that can be encoded using those bits. The actual grey value also depends on the encoding scheme used. For example, if we use $n = 32$ (four bytes), we can use an “integer” scheme to encode values in the range $\{-2^{31}, \dots, 2^{31} - 1\}$, an “unsigned integer” scheme to encode values in the range $\{0, \dots, 2^{32} - 1\}$, or a “floating point” scheme to encode real numbers in an implementation dependent range.

3.3 Color models

Up until now, we associated a single value with each pixel, that represented the pixel’s grey value (luminance). But how can we represent color images? We have already seen in chapter 1 that we can reproduce any color using a mixture of the primary colors red, green, and blue. And indeed the most common way to represent a color image is to

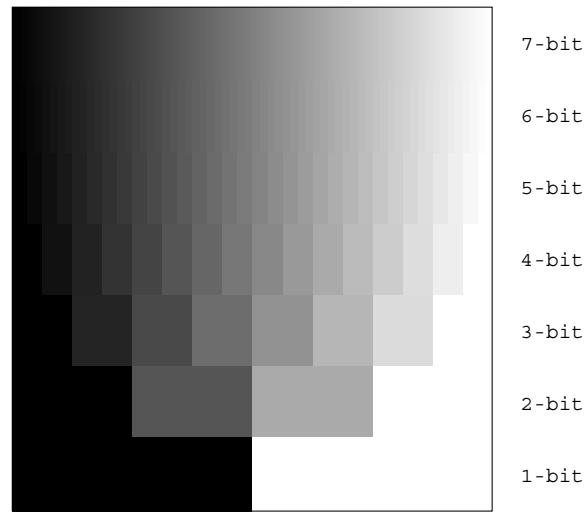


Figure 3.10 The grey scales available using n -bit quantization, with $n = 1, \dots, 7$.

associate *three* values with every pixel, namely the red (700 nm), green (546.1 nm), and blue component (435.8 nm) of the pixel's color. This color model is called the *RGB* model for color representation.

Example

Below are the red, green, and blue color components (each scaled to be between 0 and 1) of a single 3×5 color image:

R			G			B		
1	0	0	0	1	0	0	0	1
0.1	0.5	0.9	0.1	0.5	0.9	0.1	0.5	0.9
0	1	1	0	1	1	0	1	1
0.1	0.5	1	0.1	0.5	1	0	0	0
0	0	0	0.1	0.5	1	0.1	0.5	1

And the composite color image will look like: (note that equal values of the R,G, and B components will result in a shade of grey)

pure red	pure green	pure blue
dark grey	medium grey	light grey
black	pure white	pure white
dark yellow	medium yellow	pure yellow
dark cyan	medium cyan	pure cyan

The colors red, green, and blue are called the *primary* colors of the RGB model. Their mixtures cyan (blue mixed with green), magenta (red and blue), and yellow (red and green) are called the *secondary* colors.

Intermezzo

The RGB model is an *additive* model: pure red, green, blue mixed together form white. Additive models are logical models to use in case of computer or TV screens, or any device that *emits* light. They cannot be used in cases where we use *reflected* light. For example: if we mix paint, the combination of red, green, and blue will be black instead of white. The reason for this is that, e.g., green paint looks green to us because it only reflects the green part of the spectrum, and does not reflect the red or blue parts. Similarly, red paint does not reflect the blue or green parts of the spectrum. Yellow paint reflects the green and red parts, but stops the blue part. We see that the model here is *subtractive* rather than additive: yellow paint *subtracts* the blue part from white light, and cyan paint subtracts the red part from white light, so if we mix yellow and cyan, we subtract both the blue and red parts, and end up with green paint. The most common subtractive model is the CMY model (cyan, magenta, and yellow). The RGB additive model and CMY subtractive model are shown in figure 3.12. Note that the primary colors in the RGB model are secondary in the CMY model and vice versa.

The CMY model is commonly used in printing: by mixing C, M, and Y inks any color can be created. In practice, however, creating black by mixing C, M, and Y is tricky (very hard to balance right), and expensive, so a fourth, black, ink is used. This four color printing model is called the CMYK model.

Besides color models whose components describe the value of a specific primary color, there are models that have three entirely different variables: the most common amongst these is the *HSV*, or *hue, saturation, value* model, see figure 3.13. The *hue* identifies the color itself. Hue is usually represented as an angle in a color circle, where $H = 0^\circ$ corresponds to red, $H = 120^\circ$ to green, $H = 240^\circ$ to blue, and all other colors are embedded additively between them like a rainbow. The *saturation* identifies how pure the color component is: a fully saturated color ($S = 1$) is a pure color, and if the saturation is zero ($S = 0$), the color component is fully absent, and we just have a shade of grey: the dotted line $S = 0$ in the figure represents the “greys”. The *value* refers to the luminance value of the color.

The figure shows the peculiar double-cone structure of HSV-space: there is a single point “black” at the bottom, and “white” at the top. At a first glance, we may expect the H , S , and V coordinates to describe a cylindrical space, but it turns out that values

outside of the double cone are impossible, or simply map to the same color as a point on the edge of a cone. The bottom cone degenerates to a single point “black” simply because any point with a luminance value $V = 0$ is black, no matter what the color or saturation components are. The top cone degenerates to a single point “white” because to reach maximum luminance on, e.g., a computer screen we need all three color components (the red, green, and blue dots on the screen) to radiate at maximum luminance, hence the point with maximum luminance value V is always white.

There are many other color models, most of which are simple variants of the RGB, CMY, or HSV models. The models described here are the most common in image processing.

Convolution kernel g	Graph of g	Convolution effect on f
$\begin{cases} 1/\sigma & \text{if } x \leq \sigma/2 \\ 0 & \text{if } x > \sigma/2 \end{cases}$		$f(x)$ replaced by local average of f in a neighborhood of size σ
$\delta(x)$		$f(x)$ unaltered (ideal sampling)
$\frac{1}{2}(\delta(x+1) + \delta(x-1))$		$f(x)$ replaced by the average of $f(x-1)$ and $f(x+1)$
$\begin{cases} \frac{2}{\sigma} - \frac{4x^2}{\sigma^2 x } & \text{if } x \leq \sigma/2 \\ 0 & \text{if } x > \sigma/2 \end{cases}$		$f(x)$ replaced by weighted average of f in a neighborhood of size σ ; values closer to x have higher weight
$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{x}{\sigma})^2}$		$f(x)$ replaced by weighted average of f in a neighborhood of size σ ; values closer to x have higher weight

Table 3.1 Some examples of convolution kernels and their effects.

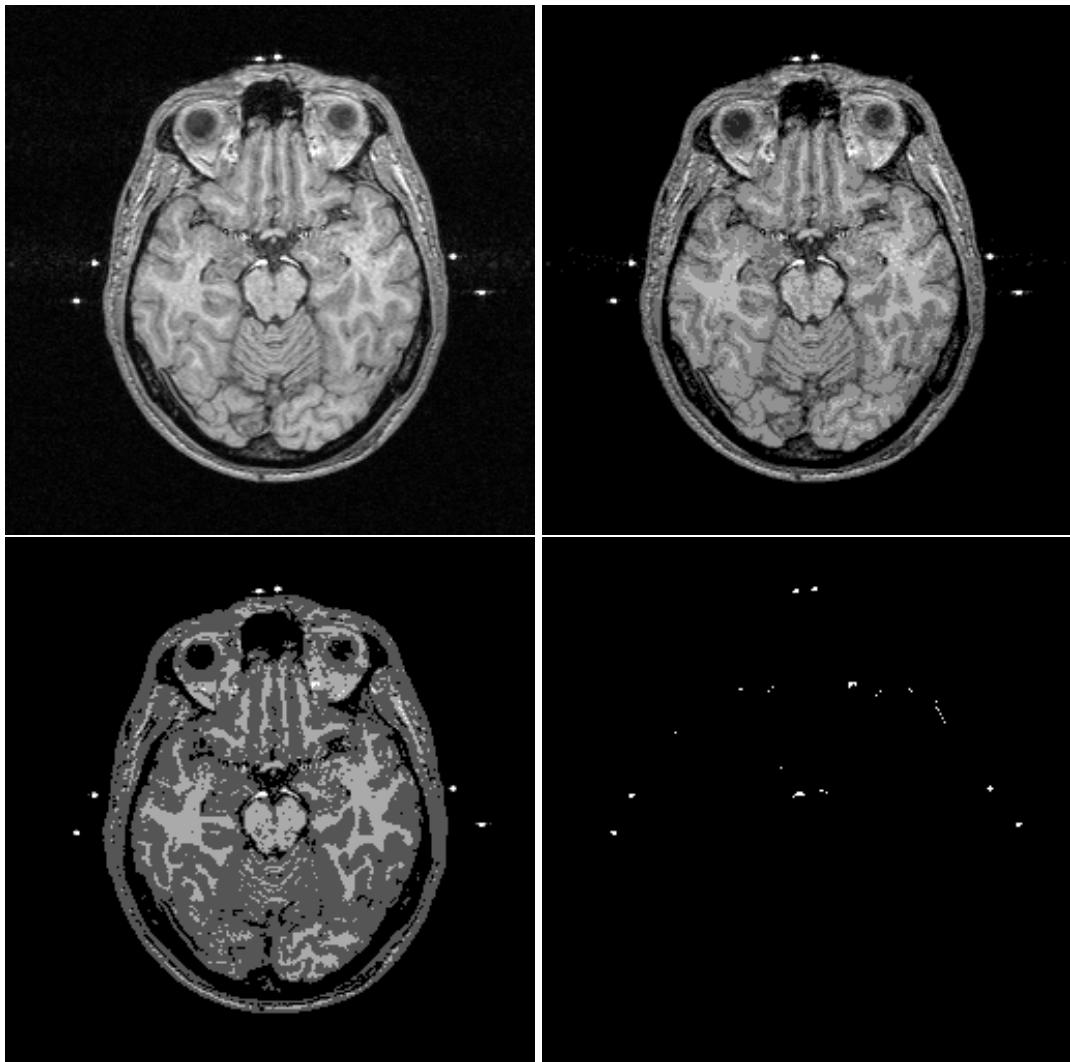


Figure 3.11 Effect of reducing the quantization level on a real image. From left to right and top to bottom: original image with $n = 8$ (256 grey levels), $n = 3$ (8 levels), $n = 2$ (4 levels), and $n = 1$ (2 levels).

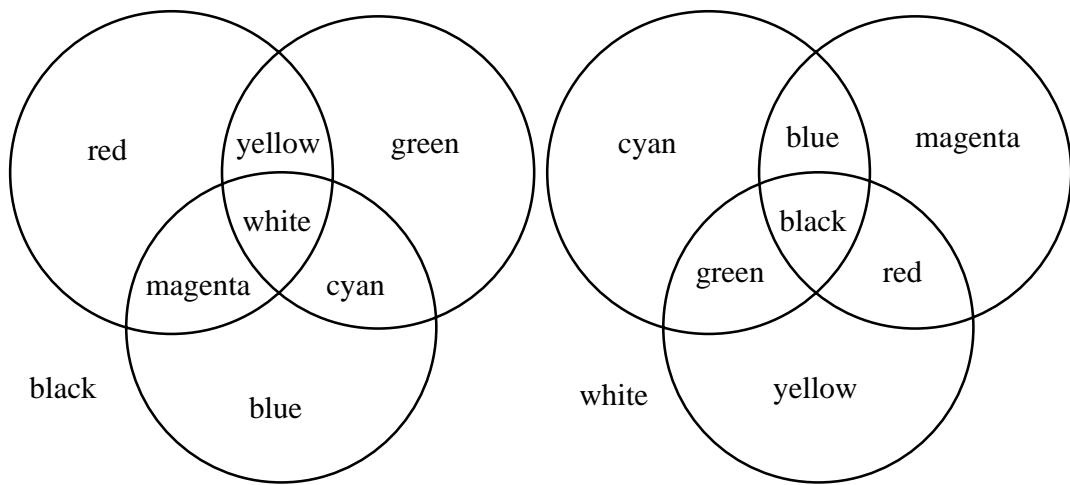


Figure 3.12 The additive (emissive) RGB color model (left), and the subtractive (reflective) CMY color model (right).

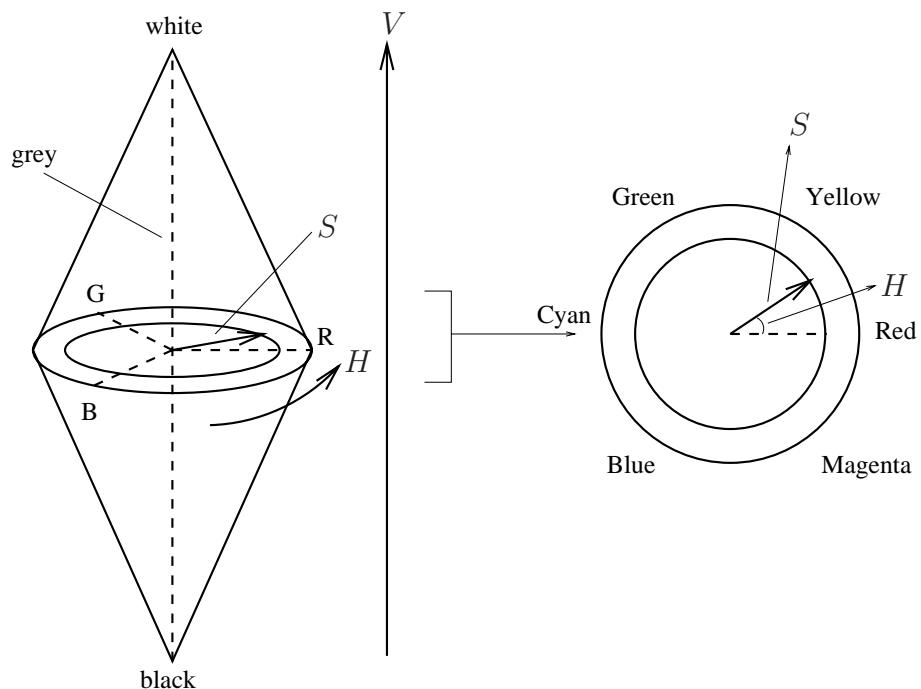


Figure 3.13 The HSV (hue, saturation, value) color model. Black, white, the pure colors red (R), green (G), blue (B), and the line of greys are indicated. A point in HSV-space is identified by three cylindrical coordinates: height V , angle H , and radius S .

Chapter 4

Operations on grey values

In chapter 1 we used an image processing example that everybody is familiar with: the adjustment of contrast and brightness on a TV screen. This is in fact an example taken from a very basic class of image processing tasks that perform *operations on grey values*. Such operations do nothing but look at the grey value of each pixel, map it to a new value, and move on to the next pixel. In mathematical terms, such an operation can be represented as a function $g(v)$ that operates on the pixel value $v = f(x, y)$ of image f at location (x, y) .

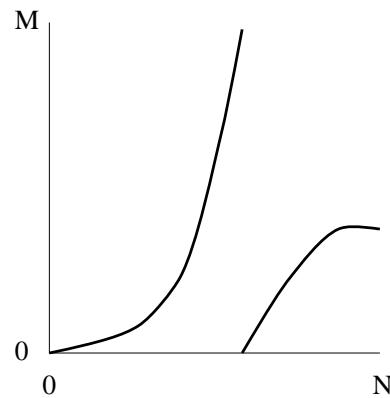
Example

Examples of simple grey value operations are $g_1(v) = v + 1$ and $g_2(v) = v^2$. Suppose we have an image $f(x, y) = \begin{array}{|c|c|c|} \hline -2 & -1 & 0 \\ \hline 0 & 1 & 2 \\ \hline \end{array}$, then $G_1(x, y) = g_1(f(x, y)) = \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline 1 & 2 & 3 \\ \hline \end{array}$, and $G_2(x, y) = g_2(f(x, y)) = \begin{array}{|c|c|c|} \hline 4 & 1 & 0 \\ \hline 0 & 1 & 4 \\ \hline \end{array}$.

Even though grey value operations are amongst the simplest of image processing tasks, it turns out that such operations are often extremely useful and powerful, and are at the heart of many image processing applications.

4.1 Examples of simple grey value remapping

It is often attractive to represent the grey value remapping function g as a graph, e.g., this one:



Here we have assumed the original grey values to be in the range $\{0, \dots, N\}$ and the remapped values in the range $\{0, \dots, M\}$. For displayed images, these values are often $N = M = 255 (= 2^8 - 1)$. For medical images, N is usually $4095 (= 2^{12} - 1)$ or $16383 (= 2^{14} - 1)$.

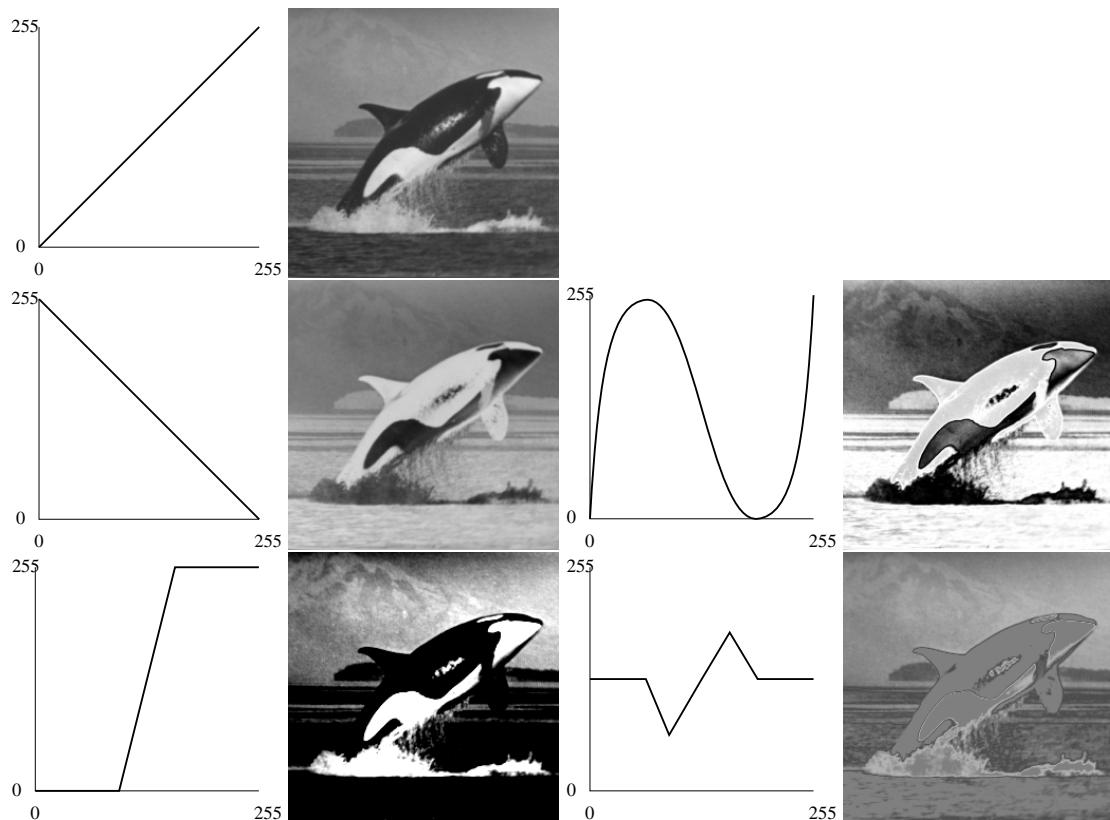


Figure 4.1 Examples of grey value remappings.

Figure 4.1 shows some examples of remapping applied to an image of an orca. The top row shows the original image. The original grey value range is $\{0, \dots, 255\}$. Since the

grey values have not been modified here, the corresponding graph shows the identity function $g(x) = x$. The left image on the second row shows the *inverse* of the image: the highest grey value is mapped to the lowest, and vice versa. The grey values have been remapped according to $g(x) = 255 - x$. Note that the “middle” grey value is not changed. In the left image on the third row we have mapped all of the low (dark) grey values to 0 and all of the highest (brightest) values to 255. This results in large patches of the image becoming uniformly black or white. The middle range of greys –about 100 to 150– is now stretched to map to the full range of 0 to 255. This causes small grey value differences in this range to be enlarged. For instance, we see some contrast emerging in the background of the image. Hence this particular mapping of grey values is called *contrast stretching* (more precisely: *windowing*). The right image on the third row shows an example where we have eliminated all of the brightest and darkest grey values. The resultant image contains only dull shades of grey. Because both the original lowest and highest grey values now map to medium grey, the large typical contrast of the orca is almost completely gone.

4.2 Gamma correction

When we display an image onto a computer or TV screen, the relation between the grey value of a pixel and the actual luminance value on the screen is generally not linear. This is because the response of the CRT¹ in terms of luminance is not linear in the voltage applied to the tube. Assuming we have grey values normalized between 0 and 255, the grey values will typically be remapped according to one of the functions of figure 4.2. The actual remapping function of a specific CRT is called the *display transfer function*

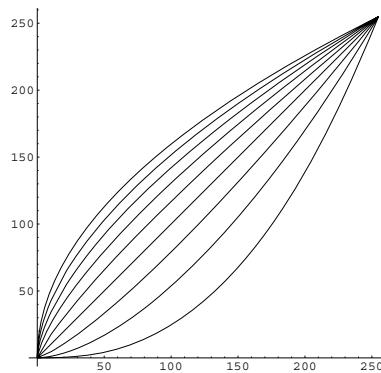


Figure 4.2 Typical Display transfer functions for a CRT.

(DTF). In the ideal case, the DTF is simply the identity function. In practice, however,

¹Cathode ray tube.

the DTF may differ considerably from the identity. Fortunately, this is hardly ever a problem. The deviation will have to be uncommonly large before, *e.g.*, humans notice that a movie on TV looks “unnatural”.

For some applications, however, it is vital that the DTF be as linear as possible. Although we can do a lot by using a very good CRT, the perfect CRT hasn’t been invented yet. To get a truly linear response from the CRT, we correct the grey values to be displayed using *gamma correction*²:

The DTF can usually be described quite accurately by $g(x) = N(x/N)^{1/\gamma}$, assuming black $\equiv 0$ and white $\equiv N$. The graphs of g for $\gamma \in \{0.4, 0.6, \dots, 2.0\}$ can be seen in figure 4.2. Suppose we have established the value of γ experimentally for a certain CRT, then we can achieve an exact linear response by first “pre-”mapping the grey values according to $g^{-1}(x)$ before displaying the image.

Example

If the DTF equals $g(x) = N(x/N)^{1/\gamma}$, then $g^{-1}(x) = N(x/N)^\gamma$. Suppose $N = 255$ and $\gamma = 1.2$ for a certain CRT, then our pre-mapping function equals $g^{-1}(x) = 255(x/255)^{1.2}$. Let’s verify that this gives correct results for a sample image:

0	1	2		0	0.33	0.76		0	1	2
100	0	160		82.93	0	145.76		100	0	160
0	127	255		0	110.47	255		0	127	255

pre-mapping

0	0.33	0.76		0	1	2
82.93	0	145.76		100	0	160
0	110.47	255		0	127	255

CRT

0	1	2
100	0	160
0	127	255

4.3 Contrast stretching

In the previous sections, we have used the term *contrast* in an intuitive manner. Formally, the contrast c between two grey values x_1 and x_2 can be defined as the absolute value of their difference: $c(x_1, x_2) = |x_1 - x_2|$.

In figure 4.1 we have already seen an important use of grey value remapping: by contrast stretching we were able to distinguish details in the background that could not be detected before. A common definition of a remapping that does contrast stretching is:

$$g(x) = \begin{cases} \alpha x & \text{for } 0 \leq x < a \\ \beta(x - a) + g_a & \text{for } a \leq x < b \\ \gamma(x - b) + g_b & \text{for } b \leq x \leq N, \end{cases}$$

assuming the original grey value range to be $\{0, \dots, N\}$. Figure 4.3 shows this graphically. The slopes α , β , and γ of the three portions of the graph determine the amount of

²The term *gamma* is an historic relic, which was used to describe the slope of the exposure versus density function in photography.

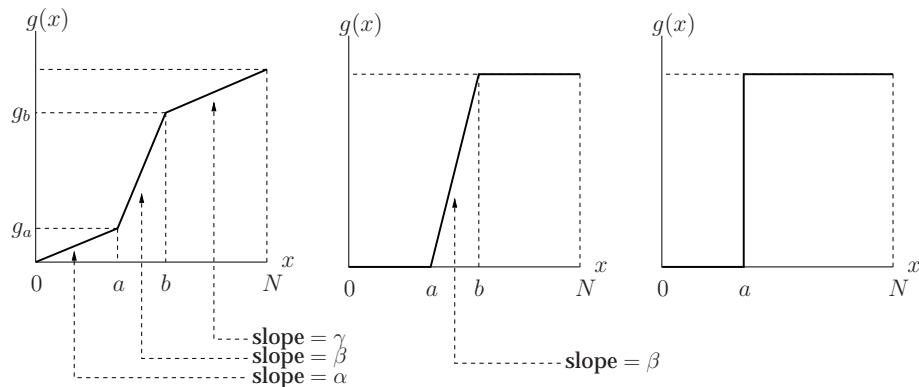


Figure 4.3 General contrast stretching (left), windowing (middle), and thresholding (right). See text for details.

contrast stretching (or contrast diminishing) taking place. If the slope is larger than one, contrast will be stretched (*i.e.*, enhanced), because the original grey values are mapped onto a larger range of grey values. A slope smaller than one means contrast is diminished, and a slope of exactly one indicates no contrast alteration.

Example

In figure 4.3, consider the left graph. If the grey values of an image were remapped according to this graph, low grey values ($0 \leq x < a$) will have diminished contrast after remapping, since the slope α is smaller than one. The same will happen at the high grey values ($b \leq x \leq N$). Contrast will be stretched in the middle region of grey values ($a \leq x < b$), since the slope is larger than one; the grey range $b - a$ will be mapped onto the larger range $g_b - g_a$.

Here is an example of this remapping (right) applied to an image (left):

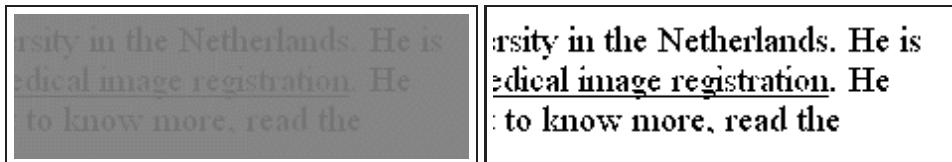


A special case arises if we choose $\alpha = \gamma = 0$. This is called *windowing* (or *clipping*), see figure 4.3. The windowing operation stretches the contrast to its maximum in a limited range (a “window”) of original grey values, $\{a, \dots, b\}$. All grey values outside this window are either mapped to zero or the maximum value as seen in the figure. Windowing allows us to focus all the available contrast onto an interesting window of grey values. It is especially useful when viewing some medical images, such as CT images. It can also help in cases of under- or overexposed photographs.

If we set $\alpha = \gamma = 0$ (windowing) **and** we set $a = b$, then we call the resulting remapping a *thresholding*, see figure 4.3. The name “thresholding” is chosen after the shape of the graph in the figure. This operation in effect “binarizes” (only two values present) the image: all grey values smaller than the threshold a are mapped to zero, and all grey values greater or equal to a are mapped to the maximum grey value. This operation is useful for processing images that are essentially binary (like pictures of written text), or in cases where we are only interested in the pixels with a grey value above or below a certain threshold.

Example

Below is a very low contrast image of a piece of text. On the right the same image after thresholding using a suitable threshold value a .



4.4 Other remapping operations

Standard functions

Many image processing and viewing programs offer the possibility to remap the grey values according to a standard function like log, exp, square root, hyperbolic sine ($= \frac{1}{2}(e^x - e^{-x})$), etc. When analyzing the effect of these remappings the rule of thumb is that contrast is stretched where the slope (*i.e.*, the first derivative of the function) is larger than one, and reduced if the slope is smaller than one.

Window slicing

A variant on thresholding is *window slicing* or *double thresholding*. The window slicing

operation sets all grey values in a window $\{a, \dots, b\}$ to the maximum grey value, and all values outside of this range to zero:

$$g(x) = \begin{cases} N & \text{for } a \leq x \leq b \\ 0 & \text{elsewhere} \end{cases}$$

Figure 4.4 shows an example of window slicing.

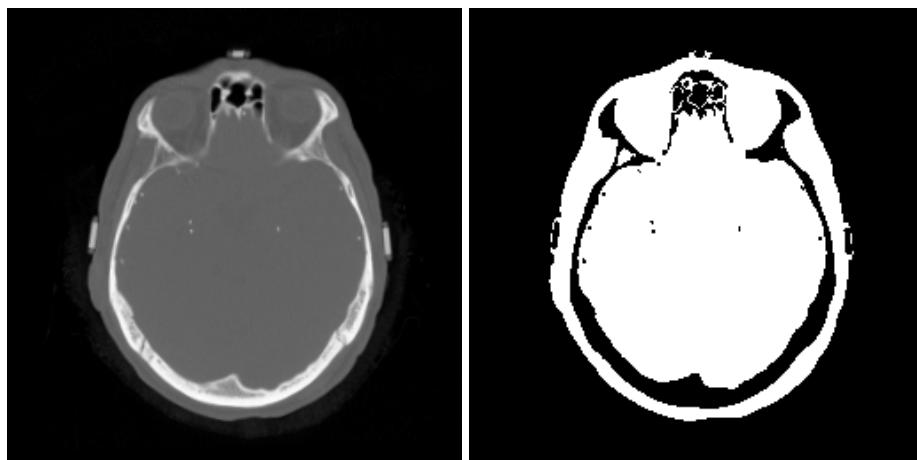


Figure 4.4 Example of window slicing. The original image (left) has grey values in the range $\{0, \dots, 255\}$. The right image shows the result after slicing the window $\{64, \dots, 123\}$, i.e., $a = 63$ and $b = 123$.

Window slicing is a binarizing operation: the resultant image has only two grey values (0 and N). Sometimes we wish to retain the original image as a “background” image, and highlight the grey values in a window. This can be achieved by *window slicing with background*:

$$g(x) = \begin{cases} N & \text{for } a \leq x \leq b \\ x & \text{elsewhere} \end{cases}$$

The window slicing graphs can be seen in figure 4.5.

Inverse

The *inverse* or *digital negative* of an image has already been shown in figure 4.1. The remapping function equals

$$g(x) = N - x,$$

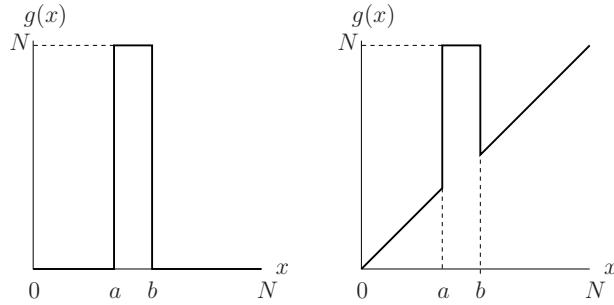


Figure 4.5 The window slicing operation (left), and the window slicing with background operation (right).

assuming the original grey value range to be $\{0, \dots, N\}$.

Range compression

Imaging devices or image processing algorithms often produce output that cannot be displayed directly as an image. For example: the GIF³ standard image format supports only 256 different grey (or color) values in an image. If we, e.g., have a medical image with grey values in the range $\{0, \dots, 4095\}$, and we wish to display it as a GIF image, then we first need to compress this range to $\{0, \dots, 255\}$. The simplest way to compress the range is to use linear compression, so

$$g(x) = cx,$$

with $c < 1$. If we need to compress a range of $\{0, \dots, M\}$ to $\{0, \dots, N\}$, then $c = N/M$.

Another type of compression is to use logarithmic compression, like

$$g(x) = c \log(1 + x).$$

This type of compression compresses differences in large grey values much stronger than differences in small values (where they could even expand), which is often a desired property with image processing algorithms.

On integer remapping

Up until now we have formulated our remapping functions g as continuously-valued functions. However, many images can only have integer grey values. The most practical solution to this problem is often to simply force the remapping function to take only integer values, e.g., by rounding the result down to the nearest integer. This rounding down can be denoted by the *entier* (or *floor*) operator: $\lfloor \cdot \rfloor$.

Note that rounding values to integers implies that it is possible that different input values x get mapped to the same output value $g(x)$!

³Graphics Interchange Format.

Example

Suppose we have this image:

0	1	2
0	1001	1002
0	5000	10000

and need to remap the values to $\{0, \dots, 255\}$ by logarithmic compression, using $g(x) = \lfloor c \log(1 + x) \rfloor$. The largest value occurring in the image is 10000, so the factor should be $c = \frac{255}{\log 10001} \approx 63.75$. The remapped image therefore will be

0	19	30
0	191	191
0	235	255

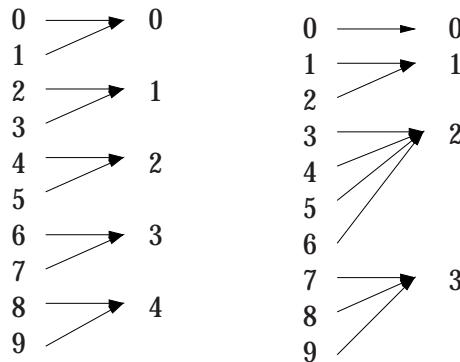
Note the contrast stretch at the low grey values, and the contrast reduction at the high values. Note also that both 1001 and 1002 get mapped to the same output value.

Binning

The effect that more than one input grey value can be mapped to a single output grey value is called *binning*⁴. If we use linear compression of an integer-valued image using $g(x) = \lfloor cx \rfloor$, the number of grey values that end up in each bin is constant and equal to $1/c$ if this is an integer. If we use logarithmic compression the number of grey values in each bin rises as the value rises.

Example

Compare the mapping of the range $\{0, \dots, 9\}$ according to $g_1(x) = \lfloor \frac{1}{2}x \rfloor$ and $g_2(x) = \lfloor^2 \log(x + 1) \rfloor$:



⁴i.e., multiple “items” get stored in the same grey value “bin”.

In the first case, each “bin” on the right side contains two original grey values. In the second case, the size of the bin doubles with each new bin. (Note that the “3” bin is truncated: 10 to 14 also map to the 3 bin.)

Sometimes we want to have a more precise control over how many grey values end up in each bin. This is illustrated in the section on histogram equalization further on.

Bit suppression and bit extraction

The grey values of digital images are of course represented as binary words in computers. It is often useful to look at the values of specific bits. For example: the least significant bit (LSB) often does not have much impact on the appearance of an image, so a good way of image range compression is often to simply ignore the LSB, and so reduce the number of actual grey values by a factor of two. Bit representation is also useful because it immediately shows the value of the term $k_i 2^i$ as shown below, which is of importance in many image analysis tasks.

We can write a positive decimal integer x as

$$x = \sum_{i=0}^M k_i 2^i \quad \text{with } M = \lceil 2\log(x+1) \rceil, \text{ and } k_i \in \{0, 1\},$$

where $\lceil \cdot \rceil$ is the *ceiling* function which rounds its argument up to the nearest integer. In this formula, the indices k_i are exactly the values of the bits if we represent x as an ordinary unsigned integer word. For example:

$$38 = (100110)_{\text{binary}} = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

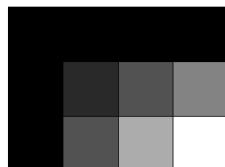
If we convert an image to a binary image that contains only the value of k_i we call this *bit extraction* of bit i . Setting bit i to zero is called *bit suppression*.

4.5 The image histogram

Grey value remapping operations like contrast stretching, windowing, thresholding, binning, etc. always have at least one parameter. A useful tool for establishing a practical value for such parameters is the image *histogram*, which is a frequency plot of the grey values occurring in an image.

Example

The image $\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 \\ 0 & 2 & 4 & 6 \end{pmatrix}$ =



has the histogram:

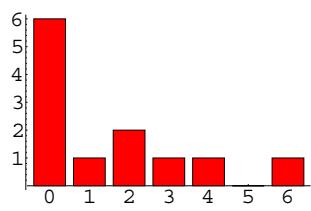


Figure 4.6 shows a larger image and its histogram.

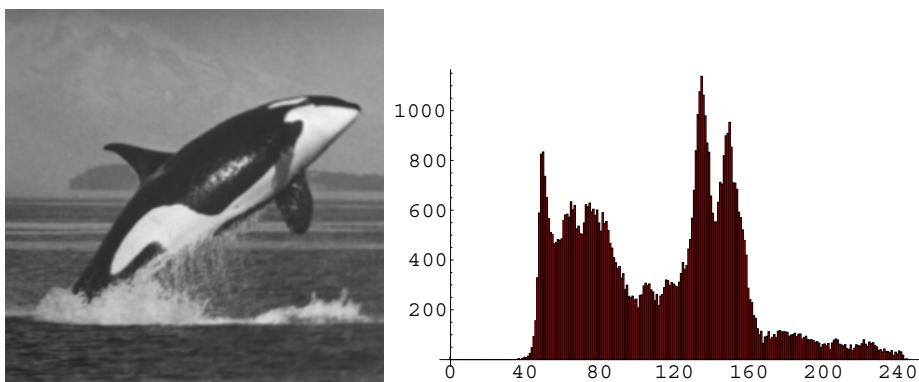
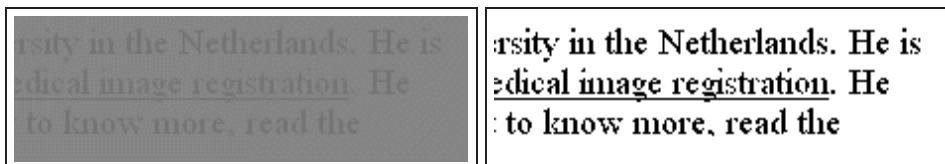
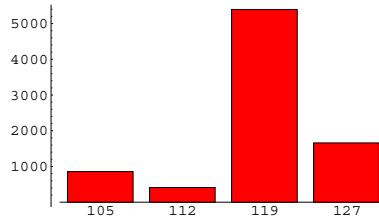


Figure 4.6 An image and its histogram.

In a previous example, we thresholded a low-contrast text image:



The correct threshold value can easily be found by looking at the image histogram:



Since there are two shades of grey in the text that are slightly darker than the two shades of grey in the background, we can assume the values 105 and 112 to correspond to the text. This is also obvious from the height of the bars: since there are far fewer text pixels than background pixels, the lowest bars should correspond to the text. So if we choose the threshold between 112 and 119 (i.e., everything below is set to zero (black) and everything above is set to the maximum grey value (white)), we get the correct thresholded result as seen above.

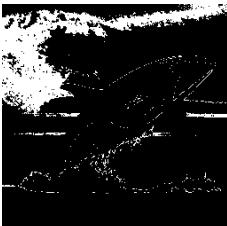
A lot can be learned from the image histogram, for example (refer to figure 4.6):

- The black part of the orca is made up of the darkest grey values occurring in the image, so we expect the peak of pixels at $x = 49$ to correspond to this image area. And indeed, if we slice the range $[41, 54]$ from the image, we get:



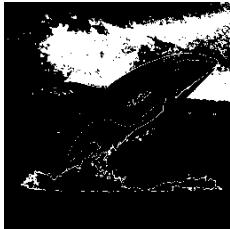
We have chosen the range window small so that the next peak is not included: this one belongs to the dark grey water at the bottom of the picture. (But already some overlap is visible.)

- When looking at the original image, we see that the most frequently occurring grey value is the grey value corresponding to the sky in the background. The highest peak in the histogram is located at $x = 134$, so we expect this peak to correspond to these background pixels. If we slice a small window around $x = 134$ (in this case $[130, 140]$), we get:



Perhaps surprisingly, we see that this is not all of the sky from the picture. Upon closer examination of the picture, we can see that the sky is slightly brighter on

the right side. So we expect the rest of the sky to correspond to the peak at $x = 149$ (the second-highest peak). Slicing the range $[140, 155]$ indeed gives us:



- The right part of the histogram is relatively flat. Since there is no clear peak, the white part of the orca is obviously not as uniformly white as we might think. Indeed, practically all of the pixels in the wide range $[200 - 255]$ are located in the white orca part. Slicing this range gives us:



which does not even give us the entire white part. The rest has even lower grey values. Since these grey values are the same ones as occurring in the water foam, we are unable to slice a range that neatly results in an image of the orca white part.

4.6 Histogram equalization

For image enhancement purposes, it is often useful to apply a remapping $g(x)$ that makes the histogram “flat”, *i.e.*, remaps the grey values so that each value occurs an equal number of times in the image. This is called *histogram equalization*. The idea behind it is that the available image contrast is used optimally. For instance, take a look at figure 4.6: from the histogram we can observe that the grey values $\{0, \dots, 40\}$ hardly occur and there are also relative few values in the range $\{160, \dots, 255\}$. For most of the image pixels, the contrast is limited to differences of values in the range $\{40, \dots, 160\}$, which constitutes less than half the possible contrast. If we could find a remapping that made each histogram value equal, the resulting image would make much better use of the available contrast. Histogram equalization is a technique that can improve contrast for many different types of images, including improperly exposed photographs.

In general, finding a remapping that does *exact histogram equalization* (*i.e.*, each grey value occurs an equal number of times in the image) is not possible for digital images. However, the following equalization algorithm approximates the ideal situation by distributing the grey values as evenly as possible over the histogram.

The histogram equalization algorithm tries to assign the ideal number of pixels i to each grey value. Failing this, it does this (assuming the output grey values are $\{0, 1, 2, \dots\}$):

- at most $1.5i$ pixels get grey value 0
- at most $2.5i$ pixels get grey value 0 or 1
- at most $3.5i$ pixels get grey value 0, 1, or 2
- etc.

It achieves this by assigning higher grey values to groups of pixels that exceed i in number, and leaving a sufficient number of grey values unused. To compute the correct new grey values, the algorithm makes use of the *cumulative histogram*: if $h(x)$ is a histogram, the cumulative histogram $c(x)$ is defined by

$$c(x) = \int_{-\infty}^x h(y) dy.$$

For digital images with an integer grey value range this boils down to

$$c(x) = \sum_{y \leq x} h(y).$$

Algorithm: histogram equalization

Given an image with dimensions d_x and d_y , i.e., it has $d_x \times d_y$ pixels, with N possible grey values ordered as $\{0, \dots, N-1\}$, and a cumulative histogram $c(v)$:

1. Determine the ideal number of times each grey value should occur in the image:

$$i = \frac{d_x \times d_y}{N}.$$

2. Remap the grey values according to

$$g^*(v) = \lfloor \frac{c(v)}{i} + 0.5 \rfloor - 1$$

3. Remap the grey values according to

$$g(v) = \begin{cases} g^*(v) & \text{if } g^*(v) \geq 0 \\ 0 & \text{if } g^*(v) < 0 \end{cases}$$

Example

Suppose we have a 4×5 image with $N = 8$ grey levels v with the following histogram $h(v)$:

v	$h(v)$
0	1
1	4
2	8
3	4
4	2
5	1
6	0
7	0

We can then compute the cumulative histogram and apply the above equalization algorithm:

v	$h(v)$	$c(v)$	$g^*(v)$	$g(v)$
0	1	1	-1	0
1	4	5	1	1
2	8	13	4	4
3	4	17	6	6
4	2	19	7	7
5	1	20	7	7
6	0	20	7	7
7	0	20	7	7

Plots of the histogram before and after equalization can be seen in figure 4.7. From the histograms, we can observe that after equalization the full image contrast is now used (*i.e.*, the brightest values are now used), and empty bins are created before histogram entries that are larger than the ideal value i .

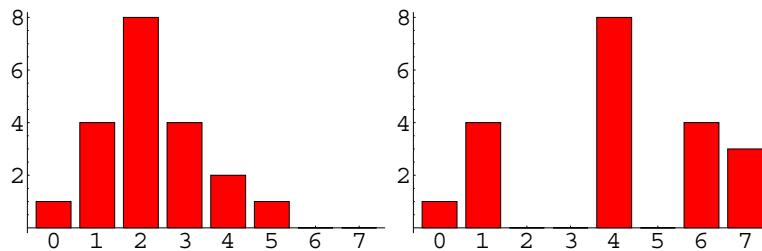


Figure 4.7 Example of histogram equalization: the original histogram can be seen on the left. On the right the histogram after equalization. See text for details.

Figure 4.8 shows an example of histogram equalization applied to a real image.

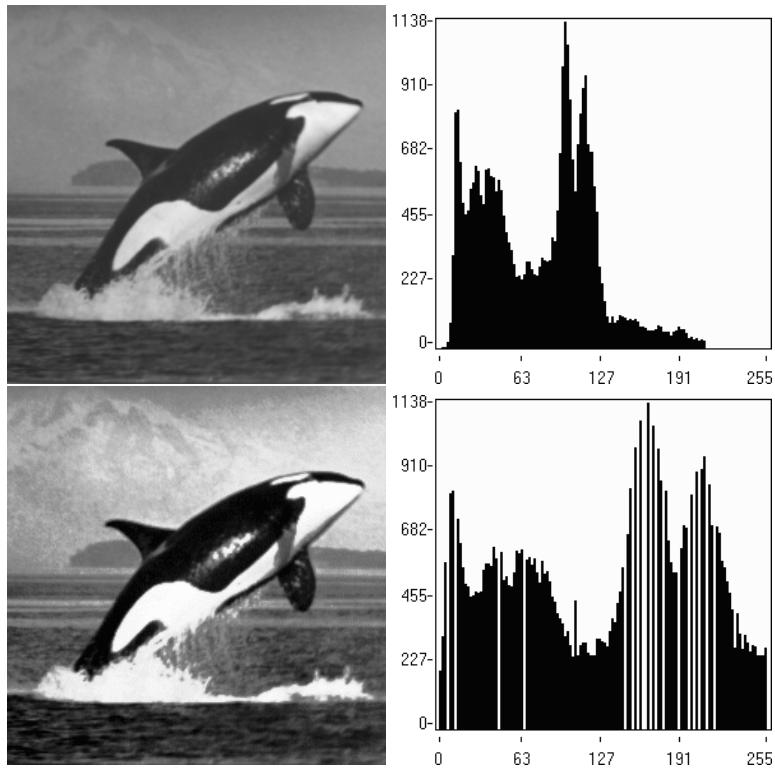


Figure 4.8 Example of histogram equalization. Top row: original image and histogram. Bottom row: after histogram equalization.

4.7 Multi-image operations

In the previous sections, we remapped the grey values of one input image to an output image. It is also frequently useful to use the grey values of two or more input images and map them to a single output image. For two input images, for example, the remapping function g is now a function of two variables, namely the two grey values at corresponding pixels in the two images. The remapping function g is often very simple, e.g., simple subtraction: $g(v_1, v_2) = v_1 - v_2$. Other frequently occurring functions are addition, multiplication, division, and logical boolean operations (AND, OR, XOR, etc.). The next paragraphs give some example applications of these operations.

4.7.1 Image subtraction

The figures 4.9 and 4.10 show applications of image subtraction, namely *motion detection* and *background elimination* (also called background subtraction). Detection of changes and background elimination are the most important applications of image subtraction.



Figure 4.9 Example of image subtraction for motion/change detection. The images in the top row look the same, but after subtraction (bottom row left) it is obvious some things are different. After histogram equalization (bottom right), more details become visible.

Two more examples: Automatic checking of industrially produced parts and automated video surveillance. Newly produced parts can be checked by placing them in front of a camera, acquire an image, and then subtract an image of what the part *should* look like. If the new part is correct, the difference image should contain only zeros⁵. In automated video surveillance, a video camera continuously monitors some area (say, a bank or store). From any image it acquires it subtracts an older image, say an image acquired a second before. If nothing happens in the area under surveillance, the difference image will contain only zeros. If the difference image contains a significant number of non-zeros however, something has happened, and the system can take action, such as sounding an alarm or start recording the images.

4.7.2 Image addition

Image addition is frequently useful for improving the quality of images in those cases where a good single image cannot be acquired. An example of this is electron microscopy: in some cases, single microscope image are so noisy or low-contrast that not much can be done with them. This can often be solved by acquiring two or more images, and adding them all together. To avoid very large grey values, we compute the

⁵Well, almost only zeros anyway. In practice there will always be small differences between the images.



Figure 4.10 Example of image subtraction for background elimination. The left and middle image are X-ray images of a patient respectively before and after injection of a contrast fluid. The contrast fluid is vaguely visible in the blood vessels in the middle image. After subtraction of the images (right image), the visibility is much better.

average rather than do straightforward addition:

$$a_n(x, y) = \frac{1}{n} \sum_{i=1}^n f_i(x, y), \quad (4.1)$$

where a_n is the average image, and f_1, \dots, f_n are the acquired n images.

The easiest implementation of averaging n images would be to acquire the n images, add all of the grey values at corresponding pixels together, divide by n , and store the results in an output average image. However, this implementation requires that all images f_1, \dots, f_n be stored in computer memory. This is frequently undesired or even impossible. To overcome this, we can implement averaging using only one image (and an accumulator) in memory:

$$a_n(x, y) = \frac{n-1}{n} a_{n-1}(x, y) + \frac{1}{n} f_n(x, y), \quad (4.2)$$

with $a_{-1} = 0$. Using this last equation, we only need to have the last average a_{n-1} in memory. That the equations 4.1 and 4.2 are identical can easily be ascertained by expanding equation 4.1 for some values of n (we leave out the coordinate pair (x, y) for easier reading):

$$\begin{aligned} a_1 &= \frac{1}{1} f_1 \\ a_2 &= \frac{1}{2} f_1 + \frac{1}{2} f_2 = \qquad \qquad \qquad \frac{1}{2} a_1 + \frac{1}{2} f_2 \\ a_3 &= \frac{1}{3} f_1 + \frac{1}{3} f_2 + \frac{1}{3} f_3 = \qquad \qquad \qquad \frac{2}{3} a_2 + \frac{1}{3} f_3 \\ a_4 &= \frac{1}{4} f_1 + \frac{1}{4} f_2 + \frac{1}{4} f_3 + \frac{1}{4} f_4 = \qquad \frac{3}{4} a_3 + \frac{1}{4} f_4 \\ &\vdots \\ a_n &= \dots = \qquad \qquad \qquad \frac{n-1}{n} a_{n-1} + \frac{1}{n} f_n \end{aligned}$$

If the images f_i are very noisy, averaging images may not give us the desired result. For example: if we take $n = 7$ and acquire the following pixel values at a certain location: $\{1, 1, 1, 1, 1, 8, 1\}$, then it is almost certain that the desired value is 1, and that the occurring 8 in the list is the result of noise in the image. If we compute the average in this case, the result would be $\frac{1+1+1+1+1+8+1}{7} = 2$. We can get rid of these “outlier” noisy pixel values by computing the modal value (modus) or –as is more common– median value instead of the average. The modal value is defined as the value that occurs most frequently in the list of values⁶. The median value is defined as the middle value of the list after the values have been sorted in ascending order⁷. A drawback of computing the median or modus is that we need to retain all of the input images in memory.

Example

I.

Suppose we acquire the following pixel values ($n = 7$): $\{1, 1, 1, 1, 1, 8, 1\}$. Then the average is 2, while both the modus and median are 1.

II.

Suppose we acquire ($n = 13$): $\{2, 3, 4, 4, 3, 2, 4, 1, 1, 2, 3, 1, 4\}$. The average is $34/13 \approx 2.62$. Sorting the list gives us: $\{1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4\}$, so the modus is 4 (highest frequency; 4 times), and the median (at the middle position in the list) is 3.

III.

Suppose we acquire ($n = 12$): $\{2, 3, 4, 4, 3, 2, 4, 1, 1, 2, 3, 1\}$. The average is $30/12 = 2.5$. Sorting the list gives us: $\{1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4\}$. Since all values occur at the same “highest” frequency, the modus equals the average of these values, 2.5. The middle two values are 2 and 3, so the median also equals 2.5.

Figure 4.11 shows the effect of averaging and taking the median value on a real image.

4.7.3 Boolean operations

x_1	x_2	$\text{OR}(x_1, x_2)$	$\text{AND}(x_1, x_2)$	$\text{XOR}(x_1, x_2)$	$\text{NOT}(x_1)$
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

⁶If there is more than one value occurring at the highest frequency, the modus is defined as the average of these values.

⁷If the list has an even number of entries, the median is defined as the average of the two middle values.

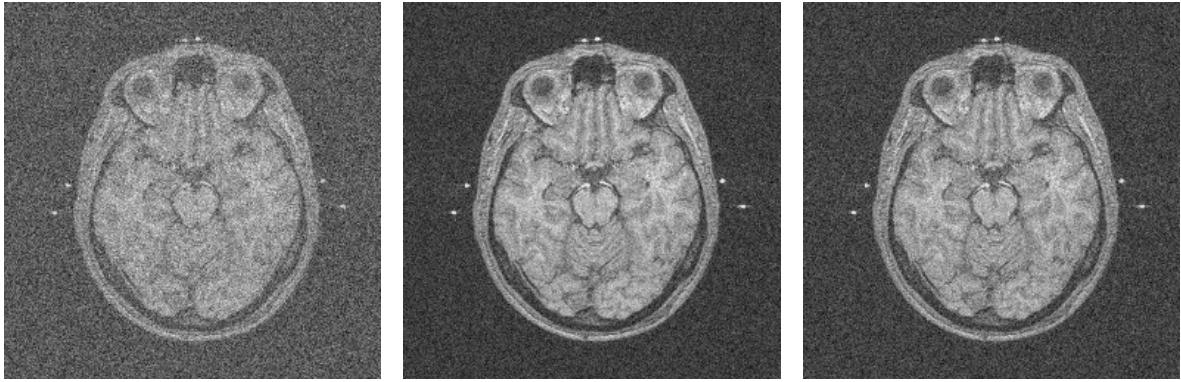


Figure 4.11 Effect of averaging and computing the median of noisy images. The left image shows an example of a noisy image. The middle image shows the effect of averaging five noisy images of the same scene. The right image shows the median values of the same five noisy images.

If we wish to combine images that contain only binary information (*i.e.*, only ones and zeros), it is natural to use binary boolean operations. Besides being very intuitive operations, an additional advantage is that boolean operations can usually be carried out relatively fast on computers. For an example, take a look at image 4.12. The first two images depict the “field of view” of two surveillance cameras (*i.e.*, how much they can “see” of a certain scene). These two images f_1 and f_2 are binary images, with pixel value one if the pixel is inside the field of view, and zero if it is outside of it. If we wish to determine whether the cameras are well-placed, *i.e.*, if they don’t leave any part of the scene invisible, don’t overlap too much, *etc.*, we can use boolean operations to answer all our questions:

$\text{OR}(f_1, f_2)$	shows the area covered by the two cameras
$\text{AND}(f_1, f_2)$	shows the area covered by both cameras simultaneously
$\text{XOR}(f_1, f_2)$	shows the area covered by only one camera
$\text{NOT}(\text{OR}(f_1, f_2))$	shows the invisible area

4.7.4 Image compositions

Image composition is the creation of a new image using elements from two input images directly. It is usually done using addition and multiplication of the input images and binary mask images, where a mask image is a binary image that defines the elements of interest in the input images. Figure 4.13 shows an example of image composition in this way. A problem with image composition is that correct mask images need to be created for a seamless composition. For many applications –such as the use of compositions in art and advertising– the mask images are drawn by hand with computer aid. In

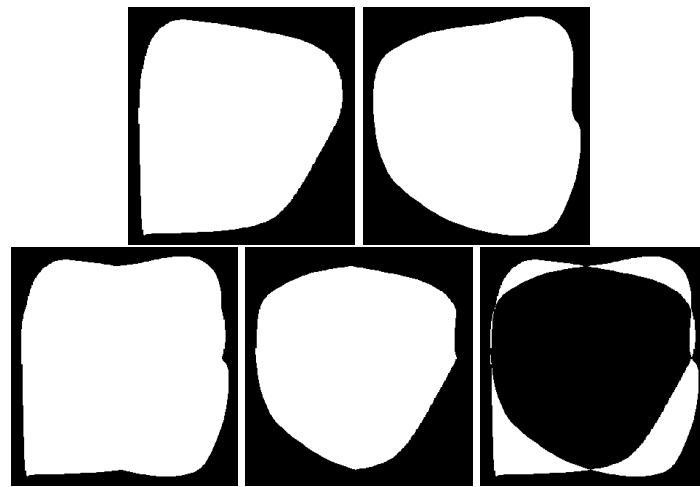


Figure 4.12 Example of boolean operations. Top row: two input images. Bottom row: boolean OR, AND, and XOR operation. See text for more details.

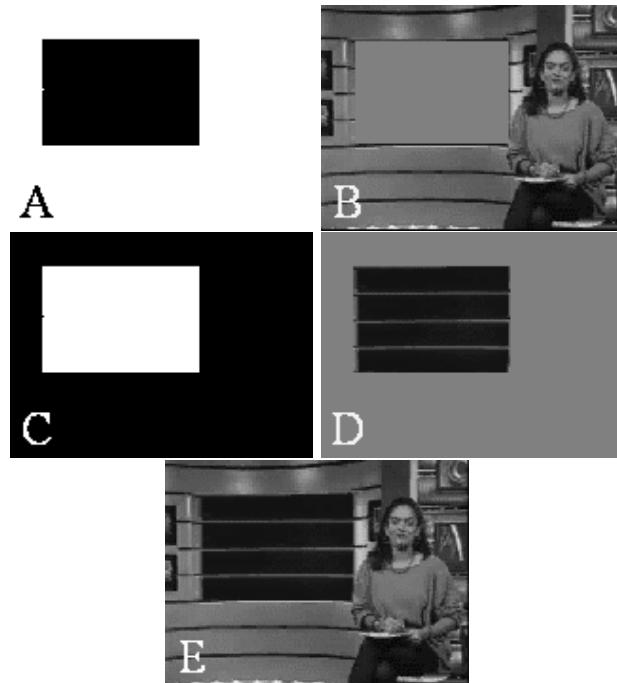


Figure 4.13 Example of image composition. The image E was composed from the input images B and D and the mask images A and C by the formula $E = AB + CD$. The mask images A and C are each others inverse, so that each pixel value in the composite image comes from either one of the input images; the values are never “mixed”, so no weird additions can occur.

the application shown in the figure, *i.e.*, a news reader, the composition must of course be done in real time. The technique used in this case is called “blue screen film” or “Chromakey”: the board behind the news reader where the second input image should be shown is colored a bright blue. An image composition computer fed with a camera image is able to recognize this blue and create a mask image from it. Using the mask image, the camera image, and an image to show at the location of the board, the computer can accurately render composite images in real time⁸. Blue screen film is also used frequently in movies: the actors perform in front of a blue screen, and the background is mixed in later.

In the example of figure 4.13 the mask images were each others inverse, so that a neat composition of the two input images is achieved. Sometimes it is desired that smooth transitions occur in composite images. In those cases, we can use smoothly varying images instead of binary mask images. An example is shown in figure 4.14.

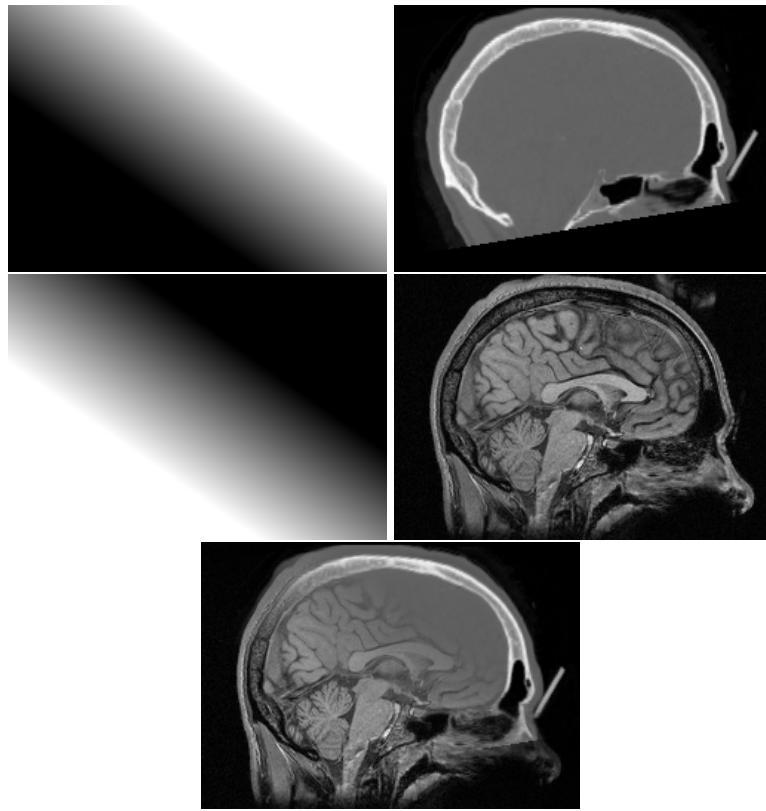


Figure 4.14 Example of smooth image composition. See previous image: the binary mask images have been replaced by smoothly varying weighting images.

⁸Now you know why news readers never wear blue...

4.8 Range considerations

The grey values of digital images are most commonly integers within a limited range, very often $\{0, \dots, 255\}$. This has many advantages, such as small filesizes (compared to, e.g., using floating point numbers that take four to eight times as much space in computer memory or on disk) and a high speed of computation (by using integer math instead of floating point operations, and using less memory compared to floating point images). The disadvantage is that we must constantly pay attention to whether mathematical operations are allowed when doing image arithmetic. For example: suppose we have two images coded using C-type unsigned chars, i.e., one byte per pixel, so with grey value range $\{0, \dots, 255\}$. If we use only unsigned char operations when doing image arithmetic with these images, even simple operations like image subtraction and addition will probably go wrong: in the former case negative values (underflows) may occur, in the latter case values larger than 255 (overflows) are likely to occur.

When underflows or overflows occur, computers will generate an error message⁹. However, truncation (roundoff) errors are never detected automatically. For example, implementing the remapping $g(x) = x/2$ should not produce any computer errors using any data type. Of course, when you use integer math, non-integer results will be rounded to the nearest integer, so some truncation errors will occur. In this case, the error should not be more than one¹⁰. However, should you implement the remapping like this: $g(x) = (\frac{x}{200}) \cdot 100$, then the truncation error (after computing $\frac{x}{200}$ using integer math) will be much larger.

In the above examples, the errors that may occur are obvious. Real life is often not that friendly. Truncation and range errors may be buried deep down in code, and even seasoned programmers occasionally cannot avoid a pitfall here. If it is not especially important that your application is optimized for speed, use these tips:

- Use integer math with care. Always use data types that can hold the full range of expected results.
- Check intermediate results of computations for truncation errors. Arrange your formulas so that truncation errors are minimal.
- Never count on the computer or compiler to exhibit some kind of special behavior in case of range or other errors. Avoid using “not_a_number” and other values that some languages support to identify infinite numbers.
- If you have to fit your results into, say, the integer range $\{0, \dots, 255\}$, then do this fitting *after* all other calculations.

⁹If you’re lucky, that is. Under some circumstances underflows or overflows are considered “features” instead of errors, and the computer goes on producing nonsense. Also, sometimes error trapping is removed or ignored to speed up the computation.

¹⁰Or even 0.5, depending on how intelligently the computer does his rounding off.

4.9 The two-dimensional histogram

In section 4.5 we pointed out the use of the image histogram for image analysis and determination of parameters for image processing tasks. If we have two different images of the same scene available, we can extract a lot of information about the images' coherence from the *two-dimensional (2D) histogram*. The 2D histogram is defined as the frequency plot of the pairs of grey values (a, b) occurring in the images A and B . Since this plot has two variables, an image processing scientist will invariably show the 2D histogram as an image, with frequency translated to pixel values.

Example

Consider these images:

$$A = \begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 1 \\ \hline 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 2 & 2 \\ \hline 0 & 0 & 2 & 2 \\ \hline \end{array} \quad B = \begin{array}{|c|c|c|c|} \hline 2 & 2 & 3 & 3 \\ \hline 2 & 3 & 3 & 3 \\ \hline 1 & 3 & 1 & 1 \\ \hline 2 & 1 & 2 & 0 \\ \hline \end{array}$$

the 2D histogram then is:

(a, b)	frequency
$(0, 0)$	0
$(0, 1)$	2
$(0, 2)$	4
$(0, 3)$	1
$(1, 0)$	0
$(1, 1)$	0
$(1, 2)$	0
$(1, 3)$	5
$(2, 0)$	1
$(2, 1)$	2
$(2, 2)$	1
$(2, 3)$	0

Which is more commonly presented in an image form (with image coordinates (a, b)):

$$\begin{array}{|c|c|c|} \hline 0 & 0 & 1 \\ \hline 2 & 0 & 2 \\ \hline 4 & 0 & 1 \\ \hline 1 & 5 & 0 \\ \hline \end{array}$$

The above example shows how the histogram can be used to quickly see relationships between grey values of the images A and B . For example, if we look at the second column, *i.e.*, if $a = 1$, then b equals 3. The inverse is not true, however, if $b = 3$ (look at the fourth row), then $a = 0$ (once) or $a = 1$ (5 times).

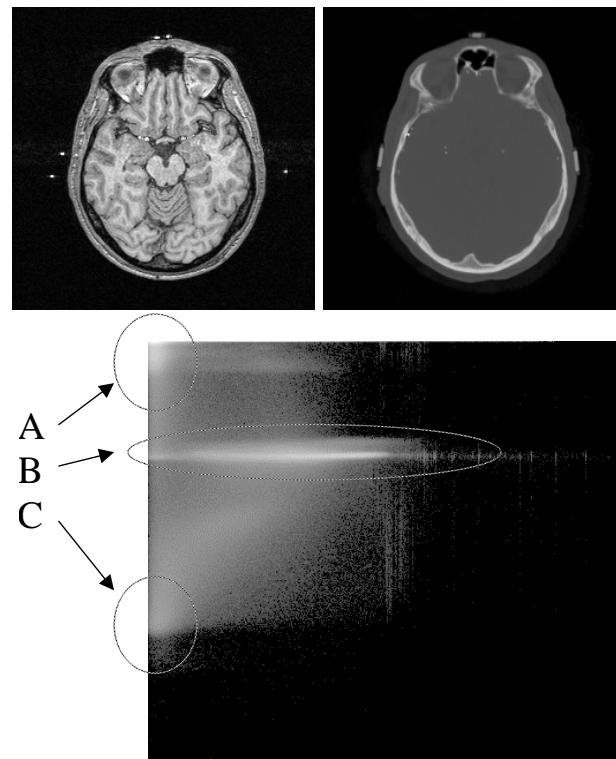


Figure 4.15 Example of a 2D histogram (bottom row) obtained from the images on the top row. The left image corresponds to the x -axis in the histogram, the right image to the y -axis. See text for details.

An example of a 2D histogram applied to real images can be seen in figure 4.15. The 2D histogram here can tell us much about the grey value relationship between the images. For starters, the 2D histogram appears to be structured, which indicates the images are possibly images of the same scene. If the 2D histogram looked like a random image, the two input images are probably unrelated. Some clusters seem to be present in the 2D histogram. For example look at the regions *A* and *C*: it seems that a low grey value in the left input image can correspond to both a low or a high grey value of the right image. Examining the images shows this to be indeed correct. Area *B* is also interesting: obviously some single grey value in the right image can correspond to many different values in the left image. This corresponds to the uniform grey area in the middle of the right image corresponding to many grey values occurring at the same location in the left image.

4.10 Noise

Noise can be described as the random change of the ideal pixel value in the acquisition phase. That is to say, at the moment when we can first perceive a measured pixel value, it has been distorted from its ideal value by a random component. Any imaging system experiences noise to some extent.

We can model the effect of noise by:

$$g(x, y) = f(x, y) + \eta(x, y), \quad (4.3)$$

where $g(x, y)$ is the image we observe, $f(x, y)$ represents the unknown ideal (undistorted) image, and $\eta(x, y)$ is the noise component. Because noise is inherently a random element in the imaging process, we are unable to reconstruct the ideal pixel values f from the observed values g . It is, however, usually possible to predict some global effects of noise, if we know something of the statistical behavior of the noise η , i.e., we have some knowledge on the statistical distribution of the stochastic variable η .

Example

In the section on image addition we observed that image averaging is often a good way to reduce noise effects. But how many images n should we average to obtain some desired noise reduction?

Suppose we have somehow ascertained that the noise component η from equation 4.3 behaves like a normally distributed variable with zero mean and variance σ^2 , i.e., $\eta \sim N(0, \sigma)$. We know that by averaging n identically distributed stochastic variables the variance of the average is reduced by a factor $\frac{1}{n}$. So the variance of the noise component after averaging n images also reduces by a factor $\frac{1}{n}$. We can now easily answer our question: if we want to reduce the noise variance by a factor of $\frac{1}{n}$, we need to average n images.

The example above assumes the noise to be normally distributed, i.e., the noise is *Gaussian*. Gaussian noise occurs frequently in image processing: for many imaging tasks, assuming the noise is additive Gaussian uncorrelated (“white”) noise¹¹ is a reasonably accurate description. Also, a very common strategy in testing image processing algorithms is to take an ideal image, test the algorithm on it, degrade the image with added Gaussian noise, test the algorithm again, etc. By adding noise ourselves we know exactly what f, g , and η are in equation 4.3, which is often essential for proper algorithm testing.

¹¹If noise is uncorrelated (“white”), this means the noise value at a certain pixel does not influence the noise value at the other pixels. In mathematical terms, the covariance of any two variables η is zero.

In practice, there are many cases where simply assuming the noise to be additive Gaussian noise does not accurately model the reality. A more sophisticated model assumes the noise to consist of two parts; a multiplicative part η_1 that depends on the pixel value –or some function h thereof– and an additive part η_2 :

$$\eta(x, y) = h(f(x, y)) \cdot \eta_1(x, y) + \eta_2(x, y). \quad (4.4)$$

It is often realistic to assume that both η_1 and η_2 have a normal distribution. A notable exception to this is so-called *speckle* noise, which is multiplicative noise with a Poisson distribution.

The noise models in the equations 4.3 and 4.4 depend only on the ideal pixel value $f(x, y)$. This is often not realistic enough to model the noise accurately. Sometimes it is necessary to use more complex models that incorporate the *neighborhood* of pixels around (x, y) instead of only (x, y) itself, and incorporate imaging characteristics such as the PSF.

Chapter 5

Simple spatial operations

In the previous chapter we discussed grey value remapping operations of the type $x \rightarrow g(x)$, where x is the grey value at a single pixel in the input image. Later we extended this to x being a vector containing the values of multiple images at a single pixel location. In *this* chapter, we will discuss operations that map grey values to new values considering a *neighborhood* of pixels, *i.e.*, we look at remapping of the type $x \rightarrow g(N)$, where N represents the pixel values in some neighborhood around the current pixel.

Because we use a neighborhood of pixels, a spatial operation is sometimes referred to as *pixel group processing*. By using a certain neighborhood of pixels, we are now able to use the *spatial* characteristics around some pixel (which is not possible using one-pixel grey value remappings). Another synonym is therefore *spatial filtering*. The operation (or the convolution kernel used) is sometimes called the *filter*.

5.1 Discrete convolution

In chapter 3 we introduced the concept of (spatial) convolution:

$$(g * f)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(a, b) f(x - a, y - b) da db, \quad (5.1)$$

where f is an image, and g some convolution kernel. A discrete version of convolution is defined by

$$(g * f)(x, y) = \sum_{(a,b) \in A} g(a, b) f(x - a, y - b), \quad (5.2)$$

where $A = \{(a, b) | g(a, b) \neq 0\}$, i.e., the set of all pairs (a, b) with a non-zero value of $g(a, b)$. Alternatively, we define

$$(g \star f)(x, y) = \sum_{(a,b) \in A} g(a, b) f(x + a, y + b). \quad (5.3)$$

We use this last definition when we need to apply discrete convolution to an image because it is more intuitive than the formal definition of equation 5.2 in this case.

Intermezzo

If we take f and g to be digital images on a unit grid, the one-dimensional convolution equation

$$(g * f)(x) = \int_{-\infty}^{\infty} g(a) f(x - a) da.$$

can be written as

$$(g * f)(x) = \sum_{a \in A} g(a) f(x - a),$$

where A is a set containing all a 's with a non-zero value $g(a)$, i.e., $A = \{a | g(a) \neq 0\}$. This equation is a discrete version of convolution. The formula for discrete convolution for two-dimensional images analogously becomes:

$$(g * f)(x, y) = \sum_{(a,b) \in A} g(a, b) f(x - a, y - b), \quad (5.4)$$

where $A = \{(a, b) | g(a, b) \neq 0\}$. Note that $(g * f)(x, y)$ is not defined for values of x and y for which values of $f(x - a, y - b)$ do not exist. Equation 5.2 has the drawback that the minus signs work counter-intuitive in practice (we will show this in the example below). Therefore we usually define discrete convolution by

$$(g \star f)(x, y) = \sum_{(a,b) \in A} g(a, b) f(x + a, y + b).$$

When there can be no confusion, we term both the “ $*$ ” and the “ \star ” operations convolutions. The \star operator is sometimes called the correlation operator. Note that the relation $f * g = g * f$ is true, but that this relation does *not* hold for the \star operator.

Having a formal definition, now what does a convolution *do*? From equation 5.3 we can see that the convolution result $(g * f)(x, y)$ is a weighted sum of image values f around (x, y) . Exactly how much each image value contributes to the convolution result is determined by the values in the convolution kernel g . We will work out an example:

Example

Suppose we have a convolution kernel g with

$$g(a, b) = \begin{cases} 1 & \text{if } (a, b) \in \{(-1, -1), (0, 0), (1, 0), (-1, 1)\} \\ 0 & \text{otherwise} \end{cases}$$

i.e., $g = \begin{array}{c} \begin{array}{ccc} -1 & 0 & 1 \end{array} \\ \begin{array}{|c|c|c|} \hline 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ \hline \end{array} \\ \begin{array}{c} -1 \\ 0 \\ 1 \end{array} \end{array}.$

Expanding equation 5.3 for this choice of g gives us:

$$(g * f)(x, y) = f(x - 1, y - 1) + f(x, y) + f(x + 1, y) + f(x - 1, y + 1).$$

So we see that the convolution result is exactly the sum of those image values f that occur in the neighborhood of (x, y) where the kernel g has value 1.

We see now that the effect of a convolution can easily be predicted if we represent the kernel in image form. It will now also be clear why we choose equation 5.3 for the working definition of discrete convolution, and not the formal equation 5.2. In the latter case, the intuitive relation between the convolution result and the pictorial representation of g is far less obvious.

Figure 5.1 shows the convolution process using a 3×3 kernel graphically.

Because discrete convolution allows us to make use of the spatial characteristics around a pixel, the number of possible applications is very large. In fact, virtually *all* image processing tasks that require spatial characteristics are implemented using convolution. In the next sections we will discuss a number of applications that require only simple kernels, generally not larger than 3×3 .

Unless specified otherwise, the kernels presented are centered around $(0, 0)$.

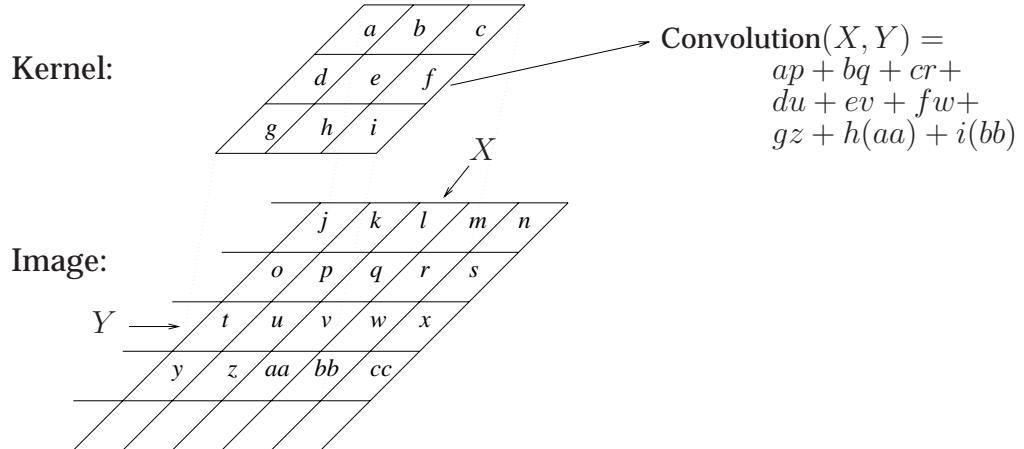


Figure 5.1 Graphical representation of discrete convolution using a 3×3 kernel.

5.1.1 Smoothing and low pass filtering

In chapter 3 we already saw that convolving with a rectangular kernel had a smoothing effect. The discrete version of such a kernel is, e.g.,

$$\begin{array}{|c|c|c|} \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \end{array} = \frac{1}{9} \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array},$$

for a 3×3 kernel. The second form is a short hand form for convenience. The effect of convolving an image with this kernel is that each pixel value is replaced by the average of itself and its eight direct neighbors. Convolving an image with this averaging kernel will have a smoothing effect on the image. It is used for noise reduction and low pass filtering. An example can be seen in figure 5.2. The general formula for an $N \times N$, with N odd, kernel that averages a pixel and all of its direct neighbors is

$$g(x, y) = \begin{cases} \frac{1}{N^2} & \text{if } x, y \in \left\{-\frac{N-1}{2}, \dots, \frac{N-1}{2}\right\} \\ 0 & \text{otherwise.} \end{cases} \quad (5.5)$$

If N is even, we cannot place $(0, 0)$ at the center of the kernel, so we cannot make the kernel symmetrical. Since symmetry is often a desired property, even-sized kernels are used far less than odd-sized kernels.

Many different kernels that compute local (*i.e.*, in a neighborhood) averages can be constructed, e.g.,

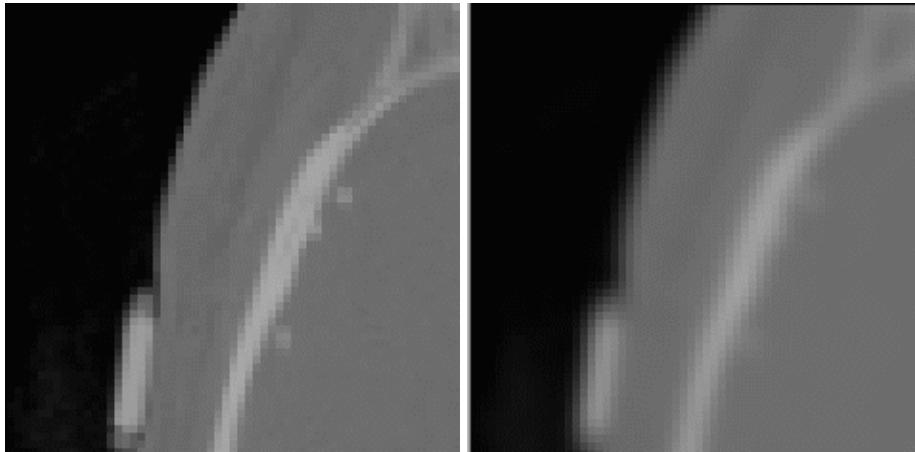


Figure 5.2 Example of convolving an image with a 3×3 averaging kernel.

$$\begin{array}{|c|c|c|} \hline
 \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \hline
 \frac{1}{8} & 0 & \frac{1}{8} \\ \hline
 \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \hline
 \end{array}
 \quad
 \begin{array}{|c|c|c|} \hline
 0 & \frac{1}{5} & 0 \\ \hline
 \frac{1}{5} & \frac{1}{5} & \frac{1}{5} \\ \hline
 0 & \frac{1}{5} & 0 \\ \hline
 \end{array}
 \quad
 \begin{array}{|c|c|c|} \hline
 0 & 0 & 0 \\ \hline
 \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline
 0 & 0 & 0 \\ \hline
 \end{array}.$$

With this last kernel it will be clear that its behavior is not isotropic, *i.e.*, it has a directional preference. In this case only pixel values in the x direction are averaged. But not even the kernels of equation 5.5 are isotropic; to achieve isotropy the kernel would have to be circular instead of square. We can only *approximate* a circular shape using square pixels, and the smaller the kernel, the worse the approximation will be.

Averaging kernels are also called *low pass filters*, since they remove high image frequencies (*i.e.*, the details), and let low frequencies “pass” through the filter. Image areas with constant pixel values (*i.e.*, only the zero frequency occurs) are not changed at all.

5.1.2 Sharpening and high pass filtering

An example of a detail enhancing kernel is

$$\begin{array}{|c|c|c|} \hline
 -1 & -1 & -1 \\ \hline
 -1 & 9 & -1 \\ \hline
 -1 & -1 & -1 \\ \hline
 \end{array}$$

How does this filter work? Notice that all the factors in the kernel add up to one, so convolution with this filter will not change the pixel value in a part of the image that has constant grey values. But if the pixel has a higher or lower grey value than its neighbors, this contrast will be enlarged by the filter, see this example:

Example

Example of applying the above detail enhancing kernel to an image:

5	5	5	5	5	5	5
5	5	5	5	5	5	5
5	5	5	5	0	0	0
5	5	5	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

→

5	5	10	15	20		
5	10	20	-20	-15		
20	25	-15	-5	0		
-15	-10	-5	0	0		
0	0	0	0	0		

Notice how the “constant” parts of the image (upper left and bottom right corner) are left unchanged, while the contrast in the parts where the 0’s and 5’s meet is greatly enhanced.

Figure 5.3 shows an example of this detail enhancing kernel applied to a real image.

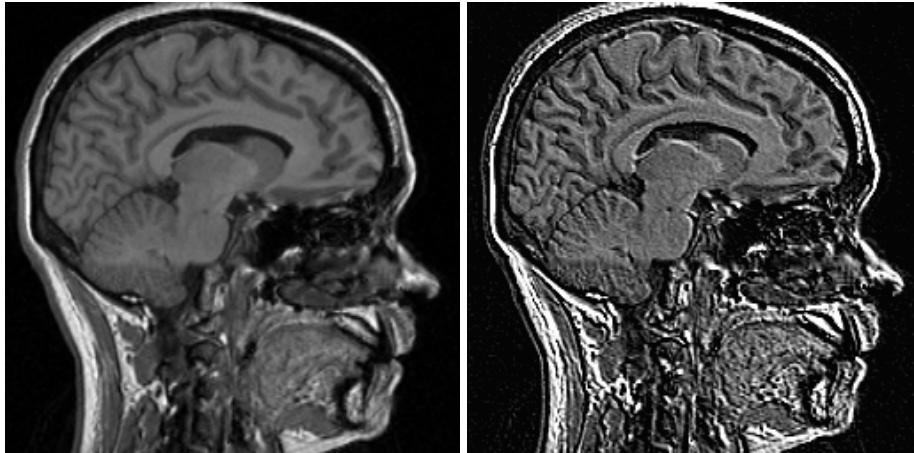


Figure 5.3 Example of applying a 3×3 detail enhancing kernel to a real image. See text for details.

It is common to call this detail enhancing kernel a high pass filter, although this is not strictly true. For one thing, constant areas are left unchanged by this filter, so this filter lets even the lowest frequency (the zero frequency) pass. More correctly, this filter is *high frequency enhancing*, while leaving the low frequencies relatively untouched.

We can construct a high pass filter by using the averaging low pass kernel from the previous section: if we *subtract* the low pass filtered image from the original, the result will contain only the high frequencies of the original image. We can use a single kernel to carry out this operation:

$$\frac{1}{9} \cdot \begin{array}{|c|c|c|} \hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}.$$

This kernel is very similar to our original detail enhancing filter, but the result is very different, as can be seen in figure 5.4. This filter acts truly as a high *pass* filter, whereas

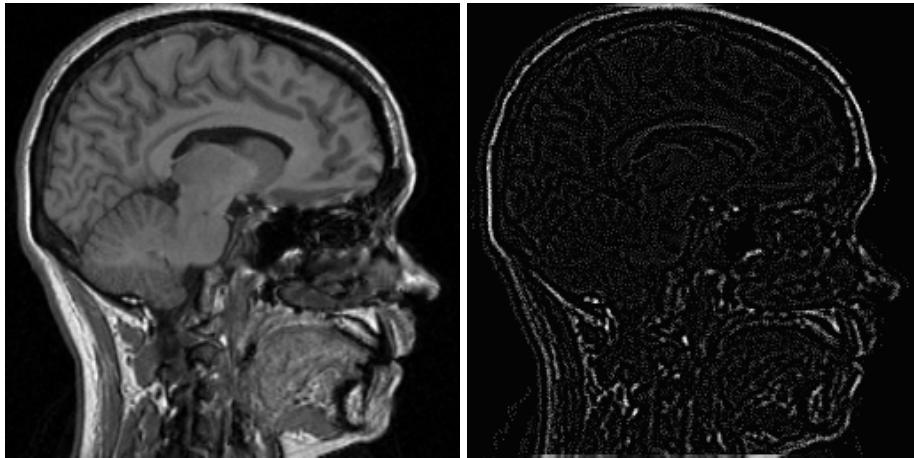


Figure 5.4 Example of high pass filtering an image. See text for details.

the original filter acted as a high frequency *enhancing* filter. This difference is explained by the different value of the center pixel in the kernel: in the case of the enhancing kernel there is a 9, so the kernel entries sum up to one. This means that constant areas remain unchanged. But the second kernel has an 8, which means that the entries sum up to 0, so constant areas are given pixel value 0.

Intermezzo

Enhancing high frequency details is something commonly done in printing processes, and is usually achieved by (1) adding an image proportional to a high pass filtered image to the original image, or (2) subtracting an image proportional to a low pass filtered image from the original. So

1. $f_{e_1}(x, y) = f(x, y) + \lambda_1 f_h(x, y)$
2. $f_{e_2}(x, y) = f(x, y) - \lambda_2 f_l(x, y),$

where f is the original image, λ_1 and λ_2 are positive constants, f_h is a high pass filtered image, f_l is a low pass filtered image, and f_{e_1} and f_{e_2} are the enhanced images. This technique is called *unsharp masking*. Figure 5.5 shows an example using technique (1) with $\lambda_1 = \frac{1}{4}$ and using the 3×3 high pass kernel defined in the text above:

$$\frac{1}{9} \cdot \begin{array}{|c|c|c|} \hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}.$$

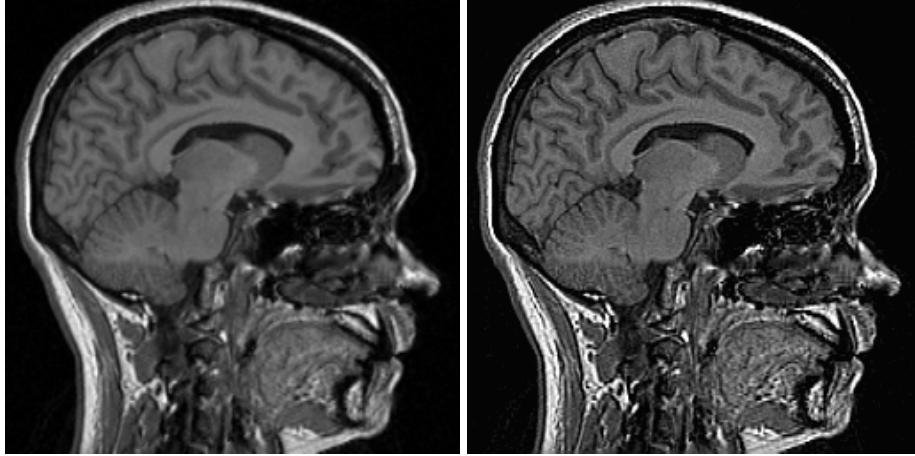


Figure 5.5 Example of unsharp masking. See text for details.

5.1.3 Derivatives

An important application of discrete convolution is the approximation of image *derivatives*. Derivatives play an important role in the mathematics of graphs: practically all of the “interesting” points of a graph (like extrema, inflection points, etc.) can be found by solving equations containing derivatives. This also holds true for images: practically all of the interesting points can be found using image derivatives.

A problem with computing derivatives of images is that images are not nice analytical functions for which we have a set of rules on how to compute derivatives. Digital images are nothing but a collection of discrete values on a discrete grid, so how do we compute derivatives? The best approach from the viewpoint of mathematical correctness uses *scale space* techniques and will be dealt with in a later chapter. In this chapter we use approximation techniques taken from numerical mathematics that can be implemented using discrete convolution. Before we do this, we introduce some elements from differential geometry that are frequently used in image processing.

5.1.3.1 Some differential geometry*

In this section we give some definitions and results from calculus and differential geometry. In all cases, f is a multiply differentiable and continuous function with $f : \mathbb{R}^n \rightarrow \mathbb{R}$

(i.e., an n dimensional continuous image), with variables x_1, x_2, \dots, x_n . a is an interior point of f . For partial derivatives, we use the shorthand $f_x = \frac{\partial f}{\partial x}$, $f_y = \frac{\partial f}{\partial y}$, $f_{xy} = \frac{\partial^2 f}{\partial x \partial y}$, etc.

The gradient and the Hamiltonian

The *nabla* or *gradient* operator ∇ is defined by

$$\nabla = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \dots, \frac{\partial}{\partial x_n} \right).$$

The *gradient* $\nabla f(a) = (\frac{\partial f}{\partial x_1}(a), \dots, \frac{\partial f}{\partial x_n}(a))$ is a vector that points in the direction of steepest ascent of f starting from a . The length of the gradient $\|\nabla f(a)\|$ (i.e., its norm; its magnitude) is a measure for the rate of ascent.

For $n = 2$, the *Hamiltonian* is the vector that is always perpendicular to the gradient: if $f = f(x, y)$, then the gradient $\nabla f = (f_x, f_y)$, and the Hamiltonian vector $= (f_y, -f_x)$.

With images, the integral curves¹ of the gradient and Hamiltonian are respectively called the image *flowlines* and *isophotes*. The image isophotes can also be defined in a much more intuitive way: they are the curves of constant grey value $f(x, y) = c$, comparable to iso-height lines on a map. See figure 5.6 for an example.

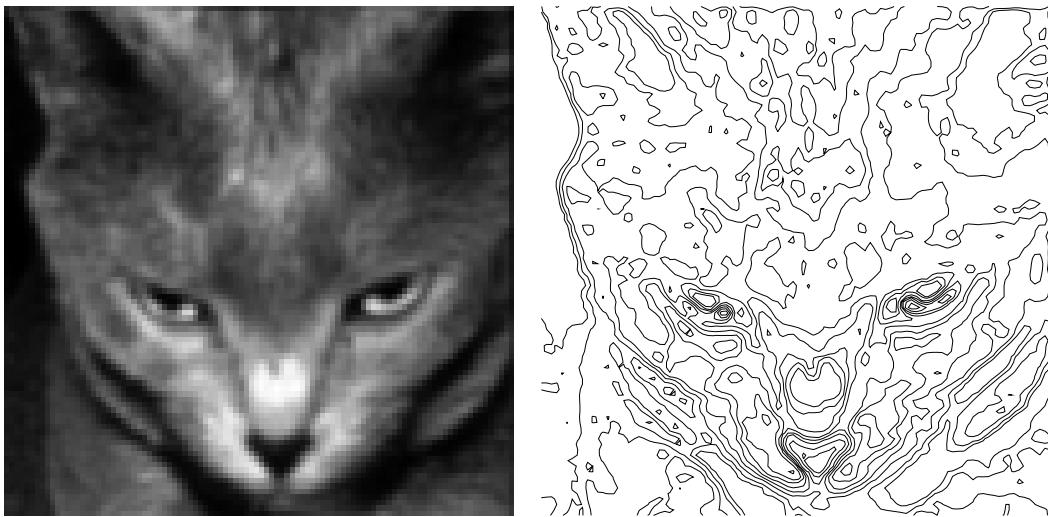


Figure 5.6 Example of an image and some of its isophotes; i.e., some curves of equal image intensity. Note how the curves get closer together in areas of high image contrast.

¹The integral curve of, e.g., the gradient vector field is a curve whose tangent equals the local gradient vector everywhere.

Stationary points

A point a is called *stationary* if $\nabla f(a) = 0$, i.e., if all of the partial derivatives $\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}$ are zero. An (interior) maximum, minimum, or saddle point is always a stationary point (but the reverse is not necessarily true). The second derivative in a can be used to establish the type of point. If $n = 1$, i.e., $f = f(x)$, a stationary point a is a maximum if $f''(a) < 0$ or a minimum if $f''(a) > 0$. If $n > 1$, we can use the Hessian H —which is the determinant of the matrix containing all second order partial derivatives—to establish the point type. For instance, if $n = 2$, i.e., $f = f(x, y)$:

$$H(a) = \begin{vmatrix} f_{xx}(a) & f_{xy}(a) \\ f_{xy}(a) & f_{yy}(a) \end{vmatrix} = f_{xx}(a)f_{yy}(a) - f_{xy}^2(a)$$

$$\left\{ \begin{array}{lll} \text{if } H(a) > 0 & \text{and } f_{xx}(a) > 0 & \text{then } f(a) \text{ is a local minimum} \\ \text{if } H(a) > 0 & \text{and } f_{xx}(a) < 0 & \text{then } f(a) \text{ is a local maximum} \\ \text{if } H(a) < 0 & & \text{then } f(a) \text{ is a saddle point.} \end{array} \right.$$

The Laplacian

The *Laplacian* Δf is defined by

$$\Delta f = \sum_{i=1}^n \frac{\partial^2 f}{\partial x_i^2}.$$

For example, if $n = 2$, $f = f(x, y)$, then $\Delta f = f_{xx} + f_{yy}$.

Example

Consider $f(x, y) = -2x^2 - y^2$, see figure 5.7. The partial derivatives are $f_x = -4x$ and $f_y = -2y$, so the gradient is $\nabla f = (-4x, -2y)$, and the Hamiltonian is $(-2y, 4x)$. The length of the gradient equals $\sqrt{16x^2 + 4y^2}$, so it is zero at $(0, 0)$, and grows as we move away from $(0, 0)$. The Laplacian Δf equals -6 .

The isophotes (the integral curves of the Hamiltonian) can be found by setting $f(x, y) = c$, which leads to $y = \pm\sqrt{-c - 2x^2}$.

There is only one stationary point: $(0, 0)$. Since $f_{xx} = -4$ and the Hessian $H = 8$, this is a maximum.

What is especially important for image processing in this section are the definitions of gradient, flowline, and isophote.

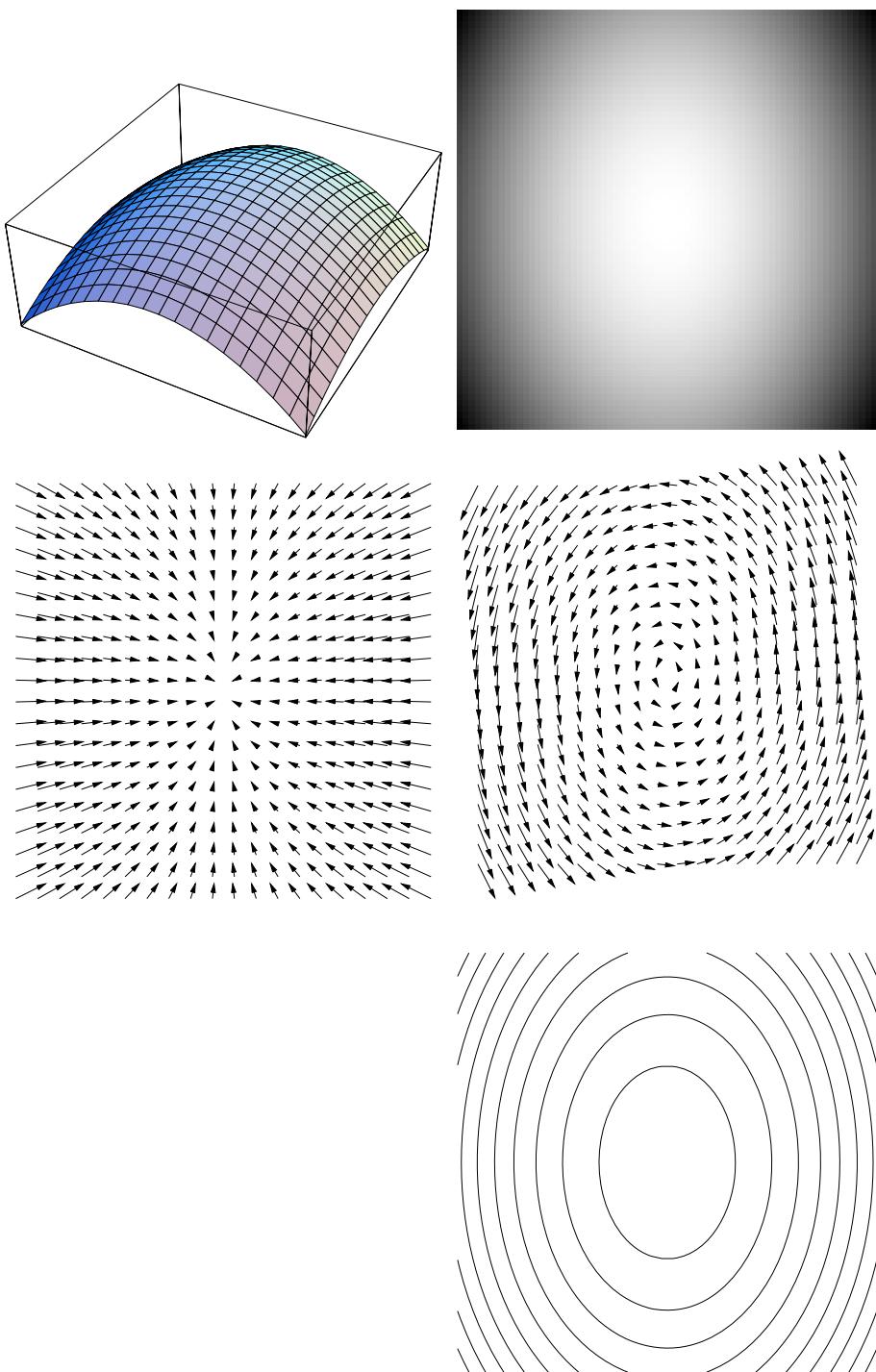


Figure 5.7 Example of basic differential geometry applied to $f(x, y) = -2x^2 - y^2$. Top left: original function. Top right: displayed as an image. Middle left: some vectors from the gradient vector field. Notice that they always point “uphill”. Middle right: some vectors from the Hamiltonian vector field. Bottom right: some isophotes, *i.e.*, curves of equal intensity. Note how the isophotes are the integral curves of the Hamiltonian. The picture of the flowlines is missing because they cannot be computed analytically in this case.

5.1.3.2 Approximating derivatives

Now that we have some use for derivatives in terms of gradients, isophotes, and flow-lines, we will introduce a way of approximating derivatives in images.

Suppose we have five equidistant samples of a one-dimensional function $f(x)$ and we wish to approximate the local first derivative $f'(x)$ using these five samples. Without loss of generality we assume the samples to be taken around zero with unit distance, i.e., we have sampled $f(-2), f(-1), f(0), f(1)$, and $f(2)$. Now we approximate the first derivative by f'_a , a linear weighted combination of our five samples, i.e., $f'_a = g_{-2}f(-2) + g_{-1}f(-1) + \dots + g_2f(2) = \sum_{i=-2}^2 g_i f(i)$. To find appropriate values for the weights g_i we demand that f'_a is exact if f equals one of the polynomials from the following set: $\{1, x, x^2, x^3, x^4\}$. This results in a system of five equations for the five unknowns g_i (each row corresponds to a polynomial, and the right hand side shows the correct value of the derivative at $x = 0$):

$$\left\{ \begin{array}{l} 1g_{-2} + 1g_{-1} + 1g_0 + 1g_1 + 1g_2 = 0 \\ -2g_{-2} + -1g_{-1} + 0g_0 + 1g_1 + 2g_2 = 1 \\ 4g_{-2} + 1g_{-1} + 0g_0 + 1g_1 + 4g_2 = 0 \\ -8g_{-2} + -1g_{-1} + 0g_0 + 1g_1 + 8g_2 = 0 \\ 16g_{-2} + 1g_{-1} + 0g_0 + 1g_1 + 16g_2 = 0 \end{array} \right.$$

or equivalently

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 \\ 4 & 1 & 0 & 1 & 4 \\ -8 & -1 & 0 & 1 & 8 \\ 16 & 1 & 0 & 1 & 16 \end{pmatrix} \begin{pmatrix} g_{-2} \\ g_{-1} \\ g_0 \\ g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Solving and substituting gives:

$$f'_a(0) = \frac{1}{12} (f(-2) - 8f(-1) + 0f(0) + 8f(1) - f(2)).$$

This result can easily be translated to images: the x -derivative of an image can be approximated using convolution with the 5×1 kernel

$$\frac{1}{12} \cdot [1 \ -8 \ 0 \ 8 \ -1],$$

and the y -derivative with the 1×5 kernel

$$\frac{1}{12} \cdot \begin{pmatrix} 1 \\ -8 \\ 0 \\ 8 \\ -1 \end{pmatrix}.$$

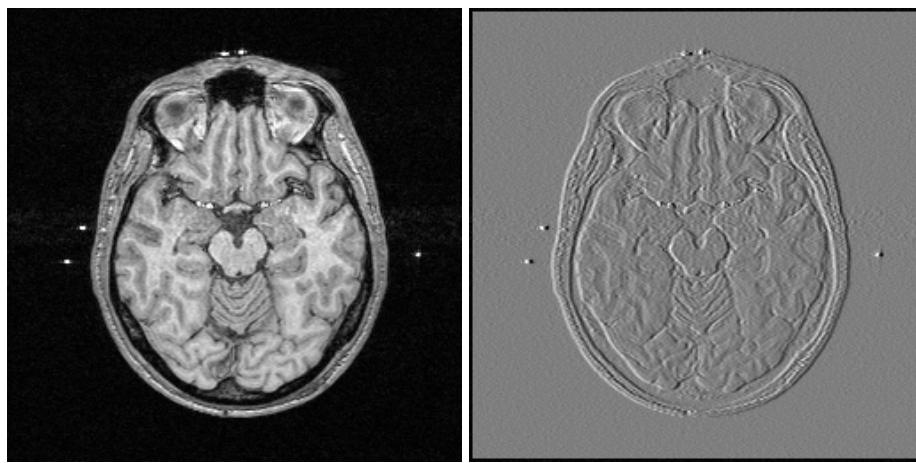


Figure 5.8 Example of the x -derivative f_x of an image f approximated by a 5×1 kernel as defined in the text.

Figure 5.8 shows an example of the x -derivative of an image approximated by the 5×1 kernel.

We can use the same technique to find kernels of different sizes for approximating derivatives, see the example below.

Example

If we want to approximate the first derivative using three pixels and demand it to be exact for the three polynomials $\{1, x, x^2\}$, we solve

$$\begin{cases} 1g_{-1} + 1g_0 + 1g_1 = 0 \\ -1g_{-1} + 0g_0 + 1g_1 = 1 \\ 1g_{-1} + 0g_0 + 1g_1 = 0 \end{cases},$$

or equivalently

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} g_{-1} \\ g_0 \\ g_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad (5.6)$$

which leads to the kernel $\frac{1}{2} \cdot \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$. The second derivative can be approximated by setting the right hand side in equation 5.6 to $(0, 0, 2)^T$, which leads to the kernel $\begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$.

Approximating cross-derivatives like f_{xy} takes a little more work, –since we cannot approximate these with flat one-dimensional kernels– but the principle remains the same, see the example below.

Example

Suppose we want to approximate f_{xy} using a 3×3 kernel g . As before, we center the kernel around $(0, 0)$, and assume a unit distance between pixels. If we label our kernel entries as

$$g = \begin{bmatrix} g_1 & g_2 & g_3 \\ g_4 & g_5 & g_6 \\ g_7 & g_8 & g_9 \end{bmatrix},$$

and we demand the result to be exact for the nine functions

$$\{1, x, y, xy, x^2, y^2, x^2y, xy^2, x^2y^2\},$$

then the system to solve becomes

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 \\ -1 & -1 & -1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ -1 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 1 \\ -1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \\ g_6 \\ g_7 \\ g_8 \\ g_9 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad (5.7)$$

which after some work leads to

$$g = \frac{1}{4} \cdot \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}.$$

Note that we did not include the functions x^3 or y^3 in the list. This is because their corresponding rows would be the same as the rows corresponding to respectively x and y , which would degenerate the equations.

Also note that this approach with a 3×3 kernel gives consistent results with the 3×1 kernels of the previous example. For example, if we want to approximate f_x , we replace the right hand side of equation 5.7 with $(0, 1, 0, 0, 0, 0, 0, 0, 0)^T$, which leads to

$$g = \frac{1}{2} \cdot \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline -1 & 0 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array},$$

which is equivalent to what we had already found before.

5.1.4 Edge detection

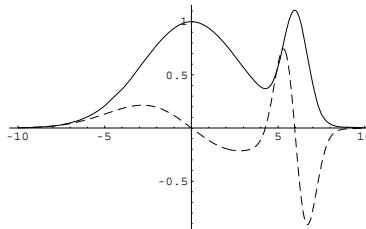
Edges are fundamental in many image processing tasks. Objects are defined by their edges, and hence edge *detection* plays an important role in finding and identifying objects.

Edges are easily located if they are “step” edges, as seen in figure 5.9, but step edges are rare occasions in many types of images. More often than not the edge of an object is not a step function, but a smoother transition of grey values across a number of pixels.

Edge pixels can be characterized as the pixels with a large local contrast, *i.e.*, regarding the image as a function, edge locations have a large slope, *i.e.*, a large (absolute) derivative. Edges are therefore often defined as the location where the derivative has a maximum or minimum, *i.e.*, where the second derivative is zero.

Example

The image below shows an example function (solid line) and its derivative (dashed line).



The sharp edge on the right side coincides with an extremum (minimum) of the derivative. On the left side of the graph the edge is much smoother, but the maximum of the derivative, *i.e.*, the locus of the steepest slope (at approximately -2.8), corresponds nicely to where most humans would place the edge point.

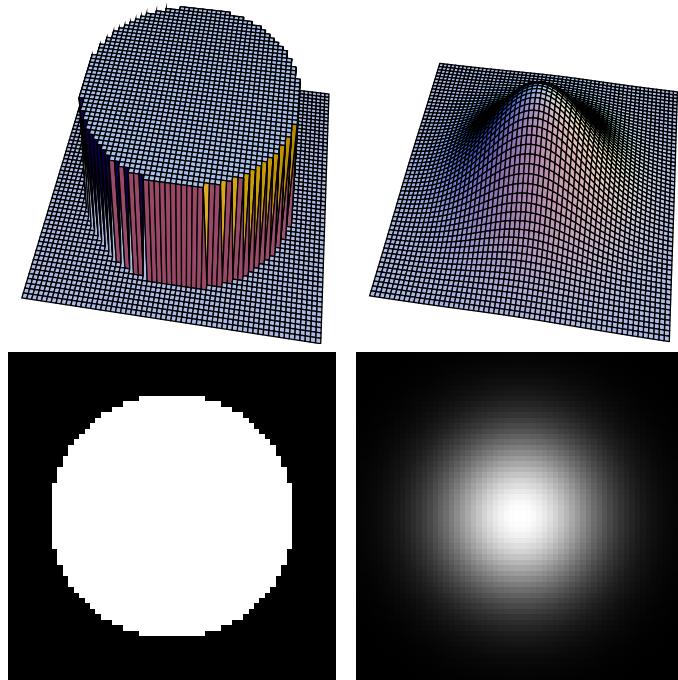


Figure 5.9 Example of a step object edge (left) and a smoother edge (right).

The derivative gives a good indication of the strength of the edge (the “edgeness”): sharp, steep edges have a large associated derivative value, and smooth edges have only a small derivative value.

For a two-dimensional image (say f), the slope of the grey value function in a pixel a depends on the direction in which you are looking: there is always a direction in which the slope is zero –the direction of the local isophote– and there is a direction in which the slope is maximum; the direction of the gradient $\nabla f(a) = (f_x(a), f_y(a))$. The maximal slope equals the norm of the gradient: $\|\nabla f(a)\| = \sqrt{f_x^2(a) + f_y^2(a)}$. An image showing the norm of the gradient² gives a good indication of edgeness, see figure 5.10. What we define as edge pixels can be extracted from this image by thresholding it.

We can compute the gradient image by using the techniques from the previous section, *i.e.*, f_x and f_y can be approximated using the kernels derived there, *e.g.*, the f_x image can be approximated by convolving the image with the kernel $\frac{1}{2} \cdot [-1 \ 0 \ 1]$.

The second derivative can also be used for finding edges: if we define an edge in a one-dimensional signal as the locus where the first derivative is maximum (or minimum),

²Usually –not entirely correctly– simply called a gradient image.

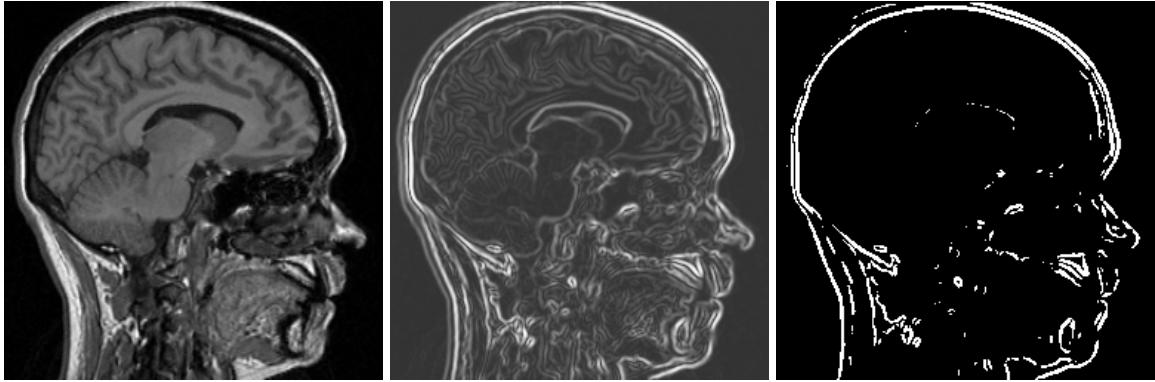


Figure 5.10 Example of edge detection. Left: original image. Middle: norm of the gradient. Right: after thresholding the gradient image.

then the second derivative will be zero at this locus. We can therefore locate edges by finding the zero-crossings of the second derivative. The two-dimensional equivalent of this for images is to find the zero-crossings of the Laplacian $\Delta f = f_{xx} + f_{yy}$. Figure 5.11 shows an example of this.

When computing edgeness images by approximation of the gradient norm $\|\nabla f\| = \sqrt{f_x^2 + f_y^2}$, we can often improve the results by using kernels slightly modified from the ones presented before for computing derivatives. For instance, consider these pairs of kernels:

	k_1	k_2
3 × 3 approximation:	$\frac{1}{2} \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{2} \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$
Prewitt:	$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
Sobel:	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$
Isotropic:	$\begin{bmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{bmatrix}$

In all three cases, we can compute an edgeness image g of an image f by computing $g = \sqrt{(k_1 * f)^2 + (k_2 * f)^2}$. Note that with every kernel the entries sum up to zero, so there will be no response in flat image areas. Ignoring all multiplicative factors, the

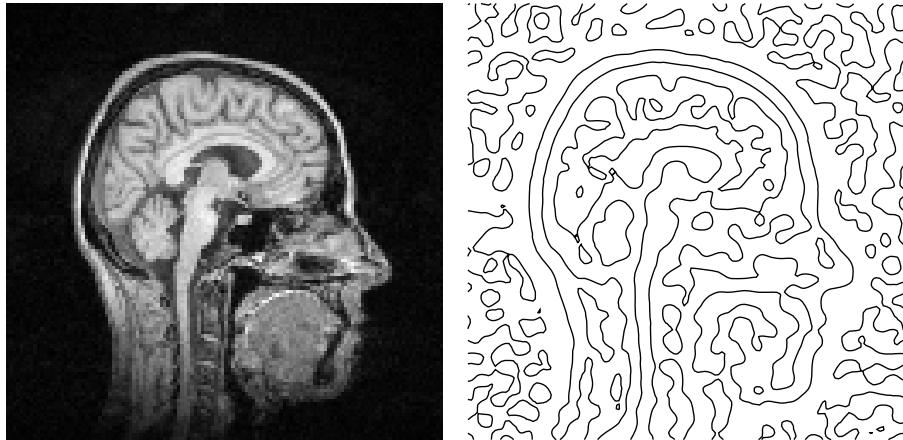


Figure 5.11 Example of edge detection. Left: original image. Right: zero-crossings of the Laplacian.

Prewitt kernels compute the average of the derivative across three image lines. The effect of this (relative to computing the derivatives by using the top two kernels) is that the result is less influenced by noise and very small structures. The Sobel kernels also compute the average derivative across three lines, but in a weighted way: the weight attached to the center line derivative is twice that of the weight attached to the other two lines. Figure 5.12 shows an example of an edgeness image computed using the Sobel kernels.

5.1.4.1 Edge orientation

The separate results $k_1 * f = g_1$ and $k_2 * f = g_2$ in the computation of edgeness using a pair of kernels form a vector (g_1, g_2) when combined. The magnitude $\sqrt{g_1^2 + g_2^2}$ of this vector is used for the edgeness image. The vector also has an orientation angle $\theta = \arctan(\frac{g_2}{g_1})$, as shown in figure 5.13. The orientation angle is a useful quantity in many image processing tasks where we are only interested in edges oriented in specific directions.

5.1.4.2 Compass operators

Edges can also be detected by using a *compass operator*. The result of a compass operator g on an image f is defined by

$$g = \max_{i \in \{0, \dots, 7\}} |g_i|,$$

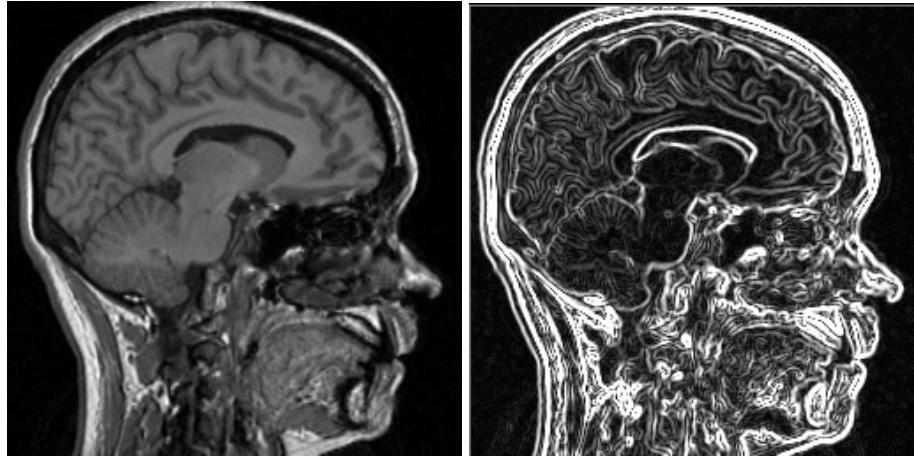


Figure 5.12 Example of edge detection. Left: original image. Right: Edgeness image computed using the Sobel kernels.

where $g_i = k_i \star f$, and k_i are rotated versions of a base kernel k . For example, if we choose a Prewitt kernel for a base kernel, the 8 rotated versions k_0, k_1, \dots, k_7 would be

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 0 \\ \hline 1 & 0 & -1 \\ \hline 0 & -1 & -1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline 0 & -1 & -1 \\ \hline 1 & 0 & -1 \\ \hline 1 & 1 & 0 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline -1 & -1 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline -1 & -1 & 0 \\ \hline -1 & 0 & 1 \\ \hline 0 & 1 & 1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & -1 & 0 \\ \hline \end{array}$$

which measure edge strengths in the eight compass directions ($\uparrow, \nwarrow, \leftarrow, \swarrow, \downarrow, \searrow, \rightarrow, \nearrow$) respectively.

A Prewitt or Sobel kernel is often used as the base kernel for a compass operator. Another one used frequently is the *Kirsch* operator, which uses for a base kernel:

$$\begin{array}{|c|c|c|} \hline 5 & 5 & 5 \\ \hline -3 & 0 & -3 \\ \hline -3 & -3 & -3 \\ \hline \end{array}.$$

5.1.4.3 Frei and Chen operator

Sometimes an edge detection method will erroneously indicate a high edgeness value at pixels where there is no edge to be found by human definitions. The Frei and Chen

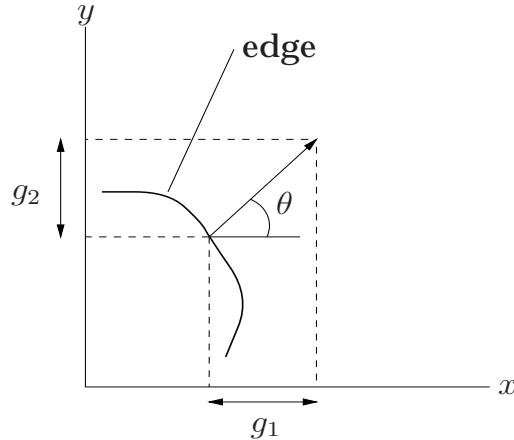


Figure 5.13 The orientation of a gradient vector as characterized by an angle θ .

operator can often remedy this. This operator uses the isotropic kernels to detect edges, but also requires that a number of other kernels have a *low* response. Nine kernels are used:

k_1	k_2	k_3	k_4
-1 0 1	-1 $-\sqrt{2}$ -1	0 -1 $\sqrt{2}$	$\sqrt{2}$ -1 0
$-\sqrt{2}$ 0 $\sqrt{2}$	0 0 0	1 0 -1	-1 0 1
-1 0 1	1 $\sqrt{2}$ 1	$-\sqrt{2}$ 1 0	0 1 $-\sqrt{2}$

k_5	k_6	k_7	k_8	k_9
0 1 0	-1 0 1	1 -2 1	-2 1 -2	1 1 1
-1 0 -1	0 0 0	-2 4 -2	1 4 1	1 1 1
0 1 0	1 0 -1	1 -2 1	-2 1 -2	1 1 1

The kernels k_1 and k_2 are the isotropic gradient kernels we saw before. The original Frei and Chen operator plotted the results of the convolution with each kernel k_i in a nine-dimensional space and then compared the results as projected in the “edge” subspace ($i \in \{1, 2, 3, 4\}$) with the results as projected in the non-edge subspace ($i \in \{5, 6, 7, 8, 9\}$). Only if the ratio was large enough was a pixel considered an edge pixel. The method is more commonly used in a simplified way: the kernels k_3 and k_4 are not used, because their response at edges is very small compared to the k_1 and k_2 responses.

5.1.4.4 Other edge detecting methods

From the large number of presented kernels it will be clear that the ideal edge-detecting kernel does not exist. In many cases, the Sobel operator is a good choice, since it performs reasonably well while requiring little computer operations and is easy to imple-

ment. If more subtle edges are to be found the Frei and Chen operator is usually the operator of choice. For specific applications, some of the kernels and operators presented in this chapter perform better than others, but *all* of them perform badly in the presence of noise³. For many applications it is necessary to use more sophisticated approaches to edge detection. One such approach is the *Marr-Hildreth* operator. This operator subtracts two images that are smoothed by Gaussian kernels of different width:

$$r = (f * g_{\sigma_1}) - (f * g_{\sigma_2}),$$

where r is the result of the operation, f the original image, and g_{σ_i} is a (2D) Gaussian function with parameter σ_i ($\sigma_2 > \sigma_1$). The effect of this operator is very similar to computing the Laplacian, but it is much less sensitive to noise. A drawback is that the two parameters σ_i will have to be carefully tuned to the task at hand. Figure 5.14 shows an example of applying the Marr-Hildreth operator. Other sophisticated approaches to

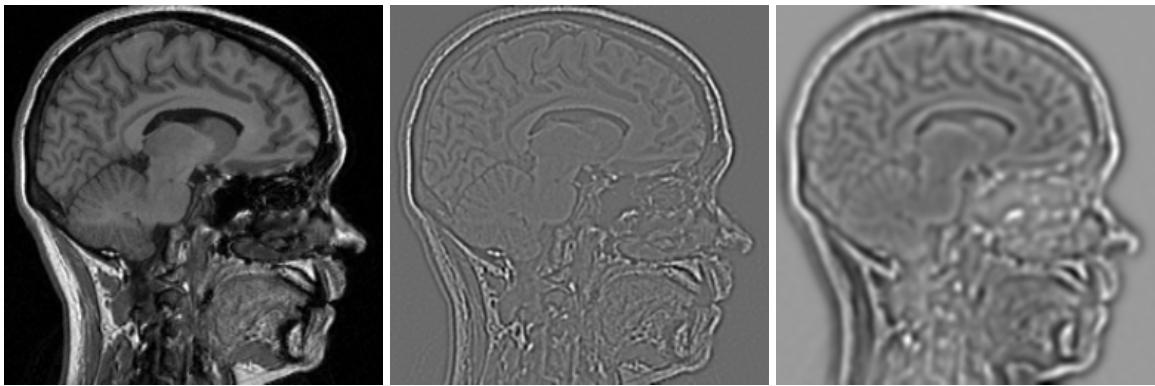


Figure 5.14 Example of the Marr-Hildreth operator compared to the Laplacian. Left: original image. Middle: Laplacian. Right: Marr-Hildreth operator.

edge detection will be covered in later chapters.

5.1.4.5 Line and isolated pixel detection

Lines can be detected using the compass operator:

k_1	k_2	k_3	k_4
-1 -1 -1 2 2 2 -1 -1 -1	-1 -1 2 -1 2 -1 2 -1 -1	-1 2 -1 -1 2 -1 -1 2 -1	2 -1 -1 -1 2 -1 -1 -1 2

³Even though the Laplacian is usually most sensitive and the Frei and Chen operator least sensitive to noise.

The kernels can be used like the edge detecting compass kernels:

$$g = \max_{i \in \{1, \dots, 4\}} |g_i|,$$

where $g_i = k_i * f$. Note that the entries of each kernel sum up to zero, so there is no response in flat image areas. The highest kernel response will occur if a one-pixel thick line is oriented along the row of twos in the kernels, but an edge (oriented along the twos) will also give a positive response. Figure 5.15 shows an example of applying this line operator.

Isolated points in images –i.e., pixels that have a grey value significantly different from their neighbors' grey values– can be detected by (thresholding of) the convolution of an image with the kernel

-1	-1	-1
-1	8	-1
-1	-1	-1

However, this kernel also responds in other image areas; it has already been presented as a high pass kernel. A better approach is to use the following kernel as the base kernel for a compass operator:

0	-1	0
0	1	0
0	0	0

and ensure all of the rotated kernel responses have large absolute values.

5.1.5 Approximating continuous convolution

The approximation of continuous convolution by a discrete convolution is a frequently occurring task in image processing. For example, it is used in simulating image acquisition, and approximating Gaussian convolution, which is useful for noise suppression and the basis of Gaussian scale space (see chapter 9). We will use the approximation of Gaussian convolution as a working example in this section.

Figure 5.16 shows a discretization of the Gaussian kernel $g(x) = \frac{1}{2\sqrt{\pi}}e^{-\frac{x^2}{4}}$ using nine pixels. These nine values $\{g(-4), g(-3), \dots, g(4)\}$ (approximated to three decimals) are



Figure 5.15 Example of a line detecting compass operator. Left: original image. Right: after applying the compass operator.

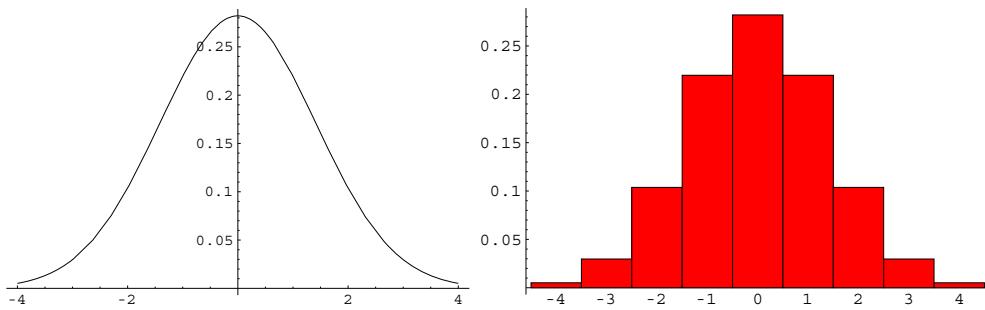


Figure 5.16 Example of a continuous Gaussian kernel $g(x) = \frac{1}{2\sqrt{\pi}}e^{-\frac{x^2}{4}}$ (left) and the same kernel discretized using 9 pixels (right).

0.005	0.030	0.104	0.220	0.282	0.220	0.104	0.030	0.005
-------	-------	-------	-------	-------	-------	-------	-------	-------

These values nicely capture the shape of the Gaussian, but we cannot use them directly as a discrete kernel, because we would then lose an important property of the Gaussian kernel, namely that the area under the curve equals one. This property ensures us that flat image areas are not changed by Gaussian convolution. For the discrete kernel to have the same property we must ensure its entries sum up to one, *i.e.*, normalize it. This is achieved by including a factor equal to the inverse of the sum of entries, *i.e.*,

$$\frac{1}{0.999} \cdot [0.005 \ 0.030 \ 0.104 \ 0.220 \ 0.282 \ 0.220 \ 0.104 \ 0.030 \ 0.005].$$

In this case, the factor is nearly one. Should we have used only the pixel values at $\{-2, -1, 0, 1, 2\}$, then the factor would have been 0.929.

Two types of error are involved in this kind of approximation. First a discretization error caused by the fact that we approximate (in this case) each unit length of the Gaussian curve by a single pixel value. This error can be reduced by denser sampling of the continuous pixels, *i.e.*, by using more pixels per unit length⁴. Second a truncation error because values of the kernel more than four units from the origin are not taken into account. This error can be reduced by extending the range in which the discrete kernel is sampled. In this case, the truncation is already very small: less than 0.5% of the area under the Gaussian curve is below the ‘tail’ ends farther than four pixels from the origin.

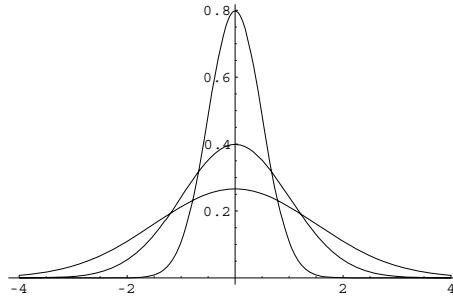


Figure 5.17 Three Gaussian kernels with $\sigma \in \{0.5, 1, 1.5\}$. The narrowest kernel has $\sigma = 0.5$.

Figure 5.17 shows three Gaussian curves with values of σ of 0.5, 1, and 1.5 respectively. If σ is small, the truncation error will be small even for small discrete kernels because the curve rapidly tends to zero as we move away from the origin. Using one sample per pixel, the discretization error will be large, because the shape of the kernel cannot be adequately captured using a few pixels around the origin. If σ is large, the discretization error will be relatively small because the shape of the kernel can be captured better. The truncation error can be kept small by choosing a large discrete kernel size. In practice, the kernel size is often chosen depending on the value of σ , *e.g.*, the kernel size is at least 3σ .

The approximation technique in the above is easily extended to two and more dimensions, as the example below shows

Example

Suppose we want to approximate convolution with the 2D Gaussian function

$$g(x, y) = \frac{1}{\sigma^2 2\pi} e^{-\frac{1}{2} \left(\frac{x^2 + y^2}{\sigma^2} \right)}$$

with $\sigma = 2$ using a 9×9 discrete kernel. Filling the kernel is done by computing the values of $g(x, y)$ for $(x, y) \in \{-4, -3, \dots, 4\} \times \{-4, -3, \dots, 4\}$, which leads to (using two digits of accuracy):

⁴With digital images using discrete convolution, however, there is not much sense in sampling denser than the pixel size, *i.e.*, to use more than one sample per pixel.

0.00073	0.0017	0.0033	0.0048	0.0054	0.0048	0.0033	0.0017	0.00073
0.0017	0.0042	0.0078	0.011	0.013	0.011	0.0078	0.0042	0.0017
0.0033	0.0078	0.015	0.021	0.024	0.021	0.015	0.0078	0.0033
0.0048	0.011	0.021	0.031	0.035	0.031	0.021	0.011	0.0048
0.0054	0.013	0.024	0.035	0.04	0.035	0.024	0.013	0.0054
0.0048	0.011	0.021	0.031	0.035	0.031	0.021	0.011	0.0048
0.0033	0.0078	0.015	0.021	0.024	0.021	0.015	0.0078	0.0033
0.0017	0.0042	0.0078	0.011	0.013	0.011	0.0078	0.0042	0.0017
0.00073	0.0017	0.0033	0.0048	0.0054	0.0048	0.0033	0.0017	0.00073

Summing the values found amounts to 0.95456, so the multiplication factor for this kernel should be $\frac{1}{0.95456}$.

The truncation error is still fairly large using a 9×9 kernel, since $\int_{-4}^4 \int_{-4}^4 g(x, y) dx dy \approx 0.911$, so about 9% of the area under the kernel is outside of the window.

Note that we can make use of the symmetry of the Gaussian in filling the discrete kernel; we only have to compute the values in one quarter of the kernel, the rest of the values can be filled in symmetrically.

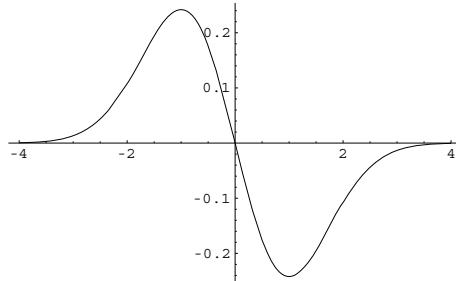
Most convolution with “acquisition type” kernels –*i.e.*, those that have an area of one under their curve and tend to zero away from the origin– can be approximated by discrete convolution in the same way as we did in the above for Gaussian convolution, *i.e.*, by sampling values in a finite discrete region, normalizing the values found, and placing the results in a discrete kernel. The discretization and truncation errors should be re-assessed for each new type of kernel.

Many “feature-extracting” kernels have the property that the area under the kernel is zero, *i.e.*, they don’t respond in flat image areas. If the kernel is odd⁵ very similar techniques as before can be used, see the example below. A difficulty only arises when a correct normalization factor must be found.

Example

Suppose we have the following feature-extracting kernel

$$f'(x) = -\frac{x}{\sigma^3 \sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x^2}{\sigma^2}\right)},$$



⁵*i.e.*, $g(-x) = -g(x)$; the curve is point-symmetric in the origin.

i.e., the derivative f' of the Gaussian

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x^2}{\sigma^2}\right)}.$$

Suppose we wish to approximate a continuous convolution with the kernel f' with $\sigma = 1$ by a discrete convolution using a 9×1 kernel. We can find the appropriate kernel entries $f_a(x)$ by evaluating f' at $x \in \{-4, -3, \dots, 4\}$ (using two digits of precision):

0.00054	0.013	0.11	0.24	0	-0.24	-0.11	-0.013	-0.00054
---------	-------	------	------	---	-------	-------	--------	----------

The kernel entries sum up to zero, which is a desired property. This means that the area under the curve of f' (which is zero) is preserved regardless of a normalization factor or the size of the discrete kernel. Because the “outer” kernel entries are very small the truncation error is very small.

Finding an appropriate normalization factor is not straightforward because the property of preservation of area under the kernel is already met. In these cases it is common to extend the property $\int f(x) dx = 1$ to the kernel $f'(x)$: by partial integration we find

$$\int f(x) dx = \int f(x) \cdot 1 dx = [f(x)x] - \int xf'(x) = \int -xf'(x) dx,$$

where the term $[f(x)x]$ is zero because $f(x)x$ is an odd function. In other words, we can find a good value for the normalization factor p by setting $\int -xf'(x) dx = 1$. The discrete equivalent of this in our example would be

$$p \left((4f_a(-4)) + (3f_a(-3)) + \dots + (-4f_a(4)) \right) = 1,$$

Which leads to $p = 1.00007$. Below are some values of p for kernels of different sizes.

size	p
3	2.06637
5	1.09186
7	1.00438
9	1.00007

5.1.6 More on kernels

Convolution has many properties similar to multiplication properties:

Commutative law: $f * g = g * f$

Associative law: $(f * g) * h = f * (g * h)$

Distributive law: $(g + h) * f = g * f + h * f.$

But this is where the similarity ends. For instance, $f * f$ can have negative values, and $f * 1$ does not generally equal f . Note that these laws do *not* apply to the $*$ operation, except for the distributive law.

The distributive law is useful in image applications where we need to compute two convolutions of one input image, and then add (or subtract) the results. According to the distributive law, we can get the same result by convolving once with the sum (or difference) of the original convolution kernels, which is usually a much cheaper operation.

Example

Given an image f , we can compute

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 5 & 0 & 0 \\ \hline 0 & 4 & 0 \\ \hline \end{array} * f + \begin{array}{|c|c|c|} \hline 2 & 3 & 4 \\ \hline 0 & 0 & 5 \\ \hline 0 & 0 & 6 \\ \hline \end{array} * f$$

in one go using

$$\begin{array}{|c|c|c|} \hline 3 & 5 & 7 \\ \hline 5 & 0 & 5 \\ \hline 0 & 4 & 6 \\ \hline \end{array} * f.$$

The associative law is useful in those cases where a succession of *continuous* convolutions needs to be computed of an image. It is often cheaper to first convolve all the convolution kernels with each other, and then convolve the image with the resultant kernel.

Example

If we wish to compute

$$f * g_{\sigma_1} * g_{\sigma_2},$$

where $g_{\sigma_i} = \frac{1}{\sigma_i \sqrt{2\pi}} e^{-\frac{1}{2} \frac{x^2}{\sigma_i^2}}$ are Gaussian kernels, and f is an image, then we can compute the same result with one image convolution by

$$f * h,$$

where

$$h = g_{\sigma_1} * g_{\sigma_2} = \frac{1}{\sqrt{2\pi} \sqrt{\sigma_1^2 + \sigma_2^2}} e^{-\frac{1}{2} \frac{x^2}{\sigma_1^2 + \sigma_2^2}}.$$

The last example shows that the convolution of two Gaussians is again a Gaussian (with σ equal to the square root of the sum of squares of the original two sigmas). This relation is not so easy to establish using standard calculus. In chapter 7 the *Fourier transform* will be introduced, a technique that greatly facilitates the computation of continuous convolutions. However, for discrete kernels, the convolution of two kernels can often be easily computed, as the example below shows.

Example

Suppose we wish to compute $g * (g * f)$ using only one image convolution, with f an image, and $g = [0 \ 1 \ 2]$. We cannot directly use the associative law, since it applies to the $*$ operator, and not directly to the \star operator. First we need to write our expression in a form using \star . This can be done by using the mirror relation

$$g_m * f = g \star f,$$

where g_m is g mirrored in the origin, i.e., $g_m = [2 \ 1 \ 0]$. So $g \star (g \star f) = g \star (g_m * f) = g_m * g_m * f$. To compute this last expression using only one image convolution, we need to compute $g_m * g_m$. Using the mirror relation again: $g_m * g_m = g \star g_m$. We can compute this last expression (using zero padding, i.e., assuming unknown values to be zero, and always taking the center pixel of the kernel for the origin):

$$\boxed{0 \ 1 \ 2} \star \boxed{2 \ 1 \ 0} = \boxed{4 \ 4 \ 1 \ 0 \ 0}.$$

Using the mirror relation one last time:

$$\boxed{4 \ 4 \ 1 \ 0 \ 0} * f = \boxed{0 \ 0 \ 1 \ 4 \ 4} \star f.$$

So the operation $g \star (g \star f)$ can be carried out using one convolution with the formula

$$\boxed{0 \ 0 \ 1 \ 4 \ 4} \star f.$$

The above example illustrates how to arrive at the associative relationship for the \star operator using the mirror relation $g_m * f = g \star f$:

$$h \star (g \star f) = (h \star g_m)_m \star f.$$

This example *en passant* shows that the convolution of two kernels generally leads to a larger kernel (or –at least– to more non-zero kernel values). Although this makes perfect sense from a mathematical point of view, it may be surprising here, as it is *not* common practice to enlarge *images* when applying a convolution operation. In the case of convolving two kernels it is usually a necessity, to avoid the loss of information.

For a second example of combining two convolutions into one convolution, we will show how a high-frequency enhancing operation and a derivative can be combined.

Example

If we want to compute $h \star (g \star f)$, where f is an image, g is the high-frequency

enhancing kernel $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$, and h is the x -derivative kernel $\frac{1}{2} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$, we

can use one image convolution by computing $(h \star g_m)_m \star f$, i.e., (note that $g_m = g$)

$$\left(\frac{1}{2} \cdot \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \star \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \right)_m \star f,$$

which equals

$$\frac{1}{2} \cdot \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & -1 & -1 \\ 1 & -9 & 0 & 9 & -1 \\ 1 & 1 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \star f.$$

In general it is *not* cheaper to combine successive *discrete* convolutions as one convolution. For example, when computing $h \star g \star f$ using two successive convolutions, with h and g 3×3 kernels, we require $2 \times 9 = 18$ multiplications and additions per pixel. If we used one 5×5 kernel $(h \star g_m)_m$, we would need 25 multiplications and additions per pixel. It is therefore useful to see if a discrete convolution operation can be carried using smaller kernels.

Example

The averaging kernel $g = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$ can be decomposed as $g = g_1 * g_2$, where $g_1 = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline \end{array}$ and $g_2 = \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline 1 \\ \hline \end{array}$. Computing $g_2 * (g_1 * f)$ takes less multiplications and additions per pixel (6) than computing $g * f$ does (9).

Separable kernels are kernels that can be separated into functions of a lower dimension, e.g., $g(x, y) = g_1(x)g_2(y)$. Convolution with a separable kernel can be split into separate one-dimensional convolutions. These one-dimensional convolutions together are generally considerably cheaper than computing one multi-dimensional convolution. For instance, in the 2D case:

$$\begin{aligned} g * f &= \iint f(a, b)g(x - a, y - b) da db = \\ &= \iint f(a, b)g_1(x - a)g_2(y - b) da db = \\ &= \int \left[\int f(a, b)g_1(x - a) da \right] g_2(y - b) db. \end{aligned}$$

Since a large number of kernels occurring in practical and theoretical image processing are separable, computing a convolution by separate one-dimensional convolutions is a much-used technique.

5.2 Simple non-linear filtering

Most of the filters we have seen so far in this chapter are *linear* filters; the filter result r at a location is a linear function of the grey values f_i in a neighborhood:

$$r = \sum_i w_i f_i,$$

where w_i are weights, i.e., the entries in a discrete convolution kernel. Other filters, such as the compass filters, use discrete convolution, but the final result of the operator is not a linear function of the grey values. Non-linear filters can by definition not (fully) be

implemented using discrete convolution. An important class of non-linear filters makes use of the order of grey values in a neighborhood, e.g., filters that take the maximum, minimum, or median value of grey values in a neighborhood. Such filters that first order the neighborhood grey values and then select a value at a specific place in the list are called *rank filters*. Filters that use the maximum and minimum grey values in a neighborhood are discussed in depth in chapter 6 on mathematical morphology. The result of the *median filter* equals the median of the ordered grey values in a neighborhood.

Example

Examples of the effect of a 3×3 median filter on simple images:

0	0	0	0	0
0	0	0	1	1
0	0	1	1	1
0	0	1	20	1
0	0	1	1	1

→

0	0	1		
0	1	1		
0	1	1		

0	1	2	3	4
5	6	7	8	9
5	5	5	9	9
5	5	5	9	9
5	5	5	9	9

→

5	5	7		
5	6	9		
5	5	9		

The median filter is often used to remove speckle noise from an image. See for instance the first example above, where the noisy outlier value of 20 is removed from the resultant image. However, the median filter also removes other small details. An example of applying a median filter to a real image is shown in figure 5.18. An interesting effect of the median filter is that edges are not shifted, even if the median filter is repeated a large number of times. Figure 5.18 shows that repeated application of the median filter removes small structures, but that larger structures are preserved. Another effect is that structures are homogenized; the number of contained grey values is reduced.

5.3 Handling image borders

In the examples in the above sections the results of discrete convolution and other neighborhood operations “lose their border”. This is of course undesired in many applications. The border loss is caused by the fact that convolution results at border pixels requires knowledge of grey values at pixel locations that are outside of the image:

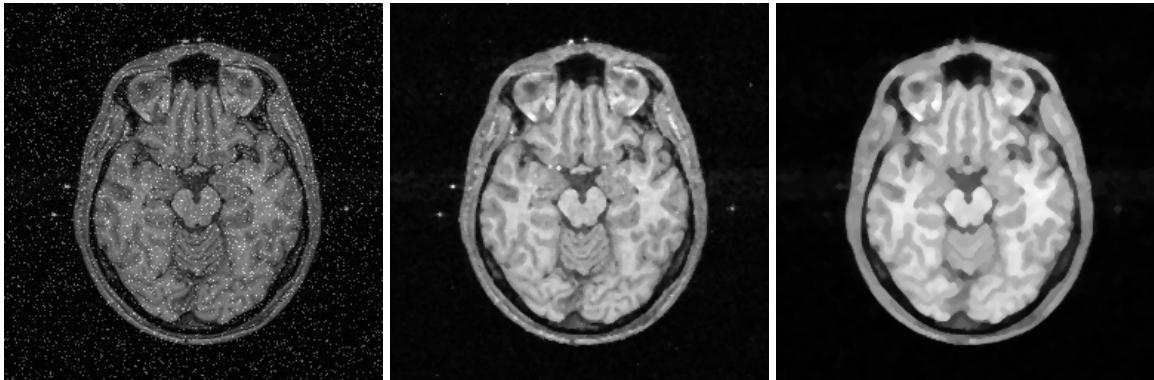
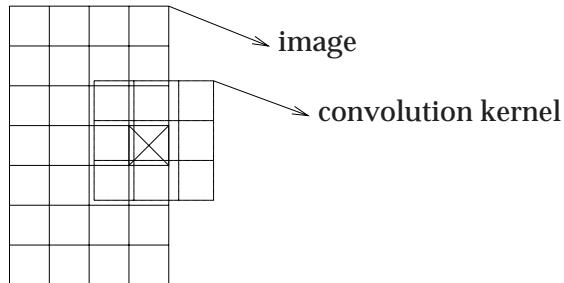


Figure 5.18 Example of applying a 3×3 median filter. The left image shows a real image with added noise. The middle and right image show this image after applying the median filter once and twelve times respectively.

Example



In this example, the convolution result at the marked pixel requires three pixel values that are located outside of the image.

The border loss increases with larger kernels: one pixel thick with a 3×3 kernel, two pixels with a 5×5 kernel, etc.

Although there is no way of establishing the unknown pixel values, we can avoid border loss by *guessing* these values. Three basic approaches exist for this:

- **Padding.** In this case the image is padded at the borders with extra pixels with a fixed grey value. In other words, a fixed value is substituted for the unknown grey values. Usually this is a value common to the image background. In some cases of non-linear filtering it may be useful to pad with a value of $+\infty$ or $-\infty$.
- **Extrapolation.** In this case a missing grey value is estimated by extrapolating it from the grey values near the border. In the simplest case the nearest known pixel value is substituted for the unknown one. In more complex cases a line, a surface,

or a more complex structure is fitted through values near the border, and unknown values are estimated by extrapolating this structure.

- **Mirroring.** In this case a mirror image of the known image is created with the border for a mirroring axis.

Example

Suppose we have an image like the one below, and we need to guess the values in the two empty spaces on the right.

0	1	2		
0	1	2		
0	1	2		

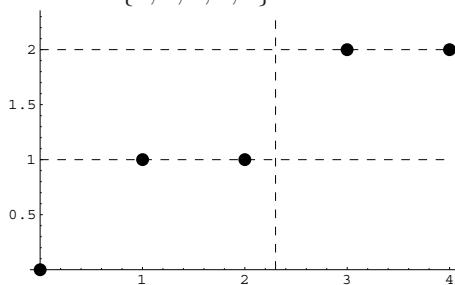
Using zero-padding these values are 0. Simple extrapolation gives values of 2. Extrapolation by fitting a line perpendicular to the border through the nearest image values would give values of 3 and 4 respectively. Mirroring the nearest grey values in the borderline gives values of 2 and 1.

5.4 Interpolation

Many image processing applications require knowledge of grey values at non-pixel locations, *i.e.*, at locations between the grid positions with known grey values. These values need to be estimated (interpolated) from the grey values in a neighborhood. Two common interpolation techniques are *nearest neighbor* and *linear* interpolation. Nearest neighbor interpolation assigns to a location the grey value of the nearest pixel location. Linear interpolation uses a weighted combination of the grey values of the nearest pixels, where a weight w is related to the distance d to a pixel location by $w = 1 - d$.

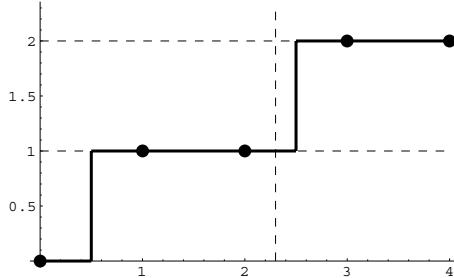
Example: nearest neighbor interpolation

Consider the signal with values $\{0, 1, 1, 2, 2\}$ at the locations $x \in \{0, 1, 2, 3, 4\}$:



Suppose we wish to interpolate the value at $x = 2.3$ (dashed vertical line) by nearest neighbor interpolation, then this value would be 1, since the nearest point ($x = 2$) has value 1.

This graph shows the values at $x \in [0, 4]$ as obtained by nearest neighbor interpolation:



For a 2D image example, consider this image:

0	1	2	3
4	5	6	7
8	9	8	7
6	5	4	3

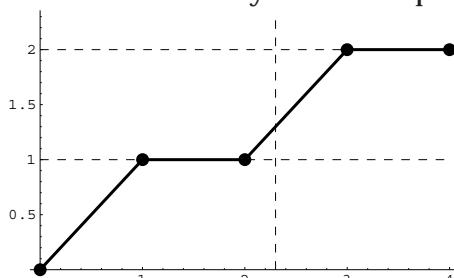
If we choose the origin $(0, 0)$ to be in the center of the top left pixel, then the grey values represent the values at the pixel locations $\{(0, 0), (1, 0), \dots, (3, 3)\}$. The interpolated value at (e.g.) $(0.8, 0.8)$ would be 5, since the nearest pixel, $(1, 1)$, has grey value 5.

Example: linear interpolation

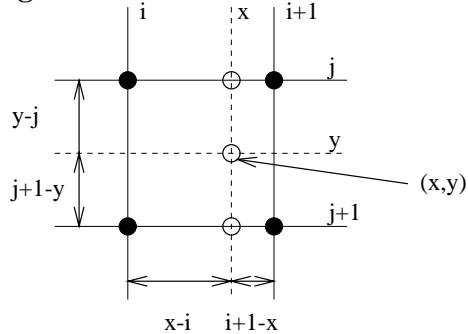
Using the same signal from the previous example, the value v at $x = 2.3$ by linear interpolation can be determined from the nearest points ($x = 2$ and $x = 3$) by a weighted average with weights $(1 - d)$, where d is the distance to the appropriate point:

$$v(2.3) = (1 - 0.3)v(2) + (1 - 0.7)v(3) = 1.3$$

This graph shows the values as obtained by linear interpolation:



For 2D images, we carry out this interpolation twice; once for each direction. Consider this part of an image:



Suppose we wish to find an interpolated value v for (x, y) which is located in the square of pixels $\{(i, j), (i + 1, j), (i, j + 1), (i + 1, j + 1)\}$ as shown above. First we interpolate in the x -direction (top and bottom open circles):

$$\begin{cases} v(x, j) &= (1 - (x - i))v(i, j) + (1 - (i + 1 - x))v(i + 1, j) \\ v(x, j + 1) &= (1 - (x - i))v(i, j + 1) + (1 - (i + 1 - x))v(i + 1, j + 1) \end{cases}$$

Second, we interpolate the values found in the y -direction:

$$v(x, y) = (1 - (y - j))v(x, j) + (1 - (j + 1 - y))v(x, j + 1).$$

These two steps can be combined in a single formula for $v(x, y)$. Linear interpolation in two dimensions is usually called *bilinear* interpolation. Figure 5.19 shows an example of bilinear interpolation applied to a real image.

The linear interpolation result shows a graph or an image that is continuous, but is not smooth at the grid points, *i.e.*, the derivative is not continuous. By fitting higher order curves to the grid points and/or demanding the derivative (and possibly higher order derivatives) to be continuous, we can obtain a smoother interpolation result. Examples of curves that satisfy these conditions are the so-called *B-spline* and *thin-plate spline* functions.

5.4.1 Interpolation by convolution

For theoretical –and sometimes practical– purposes it is useful to model interpolation by a convolution process. This model also shows the relation between nearest neighbor and linear interpolation.

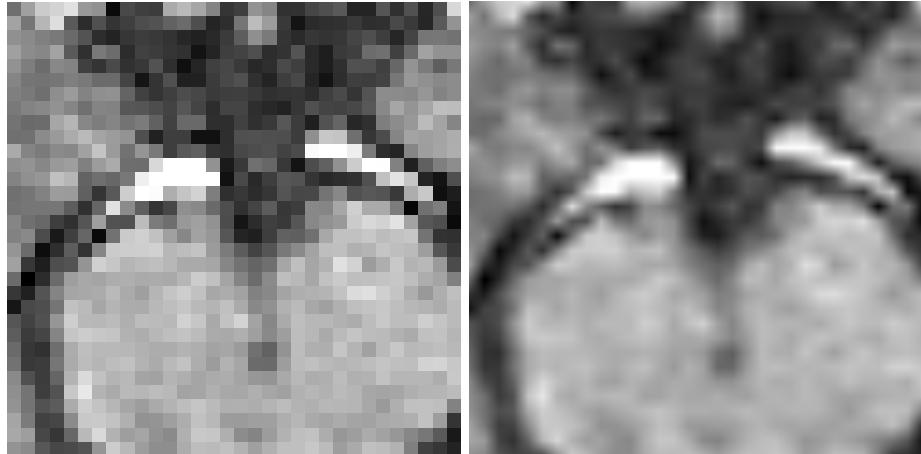


Figure 5.19 Example of bilinear interpolation: the left image, with a 30×30 resolution has been bilinearly interpolated on a 256×256 grid (right image).

A two-dimensional digital image f only has known values at the grid points (i, j) with $(i, j) \in \mathbb{N} \times \mathbb{N}$. We can model the continuous interpolated image $f(x, y)$ by a convolution

$$f(x, y) = \sum_i \sum_j f(i, j)g(x - i, y - j),$$

where g is a *continuous* kernel. We can use a summation instead of an integration because f is only defined at grid points. To simplify matters, we assume that the kernel g is separable, $g(x, y) = h(x)h(y)$ (an assumption that is almost always true in practice), so we only need to examine the one-dimensional case

$$f(x) = \sum_i f(i)h(x - i).$$

A desired property for interpolating kernels is:

$$h(i) = \begin{cases} 0 & \text{if } i \in \mathbb{Z} \setminus \{0\} \\ 1 & \text{if } i = 0. \end{cases}$$

This property ensures that the value of $f(x)$ is not altered (equals $f(i)$) at grid points.

In the case of nearest neighbor interpolation, h equals the kernel h_1 with

$$h_1(x) = \begin{cases} 1 & \text{if } |x| < \frac{1}{2} \\ 0 & \text{elsewhere.} \end{cases}$$

In the case of linear interpolation, h equals the kernel h_2 with

$$h_2(x) = \begin{cases} 1 - |x| & \text{if } |x| < 1 \\ 0 & \text{elsewhere.} \end{cases}$$

There is a relationship between h_1 and h_2 : $h_2 = h_1 * h_1$.⁶ It is therefore interesting to examine the kernels $h_3 = h_1 * h_1 * h_1$, $h_4 = h_1 * h_1 * h_1 * h_1$, etc. Figure 5.20 shows h_i with $i \in \{1, 2, 3, 4\}$. In formulae:

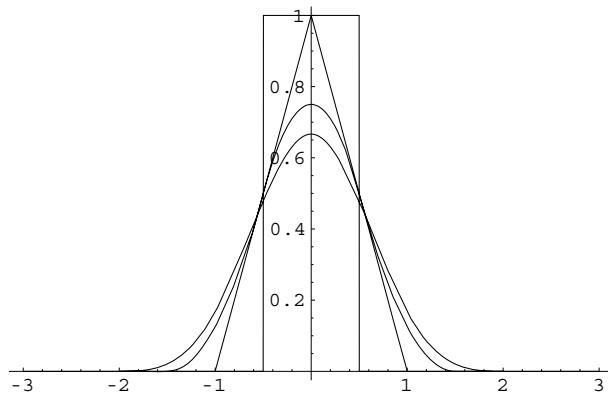


Figure 5.20 The interpolation kernels $h_i(x)$. See text for details.

$$h_3(x) = \begin{cases} \frac{1}{2}x^2 - \frac{3}{2}|x| + \frac{9}{8} & \text{if } \frac{1}{2} \leq |x| < \frac{3}{2} \\ -x^2 + \frac{3}{4} & \text{if } |x| < \frac{1}{2} \\ 0 & \text{elsewhere} \end{cases}$$

$$h_4(x) = \begin{cases} -\frac{1}{6}|x|^3 + x^2 - 2|x| + \frac{4}{3} & \text{if } 1 \leq |x| < 2 \\ \frac{1}{2}|x|^3 - x^2 + \frac{2}{3} & \text{if } |x| < 1 \\ 0 & \text{elsewhere} \end{cases}$$

These kernels h_k are known as *B-spline* interpolation kernels with k basis functions. As k approaches infinity, h_k approaches a Gaussian function, which is why the Gaussian can also be used as an interpolation kernel. Even though the h_k functions are frequently used interpolation kernels –especially the *cubic* B-spline kernel h_4 – we can see that they are not ideal since they do not meet the above mentioned desired property for $k > 2$.

The ideal kernel does in fact exist; the *sinc* function $h(x)$ defined as

$$h(x) = \frac{\sin(\pi x)}{\pi x}$$

⁶Note that, since we are dealing with symmetrical kernels, the $*$ sign may be replaced by \star everywhere in this section.

is an ideal kernel in the sense that it retains all of the spatial frequencies that are present in a digital image. Note that this kernel has the desired property⁷. In practice, we need to truncate or approximate this kernel, since we cannot work with infinitely-sized kernels. Hence, the ideal character cannot be retained in practice.

5.5 Geometric transformation

Geometric transformation is necessary when an image needs to be, e.g., optimized for some viewing position, aligned with another image, or when an image needs distortion correction. Such a transformation consists of, e.g., translation, rotation, scaling, skewing, elastic deformation, etc.

A geometric transformation is a function f that maps each image coordinate pair (x, y) to a new location (x', y') . A geometric image transformation is called *rigid*, when only translations and rotations⁸ are allowed. If the transformation maps parallel lines onto parallel lines it is called *affine*. If it maps lines onto lines, it is called *projective*. Finally, if it maps lines onto curves, it is called *curved* or *elastic*. Each type of transformation contains as special cases the ones described before it, e.g., the rigid transformation is a special kind of affine transformation. A composition of more than one transformation can be categorized as a single transformation of the most complex type in the composition, e.g., a composition of a projective and an affine transformation is a projective transformation, and a composition of rigid transformations is again a rigid transformation. Figure 5.21 shows examples of each type of transformation.

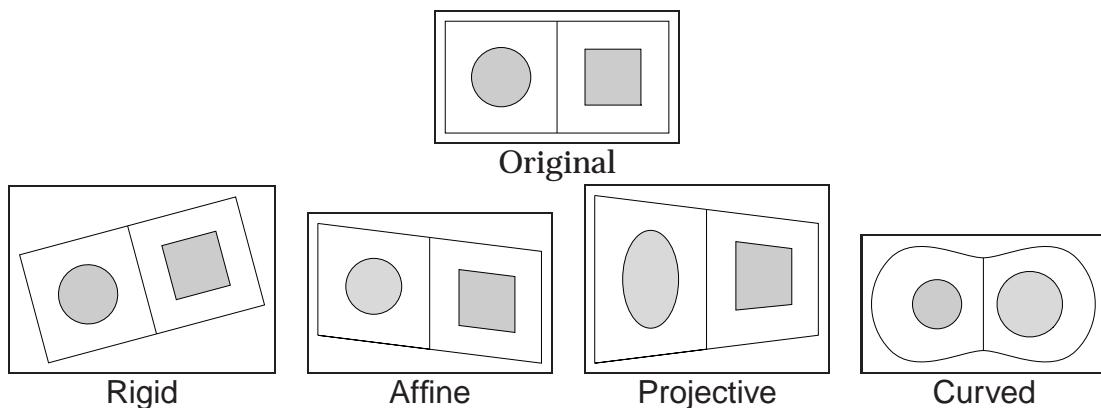


Figure 5.21 Examples of geometric transformations.

⁷Although the sinc function is singular at $x = 0$, the limit value $\lim_{x \rightarrow 0} h(x) = 1$ is commonly substituted for $h(0)$.

⁸and –technically– reflections.

Translation

Translation of an image by a vector $\begin{pmatrix} t_x \\ t_y \end{pmatrix}$ can be achieved by adding this vector to each coordinate pair:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}.$$

Rotation

Positive (anticlockwise) rotation of an image by an angle α around the origin is described by:

$$\begin{cases} x' = x \cos \alpha - y \sin \alpha \\ y' = x \sin \alpha + y \cos \alpha \end{cases},$$

or in matrix notation:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Rigid transformation

We can easily combine translation and rotation in one formula:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}.$$

This is commonly written as a single matrix multiplication:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha & t_x \\ \sin \alpha & \cos \alpha & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Affine transformation

Affine transformation can be described with a similar matrix as the one used for rigid transformations:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

except now the constants a , b , c , and d can be any number.

Scaling

Scaling is a special type of affine transformation defined by

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

The parameters a and d are the scaling parameters in the x and y -direction respectively.

Skewing

Skewing is a special type of affine transformation defined by

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & \tan \alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

The parameter α is called the skewing angle. Figure 5.22 shows some examples of the mentioned transformations.

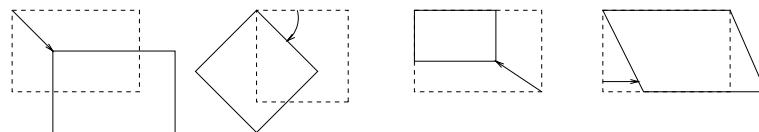


Figure 5.22 Example of (from left to right) a translation, a rotation, a scaling, and a skewing.

Example

Rotation around the image center

The rotation operation as described above does a rotation around the image *origin*. The origin is usually chosen to be at the top left corner of the image, or at the center of the top left pixel. We can rotate the image around its *center* pixel by using two additional translations to shift the origin.

Suppose we have a 256×256 pixel image, with the origin located at the top left of the image, and we wish to rotate it by 30° around the image center. The center of the image is located at $(128, 128)$. (Note: we are working with continuous coordinates here, not integer pixel locations. The origin is at the top left of the image, so

the center of the top left pixel has coordinates $(0.5, 0.5)$.) First we translate the image so that the intended rotation center becomes the image origin, i.e., we translate by a vector $\begin{pmatrix} -128 \\ -128 \end{pmatrix}$:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} -128 \\ -128 \end{pmatrix}.$$

We can now do the rotation:

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}.$$

Finally, we translate the origin back to its original location:

$$\begin{pmatrix} x''' \\ y''' \end{pmatrix} = \begin{pmatrix} x'' \\ y'' \end{pmatrix} + \begin{pmatrix} 128 \\ 128 \end{pmatrix}.$$

By substitution we can find explicit formulas for x''' and y''' directly in terms of x and y .

Projective transformation

The general transformation that describes the projection of a 3D scene with coordinates (x, y, z) to a 2D image with coordinates (x', y') is given by

$$\alpha \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

Since this type of projective transformation occurs in conjunction with 3D images, we will mention here only the special case of the *central perspective* transformation which relates two 2D images that are different projections from the same object, with a point for a light source, as described in figure 5.23. This transformation is described by

$$\begin{cases} x' = \frac{ax+by+c}{gx+hy+1} \\ y' = \frac{dx+ey+f}{gx+hy+1}. \end{cases}$$

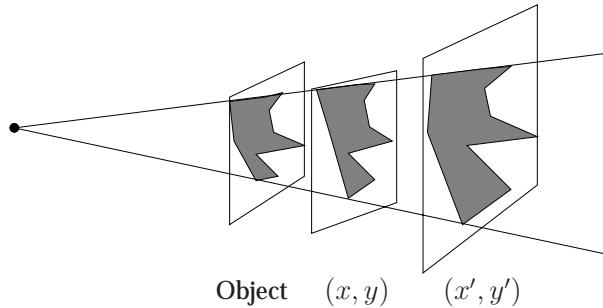


Figure 5.23 The two projection images with coordinates (x, y) and (x', y') are related by a central perspective transformation.

Intermezzo

Dental radiograph registration

Dental radiographs are X-ray projection images of teeth and surrounding structures. With many dental procedures, it is useful to make follow-up radiographs of a patient. Since the imaging circumstances cannot be exactly recreated, the images need to be registered first before an accurate comparison of the radiographs can be done. The transformation that relates the radiographs to be registered can be assumed to be a central perspective one.

To establish the correct transformation, the parameters a, b, \dots, h need to be found. This can be done by identifying four corresponding points in the two images. Their coordinates are measured and substituted in the central perspective equations. This gives us eight equations in the eight unknowns. After solving these equations we can apply the found transformation to the first image to bring it into registration with the second.

In practice, more than four points are usually located to increase the accuracy of the solution found.

Curved transformations

The transformation function f is unconstrained in the case of curved transformations, except that f must be a continuous function.

$$(x', y') = f(x, y),$$

or

$$\begin{cases} x' = f_1(x, y) \\ y' = f_2(x, y) \end{cases}$$

Since the functions f_1 and f_2 are generally unknown, polynomial approximations (usually of an order no higher than five) are often used:

$$\begin{cases} x' = a_0 + a_1x + a_2y + a_3x^2 + a_4xy + a_5y^2 + \dots \\ y' = b_0 + b_1x + b_2y + b_3x^2 + b_4xy + b_5y^2 + \dots \end{cases}.$$

Note that every affine –and hence, every rigid– transformation can be captured by this formulation by choosing appropriate values for a_i and b_i with $i \in \{0, 1, 2\}$.

Intermezzo

The “barrel” and “pincushion” distortions (shown in figure 5.24) that are common to many imaging systems can in most cases be captured and corrected well using third order polynomial transformation.

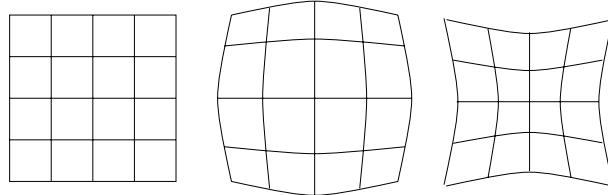


Figure 5.24 Example of barrel (middle) and pincushion (right) distortion of an image (left).

Another representation of curved transformation that is adequate for many applications is to represent the transformation as a sufficiently dense vector field, *i.e.*,

$$\begin{cases} x' = x + t_x(x, y) \\ y' = y + t_y(x, y) \end{cases}.$$

Where the displacement vectors $\begin{pmatrix} t_x(x, y) \\ t_y(x, y) \end{pmatrix}$ are known at a number of image locations (*e.g.*, all of the grid points) and the ‘gaps’ are filled in by interpolation. An example of a vector field is shown in figure 5.25.

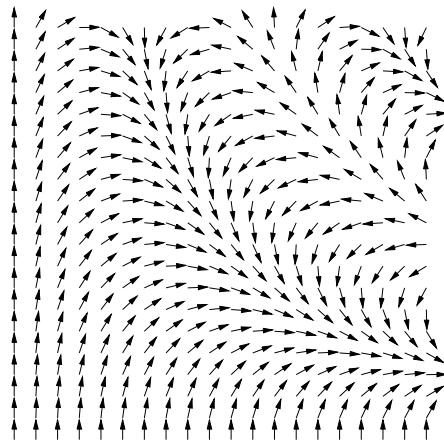


Figure 5.25 Example of a vector field representation of a curved transformation.

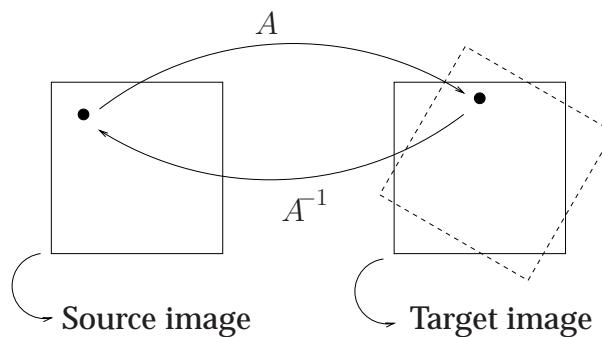


Figure 5.26 Example of backward mapping. The target image must be filled using the backward mapping A^{-1} instead of the forward mapping A .

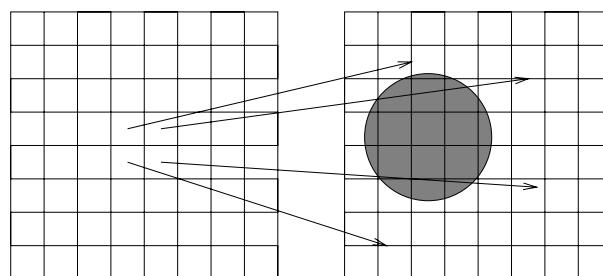


Figure 5.27 Why simple forward mapping can generally not be used for computation of a transformed image. See text for details.

5.5.1 Backward mapping

When computing the transformation of an image we often need the *inverse* (the backward mapping) of the intended transformation (the forward mapping). The reason for this is depicted in figure 5.26. Here, the forward mapping is depicted by the matrix A . By simply applying the forward mapping to all of the pixels in the source image we cannot adequately compute the grey values at all pixel locations of the target image. Figure 5.27 shows this. Here, we have drawn some example forward mappings of pixels. It will be clear that we never “reach” the pixels that are located in the circle area. These pixels should have approximately the same grey values as the ones we *do* reach in this example. By using backward mapping, this problem vanishes: by computing $A^{-1} \begin{pmatrix} x \\ y \end{pmatrix}$ for all pixel locations $\begin{pmatrix} x \\ y \end{pmatrix}$ in the *target* image we can find a corresponding location (and a grey value) in the source image. If we do not exactly end up in a pixel location of the source image, we can approximate an appropriate grey value by interpolation.

In the case of rigid transformations we can establish A^{-1} by matrix inversion. In the affine case too, but we must first check if the matrix is invertible, which is the case as long as $ad - bc \neq 0$. In the case of curved transformation finding an inverse (if it exists at all) often proves a very difficult task, and often the best way to compute a transformed image is to use forward mapping with computer graphics techniques.

Chapter 6

Mathematical morphology

Morphology is the study of shape. *Mathematical morphology* mostly deals with the mathematical theory of describing shapes using sets. In image processing, mathematical morphology is used to investigate the interaction between an image and a certain chosen *structuring element* using the basic operations of *erosion* and *dilation*. Mathematical morphology stands somewhat apart from traditional linear image processing, since the basic operations of morphology are non-linear in nature, and thus make use of a totally different type of algebra than the linear algebra.

Although the mathematics behind this¹ is fascinating, treating it here would be beyond the scope of this book. In practical image processing, it is sufficient to know that morphology can be applied to a finite set P if

1. we can *partially order* its elements, (where the ordering is denoted by “ \leqslant ”), i.e., for all $a, b, c \in P$

$$a \leqslant a$$

$$(a \leqslant b, b \leqslant a) \Rightarrow a = b$$

$$(a \leqslant b, b \leqslant c) \Rightarrow a \leqslant c,$$

and

2. each non-empty subset of P has a maximum and minimum.

Example

Any finite set of real or integer numbers is a suitable set P

The ordering “ \leqslant ” is defined as in ordinary calculus ($3 \leqslant 4$, $4 \leqslant 18$, etc.). The maximum and minimum are also defined in the usual sense (e.g., $\max\{5, 3, 4\} = 5$).

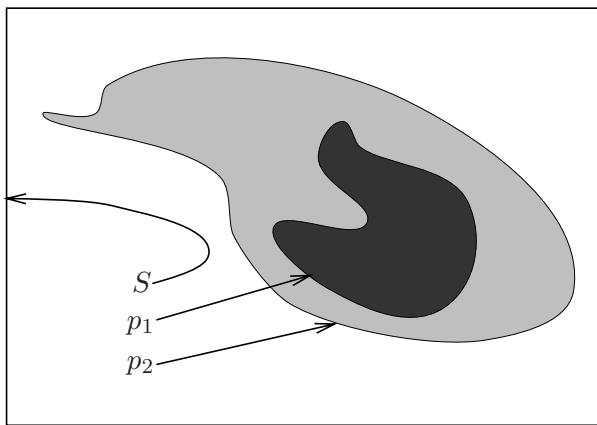
¹The mathematics of *lattices*.

This means we can apply morphology to grey-valued images (or subsets of images), because the collection of grey values can be viewed as a finite set P with ordering, maximum, and minimum well defined.

Example

The set of all subsets of a superset S is a suitable set P

The ordering “ \leqslant ” is defined by the subset relation “ \subset ”. The figure below shows an example, with $p_1 \subset p_2$, i.e., $p_1 \leqslant p_2$.



The maximum and minimum are defined by the union (\cup) and intersection (\cap) operators respectively. In this example, $\max\{p_1, p_2\} = \cup\{p_1, p_2\} = p_2$, and $\min\{p_1, p_2\} = \cap\{p_1, p_2\} = p_1$.

Besides finite sets of real and integer numbers and the set of subsets, more abstract sets P can be constructed. Mathematical morphology can be applied to any of these sets. In this chapter we are only concerned with the two sets mentioned in the examples above.

We can extend the partial ordering to digital images by applying the rules to the individual pixels. For instance, for two images f and g , the relation $f \leqslant g$ holds if:

$$f \leqslant g \iff \forall x : (f(x) \leqslant g(x)),$$

where “ $\forall x$ ” refers to all possible pixel locations. The “maximum image” and “minimum image” of two images can also be defined on a pixelwise basis:

$$\begin{aligned} (\max\{f, g\})(x) &= \max\{f(x), g(x)\} \\ (\min\{f, g\})(x) &= \min\{f(x), g(x)\}. \end{aligned}$$

The concepts of ordering, maximum, and minimum are fundamental in mathematical morphology. In fact, the maximum and minimum operator are the *only* basic operators used in mathematical morphology applied to images! And the only useful morphological operators are operators that somehow retain the order of images (*increasing* operators, see next section), or retain their maxima or minima.

6.1 Complement and operator properties

The *complement* X^c of a set X is defined as all elements not belonging to X . For an image f , the complement f^c is defined as f mirrored in a central grey-value line, as shown in figure 6.1. For an image with a range $\{L, \dots, M\}$ the complement can be written as

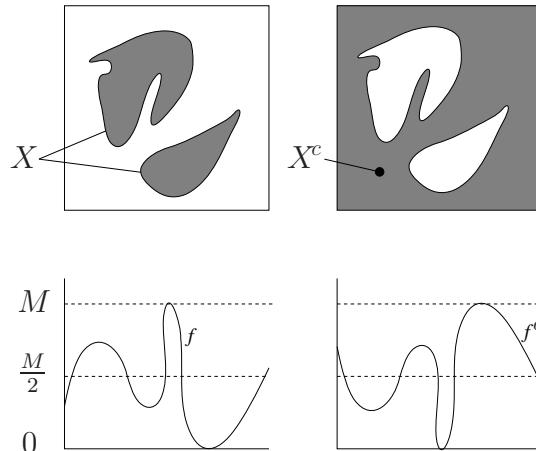


Figure 6.1 The complement of a set X (top row) and of an image or function f (bottom row).

$f^c(x, y) = L + M - f(x, y)$. Complementing twice results in the original set or function, i.e., $(X^c)^c = X$, and $(f^c)^c = f$.

Example

If f equals

0	1	5
4	6	3
7	2	8

then the range is $\{0, \dots, 8\}$, and the complement f^c can be computed by $f^c(x, y) = 8 - f(x, y)$. So f^c equals

8	7	3
4	2	5
1	6	0

A fast way of verifying that the two images are complementary is to check that the sum of two corresponding pixels is always 8. Complementing the complement again will yield the original image.

Notice that the complement definitions for sets and images are consistent: if we represent the set X in figure 6.1 as a function (an image) f , e.g., by

$$f(x, y) = \begin{cases} 1 & \text{if } (x, y) \in X \\ 0 & \text{elsewhere} \end{cases},$$

then applying the complement as defined for images will give the same result as applying the complement to the set X . Since we can convert a set to a binary image as in the formula above, we will only discuss images in the remainder of this section.

The operators φ_1 and φ_2 are called *dual* if applying φ_1 to the complement of f equals complementing the result of applying φ_2 to f , i.e., if the following holds:

$$\forall f : \quad \left(\varphi_1(f^c) = (\varphi_2(f))^c \right).$$

Example

The max and min operators applied to images are dual.

For an example, suppose f equals

0	0	1
4	2	3
3	3	4

then f^c equals

4	4	3
0	2	1
1	1	0

We see that $\max(f^c) = 4$ and that $\min f = 0$. Complementing the latter expression gives us $(\min f)^c = 4$. So max and min are dual in this example:

$$\max(f^c) = (\min f)^c.$$

Many of the pairs of morphological operators presented in the next sections are dual, e.g., erosion and dilation, and opening and closing. Some operators are self-dual ($\varphi(f^c) = (\varphi(f))^c$). For example, the median filter from the previous chapter is self-dual.

Example

Given the images f and f^c :

0	0	1
4	2	3
3	3	4

4	4	3
0	2	1
1	1	0

Then the median values are 3 and 1 respectively. Since $(1) = (3)^c$ here, the median is self-dual:

$$\begin{array}{ccc} 1 & = & (3)^c \\ \text{median}(f^c) & = & (\text{median}(f))^c \end{array}$$

An operator φ is called *increasing* if it does not alter the order of images:

$$\forall f, g : \quad (f \leq g \implies \varphi(f) \leq \varphi(g)).$$

An operator φ is called *extensive* if

$$\forall f : \quad (f \leq \varphi(f)),$$

i.e., the output is always “larger” than the input. See figure 6.2. If the reverse holds, then φ is *anti-extensive*:

$$\forall f : \quad (\varphi(f) \leq f).$$

Increasingness and (anti-)extensivity are desired properties in a large number of operations. They are also a necessity in many theoretical considerations concerning morphological operators.

An operator is called *idempotent* if applying the operator more than once has no effect:

$$\forall f : \quad (\varphi(\varphi(f)) = \varphi(f)).$$

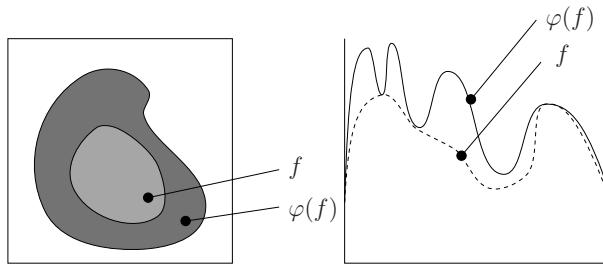


Figure 6.2 Example of an extensive operation φ applied to a set (a binary image) and a function (an image).

6.2 Relation between sets and images

Many of the morphological operators in the next section are only defined for sets. Because we want to use these operators on images, we need to define a relationship between sets and images, so we can transfer these set-operators to images. In the previous section we already defined the relation between a binary image and a set X : X is defined by all the pixels with grey value 1, the set $\{(x, y) | f(x, y) = 1\}$. This idea can be expanded so that we can define a relation between grey valued images and sets: we regard each specific grey level in the image as a set; we regard the image f as a stack of sets F with the relation

$$F(c) = \{(x, y) | f(x, y) \geq c\}.$$

This is demonstrated by figure 6.3.

We are now able to “translate” any operators defined only for sets to apply to images: we apply the operator to all of the level sets $F(c)$ that together make up the image. This seems tedious, but in practice the resulting operations can often be simplified a great deal.

6.3 Erosion and dilation

Morphological operations describe the interaction of an image with a *structuring element* S . The structuring element is usually small relative to the image. In the case of digital images, we typically use simple binary structuring elements like a cross or a square, such as the ones in figure 6.4².

²Note: whenever we display *binary* images in this book we reverse the normal convention “bright pixel = high grey value, and dark pixel = low grey value”. In the case of binary images we use the convention that *object* pixels have grey value 1 and are displayed *black*, while *background* pixels have grey value 0 and are displayed *white*. Although this may be somewhat confusing in the beginning, it usually enhances the clarity of binary images.

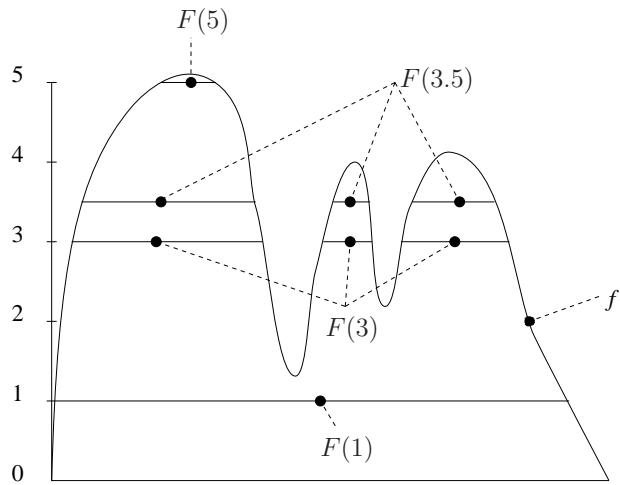


Figure 6.3 An image or function f can be viewed as a stack of sets $F(c)$. The level sets $F(1)$, $F(3)$, $F(3.5)$, and $F(5)$ are shown. See text for details.

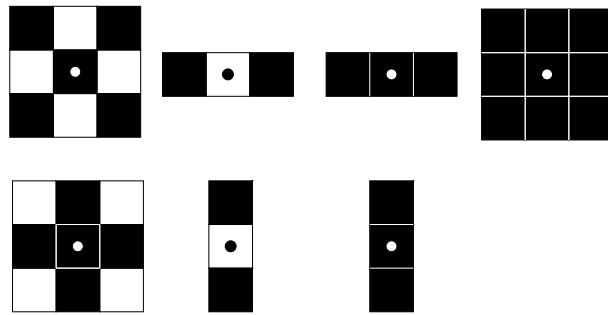


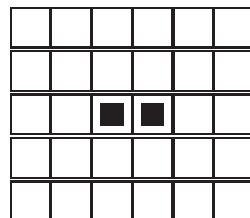
Figure 6.4 Example of simple binary structuring elements. Black pixels are part of the element, white ones are not. The pixel with the circle marks the origin.

The *dilation*³ $\delta(X)$ of a set (binary image) X by the structuring element S is defined by

$$\delta(X) = \{x + s \mid x \in X \wedge s \in S\}.$$

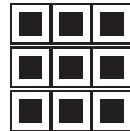
Example

Suppose the set X is represented by the marked pixels in this binary image:



³Also called Minkowski addition.

i.e., the set $X = \{(2, 2), (3, 2)\}$. Suppose S is a square structuring element S with the origin at its center:



so $S = \{(-1, -1), (0, -1), (1, -1), \dots, (1, 1)\}$. The dilation δ of X by S equals $\delta(X) = \{x+s | x \in X \wedge s \in S\}$, i.e., the set of all possible additions of an element of X and an element of S . For example: $(2, 2) + (-1, -1) = (1, 1)$ or $(3, 2) + (0, 1) = (3, 3)$. This results in the following set:

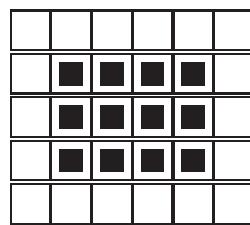


Figure 6.5 shows another example of a dilation of a set. The dilation by a *symmetrical*⁴

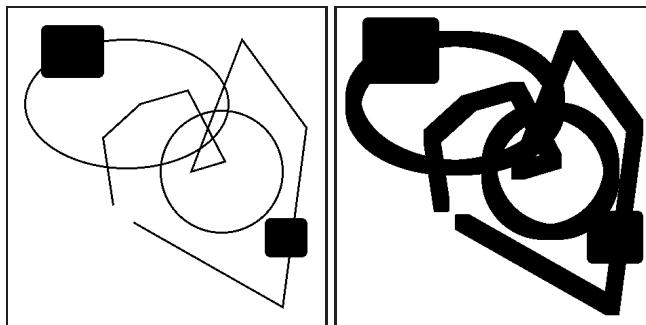


Figure 6.5 Example of the dilation (right) of a set (binary image, left) by a square structuring element (shown on the far right).

structuring element is described more intuitively by:

- Place the structuring element anywhere in the image
- Does it hit the set? Then the origin of the structuring element is part of the dilated set.

⁴Symmetrical in the origin.

If the structuring element S is not symmetrical we must use its *transpose* \check{S} in the above procedure. The transpose is the structuring element mirrored in the origin:

$$\check{S} = \{s | (-s) \in S\}.$$

The dual operation of dilation is called *erosion*⁵, and the erosion $\varepsilon(X)$ of a set X by a structuring element S is defined by

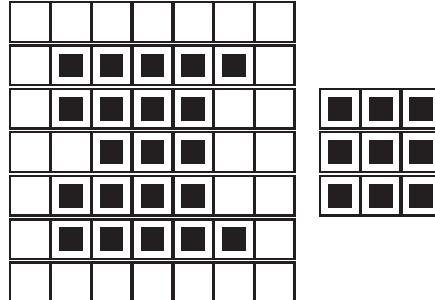
$$\varepsilon(X) = \{x | \forall s \in S, x + s \in X\}.$$

More intuitively, it is described by:

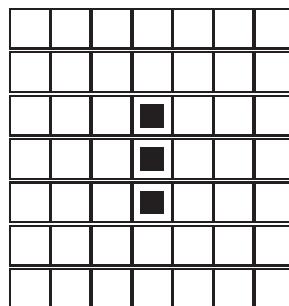
- Place the structuring element anywhere in the image
- Is it fully contained by the set (*i.e.*, a subset)? Then the origin of the structuring element is part of the eroded set.

Example

Given the set X and the structuring element S with the origin at its center:



then the erosion $\varepsilon(X)$ equals:



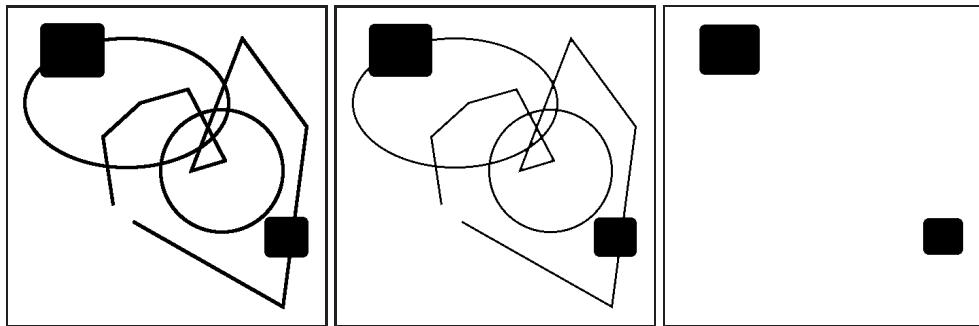


Figure 6.6 Example of erosion of a set (binary image) with a square structuring element. On the left the original set is shown. In the middle is the result after one erosion, and to the right the result after another erosion with the same structuring element.

Figure 6.6 shows another example of an erosion of a set.

Using the relation between sets and images described in the previous section we can also give formulas for erosion ε and dilation δ of a digital image f with a structuring element S :

$$(\varepsilon(f))(x) = \min_{s \in S} f(x + s)$$

$$(\delta^*(f))(x) = \max_{s \in S} f(x - s).$$

Where x and s are vector quantities if f is an image. The minus sign in the definition of the dilation is counter-intuitive in most practical use. To avoid this, we will often use the alternative definition

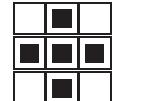
$$(\delta(f))(x) = \max_{s \in S} f(x + s).$$

Although it has taken a number of theoretical steps to arrive at the above formulas for dilation δ and erosion ε , their application to digital images is straightforward, as the next example shows.

Example

Given the image f and a 3 pixel wide cross for a structuring element S :

1	1	0	0	1	2
2	2	1	0	0	1
3	3	3	1	0	1
3	3	1	0	1	2
3	1	0	1	2	3
1	0	1	2	3	4



,

⁵Also called Minkowski subtraction.

we can compute the erosion ε of f at each pixel (x, y) by centering S at (x, y) , and then taking the minimum of all pixels of f that are ‘hit’ by S . For example, the erosion at $(2, 2)$ equals the minimum of the values $\{1, 3, 3, 1, 1\}$, i.e., 1.

The dilation is computed in the same way, except that the maximum is now taken. The erosion $\varepsilon(f)$ and dilation $\delta(f)$ are respectively:

1	0	0	0	0	1
1	1	0	0	0	0
2	2	1	0	0	0
3	1	0	0	0	1
1	0	0	0	1	2
0	0	0	1	2	3

2	2	1	1	2	2
3	3	3	1	1	2
3	3	3	3	1	2
3	3	3	1	2	3
3	3	1	2	3	4
3	1	2	3	4	4

where we used padding with values of $-\infty$ and ∞ in the cases of dilation and erosion respectively.

The figures 6.7 and 6.9 show examples of grey valued dilation and erosion applied to real images. Figure 6.8 clearly shows that the dilation is a maximum of a neighborhood defined by the structuring element.

Intermezzo

Different definitions of erosion, dilation, and other morphological operations may be found in other books. For example, the dilation is usually defined with the minus sign rather than the plus sign. This has interesting theoretical properties, like making operator duality straightforward, but has drawbacks like muddling the intuitive nature of the operation.

When comparing morphological operations from different texts, verify if their definitions match, or –if not– what the differences are.

Differences in definitions usually boil down to transposition (mirroring) of either the structuring element or the image. We have seen this problem before at the definition of discrete convolution: the formal definition (*) has nice arithmetic and other theoretical properties, but the “working” definition (correlation; *) is easier to understand in practical applications.

Note that all differences in definitions vanish if we use structuring elements or kernels that are *symmetrical* in the origin. Since the structuring element then equals its own transpose, there is no difference in the result, whatever the formal definition of an operation is. For example, the $*$ and \star operations have the same result if we use a symmetrical kernel.

In this book we will make use of the practical definitions, and avoid unintuitive transpositions as much as possible.

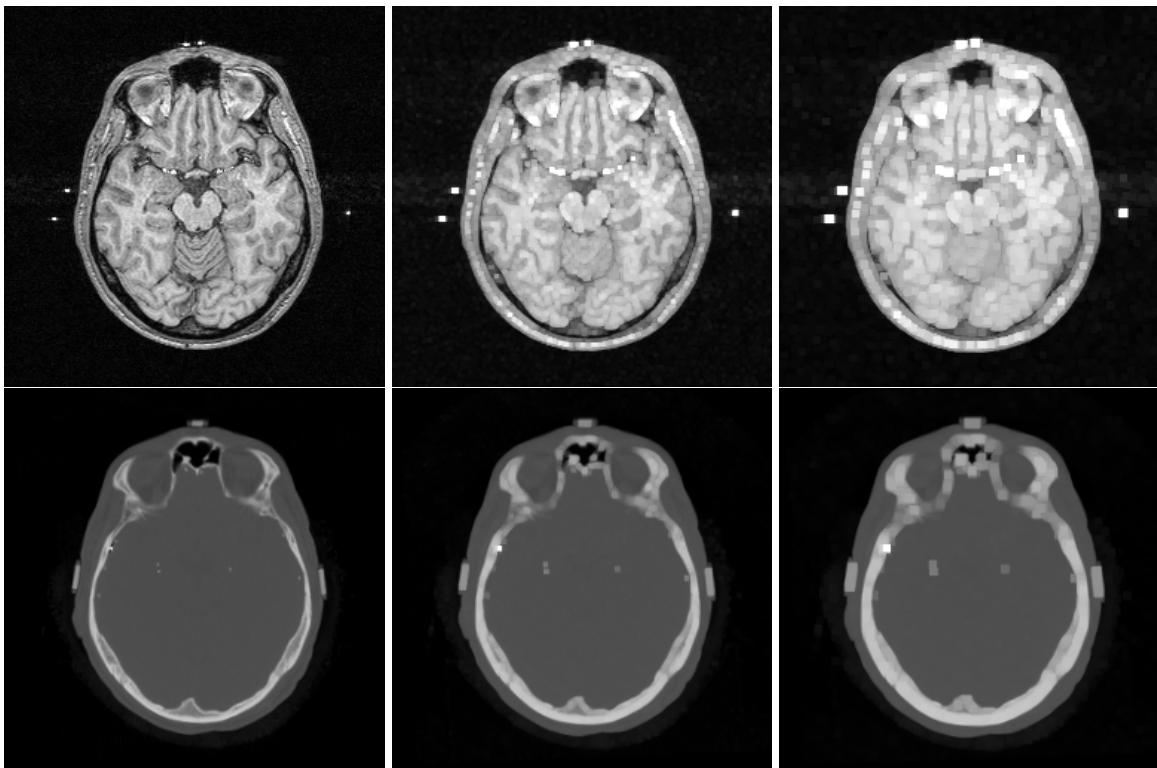


Figure 6.7 Example of dilation by a 3×3 square structuring element (middle column) and a 5×5 square structuring element (right column), applied to a 256×256 MR image (top row) and a 256×256 CT image (bottom row).

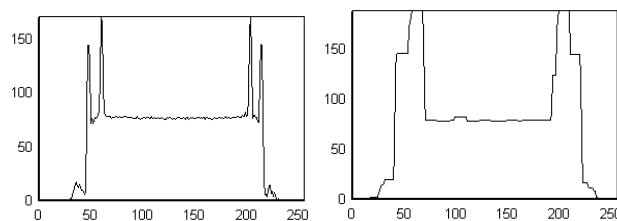


Figure 6.8 The left graph shows a plot of grey values along a horizontal line obtained from the CT image in the previous figure. The right graph shows the same plot after dilation; each local grey value at coordinate x corresponds to the maximum grey value in a small neighborhood around x in the original plot.

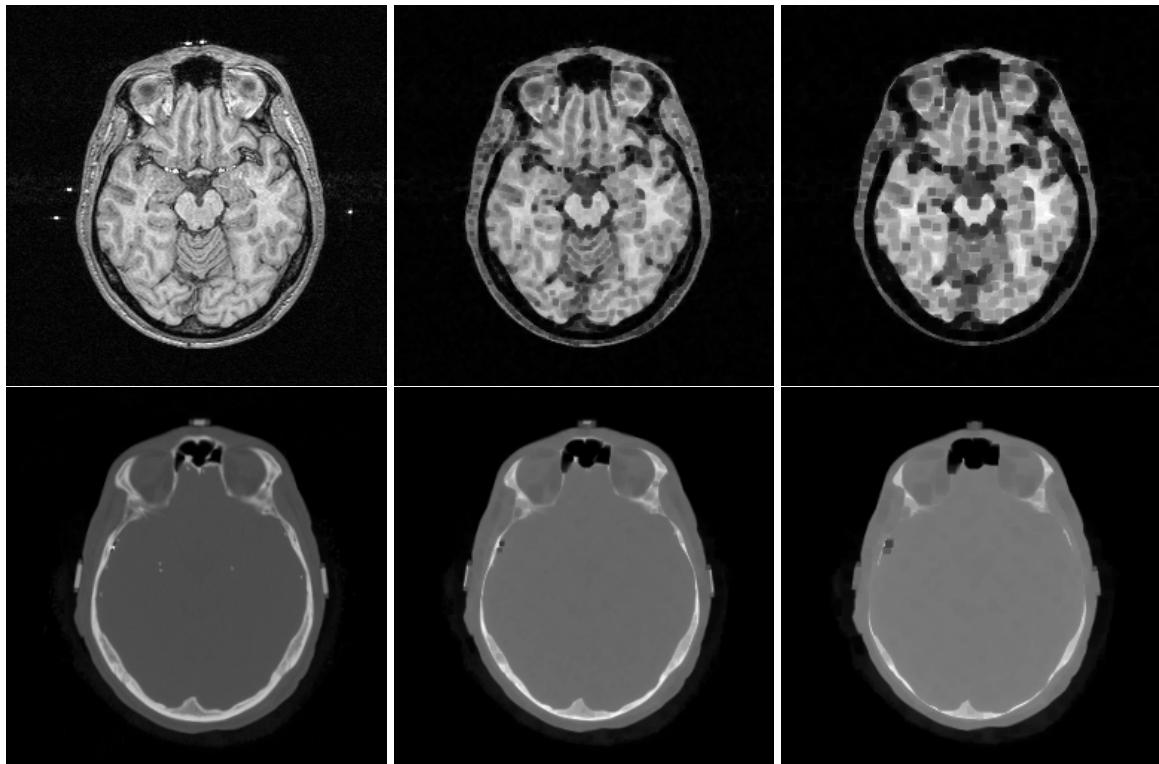


Figure 6.9 Example of erosion by a 3×3 square structuring element (middle column) and a 5×5 square structuring element (right column), applied to a 256×256 MR image (top row) and a 256×256 CT image (bottom row).

6.3.1 Properties of erosion and dilation

When examining the figures showing erosion and dilation, it will be obvious that the names *erosion* and *dilation* describe the effect of the operations well. The erosion operation “eats” chunks away at the boundaries of objects, while the dilation operation does the opposite. It is however noteworthy that –owing to the non-linear character of the operations– erosion is *not* the inverse of dilation. The inverse of either operation does not exist. For instance: many different input images can have the same erosion, so this operation cannot be reversed.

Many special properties hold for erosion and dilation. Assuming a *symmetrical* structuring element for simplicity:

Duality: erosion and dilation are dual operations:

$$(\varepsilon(f))^c = \delta(f^c).$$

Increasingness: Both the erosion and dilation are increasing operations:

$$f \leq g \Rightarrow \begin{cases} \varepsilon(f) \leq \varepsilon(g) \\ \delta(f) \leq \delta(g) \end{cases}$$

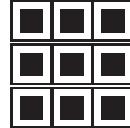
Extensivity: If the origin is part of the structuring element, the dilation is extensive, and the erosion is anti-extensive:

$$0 \in S \Rightarrow \begin{cases} \varepsilon(f) \leq f \\ f \leq \delta(f) \end{cases}$$

Separability: The symmetrical structuring element can be separated in one-dimensional parts. The erosion or dilation can be carried out using one-dimensional erosions and dilations:

$$S = \delta_{S_1}(S_2) \Rightarrow \begin{cases} \varepsilon_S(f) = \varepsilon_{S_1}(\varepsilon_{S_2}(f)) \\ \delta_S(f) = \delta_{S_1}(\delta_{S_2}(f)) \end{cases}$$

For example: dilation by



is equivalent to dilation by



followed by dilation with



6.4 Opening and closing

The composition of the erosion and dilation operations has interesting properties. The morphological *opening* γ and *closing* φ are defined by:

Opening: $\gamma(f) = \delta(\varepsilon(f))$

Closing: $\varphi(f) = \varepsilon(\delta(f))$.

To understand what e.g., a closing operation does: imagine the closing applied to a set; the dilation will expand object boundaries, which will be partly undone by the following erosion. Small, (i.e., smaller than the structuring element) holes and thin tube-like structures in the interior or at the boundaries of objects will be filled up by the dilation, and not reconstructed by the erosion, inasmuch as these structures no longer have a boundary for the erosion to act upon. In this sense the term 'closing' is a well chosen one, as the operation removes holes and thin cavities. In the same sense the opening opens up holes that are near (with respect to the size of the structuring element) a boundary, and removes small object protuberances. Examples of opening and closing can be seen in the example below and the figures 6.10 and 6.11.

Example

Consider the image f :

1	1	0	0	0	0	0
1	1	1	0	0	0	0
1	1	1	1	1	0	0
0	1	0	0	1	0	0
0	1	0	0	1	0	0
0	1	1	1	1	1	0
0	0	0	0	1	1	1
0	0	0	0	0	1	1

then the erosion $\varepsilon(f)$ and the dilation $\delta(f)$ by a cross-shaped structuring element



using padding with $+\infty$ and $-\infty$ as before, are

1	0	0	0	0	0	0
1	1	0	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	1

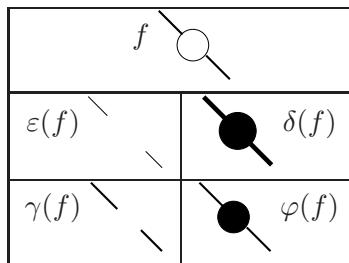
1	1	1	0	0	0	0
1	1	1	1	1	0	0
1	1	1	1	1	1	0
1	1	1	1	1	1	0
1	1	1	1	1	1	0
1	1	1	1	1	1	1
0	1	1	1	1	1	1
0	0	0	0	1	1	1

The opening $\gamma(f) = \delta\varepsilon(f)$ and closing $\varphi(f) = \epsilon\delta(f)$ are

1	1	0	0	0	0	0
1	1	1	0	0	0	0
1	1	1	0	0	0	0
0	1	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	1	1	1	0
0	0	0	0	1	1	1
0	0	0	0	0	1	1

1	1	0	0	0	0	0
1	1	1	0	0	0	0
1	1	1	1	1	0	0
1	1	1	1	1	0	0
1	1	1	1	1	1	0
0	1	1	1	1	1	0
0	0	0	0	1	1	1
0	0	0	0	0	1	1

These abstractions of the images show that the names *erosion*, *dilation*, *opening*, and *closing* actually do what their names suggest:



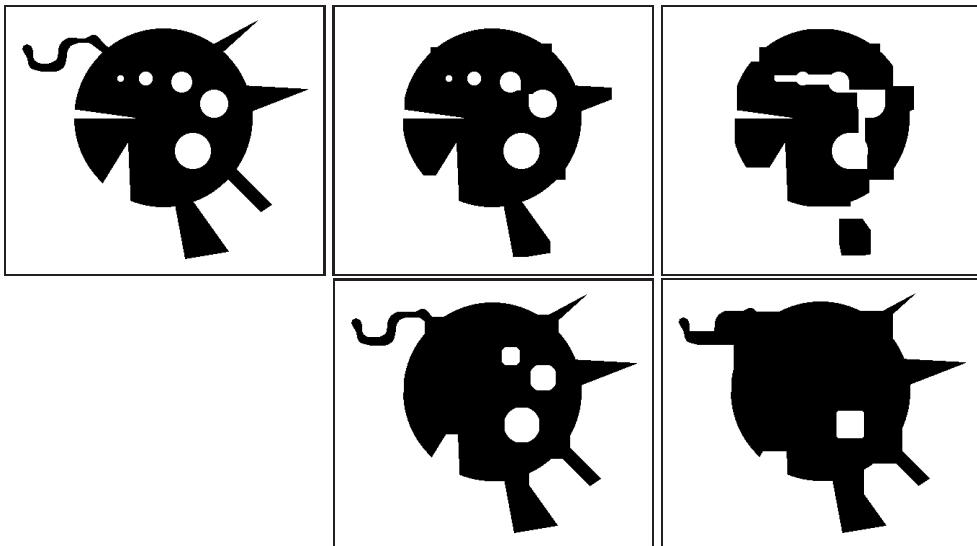


Figure 6.10 Example of opening and closing on a set (binary image). The top row shows the original, and two openings with a square structuring element of increasing size. The bottom row shows the corresponding closings.

6.4.1 Properties of opening and closing

Assuming a symmetrical structuring element:

Duality: Opening and closing are dual operations.

Increasing: Since the opening and closing are compositions of increasing operations (erosion and dilation), the opening and closing are also increasing.

Extensive: The opening is anti-extensive, the closing is extensive.

Idempotent: Both the opening and closing are idempotent operations, *i.e.*, applying them to an image twice gives us the same result as applying them only once:

$$\begin{aligned}\gamma\gamma(f) &= \gamma(f) \\ \varphi\varphi(f) &= \varphi(f)\end{aligned}$$

6.5 Geodesic operations and reconstruction

Dilation is an extensive operation; structures “grow” at their boundaries. It is often useful to constrain this growth in a way such that structures do not grow outside of some pre-defined boundaries. A way to achieve this is to use *geodesic dilation* $\delta_g(f)$, which is defined as the minimum of the ordinary dilation $\delta(f)$ and a control image g that contains the restraining boundaries:

$$\delta_g(f) = \min(\delta(f), g).$$

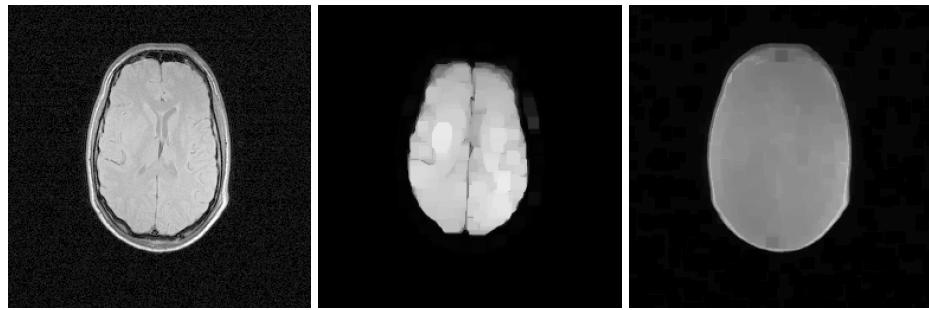


Figure 6.11 Example of opening (middle) and closing (right) by a square structuring element on a grey valued MR image (left).

Its counterpart, geodesic erosion, can be defined in a similar manner by

$$\varepsilon_g(f) = \max(\varepsilon(f), g).$$

Example.

Given f and g by

Then $\delta(f)$ and $\delta_g(f)$ using a 5×5 square structuring element are

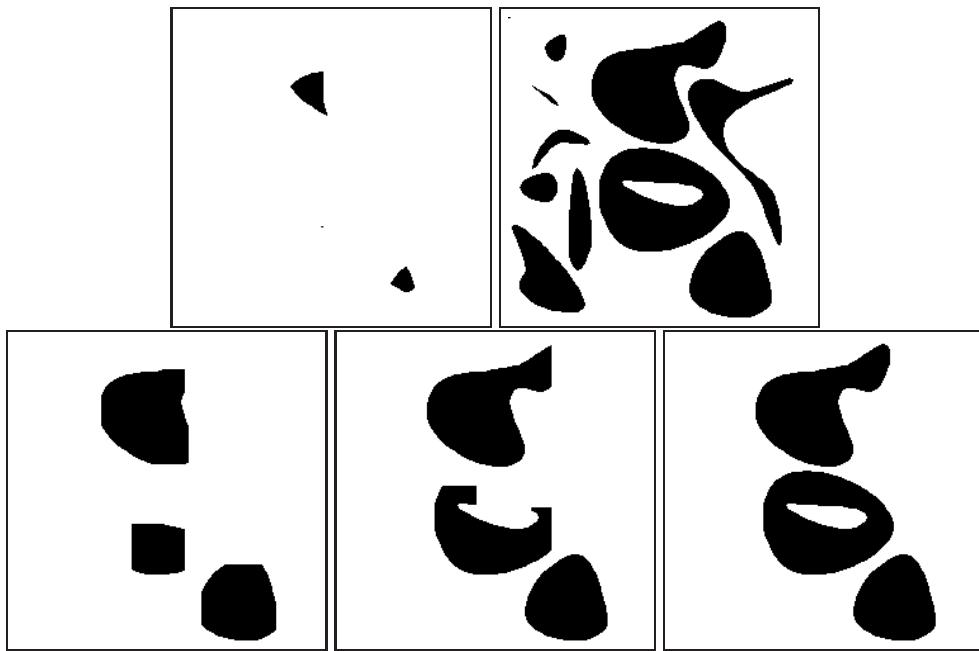
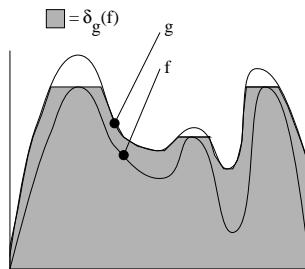


Figure 6.12 Example of geodesic dilation of a binary image. The top row shows the original image f and a control image g . The bottom row shows three geodesic dilations with a structuring element of increasing size.

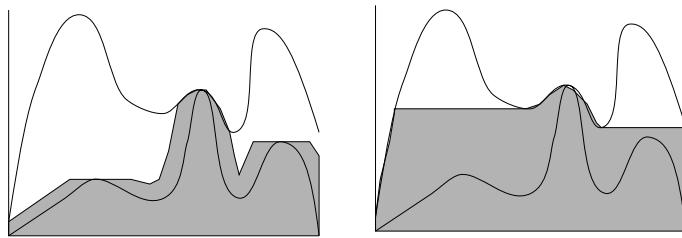
A more visual example is shown in figure 6.12.

Example



The above figure shows a one-dimensional example of a geodesic dilation $\delta_g(f)$. The boundaries of g stop f from “growing” outside of the region defined by g .

The below figure shows a geodesic dilation with a different choice of f . The left image shows one geodesic dilation step, the right image shows the result after a large number of geodesic dilations.



6.5.1 Opening by reconstruction of erosion

It can be shown that iterated application of geodesic dilation or erosion to an image will converge to a stable result. If we iterate the operations until stability occurs, this is called *reconstruction by dilation* and *reconstruction by erosion* respectively. As we know, the composition of erosion and dilation is called an opening. The special composition of erosion and reconstruction by dilation is called *opening by reconstruction of erosion* or just an *opening by reconstruction*.

Opening by reconstruction is a very effective way of removing small structures from (especially binary) images: the erosion step removes small structures from an original image. It will also erode the boundaries of larger structures, but parts of these structures remain present in the eroded image. If we now do a reconstruction by geodesic dilation *with the original image for a control image*, these parts will grow back to their original size. The small structures that were completely removed by the erosion will not grow back. Effectively, the opening by reconstruction has removed small structures, while keeping the larger structures completely intact. The figures 6.13 and 6.14 show examples of this.

The size of the structuring element in the erosion step determines what structures are removed by the opening by reconstruction. The larger we choose this element, the more structures will be removed. In figure 6.13, we can note that it is also possible to remove the *large* structures from an image rather than the small ones: this can be achieved by taking the difference image of the left and right images respectively.

6.5.2 Other openings by reconstruction

Reconstructions can be made of erosions of images, as in the previous section. Another common procedure is to reconstruct an image from an image that contains only a set of markers.

Example

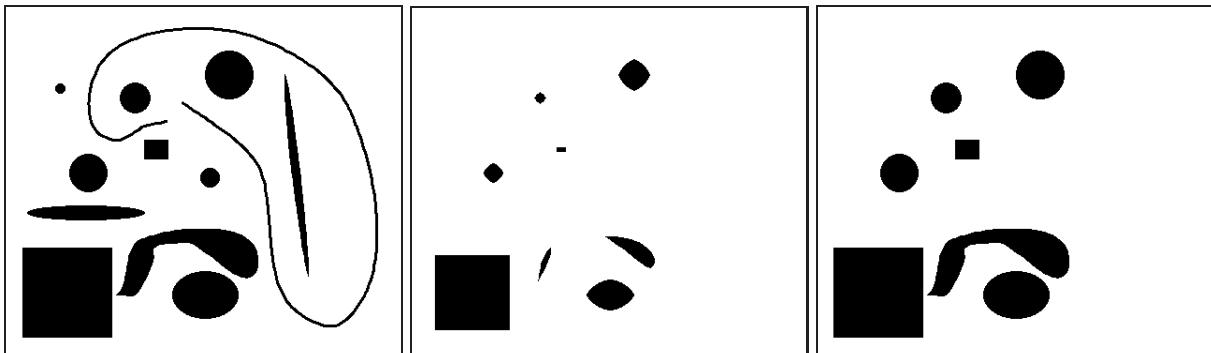
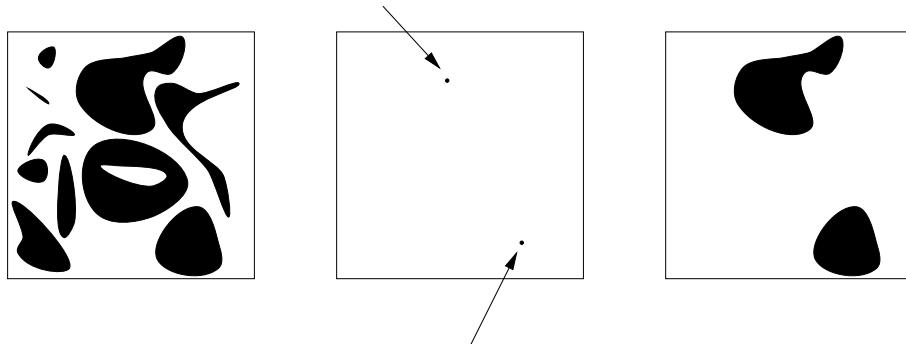


Figure 6.13 Example of opening by reconstruction on a binary image. On the left the original image is shown, in the middle the image after erosion, and on the right after opening by reconstruction using the original image for a control image. The larger structures from the original image that are still “marked” in the eroded image have been completely reconstructed. The smaller structures that were completely removed by the erosion remain absent in the reconstructed image.



Suppose we have an original image (left), and obtained a marker image (middle). Then the opening by reconstruction of the marker image using the original image for a control image is the image on the right. Only the “marked” objects have been reconstructed.

The marker image can e.g., be obtained by “user clicks” in a computer application, or can be the result of some image processing routine run on the original image.

Example

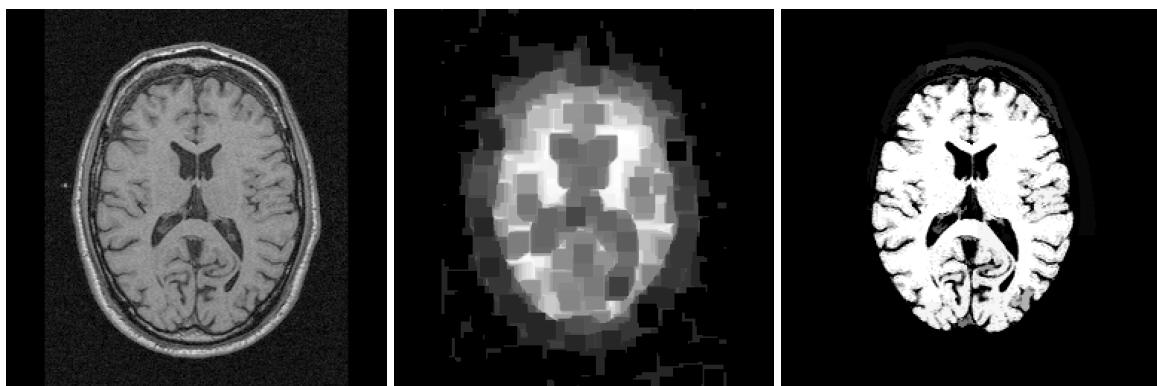
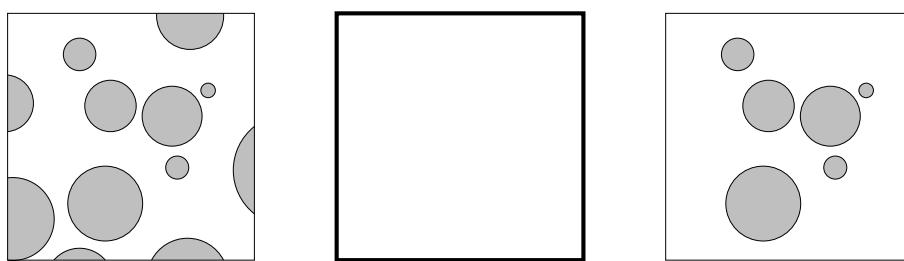


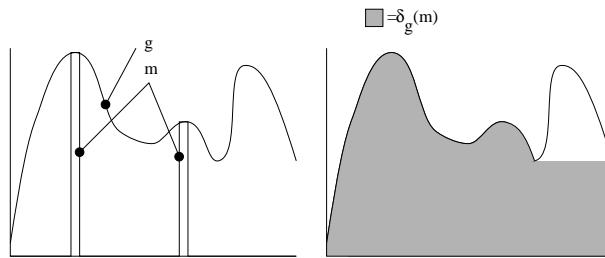
Figure 6.14 Example of opening by reconstruction of an MR image. To the left the original image is shown, in the middle the image after erosion, and on the right after opening by reconstruction –with the original image for a control image– and thresholding.



An application of reconstruction from a marker image can be the removal of objects that intersect the image boundary in a binary image (left image). If we choose the image boundary to be the marker image (middle image), then the reconstruction of the original image from this marker image will contain only those objects that intersect the boundary. If we subtract this reconstructed image from the original, we retain only those objects that do *not* intersect the boundary, as seen in the right image.

Example

Another example of an application of reconstruction from markers is the removal of unwanted maxima (and their neighborhood) from images, as shown in the image below.



The left image shows the original image and the marker image: two out of the three image maxima are marked. The right image shows the reconstruction from the marker image using the original image for a control image. The unmarked maximum and its neighborhood have vanished.

6.6 Residues

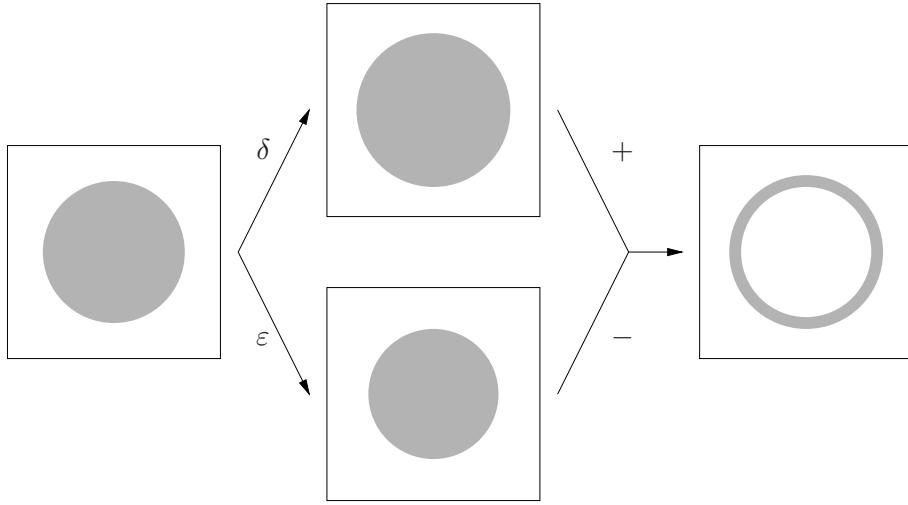
Many interesting morphological filters can be formed using *residues*, i.e., the *differences* of two or more common operations. Examples of residue operations on an image f are $\delta(f) - \varepsilon(f)$ or $f - \gamma(f)$.

We can distinguish three types of residue, based on what type of operations are used in the difference

1. Using a difference of two primitives, e.g.,
 - Morphological gradient
 - Morphological Laplacian
 - Top hat filter
2. Using differences of two families of primitives, e.g.,
 - Skeleton
 - Ultimate erosion
3. Using the hit-or-miss transformation (see section 6.6.4) in the difference, e.g.,
 - Thinning and thickening

6.6.1 Morphological gradient filters

We have seen that the erosion and dilation operations act only on the edges of objects: the erosion “eats” chunks away, while the dilation does the opposite; making objects “grow” at their edges. Using these notions, we can detect edges by examining the difference between an original image and its erosion and dilation.

Example

By taking the difference of the dilation and erosion of an image, we can detect edges of objects in the original image.

The *morphological gradient* $g(f)$ of an image f is defined by

$$g(f) = \delta(f) - \epsilon(f).$$

This “thick” gradient, that sticks out on two sides of the actual edges, can be decomposed into two “half” gradients: $g(f) = g^+(f) + g^-(f)$ with

$$\begin{aligned} g^-(f) &= f - \epsilon(f) \\ g^+(f) &= \delta(f) - f, \end{aligned}$$

where the “inner” gradient $g^-(f)$ adheres to the inside⁶ of objects, and the “outer” gradient $g^+(f)$ adheres to the outside of objects. Examples can be seen in the figures 6.15 and 6.16.

As there is a morphological equivalent of the (length of the) gradient, there is also an equivalent of the Laplacian: the morphological Laplacian $\Delta(f)$ is defined as the residue of the outer and inner gradient:

$$\Delta(f) = g^+(f) - g^-(f).$$

An example can be seen in figure 6.17.

⁶i.e., the bright side of an edge in a grey valued image.

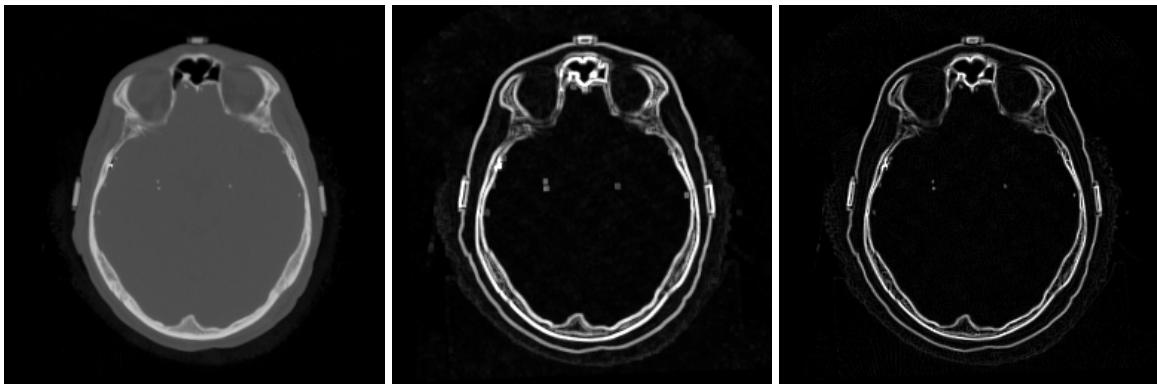


Figure 6.15 Example of the morphological gradient (middle image) and the inner gradient (right image) of a CT image (left image) with a 256×256 resolution using a 3×3 square structuring element.

6.6.2 Top hat filters

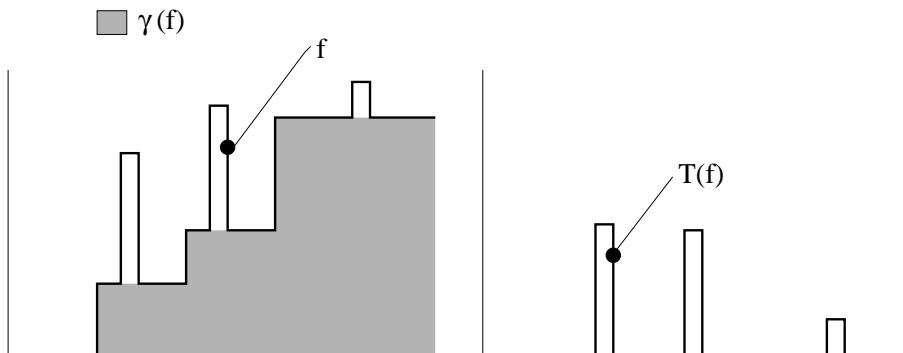
In previous sections, we have seen that an opening can be used to remove structures smaller than a certain size⁷ from an image, while not –or rather, as little as possible– altering larger structures. The dual operation, the closing, can be used to close up holes and cavities that are smaller than a certain size.

If an opening removes small structures, then the difference of the original image and the opened image should bring them out. This is exactly what the *white top hat* $T(f)$ filter does, which is defined as the residue of the original and opening:

$$T(f) = f - \gamma(f).$$

Example

The figure below shows an image f and an opening $\gamma(f)$. The opening removes small structures in f . The white top hat $T(f) = f - \gamma(f)$ extracts just these structures.



⁷Where “size” (and shape) is relative to the structuring element used.

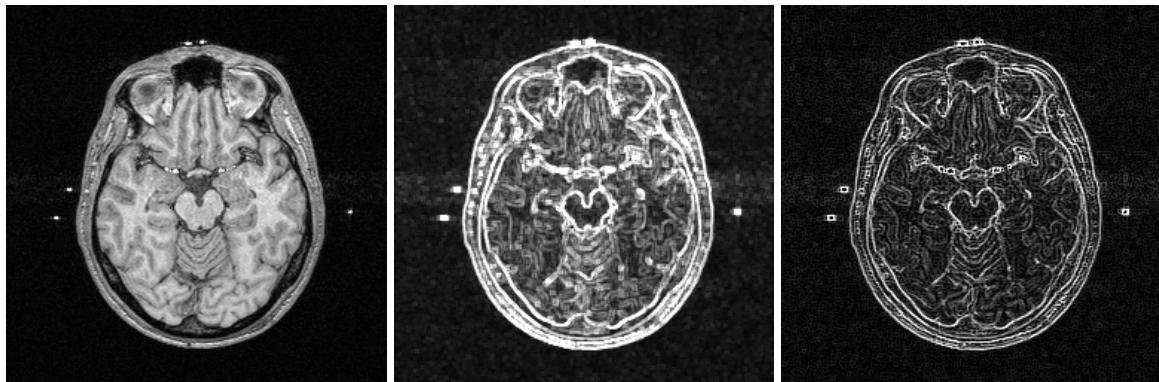


Figure 6.16 Example of the morphological gradient (middle image) and the outer gradient (right image) of an MR image (left image) with a 256×256 resolution using a 3×3 square structuring element.

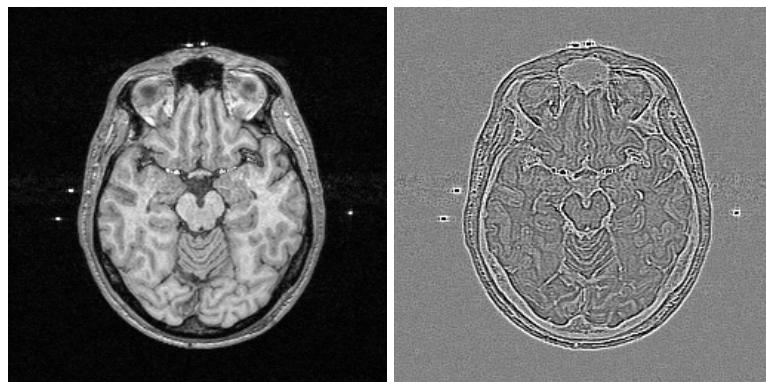


Figure 6.17 Example of the morphological Laplacian (right) of an MR image (left). The computations were done using a 3×3 square structuring element.

The above example shows another aspect of the white top hat: not only does it show small structures, but it shows them with a grey value that is *relative to the local background*: the grey value of the extracted small structures is relative to the local grey value in the neighborhood in the original image. This observation gives rise to another application of the white top hat: if we use a large enough structuring element, we effectively extract *all* structures in the original image, but with a grey value relative to the local background grey value. In this way, we can remove slow grey value variations in the background of an image, e.g., an illumination gradient in a photograph. Figure 6.18 shows an example of this use of the white top hat.

The counterpart of the white top hat is the *black top hat filter* $T^*(f)$ which is defined by

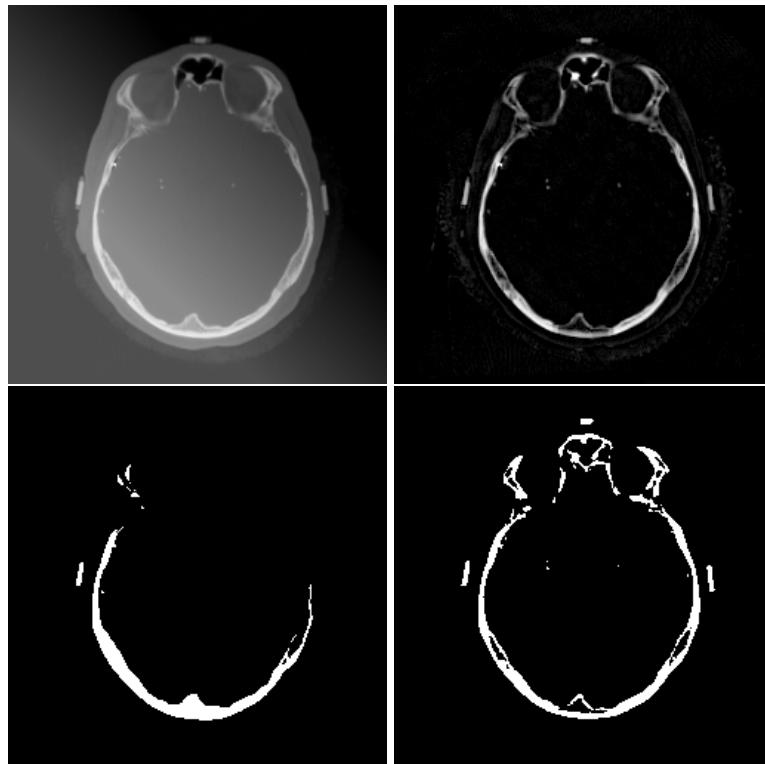


Figure 6.18 The top left image shows a 256×256 CT image with an added background grey value variation. As the image on the bottom left shows, we can no longer use thresholding to extract the bone from this image, because of the variation in grey values in the bony structures. The top right image shows the CT image with added variation after applying a white top hat filter using a square structuring element of 7×7 , a size which should be large enough to extract all the bony structures. As the bottom right image shows, thresholding *this* image gives us the desired result.

the residue of closing and the original:

$$T^*(f) = \varphi(f) - f.$$

As the white top hat extracts small “white” (larger grey value than the background) structures, the black top hat extracts small dark structures, *i.e.*, holes and cavities. Both the white and black top hat filters are idempotent.

Intermezzo

Many variations on the top hat filters exist. In most cases, the standard opening (or closing for the black top hat) is replaced by a more elaborate opening (closing),

e.g., an opening by reconstruction. Another example is the following variation of a white top hat:

$$T\text{var}(f) = f - \min(\gamma(\varphi(f)), f).$$

A variation which is less sensitive to noise than the standard top hat. The standard top hat –since it extracts all structures small relative to the structuring element– also extracts noise from the image.

6.6.3 The skeleton and ultimate erosion

The skeleton and ultimate erosion are concepts that are most intuitive in the context of sets and binary images. Although there are definitions that apply to grey valued images, we will only discuss binary versions of the skeleton and ultimate erosion here.

In the previous sections, we have discussed filters that are the residue of two primitive operations. In this section, we will discuss residues of *families* of primitives. A *family* $\{\psi_i\}$ is a set of morphological operators with a structuring element that depends on the parameter i .

Example

A family of erosions $\{\varepsilon_i\}$ by square structuring elements is for instance the following family of erosions by the structuring elements:

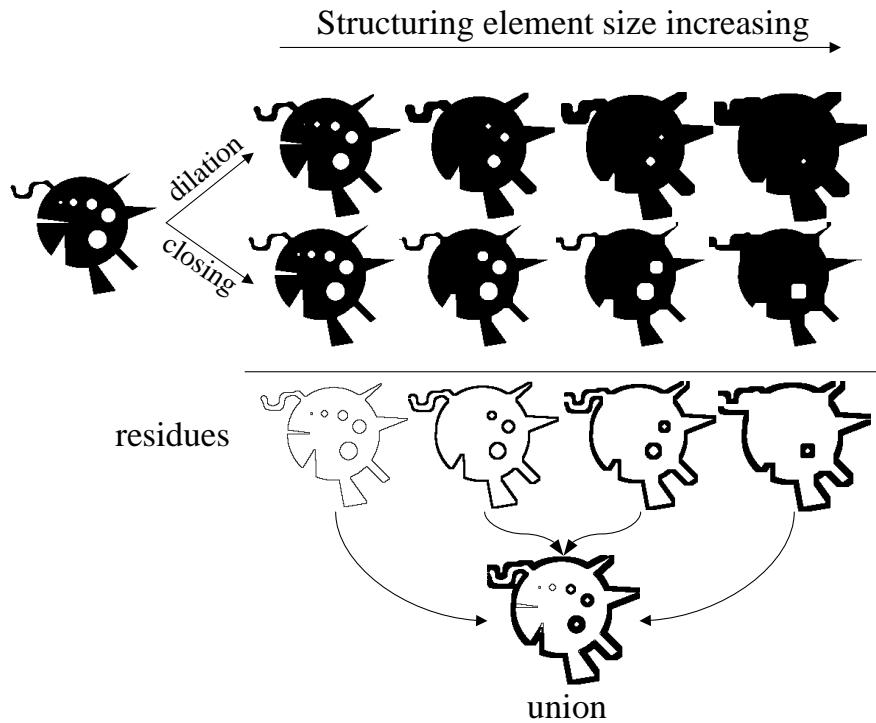


For a set (or binary image) S , the residue R of two families of primitives $\{\psi_i\}$ and $\{\varphi_i\}$ is defined as the union of the residues for each value of i :

$$R_{\{\psi_i\}, \{\varphi_i\}}(S) = \bigcup_i (\psi_i(S) - \varphi_i(S)).$$

Example

The graphical equivalent of the residue of a family of dilations and a family of closings (for four values of i) is shown below.



Intermezzo

Formally, a better definition of the residue of two families (purely in terms of sets) would be

$$R_{\{\psi_i\}, \{\varphi_i\}}(S) = \bigcup_i \left(\psi_i(S) \setminus \varphi_i(S) \right),$$

where \setminus is the “set minus” symbol, so $\psi_i(S) \setminus \varphi_i(S)$ means the set $\psi_i(S)$ without the set $\varphi_i(S)$. This definition avoids negative values that may occur in our definition.

We can easily adapt our definition to this set definition by setting all negative values to zero. In practice, the families $\{\psi_i\}$ and $\{\varphi_i\}$ are often chosen such that no negative values can occur.

6.6.3.1 Ultimate erosion

The *ultimate erosion U* of a set is defined as a series of erosions that leaves a single “marker” for each component of the set untouched. That is, each component is eroded

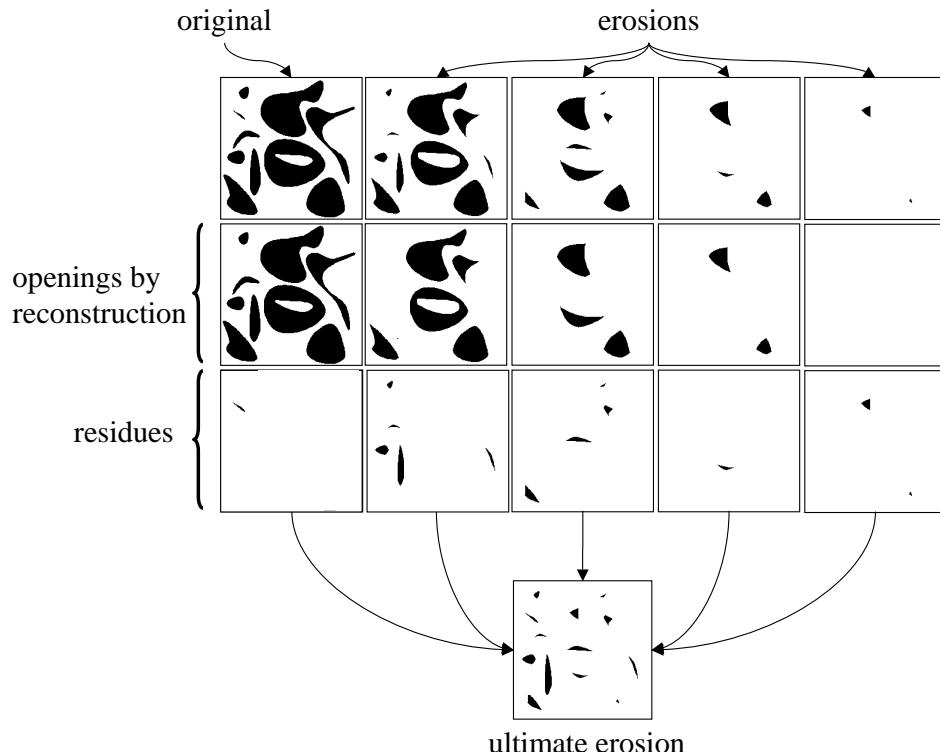
until one more erosion would remove the component entirely. The part of the component remaining before this final erosion step is retained in the ultimate erosion. We can compute the ultimate erosion by using the families of erosion and opening by reconstruction:

$$U(S) = R_{\{\varepsilon_i\}, \{\gamma_{\text{rec}, i}\}}(S),$$

where $\gamma_{\text{rec}, i}$ is an opening by reconstruction using $\varepsilon_{i+1}(S)$ for the erosion and ε_i for the control image.

Example

How does this formula for $U(S)$ work? Take a look at this figure:



The erosions will remove structures, but with the ultimate erosion we are not allowed to remove an object entirely. So we must include an eroded object in the ultimate erosion set just *before* it will be removed entirely in the next erosion step. So we must know if an object will be removed in the next step erosion step. We can do this using an opening by reconstruction (which was an erosion followed by a reconstruction using geodesic dilation). In the opening by reconstruction, objects that disappear in the next erosion step will not be reconstructed. We can find

these objects by taking the difference (residue) of the current image before and after opening by reconstruction. This residue contains exactly those remnants of objects that will disappear in the next erosion step, so this residue is part of the ultimate erosion.

Note that the ultimate erosion is not perfect in the sense that it always leaves exactly one marker for each object in the original image. In the example above, thirteen markers are placed for ten original objects. More than one marker per object appears if the original object contains relatively thin bridges between outer parts. If a bridge 'breaks' in the erosion process, this may result in two or more markers associated with the original object. It is possible to adapt the definition of ultimate erosion to force only one marker per object, but in practice the above definition is more commonly used, because it assigns multiple markers to objects that –although connected in the image– are in fact overlapping separate objects in reality.

6.6.3.2 The skeleton

The *skeleton* of a set is a stick figure representing the basic shape of the set. It is best explained using the “grass fire” analogy: imagine the objects in your set to be patches of grass. Now we set fire to all the grass at the boundaries of the objects, and assume the fire to burn with constant speed until all of the grass is gone. The skeleton of the set is formed by all points where the wave fronts of the fire meets. Figure 6.19 shows some examples of grass fire skeletons.

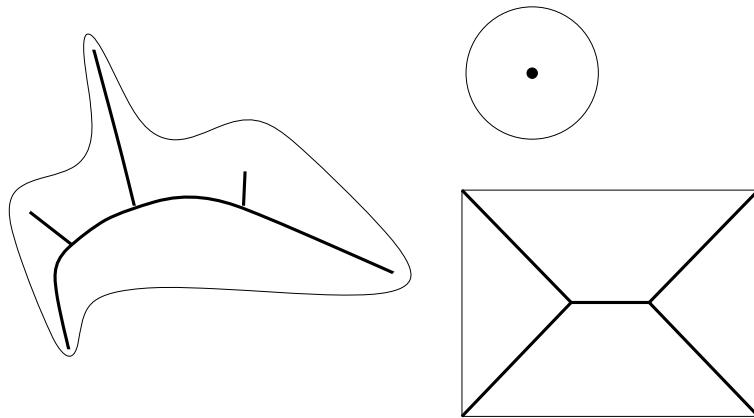


Figure 6.19 Some sets and their corresponding grass fire skeletons.

In the example on ultimate erosion (section 6.6.3.1 above), the ultimate erosion $U(S)$ is the union of five component residues, $U(S) = \cup\{U_0, U_1, \dots, U_4\}$, with $U_i = \varepsilon_i(S) -$

$\gamma_{\text{rec},i}(S)$. An interesting observation is that the dilation of U_i with δ_i results in a so-called **maximal ball**. The maximal ball is a set that fits completely inside the original set S . It is maximal in the sense that dilation with a larger element than δ_i results in a set that no longer fits inside S .

Formally, we define as the **ball** $B_i(x)$ the dilation of the point x with δ_i . x is the center of the ball. $B_i(x)$ is a maximal ball if:

$$B_i(x) \text{ is maximal} \Leftrightarrow (\nexists y, k) | B_i(x) \subset B_k(y) \subset S.$$

In words: there is no other ball in S containing $B_i(x)$. Figure 6.20 shows some examples of maximal balls.

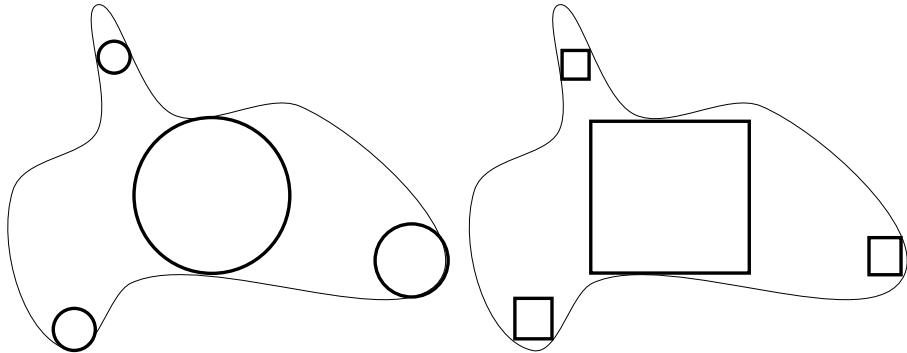


Figure 6.20 Examples of maximal balls in sets. On the left, a circular structuring element is used. On the right, the “balls” are formed by a square structuring element.

We can compute the centers of the maximal balls of size i of a set S by computing the residue $\text{MB}_i = \varepsilon_i(S) - \gamma((\varepsilon_i)(S))$, where the opening γ uses the smallest possible structuring element. The **skeleton by maximal balls** $S_{\text{MB}}(S)$ of a set S is defined as the set of centers of all maximal balls of S :

$$S_{\text{MB}} = \bigcup_{i=0}^{\infty} \{\text{MB}_i\}.$$

For this definition to be useful in the case of digital images, we must include the “trivial” 1×1 structuring element in the computation, and choose for the opening γ the next smallest structuring element.

Example

When computing the skeleton by maximal balls using the family of square structuring elements in a digital image, we use for structuring elements

<i>i</i>	structuring element
0	1×1 square
1	3×3 square
2	5×5 square
3	7×7 square
:	

and always a 3×3 square for the opening operation γ .

Example

Given the image $f =$

then $\varepsilon_0(f) = f$, and $\gamma\varepsilon_0(f)$ and MB_0 equal

$\varepsilon_1(f)$ equals

so $\gamma\varepsilon_1(f)$ is empty. Therefore $\text{MB}_1(f) = \varepsilon_1(f)$. For $i = 2$ and higher, the erosion will remove the image entirely, so the skeleton by maximal balls equals $S_{\text{MB}} = \text{MB}_0 \cup \text{MB}_1 =$



Figure 6.21 In the middle, a skeleton by maximal balls is shown. This skeleton was obtained using a square structuring element. On the right, a skeleton obtained by thinning is shown.

Figure 6.21 shows an example of a skeleton obtained in this way.

A disadvantage of the skeleton by maximal balls is that the skeleton may be disconnected within an object. In practice, instead of the skeleton by maximal balls a skeleton formed by *thinning* (see section 6.6.5) is often used.

When computing the skeleton by maximal balls, we know the size i of the maximal ball around each point in the skeleton. It is therefore easy to extend the binary skeleton image to a grey value image, where each point of the skeleton gets a grey value equal to the size of the local maximal ball. The resulting ‘function’ is called a *quench* function. Figure 6.22 shows an example of a quench function. The quench function gives us a very compact representation of the original image: given only the quench function we can reconstruct the original image by dilating each point with the structuring element of the appropriate size. It is also possible to compute all openings of the original image by dilating only those points with quench values above a certain threshold. The quench function also compactly gives shape information of the original: the “center” lines of the objects, and the value of the quench function gives the morphological distance to the nearest object boundary.

6.6.3.3 The SKIZ

In a binary image, we can also compute the skeleton of the background (the complement) of the image. This skeleton is often called the *SKIZ*, the *skeleton by influence zones*, because it shows the influence zones (the collection of points closest to an object) of the foreground objects. An example of a SKIZ can be seen in figure 6.23.

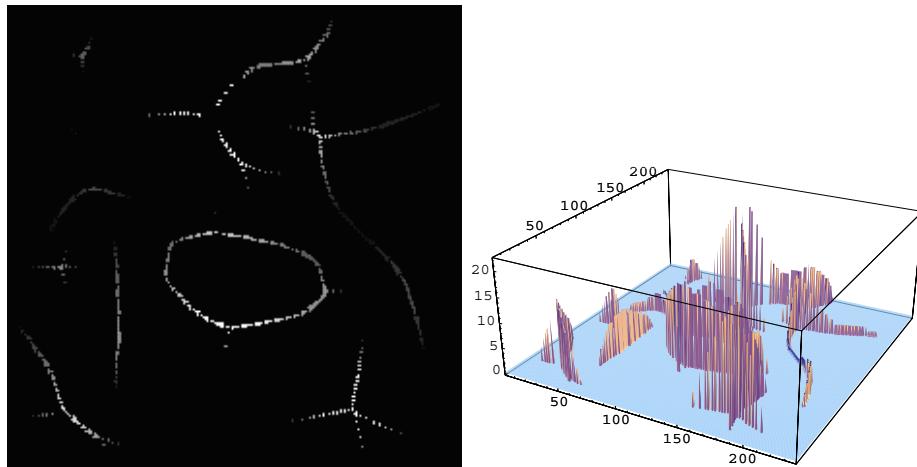


Figure 6.22 Example of a quench function obtained from the skeleton by maximal balls in the previous figure.

6.6.4 Hit-or-miss transformation

The hit-or-miss transformation is special in binary morphology because it uses a *composite* structuring element $X = (X_1, X_2)$. The hit-or-miss transformation of a set S is defined as all points x where X_1 fits in S , and X_2 fits in the background S^c .

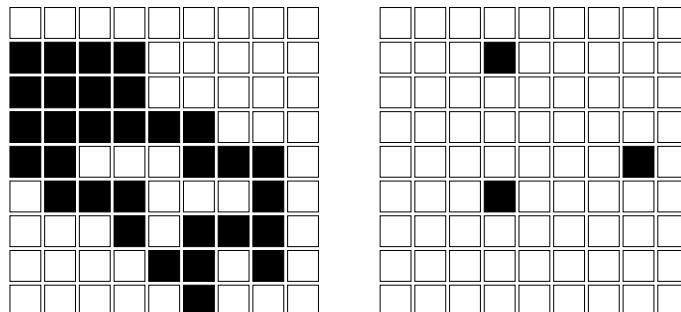
Example

Given a structuring element $X_1 = \begin{array}{|c|c|c|} \hline & & \\ \hline & \bullet & \\ \hline & & \\ \hline \end{array}$ (with the circle marking the origin) and

a structuring element $X_2 = \begin{array}{|c|c|c|} \hline & \bullet & \\ \hline & & \\ \hline & & \\ \hline \end{array}$, then the composite structuring element $X =$

$(X_1, X_2) = \begin{array}{|c|c|c|} \hline \bullet & & \\ \hline & \bullet & \\ \hline & & \\ \hline \end{array}$. If we do a hit-or-miss transformation with X on a set S , then we are looking for points where X_1 fits in an object, and X_2 fits in the background. So in this case, we are looking for a type of upper-right corner.

For example: a set and its hit-or-miss transformation:



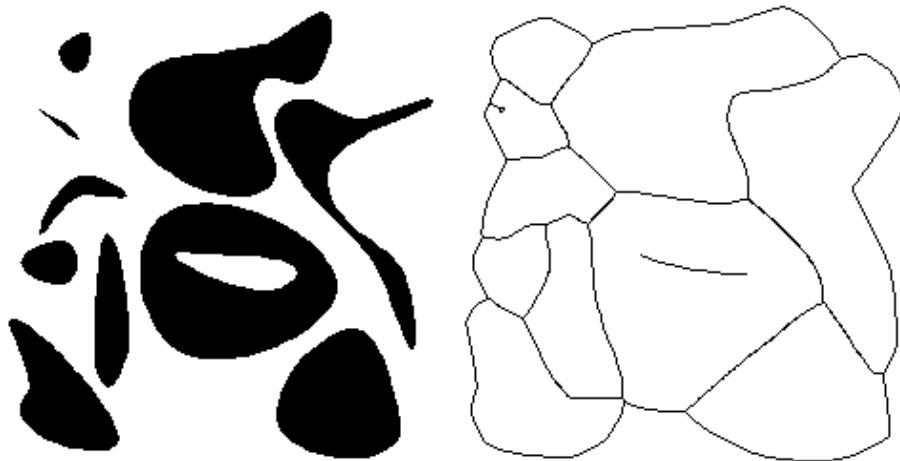


Figure 6.23 Example of a SKIZ.

Formally, we can define the hit-or-miss transformation H of a set S with a composite structuring element $X = (X_1, X_2)$ by

$$H_X(S) = \varepsilon_{X_1}(S) \cap \varepsilon_{X_2}(S^c).$$

6.6.5 Thinning and thickening

The *thinning* $\text{thin}_X(S)$ of a set S is the residue of S and its hit-or-miss transformation with the composite structuring element X :

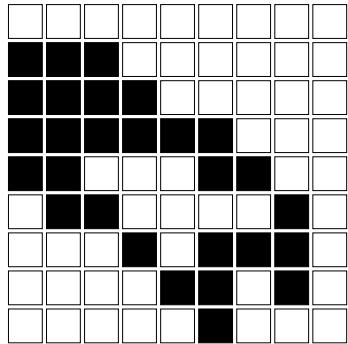
$$\text{thin}_X(S) = S - H_X(S).$$

Its dual counterpart is called *thickening* and is defined by the union of S and its hit-or-miss transformation:

$$\text{thick}_X(S) = S \cup H_X(S).$$

Example

The example in the previous section shows the hit-or-miss transformation $H_X(S)$ of a set S by X . The thinning $\text{thin}_X(S)$ equals

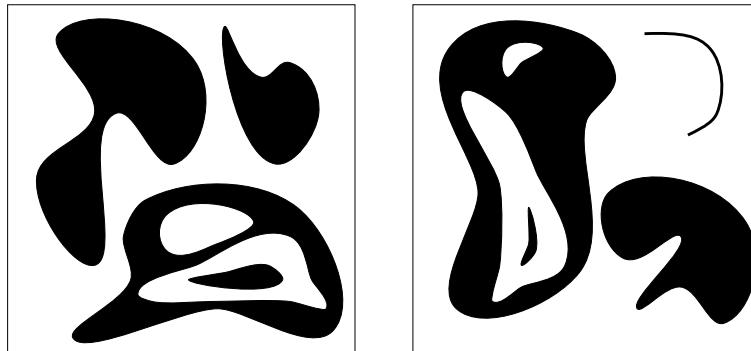


As all the elements of the hit-or-miss transformation lie inside S , the thickening $\text{thick}_X(S)$ equals S itself.

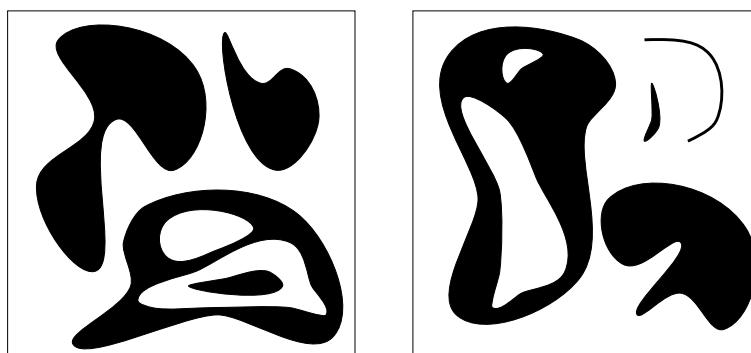
Thinning and thickening are useful operations since they allow the formulation of *homotopic* operations, *i.e.*, operations that preserve the homotopy of a set; that do not break object “connections”.

Intermezzo

Two sets are homotopic if they contain the same number of objects and each object contains the same number of holes and enclosed objects. For example: these two sets are homotopic:



And these two are *not*:



Another definition of homotopy is that the two sets can be continuously deformed to each other.

The structuring elements that are most useful can be found in the so-called *Golay alphabet*. Here, we will only discuss the *L* Golay element, which is most used. The *L* element is shown in figure 6.24. What is called the “thinning” with structuring element *L* is in

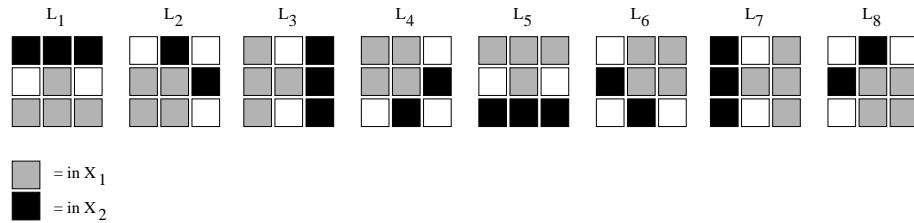


Figure 6.24 The *L* Golay element. Shown are the basic element L_1 and rotated version as they are used in thinning and thickening. Note that the rotations of *L* are not exact.

fact a sequence of thinnings with the rotated elements L_1, \dots, L_8 :

$$\text{thin}_L(S) = \text{thin}_{L_1}(\text{thin}_{L_2}(\dots(\text{thin}_{L_8}(S)))).$$

Iterated thinning of a set with *L* until the set no longer changes results in a skeleton of the set. An example of such a skeleton has already been shown in figure 6.21. The advantage of this skeleton is that it is homotopic with the original set. The skeleton by maximal balls does not have this property.

6.7 Applications

6.7.1 Alternating sequential filters

The ultimate erosion and skeleton are examples of structures that can be obtained by the residue of two families of operators. Another way of combining two families of operators $\{\varphi_i\}$ and $\{\psi_i\}$ is by creating an *alternating sequential filter* (ASF), which is a composition of operators such as:

$$\begin{aligned}\text{ASF1}_i &= \psi_i \varphi_i \dots \psi_2 \varphi_2 \psi_1 \varphi_1 \\ \text{ASF2}_i &= \varphi_i \psi_i \dots \varphi_2 \psi_2 \varphi_1 \psi_1 \\ \text{ASF3}_i &= \psi_i \varphi_i \psi_i \dots \psi_2 \varphi_2 \psi_2 \psi_1 \varphi_1 \psi_1 \\ \text{ASF4}_i &= \varphi_i \psi_i \varphi_i \dots \varphi_2 \psi_2 \varphi_2 \varphi_1 \psi_1 \varphi_1.\end{aligned}$$

In practice an ASF is only a useful operator if $\{\psi_i\}$ is decreasing as i rises, $\{\varphi_i\}$ is increasing as i rises, and $\psi_1(f) \leqslant \varphi_1(f)$, so if

$$\psi_i(f) \leqslant \dots \leqslant \psi_1(f) \leqslant \varphi_1(f) \leqslant \dots \leqslant \varphi_i(f).$$

Openings and closings are often used for the two families. ASF's are sometimes useful for dealing with very noisy images, as shown in figure 6.25.

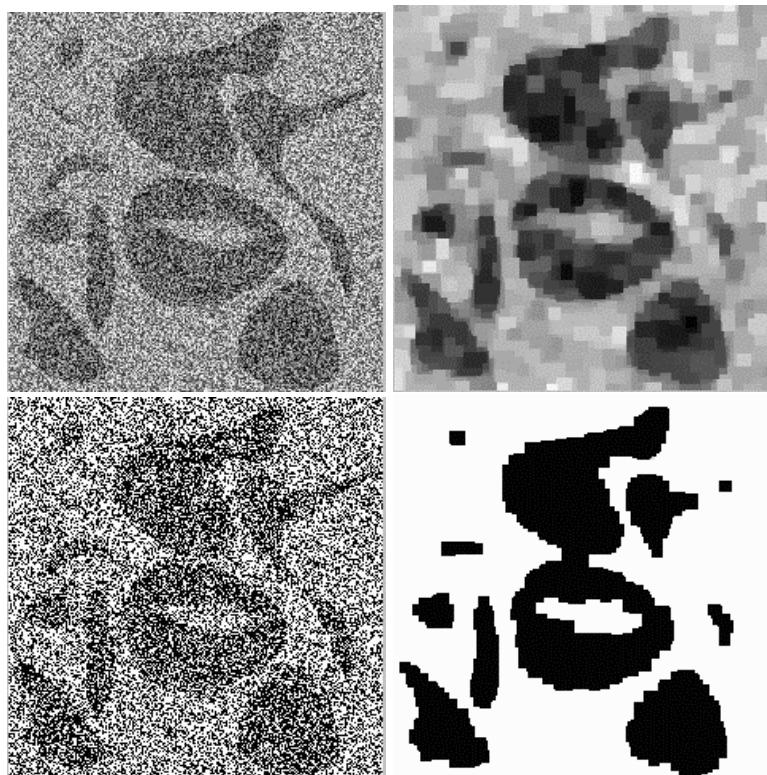


Figure 6.25 The top left shows a very noisy 256×256 image. In this case thresholding (bottom left) will not give us the objects that are vaguely visible in the original to the human observer. However, after application of an ASF using openings and closings with square structuring elements of sizes 3 to 7 (top right), thresholding does give us approximately the desired result (bottom right).

6.7.2 Granulometry

It is often useful to know the *size distribution* of an image, *i.e.*, *e.g.*, to know what fraction of an image is filled with objects or object parts of a certain size i . The study of the size distribution in an image is called *granulometry*. In previous sections, we saw

that an opening removes objects related to the size of the structuring element from an image, so it seems logical to use openings (and closings) of various sizes to study the size distribution of an image. A practical approach to measuring a size distribution of an image f is

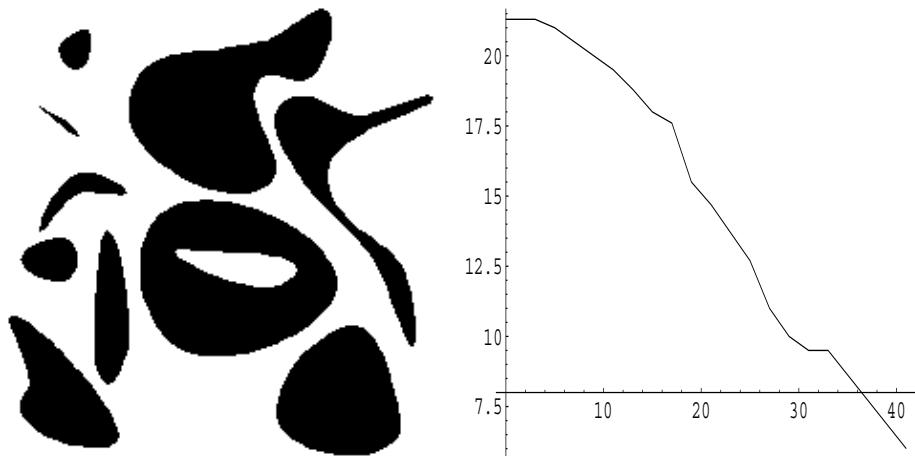
1. Open the image with a structuring element of a certain size.
2. Do a size measurement on the current image, e.g., the area of all object pixels, or the sum of all grey values.
3. Enlarge the structuring element.
4. Repeat from 1 until the opening removes the entire image.

This is in fact an example of a granulometry with the family of openings $\{\gamma_i\}$. In general, any decreasing⁸ family $\{\psi_i\}$ with the following property can be used for a granulometry: $\psi_i \psi_j(f) = \psi_{\max\{i,j\}}(f)$, i.e., applying the “strongest” ψ has the same effect as applying both ψ_i and ψ_j .

The size measurements together make up a granulometric curve.

Example

An example of a granulometric curve of an image is:



The x axis shows the size of the opening by a square structuring element. The y axis shows the number of pixels ($\cdot 10^3$) contained in the opening. The original image contains 57121 (239×239) pixels.

The granulometric curve obtained with granulometry by opening is often extended using closings, as figure 6.26 shows.

⁸ $\psi_i(f) \leq \psi_j(f)$ if $i > j$.

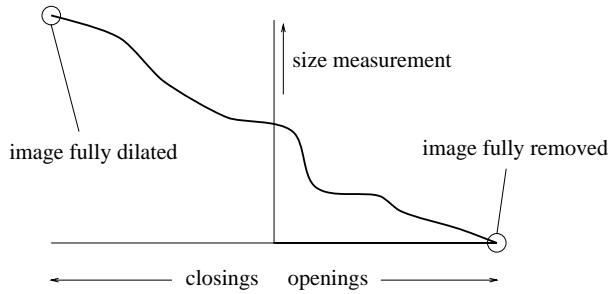


Figure 6.26 Extension of a granulometric curve with closings. The closing measurements are shown on the left side of the origin, moving to the left increases the closing structuring element size. On the right of the origin are the opening measurements. Moving to the right increases the opening structuring element size.

6.7.3 Toggle mapping

Toggle mapping is a morphological technique to enhance contrast in images. Given an anti-extensive transformation ψ_1 (e.g., erosion) and an extensive transformation ψ_2 (e.g., dilation) of an image f , the toggle map f_{TM} is defined by

$$f_{\text{TM}} = \begin{cases} \psi_1(f) & \text{if } f - \psi_1(f) < \psi_2(f) - f \\ \psi_2(f) & \text{otherwise.} \end{cases}$$

In other words, f is replaced by the transformation value $\psi_i(f)$ that is closest to f .

A type of toggle mapping that is also known as *morphological deblurring* uses erosion for ψ_1 and dilation for ψ_2 . So in this operation each image value of f is either replaced by the eroded value or the dilated value of f , whichever one is closest to the original value of f . The figures 6.27 and 6.28 show examples of deblurring applied to real images.

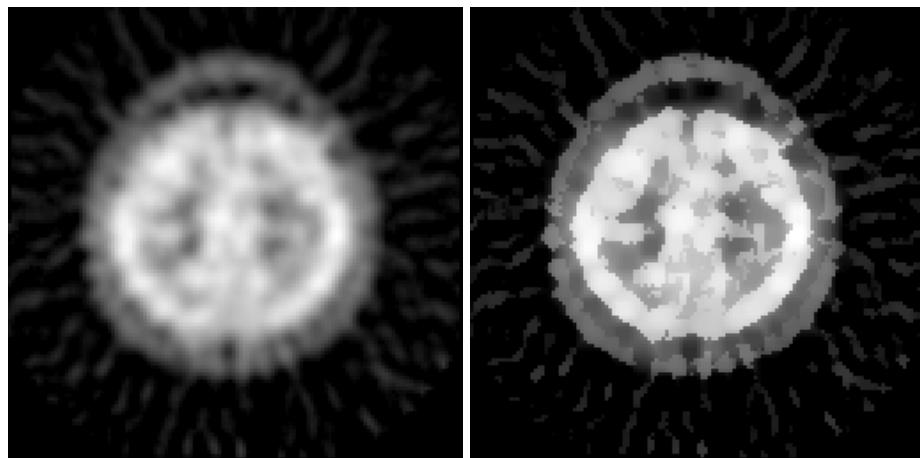


Figure 6.27 Example of morphological deblurring using a 3×3 square structuring element. The original image has a resolution of 128×128 .

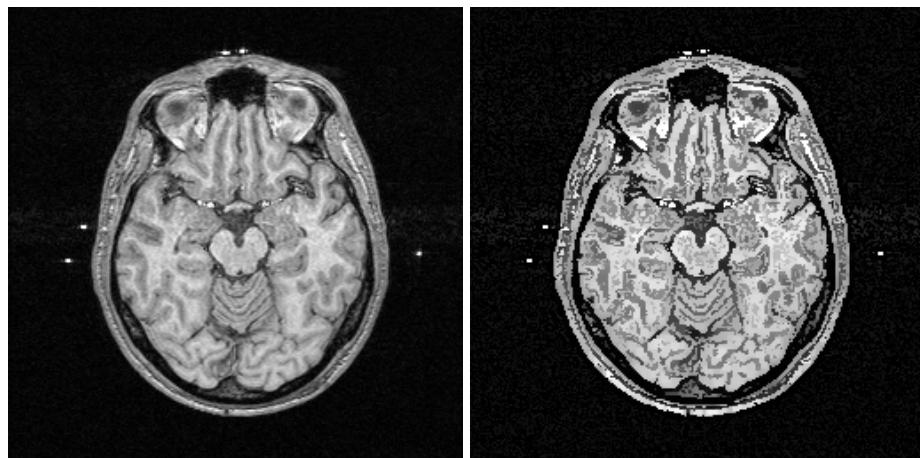


Figure 6.28 Example of morphological deblurring using a 5×5 square structuring element. The original image has a resolution of 256×256 .

Chapter 7

The Fourier transform

In chapter 3 we introduced the concept of *image frequencies*. We used images of sinusoids to show the relation between the frequency of the sinusoids and image detail: a high frequency corresponded to a high level of detail, and a low frequency to a low level of detail. This seems natural; the grey values along a line in an image will cycle between black and white. If we encounter an area with many details, then the cycling must be done fast (high frequency), while relatively smooth image areas require only slow cycling (low frequency).

In this chapter we will investigate further the relation between sinusoids of different frequencies and digital images.

7.1 The relation between digital images and sinusoids

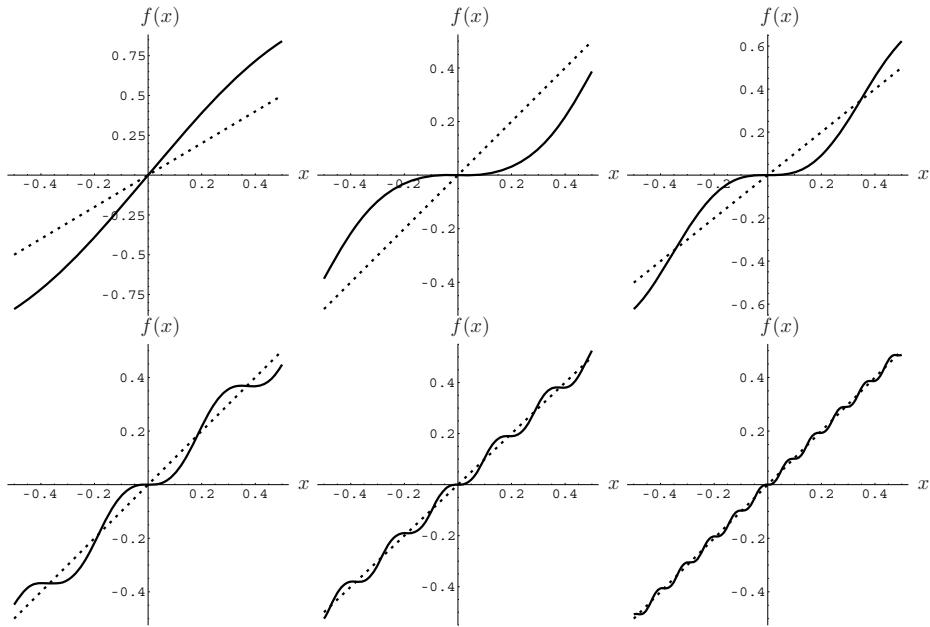
In chapter 3, we compared image frequency to the frequency of sinusoids. But how can we determine the “frequency” of the grey values along a line in an image if the graph of these grey values looks nothing like a sinusoid? The answer can be gleaned from this statement: *any digital image can be modeled as a sum of sinusoids*. More general, any function f for which f and f' are piecewise continuous can be modeled as a sum of sinusoids.

Example

The function $f(x) = x$ on the domain $[-\frac{\pi}{2}, \frac{\pi}{2}]$ can be written as

$$\begin{aligned} f(x) &= \sin(2x) - \frac{\sin(4x)}{2} + \frac{\sin(6x)}{3} - \frac{\sin(8x)}{4} + \frac{\sin(10x)}{5} - \frac{\sin(12x)}{6} + \dots \\ &= \sum_{j=1}^{\infty} \frac{1}{j} (-1)^{j+1} \sin(2jx). \end{aligned}$$

To show the validity of this we show the graph of $f(x) = x$ (dashed line) together with a graph of the first k terms of the series of sines. For $k \in \{1, 2, 4, 8, 16, 32\}$ this gives:



The series converges to the correct function.

Example

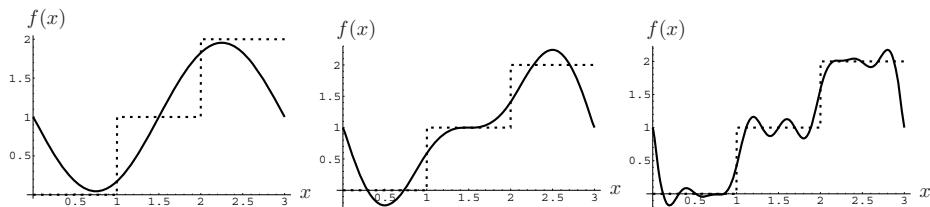
We can also use sinusoids to model functions that seem completely alien to sines, e.g., the step function

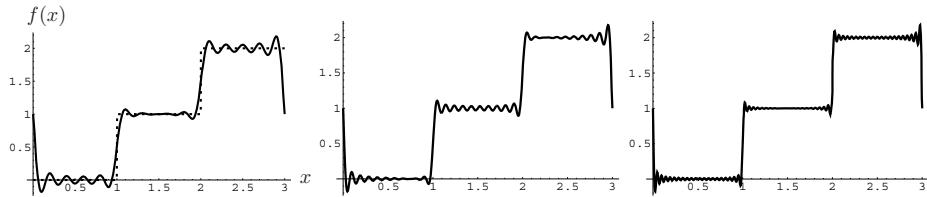
$$f(x) = \begin{cases} 0 & \text{if } 0 \leq x \leq 1 \\ 1 & \text{if } 1 < x \leq 2 \\ 2 & \text{if } 2 < x \leq 3, \end{cases}$$

which can be modeled by

$$f(x) = 1 - \frac{3 \sin(\frac{2\pi x}{3})}{\pi} - \frac{3 \sin(\frac{4\pi x}{3})}{2\pi} - \frac{3 \sin(\frac{8\pi x}{3})}{4\pi} - \frac{3 \sin(\frac{10\pi x}{3})}{5\pi} - \frac{3 \sin(\frac{14\pi x}{3})}{7\pi} + \dots$$

as these graphs show for 2, 4, 8, 16, 32, and 64 terms of this series:





7.1.1 The Fourier transform

Given a function $f(x)$, how can we determine how often each frequency¹ occurs in it? And how do these frequencies relate to $f(x)$? The answer to these questions can be found using the *Fourier transform* of $f(x)$, denoted by $\mathcal{F}[f](u)$:

$$F(u) = \mathcal{F}[f](u) = \int_{-\infty}^{\infty} f(x)e^{-i2\pi ux}dx,$$

where i is the complex number defined by $i^2 = -1$.

The Fourier transform \mathcal{F} transforms the function $f(x)$ into a new function $F(u)$ of a new variable u . *The value of $F(u)$ gives information on the spatial frequency u in the function f .* Note that $F(u)$ may have complex values. Typically, we are interested in ‘how much’ frequency u is present in the image f : this can be measured by the magnitude of the Fourier transform at frequency u , i.e. $|F(u)|$. Another often used measure is the magnitude squared; the *power spectrum* $|F(u)|^2$.

Fourier transforming a function is an invertible operation; the original function can be computed from the Fourier transform by the *inverse* (or *backward*) Fourier transformation \mathcal{F}^{-1} :

$$f(x) = \mathcal{F}^{-1}[F](x) = \int_{-\infty}^{\infty} F(u)e^{i2\pi ux}du.$$

Intermezzo

There are many different definitions of the Fourier transform and its inverse. The definitions most commonly used in the physical sciences, mathematics, and electrical engineering all differ slightly. The basic behavior of the transform is the same in all cases, though. Common differences are:

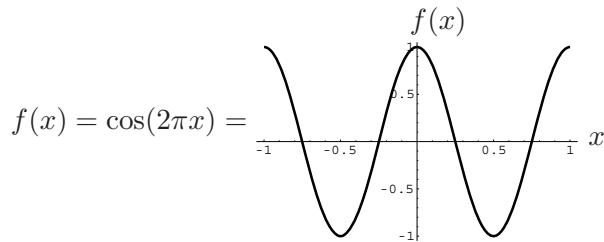
¹Where frequency is defined as the number of periods (cycles) of a sinusoid per unit length, i.e., $\sin(2\pi x)$ has frequency 1, $\sin(x)$ has frequency $\frac{1}{2\pi}$, etc.

- Swapping the minus sign in the exponent between the forward and the backward transformation.
 - Omitting the factor 2π in the exponent.
 - Including a global factor $\frac{1}{\sqrt{2\pi}}$.
-

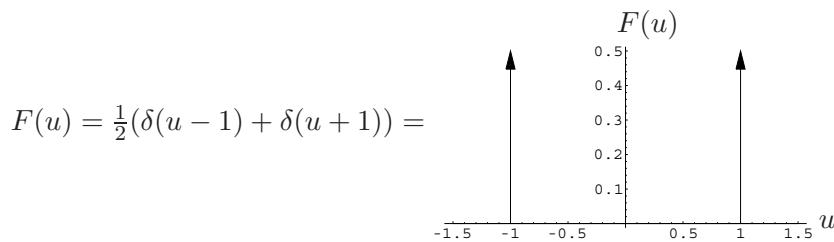
Example

Explicit computation of the Fourier transform of a given analytic function $f(x)$ is often a difficult task. In this example we do not show the mathematics used to compute the transforms. If you can't work it out by yourself, this is not important. Such mathematical skills are not necessary to understand and use the Fourier transform in practice.

Given the function



what do we expect for the Fourier transform $F(u)$? Since f contains only one frequency (frequency one), we expect $F(u)$ to have some value if u equals 1, and be zero everywhere else. This is *almost* what happens; the Fourier transform equals

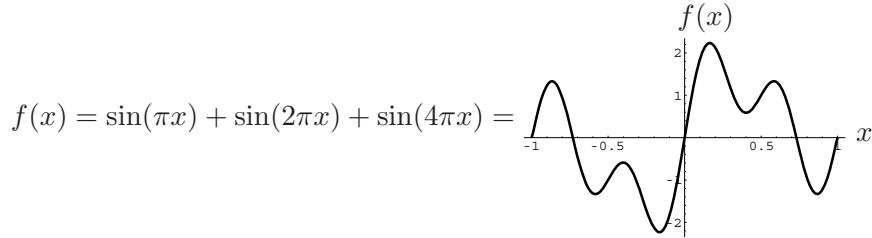


where δ is the delta function defined in chapter 3. $F(u)$ not only has a non-zero value at the expected frequency (1), but also at the negative frequency mirror (-1). This phenomenon is common to the Fourier transform.²

²Without going into the mathematical details, a hint to explain the occurrence of negative frequency mirrors is the fact that the function $\cos(2\pi x)$ equals the function $\cos(-2\pi x)$.

Example

The function



is a sum of three distinct sinusoids, so we expect the Fourier transform to be zero everywhere, except at the three occurring frequencies 0.5, 1, and 2 (and their negative mirrors -0.5 , -1 , and -2). This is indeed what happens, since the Fourier Transform equals

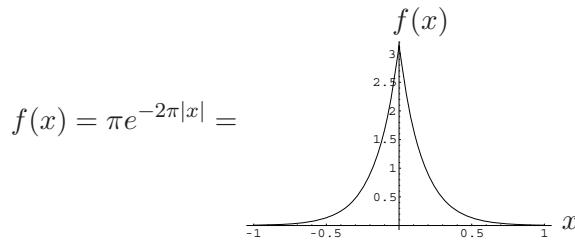
$$\begin{aligned} F(u) &= -\frac{1}{2}i \left(\delta(u - \frac{1}{2}) + \delta(u - 1) + \delta(u - 2) + \right. \\ &\quad \left. -\delta(u + \frac{1}{2}) - \delta(u + 1) - \delta(u + 2) \right) \\ &= \begin{array}{c} I(F(u)) \\ \hline -2 & -1 & 0 & 1 & 2 \\ \uparrow & \uparrow & \uparrow & \downarrow & \downarrow \\ 0.4 & 0.2 & 0 & -0.2 & -0.4 \end{array} \end{aligned}$$

Note that the graph shows the imaginary part $I(F(u))$ of the complex value $F(u)$. In this example, $F(u)$ is a pure imaginary number; its real component is zero. In general, odd functions (like the sine function) have an imaginary Fourier transform, and even functions (like the cosine in the previous example) have a real Fourier transform. In practice, signals and images are seldom so well-behaved as to be even or odd, so we may expect their Fourier transforms to be all over the complex plane.

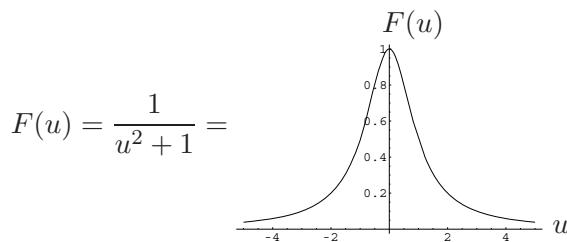
Example

In the examples above, the Fourier transforms had a “spike” character (consisted only of delta functions), because the functions $f(x)$ contained only distinct frequency components. In this example we choose for f a function that has no such distinct components.

Given



then the Fourier transform equals



Since our primary interest is two-dimensional functions (images), we introduce the two-dimensional Fourier transform $F(u, v)$ of $f(x, y)$:

$$F(u, v) = \mathcal{F}[f](u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) e^{-i2\pi(ux+vy)} dx dy,$$

and the inverse transform:

$$f(x, y) = \mathcal{F}^{-1}[F](x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} F(u, v) e^{i2\pi(ux+vy)} du dv.$$

The variable u in the Fourier transform now corresponds to frequencies of sinusoids oriented along the x direction. v Corresponds to frequencies in the y direction, in the following manner:

- $F(u, 0)$ is related to the frequency u of sinusoids occurring parallel to the x axis of the image.
- $F(0, v)$ is related to the frequency v of sinusoids occurring parallel to the y axis of the image.

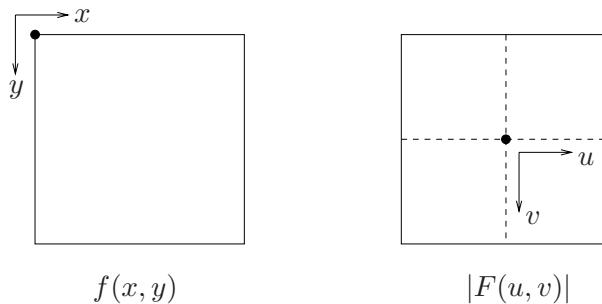
- $F(u, v)$ is related to the frequency $\sqrt{u^2 + v^2}$ of sinusoids parallel to a line from the origin to (u, v) .

Below are some examples of Fourier transforms of images, clarifying these concepts. But first we need some conventions on displaying transforms of images.

Intermezzo: display of the two-dimensional Fourier transform

To avoid problems with displaying the complex-valued transform $F(u, v)$ of an image $f(x, y)$, a common approach is to display only the *magnitude* $|F(u, v)|$ (the modulus value), and ignore any phase information of $F(u, v)$. This makes sense if we are only interested in a picture showing ‘how much’ each frequency occurs in an image f .

It is also common to display $|F(u, v)|$ for negative values of u and v . For this, the origin of the image is shifted to the image center.



Typically, the values of $|F(u, v)|$ are such that direct display of these values results in dark, low-contrast images. To enhance contrast, $\log(|F(u, v)| + 1)$ (or a similar remapping of $|F(u, v)|$) is often displayed instead.

Figure 7.1 shows examples of Fourier magnitude images of images containing only sinusoids. Note that

- Since the images contain only distinct frequencies, the Fourier images contain only ‘dots’, i.e., have non-zero values only at specific locations.
- Since the coordinates of the dots equal the frequencies of the sinusoids occurring in the original image, higher image frequencies cause the dots to move away from the origin.
- The dots are oriented from the origin in the same direction as the frequency ‘wave’ in the original image. If we rotate the image (e.g., consider the first and second rows), the Fourier transform rotates with it.

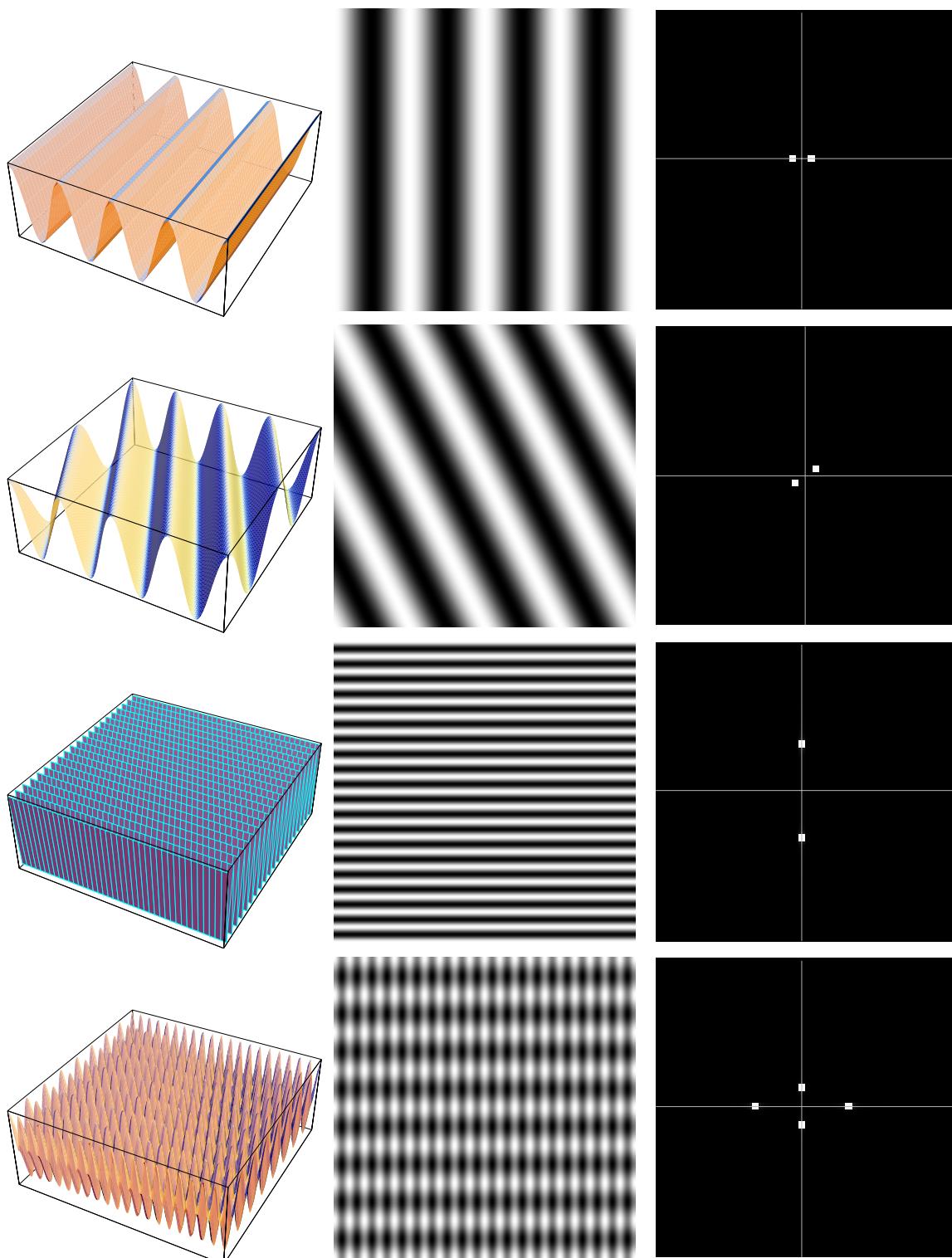


Figure 7.1 Examples of Fourier magnitude images (right column) of images containing only sinusoids (left and middle column). Axes have been added for clarity. See text for details.

rsity in the Netl
medical image reg
: to know more,

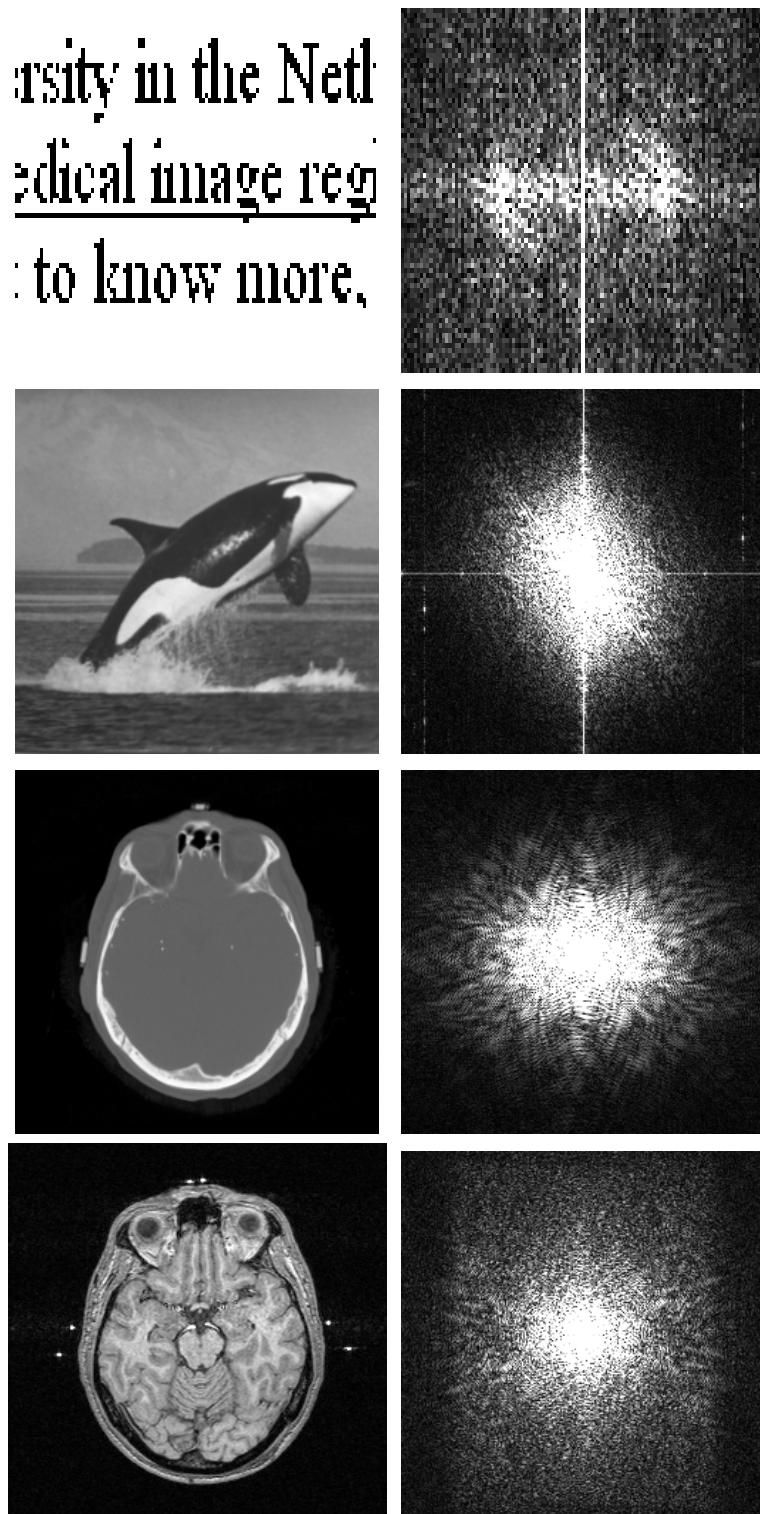


Figure 7.2 Examples of Fourier magnitude images (right column) of real images (left column). The top example is of a binary image, the other images are grey-valued. See text for details.

- The dots are oriented in the direction of the waves, so *perpendicular* to the edges occurring in the original image.

Figure 7.2 shows some examples of Fourier magnitude images of real images. In the second example, we see an oriented cluster in the Fourier transform perpendicular to the orientation of the orca, *i.e.*, consistent with the sinusoids examples, where Fourier ‘dots’ are directed perpendicular to edges in the image.

7.1.2 Fourier series

At the beginning of section 7.1 we showed some examples of how functions could be written as a series of sinusoids of distinct frequency. Such a series is called a *Fourier series* of a function f . Although it is closely related to the Fourier transform, the series and the transform are essentially different concepts:

- The Fourier series writes the function f as a sum of sinusoids, each with a distinct frequency. If the series converges, its value equals the value of f .
- The Fourier transform transforms $f(x)$ into a new function F of a new variable u . The values of $F(u)$ give information on the frequency components of f at frequency u .

7.2 Image processing in the frequency domain

The frequency domain, (*i.e.*, the domain of the variables u and v of the Fourier transform $F(u, v)$ of an image $f(x, y)$), allows access to a new range of image processing operations directly modifying the image frequency characteristics, something which is usually much harder to achieve by spatial domain processing. Moreover, the frequency domain displays many interesting mathematical properties that cause that even image processing tasks that are not primarily related to image frequencies may be carried out advantageously in the frequency domain.

In this section, we will explore some typical frequency domain processing tasks, using the Fourier transform as defined in section 7.1.1.

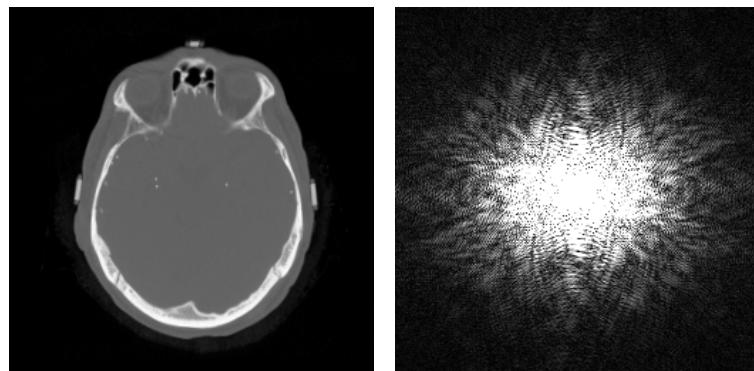
A common frequency domain processing technique is the removal of frequencies within a certain range from an image. This technique

- computes the Fourier transform $F(u, v)$ of an image $f(x, y)$
- computes a new transform $F'(u, v)$ by setting specific values of $F(u, v)$ to zero

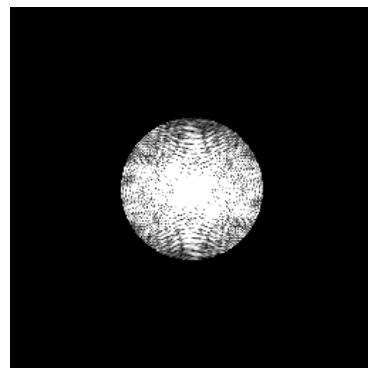
- computes the inverse Fourier transform $f'(x, y)$ of $F'(u, v)$, which equals $f(x, y)$ without the frequencies that were set to zero.

Example

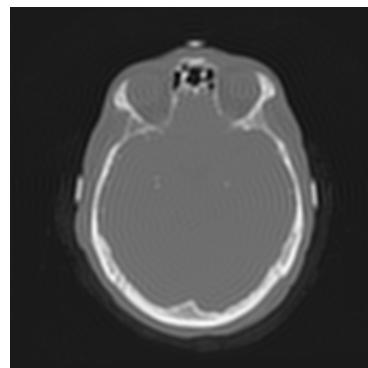
Given this image and its Fourier magnitude image:



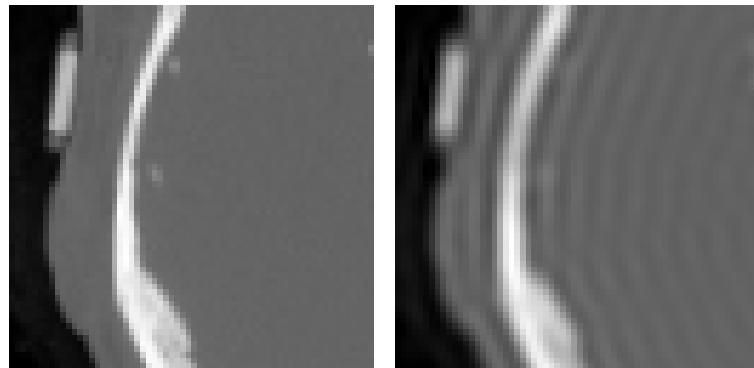
Suppose we want to remove high frequencies from this image (*i.e.*, create a low-pass filter): then we can set all frequency values $F(u, v)$ to zero if they are farther than a certain distance from $(0, 0)$, and the Fourier magnitude image becomes:



If we transform this image back to the spatial domain, the resulting image is:



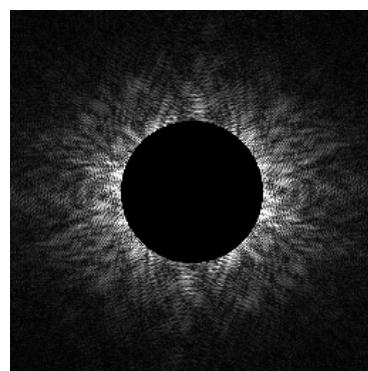
By removing the high frequencies, small details have vanished from the image, as is clear from these blow-ups of the original and low-pass filtered image:



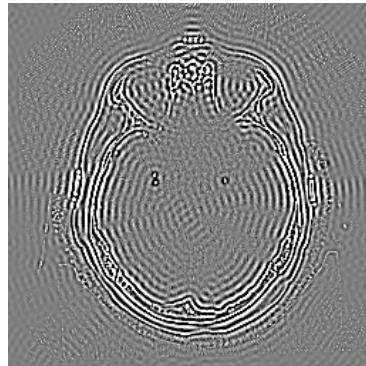
If we compare the low-pass filter from the example above to the one defined by convolution in chapter 5, it will be clear that this new filter is much more flexible; we now have more complete control as to which frequencies are to be removed and retained by the filter.

Example

We can create a high-pass filter by setting all values of $F(u, v)$ at frequencies (u, v) closer than a certain distance to $(0, 0)$ to zero. Using the image from the previous example, the Fourier magnitude image then looks like:



the inverse transform (high-pass filtered image) image then is:



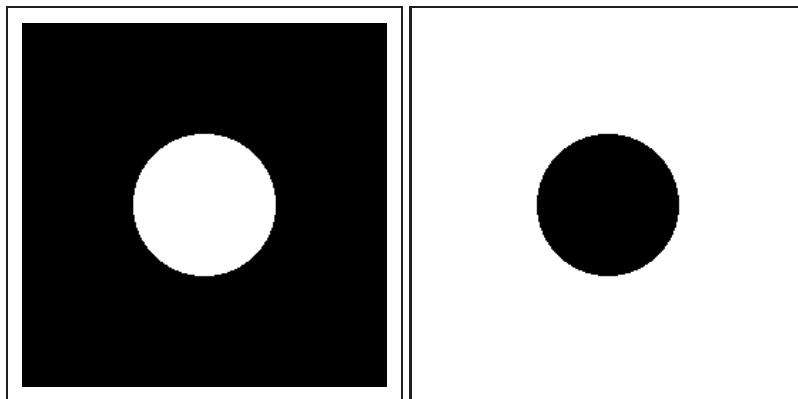
In the examples above, the result image $f_r(x, y)$ is obtained by computing

$$f_r(x, y) = \mathcal{F}^{-1}[F(u, v)M(u, v)],$$

where $F(u, v)$ is the Fourier transform of $f(x, y)$ and $M(u, v)$ is a *mask* image. The mask image is an image that has value 0 at pixel locations (u, v) where $F(u, v)$ is to be set to zero, and value 1 at pixel locations where $F(u, v)$ is not to be altered. Multiplying F with the mask image M is called *masking* F .

Example

In the last two examples, the mask images are respectively:



where black represents a zero value and white represents one.

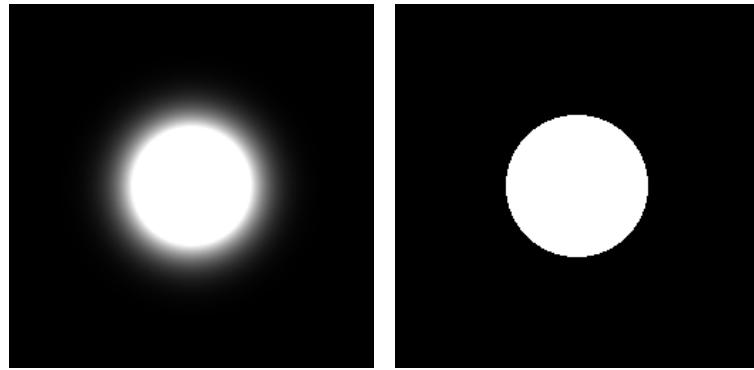
A more sophisticated filter can be constructed if we allow more smooth transitions in the mask image M than just “step” transitions from 0 to 1 and vice versa.

Example

A new low-pass filter can be constructed by choosing for the mask image M a function G_σ , with

$$M(u, v) = G_\sigma(u, v) = e^{-2\pi^2\sigma^2(u^2+v^2)}.$$

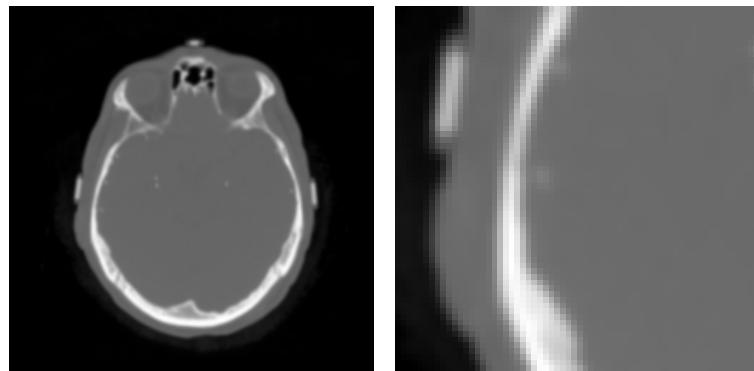
Compare this mask image for a certain choice of σ with the binary mask image used before:



Applying this low-pass filter to our test image, *i.e.*, computing

$$f_r(x, y) = \mathcal{F}^{-1}[F(u, v)G_\sigma(u, v)],$$

the resulting image (together with a blow-up) is:



By changing the parameter σ we can let more (lower σ) or less (higher σ) frequencies pass.

By using smooth mask images, less artifacts in the result images are produced than when using a mask image that has sharp edges. Especially “ringing” artifacts around image objects (see blow-ups in the examples) may be avoided in this way.

7.2.1 Filter shapes

In the previous section, we computed

$$f_r(x, y) = \mathcal{F}^{-1}[F(u, v)M(u, v)],$$

where the function M was used to select which frequencies of the original image f end up in the ‘filtered’ result f_r . One choice of M was a binary-valued function, often called a *rectangle*, *boxcar* or *block filter*, because of its shape when plotted (see figure 7.3). The frequency where M jumps from 1 to 0 is called the *cutoff frequency*.

A second choice for M was an exponential function G (see figure 7.3, which is called a *Gaussian filter* because G is the Fourier transform of a Gaussian function. A third often

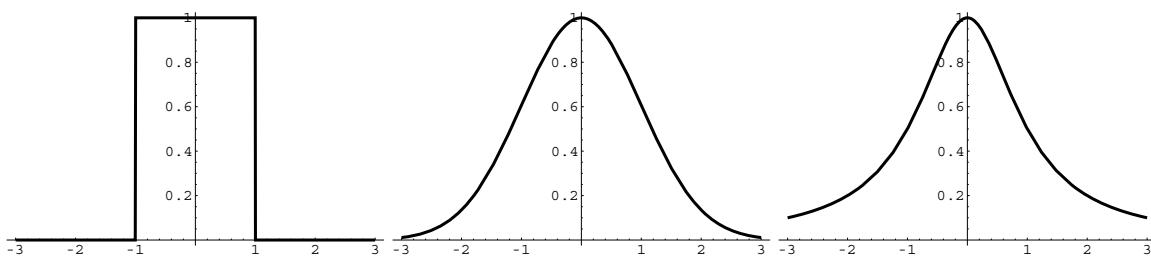


Figure 7.3 One-dimensional examples of frequency domain filter shapes. Left: block filter, Middle: Gaussian filter, Right: Butterworth filter.

used filter shape is the *Butterworth filter* B (see figure 7.3), the simplest version of which equals

$$B(u) = \left(1 + \left(\frac{u}{u_0} \right)^{2n} \right)^{-1},$$

or in two dimensions

$$B(u, v) = \left(\left(1 + \left(\frac{u}{u_0} \right)^{2n} \right) \left(1 + \left(\frac{v}{v_0} \right)^{2n} \right) \right)^{-1},$$

where n is called the order of the function, and the parameters u_0 and v_0 determine the width of the filter. The value $B(u)$ is always 1 at $u = 0$ (for positive n) and the value is $\frac{1}{2}$ at $u = u_0$. The order n controls the steepness of the filter. Positive values of n shape the filter into a low-pass form, and negative values into a high-pass form. Figure 7.4 shows some examples of the Butterworth filter for different values of n .

Many alternative filters exist besides the mentioned block, Gaussian and Butterworth filters, each with specific properties. Much used filters are the Bartlett filter (linear / triangular shape), the Welch filter (parabolic shape), the Chebyshev filter, and the Blackman, Hamming and Hanning filters (with shapes derived from cosines).

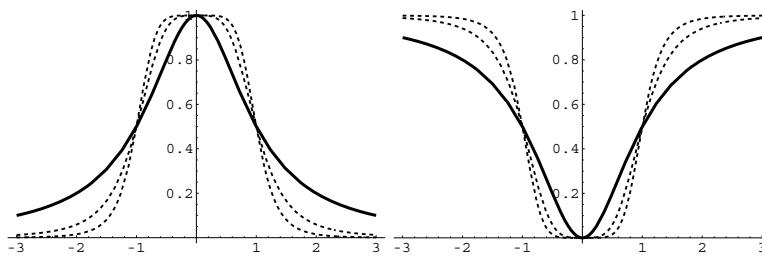


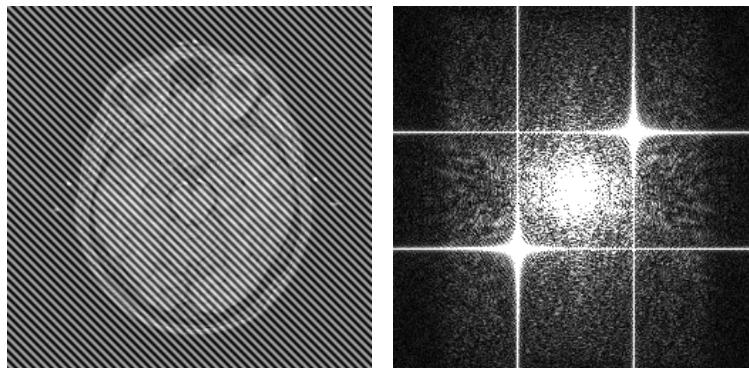
Figure 7.4 Example of the Butterworth filter with $u_0 = 1$. On the left, the filter for orders $n = 1$ (solid line), and $n = 2$ and $n = 3$ (dashed lines) can be seen. On the right, the same plots for $n = -1$ (solid line) and $n = -2$ and $n = -3$ (dashed lines).

7.2.2 Removing periodic noise

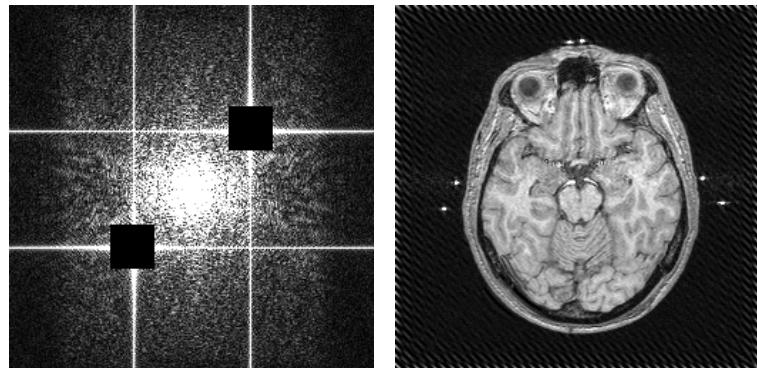
Many imaging devices display a characteristic artifact known as *periodic noise*, where the ideal images are distorted by the addition of a periodic wave pattern to the image. If we can establish the frequency of this wave pattern, we can often remove much of the artifact from the image by masking of the Fourier transform of the image.

Example

Given an image with periodic noise and its Fourier transform magnitude image:



we suspect the two bright spots in the Fourier image to be caused by the periodic noise. The orientation of the spots also corresponds with the orientation of the wave pattern in the original image. By masking the Fourier transform such that the value of $F(u, v)$ in the area around the spots is set to zero, and transforming the result back to the spatial domain, we can remove most of the periodic noise artifact:



7.3 Properties of the Fourier transform

The Fourier transform has many special mathematical properties. A few are especially of interest in practical image processing, and will be covered in the next sections.

7.3.1 Separability

In the formula of the two-dimensional Fourier transform of a function $f(x, y)$, we can separate the two-dimensional integral into two one-dimensional parts:

$$F(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) e^{-i2\pi(ux+vy)} dx dy = \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} f(x, y) e^{-i2\pi ux} dx \right] e^{-i2\pi vy} dy.$$

We can therefore compute $F(u, v)$ by computing two one-dimensional integrals; namely first the integral between the square brackets, integrating to x , second the outer integral, integrating to y . This property is useful for computing the Fourier transform of an image, because we do not need to ‘traverse’ the entire two-dimensional image for each computation of a transform value. Instead, we can use the value of the integral along each image line (the integral between brackets), store them as intermediate results, and then compute the integral along each image column (outer integral).

7.3.2 Linearity

Given two images $f(x, y)$ and $g(x, y)$ and their respective Fourier transforms $F(u, v)$ and $G(u, v)$, then the following linearity relationship holds:

$$\mathcal{F}[f(x, y) + g(x, y)] = F(u, v) + G(u, v),$$

and, more general,

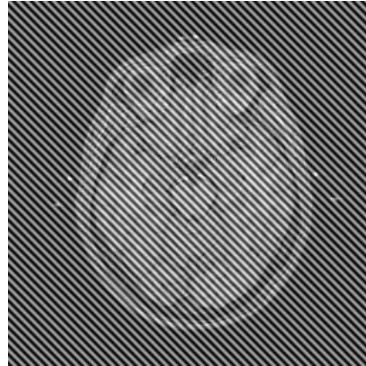
$$\mathcal{F}[af(x, y) + bg(x, y)] = aF(u, v) + bG(u, v),$$

where a and b are arbitrary constants. In words, this relationship says that the Fourier transform of a sum equals the sum of the Fourier transforms.

This property can, e.g., be useful when we know an image to be the sum of an ‘ideal’ image and a distorting image (noise), and we are able to estimate the Fourier transform of the noise image.

Example

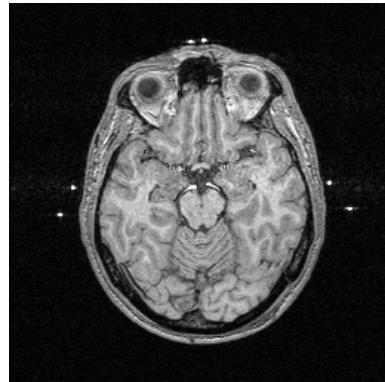
Consider the image distorted with periodic noise from section 7.2.2:



We assume this image g to be the sum of an ideal image f and a noise image η : $g(x, y) = f(x, y) + \eta(x, y)$. We define the Fourier transforms of g , f , and η by G , F , and H respectively. We further assume that η contains only a single frequency (which seems to be the case when we observe the image g), that can be modeled by a sinusoid. Using measurements from the image background, we can estimate the orientation, frequency, and amplitude of η . With this information, we can estimate $H(u, v)$. Since $G = F + H$, we can now approximate the ideal image f by subtracting H from G , and then computing the inverse Fourier transform:

$$f = \mathcal{F}^{-1}[F] = \mathcal{F}^{-1}[G - H].$$

This results in effective removal of the periodic noise; the result looks like:



7.3.3 Convolution property

In previous chapters it will have become clear that *convolution* is a very important concept in image processing. Not only can image acquisition nearly always be described as a convolution; many image processing operations can also be formulated as convolutions. But, convolution is a mathematical operation of some complexity, and practical implementation may suffer from this. Fortunately, the Fourier transform has a property that greatly reduces the complexity of working with convolutions. Given functions (or images) f and g and their respective Fourier transforms F and G :

$$\mathcal{F}[f * g] = FG.$$

This means that the difficult operation of convolution in the spatial domain is equivalent to the much simpler operation of *multiplication* in the frequency domain. We can therefore compute the convolution $f * g$ using only Fourier transforms and multiplication:

$$f * g = \mathcal{F}^{-1}[\mathcal{F}[f * g]] = \mathcal{F}^{-1}[FG],$$

so computation of $f * g$ can be carried out by:

- Computing the Fourier transforms F and G of f and g respectively.
- Computing the multiplication FG .
- Computing the inverse Fourier transform $\mathcal{F}^{-1}[FG] = f * g$.

Since efficient and fast programs to compute Fourier transforms of images exist and are widely available, using Fourier transforms to compute convolutions is by far the most common technique used.

Example

Suppose we wish to compute the convolution $g * f$ of an image f with a Gaussian kernel $g(x, y) = \frac{1}{\sigma^2 2\pi} e^{-\frac{1}{2} \frac{x^2+y^2}{\sigma^2}}$.

The Fourier transform of g equals $G(u, v) = e^{-2\pi^2\sigma^2(u^2+v^2)}$. Assuming that we can compute the Fourier transform $F(u, v)$ of the image f using standard software, we can compute the convolution result by evaluating $\mathcal{F}^{-1}[F(u, v)e^{-2\pi^2\sigma^2(u^2+v^2)}]$.

7.3.4 Derivative property

The Fourier transform can also be used to compute derivatives of functions: given a signal $f(x)$, the following relation holds:

$$\mathcal{F}[f'] = 2\pi i u \mathcal{F}[f],$$

where f' is the derivative of f . As with convolution, the computation of derivatives collapses to a simple multiplication in the Fourier domain. The relation can be extended to computing partial derivatives of higher-dimensional functions; given a function $f(x, y)$ and its Fourier transform $F(u, v)$:

$$\begin{aligned}\mathcal{F}\left[\frac{\partial f}{\partial x}\right] &= 2\pi i u F(u, v) \\ \mathcal{F}\left[\frac{\partial f}{\partial y}\right] &= 2\pi i v F(u, v) \\ \mathcal{F}\left[\frac{\partial^2 f}{\partial x \partial y}\right] &= -4\pi^2 u v F(u, v) \\ &\text{etc.}\end{aligned}$$

Theoretically, these relations can be used to compute derivatives of images using only Fourier transforms and multiplications. In practice though, the results are not often very useful (we will explain this in chapter 9). A slight modification of this method however –which combines it with a well-chosen low-pass filter– suddenly makes it an effective tool for computing image derivatives. This extended method will be treated in detail in chapter 9.

7.3.5 Other properties

The Fourier transform has many more interesting properties than the ones listed above. Some are listed in table 7.1. Table 7.2 shows some much used Fourier transforms.

name	function	Fourier transform
linearity	$af(x, y) + bg(x, y)$	$aF(u, v) + bG(u, v)$
translation	$f(x + a, y + b)$	$F(u, v)e^{i2\pi(au+bv)}$
rotation	$f(\mathbf{A}(x, y))$	$F(\mathbf{A}(u, v))$
scaling	$f(ax, by)$	$\frac{1}{ ab }F\left(\frac{u}{a}, \frac{v}{b}\right)$
convolution	$f(x, y) * g(x, y)$	$F(u, v)G(u, v)$
multiplication	$f(x, y)g(x, y)$	$F(u, v) * G(u, v)$
correlation	$f(x, y) \star g(x, y)$	$F(-u, -v)G(u, v)$
separability	$f(x)g(y)$	$F(u)G(v)$
derivative	$f'(x)$	$2\pi i u F(u)$

Table 7.1 Fourier transform properties. Upper case letters denote Fourier transforms of the corresponding lower case letter functions.

7.4 Inverse filtering

In chapter 3 we described image acquisition by a convolution process:

$$g(x, y) = h(x, y) * f(x, y),$$

where g is the acquired image, f is the ‘ideal image’, and h is the point spread function (PSF) of the imaging device. Fourier transforming this expression gives us

$$G(u, v) = H(u, v)F(u, v).$$

This means that if $H \neq 0$, we can reconstruct our ideal image f from F :

$$F(u, v) = \frac{G(u, v)}{H(u, v)}.$$

All we need is the PSF h (or a good estimate) of the imaging device, so we can compute H , then F , and finally f . Reconstructing the ideal image f in this way is called *inverse filtering*, where $\frac{1}{H}$ is the inverse filter of the convolution filter H . As we are in fact reversing the convolution process, this process is also called *deconvolution*.

Are we now able to remove all the blur induced by the imaging device? Can we sharpen out-of-focus photographs? Unfortunately, even though we are sometimes able to enhance images to some extent, in most cases the answer to these questions is no. The reason for this is that we demanded that H not be zero, which is not a reasonable demand in practice. Values of H that are zero or close to zero cause the inverse filter to ‘blow up,’ disrupting the reconstruction.

name	function	Fourier transform
Sinusoids	$\cos(2\pi ax)$	$\frac{1}{2}(\delta(u-a) + \delta(u+a))$
	$\sin(2\pi ax)$	$-\frac{1}{2}i(\delta(u-a) - \delta(u+a))$
	$e^{i2\pi ax}$	$\delta(u-a)$
	$e^{i2\pi(ax+by)}$	$\delta(u-a, v-b)$
Gaussian	$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\frac{x^2}{\sigma^2}}$	$e^{-2\pi^2\sigma^2 u^2}$
2D Gaussian	$\frac{1}{\sigma^2 2\pi}e^{-\frac{1}{2}\frac{x^2+y^2}{\sigma^2}}$	$e^{-2\pi^2\sigma^2(u^2+v^2)}$
Dirac-delta	$\delta(x)$	1
Grid (or Shah)	$\sum_{-\infty}^{\infty} \sum_{-\infty}^{\infty} \delta(x-i, y-j)$	Grid(u, v)
Rectangle	$\begin{cases} 1 & \text{if } x \leq \frac{1}{2} \text{ and } y \leq \frac{1}{2} \\ 0 & \text{elsewhere} \end{cases}$	sinc(u, v)

Table 7.2 Some Fourier transforms.

The adverse effect of small values of H can be limited in a number of ways, one of which is to use the *pseudoinverse* filter H_p . We compute

$$F = GH_p,$$

where H_p is defined by

$$H_{p,1}(u, v) = \begin{cases} \frac{1}{H(u, v)} & \text{if } |H| > \varepsilon \\ 0 & \text{if } |H| \leq \varepsilon, \end{cases}$$

where ε is a suitably small positive constant.

A more sophisticated approach is not to use a threshold ε , but to let the value of the inverse filter smoothly go to zero if $|H|$ gets very small. In this case we define the pseudoinverse filter H_p by

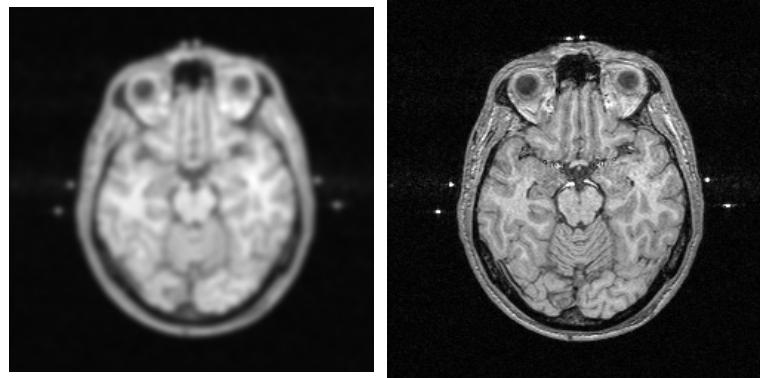
$$H_{p,2}(u, v) = \frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + S(u, v)}.$$

If $S(u, v)$ is determined optimally from the amount of noise³ at the frequencies (u, v) , then this filter is called a *Wiener* filter. Since the mathematics behind a Wiener filter are fairly complex, we will not detail the computation of S here. In practice, S is often set to an empirically established constant value. This usually produces acceptable results if the original image does not contain too much noise.

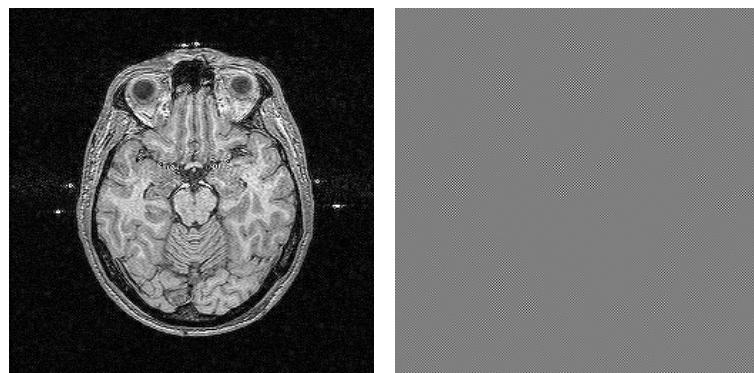
³Or, more accurately, from the *signal to noise* ratio.

Example

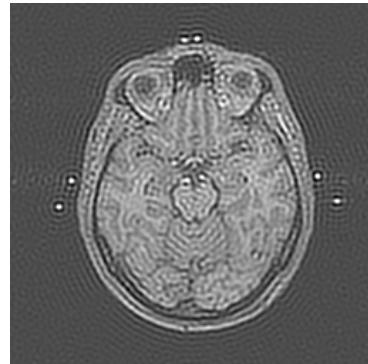
Suppose we have an input image f_b we know to be blurred by Gaussian convolution: $f_b = g * f_i$, where g is a Gaussian kernel, and f_i the ‘ideal’ image. Fourier transforming gives us $F_b = GF_i$, so we can theoretically find f_i by computing $\mathcal{F}^{-1}\left[\frac{F_b}{G}\right]$. In the case where the convolution is *exactly* Gaussian and we know the width of the Gaussian, inverse filtering works very well. Here are f_b and the reconstructed f_i in this case:



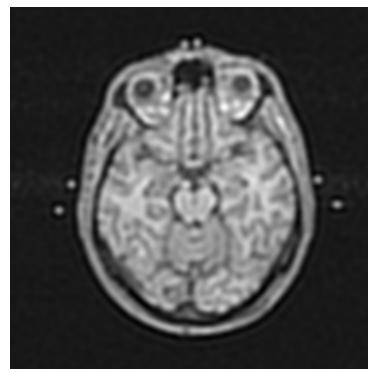
But look what happens to the reconstructed f_i if we estimate the width of the Gaussian incorrectly; in these cases a 3% and 17% error in the width σ :



Now, small values of G have caused severe artifacts. Using the pseudoinverse filter $H_{p,1}$ in the latter case can remove the worst artifacts (using a suitable value for ε):



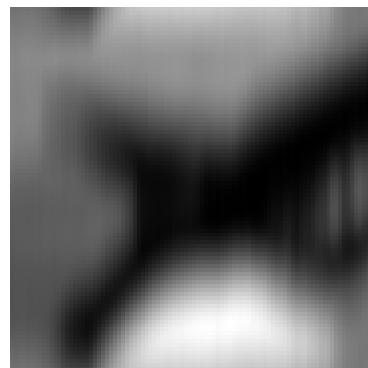
Using $H_{p,2}$ with a suitable constant value for S gives us a somewhat better result:



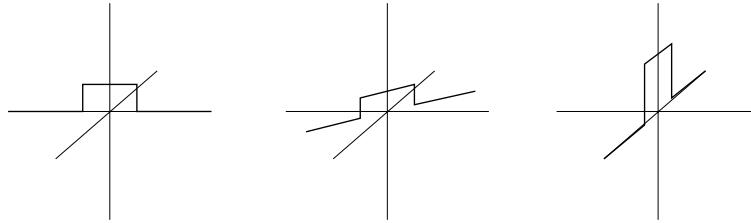
We can use inverse and pseudoinverse filtering to reverse –or, at least, attempt to reverse– any convolution process. An example of this is the removal of motion blur. Blurring of images caused by movement of the imaging device or the object imaged is a very common problem in many imaging areas, notably those where imaging requires a relatively long exposure time.

Example

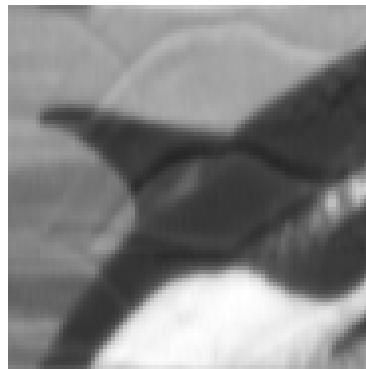
Suppose we have the following image:



where we suppose this image to be blurred by a movement of the camera during acquisition. We assume that in the blurred image, each pixel value is the average of the values of a number of pixels in the ideal image that are along a line in the direction of blur. So the blurring can be described as a convolution with a rectangular kernel, such as the ones below:



where the height axis shows the kernel value, and the other two axis span the image plane, so the convolution effect will be blurring in a horizontal, diagonal, and vertical direction respectively. From the input image, we need to estimate the direction of blur, which determines the orientation of the kernel, and the extent of the blurring, which determines the width of the kernel. In this particular case, we judge the blurring direction to be vertical, and gauged the amount of blur by trial-and-error. The blurring kernel h is a vertically oriented rectangular filter, which we model by $h(x, y) = \text{rect}_c(y)\delta(x)$, where $\text{rect}_c(y)$ is a rectangular kernel with value $\frac{1}{2c}$ if $|y| \leq c$ and zero value otherwise. The delta function $\delta(x)$ ensures the kernel is infinitely narrow and has values only along the y -axis. Although this may look like a function that is difficult to work with, its Fourier transform is quite friendly: $H(u, v) = \frac{\sin(2\pi cv)}{2\pi cv}$. Using the pseudoinverse filter $H_{p,2}$ with a suitable constant value for S , we are able to deblur the input image to:



7.5 The Discrete Fourier Transform

In the previous sections we have used Fourier transforms of images, even though we have formally defined the Fourier transform for continuous functions only. Since a dig-

ital image is not a continuous function, but a matrix of sampled data, we have used an adapted type of Fourier transform, called the *discrete Fourier transform* (DFT). The DFT $F(u, v)$ of an image $f(x, y)$ with dimensions $d_x \times d_y$ is defined by

$$F(u, v) = \sum_{x=0}^{d_x-1} \sum_{y=0}^{d_y-1} f(x, y) e^{-i2\pi(\frac{ux}{d_x} + \frac{vy}{d_y})},$$

where $u \in \{0, 1, \dots, d_x - 1\}$, and $v \in \{0, 1, \dots, d_y - 1\}$. The inverse transformation is defined by

$$f(x, y) = \frac{1}{d_x d_y} \sum_{u=0}^{d_x-1} \sum_{v=0}^{d_y-1} F(u, v) e^{i2\pi(\frac{ux}{d_x} + \frac{vy}{d_y})}.$$

The DFT image of an $d_x \times d_y$ image is again a $d_x \times d_y$ image, which generally is complex-valued. Computing a larger DFT image is not useful, because the $d_x \times d_y$ image already captures all of the frequency information up until the Nyquist frequency of the original.

The DFT has all of the properties we defined for the ordinary Fourier transform.

Many algorithms can be found that efficiently compute the DFT of an image. The algorithm complexity can be reduced by using the point symmetry⁴ and the separability of the transform, and by using smart reordering of the image data. The most common method of computing the DFT is known as the *fast Fourier transform* or FFT.

7.6 Other transforms

The Fourier transform is based on the fact that functions and images can be viewed as a sum of sinusoids. The frequency of these sinusoids is closely related to image aspects such as the level of detail in an image. The Fourier transform $F(u)$ of a function $f(x)$ gives us information on the sinusoids at frequency u , and the previous sections have hopefully shown that the Fourier transform is a very powerful transform in the sense that careful manipulation of $F(u)$ can be used to perform a wide range of image processing tasks.

Since the Fourier transform decomposes an image into sinusoids, these functions are called the *basis functions* of the Fourier transform. The fact that an image can be viewed as a sum of sinusoids raises the question whether there are alternative choices for basis

⁴To be more precise, the transform is *conjugate* symmetric if the input image has real values.

functions, and hence if there are alternative transforms that may be useful for image processing. The answer is yes: many families of functions are suitable as basis functions; only basic mathematical requirements need to be fulfilled. The associated transforms can have very diverse properties. Some transforms may have properties similar to the Fourier transform, yet may be easier to compute. Others may be better suited to compactly represent the shapes and sizes of objects. Still others may present the essential image content compactly, and so are useful in image compression. Some dozen or so transforms pop up regularly in image processing applications, but none have gained the popularity (or, in fact, the usefulness) of the Fourier transform. Well-known transforms are, *e.g.*, the Hartley, cosine, sine, Hadamard, Karhunen-Loéve, Slant, and various Wavelet transforms. A few of these transforms are listed below.

The Hartley transform

If we rewrite the Fourier transform $F(u)$ of $f(x)$ in terms of sines and cosines:

$$\begin{aligned} F(u) &= \int_{-\infty}^{\infty} f(x)e^{-i2\pi ux}dx \\ &= \int_{-\infty}^{\infty} f(x) (\cos(2\pi ux) - i \sin(2\pi ux)) dx, \end{aligned}$$

then the Hartley transform $H(u)$ is defined by leaving out the complex i :

$$H(u) = \int_{-\infty}^{\infty} f(x) (\cos(2\pi ux) - \sin(2\pi ux)) dx,$$

and the discrete Hartley transform (DHT) is defined by

$$H(u) = \sum_{x=0}^{d-1} f(x) \left(\cos \frac{2\pi ux}{d} - \sin \frac{2\pi ux}{d} \right).$$

A fast Hartley transform (FHT) can also be defined in a similar fashion as with the Fourier transform.

The Hartley transform is a computationally attractive alternative for the Fourier transform. Note that the Hartley transform produces real-valued output when a real-valued input image is used, while the Fourier transform is generally complex-valued in this case. On the downside, expressions and properties tend to be more complicated than in the Fourier case. It is possible to use the Hartley transform in all of the examples in the

sections on the Fourier transform, producing the same results, but –as mentioned– the expressions are more difficult and somewhat less easy to work with.

The cosine transform

The discrete cosine transform (DCT) $F(u)$ of a function $f(x)$ is defined by

$$F(u) = c(u) \sum_{x=0}^{d-1} f(x) \cos \frac{\pi u(2x+1)}{2d},$$

where $c(u)$ is defined by

$$c(u) = \begin{cases} \sqrt{d-1} & \text{if } u = 0 \\ \sqrt{2}\sqrt{d-1} & \text{otherwise} \end{cases}$$

The cosine transform can be computed much faster than the Fourier transform. Its most used property is the fact that essential image content can be represented in a very compact form. Hence the cosine transform is much used in image compression techniques.

The wavelet transform

The wavelet transform is a generalized type of transform in the sense that its basis functions are not pre-determined. In the transforms treated in the above, the basis functions were a particular combination of sines and cosines in each case. With the wavelet transform the basis functions are all computed from a single *mother wavelet* $\psi(x)$, which can be any function as long as it obeys certain mathematical rules. The other basis functions are created by scaling and translating the mother wavelet.

Given a mother wavelet $\psi(x)$, the family of basis functions is generated by scaling with a factor a and translation by a factor b :

$$\psi_{a,b}(x) = \frac{1}{\sqrt{|a|}} \psi \left(\frac{x-b}{a} \right).$$

The continuous wavelet transform is then defined by

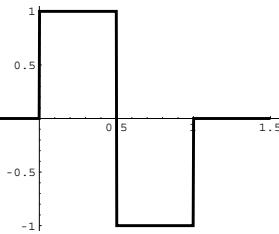
$$F(a, b) = \frac{1}{\sqrt{|a|}} \int_{-\infty}^{\infty} f(x) \psi \left(\frac{x-b}{a} \right) dx.$$

The discrete form is somewhat more complex than may be expected, because the possible values of a and b are restricted by the use of a discrete grid. A common definition for integer values of a and b is

$$F(a, b) = \int_{-\infty}^{\infty} f(x) 2^{-\frac{a}{2}} \psi(2^{-a}x - b) dx.$$

A popular choice for the mother wavelet is the *Haar function* $\varphi(x)$, which is defined by

$$\varphi(x) = \begin{cases} 1 & \text{if } 0 \leq x < \frac{1}{2} \\ -1 & \text{if } \frac{1}{2} \leq x < 1 \\ 0 & \text{otherwise} \end{cases}$$



The blocky shape of this function makes it especially suitable for use with digital images.

Being a general transform, the applications of the wavelet transform are extensive and diverse. Since it is a fairly new transform, its scope is not well known yet, but it has already proven its worth in *multiresolution* image analysis, which will be treated in chapter 9.

The Karhunen-Loëve transform

This transform is also known as *principal component analysis*. In the social sciences, it is called *factor analysis*. In yet other areas of science, it appears under other names such as the Hotelling transform or eigenvalue analysis.

The Karhunen-Loëve transform (KLT) is different from other transforms in the sense that its basis functions are determined from the image or function itself. In general, this transform is especially useful for getting rid of redundant parameters in problems that have many parameters. For example: suppose a scientist wishes to study the occurrence of a certain phenomenon, and has compiled a list of twenty parameters that may be connected to the occurrence of this phenomenon. (One may think of, e.g., a heart attack as the phenomenon, and blood pressure, heart rate, etc. as parameters.) It may be that some of the parameters in his list are redundant. It may also be that only a *combination* of certain parameters is significant to the occurrence of the phenomenon. The KLT is useful in determining whether these are the case.

Let us consider the two parameter example from figure 7.5, where the two parameters are represented by x and y . In this figure, each closed dot represents the x and y parameter values at the occurrence of the phenomenon. An open dot represents a measurement of x and y when no phenomenon occurred. By observing the picture, it will be obvious that the parameters u and v are much better suited for discriminating between the occurrence and non-occurrence of the phenomenon; just take a look at the (x, y) and the (u, v) coordinates of each dot. The u and v axes are called the *principal axes* of the cloud of black dots. *The KLT is used for transforming from (x, y) to (u, v) space.* We will not cover all of the necessary mathematics of the KLT here, but an equivalent technique is covered in chapter 8.

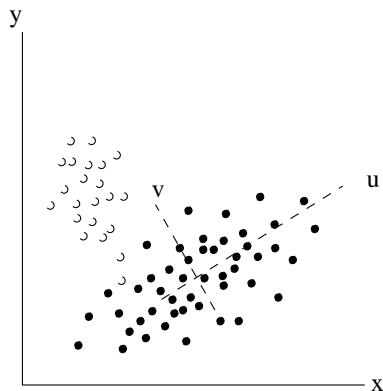


Figure 7.5 The principal axes u and v of a data cloud.

The cloud of black dots can be viewed as a grey-valued image $f(x, y)$ if we say that the grey value $f(x, y)$ equals the number of times a black dot occurs in pixel (x, y) . By this simile, we can use the KLT to find the center and orientation of roughly elliptical shapes in grey-valued images.

Chapter 8

Object representation and analysis

Image segmentation deals with dividing the image into meaningful structures. When segmenting an image, we are usually trying to find objects that are depicted in the image. More generally, many image processing tasks are aimed at finding a group of pixels in an image that somehow belong together. Finding such groups is essential if we want to do measurements on objects depicted in an image. For example, if we want to determine the length of a car depicted in a digital photograph, we first need to determine the group of pixels that makes up the car. In a medical image, if we want to determine the volume of the left heart ventricle, we need to determine which pixels make up this ventricle. Finding such groups of pixels is also an essential step in many image interpretation tasks, such as automated object recognition.

We will call all such ‘groups of pixels’ *objects*, even though they do not necessarily represent a physical object. In this chapter, we will assume an object has already been detected in an image, and we will concentrate on how to use the associated set of pixels for doing shape and other measurements such as length, surface area, orientation, roundness measurements, etc. We will also show alternative representations for objects which are often of importance in follow-up image interpretation tasks and in compactly storing object information.

8.1 Object shape measurements

Measurements like area, length, curvature etc. of an object are often the simplest measurements we can do. Because they reduce the pixel configuration of an object to a single scalar number, such measures will hide much of the detailed shape information of the original object. This is not necessarily a drawback; many applications can be handled well using only global scalar shape measurements. The difficulty usually lies

in selecting the type of measurement appropriate for your application; in selecting the type of measurement that reduces the information you want to measure to a scalar.

The literature provides us with a large number of basic shape measurements, but there is often no consensus on their mathematical definition. Examples are measures like roundness, elongation, compactness, etc. Two different books on image processing will likely give you two different definitions of such measures. This is in itself not a problem as long as these measures are used in a qualitative sense, but it is something to beware of.

Below are some of the most common global scalar shape measurements. Strictly speaking, some are not actual *shape* measurements (area, for instance), but are still important in this context. The 'true' shape measures are invariant under rigid and scale transformations, i.e., do not alter when an object is moved or scaled, and are identified in table 8.1.

Area. The area of an object can be determined by counting all the pixels contained in the object.

Perimeter. The perimeter is the total length of the object boundary. The object boundary is the collection of all boundary pixels, which are all object pixels that have a non-object pixel for a neighbor¹. The perimeter can be measured by tracing the boundary of the object and summing all the steps of length 1 or $\sqrt{2}$ you take, see figure 8.1. A faster way (but with a different result) is to disallow diagonal steps, and trace the boundary by only moving up, down, left or right, see figure 8.1.

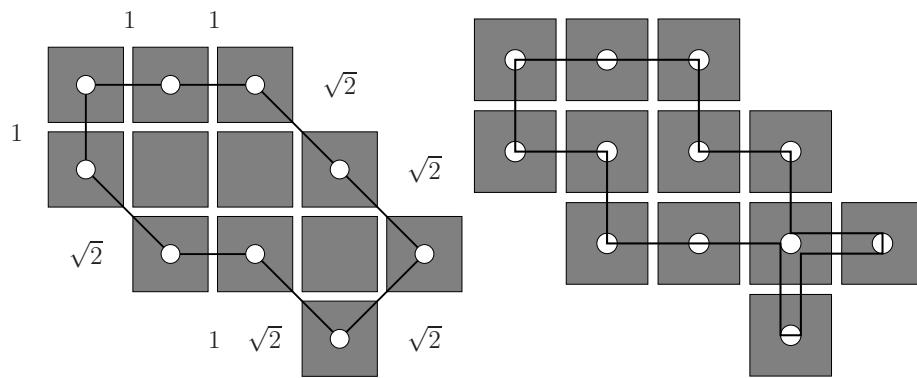


Figure 8.1 Computing the perimeter of an object. Left: $4 \times 1 + 5 \times \sqrt{2} \approx 11.07$. Right: when diagonal steps are not allowed, the length increases to 14.

The area and perimeter can also be determined faster by using an object representation that describes only the object boundaries. This is described in section 8.3.

¹Usually, only object pixels that have a non-object pixel to their left or right, or at their top or bottom, are considered a boundary pixel. Pixels that are only diagonally connected to a non-object pixel are not considered boundary pixels.

Almost all object measures are sensitive to the image resolution; the same image acquired at a different resolution will give different area and perimeter values. This is not very surprising, but it is important to realize that the effect of changing resolution is different for different measures. For instance, the perimeter will generally increase with increasing resolution, while the area will converge to a stable value.

Compactness. The compactness c can be defined as

$$c = \frac{l^2}{4\pi A},$$

where l is the perimeter, and A the object area. This measure is somewhat confusing because very compact objects have a *low* c value, and objects become less compact when c rises. The factor 4π ensures that c equals 1 (the lowest possible value) for a circle. Many books omit this factor in their definition. The reciprocal $1/c$ is sometimes called the *roundness* of an object. Figure 8.1 shows example objects with high and low compactness.

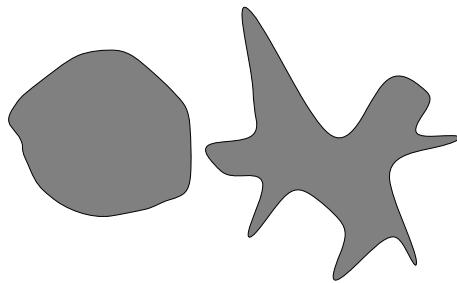


Figure 8.2 On the left an object with a low c value is shown, on the right one with a higher c value. The left object is clearly the more compact of the two.

Longest chord. A line connecting two pixels on the object boundary is called a *chord*. For many shapes, the length and orientation of the longest possible chord (see figure 8.3) give us an indication of size and orientation of the object. If the chord runs between the boundary pixels (x_1, y_1) and (x_2, y_2) , then its length l_c and orientation are given by

$$\begin{aligned} l_c &= \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \\ \tan \alpha &= \frac{y_2 - y_1}{x_2 - x_1}, \end{aligned}$$

where α is the angle of the chord with the positive x -axis. The longest chord can be found by examining all possible combinations of two boundary pixels and comparing their distances.

Longest perpendicular chord. The maximum length l_p of all chords that are perpendicular to the longest chord can give us additional shape information. The *eccentricity* (sometimes called *elongation*) of an object is defined by the ratio $\frac{l_c}{l_p}$ of the lengths of the longest and longest perpendicular chord. Figure 8.3 shows some examples of objects with high and low eccentricity.

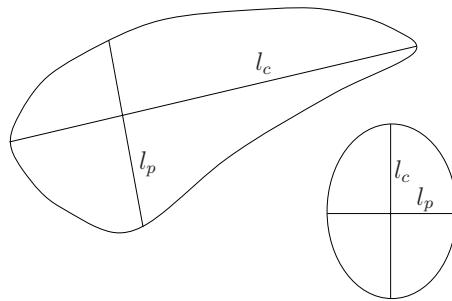


Figure 8.3 Two figures with the longest chord (l_c) and longest perpendicular chord (l_p) drawn in. The left figure has high eccentricity (high l_c to l_p ratio), the right figure has low eccentricity.

It is also possible to use *all* boundary chords for a shape measurement; the distribution of the lengths and orientation of all possible chords can give important shape information.

Minimal bounding box area. A *bounding box* of an object is a rectangle containing the object and which sides touch the object. By examining all bounding boxes when varying their orientation (see figure 8.4), we can find the bounding box with the smallest area.

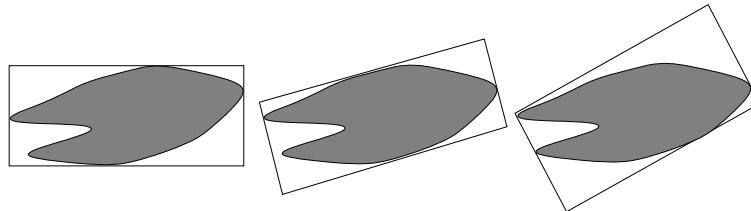


Figure 8.4 Bounding boxes of an object in three different orientations. The middle figure shows the bounding box with minimal area.

The minimal bounding box area A_m can be approximated by the product $l_c l_p$, which may be easier to compute.

Rectangularity. The *rectangularity* (sometimes called *extent*) of an object is defined by the ratio A/A_m of the object area (A) and minimal bounding box area (A_m). This ratio is 1 if the object is a rectangle, and smaller for all other shapes.

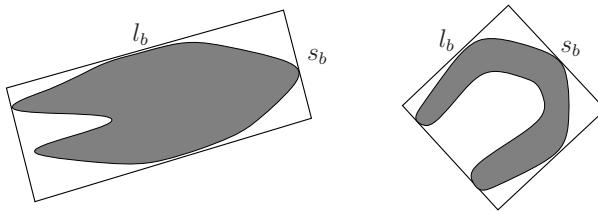


Figure 8.5 The elongation is defined as the aspect ratio l_b/s_b of the minimal bounding box. The right figure shows an example of an object shape where this ratio fails to produce the desired result.

Elongation. The *elongation* can be defined as the aspect ratio (the ratio of the longest l_b and shortest side s_b) of the minimal bounding box. Figure 8.5 shows some examples.

The figure also shows a case where the elongation measure fails. An alternative definition for elongation that works better in this case is $\frac{A}{t^2}$, where t is the ‘thickness’ of the object. A measure for thickness is the number of erosion steps (using, e.g., a 3×3 square structuring element) necessary to completely remove the object.

Curvature. Because any boundary made up out of pixels only has right angles, the definition of boundary curvature for an image object is different from conventional definitions. A common way of defining the curvature in a boundary pixel i is by using the angle α between the lines intersecting at boundary pixel i , each starting at a pixel n boundary pixels removed from boundary pixel i , see figure 8.6.

The global object curvature can then be defined as the ratio c_s/c_t , where c_t is the number of pixels in the object boundary and c_s is the number of pixels where the curvature exceeds a certain threshold. The global curvature has two parameters: this threshold and the number n . Increasing n decreases the sensitivity to very local curving of the boundary.

Bending energy. Another way to aggregate the local curvature into a global measure is to compute the *bending energy* B of the boundary which is defined by

$$B = \frac{1}{l} \sum_{i=1}^{c_t} \alpha^2(i),$$

where l is the perimeter and α is the angle as defined before.

Table 8.1 summarizes all of the measures in this section.

Example

Below are a number of images containing objects of various distinct types. Next to the images are measures that are suited for discrimination between the various types of objects.

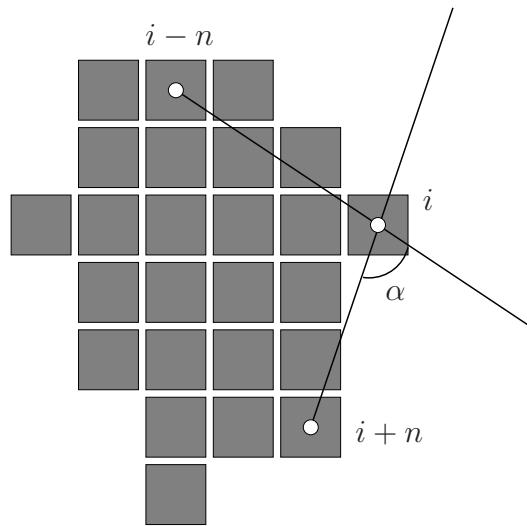
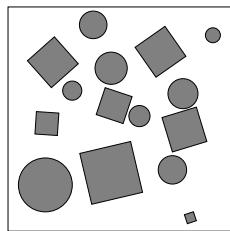


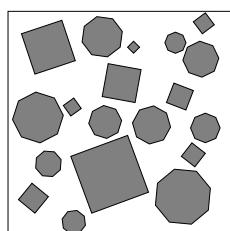
Figure 8.6 The curvature at boundary pixel i can be determined by the angle α between the lines intersecting at i starting at the boundary pixels n boundary steps away. In this case, n equals 3.

Image	Description	Best suited measure
	Two sizes of circles	Area
	Circles and ellipses	Eccentricity
	Ellipses in two orientations	Orientation of longest chord



Circles and squares

Rectangularity or compactness



Squares and octagons

Rectangularity or compactness

Note that in many of these simple examples, many measures from the list can serve as a good discriminator. For instance, in the first example, perimeter, longest chord, etc. will serve as well as area. In the second example, compactness, elongation etc. will also do. In practice, things are not generally that easy, and the measures chosen need to be carefully tested on their discriminative behavior.

Name	Symbol / Formula
Area	A
Perimeter	l
Compactness (*)	$c = l^2/(4\pi A)$
Roundness	$1/c$
Longest chord, length	$l_c = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Longest chord, orientation	$\tan \alpha = (y_2 - y_1)/(x_2 - x_1)$
Longest perpendicular chord	l_p
Eccentricity (*)	l_c/l_p
Minimal bounding box area	$A_m = l_b \cdot s_b$
Rectangularity (extent) (*)	A/A_m
Elongation (*)	l_b/s_b
Elongation (2) (*)	A/t^2
Curvature	c_s/c_t
Bending energy	$B = \frac{1}{l} \sum_{i=1}^{c_t} \alpha^2(i)$

Table 8.1 Summary of shape measures. An asterisk (*) denotes if a measure is a 'true' shape measure, i.e., does not change when the object is moved or scaled. See text for details.

8.1.1 Skeleton based measurements

Most of the measures in the previous section are computed easiest taking only the boundary pixels of an object into account. Since the boundary completely describes the object, we do not necessarily lose information when using only the boundary for a measurement. But the same holds for a measurement using the object skeleton together with its quench function (see chapter 6). Using this representation or only the skeleton of an object has advantages for computing some shape measures, such as object thickness and some topological measures.

Number of holes. An important topological measure is the number of holes inside an object. Given a skeleton of an object –which has topologically the same structure as the object²– we can determine the number of holes from this equation:

$$h = 1 + \frac{b - e}{2},$$

where h is the number of holes, and b and e are the number of branches and endpoints of the skeleton.

Intermezzo

The relation $h = 1 + \frac{b-e}{2}$ is understood easiest if we rewrite it to

$$e = 2 + b - 2h,$$

which can be generated using the following reasoning

- A line segment has two endpoints. $e = 2$
- Each branch causes an additional endpoint. $e = 2 + b$
- Each hole causes a cycle in the skeleton, which means two endpoints disappear per hole. $e = 2 + b - 2h$

A necessary condition is that a line only branches into two lines, not more.

²In terms of the skeletons defined in chapter 6, this means we must use the skeleton by thinning rather than the skeleton by maximal balls.

Although all topological attributes of the original object can be derived from the skeleton, it is in many cases not practical to do so. The reason for this is that –even though the skeleton is a *very compact* structure topologically equivalent to the object– it is often not easier to interpret or use than the object itself.

Object thickness. The *quench* function assigns to each point of the skeleton the radius of the maximal ball that fits at that particular point of the skeleton. The quench function therefore gives us a measure of local thickness of the object. In the computation of the skeleton by maximal balls, we automatically obtain the quench values. Figure 8.7 shows an image containing objects with the quench function superimposed. High quench values (high grey value) indicate a large local object width.



Figure 8.7 A binary image with the quench function superimposed on it. High quench values indicate a high local object thickness.

The skeleton by maximal balls is chosen here because we automatically obtain the quench values when computing it. However, in the digital case, it is a discontinuous skeleton and very susceptible to small boundary changes, causing branches in the skeleton where the intuition would not place them. In that respect, the skeleton by thinning would have been a better choice. This skeleton, however, does not supply us with quench values. A solution is to use the skeleton by thinning in combination with a *distance transform* of the image.

The distance transform. The distance transform of a binary image assigns to each object pixel its distance to the closest background (non-object) pixel. Figure 8.8 shows an example of a distance transform.

Computing a distance transform using exact euclidean distances as in figure 8.8 is a relatively time-consuming task. In practice, an approximation using only integer numbers is often used.

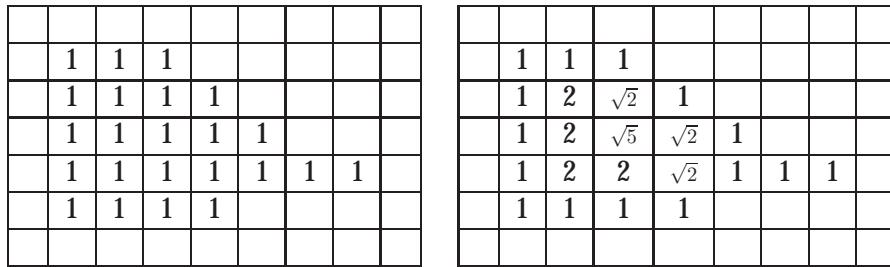


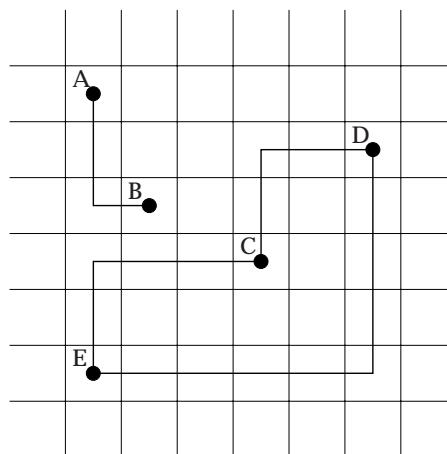
Figure 8.8 An image and its Euclidean distance transform. The pixel values in the distance transform equal the distance to the nearest background pixel. Background pixels are left blank for clarity.

The simplest distance transform measures the distance between an object pixel and its nearest background pixel using the *city block* distance d_c , which is the length of a shortest connecting path that has only right angles. The city block distance between two pixels (x_1, y_1) and (x_2, y_2) is defined by

$$d_c((x_1, y_1), (x_2, y_2)) = |x_2 - x_1| + |y_2 - y_1|.$$

The city block distance between two pixels is always an integer.

Example



This figure shows some example pixels and shortest connecting paths that contain only right angles. Note that there are many different paths of equal 'shortest' length in each case. The city block distance is the length of the shortest path, i.e.,

Pixels	City block distance
A, B	3
C, D	4
C, E	5
D, E	9

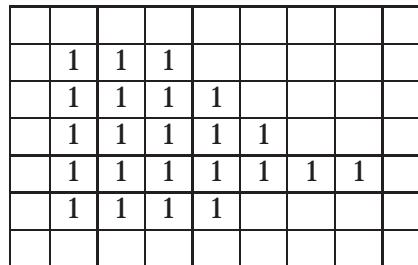
The distance transform DT_c of a binary object image f using the city block distance can be computed using a sum of erosions:

$$\text{DT}_c(f) = \sum_{i=0}^n \varepsilon_i(f),$$

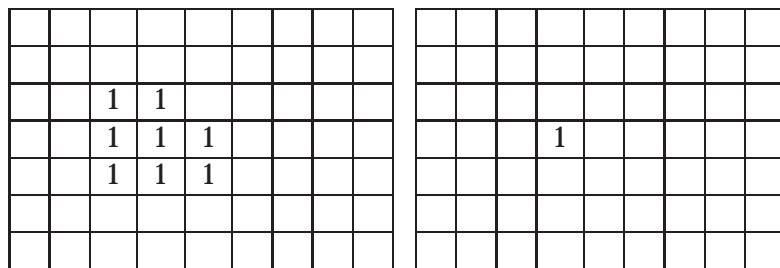
where $\varepsilon_i(f)$ is the i -th iteration of erosion of f using a '+'-shaped 3×3 structuring element. The variable n is chosen large enough such that $\varepsilon_n(f)$ is an empty image.

Example

Given this image $f = \varepsilon_0(f)$:



Then its erosions $\varepsilon_1(f)$ and $\varepsilon_2(f)$ using a 3×3 '+'-shaped structuring element equal:



The sum $\varepsilon_0(f) + \varepsilon_1(f) + \varepsilon_2(f) = \text{DT}_c(f)$ then equals:

1	1	1						
1	2	2	1					
1	2	3	2	1				
1	2	2	2	1	1	1		
1	1	1	1					

There are many alternative algorithms to compute this distance transform. This approach using erosions –although very transparent– is in fact a very inefficient algorithm. An algorithm that requires only two passes over the image is:

1. In the image, set all background pixel values to 0, and all object pixel values to ∞ .³
2. Scan the image from left to right and top to bottom. For each pixel (x, y) , replace its current value $v(x, y)$ by

$$v(x, y) = \min\{ v(x - 1, y - 1) + 2, \\ v(x, y - 1) + 1, \\ v(x + 1, y - 1) + 2, \\ v(x - 1, y) + 1, \\ v(x, y) \},$$

using padding with $+\infty$ if necessary.

3. Scan the image from right to left and bottom to top. For each pixel (x, y) , replace its current value $v(x, y)$ by

$$v(x, y) = \min\{ v(x, y), \\ v(x + 1, y) + 1, \\ v(x - 1, y + 1) + 2, \\ v(x, y + 1) + 1, \\ v(x + 1, y + 1) + 2 \}.$$

Instead of the values '+1' and '+2' –which are correct for the city block distance– other values (such as 3 and 4) may be used if a better approximation of euclidean distances (up to a factor) is required.

Example

In the first pass over the image, we can visualize the minimum operation by fitting the following kernel:

³In practice, any value larger than the largest possible distance will do.

+2	+1	+2
+1	+0	

where the double edged pixel is placed over the current pixel in the image, (x, y) . The current pixel value $v(x, y)$ is then replaced by the minimum of the sums of the kernel values and the underlying image pixel values. (Note that the pass is then continued in the image with the replaced value; the replacement is not done in a temporary image as customary in many other algorithms, but directly in the current image.)

The backward pass can in a similar fashion be visualized, now using the kernel

	+0	+1
+2	+1	+2

Here is the same test image as before, and the initialization image of the algorithm (step 1):

1	1	1							
1	1	1	1						
1	1	1	1	1					
1	1	1	1	1	1	1			
1	1	1	1						

∞	∞	∞							
∞	∞	∞	∞						
∞	∞	∞	∞	∞					
∞	∞	∞	∞	∞	∞				
∞	∞	∞	∞						

After the first pass and second pass the image becomes:

1	1	1							
1	2	2	1						
1	2	3	2	1					
1	2	3	3	2	1	1			
1	2	3	4						

1	1	1							
1	2	2	1						
1	2	3	2	1					
1	2	2	2	1	1	1			
1	1	1	1						

Note that the result after the first pass shows the distance to the nearest background pixel to the left or top, and that this is ‘corrected’ to include the remaining two sides in the second pass.

The distance transform is closely related to the skeleton (and quench function) of an object. Figure 8.9 shows an example image and its distance transform. The distance transform of a binary image is a grey valued image. The figure shows that the ‘strings’ of maxima occurring in the distance transform occur approximately at the location of the skeleton.

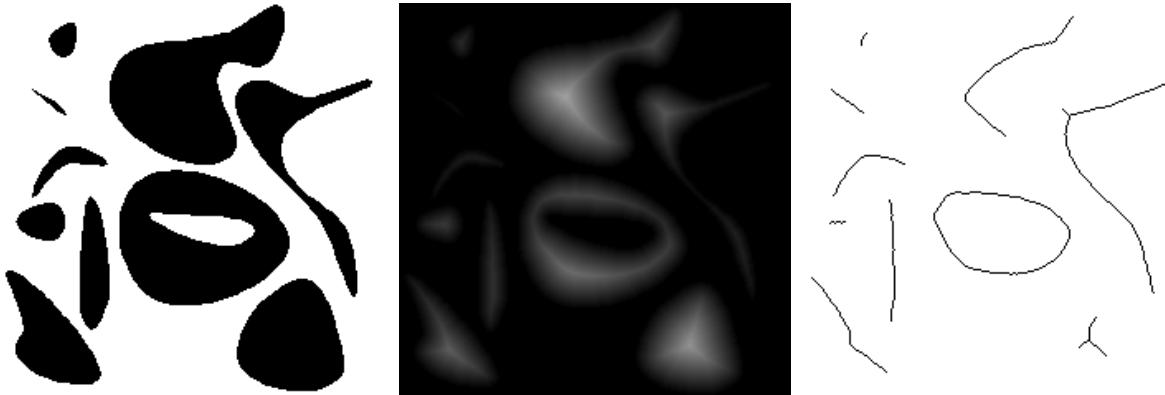


Figure 8.9 An example figure, its distance transform, and its skeleton by thinning. Note that the maxima in the distance transform approximately correspond to the skeletal pixels.

Combining a skeleton by thinning and a distance transform of an object image gives us information on the object thickness: the value of the distance transform at a pixel that is part of the skeleton gives an indication of the local object thickness. Because of the way the skeleton and distance transform are implemented, the skeletal pixels do not usually coincide *exactly* with the maxima of the distance transform. But the match is close enough to be useful in many practical applications. Figure 8.10 shows an example where the distance transform image has been multiplied by the skeleton image. The result is a quench-like function showing local object thickness.

8.1.2 Moment based measurements

If we have an object described by a binary image $f(x, y)$,⁴ then we can write the area A of the object as a summation:

$$A = \sum_x \sum_y f(x, y),$$

⁴i.e., $f(x, y) = 1$ if (x, y) is part of the object, and $f(x, y) = 0$ if (x, y) is part of the background.



Figure 8.10 A multiplication of the distance transform image and the skeleton image from the previous figure. The result is a quench-like image showing local object thickness. The image has been dilated for visibility.

where the indices x and y run over the entire image. We can also compute the average x and y coordinates of the object pixels by a summation:

$$\left\{ \begin{array}{lcl} \bar{x} & = & \frac{\sum \sum xf(x,y)}{\sum \sum f(x,y)} = \frac{1}{A} \sum \sum xf(x,y) \\ \\ \bar{y} & = & \frac{\sum \sum yf(x,y)}{\sum \sum f(x,y)} = \frac{1}{A} \sum \sum yf(x,y) \end{array} \right.$$

The average coordinates (\bar{x}, \bar{y}) denote the *centroid* or *center of gravity* or *center of mass* of the object.

Example.

The area A equals $\sum \sum f(x, y)$, so in fact the number of grey pixels in the image:
 $A = 23$.

$$\bar{x} = \frac{\sum \sum x f(x, y)}{A} = \frac{1+2+3+1+2+3+4+5+6+7+0+1+2+3+4+5+6+7+1+2+3+2+3}{23} = \frac{73}{23} \approx 3.2.$$

$$\bar{y} = \frac{\sum \sum y f(x, y)}{A} = \frac{2+0+1+2+3+0+1+2+3+4+0+1+2+3+4+1+2+1+2+1+2+1+2}{23} = \frac{40}{23} \approx 1.7.$$

The computed centroid $(3.2, 1.7)$ corresponds to the point where we would intuitively place it.

The summation formulas for area and centroid fit into a larger class of formulas computing *moments*:

The *moment of order p* of a function $f(x)$ is defined by

$$m_p = \int_{-\infty}^{\infty} x^p f(x) dx.$$

For two-dimensional functions the moment of order $p + q$ is defined by

$$m_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^p y^q f(x, y) dx dy.$$

For a digital image $f(x, y)$ this becomes

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y),$$

where the sums are evaluated over the entire image. Moments can be used to measure a variety of physical properties of an object.

Given the definition of image moments, we now see that for the area and centroid holds:

$$A = m_{00}, \quad \bar{x} = \frac{m_{10}}{m_{00}}, \quad \bar{y} = \frac{m_{01}}{m_{00}}.$$

The *central moments* μ_{pq} are defined by

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y).$$

Central moments of order two can be used to find the axis of least moment of inertia of an object; an axis that for many objects corresponds to the intuitive ‘length’ axis of the object through (\bar{x}, \bar{y}) . The angle θ of this axis with the positive x -axis equals

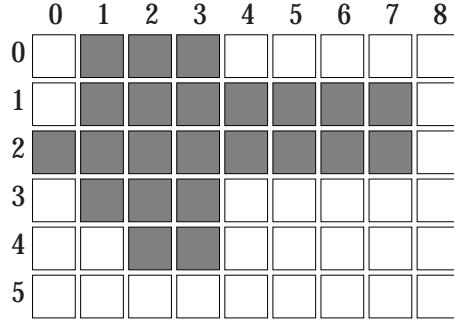
$$\theta = \frac{1}{2} \arctan \left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}} \right).$$

The same moments can give us a new measure for the object eccentricity e :

$$e = \frac{(\mu_{20} - \mu_{02})^2 + 4\mu_{11}}{m_{00}}.$$

Example

Using the figure from the previous example:



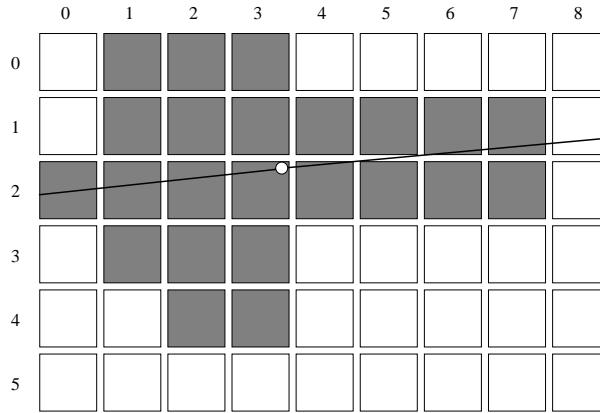
We established before that $(\bar{x}, \bar{y}) = (\frac{73}{23}, \frac{40}{23}) \approx (3.2, 1.7)$. We can now find $\theta = \frac{1}{2} \arctan \left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}} \right)$ by evaluating $(x - \bar{x})(y - \bar{y})$, $(x - \bar{x})^2$, and $(y - \bar{y})^2$ for all object pixels (where $f(x, y) = 1$), as in the table below. The totals of the right three columns then equal μ_{11} , μ_{20} , and μ_{02} respectively.

(x, y)	$x - \bar{x}$	$y - \bar{y}$	$(x - \bar{x})(y - \bar{y})$	$(x - \bar{x})^2$	$(y - \bar{y})^2$
(0, 2)	-73/23	6/23	-438/529	5329/529	36/529
(1, 0)	-50/23	-40/23	2000/529	2500/529	1600/529
(1, 1)	-50/23	-17/23	850/529	2500/529	289/529
(1, 2)	-50/23	6/23	-300/529	2500/529	36/529
(1, 3)	-50/23	29/23	-1450/529	2500/529	841/529
(2, 0)	-27/23	-40/23	1080/529	729/529	1600/529
(2, 1)	-27/23	-17/23	459/529	729/529	289/529
(2, 2)	-27/23	6/23	-162/529	729/529	36/529
(2, 3)	-27/23	29/23	-783/529	729/529	841/529
(2, 4)	-27/23	52/23	-1404/529	729/529	2704/529
(3, 0)	-4/23	-40/23	160/529	16/529	1600/529
(3, 1)	-4/23	-17/23	68/529	16/529	289/529
(3, 2)	-4/23	6/23	-24/529	16/529	36/529
(3, 3)	-4/23	29/23	-116/529	16/529	841/529
(3, 4)	-4/23	52/23	-208/529	16/529	2704/529
(4, 1)	19/23	-17/23	-323/529	361/529	289/529
(4, 2)	19/23	6/23	-114/529	361/529	36/529
(5, 1)	42/23	-17/23	-714/529	1764/529	289/529
(5, 2)	42/23	6/23	252/529	1764/529	36/529
(6, 1)	65/23	-17/23	-1105/529	4225/529	289/529
(6, 2)	65/23	6/23	-390/529	4225/529	36/529
(7, 1)	88/23	-17/23	-1496/529	7744/529	289/529
(7, 2)	88/23	6/23	528/529	7744/529	36/529
\sum			-2622/529	47242/529	15042/529

Which gives us

$$\theta = \frac{1}{2} \arctan \left(\frac{2 \frac{-2622}{529}}{\frac{47242}{529} - \frac{15042}{529}} \right),$$

so $\theta \approx -4.6^\circ$, and the centroid and axis of least moment of inertia look like:



Intermezzo*

With a little mathematics, the orientation angle θ of the axis of minimal rotational inertia can readily be derived.

Assuming that f is a binary image with $f(x, y) = 1$ if (x, y) is part of the object O , and $f(x, y) = 0$ otherwise, we can write the central moments as

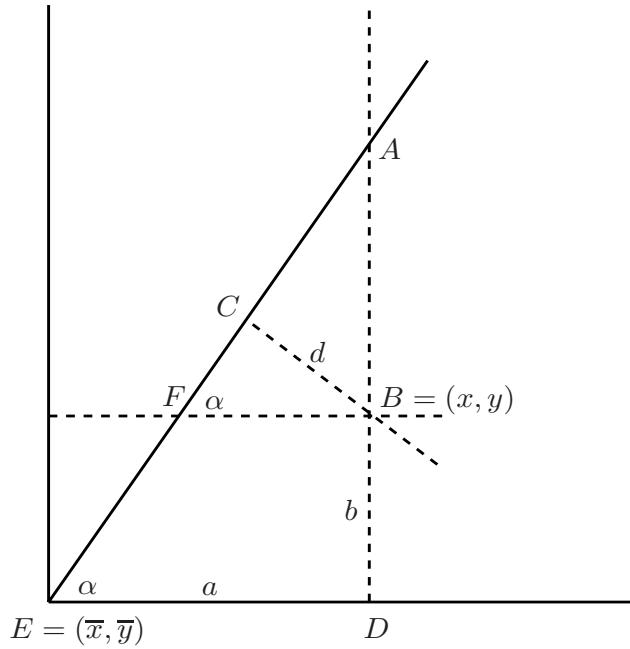
$$\mu_{pq} = \sum_{(x,y) \in O} \sum (x - \bar{x})^p (y - \bar{y})^q.$$

Physics dictates that the angle θ is the angle where the sum of squared distances of all object pixels –to the axis oriented by θ through (\bar{x}, \bar{y}) – is minimal, so⁵

$$\theta = \arg \min_{\alpha} \sum_{(x,y) \in O} \sum d(x, y)^2,$$

where $d(x, y)$ is the distance of pixel (x, y) to the axis through (\bar{x}, \bar{y}) oriented by the angle α . For an arbitrary pixel (x, y) and angle α we can draw this diagram:

⁵The \arg operator returns the argument of the function it operates on. In this case, it returns the value of the angle α where the minimum sum of distances is achieved.



where we have used $a = x - \bar{x}$ and $b = y - \bar{y}$. We need to compute the distance $d = BC$:

1. $\tan \alpha = \frac{AD}{ED}$, so $AD = a \tan \alpha$.
2. Since the triangles ADE and ABF are congruent, we know that $\frac{BF}{DE} = \frac{AB}{AD}$, so $BF = \frac{AB}{AD}DE = \frac{a \tan \alpha - b}{a \tan \alpha}a = \frac{a \tan \alpha - b}{\tan \alpha}$.
3. $\sin \alpha = \frac{BC}{BF}$, so $\sin \alpha = BC \frac{\tan \alpha}{a \tan \alpha - b}$, and hence $BC = \frac{\sin \alpha(a \tan \alpha - b)}{\tan \alpha} = a \sin \alpha - b \cos \alpha$.

Substituting gives us

$$\begin{aligned}\theta &= \arg \min_{\alpha} \sum_{(a,b) \in O} \sum (a \sin \alpha - b \cos \alpha)^2. \\ &= \arg \min_{\alpha} \sum_{(a,b) \in O} (a^2 \sin^2 \alpha - 2ab \sin \alpha \cos \alpha + b^2 \cos^2 \alpha).\end{aligned}$$

The minimal value can be found by setting the derivative to α to zero:

$$\begin{aligned}\sum_{(a,b) \in O} (a^2(2 \sin \theta \cos \theta) - 2ab(\cos^2 \theta - \sin^2 \theta) + b^2(-2 \cos \theta \sin \theta)) &= 0 \\ \sum_{(a,b) \in O} (a^2 \sin(2\theta) - 2ab \cos(2\theta) - b^2 \sin(2\theta)) &= 0 \\ \sum_{(a,b) \in O} \left(a^2 - 2ab \left(\frac{1}{\tan(2\theta)} \right) - b^2 \right) &= 0 \\ \mu_{20} - \frac{2}{\tan(2\theta)} \mu_{11} - \mu_{02} &= 0 \\ \frac{1}{\tan(2\theta)} &= \frac{\mu_{20} - \mu_{02}}{2\mu_{11}} \\ \theta &= \frac{1}{2} \arctan \left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}} \right).\end{aligned}$$

Fitting a bounding box. A good approximation for the minimal bounding box is the bounding box that is oriented with the angle θ defined in the above. If we transform all object boundary pixels (x, y) to (x', y') by the following transformation (see figure 8.11), effectively translating then rotating the coordinate system:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x - \bar{x} \\ y - \bar{y} \end{pmatrix},$$

then the boundary pixels are now in a coordinate system that is natural to the bounding box. Finding the boundary pixels that touch the bounding box is now easy, since these are the pixels where x' or y' are maximal or minimal. Once we have found these, we can calculate the dimensions of the bounding box, and shape features like rectangularity and elongation.

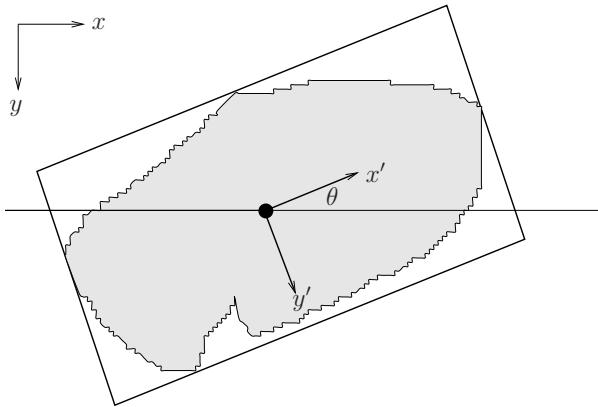


Figure 8.11 By rotating the (x, y) coordinate system by the angle θ to the (x', y') system we get a coordinate system that is natural to the bounding box. The pixels where the object touches the box are the pixels where x' or y' are maximal or minimal.

Fitting an ellipse. Besides fitting a bounding box, it is also often useful to fit an *ellipse* to the object. The ellipse center is placed at (\bar{x}, \bar{y}) , its longest axis is aligned with θ , and the lengths of its axes l_1 and l_2 can be determined from these formulas:

$$\begin{aligned} I_1 &= \sum \sum (a \cos \theta + b \sin \theta)^2 \\ I_2 &= \sum \sum (a \sin \theta - b \cos \theta)^2 \\ l_1 &= \left(\frac{4}{\pi}\right)^{1/4} \left(\frac{I_1^3}{I_2}\right)^{1/8} \\ l_2 &= \left(\frac{4}{\pi}\right)^{1/4} \left(\frac{I_2^3}{I_1}\right)^{1/8} \end{aligned}$$

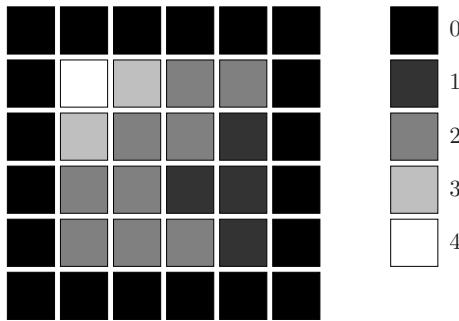
Where the summations run over all coordinates (a, b) in the object relative to the centroid. If the axes lengths are chosen according to these formulas, the ellipse will have the same rotational inertia as the object. The fraction l_1/l_2 is yet another measure for eccentricity of the object.

8.1.3 Grey value moments

The formulas for the moments m_{pq} and central moments μ_{pq} do not only apply to binary images, but to grey valued images as well. Effectively, the moments are then weighed by the image grey values. This weighing has a realistic physical analogue, e.g., in the computation of the center of mass of a physical object: the object may be composed of parts made out of different materials with different mass density. Since the distribution of mass owing to different parts has an impact on the location of the center of mass, the moments involved in the computation of this center must be weighed by the local mass density. In a similar vein, it is often useful when computing the center of mass of an object in an image to use weighing by grey values.

Example

Given this image $f(x, y)$:



We consider all pixels with a grey value not equal to zero part of the ‘object’ (a 4×4 square). The grey value weighted center of mass (\bar{x}, \bar{y}) can be computed using exactly the same moment formulas as before:

$$\begin{cases} \bar{x} = \frac{m_{10}}{m_{00}} = \frac{\sum \sum x f(x,y)}{\sum \sum f(x,y)} \\ \bar{y} = \frac{m_{01}}{m_{00}} = \frac{\sum \sum y f(x,y)}{\sum \sum f(x,y)}, \end{cases}$$

which gives us

$$\begin{cases} \bar{x} = \frac{1 \cdot 4 + 2 \cdot 3 + 3 \cdot 2 + 4 \cdot 2 + 1 \cdot 3 + 2 \cdot 2 + 3 \cdot 2 + 4 \cdot 1 + 1 \cdot 2 + 2 \cdot 2 + 3 \cdot 1 + 4 \cdot 1 + 1 \cdot 2 + 2 \cdot 2 + 3 \cdot 2 + 4 \cdot 1}{4 + 2 \cdot 3 + 9 \cdot 2 + 4 \cdot 1} = \frac{70}{32} \approx 2.19 \\ \bar{y} = \frac{1 \cdot 4 + 1 \cdot 3 + 1 \cdot 2 + 1 \cdot 2 + 2 \cdot 3 + 2 \cdot 2 + 2 \cdot 2 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 2 + 3 \cdot 1 + 3 \cdot 1 + 4 \cdot 2 + 4 \cdot 2 + 4 \cdot 2 + 4 \cdot 1}{4 + 2 \cdot 3 + 9 \cdot 2 + 4 \cdot 1} = \frac{73}{32} \approx 2.28. \end{cases}$$

Note that this center of mass is displaced from the geometrical object center towards the brighter part of the object.

Grey value weighted moments can be computed in two ways. They can be computed by applying the moment formulas directly to the grey valued image, but it is also possible to use a hybrid binary/grey value method, where the moment is computed using only object pixels, but using their original grey value in the moment formulas (see figure 8.12).

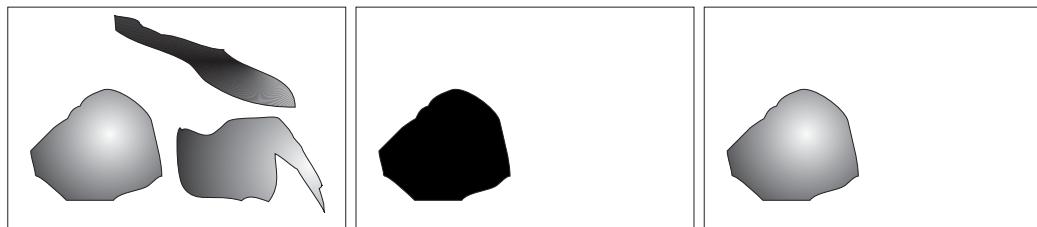
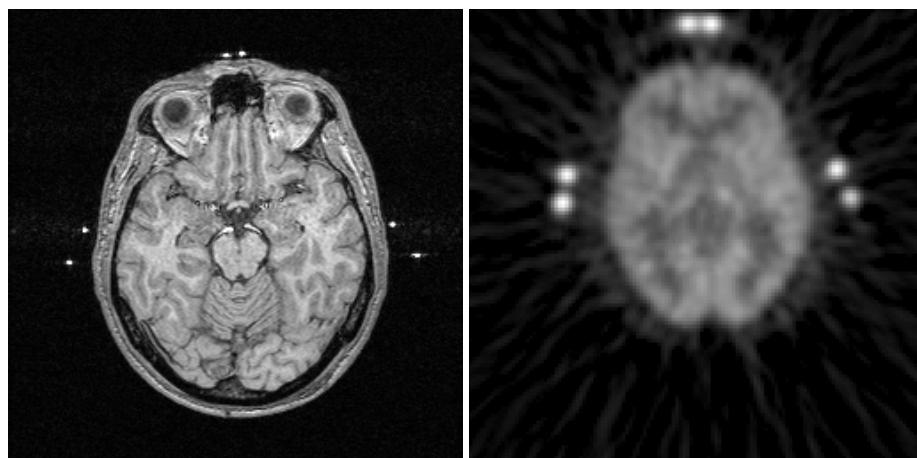


Figure 8.12 Left: original image. Middle: segmented object. Right: multiplication of the original and object image; showing the object with its original grey values. By computing moments on the right image, the moment is confined to the single object, but still grey value weighted.

Intermezzo: finding spots in blurry images

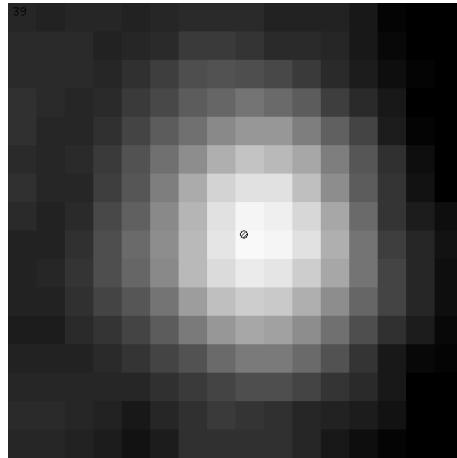
The (grey value weighted) center of mass is often useful to locate the exact position of a spot that has been ‘blurred’ over an image region by the image acquisition process.

An example of this is the use of markers attached to patients in medical imaging. Such markers are used as reference points in images. In the images below, they show up as three pairs of bright dots just outside of the head:



The markers used in the right image have the same physical dimensions as the ones used in the left image. Clearly, blurring of the marker spots has occurred in the right image.

This image shows a zoom of one of the markers in the right image, and the grey value weighted center of mass:



8.2 Multiple object measurements

8.2.1 Aggregating separate object measurements

It is often useful to obtain aggregate measures concerning *all* objects contained in an image. The easiest way to obtain such measures is to do measurements on all of the objects separately, and aggregate the results using statistical techniques. In this way, we can, *e.g.*, compute the average object area, or the variance of the object perimeter.

Example

Given an image with five objects of which we have measured the area:

Object	1	2	3	4	5
Area	10	12	4	10	12

The average area then equals

$$\bar{A} = \frac{10 + 12 + 4 + 10 + 12}{5} = 9.6.$$

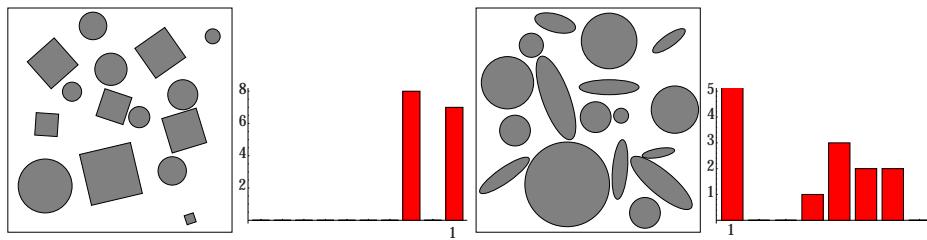
The variance of the area equals

$$\sigma_A^2 = \frac{(10-9.6)^2 + (12-9.6)^2 + (4-9.6)^2 + (10-9.6)^2 + (12-9.6)^2}{5} \approx 8.64.$$

Instead of reducing all separate object measurements to a global measure such as area or variance, it is also worthwhile to collect all measurements in a histogram and examine the statistical distribution. This can often give us vital information on the shape variation of objects in an image.

Example

Below are two images and a histogram of respectively rectangularity and eccentricity:



The rectangularity distribution tells us that there are only two distinct values present in the image. It is likely there are only two shapes in the image. Since one of the rectangularity values is one, we know the corresponding shape is a rectangle. A quick glance at the image confirms this. The distribution can easily be used to divide the image objects into two groups of similarly shaped objects by thresholding the rectangularity using any threshold between the two columns in the histogram.

The eccentricity distribution is slightly more difficult: there appear to be two groups of shapes, with the eccentricity of the second group spread over a small region. Examining the image shows why: the first group consists of circles with constant eccentricity, the second group of ellipses with a variation of their eccentricity. In this case, it is still possible to divide the objects into circles and ellipses by thresholding the eccentricity value, where the threshold can be any value between the two groups in the histogram.

We have shown only a few of the simplest examples of using multiple object measurements. In practice, the number of approaches is very large, only limited by the number of possible statistical methods. The circumstances of the application determine what approach is best suited for a particular problem.

In many instances it is also possible to obtain a global measure concerning multiple objects, but *without doing measurements on individual objects*. An example is the size distribution of objects we have seen in chapter 6 (the granulometry), which uses only openings applied to the entire image. Another example is finding the average object orientation by fitting an ellipse to its Fourier magnitude image, and determining the orientation of the ellipse's long axis. Such an approach may need less computations than approaches that require a measurement of each separate object. There are again few basic rules to finding such measures; the number of possible approaches is very large and application dependent.

8.2.2 Counting objects

The most basic multiple object measurement is determining the number of objects contained in an image. If we have a binary image f with $f(x, y) = 1$ if (x, y) is part of an object, and $f(x, y) = 0$ otherwise, then we can count the number of objects by using the following algorithm (see figure 8.13):

1. Set *number of objects* to 0.
2. Scan the image until we reach an object pixel (x, y) , i.e., $f(x, y) = 1$.
 - (a) Increase *number of objects* with 1.
 - (b) Find all pixels connected to (x, y) , i.e., find the entire object.
 - (c) Remove the object from the image by setting all its pixel values to 0.
3. Continue scanning (back to step 2) until we reach the end of the image.

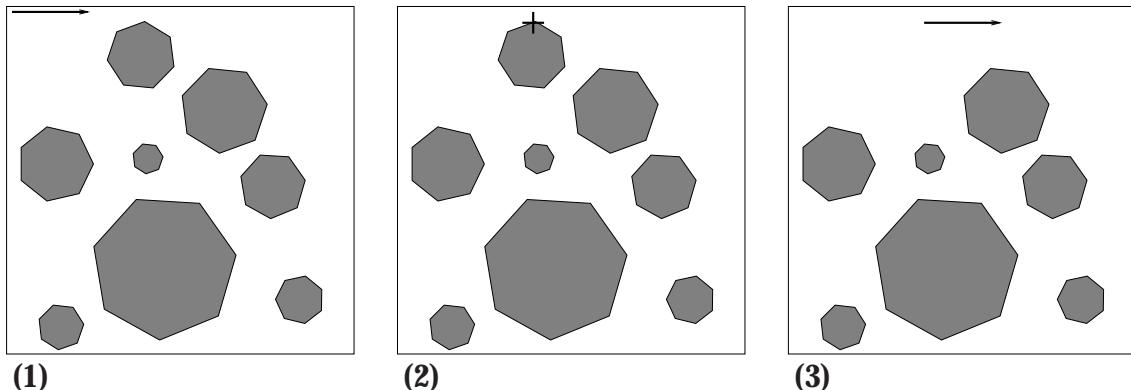


Figure 8.13 Counting the objects contained in an image. (1) *number of objects* = 0. Start scanning until we find an object. (2) Object found. Increase *number of objects*. Find the entire object and remove it. (3) Continue scanning until we reach the end of the image.

The variable *number of objects* will hold the correct number of objects after termination of the algorithm. An alternative counting algorithm is:

1. Ultimately erode the image. This leaves a single marker for each object.
2. If necessary, reduce the markers to isolated points by thinning using structuring elements that remove everything but isolated points. The result g now contains just as many isolated points as f contained objects.
3. Count the number of isolated points in g . This equals $\sum\sum g(x, y)$.

Many other alternative counting algorithms exist. A slight modification of the original algorithm allows us to assign a separate label to each object. Such labels are useful in any application where we need to be able to select a particular object, e.g., for a separate object measurement, visualization, etc.

1. Set *number of objects* to 0.
2. Scan the image until we reach an object pixel (x, y) , i.e., $f(x, y) = 1$.
 - (a) Increase *number of objects* with 1.
 - (b) Find all pixels connected to (x, y) , i.e., find the entire object.
 - (c) Set all of the found object pixels to a unique value, e.g., $(\text{number of objects} + 1)$.
3. Continue scanning (back to step 2) until we reach the end of the image.

After termination, the objects will be labeled $\{2, \dots, n + 1\}$, where n equals the number of objects, see figure 8.14. Each pixel of an object will now have the same unique value. Selecting a particular object i for a measurement now equals selecting all pixels with value i . Computing the area of each object is now particularly easy, since the area of object i now equals histogram entry i .

1	1	1					
1	1		1	1			
			1	1			
					1	1	1
	1	1			1	1	1
					1	1	1
					1	1	1
		1	1	1	1	1	

2	2	2					
2	2		3	3			
			3	3			
					4	4	4
					4	4	4
5	5				4	4	4
					4	4	4
		6	6	6	7	7	7

Figure 8.14 Image containing objects before and after labelling. Zero pixel values are left blank for clarity.

8.2.3 Objects intersecting the image boundary

When computing multiple object measures, attention must be paid to objects intersecting the image boundary. Until now, we have tacitly assumed this does not occur, but this is not a very realistic assumption in many cases. Depending on the measurement task, objects intersecting the boundary must be treated differently. For example:

- When we are interested in the *number of objects per unit area*, we cannot simply divide the number of objects by the image area, because objects may be straddling the image boundary. A common approach to remedy this is to not count objects that intersect the right or lower image boundary. The idea is that what you ‘remove’ at the lower and right boundaries is ‘sticking out’ at the upper and left boundaries of the image.
- When we are interested in the *object density*, i.e., the ratio of object area and image area, we must take *all* objects into account, also those intersecting the boundary.
- When doing *shape measurements* (like eccentricity, rectangularity etc.) we must take only those objects not intersecting any image boundary into account. The reasoning is that shape measurements of partial objects are unreliable because the real shape of the object is unknown. For example, consider the eccentricity of an ellipse cut off along its short axis, which is off by a factor of two.

In the last item, we have oversimplified matters: disregarding objects intersecting the boundaries when doing shape measurements may give results that are biased towards smaller objects. This is because a larger object will have a larger probability of intersecting the boundary, and therefore has a larger probability of being ignored in the shape measurement. To compensate for this bias, we can measure all objects intersecting the top and left boundary in a smaller image as defined in figure 8.15. This method produces unbiased results as long as no object intersecting the top or left boundary of the smaller image also intersects the original image top or left boundary.

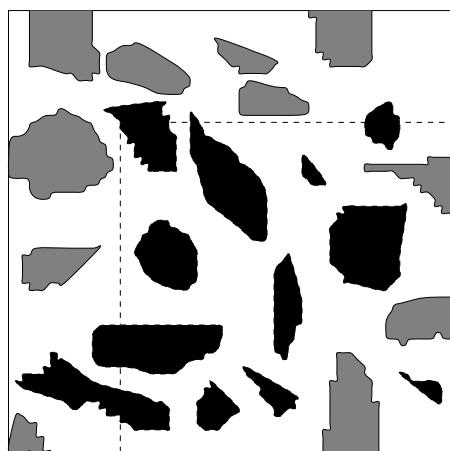


Figure 8.15 In images where objects intersect the image boundaries, unbiased shape measurements can be done by measuring only those objects fully contained in and intersecting the top and left boundaries of a smaller image (the black objects). The smaller image is defined here by the dashed line.

8.2.4 Overlapping objects

In the previous sections we have assumed that the objects contained in an image are fully separated, *i.e.*, do not overlap. In many applications this is not a realistic assumption. Overlapping of objects can cause biased or downright incorrect measurements. Often we are unable to correct for this, simply because we can at best make an educated guess as to what the real objects look like; the occlusion of one object by another introduces an uncertainty we cannot compensate for.

Counting overlapping objects. If we assume that the overlapping line as defined in figure 8.16 is short relative to the objects' width, then we can use a variation of ultimate erosion to count objects in an image. If the overlapping line is short, then iterated erosion of two overlapping objects will separate the objects before one of the objects is fully eroded. If we continue eroding *but do not completely remove an object*, we will end up with a marker for each object, see figure 8.17. Counting all of the markers is then a measure for the number of objects contained in the original image. An assumption here is that the ultimate erosion actually leaves just as many markers as there were overlapping objects, which cannot be guaranteed in practice.

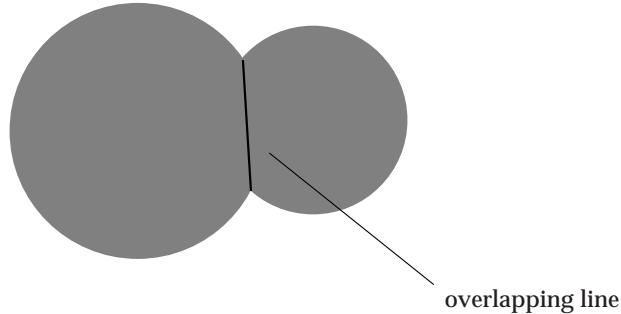


Figure 8.16 The overlapping line of two overlapping objects.

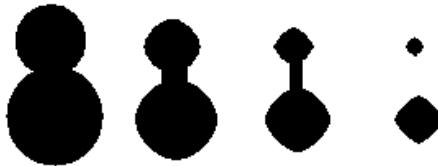


Figure 8.17 Erosion of overlapping objects with a relatively short overlapping line will separate the objects before removing them. If the erosion is ultimate, a marker will be left for each object, so in this case two markers.

Separating objects. Sometimes the shape formed by two overlapping objects gives us a hint that we are dealing with overlapping objects instead of one single object. If we

can separate such shapes, we can remove some of the bias in measurements caused by overlapping objects. Shape measurements will still be unreliable, but we can at least identify some locations of unreliable shape measurements, and we can give a better estimate of the number of objects contained in the image.

Figure 8.18 shows an image containing overlapping circles. Although a human observer will be quick to conclude where overlap occurs, the task of separating the circles is not easy formulated as an image processing algorithm. The SKIZ shown in the same figure gives an indication on how we may approach this: branches extend to the points where objects overlap. If we can extend these branches to connect, the connecting lines we have drawn can be used as object separation lines. We can try to draw these lines in the SKIZ by extending branches and examining if they meet other branches, or we can try modifying the original image so its SKIZ will have connected branches that may serve as separating lines in the original image. This modification of the original image consists of morphological operations (usually openings) to try and separate overlapping objects.

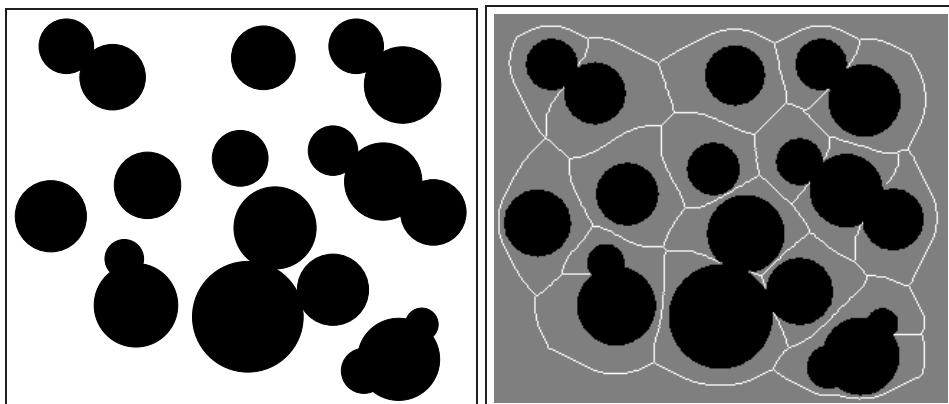


Figure 8.18 An image containing overlapping circular objects and the same image together with its SKIZ.

Another approach to separate objects is to make use of the *watershed* of an image. The watershed is a morphological transform of an image. Its name refers to the analogy of the water drainage pattern of a landscape.

The watershed of a grey valued image is defined as follows (see figure 8.19): think of the image as a landscape, with the grey value being a height coordinate. Make a hole at each of the local minima of the image, and start flooding the landscape with water from these holes; the water rising with the same speed throughout the image. Build a dam at each location where two waterfronts meet. When the image is entirely flooded, the dams together make up the watershed.

As is clear from the resultant watershed image in the figure, the watershed forms a good 'object-separating' network. The watershed can also be used to separate objects in

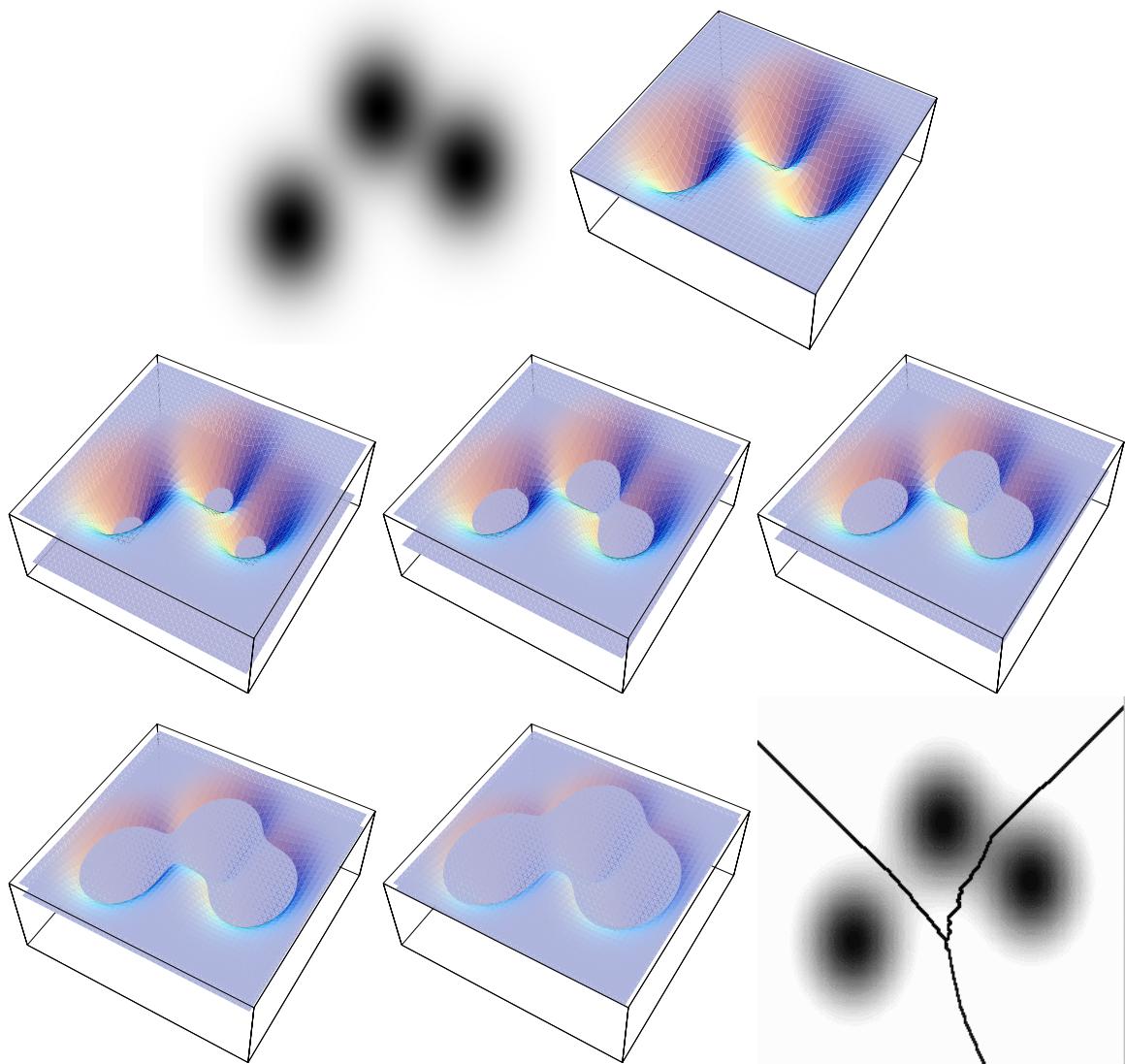


Figure 8.19 Formation of the watershed of a grey valued image. The top row shows the original image and a 'landscape' version. The next rows show the immersion process. The location where waterfronts meet is part of the watershed. The final picture shows the original with the watershed overlaid.

binary images. The watershed algorithm depicted in figure 8.19 is applied to an image where objects have dark values, and the lowest values occur within the objects. If we have a binary object image, we can distance transform it and complement the result to get a similarly valued image. If we apply this algorithm to the image in figure 8.18, and use the watershed to separate the objects we obtain the result seen in figure 8.20. Note that objects that overlap too much still cannot be separated.

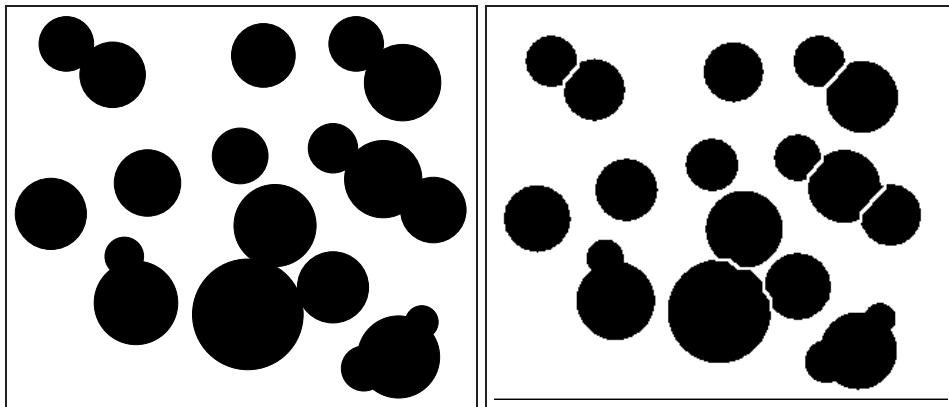


Figure 8.20 Example of object separation using the watershed. See text for details.

8.2.5 Object distribution

When an image contains multiple objects, we are often interested in how these objects are located with respect to each other. The three most characteristic distributions are: clustered, evenly distributed, and random. An interesting physical analogy is that objects attract each other (form clusters), repel each other (become evenly spaced), or do not interact (random distribution). Figure 8.21 shows examples of the three distributions. For simplicity, we show only an example using small circular objects.

A useful tool when examining the object distribution is the histogram of nearest object distances: for each object, the distance to the nearest object is measured, and the results are collected in a histogram. The ‘distance’ measured can be a boundary-to-boundary distance, or a centroid-to-centroid distance. These two distances are easily related when the image contains only small circles as in our examples, and corresponds to an intuitive object distance in this case. When objects have more complex shapes, however, the measured distance may not show this correspondence. For lack of a more suitable distance definition, these two distances are still often used in practice.

If only the three characteristic cases (clustered, evenly spaced and random) need to be distinguished, the histogram of nearest object distances can be used together with heuristics. In the clustered case, the histogram will mainly show very small distances. In the evenly spaced case, the histogram will mainly show distances of a certain fixed value. In the random case, the histogram will be more dispersed than in the other two cases.

Cluster tendency*

Interpretation of the object distances histogram in a more formal statistical framework seems an appropriate way to examine the clustering (or non-clustering) behavior of objects in an image. However, in most practical cases the underlying statistics are extremely complex and difficult to make use of.

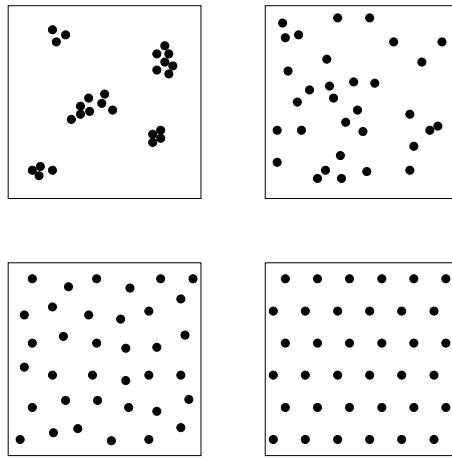


Figure 8.21 Examples of object distribution. Top left: clustered. Top right: random. Bottom left: (approximately) evenly spaced. Bottom right: evenly spaced, locked to grid.

For example, let us assume a random object distribution. For simplicity, we assume the objects to be single points randomly distributed within a circle of radius R . We can then formulate the distribution of the inter-object distances. Given this distribution, we can formulate a statistical test to verify if a certain image complies with a 'random object distribution' hypothesis.

Even in this simplified case, the distribution of object distances is very difficult to find. The correct distribution of the distance r is in fact

$$P[r/(2R) \leq t] = (2t)^2 I_{1-t^2}(3/2, 1/2) + I_{t^2}(3/2, 3/2),$$

for $0 \leq t \leq 1$, where I equals the incomplete Beta function:

$$I_x(p, q) = \frac{\int_0^x z^{p-1} (1-z)^{q-1} dz}{\int_0^1 z^{p-1} (1-z)^{q-1} dz}.$$

If we only consider the distribution of *nearest* object distances, then the distribution is much simpler when assuming random object distribution: these distances have a Poisson distribution with parameter $\sqrt{A/(4n)}$, where A is the image area and n is the number of objects contained in the image.

8.2.6 The Hough transform

The Hough transform of an image is a valuable tool for finding lines in images. The basic idea is to transform the image from normal (x, y) image space into the parameter

space that is appropriate for describing lines. For example, the line $y = ax + b$ has two parameters a and b . By varying a and b , we can model any line⁶ in the image. Each line in the image is therefore represented by a dot in (a, b) space. Figure 8.22 shows some examples of this.

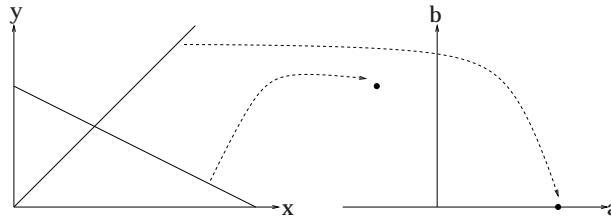


Figure 8.22 The left picture shows two lines $y = ax + b$, one with $(a, b) = (1, 0)$, and one with $(a, b) = (-\frac{1}{2}, 1)$. The Hough transform of this image—which is simply the (a, b) space of the image—therefore shows two dots: $(1, 0)$ and $(-\frac{1}{2}, 1)$.

If we have a binary image containing lines, we can convert the image to (a, b) space in the following way: create an empty image H . Then, for each object pixel (*i.e.*, a pixel that is part of a line) in the original image, determine which pairs (a, b) create a line $y = ax + b$ that passes through the object pixel, and increase $H(a, b)$ for those pairs (a, b) .

Although theoretically correct, this algorithm is not very practical, because there is an infinite number of possible lines passing through the object pixels, and hence there is an infinite number of corresponding pairs (a, b) . A more practical algorithm examines only a discrete number of pairs (a, b) , as in this algorithm:

1. Select a set of discrete values for a and b , *e.g.*, $a \in \{0, 0.1, 0.2, 0.3 \dots, 100\}$, $b \in \{0, 1, \dots, 1000\}$.
2. Create a digital image H , and set all values $H(a, b)$ to 0.
3. For each object pixel p :
 - For each value of a_i in the defined set of values:
 - Compute the corresponding b that forms a line $y = a_i x + b$ that passes through p .
 - Round b to the nearest value b_j that is part of the defined set of values.
 - Increase $H(a_i, b_j)$ by 1.

Figure 8.23 shows an example of some possible lines passing through a pixel p .

The resulting image $H(a, b)$ of the algorithm above is called a Hough transform of the original image; a remapping of the original image to a space where the coordinates are

⁶Except vertical lines, where a is infinite.

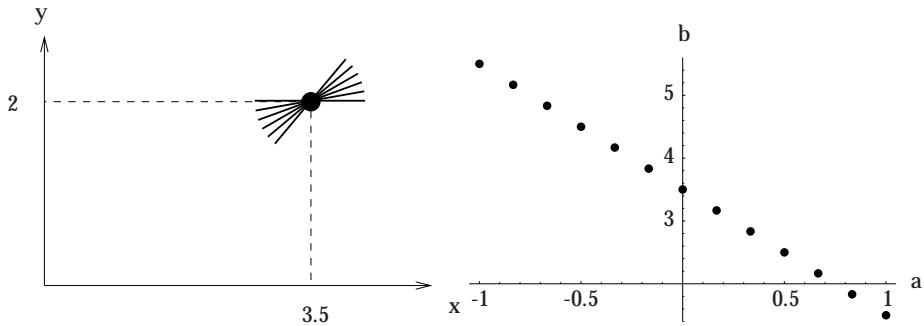


Figure 8.23 Some lines $y = ax + b$ that can be drawn through a given point $(x, y) = (3.5, 2)$ are shown in the left figure. If we vary the coefficient a of those lines discretely, and compute the accompanying value for b , we obtain the plot on the right.

the parameters of lines in the original image. Here's how we can use H to detect lines in the original image: *since all pixels lying on a line have identical values a and b , $H(a, b)$ will have local maxima at points corresponding to lines in the original image.*

Given an image containing lines and the transform $H(a, b)$ computed by the algorithm above, we can determine the equations of the lines in the original image by locating the (a_i, b_i) where H is locally maximum. The corresponding line equation then is $y = a_i x + b_i$. So detecting lines in the original image is equivalent to finding maxima in H .

In practice, the characterization of lines by the parameters a and b is not very efficient, because an equidistant discretization of a corresponds to a set of lines whose angular distance is *not* equidistant. For example, suppose we choose $a \in \{1, 2, 3, 4, 5, 6\}$, then this leads to lines with approximate angles of $\{45, 63, 72, 76, 79, 81\}$ degrees. A better line characterization is to use the distance from the origin d and a normal angle θ with the positive x -axis, as shown in figure 8.24. The equation of a line in terms of θ and d is $x \cos \theta + y \sin \theta = d$. An example of an image and its Hough transform is shown in figure 8.25.

The Hough transform can also be used to detect straight edges instead of lines by a simple modification: in the original image, we first compute an edge image. The edge image now shows a line where the original image has a straight edge. Applying our algorithm to the edge image will find lines, and hence also the straight edges of the original image. A much used procedure to find straight edges is:

1. Compute an edge image from the original image, e.g., by using a Sobel filter.
2. Threshold the edge image to obtain a binary image.
3. Hough-transform the result
4. Locate local maxima in the Hough transform and compute the corresponding lines in the original image.

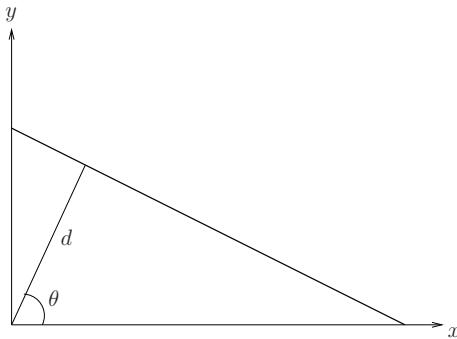


Figure 8.24 The parameters d and θ uniquely characterize a line.

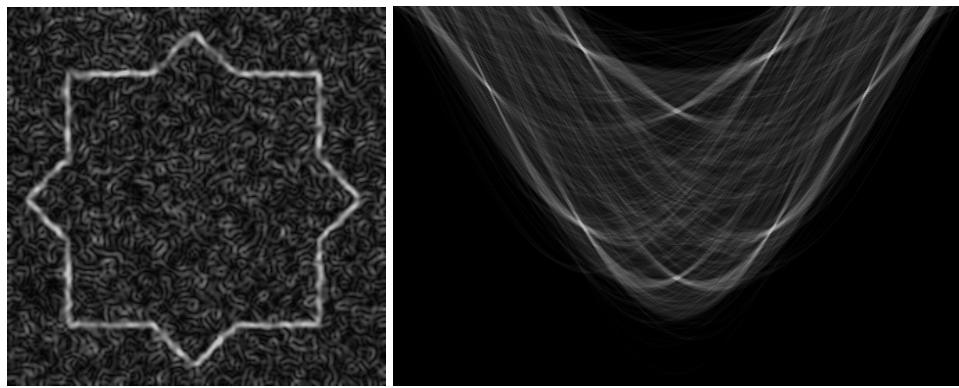


Figure 8.25 Example of a Hough transform. Left: original image. All line segments in the image can be covered by drawing eight obvious lines. The Hough transform ((d, θ) space) is shown on the right. Note that eight local maxima are visible in the Hough space.

Figure 8.26 shows an example of using this procedure for finding straight edges in an image.

In the procedure described above, the threshold step is often omitted, and the grey valued edge image is used instead of a binary image. In this case, the Hough transform is computed as before, but the value of $H(a, b)$ is not increased by one, but with a value equal to the grey value of the current pixel in the edge image.

Using the Hough transform to locate lines and edges is a very robust technique; it is relatively insensitive to noise, object occlusion, missing line parts, etc. The reason for this is that all of these artifacts will have an impact on the Hough transform, i.e., the correct local maxima of H will become smaller, but the artifacts will need to be extensive in order for the local maxima to disappear.

Extension of the Hough transform. We have seen how the Hough transform can be used to detect lines by mapping object pixels to the parameter space of lines: the (a, b)

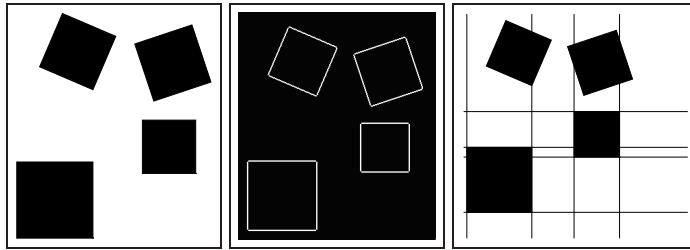


Figure 8.26 Using the Hough transform to locate straight edges in an image. Left: original image. Middle: thresholded Sobel edge image. Right: original image with lines corresponding to maxima in the Hough transform at 0 and 90 degrees.

or (d, θ) space. We can use the same techniques to detect, e.g., circles in an image: we remap the object pixels to a parameter space. The parameters can be a , b , and r , where (a, b) denotes the circle center, and r the circle radius. The algorithm to detect circles is basically the same as with detecting lines, except now we have three parameters: for each object pixel p , see which values of a , b and r form a circle that contains p , and raise $H(a, b, r)$ by one. Then detect the local maxima of H to find the corresponding circles in the original image. Using similar variations of the Hough transform we are able to detect all kinds of parametrized shapes.

The Hough transform can be considered a direct measurement of an image in the sense that it shows us the distribution of lines⁷ contained in the image. It is also often used as a tool to find and select the objects in an image we are interested to take a measurement of.

8.3 Object representation

In many applications, it is useful to not only have an object described in the form of a binary image, but also (or solely) in a representation that describes only the boundary of the object. Such a representation is often much more compact than the original image. It also retains all of the shape information of the original object. Note that many of the measures presented in the first section of this chapter require only knowledge of the boundary of an object. We are therefore still able to compute these measures given only a boundary representation.

Another compact object representation is the object skeleton together with a quench function. This representation also retains all the shape information of the original object, so it can theoretically be used for shape measurements.

⁷Or of other shapes, depending on the type of transform chosen.

8.3.1 Boundary representation: chain codes

The boundary of an object in a digital image can be described by picking a starting pixel in the boundary, looking in which direction a neighboring boundary pixel lies, moving to this pixel and then looking in which direction the next pixel lies, and so on until we reach the starting pixel again. Such a list of directions is called a *chain code*.

The eight compass directions can be encoded by a digit as shown in figure 8.27. If we only allow right angles in our boundary description, four directions suffice. The example below shows a chain code of an object.

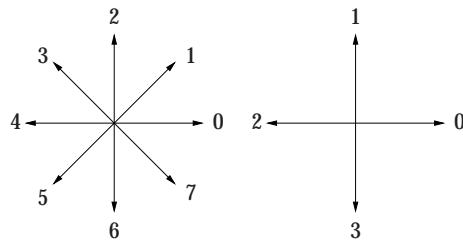
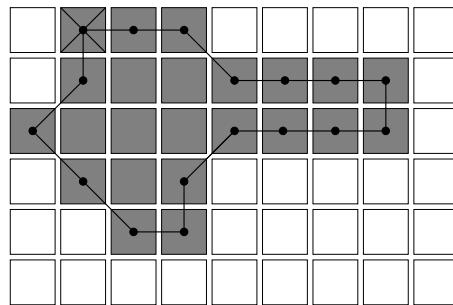


Figure 8.27 In a chain code, digits are used to identify directions. The left picture shows the digits assigned when all eight directions of neighboring pixels are used. The right picture shows the digits used when only four directions are used in the chain code.

Example

Given this object, with the boundary pixels marked:

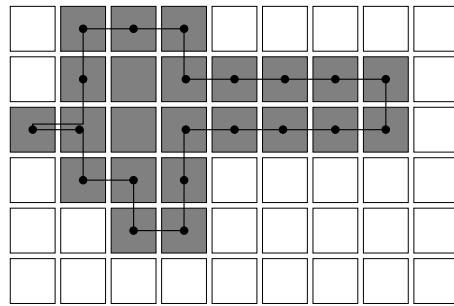


The chain code, when starting from the boundary pixel with the cross and tracing the boundary clockwise, is:

$$\{0, 0, 7, 0, 0, 0, 6, 4, 4, 4, 5, 6, 4, 3, 3, 1, 2\},$$

which is the chain code using eight directions. When we use only four directions, the chain code becomes:

$$\{0, 0, 3, 0, 0, 0, 0, 3, 2, 2, 2, 2, 3, 3, 2, 1, 2, 1, 2, 0, 1, 1\}.$$



The use of four or eight directions in a chain code is dependent on what we consider to be ‘connected pixels’ in the boundary. When using eight directions, we assume all eight pixels nearest to a boundary pixel to be neighbors. When using four directions, we obviously (see the example above) consider only the pixels directly to the top, bottom or at the left or right of a pixel to be connected. We call these two cases ‘8-connected’ and ‘4-connected’.

When looking at the boundary traces in the example above, some find the 8-connected case to be the more logical choice. However, things are not as trivial as they may seem, because of the following paradox:

When we assume an object to be 8-connected, the background is 4-connected.
When we assume an object to be 4-connected, the background is 8-connected.

So we cannot choose the same ‘connectedness’ for the foreground and background of an image. The paradox can be made clear using figure 8.28.

Even when assuming 8-connectivity between object pixels, we can still describe the object boundary using only four directions by describing the *outer* boundary by a chain code instead of the path between the centers of the boundary pixels. Since this path has only right angles, we can describe it using only four directions. This is demonstrated in figure 8.29.

In our example, we needed a list of 17 digits (‘links’) to describe the boundary using 8-connectivity, and 22 digits when using 4-connectivity. Because the 4-connected chain is longer, it would seem that the 4-connected case requires more computer memory to store the chain, but the reverse is true in this example. This is because the 4-connected chain requires only 2 bits per link (we only need to encode 0, …, 3), and the 8-connected chain requires 3 bits per link (for encoding 0, …, 7). So the 4-connected chain needs less bits: $22 \times 2 = 44$ instead of $17 \times 3 = 51$ bits.

The digits in a chain code can be rotated without any effect, *i.e.*, the boundary described does not change.

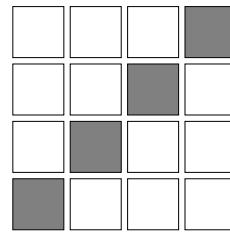


Figure 8.28 The connectivity paradox: If we assume 8-connectedness for object pixels, then diagonal pixels are considered connected, and this image contains only one object. This implies that the background is fully separated into two areas (top-left and bottom-right). But this implies that the background cannot be 8-connected, because then all background pixels would be connected, and there would be only one background area. Assuming the object to be 4-connected leads to the same paradox: in that case the background must be 8-connected.

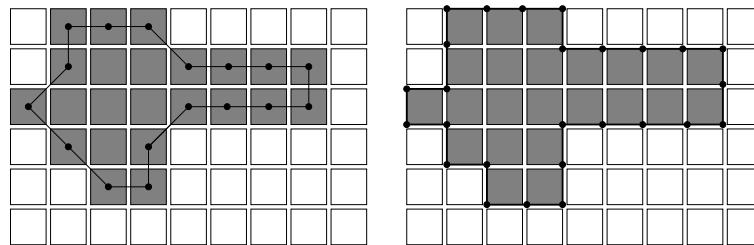
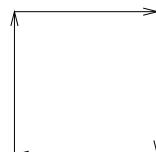


Figure 8.29 Chain through boundary pixel centers (left) and chain enclosing the entire object, the outer boundary (right). The outer boundary can be described by a chain code using only four directions.

Example

This simple chain:



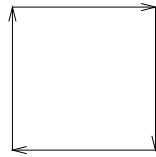
Can be described by the chain $\{0, 3, 2, 1\}$, but also by the rotated chains $\{3, 2, 1, 0\}$, $\{2, 1, 0, 3\}$ and $\{1, 0, 3, 2\}$.

It is sometimes useful to have an algorithm that rotates a chain code to a fixed position no matter what the initial rotation of the code is. For example, an algorithm that rotates all four of the chain codes in the example above to $\{0, 3, 2, 1\}$. Such an algorithm makes

comparing two chain codes easy: we can immediately see if the two codes describe the same object. A simple algorithm to do this is to rotate the chain code until the number formed by all digits together (when placed in a row and viewed as a single number) is minimal.

The derivative chain code. The digits in the chain codes described in the above stand for directions relative to the image frame. For example, in the 4-connected case: right (0), up (1), left (2), or down (3). It is also possible to describe a direction *relative to the previous direction taken*: we can move in the same direction (0), turn left (1), reverse (2), or turn right (3). A chain code built according to this principle is called a *derivative* (or relative) chain code.

Example



The square described by the chain code $\{0, 3, 2, 1\}$ can also be described by the derivative chain code $\{3, 3, 3, 3\}$ (turn right, turn right, turn right, turn right).

The derivative chain code can be obtained from the ordinary chain code by taking the difference⁸ $j - i$ of each pair of successive digits $\{i, j\}$. This representation is slightly easier to use than the ordinary chain code if we are interested in *variation* in the boundary directions rather than its absolute directions.

8.3.2 Measurements using chain codes

Perimeter. The perimeter of an object can easily be derived from a chain code. In the case of a 4-connected chain code, the perimeter equals the number of links (digits) in the chain. This will be immediately obvious when looking at the right picture in figure 8.29. In the case of an 8-connected chain code, we must take into account that the diagonal steps are of length $\sqrt{2}$. The perimeter l then equals $n_{0,2,4,6} + n_{1,3,5,7}\sqrt{2}$, where $n_{0,2,4,6}$ equals the number of links that are 0, 2, 4, or 6 in the chain, and $n_{1,3,5,7}$ equals the number of links that are 1, 3, 5, or 7.

Area. The area of an object equals the difference of the area below its 'upper' and 'lower' boundary, as shown in figure 8.30.

⁸Modulo 4 in the case of 4-connectedness or modulo 8 in the case of 8-connectedness to avoid negative numbers.

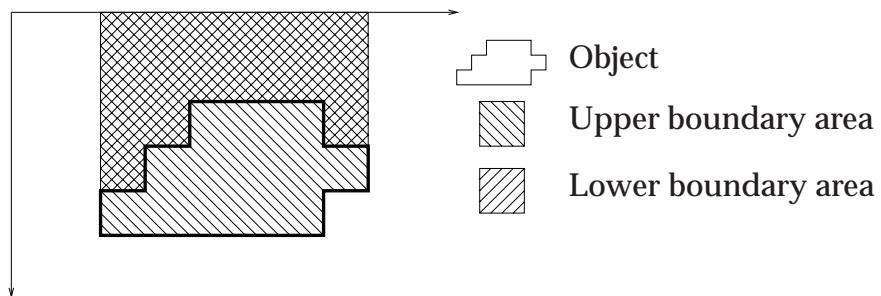


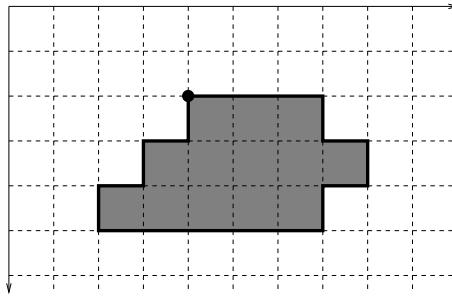
Figure 8.30 The area of an object equals the difference of the upper and lower boundary area.

Given a chain code of the outer boundary of an object, we can compute these two areas. The area of the 'column' below each pixel equals its y -coordinate. So if we keep track of the y -coordinate when following the chain (influenced by the chain codes 1 and 3), and assume the lower boundary to be traced by movements to the right (chain code 0), and the upper boundary by movements to the left (chain code 2), we can compute the area enclosed by the chain. Note that the object area does not depend on its distance from the x -axis ($y = 0$). If we compute the upper and lower area relative to $y = 1$ (or any other value), both areas decrease by the same amount, and the difference of the two still equals the actual object area. This means that the y -coordinate we need to keep track of does not need to be relative to the axis $y = 0$. In fact, we can set it to any value at the start of the algorithm, as long as we keep track of the changes in the y -coordinate. A smart algorithm that does all this is:

1. Set the variables *area* and *y-position* to zero.
2. Process each digit in the chain code:
 - If *digit* = 0 decrease *area* by *y-position*.
 - If *digit* = 1 decrease *y-position* by 1.
 - If *digit* = 2 increase *area* by *y-position*.
 - If *digit* = 3 increase *y-position* by 1.

We assumed that movements to the right traced the lower boundary, and movements to the left the upper boundary. If this should have been the other way around, then the algorithm produces a negative area value. Taking the absolute value will correct this.

Example



The (outer boundary) chain code of the object above is

$$\{0, 0, 0, 3, 0, 3, 2, 3, 2, 2, 2, 2, 2, 1, 0, 1, 0, 1\}.$$

Carrying out the algorithm presented above results in the following table:

chain code	-	0	0	0	3	0	3	2	3	2	2	2	2	2	1	0	1	0	1
area	0	0	0	0	-1	-1	1	1	4	7	10	13	16	16	14	14	13	13	
y-position	0	0	0	1	1	2	2	3	3	3	3	3	3	2	2	1	1	0	

The final area, 13, is correct. A check that everything went as planned is that *y-position* should be zero again at the end of the algorithm.

Center of mass. The formulas to compute the center of mass (\bar{x}, \bar{y}) of a single object image $f(x, y)$ were

$$\left\{ \begin{array}{lcl} \bar{x} & = & \frac{\sum \sum x f(x, y)}{\sum \sum f(x, y)} = \frac{1}{A} \sum_{(x,y) \in O} x \\ \bar{y} & = & \frac{\sum \sum y f(x, y)}{\sum \sum f(x, y)} = \frac{1}{A} \sum_{(x,y) \in O} y, \end{array} \right.$$

where O is the object, and A the area of the object. We have already seen how to compute the area from a chain code. The center of mass can be computed using a similar approach. In the case of computing \bar{x} , we only need to keep track of the current x coordinate (influenced by chain codes 0 and 2), and each time we move up (1) or down (3) we must subtract or add an appropriate value from an intermediate sum, ending up with $\sum x$.

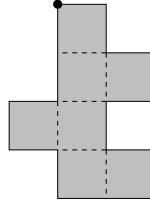
An algorithm to compute \bar{x} (relative to the x -coordinate of the starting position of the chain) from an outer boundary chain code is:

1. Set the variables *sum*, *x*, and *value* to zero.
2. Process each digit of the chain code:
 - if *digit* = 0 increase *x* by 1 and increase *value* by $(x - 1/2)$.
 - if *digit* = 1 decrease *sum* by *value*
 - if *digit* = 2 decrease *value* by $(x - 1/2)$ and decrease *x* by 1.
 - if *digit* = 3 increase *sum* by *value*
3. Divide *sum* by the object area to obtain \bar{x} .

The algorithm for computing \bar{y} is identical up to a rotation of the chain codes. The next example shows how this algorithm works.

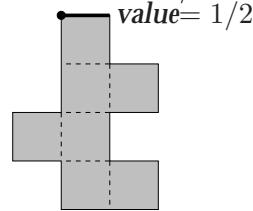
Example

Assuming we already have computed the area *A* of an object, what we still need is the sum of *x*-coordinates of all the object pixels, $\sum x$. Suppose we have this object:

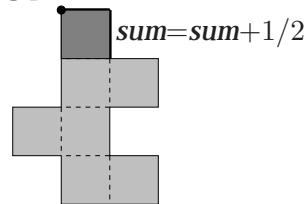


and its outer boundary chain code: {0, 3, 0, 3, 2, 3, 0, 3, 2, 2, 1, 2, 1, 0, 1, 1}. Following the algorithm step by step:

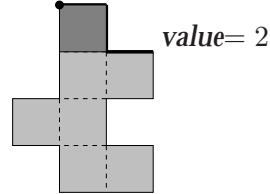
- *sum*, *x*, and *value* are set to zero.
- The first digit, 0 increases *x* to 1 and *value* to $1/2$.



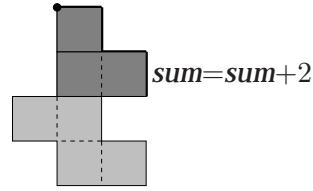
- The second digit, 3, increases *sum* by *value*. Note that this is exactly the contribution of the top pixel to $\sum x$; the *x*-coordinate of its center is $1/2$ (relative to the *x*-coordinate of the starting point).



- The next digit, 0, increases x to 2, and $value$ to 2.



- The next digit, 3, increases sum by $value$. Note that this is exactly the contribution of the two pixels in the current row to $\sum x$ (respectively $1/2$ and $3/2$).



- Continuing the algorithm in this fashion, we end up with $sum = 9/2$ when we reach the starting point again. The value \bar{x} therefore equals $sum/area = 9/14$.

With minor modifications (in the computation of $value$), this algorithm can be used to compute all object moments. Note that the computation of A , \bar{x} , and \bar{y} can be integrated into one 'pass', so the chain needs to be traversed only once.

Other measures. Theoretically, all object shape measures can be computed from the chain code of the object boundary. Some –such as compactness or roundness as defined in section 8.1– are now easy to compute because only the object perimeter and area are necessary for their computation. Other measures, that are closely tied to local boundary shape –such as curvature and bending energy– can also be computed from a chain code with relative ease. Measures that require more global information of the object however (which are most other measures from table 8.1), cannot easily be computed from a boundary representation such as a chain code. In these cases using a chain code is not computationally attractive (compared to using a region representation of the object), nor easy to use.

8.3.3 Boundary representation: distance from the center of mass

An object can be represented by a function that shows the distance of each boundary point to the object centroid. The variable of such a function is usually an angle. The resultant function is characteristic for the shape of the object. Figure 8.31 shows some example functions for simple objects.

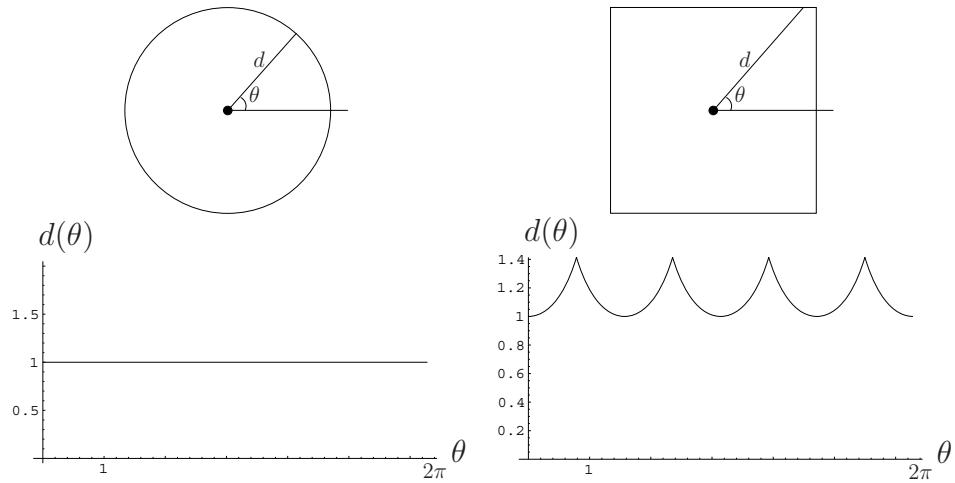
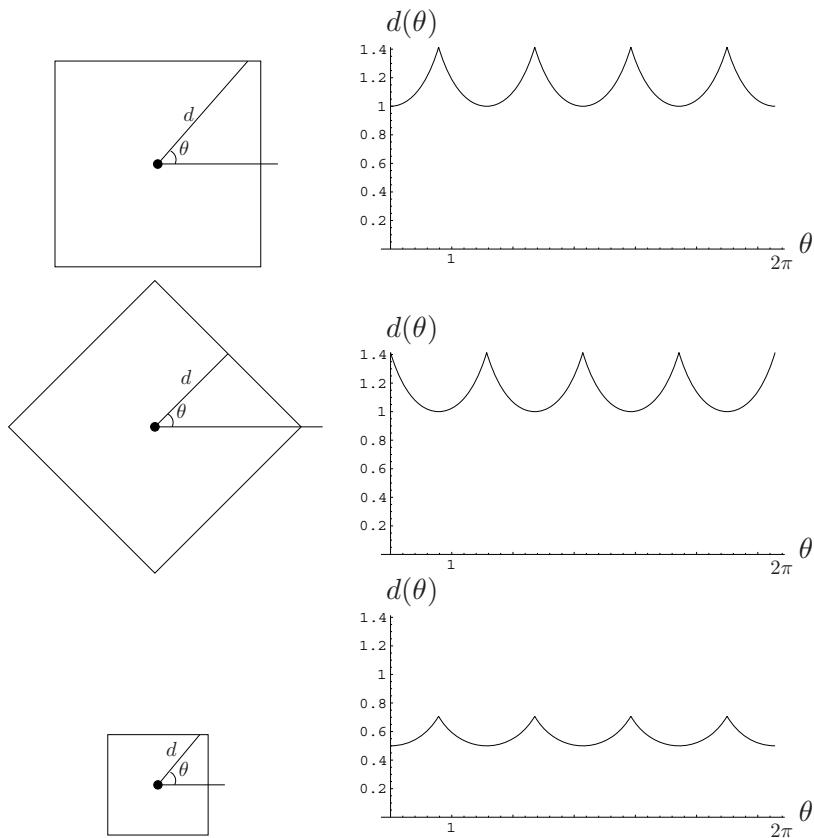


Figure 8.31 The distance d of boundary pixels to the object center of mass can be plotted as a function of the angle θ . Two example functions for a circle and a square are shown.

This distance representation is not very useful for most types of measurements, but it is excellently suited for comparing shapes; for evaluating the similarity between two objects, even when their orientation or scale is different. For example, the distance functions of two squares aligned in the same way but of different sizes, differs only by a scale factor. The distance functions of two squares of the same size that are rotated with respect to one another are the same up to a translation along the θ axis.

Example



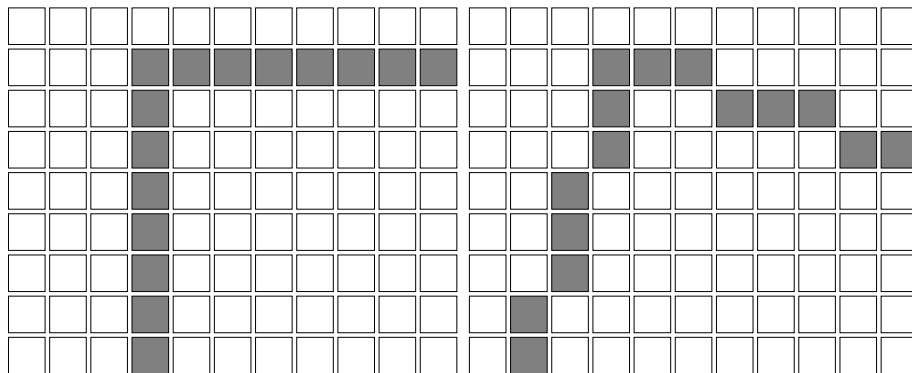
These three distance plots shows the effects of rotating and scaling an object. The first row shows the distance plot of a square. The second row shows the distance plot of the same square, rotated by $\pi/4$ radians. This results in a shifting of the distance plot along the θ axis by the same angle. The third row shows a scaled version of the square; both the x and y dimensions have been scaled by a factor $1/2$. The resultant values in the distance plot are then also scaled by a factor of $1/2$ compared to the original plot.

We can compensate for the difference in scale by normalizing the distance plot such that all values lie between 0 and 1. Applied to the three distance plots in the example above, this means that the first square and the scaled square will have identical distance plots. Compensating for the rotation can be done by shifting the distance plot until its global minimum is located at $\theta = 0$. This of course requires that a global minimum exists, which may not be the case. Assuming the distance plot has a global minimum, we are now able to determine if two objects have identical shape –regardless of their size or orientation– by normalizing and shifting their distance plots; if the objects are identical, the plots will be the same.

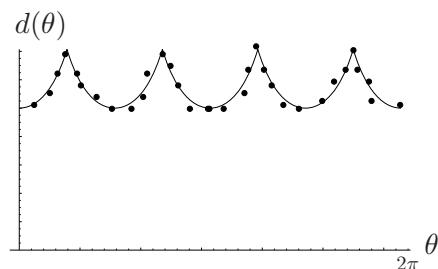
This last statement needs to be relaxed somewhat in the case of digital images: because of the pixelating process, the image of an object is dependent on its orientation and the

image resolution. The distance plot of a square and the distance plot of the same square rotated may therefore differ.

Example



Above are a zoom of a digital image showing the corner of a square and a zoom of the same square, slightly rotated. Because of the pixelation of the original square shape, the distance plot will not be ideal here. For example, the distance plot of a pixelated image –with the distance of each pixel represented by a dot and with the ideal plot overlaid– may look like:

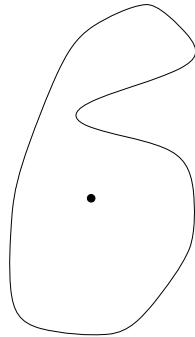


Increasing the resolution will make the difference between the distance plot and the ideal plot smaller.

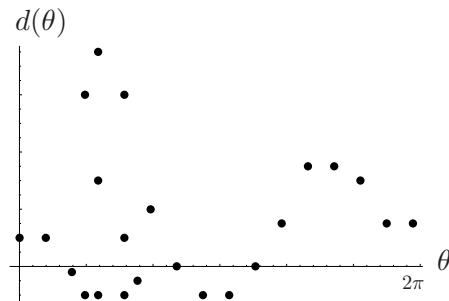
The distance plot may have multiple values at a single angle θ , as the next example demonstrates.

Example

This image:



has *three* distance values at angles between approximately 55 and 100 degrees (between $\frac{5}{16}\pi$ and $\frac{9}{16}\pi$). If we plot the distance values at a number of discrete angles, the plot looks like: (note that the θ axis does not correspond to $d = 0$)



8.3.4 Boundary representation: Fourier descriptors

The boundary pixels of an object can be presented as a list of coordinates $(x(k), y(k))$, where k runs from 0 to $(n - 1)$, with n being the number of boundary pixels. If we think of the image coordinate system as a complex coordinate system with the x -axis being the real axis and the y -axis being the imaginary axis, then we can represent each boundary pixel $(x(k), y(k))$ as a single complex number $u(k)$, with $u(k) = x(k) + iy(k)$. We can compute the discrete Fourier transform $U(v)$ of the sequence $u(k)$:

$$U(v) = \sum_{k=0}^{n-1} u(k) e^{-i2\pi \frac{kv}{n}},$$

where v also runs from 0 to $(n - 1)$. The (complex) numbers $U(v)$ are called the *Fourier descriptors* of the boundary. The Fourier descriptors share a number of properties with the distance-from-center-of-mass representation: both are not especially useful for direct object measurements, but are well suited for the comparison of objects. Like the

distance plot, the Fourier descriptors can be made independent of object scale and orientation. In addition, translation of an object will only have an effect on the descriptor $U(0)$.

The following properties hold:

name	boundary	Fourier descriptors
translation	$u(k) + t$	$U(v)$ if $v \neq 0$
scaling	$su(k)$	$sU(v)$
sequence order	$u(k - k_0)$	$U(v)e^{-i2\pi k_0 \frac{v}{n}}$
rotation	$u(k)e^{i\varphi}$	$U(v)e^{i\varphi}$

We can make the description independent of scaling by scaling the Fourier descriptors such that the maximum descriptor value magnitude equals 1. We can then compare two objects by comparing their Fourier descriptors, where we vary one list of descriptors by altering the rotation. If the objects have the same shape, there will be a specific rotation angle where the Fourier descriptor lists of the two objects are the same⁹, except for $U(0)$.

An advantage of using Fourier descriptors as a boundary representation over chain codes and distance-plots is that they are less sensitive to noise in the boundary. Fourier descriptors also make a very compact boundary description, because in many cases setting half (or even more) of the smaller descriptors to zero will not significantly alter the object represented. This means that in general a small fraction of the Fourier descriptors (the largest ones) already describes the object quite accurately. In practice, setting small Fourier descriptors to zero will often make comparing objects *more* robust rather than less.

8.3.5 Boundary representation: splines

The boundary of an object in a digital image will –since it is made up of square pixels– always be very ‘blocky’ when viewed from up close. This can be a nuisance in many applications, e.g., when measuring a quantity that expects the boundary to be smooth or to have a continuous first derivative. The visualization of objects also often suffers from this blocky nature. Visualizing boundaries that have been smoothed often produces much more natural and pleasing images to human observers.

An object boundary representation that is smooth and continuous, yet is compact and does not deviate much from the original boundary is a representation using *splines*. Splines are local polynomial expressions based on a small set of local points called *control points*. Many different expressions are possible, satisfying different smoothness and

⁹This is not the most efficient technique for comparing objects by their Fourier descriptors, but this one is easiest to understand. More efficient techniques tend to be much more mathematically complex.

adherence constraints with respect to the represented boundaries. In practice, *Bézier curves* and *B-splines* are used most often. For a thorough review of these splines we refer the reader to a computer graphics or CAD/CAM book. Here, we will suffice with some examples (see figure 8.32) showing smooth B-spline curves given a set of 'blocky' control points.

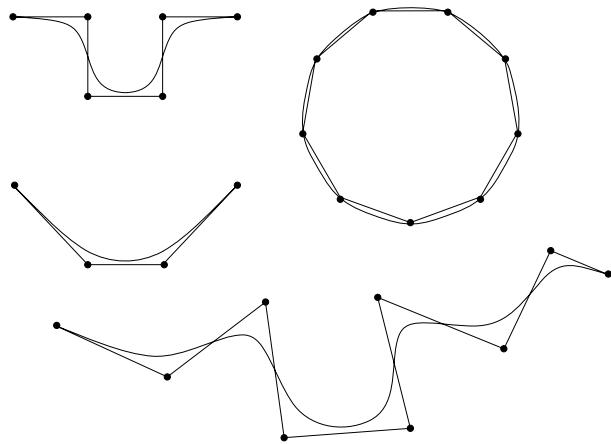


Figure 8.32 Some examples of B-spline curves given a set of control points (black dots).

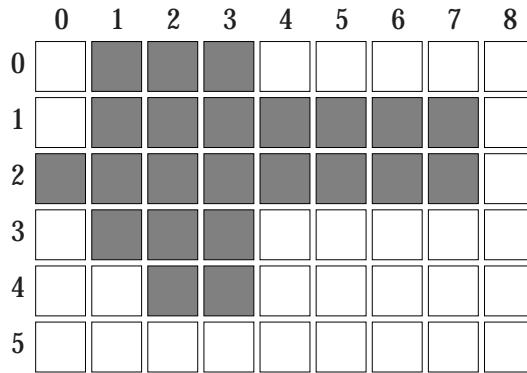
8.3.6 Region representation: run-length codes

Besides representing an object by describing its boundary, we can also represent it by describing the entire region it occupies. A common way to do this is to use *run-length* codes.

An object in a binary image can be considered as being made up out of a number of rows ('runs') of black pixels. All run-length encoding does is looking where these rows start, and how long they are.

Example

Consider this image:



In row 0, the run starts at pixel (1, 0) and runs (from left-to-right) for 3 pixels. This gives run-length code

(1, 0) 3.

In row 1, the run start at pixel (1, 1) and runs for 7 pixels. This gives run-length code

(1, 1) 7.

Continuing in this fashion, the run-length encoding of the object is

(1, 0) 3
(1, 1) 7
(0, 2) 8
(1, 3) 3
(2, 4) 2.

Run-length encoding is used in practice for some types of image compression. The method as presented here does not have a very high compression ratio when compared to other compression algorithms, but it is a very simple technique, can be performed fast, and does not require any complicated operations. Compressing requires only one sequential (left-to-right and top-to-bottom) pass over the image. For these reasons, run-length encoding is used in the transmission of facsimile messages. Another practical reason is that the receiving fax machine can start decompressing (and drawing) the image as soon as the first run-length code is transmitted. There is no need to wait for the entire image to be transmitted.

Much effort has been spent in compression of run-length codes to speed up transmission of images. Extra factors to consider here are the influence of noise during transmission (changing a few bits should not result in an entirely different image), that transmission can begin directly after the coder has started encoding the image, and that

decompression can begin directly after the first codes have been received. A simple but effective way for additional compression is to use short codewords for runs that occur frequently. Another is to not transmit the black runs and their starting points, but the alternating white and black pixel runs. For instance, the image in the example above starts with a white run of one pixel, then a black one of three pixels, then a white one of six pixels (running over to the next line), etc. The run-length encoding then is $\{1, 3, 6, 7, 1, 8, 2, 3, 7, 2, 14\}$. This encoding, together with the x -dimension of the encoded image completely represents this image.

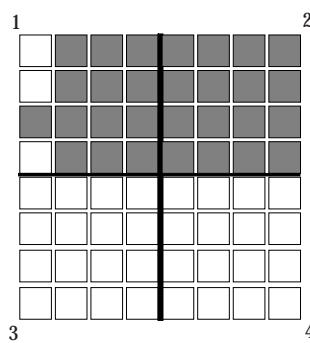
Although some measurements –such as area and moments– can easily be computed from run-length encoding, this representation is not used much for measurements, because the convenient ordering of boundary points present in boundary representations is missing.

8.3.7 Region representation: quadtrees

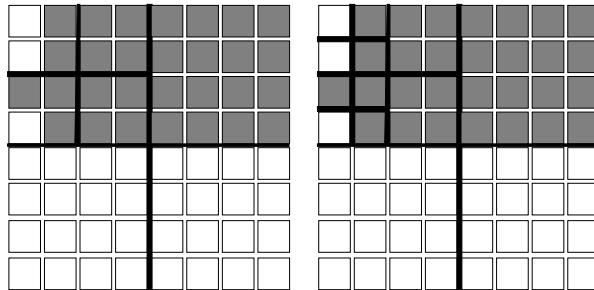
The quadtree representation divides the image into four quadrants and looks if each quadrant is uniformly white or black. If a quadrant is not uniform, it is itself subdivided into quadrants, and so on until all examined quadrants are uniform. The representation is called a quadtree because the result is commonly visualized as a tree graph.

Example

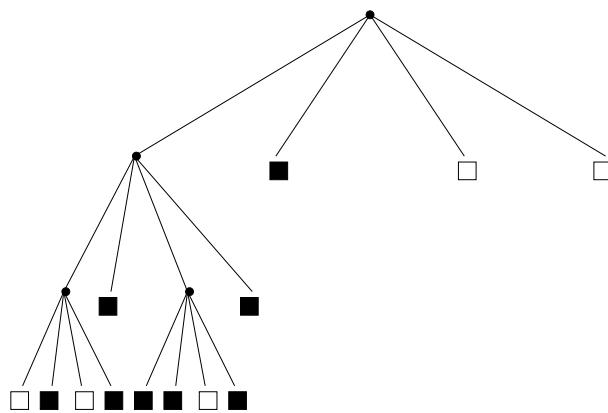
This shows an image and its initial subdivision into four quadrants:



After a second and third pass subdividing those image quadrants that are not uniformly black or white, we obtain these images:



We can represent this by this tree, where the four branches of a node correspond to quadrants ordered 1 to 4 as in the first image above:



As in the example above, each of the tree nodes can be black or white (denoting the entire quadrant is black or white), or it can branch into four new quadrants. We can decode the quadtree using a three letter code: *b* for black, *w* for white, and *g* (grey) denoting a branching node. Each *g* is followed by four letters describing its subtree. Describing the tree from the example above top-to-bottom and left-to-right in this way we obtain:

gggwbbgbbwbbww.

Since we only need a few bits to encode a *g*, *b* or *w*, the quadtree can very compactly describe the entire image.

Like the other region-based object representation presented, run-length coding, the quadtree representation is not very interesting for doing image measurements. It is however interesting as a tool for image compression, and has applications in image visualization. The latter is because it can easily be determined from a quadtree which image areas are uniform and can be visualized fast, and which image areas still require a more detailed visualization.

8.3.8 Other representations

In the previous sections, we have investigated some object representations that were principally based on the object boundary or object region. We have seen other representations, such as the skeleton with the quench function, that represent an object from its *medial* structure rather than the boundary or region. Intrinsically, there is no difference between boundary, medial or other representations because they are just different ways of representing the same object. It is only the viewpoint from which they were constructed that is different. Section 8.1.1 showed that for some types of measurement, medial object representations may have advantages over a boundary representation. In practice, the type of object representation used should be the one most suited to the image processing task at hand. Besides boundary, region, or medial representations, dedicated representations with a specific task in mind can be used. The variation in possibilities here is as large as the number of possible specific tasks, ranging from a representation using geometric primitives (common in CAD) to spiral decomposition, as depicted in figure 8.33.

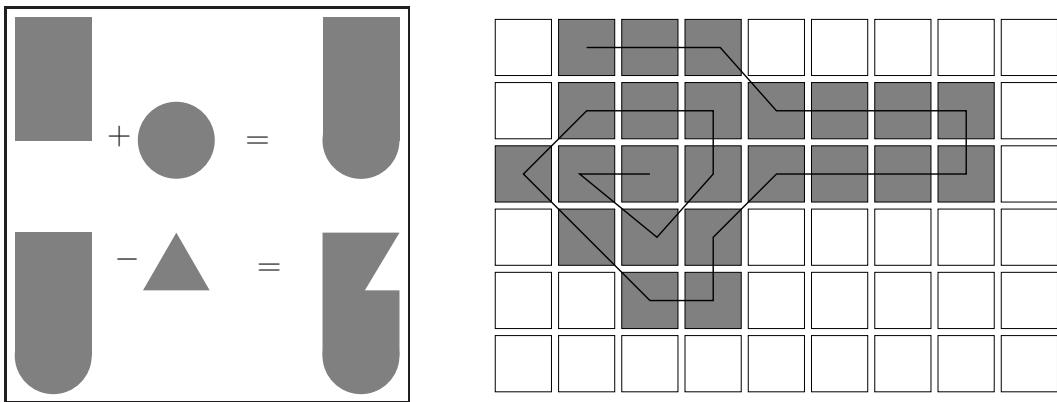


Figure 8.33 Two examples from a wide range of possible object representations. Left: The object viewed as a boolean combination of simple geometric primitives. Right: spiral decomposition.

Chapter 9

Scale space

Scale is one of the most important concepts in human vision. When we look at a scene, we instantaneously view its contents at multiple scale levels. For example, take a look at figure 9.1. This painting contains a wealth of objects 'living' at a wide range of scales, from small-scale objects such as the cracks in the pavement to large-scale structures such as the building in the background. Humans apparently are not easily confused by the occurrence of multiple scales in a single scene, because we have no trouble at all identifying what is building and what is not, despite the fact that the building consists of numerous details and components at lower scales.

Another vision phenomenon is that details (low-scale structures) simply seem to *disappear* if we move away from an object. For example, suppose you are facing the wall of a large brick building, close enough to distinguish individual bricks and the mortar between. If you move away from the building, the layers of mortar will disappear at a certain distance, giving the wall a uniform look. Instead of disappearing, details may also seem to *merge* under particular circumstances. For example, a row of shrubs may seem to be a single structure when seen from a large enough distance.

Including the notion of scale into computer vision and image processing techniques is no trivial matter. In the previous chapters, almost all of the techniques presented focused on small-scale image structures. The smallest scale in digital images is the scale of the individual pixels (the *inner scale* of the image), and our spatial operators mostly used discrete kernels with a size very close to this scale, such as 3×3 kernels, or continuous kernels with a small support. In this chapter, we will examine a technique to control the scale with which operators 'look' at a digital image.

9.1 Scale space

The key to adjusting an operator to a certain scale seems to be altering the image *resolution*. Put naively, we could re-sample a digital image at a low resolution to bring



Figure 9.1 Canaletto's *San Marco* square. An example of a scene with objects at multiple scales.

large-scale object into the range of, e.g., our 3×3 kernel. For an example, see figure 9.2, where we have decreased the resolution so that a 3×3 kernel operates on larger-scale structures than in the original image. The resolution of an image can be decreased by

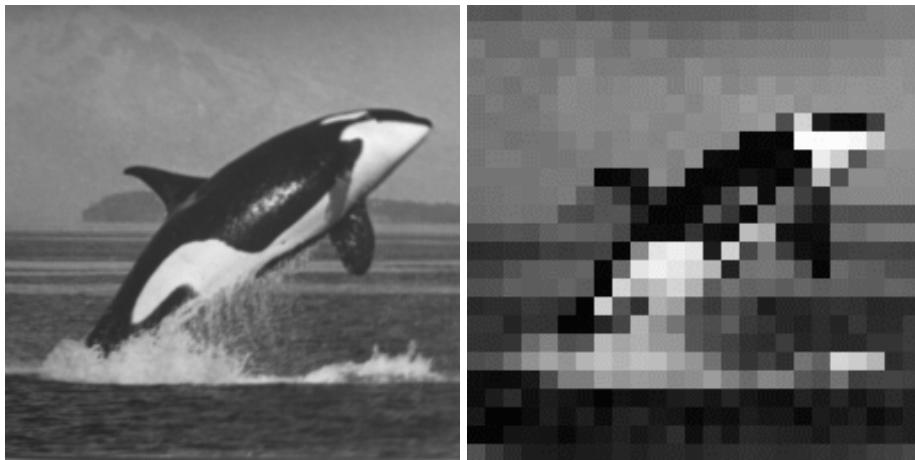


Figure 9.2 A 256×256 image, and the same image re-sampled to a resolution of 25×25 pixels using nearest neighbor interpolation. Using a 3×3 kernel on the right image now operates on structures of a much larger scale than in the original image. Notice that the re-sampling apparently introduces structures not present in the original. Consider for instance the shape and patterning of the orca's body as suggested by the low-resolution image, and compare it to the actual shape and pattern of the original.

merging the grey values of a block of pixels into the new grey value of a larger pixel. This merging can, e.g., be done by averaging grey values, by taking the center pixel value, or by taking the maximum, minimum, modal, nearest neighbor value, etc. This technique of lowering the resolution to bring large-scale structures into 'kernel range' and remove small-scale structures, is a much-used and easy to implement technique; we will come back to this technique in section 9.3. It however suffers from some problems that are inherent to the very blocky nature of representing low-resolution images by square pixels, the most serious of which is often the occurrence of *spurious resolution*: the apparent creation of structures that are not present in the original image. Spurious resolution can be seen in figure 9.2.

We are looking for an operation that lowers resolution but does not introduce new details into the image; that avoids spurious resolution. We have already come across such an operation many times: convolution with a Gaussian. Figure 9.3 shows example of resolution lowering by convolution with a Gaussian. Note that we use the term *resolution* here in a more abstract, intrinsic sense than before: upon convolution with a Gaussian, the actual number of pixels used in an image need not change, so the resolution in terms of the number of pixels used does not change. However, after convolution the smallest recognizable detail in the image has become larger, and in that sense the resolution *is* lower than before. We can also say that the intrinsic resolution of the image

had actually lowered, because it is possible to re-sample the image using fewer pixels than in the original *and retain all of the information contained in the image*. In practice, we say that Gaussian convolution lowers the image resolution, even if we represent the convolved image using the same number of pixels as with the original image. The

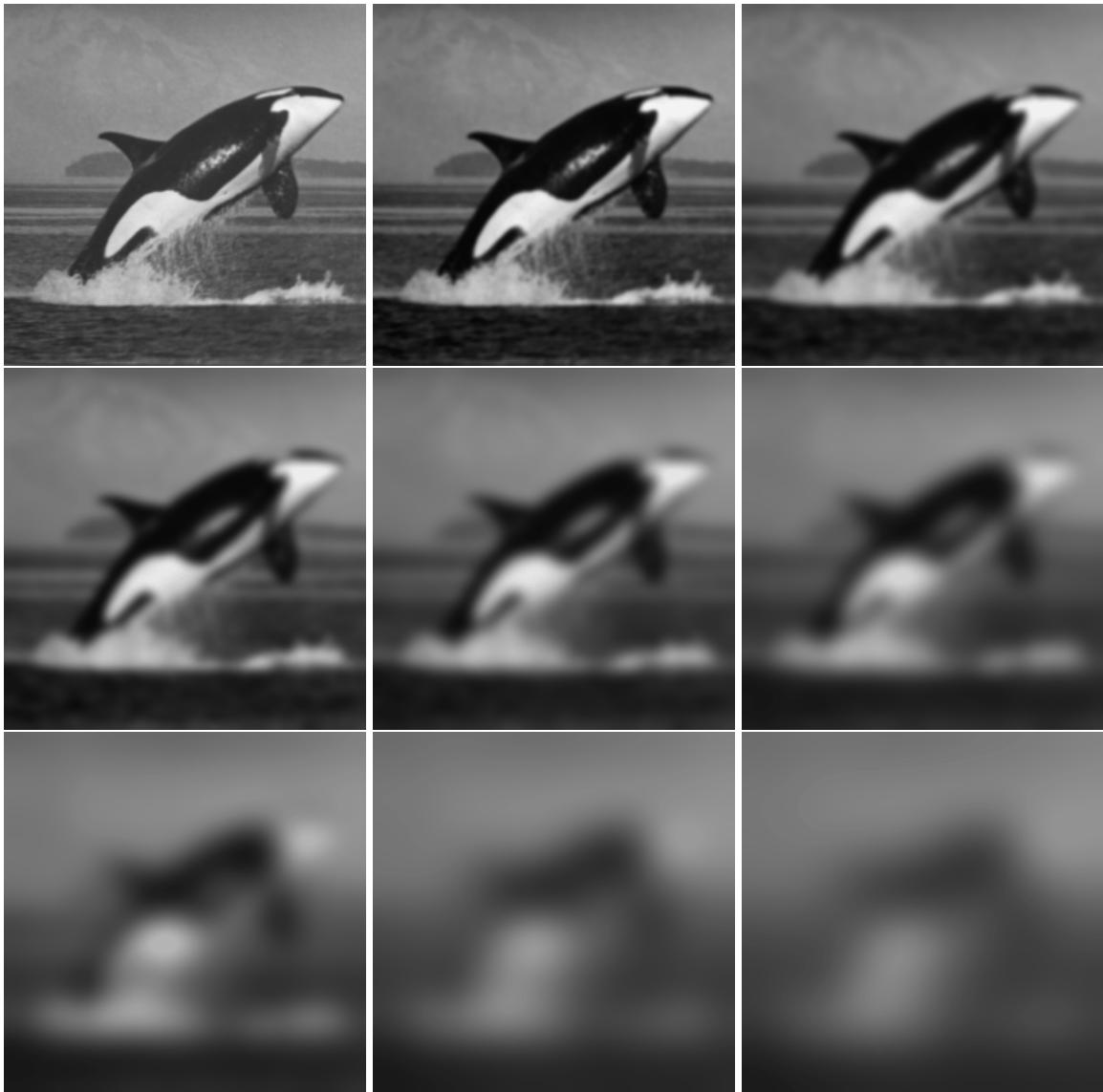


Figure 9.3 Lowering the resolution by Gaussian convolution: the top left image is convolved with a Gaussian kernel of increasing width (other images). Note that more and more details vanish from the images, and that effectively the resolution of the images is lowered.

Gaussian kernel is not the only convolution kernel that lowers resolution without creating spurious detail. But if we add the demand that we want to be able to combine two resolution lowering steps (*i.e.*, Gaussian convolution steps) into one single step, than the

Gaussian kernel is the only possible kernel. This demand makes intuitive sense: suppose we wish to lower the resolution by some amount a , and after that by some amount b , then it should be possible to do this in one step, lowering the resolution by an amount $a + b$.

The stack of images as seen in figure 9.3 is called a Gaussian *scale space*. We denote the Gaussian scale space of an image $f(x, y)$ by $f_\sigma(x, y)$, where σ is the standard deviation of the Gaussian kernel $g_\sigma(x, y)$ used in the convolution:

$$f_\sigma(x, y) = g_\sigma(x, y) * f(x, y).$$

The Gaussian scale space is called a one-parameter extension of the original image, because we now have three parameters (x , y , and σ) instead of two (x and y). The resolution parameter σ is called the *scale* of the image $f_\sigma(x, y)$.

The Gaussian scale space can be viewed as a stack of images, where the original image is at the bottom of the stack ($f_0(x, y) = f(x, y)$), and the image resolution gets lower as we rise in the stack, see figure 9.4.

With the Gaussian scale space, we now have a construct that allows us to ‘select’ an image at a certain level of resolution. The next question is how to use this construct so that we are able to apply an operator to an image only at a certain scale level σ_1 . In general, it is not very effective to apply such an operator directly to the image $f_{\sigma_1}(x, y)$. The reason for this is that –even though the intrinsic resolution of the image $f_{\sigma_1}(x, y)$ is lower than the resolution of the original¹– we still use the same number of pixels as in the original image to represent the image at scale σ_1 . Subsampling the image so that we use less pixels than in the original image can partially solve this problem. However, there is a method that solves this problem in a mathematically sound way when the operator we wish to apply is a differential operator. This method is covered in the next section.

9.1.1 Scaled differential operators

Given an image f_{σ_1} ; an image at scale σ_1 : $f_{\sigma_1} = g_{\sigma_1} * f$. Suppose we wish to compute the derivative of the image f_{σ_1} to x , i.e., the derivative of f to x at scale σ_1 . Consider this equation:

$$\frac{\partial f_{\sigma_1}}{\partial x} = \frac{\partial}{\partial x}(g_{\sigma_1} * f) \xrightarrow{\mathcal{F}} 2\pi i u(G_{\sigma_1} F) = (2\pi i u G_{\sigma_1}) F \xrightarrow{\mathcal{F}^{-1}} \frac{\partial g_{\sigma_1}}{\partial x} * f.$$

¹Assuming that $\sigma_1 \neq 0$.

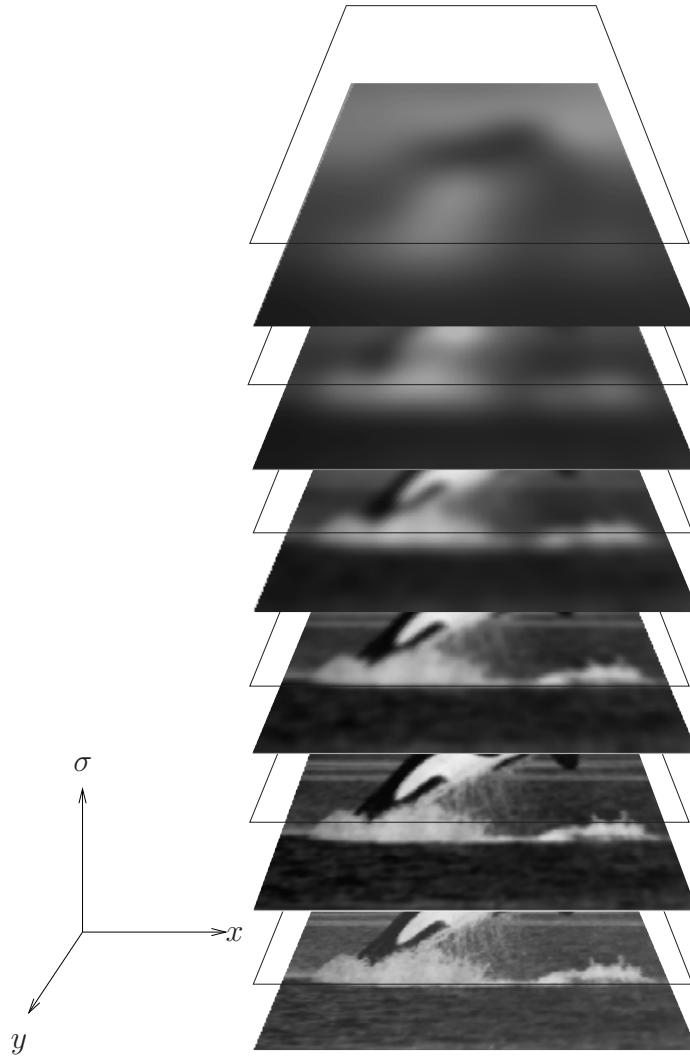


Figure 9.4 The Gaussian scale space of an image. The original image rests at the bottom of the stack ($\sigma = 0$), and the scale σ increases as we rise in the stack.

Here, \mathcal{F} and \mathcal{F}^{-1} denote the Fourier and inverse Fourier transform. In this equation, we have used the properties that convolution in the spatial domain becomes multiplication in the Fourier domain, and that taking the derivative to x in the spatial domain becomes multiplication with $2\pi i u$ in the Fourier domain. Cutting out the middle part of the equation gives us:

$$\frac{\partial f_{\sigma_1}}{\partial x} = \frac{\partial g_{\sigma_1}}{\partial x} * f.$$

The left term shows us what we wish to compute: the scaled (σ_1) derivative of f to x . The right term shows us how we can compute this in a well-posed mathematical way:

we compute the derivative *of the Gaussian kernel* and convolve the result with our original image f . This way, we avoid having to compute the derivative of the digital image f , which is an ill-posed problem. Instead, we only need to compute the derivative of a Gaussian, which is a well-posed problem, because the Gaussian function is a differentiable function. If we compute the scaled derivative entirely in the Fourier domain, only multiplications are needed.

Figure 9.5 shows some examples of scaled x derivatives of an image.

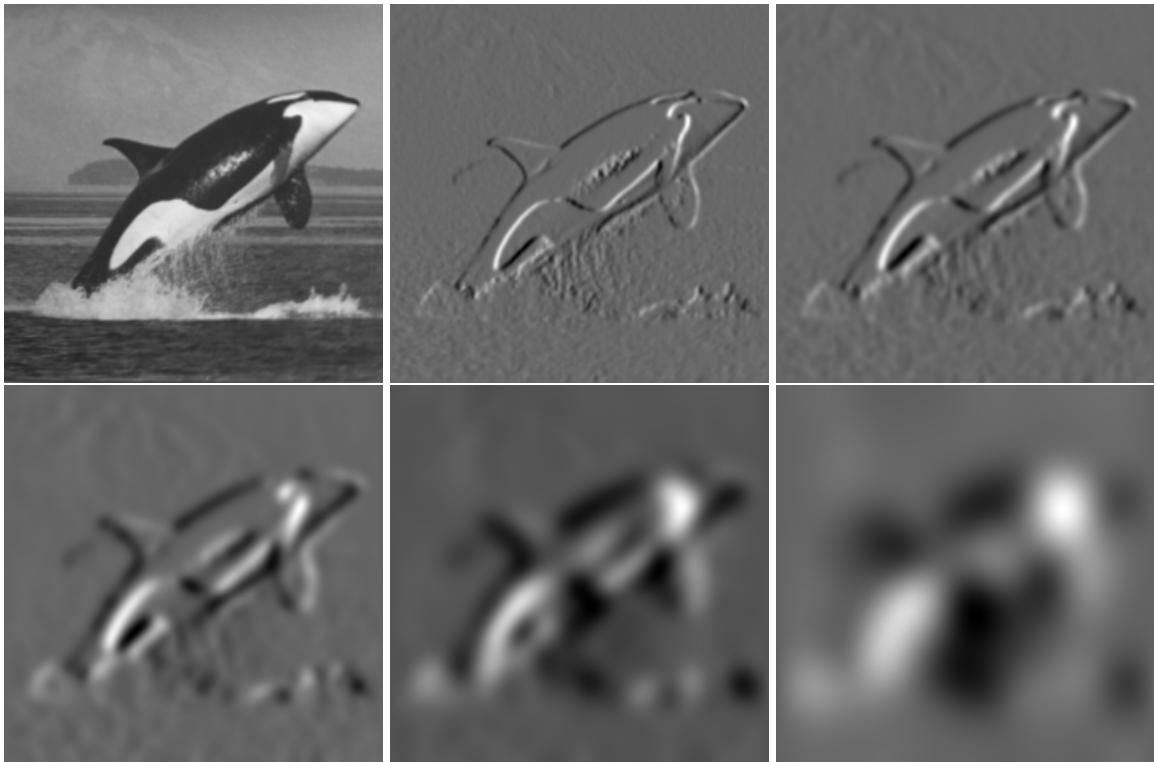


Figure 9.5 Scaled x derivatives of the top-left (256×256) image, respectively with $\sigma = 1, 2, 4, 8, 16$ pixels.

This example with the scaled x derivative of an image can be generalized to all spatial derivatives: to compute a scaled spatial derivative, we need only compute the appropriate derivative of the Gaussian kernel (with standard deviation σ equal to the desired scale), and convolve the result with the original image. So, if we denote partial derivatives by subscripts x or y , for example:

$$\begin{aligned} f_{\sigma_1,x} &= g_{\sigma_1,x} * f \\ f_{\sigma_1,y} &= g_{\sigma_1,y} * f \\ f_{\sigma_1,xy} &= g_{\sigma_1,xy} * f \\ f_{\sigma_1,xxx} &= g_{\sigma_1,xxx} * f \\ \text{etc.} \end{aligned}$$

9.1.2 Scaled differential features

Many image features, such as edges, ridges, corners, etc. can be detected using differential operators. In many cases, the detection of interesting features benefits from tuning the differential operators to the appropriate scale.

Invariants. In many applications, features like the x or y derivative of an image are not by themselves interesting features. More interesting are so-called *invariant* features –also concisely denoted simply by *invariants*– i.e., features that are insensitive to some kind of transformation. For example, the x derivative of an image is not invariant under rotation, because the following two operations give different results: (1) computing the x derivative, and (2) rotating the image, computing the x derivative, then rotating the result back. The gradient norm ($\sqrt{f_x^2(a) + f_y^2(a)}$, in a point a), however, is invariant under rotation because the two operations: (1) computing the gradient norm, and (2) rotating the image, computing the gradient norm, then rotating the result back, yields the same result. All of the features treated in the below are invariant under translations and rotations.

Gradient-based coordinate system. The gradient vector plays an important role in the detection of many different types of features. It is therefore often convenient to abandon the usual (x, y) Cartesian coordinate frame, and replace it with a *local* coordinate frame based on the gradient vector w and its right-handed normal vector v . This local (v, w) frame is sometimes called a *gauge* or *Frenet* frame. In formulas, the v and w vectors can be written as

$$w = \begin{pmatrix} f_x \\ f_y \end{pmatrix} \quad \text{and} \quad v = \begin{pmatrix} f_y \\ -f_x \end{pmatrix}.$$

Figure 9.6 shows an example of an image where we have drawn several (v, w) frames. Note that the gradient vector w always points ‘uphill’, and that the normal vector v is always in the direction of an isophote (i.e., a curve of equal image intensity). It may at first seem strange to use a coordinate system that may point a different way in each pixel, but it is in fact very convenient when we consider features invariant under rotations (and translations). This is because when we rotate an image, the (v, w) frames *rotate with it*, because the v and w vectors are based on the local image structure. This means that all derivatives computed in the v or w directions are also invariant. Put differently, the (v, w) frame captures some of the intrinsic differential structure of the image, independent of how you have chosen your (x, y) coordinate system.

As we denote the derivative of an image in the x direction by f_x , we adopt a similar notation for derivatives in the v or w direction. For example, the first derivative of f in the w direction is denoted by f_w , the second by f_{ww} , etc.

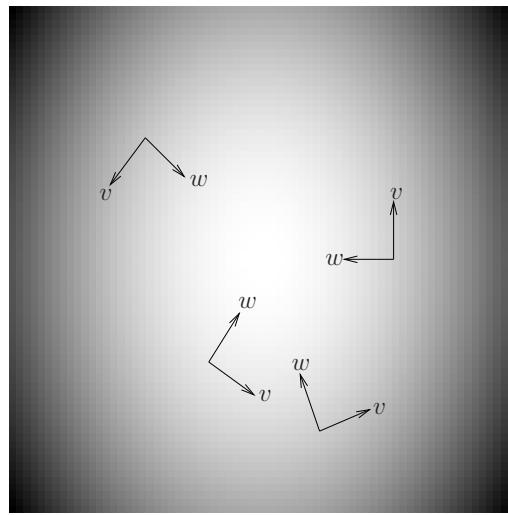


Figure 9.6 The (v, w) coordinate frame drawn at four random points in an image. The w vector equals the local gradient.

For the actual computation of expressions such as f_w , f_{vw} , f_{www} , etc. we usually need to rewrite them in a Cartesian form (using f_x , f_y , f_{xy} , etc. which we know how to compute). The intermezzo below shows how this ‘translation’ can be achieved.

Intermezzo*

Derivatives of a function f specified using v and w subscripting can be translated to a Cartesian form (using only partial derivatives to x and y) by using the definition of the *directional derivative*:

The derivative of f in the direction of a certain row vector a is defined as

$$f_a = \frac{1}{\|a\|} (a \cdot \nabla) f,$$

where ∇ is the Nabla vector defined by $\nabla = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix}$, and the dot (\cdot) denotes the inner product.

For example, if we take a to be the vector $(1, 2)$, then f_a equals:

$$\begin{aligned} f_a &= \frac{1}{\sqrt{5}} \left((1 \ 2) \cdot \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} \right) f = \frac{1}{\sqrt{5}} \left(\frac{\partial}{\partial x} + 2 \frac{\partial}{\partial y} \right) f = \frac{1}{\sqrt{5}} \left(\frac{\partial f}{\partial x} + 2 \frac{\partial f}{\partial y} \right) = \\ &= \frac{1}{\sqrt{5}} (f_x + 2f_y). \end{aligned}$$

Note that this directional derivative definition is consistent with the definition of the Cartesian derivatives. For example, f_x , i.e., the directional derivative of f in the direction $(1, 0)$:

$$f_x = \frac{1}{1} \left((1 \ 0) \cdot \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} \right) f = \left(\frac{\partial}{\partial x} \right) f = \frac{\partial f}{\partial x}$$

We can now compute the derivatives f_v and f_w :²

$$f_w = \frac{1}{\sqrt{f_x^2 + f_y^2}} \left((f_x \ f_y) \cdot \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} \right) f = \frac{f_x f_x + f_y f_y}{\sqrt{f_x^2 + f_y^2}} = \sqrt{f_x^2 + f_y^2}.$$

$$f_v = \frac{1}{\sqrt{f_x^2 + f_y^2}} \left((f_y \ -f_x) \cdot \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} \right) f = \frac{(f_y f_x - f_x f_y)}{\sqrt{f_x^2 + f_y^2}} = 0.$$

Perhaps not surprisingly, the derivative f_w equals the gradient magnitude, and f_v is zero. The latter is because v is in the direction of the local isophote, i.e., the curve where f does not change value.

Higher order directional derivatives can be computed using the following extension:

$$f \underbrace{a \dots a}_{n \text{ times}} = \frac{1}{||a||^n} (a \cdot \nabla)^n f.$$

For example, the Cartesian form of f_{vv} is

$$\begin{aligned} f_{vv} &= \frac{1}{||v||^2} \left((f_y \ -f_x) \cdot \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix} \right)^2 f \\ &= \frac{1}{f_x^2 + f_y^2} \left(f_y \frac{\partial}{\partial x} - f_x \frac{\partial}{\partial y} \right)^2 f \\ &= \frac{1}{f_x^2 + f_y^2} \left(f_y^2 \frac{\partial^2}{\partial x^2} - 2f_x f_y \frac{\partial^2}{\partial x \partial y} + f_x^2 \frac{\partial^2}{\partial y^2} \right) f \\ &= \frac{f_y^2 f_{xx} - 2f_x f_y f_{xy} + f_x^2 f_{yy}}{f_x^2 + f_y^2}. \end{aligned}$$

²Note that we compute the directional derivative along a straight line; the derivative along the v or w vector. We do not compute the derivative along the integral curves of the local v or w vectors. The latter form gives distinctly different results. Also, note that in our expressions we take the derivatives f_x and f_y to be constants, so the differential operators affect only f itself.

Edges. In chapter 5 we introduced the gradient $\nabla f(b)$ in a certain point b as a vector that always points in the most uphill direction if we view the image as an intensity landscape. The gradient is also perpendicular to the local isophote. The norm of the gradient $\|\nabla f(b)\| = \sqrt{f_x^2(b) + f_y^2(b)}$ is a measure for the edgeness in b .

Another edgeness measure is based on the second derivatives of f : the Laplacian, $\Delta f = f_{xx} + f_{yy}$. In this case, edges are assumed to be located at the zero crossings of the measure. Both the gradient norm and the Laplacian are computed using first or second order partial derivatives of the image f . We can now compute these derivatives in a scaled way, using a certain scale σ . The figures 9.7 and 9.8 show some examples of scaled edgeness images. Note the results of high-scale operators in noisy environments in figure 9.8.

Ridges. If we view an image as an intensity landscape, one of the most characteristic features are the landscape's *ridges*. The definition of a ridge in terms of image derivatives still sparks some debate, apparently because different people usually have different ideas as to what exactly the ridges in an image are³. In fact, many different definitions can be found, each defining similar yet different ridge-like structures in an image.

One ridge definition that appeals to the intuition is one based on the (v, w) gradient-based coordinate frame: figure 9.9 shows part of an image depicted as an intensity landscape. At a ridge point (1), and two other randomly chosen points, we have drawn the gradient vector w . If we plot the grey values around the ridge point in the v and the w direction (left two plots of the four), then only the v direction plot shows a concave profile. This is because the v direction is the direction crossing the ridge. The same plots at a non-ridge point do not show any significant concavity. This is the basis of our definition for ridgeness: the concavity of the grey-value profile in the v direction. Since the second derivative is a measure for concavity, our ridgeness measure is simply f_{vv} . For the Cartesian form of this measure see the intermezzo above. Figure 9.10 shows some examples of f_{vv} images at different scales. The measure not only detects ridges (low f_{vv} values), but also ‘valleys’ (high f_{vv} values), see figure 9.11. Figure 9.9 can also be used to define another ridgeness operator: In any non-ridge point, it can be observed that the local gradient is pointing roughly toward the ridge. In a ridge point, the gradient is directed *along* the ridge. This means that if we travel from the left point to the right one via the ridge point, the gradient will turn swiftly as we cross the ridge. The gradient turning speed when traveling in the v direction through an image may therefore be another ridgeness operator. We can measure this turning speed as follows: the orientation of the gradient can be characterized by an angle θ with the positive x -axis, i.e., $\theta = \arctan(f_y/f_x)$. Its rate of change (turning speed of the gradient) equals its first derivative, here taken in the v direction. This results in the expression $\frac{1}{\|v\|}(v \cdot \nabla)\theta$, which

³Unfortunately, this holds true to some extent for most types of image features.

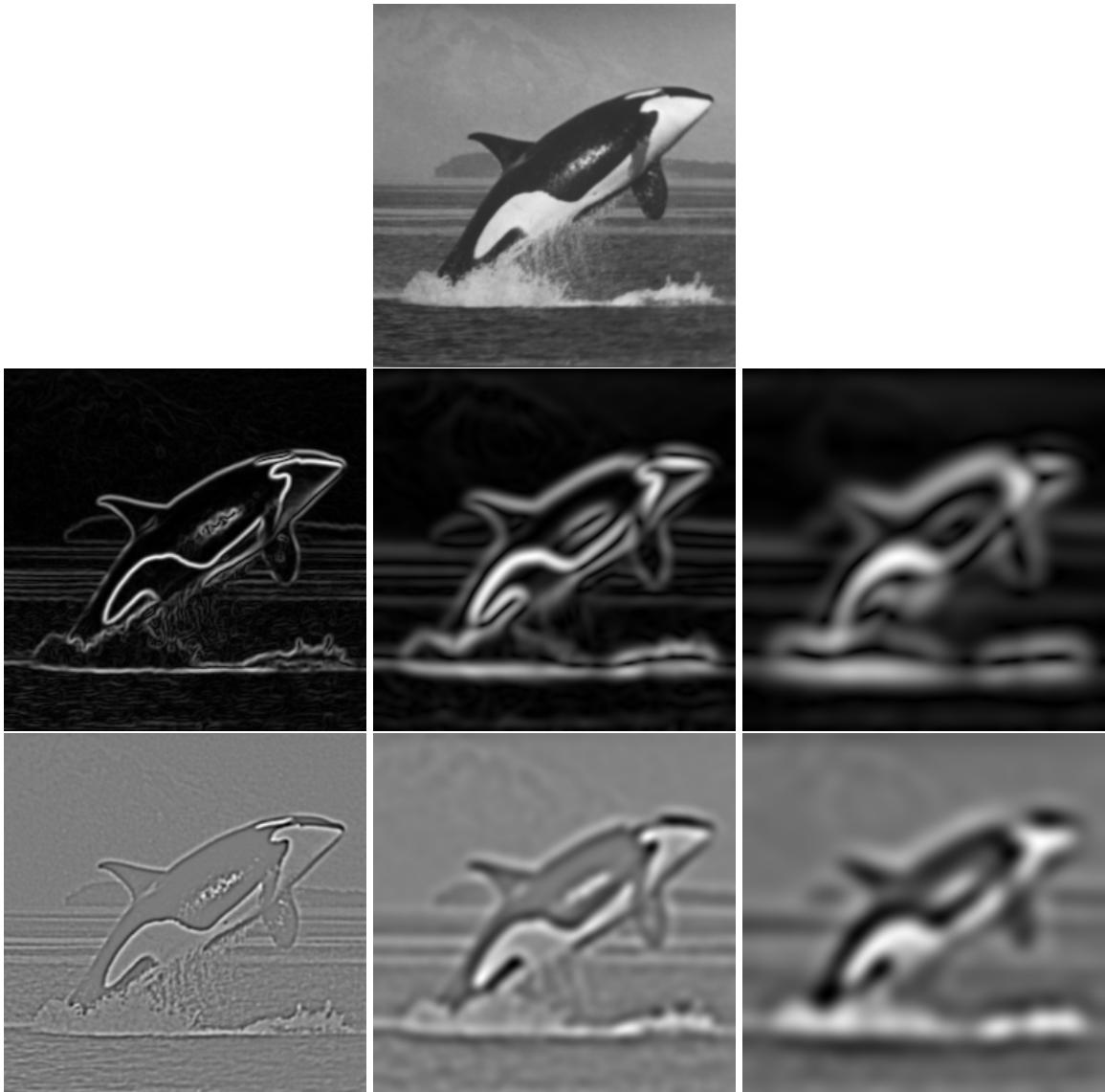


Figure 9.7 Edgeness images at different scales. Top row: original, 256×256 image. Middle row: gradient norm at scales 1, 4 and 8 pixels. Bottom row: Laplacian images at scales 1, 4, and 8 pixels. Note that details vanish from the edgeness images as the scale increases. However, the localization certainty decreases with increasing scale; the ‘resolution’ of the edge is lowered.

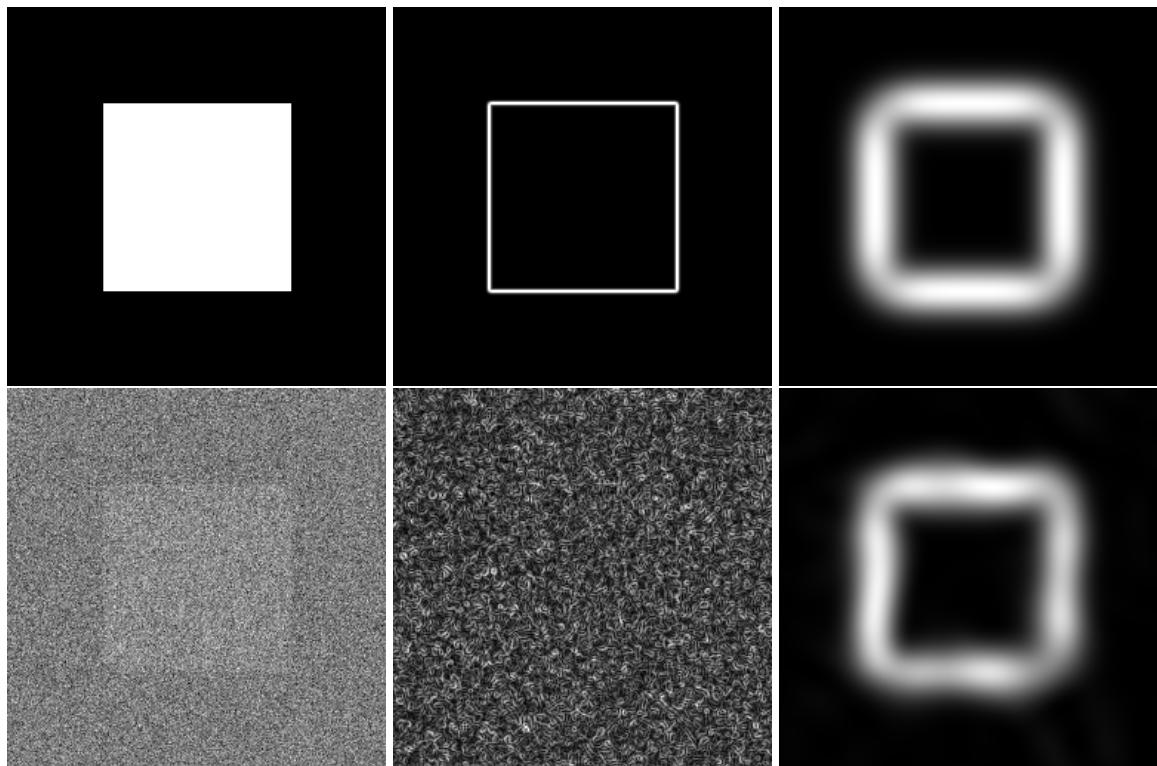


Figure 9.8 Gradient norm edgeness images of a 256×256 artificial image (top left), and of the same image with added noise (bottom left). The middle and right column show the edgeness images at scales of 1 and 16 pixels respectively. Note that only the 16 pixel scale performs well in the noisy image.

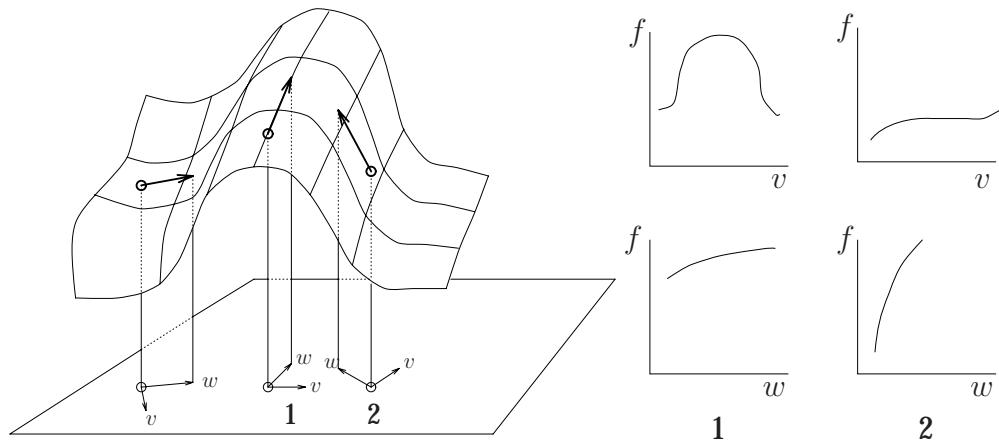


Figure 9.9 The grey-value profiles in the v and w (gradient) directions at a ridge point (1), and a non-ridge point (2). Note that only the v -profile in the ridge point is markedly concave.

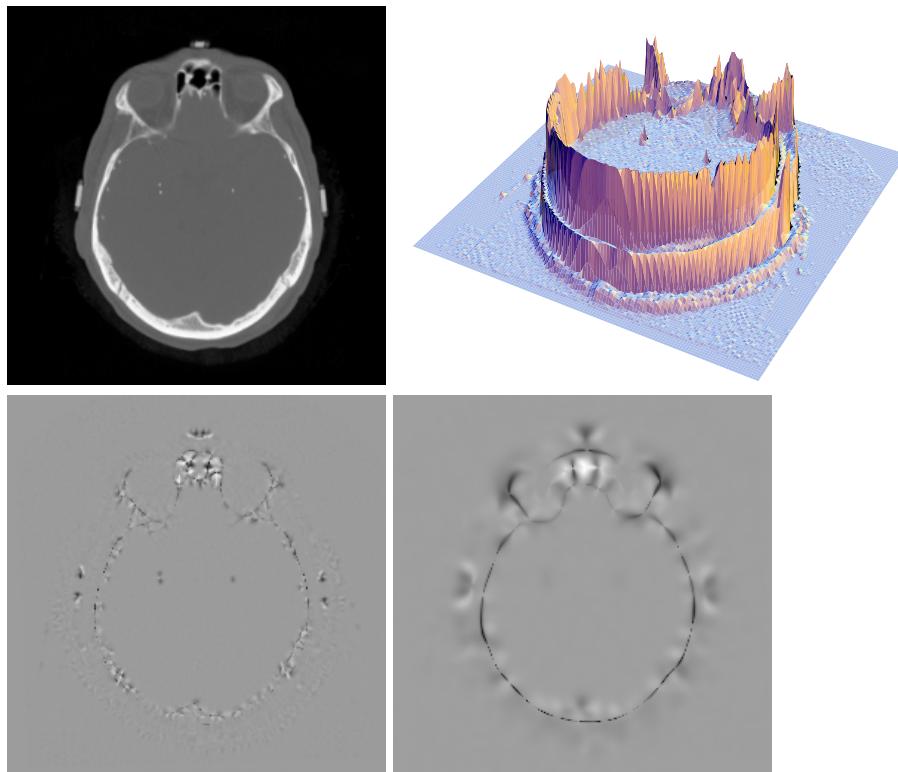


Figure 9.10 The top row shows an original image (256×256 pixels) and its landscape version. The landscape image shows that a roughly circular ridge is present in the image. However, the ridge is locally very jagged, so the ridgeness operator f_{vv} will not show much at a low scale of one pixel (bottom left). At a scale of four pixels, the f_{vv} image nicely shows the ridge curve.

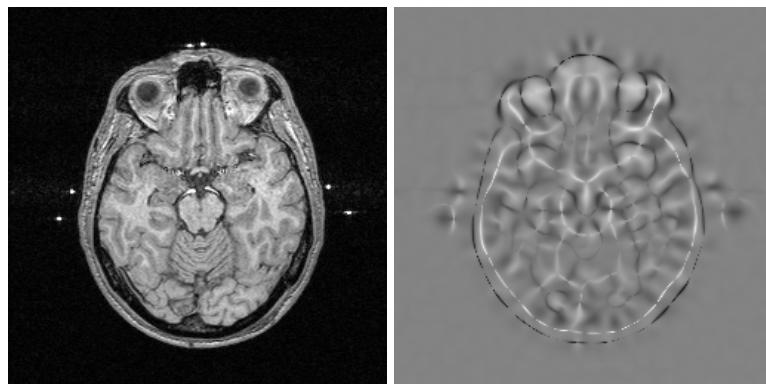


Figure 9.11 Example of the f_{vv} ridgeness operator. Left: original image (256×256 pixels). Right: f_{vv} image at a scale of 4 pixels. Note that ridges in the original form dark curves, whereas valleys from bright curves.

can be worked out in the same manner as before:

$$\frac{1}{\|v\|}(v \cdot \nabla)\theta = \frac{2f_x f_y f_{xy} - f_y^2 f_{xx} - f_x^2 f_{yy}}{(f_x^2 + f_y^2)^{\frac{3}{2}}}.$$

It turns out this expression is very similar to the f_{vv} ridgeness operator. In fact, it equals $-\frac{f_{vv}}{f_w}$, so the only difference is the minus sign and a normalization with respect to the gradient magnitude f_w . This latter aspect makes that the $\frac{f_{vv}}{f_w}$ responds more in low-contrast image areas than f_{vv} . The measure $\frac{f_{vv}}{f_w}$ is also known in literature as the *isophote curvature*. Figure 9.12 shows some examples of $-\frac{f_{vv}}{f_w}$ images.

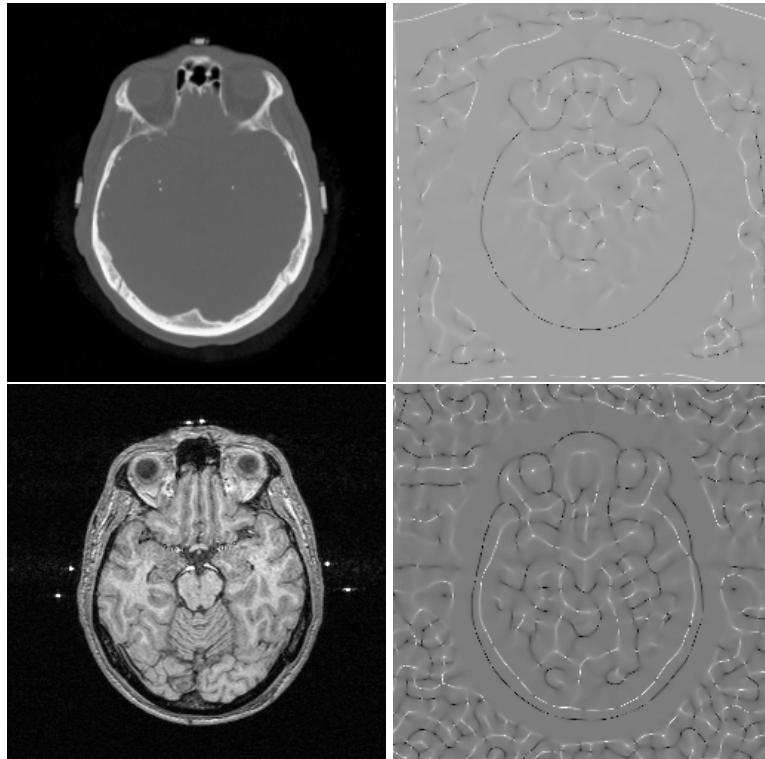


Figure 9.12 Examples of the $\frac{f_{vv}}{f_w}$ ridgeness operator. Left column: original 256×256 images. Right column: $\frac{f_{vv}}{f_w}$ images at a scale of four pixels. Note that the operator also responds in low contrast image areas.

Other differential geometric structures. It would be beyond the scope of this book to treat in detail other differential geometric structures. It is however illustrative to show some examples of other operators, to give some idea of their capabilities in relation to image processing. Table 9.1 lists a number of operators (all invariant under rotation and translation). Note that names for expressions (such as *cornerness*) are descriptive at best, and should not be taken in too literal a sense. The figures 9.13 through 9.16 show some example images.

Name (expression)	Cartesian form
Edgeness, f_w	$\sqrt{f_x^2 + f_y^2}$
Ridgeness, f_{vv}	$\frac{f_y^2 f_{xx} - 2f_x f_y f_{xy} + f_x^2 f_{yy}}{f_x^2 + f_y^2}$
Isophote curvature; ridgeness, $-\frac{f_{vv}}{f_w}$	$-\frac{f_y^2 f_{xx} - 2f_x f_y f_{xy} + f_x^2 f_{yy}}{(f_x^2 + f_y^2)^{\frac{3}{2}}}$
Cornerness, $f_{vv} f_w^2$	$f_y^2 f_{xx} - 2f_x f_y f_{xy} + f_x^2 f_{yy}$
Flowline curvature, $-\frac{f_{vw}}{f_w}$	$\frac{f_x f_y (f_{yy} - f_{xx}) + f_{xy} (f_x^2 - f_y^2)}{(f_x^2 + f_y^2)^{\frac{3}{2}}}$
Isophote density, f_{ww} (or $\frac{f_{ww}}{f_w}$)	$\frac{f_{xx} f_x^2 + 2f_{xy} f_x f_y + f_{yy} f_y^2}{f_x^2 + f_y^2}$
Umbilicity	$\frac{2(f_{xx} f_{yy} - f_{xy}^2)}{f_{xx}^2 + 2f_{xy}^2 + f_{yy}^2}$
Unflatness	$f_{xx}^2 + 2f_{xy}^2 + f_{yy}^2$
Checkerboard detector	$-(f_{xxxx} f_{yyyy} - 4f_{xxxx} f_{xyyy} + 3f_{xxyy} f_{xxyy})^3 + 27(-f_{xxxx} (f_{xxxx} f_{yyyy} - f_{xyyy} f_{xxyy}) + f_{xxxx} (f_{xxyy} f_{yyyy} - f_{xyyy} f_{xxyy}))^2$

Table 9.1 Examples of differential invariants.

9.2 Scale space and diffusion*

The physical processes of diffusion and heat conduction offer a new view of scale space. At a first glance, these processes and scale space appear to be unrelated, but consider figure 9.17. Suppose the left graph represents the temperature as a function of length along a metal rod. After a certain amount of time, the temperature distribution will have evolved to the state represented by the right graph. As you may have guessed from the shape of the graphs, the right graph is nothing but the left graph convolved with a Gaussian. The temperature distribution at any time after the initial state can be computed by convolving the initial distribution with a Gaussian, where the width σ is directly related to the time a certain state occurs. Figure 9.18 shows some evolution states. The collection of all evolution states equals the scale space of the original state.

The physical process of heat conduction is usually modeled by the following partial differential equation:

$$f_t = \alpha^2 f_{xx},$$

where $f = f(x, t)$ represents the temperature at position x at time t , and α is a material-dependent constant. This equation is known as the *heat* or *diffusion* equation (it can be used as a model for either process). It models how the temperature distribution evolves: the change in time (f_t) depends only on the current temperature distribution, and some

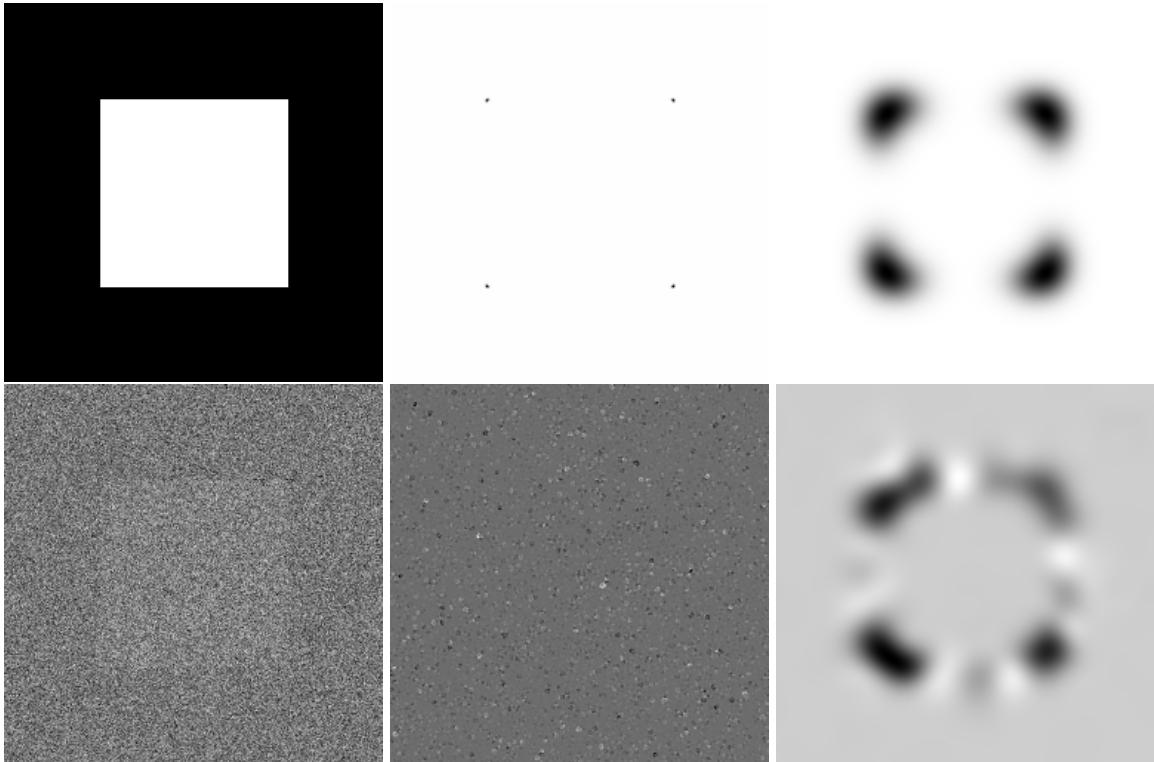


Figure 9.13 Top row: original (256×256) image, and two cornerness images at scales of 1 and 16 pixels. Bottom row: same original image, with added noise, and the cornerness images at the same two scales. Note that only the large scale can cope with the (small scale) noise.

nifty modeling that we won't report here shows that the change in time is proportional to the second spatial derivative of f , as the equation shows. The two-dimensional form of this equation is

$$f_t = \alpha^2 \Delta f,$$

where Δf equals the Laplacian $\Delta f = \nabla \cdot (\nabla f) = f_{xx} + f_{yy}$. If we assume the function f has an infinite spatial domain, it can be shown that a solution $f(x, \tau)$ at a certain time τ can be found by convolving the initial state $f(x, 0)$ with a Gaussian function with width $\sigma = \sqrt{2\alpha^2 t}$.

In terms of images, this means that we can regard a scaled image $f_\sigma(x, y)$ as the solution of the heat equation at a time related to σ , where the original image $f = f_0(x, y)$ is the initial 'temperature distribution', if we think of a grey value as a temperature.

The heat conduction model can be extended to include some common phenomena. For example, in practice we may need to model the heat conduction in situations where

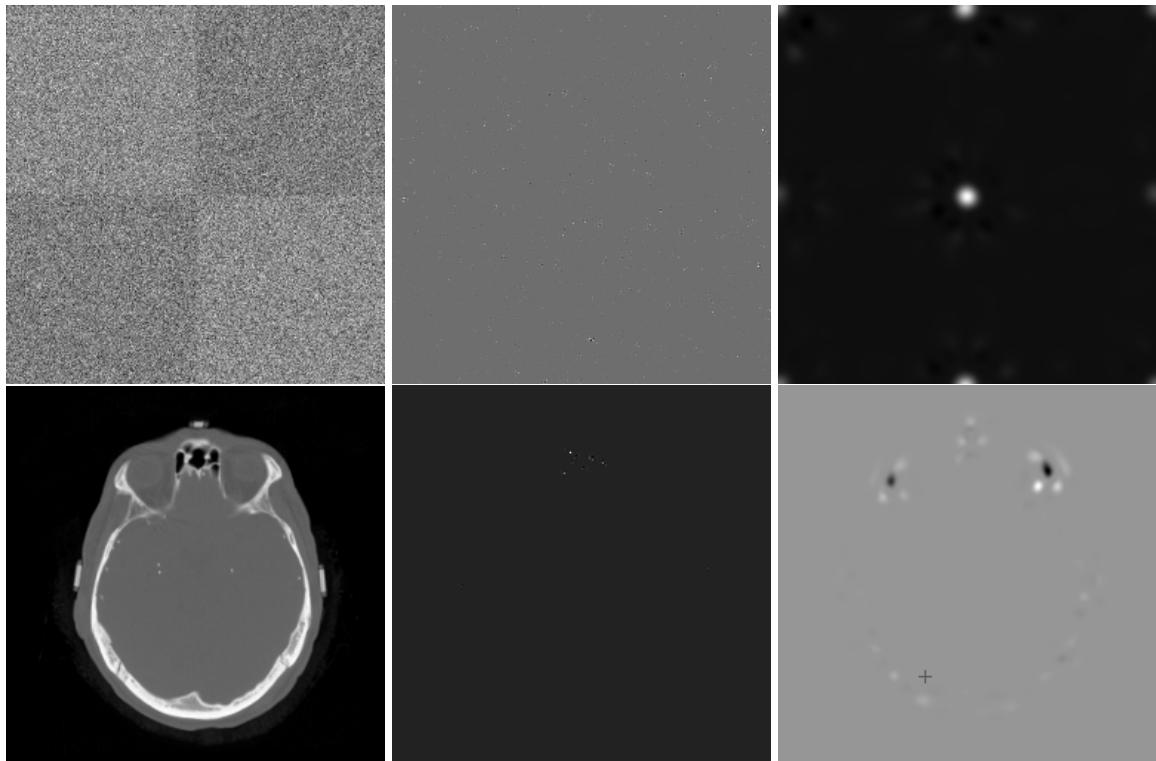


Figure 9.14 Top row: original noisy (256×256) image of part of a checkerboard, and two checkerboard detector images at scales of 1 and 16 pixels. Bottom row: 256×256 CT image and checkerboard detector images at scales of 1 and 8 pixels

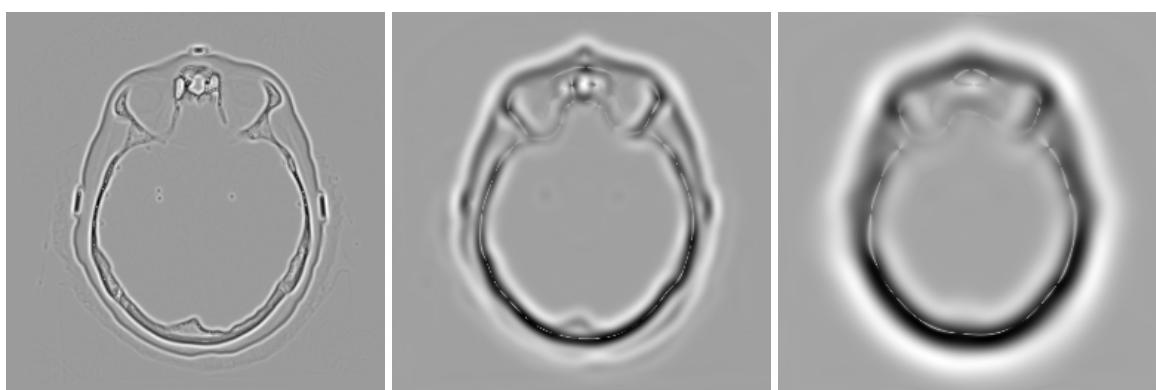


Figure 9.15 f_{ww} isophote density images at scales of 1, 4, and 8 pixels of the CT image in the previous figure.

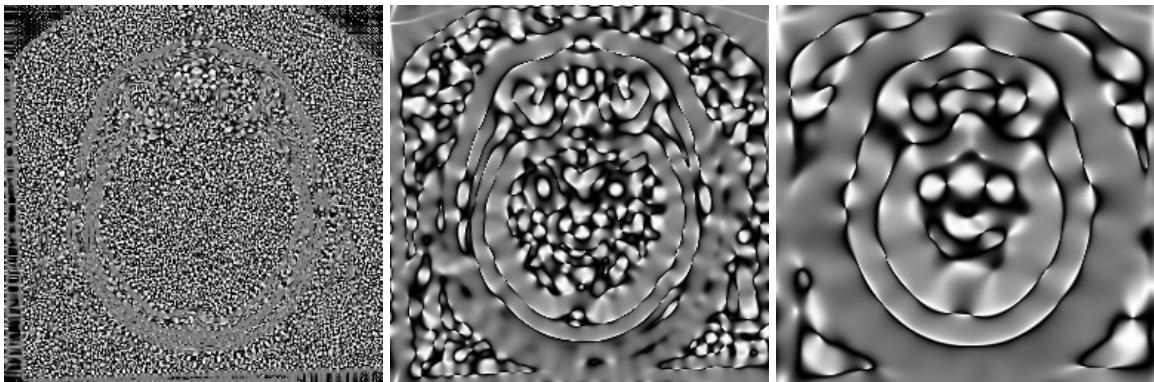


Figure 9.16 Umbilicity images at scales of 1, 4, and 8 pixels of the CT image used in the previous figures. The umbilicity operator distinguishes elliptical (positive value) and hyperbolic (negative value) areas of the image intensity landscape.

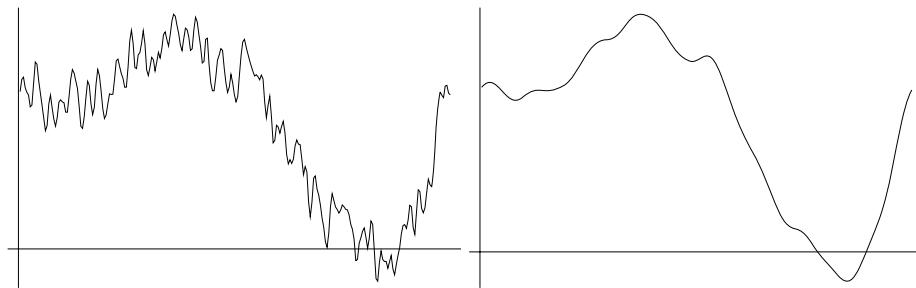


Figure 9.17 Scale space and the physical process of heat conduction. Left: temperature as a function of length of a metal rod. Right: the same function, after some time has elapsed and normal heat conduction has taken place. These functions are related to scale space: the right graph is in fact the left graph convolved with a Gaussian.

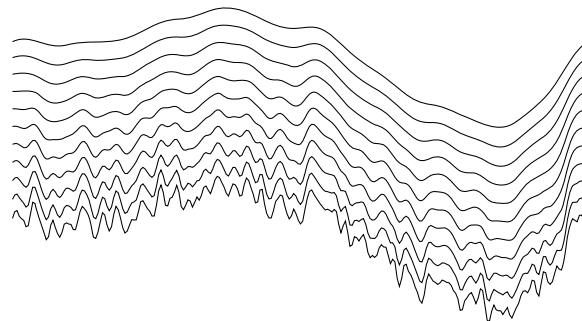


Figure 9.18 Some temperature evolution states of the function in the previous figure (plotted vertically above each other for clarity). The space of the temperature evolution equals the scale space of the original state.

different materials –with different heat conduction speeds– are involved. An extended heat equation that can handle heat conduction that changes in space and time is

$$f_t = \nabla \cdot (c \nabla f),$$

where $c = c(x, y, t)$ is a measure for the local heat conduction. If c is very small at a certain location, then almost no heat will flow across that location when time passes. This is where the analogy of scale space vs. heat conduction becomes interesting: it is often a drawback of scale space that edges of objects become very blurred (diffused!) at high scales. Sometimes, it is useful if we could somehow preserve the edges of objects at a scale σ or larger. We can do this by making $c(x, y, t)$ small if (x, y) is located at an edge. To determine if (x, y) is located at an edge, we can use the gradient magnitude f_w (computed at the right scale) as an edgeness measure. A suitable formula for c (one of many, really) is

$$c = \frac{1}{1 + \left(\frac{f_w}{K}\right)^2},$$

where K is a positive constant. The heat equation with this c substituted is known as the *Perona-Malik* equation. Applying this equation to an image creates a totally different scale space than when $c = 1$, as in the 'normal' scale space.

We can create other scale spaces by modifying the c function so that diffusion around some type of geometrical structures (edges, ridges, etc.) is encouraged or discouraged. The generating equations can then, e.g., be $f_t = f_w$, $f_t = f_{vv}$, $f_t = f_{vv}^{1/3} f_w^{2/3}$, etc., to respectively encourage diffusion at edges, at ridges (diffuse in the center of objects rather than at the edges), and at corners⁴. Figure 9.19 shows some examples. Figure 9.20 shows a noise reduction application.

While computing a scaled image in the Gaussian scale space is relatively easy –it requires only convolution of the original image with a Gaussian– images in the other scale spaces mentioned cannot usually be computed directly, because the necessary convolution kernel is different in each location of the image. In practice, scaled images are computed by letting the underlying differential equation evolve to the desired scale (time), using a numerical method (such as Euler's method) to approximate the solution of the equation.

⁴Evolution of an image according to these three equations are also known as normal flow, Euclidean shortening flow, and affine shortening flow respectively.



Figure 9.19 Top row: original image and two scaled versions from a scale space generated by the equation $f_t = f_{vv}$. Bottom row: three scaled images from a scale space generated by the Perona-Malik equation.

9.3 The resolution pyramid

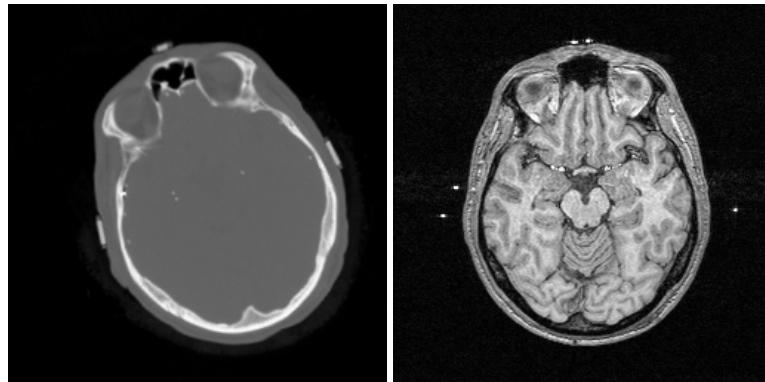
At the start of this chapter, we mentioned a simple method for lowering the resolution of an image: divide the image in regular blocks of pixels (say 2×2 pixels each), and merge the pixels in each block to a single new pixel. This merging can be done by, e.g., averaging the pixel values, taking (e.g.) the top-left pixel value, or taking the maximum or minimum pixel value. We can apply the same operation to the results, and again to its results, etc. creating a series of images –each image with a lower resolution than the image before it– called a *resolution pyramid*. Figure 9.21 shows an example.

This method of resolution lowering lacks most of the useful mathematical properties that scale space has, and it is not straightforward how to lower the resolution by a factor that is not integer. Nevertheless, the resolution pyramid is extremely useful in its own right in many different applications. This is because an image in the pyramid retains much of the global structure of the original image (exactly how much depending on the pyramid level), while taking less memory bytes to store. In general, processing of a pyramid image is significantly faster than processing the original image. A common

way to employ the pyramid is to do fast and coarse processing on the top level image (the lowest resolution image), and use its results to guide processing at the level below it, continuing until the original image (the bottom level of the pyramid) is reached. For an example, we show the use of a pyramid in an image registration application.

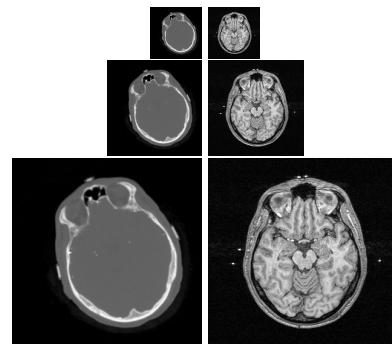
Example: image registration using a multiresolution pyramid

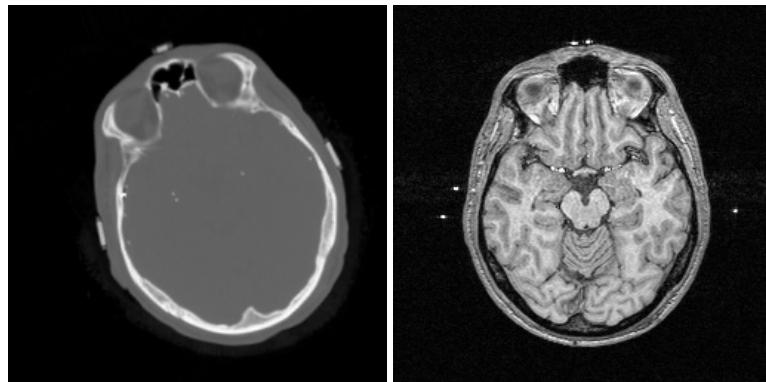
Suppose we have two 256×256 images that need to be brought into registration by rotation and translation of one of the two:



We want the registration error to be no more than one pixel size anywhere in the image. The straightforward 'brute force' approach to find the correct registration is to examine all possible transformations, computing a suitable registration measure in each case, in finally picking the transformation for which the measure is maximum. Unfortunately, this approach results in over 60 million configurations to check, and so is not usually a feasible approach to take.

A better approach is to build a (e.g.) four level pyramid:





The top level images have a 32×32 resolution, and the number of possible transformations that are no more than one pixel length apart is now low enough for each to be examined separately. Each examination in itself is also much faster than examining the original, because the number of pixels is reduced by a factor of 64.

The best transformation found at the top pyramid level will be too coarse to be very accurate at the next pyramid level, but it can function as a suitable starting position at this level. At this level, we again examine a number of transformations and pick out the best one, but keep the number of transformations down by examining only a suitable range of transformations around the starting position. In this fashion, we continue down the pyramid until we reach the original image, where we can find the optimal transformation by examining only a relatively small number of transformations around the starting position.

The possibility of a coarse-to-fine strategy such as the one described above is not only an asset of the resolution pyramid, but also of any scale space. The implementation and use of the strategy is most straightforward in the case of the resolution pyramid. The pyramid, however, lacks many of the useful mathematical properties of scale space that enable a more robust approach using scale space techniques instead of the pyramid in many applications.

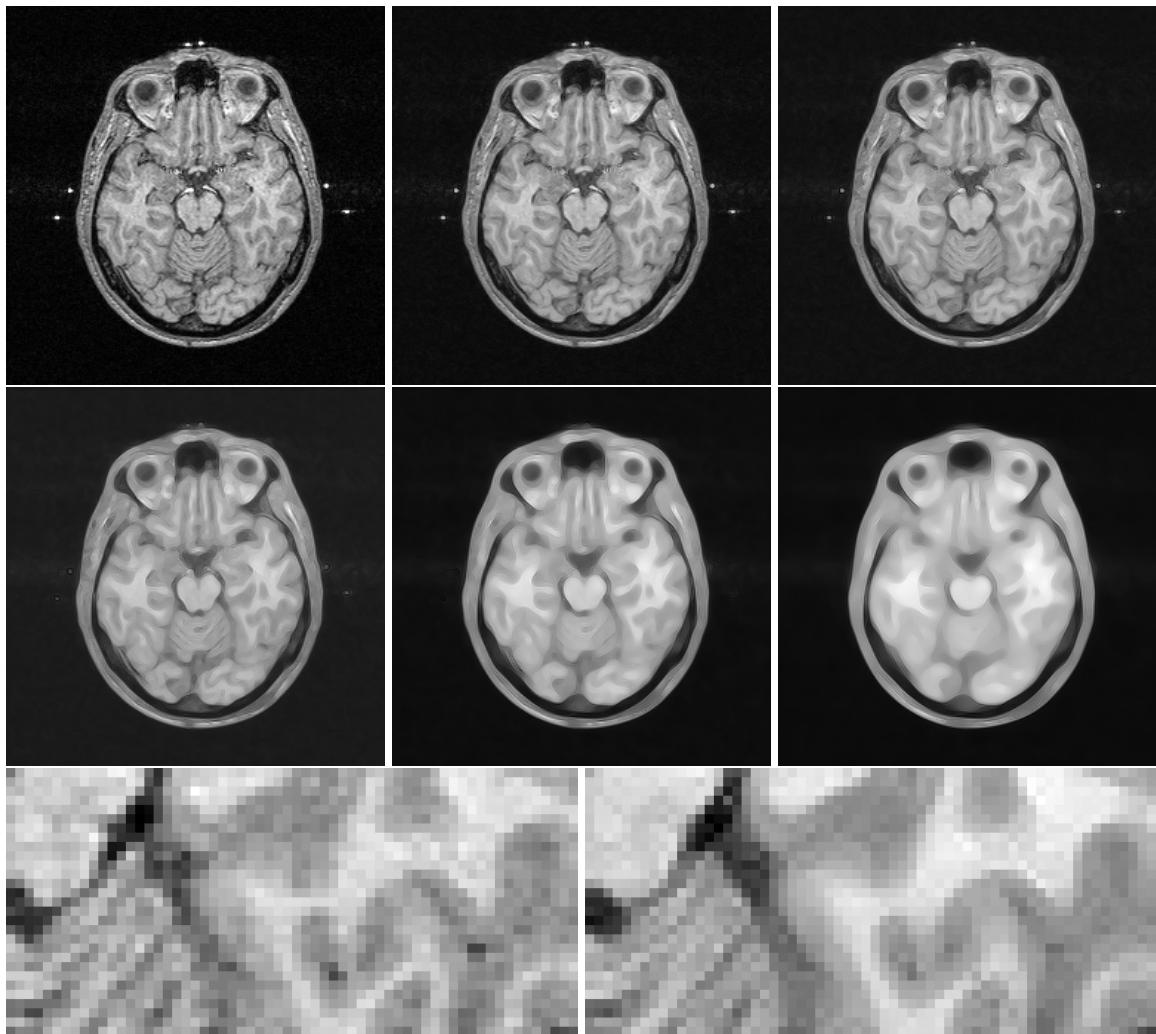


Figure 9.20 Scale space generated from the top left image using the equation $f_t = f_{vv}$. A zoom of the original and top right image shows that a moderate scaling of an image removes noise without touching essential image structures much.

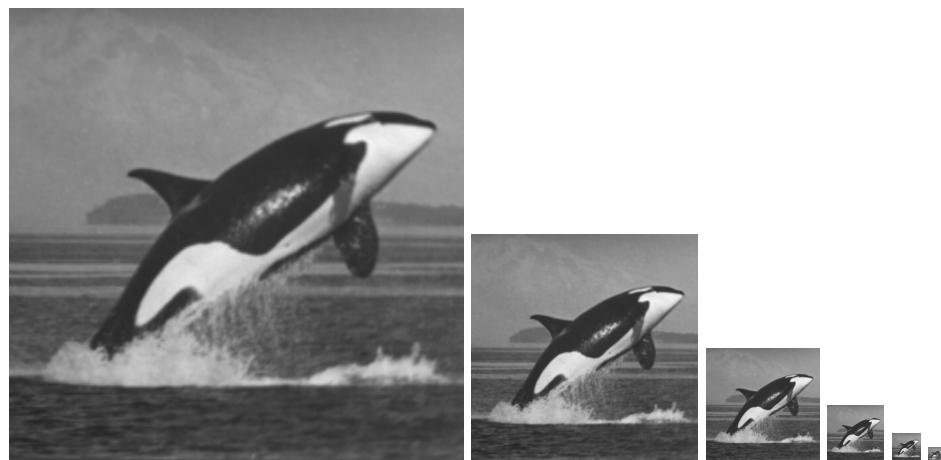


Figure 9.21 Example of a resolution pyramid. The original 256×256 image is on the left. The next image is formed by averaging each block of 4 pixels into a single pixel, forming a 128×128 image. Each following image is formed in the same manner; the final image has a resolution of 8×8 pixels.

Chapter 10

Segmentation

The division of an image into meaningful structures, *image segmentation*, is often an essential step in image analysis, object representation, visualization, and many other image processing tasks. In chapter 8, we focussed on how to analyze and represent an object, but we assumed the group of pixels that identified that object was known beforehand. In this chapter, we will focus on methods that find the particular pixels that make up an object.

A great variety of segmentation methods has been proposed in the past decades, and some categorization is necessary to present the methods properly here. A disjunct categorization does not seem to be possible though, because even two very different segmentation approaches may share properties that defy singular categorization¹. The categorization presented in this chapter is therefore rather a categorization regarding the *emphasis* of an approach than a strict division.

The following categories are used:

- **Threshold based segmentation.** Histogram thresholding and slicing techniques are used to segment the image. They may be applied directly to an image, but can also be combined with pre- and post-processing techniques.
- **Edge based segmentation.** With this technique, detected edges in an image are assumed to represent object boundaries, and used to identify these objects.
- **Region based segmentation.** Where an edge based technique may attempt to find the object boundaries and then locate the object itself by filling them in, a region based technique takes the opposite approach, by (e.g.) starting in the middle of an object and then “growing” outward until it meets the object boundaries.

¹In much the same way as the platypus does not seem to fit in any of normal zoological categories. It seems that any segmentation method categorized as a mammal, upon closer inspection, appears to have some aspect that shows it to be laying eggs.

- **Clustering techniques.** Although clustering is sometimes used as a synonym for (agglomerative) segmentation techniques, we use it here to denote techniques that are primarily used in exploratory data analysis of high-dimensional measurement patterns. In this context, clustering methods attempt to group together patterns that are similar in some sense. This goal is very similar to what we are attempting to do when we segment an image, and indeed some clustering techniques can readily be applied for image segmentation.
- **Matching.** When we know what an object we wish to identify in an image (approximately) looks like, we can use this knowledge to locate the object in an image. This approach to segmentation is called matching.

Perfect image segmentation –*i.e.*, each pixel is assigned to the correct object segment– is a goal that cannot usually be achieved. Indeed, because of the way a digital image is acquired, this may be impossible, since a pixel may straddle the “real” boundary of objects such that it partially belongs to two (or even more) objects. Most methods presented here –indeed most current segmentation methods– only attempt to assign a pixel to a single segment, which is an approach that is more than adequate for most applications. Methods that assign a segment probability distribution to each pixel are called *probabilistic*. This class of methods is theoretically more accurate, and applications where a probabilistic approach is the only approach accurate enough for specific object measurements can easily be named. However, probabilistic techniques add considerable complexity to segmentation –both in the sense of concept and implementation– and as such are still little used.

Perfect image segmentation is also often not reached because of the occurrence of *oversegmentation* or *undersegmentation*. In the first case, pixels belonging to the same object are classified as belonging to different segments. A single object may be represented by two or more segments. In the latter case, the opposite happens: pixels belonging to different objects are classified as belonging to the same object. A single segment may contain several objects. Figure 10.1 shows a simple example of over- and undersegmentation.

10.1 Threshold based segmentation

Thresholding is probably the most frequently used technique to segment an image. The thresholding operation is a grey value remapping operation g defined by:

$$g(v) = \begin{cases} 0 & \text{if } v < t \\ 1 & \text{if } v \geq t, \end{cases}$$

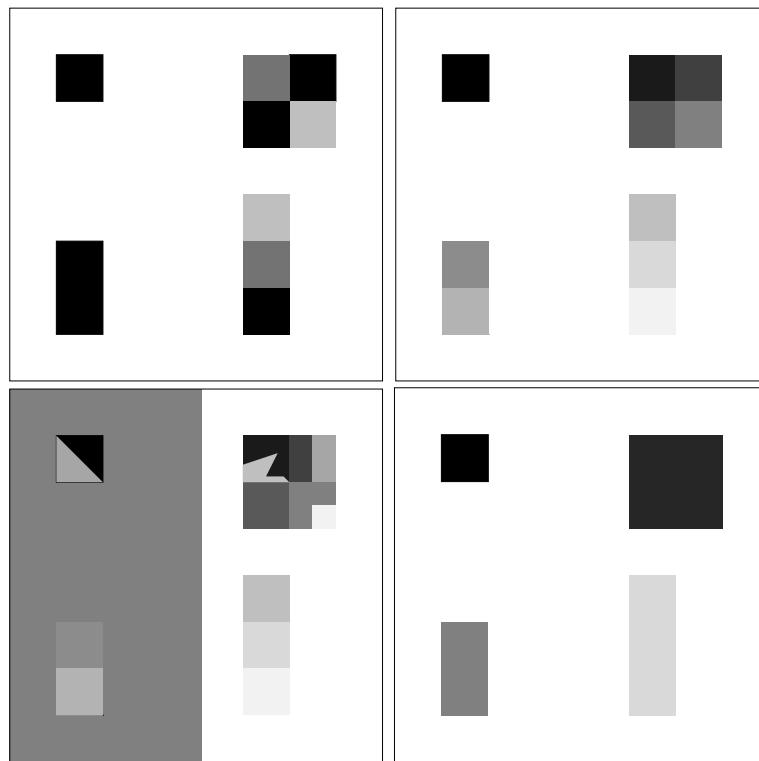


Figure 10.1 An original image is shown at the top left. If it is known that this image contains only uniformly sized squares, then the image on the top right shows the correct segmentation. Each segment has been indicated by a unique grey value here. The bottom left and right images show examples of oversegmentation and undersegmentation respectively.

where v represents a grey value, and t is the threshold value. Thresholding maps a grey-valued image to a binary image. After the thresholding operation, the image has been segmented into two segments, identified by the pixel values 0 and 1 respectively.

If we have an image which contains bright objects on a dark background, thresholding can be used to segment the image. See figure 10.2 for an example. Since in many types of images the grey values of objects are very different from the background value, thresholding is often a well-suited method to segment an image into objects and background. If the objects are not overlapping, then we can create a separate segment from each object by running a labelling algorithm (see chapter 8) on the thresholded binary image, thus assigning a unique pixel value to each object.

Many methods exist to select a suitable threshold value for a segmentation task. Perhaps the most common method is to set the threshold value interactively; the user manipulating the value and reviewing the thresholding result until a satisfying segmentation has been obtained. The histogram is often a valuable tool in establishing a suitable threshold value. In figure 10.3 we show the image from the previous figure together with its

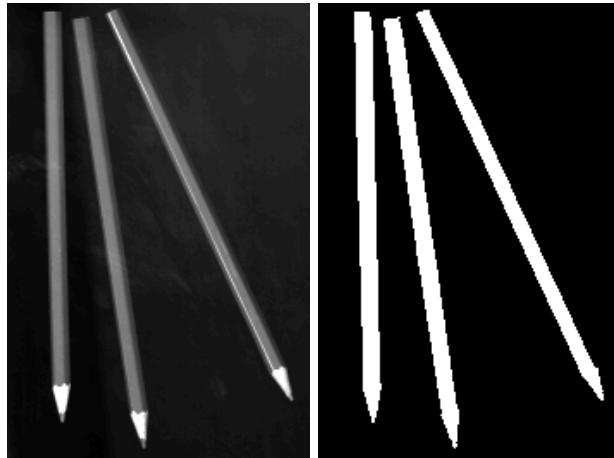


Figure 10.2 Example of segmentation by thresholding. On the left, an original image with bright objects (the pencils) on a dark background. Thresholding using an appropriate threshold segments the image into objects (segment with value 1) and background (segment with value 0).

histogram, and the thresholding results using four different threshold values obtained from the histogram. In the next section, threshold selection methods are discussed in more detail.

When several desired segments in an image can be distinguished by their grey values, threshold segmentation can be extended to use multiple thresholds to segment an image into more than two segments: all pixels with a value smaller than the first threshold are assigned to segment 0, all pixels with values between the first and second threshold are assigned to segment 1, all pixels with values between the second and third threshold are assigned to segment 2, etc. If n thresholds (t_1, t_2, \dots, t_n) are used:

$$g(v) = \begin{cases} 0 & \text{if } v < t_1 \\ 1 & \text{if } t_1 \leq v < t_2 \\ 2 & \text{if } t_2 \leq v < t_3 \\ \vdots & \vdots \quad \vdots \\ n & \text{if } t_n \leq v. \end{cases}$$

After thresholding, the image has been segmented into $n + 1$ segments identified by the grey values 0 to n respectively. Figure 10.4 shows an example using two thresholds.

10.1.1 Threshold selection

Many methods exist to find a suitable threshold for segmentation. The simplest method is the interactive selection of a threshold by the user –possibly with the aid of the image

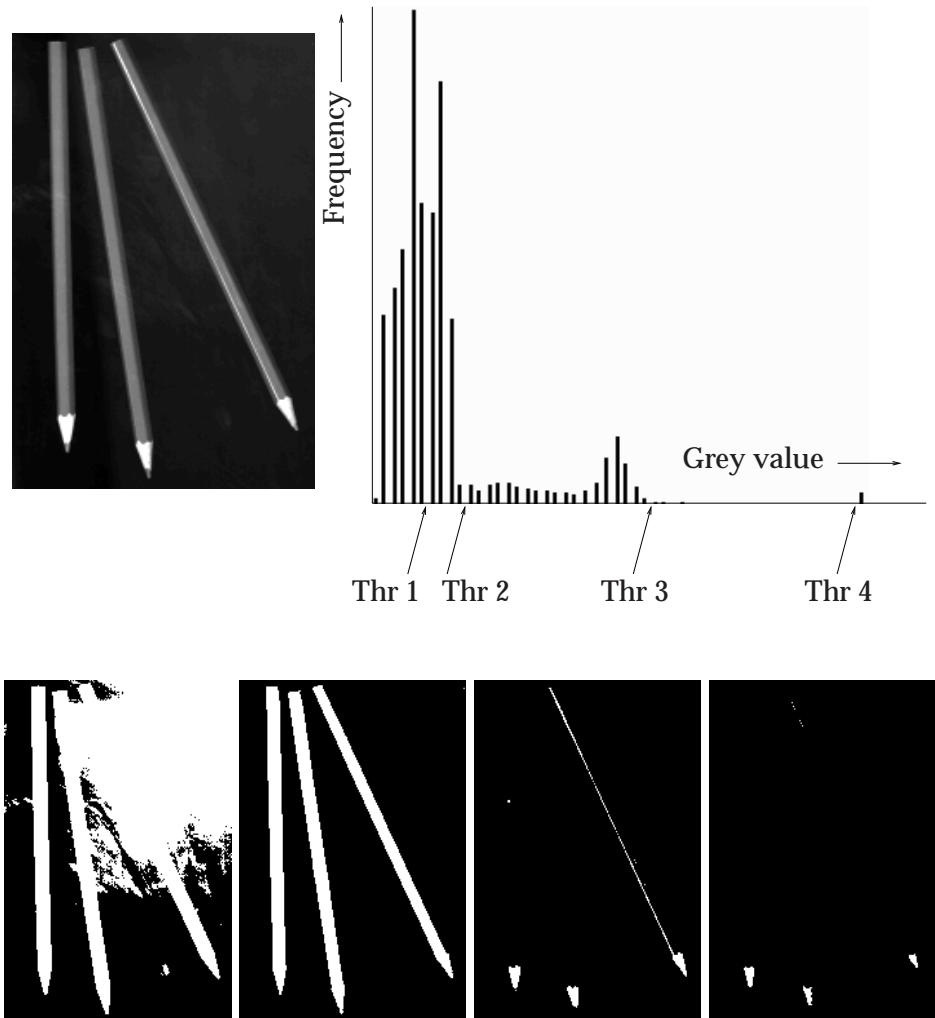


Figure 10.3 Example of threshold selection from the histogram. Top row: original image and histogram. Four special grey values (indicated by Thr 1,2,3,4) have been chosen. The bottom row shows the respective thresholding results at each of the values. At a first glance, the original image appears to have only three grey values. But the histogram shows that the grey value distribution is more diffuse; that the three basic values are in fact spread out over a certain range. Because the background grey values occur most frequently, we expect all of the large values in the left part of the histogram to correspond to the background. This is indeed the case. The result of threshold 1 shows that the peak between thresholds 1 and 2 also corresponds to background values. The result of threshold 2 shows the desired segmentation; every grey value to the right of threshold 2 corresponds to the pencils. Threshold 4 shows that the right-most little peak corresponds to the bright grey value in the tips of the pencils.

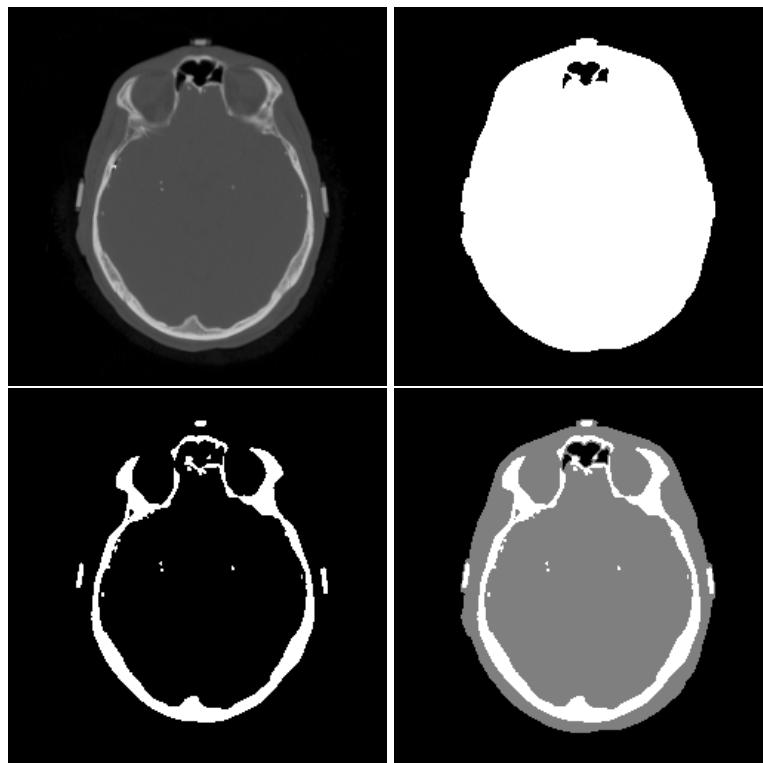


Figure 10.4 Example of using multiple thresholds for segmentation. Top left: original image. Top right: thresholding result after using a low threshold value to segment the image into head and background pixels. Bottom left: result after using a higher value to segment the bone pixels. Bottom right: result after using both thresholds at once.

histogram – a method that is usually accompanied by a graphical tool which lets the user immediately assess the result of a certain choice of threshold. Automatic methods often make use of the image histogram to find a suitable threshold. Some of these methods are detailed below.

Using histogram extrema. A typical objects-on-a-background image as shown in figure 10.5 will have a bimodal² histogram. The peaks of the histogram will not generally be as sharp as we would like them to be, because of the influence of image degrading factors such as noise, illumination artifacts and partial volume effects.³

In practice, the curves in the histogram corresponding to certain objects may overlap. When this happens, an errorless segmentation based on global thresholding is no longer

²Two-peaked.

³The partial volume effect occurs when a pixel is on or near the boundary of two or more image objects; when the area that influences the grey value of the pixel contains parts of more than one object. With many types of image acquisition, the grey value of the pixel will then not correspond to the expected grey value of one of the objects, but will be a weighted average of the grey values of the objects involved.

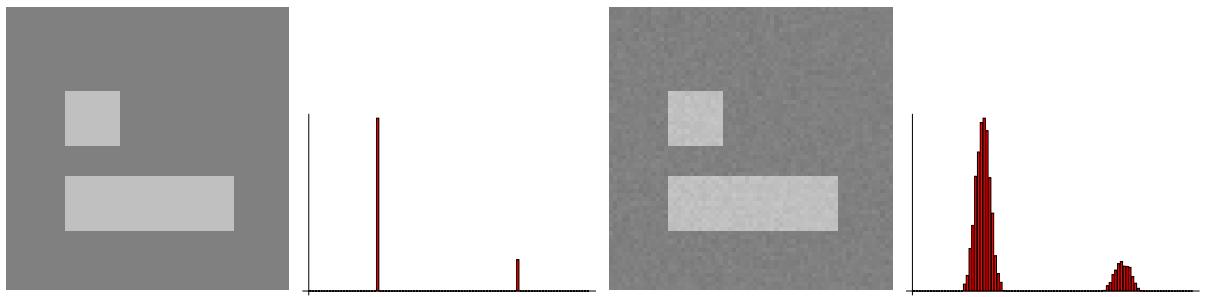


Figure 10.5 At the top left, an image containing only two grey values is shown. The histogram (next to it) shows only two peaks. In practice, image artifacts will cause the image and its histogram to degrade from this ideal situation. In particular, the histogram blurs as is shown in the image and histogram on the right. In practice, it is quite possible that the two curves in the histogram start to overlap.

possible.

Regardless of whether or not there is overlap in the histogram, we can use the maxima (peaks) of the histogram to establish a segmentation threshold. This threshold t may be midway between two peaks with grey values p_1 and p_2 :

$$t = \frac{p_1 + p_2}{2},$$

or better still, it may be the grey value at the minimum between the two peaks:

$$t = \arg \min_{v \in [p_1, p_2]} H(v),$$

where $H(v)$ gives the histogram value at grey value v , and we assume that p_1 is smaller than p_2 . Figure 10.6 shows why the minimum choice for t is usually better than the midway choice.

The histogram is often not as smooth as we would like, as the figure shows. For many histogram analysis tasks (such as finding the global extrema) it is therefore useful to smooth the histogram beforehand. This can, e.g., be done by convolution with a Gaussian or a discrete averaging kernel.

Minimum variance within segments. If we assume that a segment should have relatively homogeneous grey values, it makes sense to select a threshold that minimizes the variance of the original grey values within segments. Alternatively, we may select a threshold that maximizes the variance *between* objects and background, or one that attempts to optimize both these “within” and “between” variances.

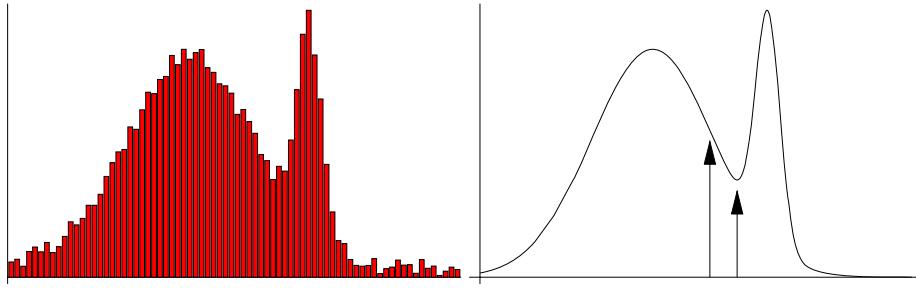


Figure 10.6 Example comparing a threshold midway between peaks and at the minimum between peaks. On the left, a bimodal histogram is shown. The right image shows a smoothed version of this histogram, together with an arrow showing the threshold midway between the peaks (left arrow) and showing the threshold at the minimum (right arrow). The threshold at the minimum will misclassify fewer pixels.

We will give an example using one threshold, *i.e.*, two segments. For simplicity, we normalize the histogram $H(v)$ of the image to a proper distribution $h(v)$, *i.e.*, $\sum h(v) = 1$. This can be achieved by setting $h(v) = H(v)/n$ for all v , where n is the total number of pixels in the image. The variance of the grey values σ^2 in the image then (by definition) equals

$$\sigma^2 = \sum_v (v - \mu)^2 h(v),$$

where $\mu = \sum_v v h(v)$ is the mean grey value of the image. If we segment the image with threshold value t into a background segment 0 and an object segment 1, then the variance of grey values within each segment (respectively σ_0 and σ_1) is⁴

$$\begin{aligned}\sigma_0^2 &= \sum_{v < t} (v - \mu_0)^2 h(v) \\ \sigma_1^2 &= \sum_{v \geq t} (v - \mu_1)^2 h(v),\end{aligned}$$

where

$$\mu_0 = \frac{1}{h_0} \sum_{v < t} v h(v)$$

and

$$\mu_1 = \frac{1}{h_1} \sum_{v \geq t} v h(v)$$

⁴Note that these definitions are different –by a factor– from the ‘real’ statistical definition. This factor is corrected for later in the definition of σ_w^2 . This formulation requires less operations to compute σ_w^2 .

are the mean grey values of the respective segments 0 and 1. The probabilities h_0 and h_1 that a randomly selected pixel belongs to segment 0 or 1 are

$$\begin{aligned} h_0 &= \sum_{v < t} h(v) \\ h_1 &= \sum_{v \geq t} h(v). \end{aligned}$$

Note that $h_0 + h_1 = 1$. The total variance within segments σ_w^2 is

$$\sigma_w^2 = h_0\sigma_0^2 + h_1\sigma_1^2.$$

This variance only depends on the threshold value t : $\sigma_w^2 = \sigma_w^2(t)$. This means that we can find the value of t that minimizes the variance within segments by minimizing $\sigma_w^2(t)$.

The variance *between* the segments 0 and 1, σ_b^2 , is

$$\sigma_b^2 = h_0(\mu_0 - \mu)^2 + h_1(\mu_1 - \mu)^2.$$

Again, this variance is only dependent on the threshold value t . Finding the t that maximizes σ_b^2 maximizes the variance between segments. A hybrid approach that attempts to maximize σ_b^2 while minimizing σ_w^2 is to find the threshold t that maximizes the ratio σ_b^2/σ_w^2 .

If more than two segments are required, the method described above can be extended to use multiple thresholds. The variances σ_w^2 and σ_b^2 will then be functions of more than one threshold, so we need multi-dimensional optimization to find the set of optimal thresholds. Since this is especially cumbersome if the number of segments is large, a more practical algorithm that minimizes the variances within segments is often used, an iterative algorithm known as *K-means clustering*.

Algorithm: K-means clustering

The objective of the *K*-means clustering algorithm is to divide an image into *K* segments (using $K - 1$ thresholds), minimizing the total within-segment variance. The variable *K* must be set before running the algorithm.

The within-segment variance σ_w^2 is defined by

$$\sigma_w^2 = \sum_{i=0}^{K-1} h_i \sigma_i^2,$$

where $h_i = \sum_{v \in S_i} h(v)$ is the probability that a random pixel belongs to segment *i* (containing the grey values in the range S_i), $\sigma_i^2 = \sum_{v \in S_i} (v - \mu_i)^2 h(v)$ is the variance of grey values of segment *i*, and $\mu_i = \sum_{v \in S_i} v h(v)$ is the mean grey value in segment *i*. All definitions are as before in the case with a single threshold.

1. Initialization: distribute the $K - 1$ thresholds over the histogram. (For example in such a way that the grey value range is divided into K pieces of equal length.) Segment the image according to the thresholds set. For each segment, compute the 'cluster center', i.e., the value midway between the two thresholds that make up the segment.
2. For each segment, compute the mean pixel value μ_i .
3. Reset the cluster centers to the computed values μ_i .
4. Reset the thresholds to be midway between the cluster centers, and segment the image.
5. Go to step 2. Iterate until the cluster centers do not move anymore (or do not move significantly).

Note that this algorithm –although it minimizes a variance– does not require any variance to be computed explicitly!

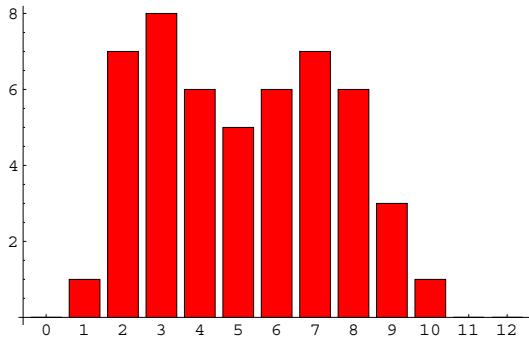
The version of this algorithm with only one threshold (two segments) can be extracted from the general form above. In practice, the initialization step in this case is often slightly different to try to get the initial threshold close to the optimal position, so the algorithm will converge faster. A heuristic rule is often used to achieve this. The algorithm –with a widely employed heuristic– then becomes:

Algorithm: Iterative thresholding

1. Assume that the four corner points of the image are background pixels (part of segment 0), and set μ_0 to the average grey value of these four pixels. Assume all of the other pixels are object pixels, and set μ_1 to their average grey value.
 2. Set the threshold t to $t = \frac{1}{2}(\mu_0 + \mu_1)$, and segment the image.
 3. Recompute μ_0 and μ_1 , the mean of the original grey values of the two segments.
 4. Go to step 2 and iterate until the threshold value no longer changes (or no longer changes significantly).
-

Example

If we have an image with the following histogram:



and if we know that the four corner pixels have grey values 2, 2, 2, and 1 respectively, then the initial value for μ_0 is $\mu_0^{(0)} = \frac{2+2+2+1}{4} = 1.75$. We can compute the average value of the remaining pixels from the image or the histogram: $\mu_1^{(0)} \approx 5.46$. So we set the initial threshold $t^{(0)} = \frac{1}{2}(\mu_0^{(0)} + \mu_1^{(0)}) \approx 3.61$.

Thresholding then results in a segment 0 containing all pixels with original grey values in the range $\{0, \dots, 3\}$, and a segment 1 containing pixels with grey values in the range $\{4, \dots, 12\}$.

Computing the mean values of these two segments, we find $\mu_0^{(1)} \approx 2.44$ and $\mu_1^{(1)} \approx 6.44$. The new threshold will therefore be $t^{(1)} = \frac{1}{2}(\mu_0^{(1)} + \mu_1^{(1)}) \approx 4.44$.

Thresholding then results in segment 0 containing the pixels with values in $\{0, \dots, 4\}$, and segment 1 with the range $\{5, \dots, 12\}$. Recomputing the mean segment values gives $\mu_0^{(2)} \approx 2.86$ and $\mu_1^{(2)} \approx 6.96$, so $t^{(2)} \approx 4.91$.

This last threshold will result in the same segmentation as the previous step, so we terminate the algorithm here. The last computations of the segments 0 and 1 are assumed to be the optimal ones.

Optimal thresholding. Figure 10.5 showed us that an object that theoretically has a uniform grey value and should cause a single peak in the histogram, in practice shows a distribution of grey values in the histogram. This distribution can often be approximated using a known statistical distribution function such as the Gaussian. *Optimal thresholding* is a technique that approximates the histogram using a weighted sum of distribution functions, and then sets a threshold in such a way that the number of incorrectly segmented pixels (as predicted from the approximation) is minimal.

For an example, we assume the histogram is bimodal (two segments; objects and background), and that the histogram can be approximated well using a sum of two Gaussians. In this case, this means we model the normalized histogram h as

$$\begin{aligned} h(v) &\approx P_0 g_0(v) + P_1 g_1(v) \\ &= (1 - P_1)g_0(v) + P_1 g_1(v), \end{aligned}$$

where g_0 and g_1 are Gaussian functions with unknown mean and variance, and P_0 and P_1 are the global probabilities –also unknown– that a pixel belongs to segment 0 or 1 respectively. Note that $P_0 + P_1 = 1$. This leaves us with five unknowns –two means, two variances, and P_1 – to be estimated from the image histogram. Figure 10.7 shows an example bimodal histogram and its approximation using two Gaussians. The estimation of the unknowns is usually done using a non-linear curve fitting technique which minimizes the sum of squared distances (as a function of the unknowns) between the histogram data and the fitted curve, *i.e.*,

$$\text{minimize} \sum_v (h(v) - m(v))^2,$$

where $m(v)$ is the chosen model for the histogram, which has the unknowns for parameters (here $m(v) = P_0g_0(v) + P_1g_1(v)$). The estimation of the (here) five unknowns can therefore be formulated as the minimization of a function of five variables. How this minimization is achieved is beyond the scope of this book, but techniques for this are fairly standard and can easily be found in mathematics literature.

Assuming we can estimate all of the unknowns, the optimal threshold t is the grey value where the two Gaussian curves intersect:

$$(1 - P_1)g_0(t) = P_1g_1(t).$$

Intermezzo*

This last equation can be deduced as follows:

The normalized histogram is approximated by

$$h(v) \approx (1 - P_1)g_0(v) + P_1g_1(v),$$

with $g_0(v)$ the distribution of background pixels, and $g_1(v)$ the distribution of object pixels. P_1 and $(1 - P_1)$ are the fractions of object and background pixels respectively. If we threshold the image at a certain threshold t , then the fraction of background misclassified as object pixels is

$$(1 - P_1) \int_t^{\infty} g_0(v)dv.$$

The fraction of object pixels misclassified as background pixels is

$$P_1 \int_{-\infty}^t g_1(v)dv.$$

The total error fraction $E(t)$ is the sum of these partial errors:

$$E(t) = (1 - P_1) \int_t^{\infty} g_0(v)dv + P_1 \int_{-\infty}^t g_1(v)dv.$$

The error $E(t)$ is minimal when the derivative is zero:

$$\begin{aligned} E'(t) &= 0 \\ (1 - P_1)(-)g_0(t) + P_1g_1(t) &= 0 \\ (1 - P_1)g_0(t) &= P_1g_1(t). \end{aligned}$$

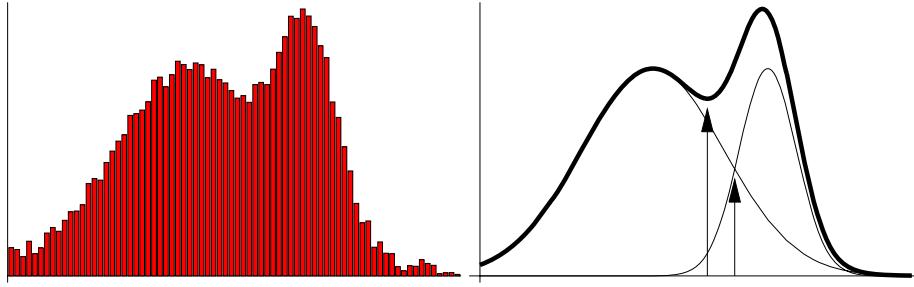


Figure 10.7 Example of optimal thresholding. Left: image histogram. Right: approximation of the histogram (thick curve) using two Gaussians (thin curves). The right arrow shows the optimal threshold. For comparison, the left arrow shows the threshold value if we had used the minimum-between-peaks criterion.

Optimal thresholding may be extended to use multiple thresholds, *i.e.*, the normalized histogram is approximated using a sum of more than two Gaussians:

$$h(v) \approx \sum_{i=0}^{k-1} P_i g_i(v),$$

where g_i is a Gaussian, P_i the global probability of a pixel belonging to segment i , and k the number of Gaussians used to model the histogram. Since the addition of each Gaussian to the model adds three parameters to be estimated, using this model becomes increasingly fickle for large k . Moreover, selecting an appropriate value for k can be a difficult problem by itself. The model can also be extended using other distributions than the Gaussian. The choice of the distribution function should be based on knowledge of the image acquisition process, from which the correct choice can often be deduced.

Histogram estimation. The optimal thresholding method described above uses an estimation of the histogram –actually, a smooth curve that best fits the measured image histogram– rather than the histogram itself. Some of the other threshold selection methods described previously also benefit from using a smooth estimate instead of the raw histogram data. For example, the methods that require finding the extrema of the histogram curve may be better off using a smooth estimate, since this way the chance of getting stuck in a local optimum is decreased, and the process is less susceptible to noise in the raw histogram data.

Good estimation of the histogram often does not require that all of the available pixel information is used. Using a small fraction of the grey values is usually sufficient for a good estimate. For example, instead of using all of the pixels (and their grey values) in the estimation process, we could randomly select 5% of the image pixels and use only those in the process. In practice –unless the noise is exceedingly large or the number of pixels in the image very small– adding the remaining 95% will not significantly improve the estimation.

The approach to estimation used in optimal thresholding is an example of a *parametric* approach. In this type of approach, the histogram is modeled by a function with a fixed number of parameters, and the function is fitted to the histogram by twiddling with the parameters. A *non-parametric* approach to histogram estimation is any approach that does not require the computation or approximation of parameters. The most frequently used method in this category is called *Parzen windowing*.

Parzen windowing estimates the histogram by a sum of simple distribution functions. In contrast to parametric techniques, the number of distribution functions is large rather than small. In fact, one function may be used for each pixel that is employed in the estimation process. Each distribution function used is identical to all others, and any parameters of the distribution function are fixed to identical values beforehand. For example, the estimate may be a sum of Gaussians, all with a mean of 0 and a variance of 1. In general the Parzen windowing estimate $p(v)$ of the normalized histogram can be written as

$$p(v) = \frac{1}{N} \sum_i d(v - v_i),$$

where N is the number of pixels from the original image used in the estimation, v_i is the grey value of a pixel, $i \in \{1, 2, \dots, N\}$, and d is a distribution function (kernel). The N pixels are usually sampled randomly or equidistantly from the original image. The kernel d is usually a Gaussian or a block or triangular function centered around zero with unit area, but other functions may also be used. Figure 10.8 shows an example of a Parzen histogram estimate using a Gaussian function for the kernel d .

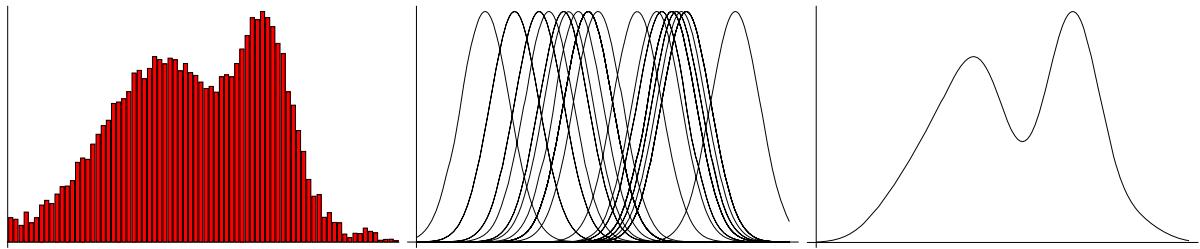


Figure 10.8 Example of Parzen windowing to estimate the histogram. In this example, the Gaussian function with zero mean and unit variance was chosen for the distribution kernel. A small fraction of the pixels was sampled randomly from the image, and a Gaussian placed at the corresponding grey value in the histogram. Left: original image histogram. Middle: Gaussian functions of the sampled pixels. Right: normalized sum of the Gaussians in the middle figure; the Parzen windowing estimate of the histogram.

In practice, non-parametric approaches such as Parzen windowing are often more flexible than parametric ones, as well as less computationally expensive. The better flexibility stems from the fact that no strong assumptions are made on the shape of the histogram, while parametric approaches effectively straitjacket the shape by assuming the histogram to be shaped like –for instance– the sum of two Gaussians.

10.1.2 Enhancing threshold segmentation

The segmentation of an image by thresholding is totally insensitive to the *spatial context* of a pixel. There are many cases conceivable where a human observer may decide on the basis of spatial context that a pixel is not part of a segment, even though it satisfies a threshold criterion. For example, noise in an image may cause many very small segments⁵ when thresholding, while in many applications it is known that such small segments cannot physically exist (see figure 10.9). Or: segment boundaries may appear bumpy while we know them to be smooth. Or: thresholding shows two objects connected by a thin ‘bridge’, while we know them to be separate objects.

Many techniques exist that address such problems. These techniques can be divided into three categories: processing of the original image prior to segmentation, processing of the segmented result, or adaptation of the segmentation process. Often, a combination of these techniques is used in a single application. In this section we will give some examples of common approaches.

The next two examples show the use of non-linear filters such as a median filter and morphological operators for removing (often noise related) small scale artifacts.

⁵Or, depending on the implementation, a single segment including many small disjunct fragments.



Figure 10.9 Example of noise causing small erroneous segments and object holes. Left: original noisy image. Middle: after thresholding. Right: the histogram of the original image. Since the two curves corresponding to object and background grey values overlap for a small part, we can be certain there exists no ideal threshold that avoids the small errors.

Example: removing small segments and object holes

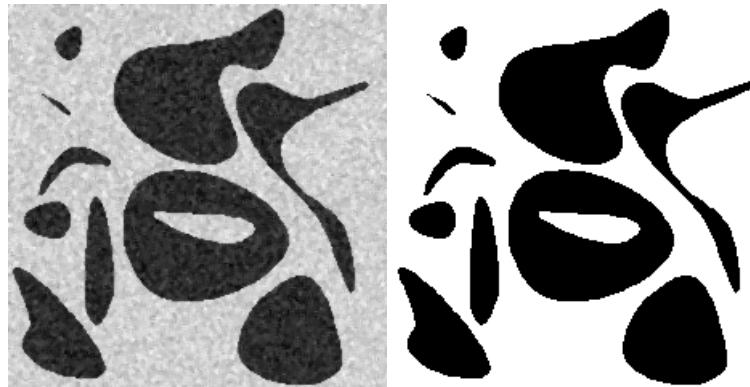
Figure 10.9 shows that noise can cause small erroneous segments and object holes after thresholding. There are many possible approaches to remove these small artifacts. For example, we could use a closing⁶ to remove the holes, and an opening to remove the small segments. We may also search for isolated pixels (or segments below a certain area) and set their grey value to the value of their neighborhood. Another approach is the application of a median filter to the segmented result, which also effectively removes all small artifacts. Below is the result of applying a 3×3 median filter to the thresholding result in figure 10.9. The original image has dimensions 256×256 .



The median filter may also be applied *prior* to the thresholding. This effectively removes the outlier grey values that caused the original segmentation artifacts.

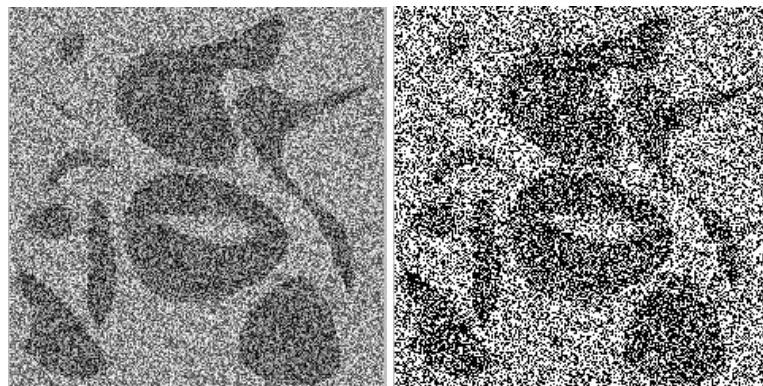
⁶Assuming the binary-image convention here: objects (black) have grey value 1 and the background (white) has grey value 0.

Below are the original image after application of a 3×3 median filter and thresholding:

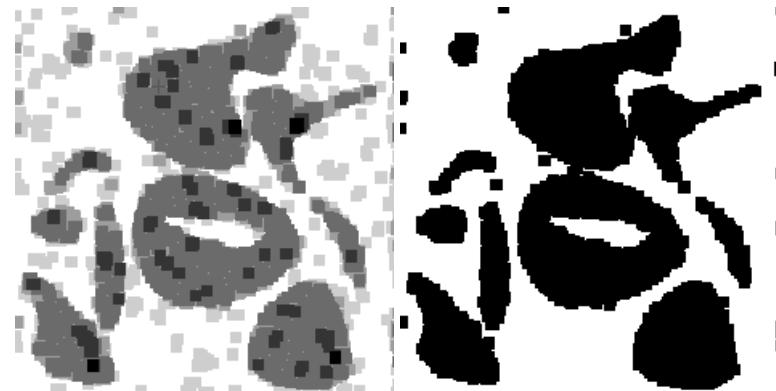


Example: pre- and postprocessing using morphology

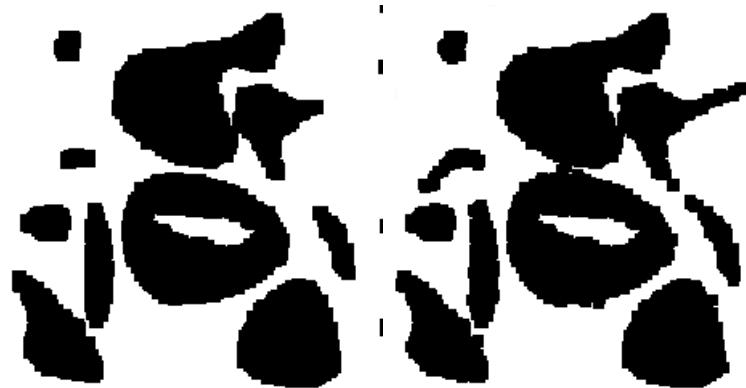
Consider these images:



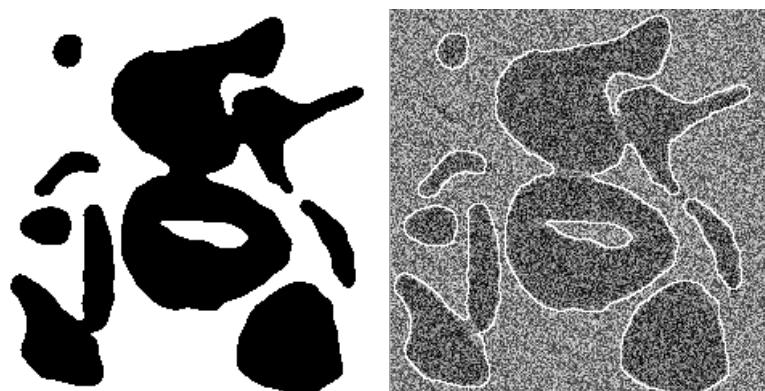
The left image is a noisy original (256×256) image, the right image a binary image after thresholding the original. Needless to say, direct thresholding performs poorly on this type of original. The influence of noise can be reduced, however, by making the object in the original more coherent using a morphological operation such as closing. The left image below shows the original image after closing using a 7×7 square, the right image the result after thresholding this image:



Although the result is much improved compared to the original thresholding, some small erroneous segments are still present. Post-processing of the result can remove these. The left image below shows the result of closing using a 9×9 square, and the right image shows a result where all segments below a certain area have been removed:



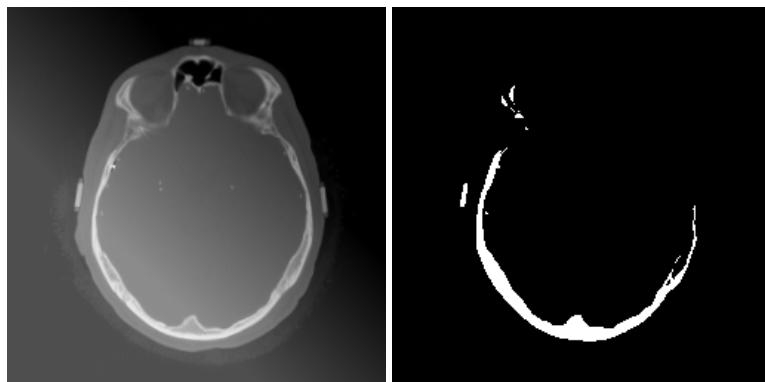
The 'blockiness' of the last result is an artifact of using square structuring elements. If a smoother result is preferred, a slight diffusion and thresholding of the last image gives the left image below. For verification of the final result, the right image below shows the original image again with overlaid contours obtained from the left image.



Large scale grey value artifacts, such as the occurrence of an illumination gradient across an image, can be handled in various ways. In chapter 6, we have already shown that the morphological white top hat transform (see figure 6.18) can be effective to remove such a gradient. Fourier techniques are also often effective, because the gradient image is a distinct low-frequency feature. A popular technique is the subtraction of a low-pass filtered image from an original image. Below are two examples showing yet other approaches. The first is an adaptation of the thresholding algorithm that bases a threshold on the *local* image grey values, *i.e.*, based only on the neighborhood of a pixel rather than all of the grey values in an image. The second is a pre-processing technique that estimates and removes the large grey value artifact from the image.

Example: local thresholding

All of the thresholding techniques presented before applied a *global* threshold to an image, *i.e.*, the threshold was the same for the entire image. This can not give good segmentation result for images where there is an 'illumination' gradient such as this one (shown with a global thresholding result on the right):



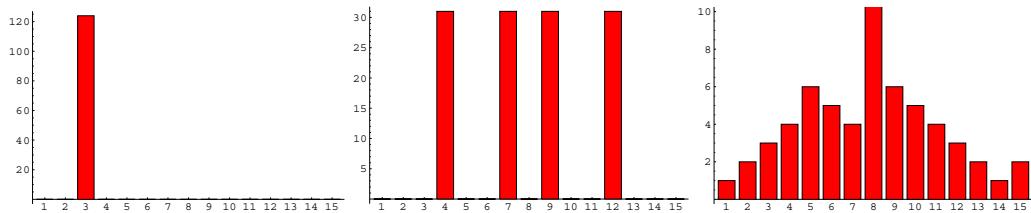
In this image, a bone pixel on the lower-left of the image has a grey value of approximately 230, while a bone pixel on the top-right has a grey value of only 100. This large gradient of grey values causes the global histogram to be diffused, and no good global threshold to extract bone may be found.

If we tile the image into, say, 50 subimages, then the grey value of bone pixels in each subimage will be relatively constant, because the gradient effect is comparatively small in each subimage. For each subimage, we can now find an appropriate threshold to extract the bone pixels. Merging the segmentation results of each subimage into a complete image again will give us a good segmentation of the whole image. Thresholding with a threshold that is not global, but based on image structure and the histogram of subimages is called local thresholding.

Local thresholding can be used effectively when the gradient effect is small with respect to the chosen subimage size. If the gradient is too large, similar errors as in the global example will now occur within a subimage, and the segments found within subimages will no longer match up adequately at the subimage seams. This is a telltale that the subimage size should be reduced. However, we cannot make the subimage too small, because then a reliable histogram can no longer be made.

Example: using histogram entropy

Consider these three histograms of images A , B , and C respectively, each containing 124 entries:



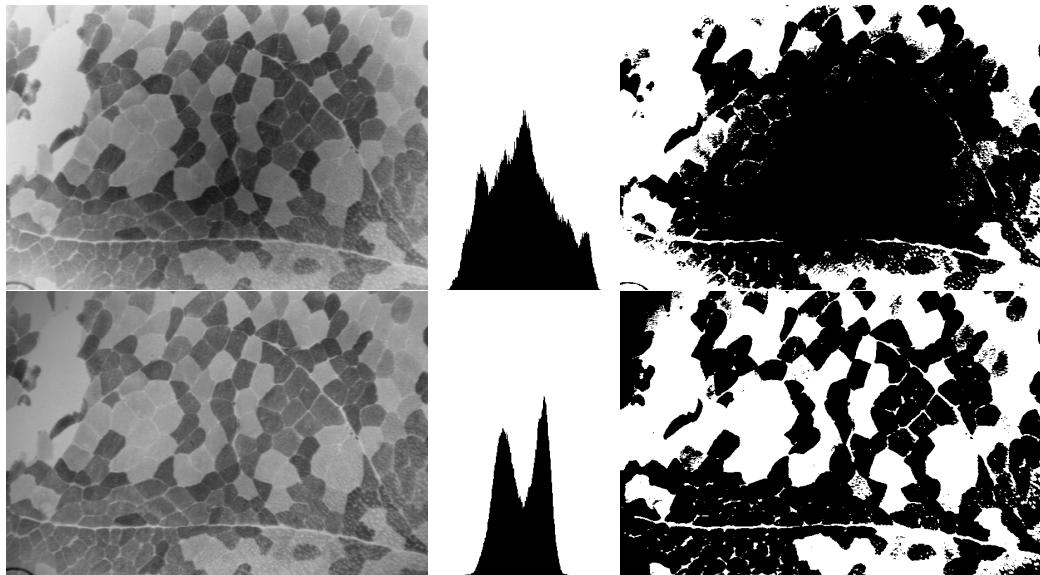
If we randomly pick a pixel from image A , we can be certain beforehand that its grey value will be three. If we randomly pick a pixel from B , we are sure it's 4, 7, 9, or 12, but each is equally likely. If we pick a pixel from C , we are even less certain; it may be anything from 1 to 15. Eight seems most likely, but even then we have about a 90% probability of being wrong.

These three histograms are examples of histograms where the *uncertainty* of grey values increases from A to C . Another word for this uncertainty is the *entropy* of a histogram. An often used numerical measure for entropy is the *Shannon* entropy H :

$$H = - \sum_{h(v) \neq 0} h(v) \log h(v),$$

where $h(v)$ is the normalized histogram of an image. Images with a histogram that has only few crisp peaks usually have a low entropy (low uncertainty), and images with a diffuse, flat histogram have a large entropy.

When an image is distorted by a large scale gradient artifact, its histogram will diffuse, and its entropy will rise. To correct the artifact, we subtract possible artifact images from the corrupted image until we find a result that has minimal entropy. We will not detail the search process of finding the correct artifact image here, but an example of a light microscope image where the gradient image is removed by minimizing the entropy is given below: the top row shows a corrupted image (with gradient artifact), its histogram and the result of thresholding. The bottom row shows the same images after correction using histogram entropy minimization.



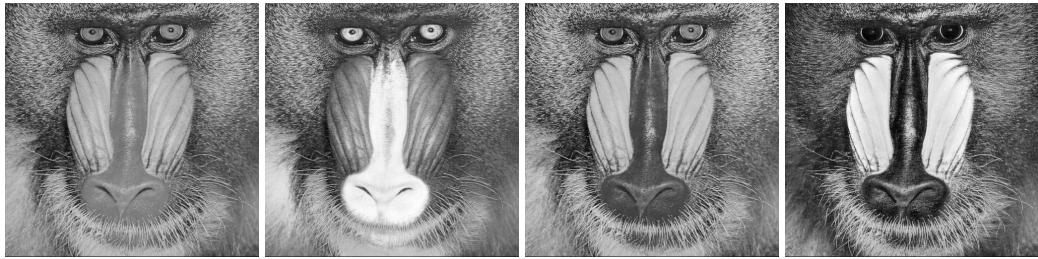
10.1.3 Multispectral thresholding

Many types of images have more than one value attached to each pixel. For example, a color image will usually have three values (e.g., RGB values) attached to each pixel. The separate value images, in this case the R, G, and B images, are often called *channels*. For another example, we may have n photographs of the same scene, so we can attach n grey values to each pixel location. Yet another example is an MR image, which may have several physical measurement values attached to each pixel.⁷ We may also attach computed values to a pixel location, for instance, next to the grey value we attach the result of locally applying a Sobel operator (or any other image processing routine) to a pixel location. Such multi-valued images allow an extension of thresholding called *multispectral thresholding* to be applied. The most basic form of multispectral thresholding entails a separate thresholding applied to each channel, and the results are combined using a boolean combination.

Example

A color image has three channels associated with it. Below is a grey valued representation of a color image of a baboon, and the red, green and blue channel of the same color image:

⁷The most common measurements are known as T_1 , T_2 and PD values.



Suppose we wish to find the image parts that are yellow. We know that yellow is a composite of red and green, so we can find the yellow pixels by thresholding the red and green channels: we take all pixels where both the red and green values are above a threshold. Unfortunately, white or bright grey pixels also have a high red and green value, as well as a high blue value. To find only yellow pixels, we demand the blue value to be *below* a threshold. Below on the left is the result after thresholding and combining (by a logical 'and') the red and green channel, and on the right the result after excluding the pixels with a high blue value: the yellow pixels.



By thresholding each channel separately and combining the results using boolean operations, we effectively find segments in rectangular regions of the 'channel space', as shown in figure 10.10. Channel space is the n -dimensional space (where n is the number of channels), where each image pixel is plotted using its channel values for coordinates. For example, a pure red pixel has coordinates $(1, 0, 0)$ in the RGB channel space. The figure also gives an example of how other regions than rectangles can be segmented by thresholding some function of all channel values.

10.2 Edge based segmentation

Since a (binary) object is fully represented by its edges, the segmentation of an image into separate objects can be achieved by finding the edges of those objects. A typical approach to segmentation using edges is (1) compute an edge image, containing all

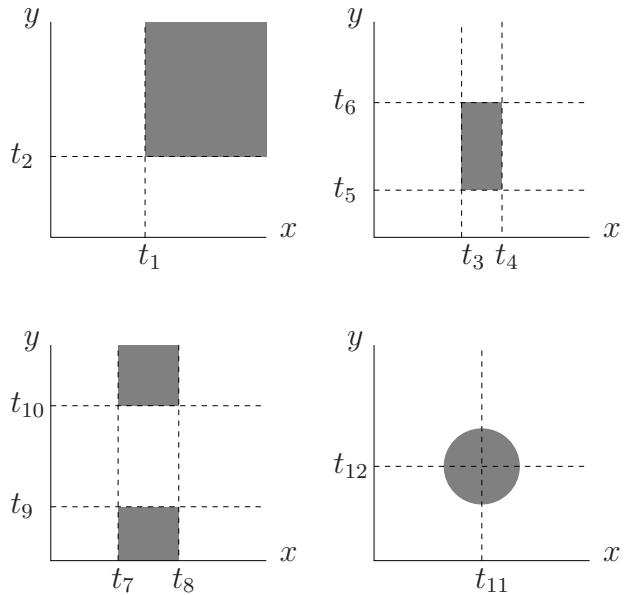


Figure 10.10 Examples of multispectral thresholding, using two channels X and Y . Top left: $(x \geq t_1 \text{ AND } y \geq t_2)$ Top Right: $(t_3 \leq x < t_4 \text{ AND } t_5 \leq y < t_6)$ Bottom left: $(t_7 \leq x < t_8 \text{ AND } (y < t_9 \text{ OR } y \geq t_{10}))$ Bottom right: $((x - t_{11})^2 + (y - t_{12})^2 < r^2)$, where r is the circle radius.

(plausible) edges of an original image, (2) process the edge image so that only closed object boundaries remain, and (3) transform the result to an ordinary segmented image by filling in the object boundaries. The edge detection step (1) has been discussed in chapter 9. The third step, the filling of boundaries, is not a difficult step, and an example of how this can be achieved is given for the technique below. The difficulty often lies in the middle step: transforming an edge (or edgeness) image to closed boundaries often requires the removal of edges that are caused by noise or other artifacts, the bridging of gaps at locations where no edge was detected (but there should logically be one) and intelligent decisions to connect those edge parts that make up a single object. The section on edge linking below addresses some of these difficulties. The next sections on watershed segmentation and active contours deal with methods that avoid having to link edge parts by manipulating –like a rubber band– contours that are always closed⁸ until they best fit an edge image.

In those rare cases where an edge image already shows perfect closed object boundaries, edge-based segmentation can be achieved by the following algorithm:

Algorithm: edge-based segmentation

Given an image f ,

⁸Or, at least, already linked.

1. Compute an edgeness image ∇f from f . Any preferred gradient operator can be used for this.
2. Threshold ∇f to an image $(\nabla f)_t$, so we have a binary image showing edge pixels.
3. Compute a Laplacian image Δf from f . Any preferred discrete or continuous Laplacian operator may be used.
4. Compute the image $g = (\nabla f)_t \cdot \text{sgn}(\Delta f)$.

The sgn operator returns the sign of its argument. The result image g will therefore contain only three values: 0 at non-edge pixels of f , 1 at edge pixels on the bright side of an edge, and -1 at edge pixels on the dark side of an edge.

The image g contains the boundaries of the objects to be segmented. The Laplacian is used to facilitate the final steps of the algorithm: turning the boundary image into a segmented image h containing solid objects. If we traverse the image g from left to right, two adjacent pixels with values -1 and 1 means we move into an object, and the values 1 and -1 means we move out of one. The image h can therefore be created by setting all pixel values to zero, except for those pixels that are between the transitions $1 \rightarrow -1$ and $-1 \rightarrow 1$ in each line of g , which are set to 1. If unique values are desired for each separate segment, a labelling algorithm can be run on h .

Figure 10.11 shows an example of applying this algorithm.

10.2.1 Edge linking

Edge detection very seldom gives you the perfect unambiguous and closed boundaries you need for a direct segmentation. There will frequently be spurious edges detected where they shouldn't be, and gaps occur where there should be edges. The latter problem of partial edges needs some form of *edge linking* to tie the available partial edges into an object boundary.

The Hough transform. If we have some idea of what the edges should look like (e.g., lines, circles, etc.) we can use the Hough transform (see chapter 8) to find the ideal edges that best fit the partial edges. This approach works best when the objects in the image have fairly simple (i.e., few parameters) parametric shapes. Finding complex shapes with the Hough transform is often a time-consuming task. Another disadvantage is that the Hough transform is inflexible in the sense that unexpected variations in object shapes cannot usually be coped with. When lines are used, some post-processing of the result is necessary to connect the correct line segments, as figure 10.12 shows.

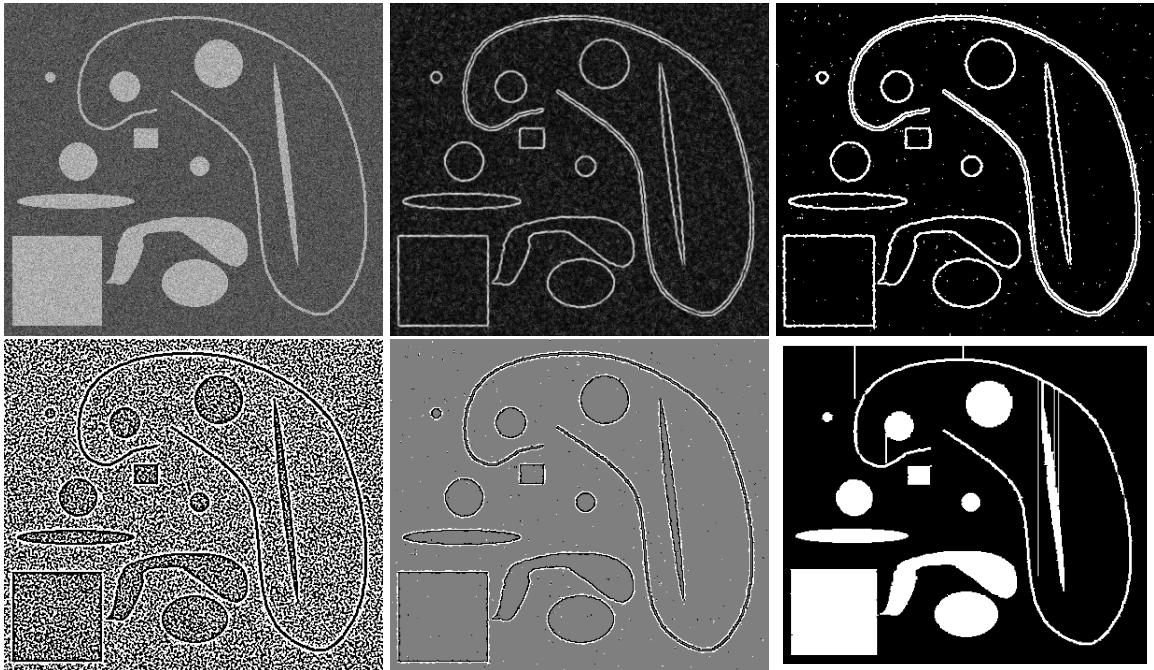


Figure 10.11 Example of edge-based segmentation. Top left: original 400×350 artificial image with added noise. Top middle: edgeness image; computed using a scale space operator (f_w) with $\sigma = 1$ pixel. Top right: same image after thresholding. Bottom left: sign of Laplacian image. Laplacian image computed using a scale space operator (f_{ii}) with $\sigma = 1$ pixel. Bottom middle: product of Laplacian sign and thresholded edge image. Bottom right: result after filling in of the boundaries as in the algorithm above. The noise was added to the original image to show some of the artifacts that they cause: notches in the edges of segmented objects, and lines where the filling algorithm encounters erroneous sign transitions. If no or less noise is added to the original, the segmentation result is perfect.

Neighborhood search. A local approach to edge linking is searching in a small neighborhood around an edge pixel for possible candidates to link the edge pixel to. A typical procedure is to select a starting edge pixel, find the best suitable link candidate in a small neighborhood, link this pixel to the first, then start a new search around the newly linked pixel, and so on. The process is stopped when one of the following occurs: (1) no suitable linking candidate is found, (2) the image boundary is reached, or (3) the pixel to be linked is already part of the current linked boundary.

Many criteria can be used to select a pixel B that is suitable to be linked to the current pixel A . For example:

- B must have a high edgeness value, and/or the edgeness value should be close to the edgeness value in A .

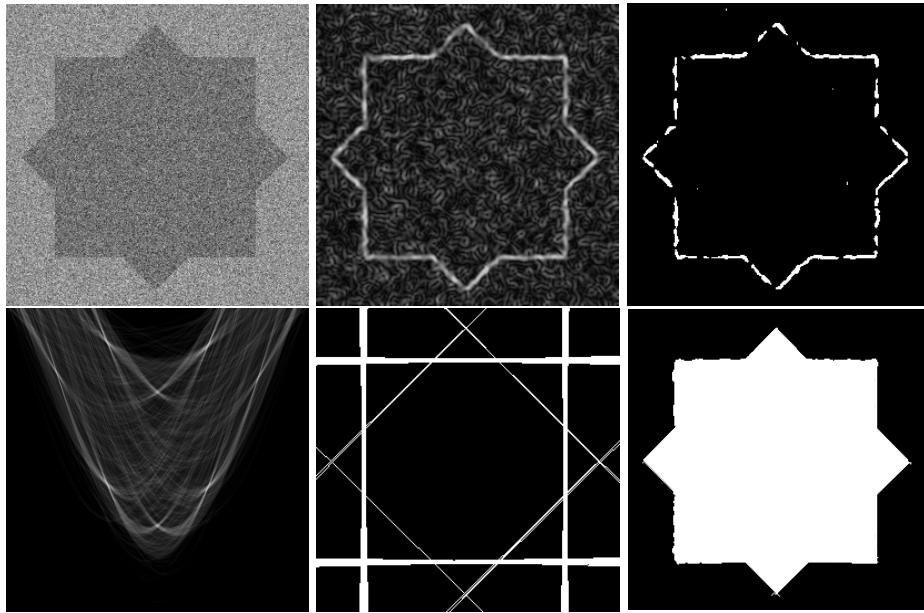


Figure 10.12 Example of edge linking using the Hough transform. Top left: original noisy 256×256 image. Top middle: edge image, computed using the f_w operator at a scale of $\sigma = 2$ pixels. Top right: threshold of this image, showing gaps in the object boundary. Bottom left: the Hough transform (lines) of the thresholded edge image, showing eight maxima. Bottom middle: the corresponding line image of the maxima in the Hough transform. Note that intelligent processing of this image is necessary to obtain only those line segments that make up the original object. (In this case we dilated the binary edge image, took the logical ‘and’ with the Hough lines image, and performed an opening by reconstruction of erosion.) Bottom right: result after processing and filling; we now have a closed boundary.

- The direction of the gradient in B should not differ too much from the gradient direction in A , and/or the direction from A to B should not differ too much from the direction of previous links.

The latter criterion ensures that edges cannot bend too much, which ensures that the search is kept on the right path when we reach a point where edges cross. It has also proven to be a necessary heuristic for noisy images. A disadvantage, however, is that edges at places where objects really have sharp angles are not linked. In these cases, a post-processing of all found linked boundary segments is necessary, where the boundary segments themselves are linked (using proximity and other criteria) to form closed boundaries. In addition, some form of post-processing is usually necessary to remove those linked edge segments that are unlikely to belong to realistic boundaries. The removal criteria are often application dependent, but commonly edge segments below a certain length are removed.

A problem by itself is the proper choice of starting points –the *seeds*– for the search pro-

cesses. These starting points may be indicated by a user or determined in an application-dependent way. It is also possible to use the (say) hundred points that have the highest edgeness value for starting points. This works well when the contrast of objects and background is about the same for all objects. If this is not the case, it may be that no starting points are picked at low-contrast objects. An approach that remedies this problem is to pick a hundred starting points with highest edgeness values, do edge linking, and then pick a new set of starting points (with highest edgeness), *disregarding all pixels tracked in the first run* (and possibly their neighborhoods too).

Many enhancements to the local search process have been proposed, and we give two of them here that address some common problems. A problem that often occurs is that tracked edge segments are more convoluted than they should realistically be, because of noise or the occurrence of thick edges. To ensure the edge tracking remains central to a thick edge, an additional criterion to use in the edge tracking is the demand that the *ridgeness of edgeness* is high. This is illustrated in figure 10.13.

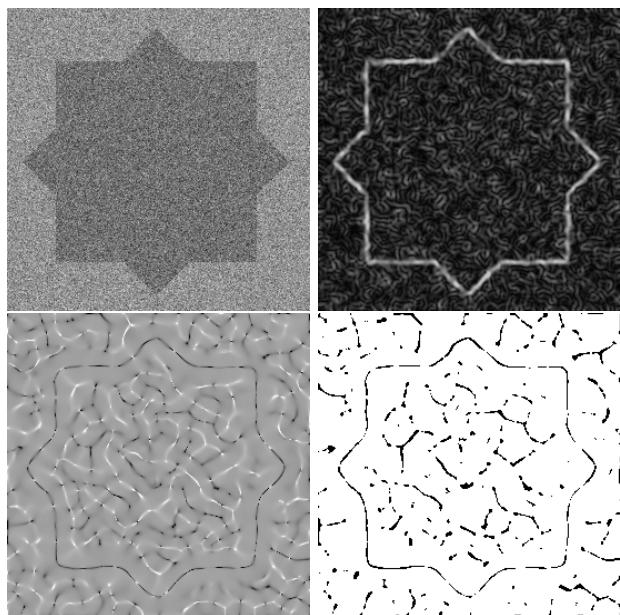


Figure 10.13 Noise and thick edges may cause edge tracking to meander. The use of a ridgeness in addition to (or instead of) the edgeness image in the tracking process may be beneficial, since the ridge is by definition a narrow structure. Top left: Original 256×256 noisy image. Top right: edgeness image (f_w , $\sigma = 2$ pixels). Note the thickness of the edges. Bottom left: ridgeness image of the edgeness image (f_{vv}/f_w , $\sigma = 4$ pixels). Bottom right: thresholded ridgeness image. Note that the ridges are much thinner than the edges.

Another problem is that noise may locally disrupt an edge, causing the edge tracking to locally follow a spurious edge branch –that usually ends quickly– instead of following the main edge. This can be addressed by the following adaptation of the search process:

instead of linking an edge pixel to the best linking candidate and then moving on to this candidate, we make a small list of suitable linking candidates. If linking of the best candidate results in a short dead end, we backtrack and see if the next best candidate results in a longer branch.

Network analysis: mathematical programming. If we not only have a good starting point A for a boundary segment, but also a likely end point B (or a set of potential end points), then the edge linking process is effectively the same as finding the best path from A to B . To find this best path, we first need to construct some network from which all possible paths from A to B can be gleaned. Figure 10.14 shows an example of how such a network can be constructed from a gradient image: each edge pixel is transformed to a network node –only pixels with an edgeness value above a certain threshold are part of the network– and arrows are drawn to all neighboring nodes (pixels) that can be reached from a certain node. An arrow is drawn if certain criteria are met, e.g., the gradient direction at the two nodes does not differ too much.

Our goal now is to find the best path, i.e., the best sequence of arrows, from A to B . ‘Best’ can be defined in several ways here, depending on the application. The best path may, e.g., be the shortest path, the path with highest average edgeness value, the path with least curvature, or a combination of these and yet other criteria. These criteria are included in our network by assigning costs to each arrow. For instance, if we are interested in the shortest path, we assign distances to each arrow: 1 to each horizontal or vertical arrow, $\sqrt{2}$ to each diagonal arrow. If we are interested in a path that is both short and runs along nodes with high edgeness values, we may assign an arrow cost that is a weighed sum of a distance term and a term inversely proportional to edgeness. The cost of each *path* from A to B is defined as the sum of costs of the arrows that make the path. The best path is the path with lowest cost.

The problem of finding the best path has been –and still is– an area of considerable research. It is known as the *shortest-route problem* in the field of operations research, and many algorithms for (approximately) solving the problem can be found. The problem solving strategy is known as mathematical programming.⁹ Other areas of research may present the problems under different headings, such as graph searching or network analysis. We present here a simple but effective algorithm for finding the best path.

Algorithm: Solving the shortest-route problem

Problem: find the shortest route from A to B given a network of nodes and branches, with given distances (costs) for each branch.

The idea of this iterative algorithm is to find the n -th nearest node to A , where n is the iteration number. For example, if $n = 1$, we find the nearest node to A . If $n = 2$, we find the second closest node to A , etc. The algorithm is terminated if the n -th closest node is the desired end node B .

⁹Mathematical programming also contains the techniques indicated as *dynamic programming* in other books in reference to the problem at hand.

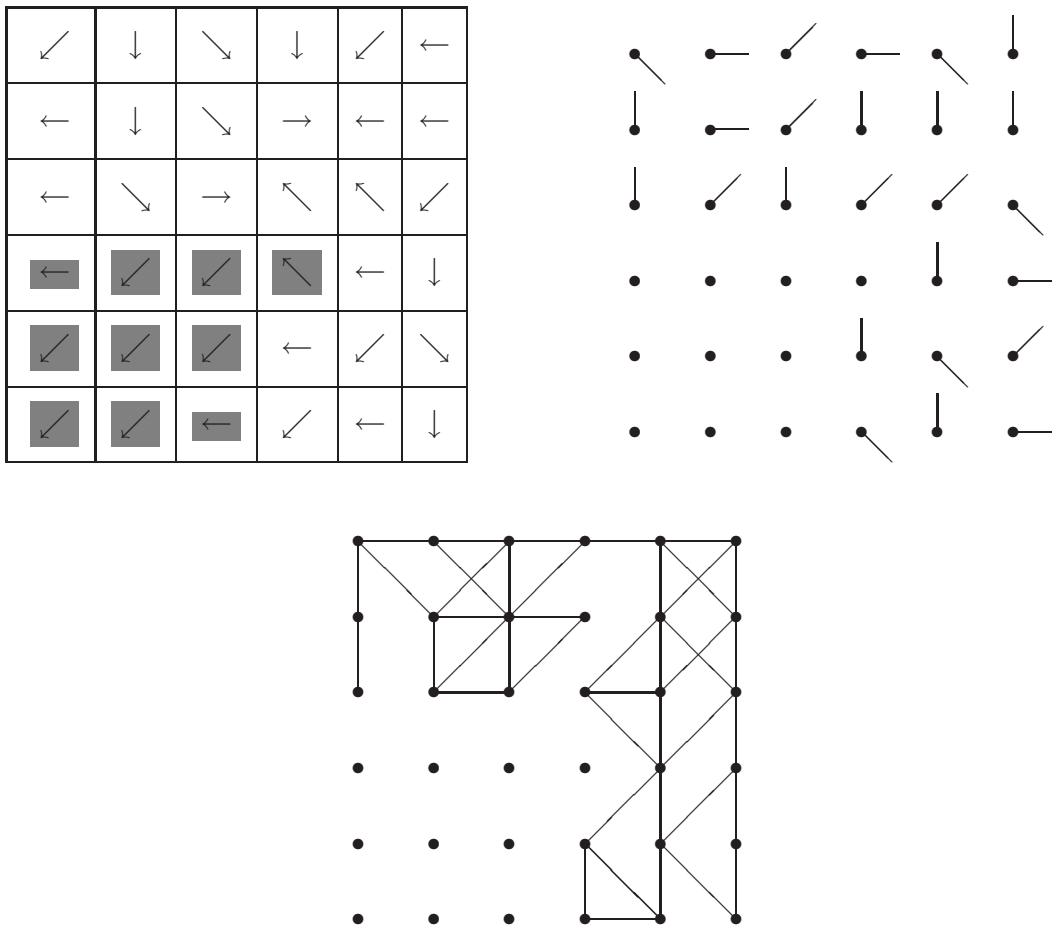
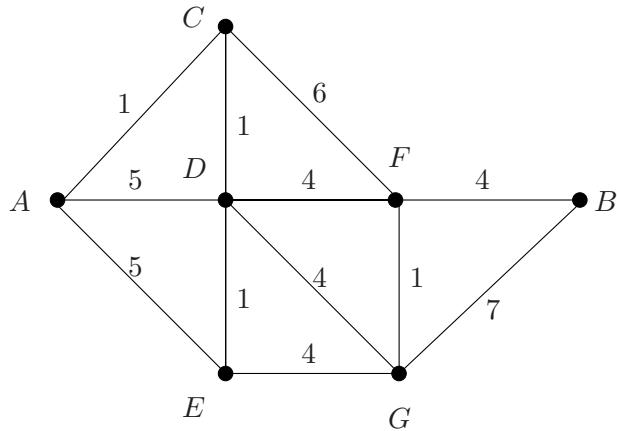


Figure 10.14 Example of creating a network from a gradient image. Top left: image with arrows indicating the gradient direction. (Actually, the nearest direction in the set of $\{0, 45, 90, 135, 180, 225, 270, 315\}$ degrees.) Pixels where the gradient (edgeness) value falls below a threshold are marked in grey. Top right: network created from the gradient image. Each pixel has been turned into a node. Lines show the significant gradients, turned 90 degrees so they are oriented along the edge at a node. Only the directions north, northeast, east, and southeast are used; they are unique modulo 180 degrees. Bottom: final network. For each node with an arrow in the top right network, we examine the nodes in the approximately along the edge direction of a node (not more than 45 degrees deviation). If the gradient direction of the new node differs by 45 degrees (modulo 180) or less from the current direction, a connection is drawn.

The input for iteration n is all $n - 1$ nearest nodes found in previous iterations (called solved nodes, all other nodes are called unsolved) that are connected to unsolved nodes. For each solved node, the shortest path to A and its length is known from previous iterations. Each of these input nodes provides a candidate for the n -th nearest node to be found in this iteration. This candidate is the unsolved node connected to the solved node by the shortest branch. The n -th nearest node is then the candidate with the shortest route to A .

Example

Given this network:



we apply the algorithm described above. At the start of the algorithm, the only solved node is A .

First iteration: we find candidate nodes by finding –for each solved node connected to unsolved nodes– the unsolved node connected by the shortest path. We have only one solved node: A . The closest unsolved connected node is C , with path length 1. C is added to the set of solved nodes.

Second iteration: for the solved node C , the candidate node now is D (closest connected unsolved node), with path length 2. For the solved node A , we have two candidates: D and E , because both can be reached by a branch of length 5. From all candidates, we select D as second-nearest node, because path length 2 is the shortest.

Third iteration: solved node A provides candidate E , solved node C provides candidate F , solved node D provides candidate E . Path lengths are 5, 7, and 3 respectively, so E becomes the third-nearest node.

Fourth iteration: solved node A can no longer provide candidates: there are no connected unsolved nodes. Node C provides F , D provides F and G , and E provides G , with path lengths of 7, 6, 6, and 7 respectively. There is a tie between nodes F and G ; both are fourth-nearest node.

Fifth iteration: only F and G of the solved nodes have a connected unsolved node, B in both cases. The path lengths are 10 and 13 respectively. B is fifth-nearest node, and the algorithm terminates.

Backtracking provides us with the optimal path: from B to F to D to C to A ; or the right way round: $ACDFB$.

The table below summarizes the course of the algorithm:

n	solved nodes connected to unsolved nodes	closest connected unsolved node	path length	n -th nearest node
1	A	C	1	C
2	A	D	5	
		E	5	
	C	D	$1 + 1 = 2$	D
3	A	E	5	
	C	F	$1 + 6 = 7$	
	D	E	$2 + 1 = 3$	E
4	C	F	$1 + 6 = 7$	
	D	F	$2 + 4 = 6$	F
		G	$2 + 4 = 6$	G
	E	G	$3 + 4 = 7$	
5	F	B	$6 + 4 = 10$	B
	G	B	$6 + 7 = 13$	

10.2.2 Watershed segmentation

In chapter 8 we introduced the watershed transformation as a means to separating overlapping objects. A watershed is formed by ‘flooding’ an image from its local minima, and forming ‘dams’ where waterfronts meet. When the image is fully flooded, all dams together form the watershed of an image. The watershed of an edgeness image (or, in fact, the watershed of the original image) can be used for segmentation. Figure 10.15 shows how this works. The idea is that when visualizing the edgeness image as a three-dimensional landscape, the catchment basins of the watershed correspond to objects, *i.e.*, the watershed of the edgeness image will show the object boundaries. As the landscape in the figure shows, the object boundaries mark the catchment basins, but there are small defects because of image artifacts. Because of the way the watershed is constructed –it forms ‘dams’ where waterfronts meet– these small defects do not disturb the watershed segmentation much.

Typically, some pre- and post-processing of the watershed image is needed to obtain a good segmentation. Usual post-processing is the filling in of the boundaries to obtain

solid segments. In the case of watershed segmentation, we do not need to worry about small ‘leaks’ in a boundary (a small leak between two segments will fill in the two segments as being a single segment). Because of the way the watershed is constructed, leaks cannot occur.

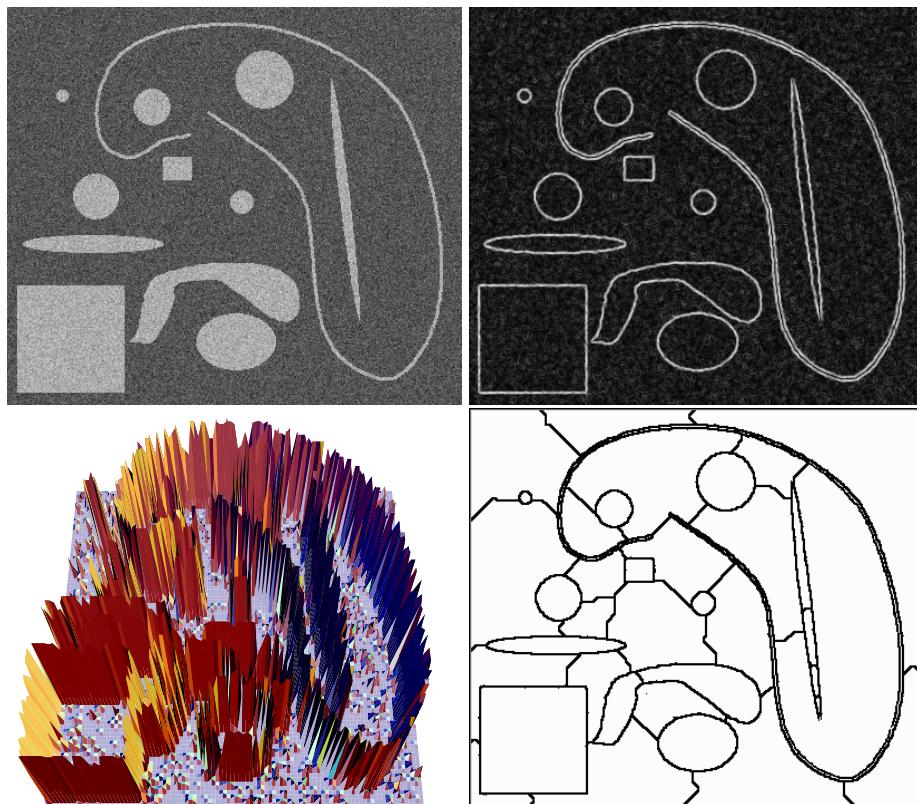


Figure 10.15 Example of edgeness watershed segmentation. Top left: original noisy image. Top right: edgeness image. Bottom left: landscape version of the edgeness image. Note that objects are surrounded by a ‘palisade’, that will hold the water in the flooding process that forms the watershed. Dams will be formed when the water level reaches the tops of the palisades. An uneven palisade height will not disturb the formation of the correct dam marking an object boundary. Bottom right: all dams formed; the watershed of the edgeness image.

Some pre- and post-processing is usually necessary to avoid an oversegmentation. In the example in our figure, we set all grey values below a threshold to zero. This ensures no dams are formed early in the flooding process. Since such dams are caused by very weak edges, they are unlikely to correspond to object boundaries, and should not be part of the segmentation watershed. Our example still shows some oversegmentation: besides the correct object boundaries, several erroneous branches are present. Many approaches have been proposed to remove these erroneous branches. One effective method is to merge those segments that are likely to belong together. How this likelihood is measured is discussed in the section on region based segmentation. An-

other method that is effective in our example is to compute the average edgeness value of each branch, and remove those branches where this average falls below a threshold. Figure 10.16 shows that this effectively removes all spurious branches from our example.

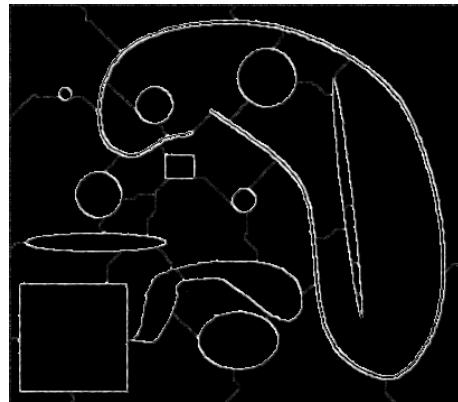


Figure 10.16 Example of removing the spurious branches of the edgeness watershed to reduce oversegmentation of the original image. The image shows a multiplication of the edgeness image and its watershed from the previous figure, clearly showing that the erroneous branches can be easily removed by thresholding the average edgeness of branches.

10.2.3 Snakes

An *active contour* or *snake* is a curve defined in an image that is allowed to change its location and shape until it best satisfies predefined conditions. It can be used to segment an object by letting it settle –much like a constricting snake– around the object boundary.

A snake C is usually modeled as a parametrized curve $C(s) = (x(s), y(s))$, where the parameter s varies from 0 to 1. So, $C(0)$ gives the coordinate pair $(x(0), y(0))$ of the starting point, $C(1)$ gives the end coordinates, and $C(s)$ with $0 < s < 1$ gives all intermediate point coordinates. The movement of the snake is modeled as an energy minimization process, where the total energy E to be minimized consists of three terms:

$$E = \int_0^1 E(C(s))ds = \int_0^1 (E_i(C(s)) + E_e(C(s)) + E_c(C(s)))ds.$$

The term E_i is based on internal forces of the snake; it increases if the snake is stretched or bent. The term E_e is based on external forces; it decreases if the snake moves closer to a part of the image we wish it to move to. For example, if we wish the snake to move to

edges, we may base this energy term on edgeness values. The last term E_c can be used to impose additional constraints, such as penalizing the creation of loops in the snake, penalizing moving too far away from the initial position, or penalizing moving into an undesired image region. For many applications, E_c is not used, *i.e.*, simply set to zero. Common definitions for the internal and external terms are

$$\begin{aligned} E_i &= c_1 \left\| \frac{dC(s)}{ds} \right\|^2 + c_2 \left\| \frac{d^2C(s)}{ds^2} \right\|^2 \\ E_e &= -c_3 \|\nabla f\|^2, \end{aligned}$$

where the external term is based on the assumption that the snake should be attracted to edges of the original image f . By using other external terms, we can make use of different image features, making the snake follow ridges, find corner points, etc. The constants c_1 , c_2 , and c_3 determine the relative influence of each term on the movement of the snake. The elasticity of the snake is controlled by c_1 ; the norm of the derivative $\frac{dC(s)}{ds}$ measures how much the snake is stretched locally, so setting c_1 to zero will result in high stretch values having no impact on the energy, hence the snake may stretch infinitely. Alternatively, a high (relative to the other c 's) value of c_1 makes that the snake can stretch very little. In the same manner, c_2 controls how much the snake can bend (*i.e.*, its stiffness or rigidity), since the second derivative $\frac{d^2C(s)}{ds^2}$ is a measure for the snake's curvature, and c_3 sets the relative influence of the edge attraction force.

When implementing a snake, it is commonly modeled by a connected series of splines. This model allows the snake –on the one hand– to be easily connected to the mathematical model: the spline representation is easily converted to the parametrized curve $C(s)$, and its derivatives can also be computed in a straightforward manner. On the other hand, the discrete set of control points of the spline give a handle to optimizing the snake; only the control points need to be moved, and the set is easily related to discrete images. The optimization itself is not usually carried out by direct minimization of the energy functional E , but by numerically solving its Euler-Lagrange form:

$$-\frac{d}{ds} \left(c_1 \left\| \frac{dC(s)}{ds} \right\|^2 \right) + \frac{d^2}{ds^2} \left(c_2 \left\| \frac{d^2C(s)}{ds^2} \right\|^2 \right) + \nabla(E_e + E_c) = 0.$$

Figure 10.17 shows an example of a snake used to find the boundary of an object.

For a snake to converge to the desired location and shape, we need a good starting estimate in most practical applications. This can be provided by the user, or by processing and linking of extracted edges as described in the previous section. To improve the chances of the snake converging to the desired shape, many current algorithms have added features to the original snake algorithm, such as:

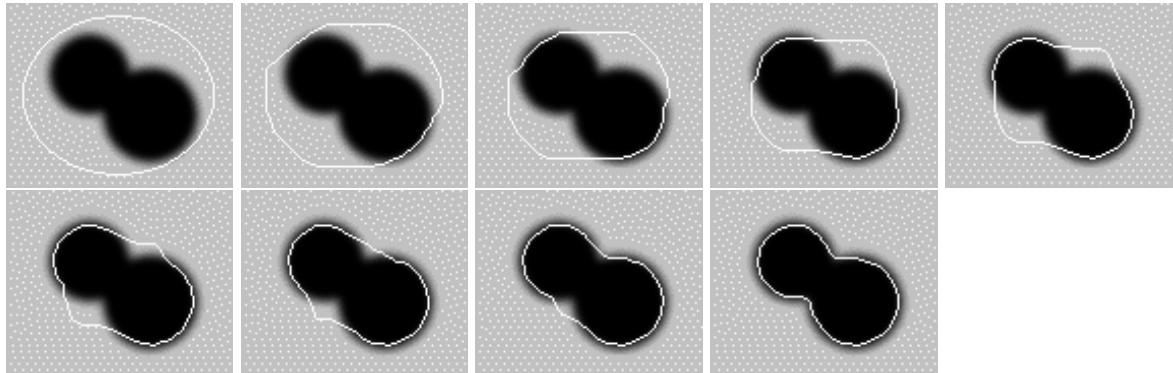


Figure 10.17 Example of a snake. The sequence of images shows the evolution of the snake from a user-defined original position to its final state.

- An inflation force: if the initial snake is too far to the inside of a boundary, the inflation force (like in a balloon) causes it to move closer and avoid getting stuck on local spurious edges. For an external snake, a deflating force can be used.
- The possibility of a snake to split itself into separate snakes.
- A ‘growing’ ability: if a snake stretches too much locally, extra control points can be inserted there.
- Added region criteria to closed snakes: this entails that besides the usual energy terms, energy terms that measure aspects of the *region* enclosed by a snake are added. For example, we may demand that the region has minimal grey value variance (which ensures that the snake encloses a homogeneous area), or that the entropy or a higher central moment of the histogram of the enclosed region is minimal.

10.2.3.1 Implicit snakes*

We usually regard a discrete image as a collection of grey values arranged on a grid. But if we are especially interested in image contours, it may be helpful to think of the image as a set of contours. The most natural set of contours that describe an image are its isophotes; the iso-height lines if the image is viewed as a landscape. The complete set of isophotes fully describes an image. Figure 10.18 shows some isophotes of an image.

In the classical snake approach described in the previous section, the initial snake is defined explicitly, and is then evolved subject to internal and external forces. An alternate approach to snakes is to evolve a natural image curve –say, an isophote– subject to internal and external forces. This is called an *implicit* snake, since it is implicitly defined by the geometrical image structure. The evolution of an implicit snake is also different from the classical explicit snake; where the classical snake ‘crawls’ over the image to



Figure 10.18 An image and some of its isophotes. The full collection of isophotes gives a complete description of the image.

its final destination and shape, in the case of the implicit snake, we modify the image itself. For example, suppose our implicit snake is the isophote defined by¹⁰ grey value 100. Then the evolved snake is obtained by processing the image and then extracting isophote 100.

An implicit snake is not necessarily an isophote. It may be any curve defined in terms of the geometry of the image. A common mathematical way of expressing the implicit snake is to define it as the zero-level curve of a function g on the image f : $g(f) = 0$, where g is modeled so it extracts the isophote, flowline, or whatever curve you want.¹¹ For the sake of simplicity, we will show here only the evolution of implicit snakes defined by isophotes.

Controlling the evolution of the implicit snake is the tricky bit in the process here. We have simply said that evolving the snake is done by processing the image, but what processing is necessary to make the snake evolve the way we want? The next paragraphs deal with this problem.

If we have a curve C parametrized by s , then $\frac{\partial C}{\partial t}$ denotes the evolution (change, movement) of the curve over time t . This movement can be locally decomposed into movement *along* the curve, and movement *perpendicular* (normal) to the curve:

$$\frac{\partial C}{\partial t} = \alpha(s, t)\mathbf{T} + \beta(s, t)\mathbf{N},$$

¹⁰The isophote at value x can be extracted from a grey-valued image f by thresholding it at value x and then taking the contours of the binary result.

¹¹An additional constraint is that $\nabla g(f) \neq 0$.

where \mathbf{T} and \mathbf{N} are the local tangential and normal unit vectors to the curve, and α and β are arbitrary functions. Tangential movement, however, does not alter the curve,¹² so we can effectively write the curve evolution as an evolution in the normal direction only:

$$\frac{\partial C}{\partial t} = \beta'(s, t)\mathbf{N},$$

where β' depends only on α and β . Without proof, we note here that if the curve C is an isophote of the image f , then the curve evolution can be written in terms of *image* evolution by the equivalence:

$$\frac{\partial C}{\partial t} = \beta'(\cdot)\mathbf{N} \iff \frac{\partial f}{\partial t} = -f_w\beta'(\cdot),$$

where we use the local gradient-based coordinate system (v, w) defined in chapter 9. The arguments of β' on the left side must be ‘translated’ to arguments in image form. This is not a trivial matter in general. A frequently occurring argument is the isophote curvature κ , which (as we have seen before) can be expressed as $-f_{vv}/f_w$. Table 10.1 lists some frequently used curve evolutions. Note that we have seen the image evolution equations before as the equations driving non-linear diffusion in chapter 9.

Curve evolution	Image evolution
$\frac{\partial C}{\partial t} = c\mathbf{N}$	$\frac{\partial f}{\partial t} = cf_w$
$\frac{\partial C}{\partial t} = \kappa\mathbf{N}$	$\frac{\partial f}{\partial t} = f_{vv}$
$\frac{\partial C}{\partial t} = (\alpha + \beta\kappa)\mathbf{N}$	$\frac{\partial f}{\partial t} = \alpha f_w + \beta f_{vv}$
$\frac{\partial C}{\partial t} = \kappa^{1/3}\mathbf{N}$	$\frac{\partial f}{\partial t} = f_{vv}^{1/3}f_w^{2/3}$

Table 10.1 Some examples of implicit snake curve evolution equations and the corresponding image evolution equations.

A type of implicit snake that can be used well for many segmentation tasks is controlled by the equation $\frac{\partial C}{\partial t} = g\kappa\mathbf{N} - \nabla g$, where g is a function that decreases with edgeness (such as $g = \exp\left(\frac{-f_w^2}{k^2}\right)$), and κ is the isophote curvature. The image evolution form is $\frac{\partial f}{\partial t} = g f_{vv} + \nabla g \cdot \nabla f$. Without going into the mathematical explanation, we note that this snake will be attracted to edges in such a way that evolving an image will result in an image where segmentation can optimally be done using simple thresholding. Figure 10.19 shows an example of this.

¹²Only its parameterization.

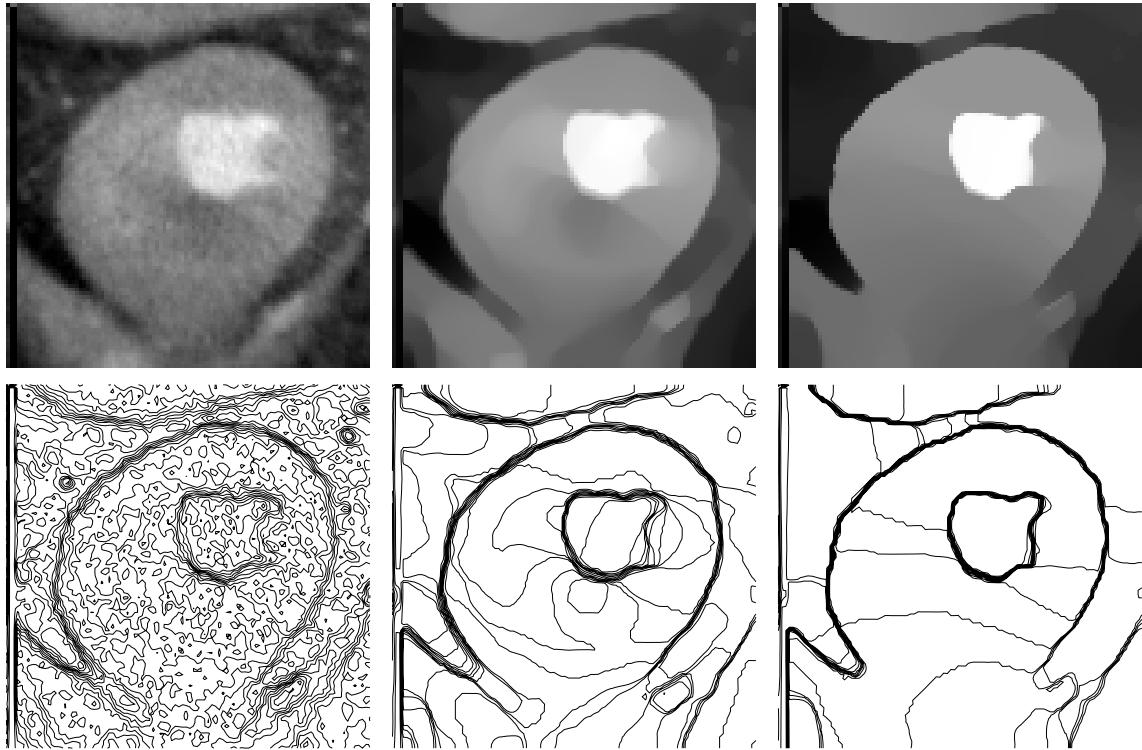


Figure 10.19 Example of implicit snakes. Top left: original image. Top middle: intermediate stage of the implicit snake algorithm mentioned in the text. Right: final stage. Bottom row: corresponding snakes; the isophotes at twenty separate grey levels.

There are several advantages of implicit over explicit snakes. First, there are less topological restrictions; the topology of the snake may even change (e.g., the snake of a level set may start out as two separate contours that merge in the evolution). Second, most evolution schemes allow the simultaneous evolution of more than one snake, possibly *all* level sets (say, all isophotes) of an image. Last, the image evolution schemes can directly be applied to three-dimensional (or higher dimensional) images, while the extension of classical snakes to three dimensions is a far-from-easy task.

10.3 Region based segmentation

In the previous section, objects were found by locating their boundaries. In this section, we discuss the dual approach of finding the object *region* instead of its edges. In theory, finding an object by locating its boundary and finding it by establishing the region it covers will give you exactly the same object; the boundary and the region are just different representations of the same object. In practice, however, taking an edge based ap-

proach to segmentation may give radically different results than taking a region based approach. The reason for this is that we are bound to using imperfect images and imperfect methods, hence the practical result of locating an object boundary may be different from locating its region.

Region based segmentation methods have only two basic operations: *splitting* and *merging*, and many methods even feature only one of these. The basic approach to image segmentation using merging is:

1. obtain an initial (over)segmentation of the image,
2. merge those adjacent segments that are similar –in some respect– to form single segments,
3. go to step 2 until no segments that should be merged remain.

The initial segmentation may simply be all pixels, *i.e.*, each pixel is a segment by itself. The heart of the merging approach is the similarity criterion used to decide whether or not two segments should be merged. This criterion may be based on grey value similarity (such as the difference in average grey value, or the maximum or minimum grey value difference between segments), the edge strength of the boundary between the segments, the texture of the segments, or one of many other possibilities.

The basic form of image segmentation using splitting is:

1. obtain an initial (under)segmentation of the image,
2. split each segment that is inhomogeneous in some respect (*i.e.*, each segment that is unlikely to *really* be a single segment).
3. go to step 2 until all segments are homogeneous.

The initial segmentation may be no segmentation at all, *i.e.*, there is only a single segment, which is the entire image. The criterion for inhomogeneity of a segment may be the variance of its grey values, the variance of its texture, the occurrence of strong internal edges, or various other criteria. The basic merging and splitting methods seem to be the top-down and bottom-up approach to the same method of segmentation, but there is an intrinsic difference: the merging of two segments is straightforward, but the splitting of a segment requires we establish suitable sub-segments the segments can be split into. In essence, we still have the segmentation problem we started with, except it is now defined on a more local level. To avoid this problem, the basic splitting approach is often enhanced to a combined *split and merge* approach, where inhomogeneous segments are split into simple geometric forms (usually into four squares) recursively. This of course creates arbitrary segment boundaries (that may not be correlated to realistic boundaries), and merge steps are included into the process to remove incorrect boundaries.

10.3.1 Merging methods

Region growing. Many merging methods of segmentation use a method called *region growing* to merge adjacent single pixel segments into one segment. Region growing needs a set of starting pixels¹³ called *seeds*. The region growing process consists of picking a seed from the set, investigating all 4-connected neighbors of this seed, and merging suitable neighbors to the seed. The seed is then removed from the seed set, and all merged neighbors are added to the seed set. The region growing process continues until the seed set is empty. The algorithm below implements an example with a single seed, where all connected pixels with the same grey value as the seed are merged.

Algorithm: Region growing

The data structure used to keep track of the set of seeds is usually a *stack*. Two operations are defined on a stack: *push*, which puts a pixel (or rather, its coordinates) on the top of the stack, and *pop*, which takes a pixel from the top of the stack.

In the algorithm, the image is called f , the seed has coordinates (x, y) and grey value $g = f(x, y)$. The region growing is done by setting each merged pixel's grey value to a value h (which must not equal g). The pixel under investigation has coordinates (a, b) . The algorithm runs:

1. push (x, y)
2. as long as the stack is not empty do
 - (a) pop (a, b)
 - (b) if $f(a, b) = g$ then
 - i. set $f(a, b)$ to h
 - ii. push $(a - 1, b)$
 - iii. push $(a + 1, b)$
 - iv. push $(a, b - 1)$
 - v. push $(a, b + 1)$

The final region can be extracted from the image by selecting all pixels with grey value h . To ensure the correctness of the result, we must select h to be a value that is not present in the original image prior to running the algorithm. The statement that decides if a pixel should be merged ('if $f(a, b) = g$ ') can be modified to use a different merging criterion. A simple modification would be to allow merging of pixels that are in a certain range of grey values ('if $l < f(a, b) < h$ '). An example of this is shown in figure 10.20.

¹³Which usually contains only a single pixel.

Another modification is to allow merging only if the gradient value of the candidate pixel is low (a high gradient could signify we are at the edge of an object). This last criterion is often combined with a grey value range criterion to ensure the region growing does not grow right through a weak spot in the boundary where the gradient value is low. More elaborate merging criteria can be used, but they do not usually result in as good a segmentation as may be hoped for, since the heart of region growing is the evaluation of the criterion at a single pixel –the merging candidate– which may be prone to noise. Elaborate criteria are usually more effectively used when evaluating whether to merge two segments of a larger size.



Figure 10.20 Example of region growing based on a grey level range. On the left, an original 256×256 image is shown, with a grey level range of 256. On the right, the result of region growing with the seed roughly at the center of the structure, allowing a grey value range of ± 30 around the grey value of the seed.

Region merging. In this section, we will assume we have the ‘larger size’ segments as meant above available, but we still have an oversegmentation of the image, so we still need to do region merging to obtain a proper segmentation. Such an oversegmentation can, e.g., be obtained by

- a watershed segmentation such as in figure 10.21, or
- a multiple thresholding as in figure 10.4 –but with many more thresholds– followed by a labeling step,
- using implicit snakes as in figure 10.19 (with segments defined by the contours on the bottom right), or
- a series of region growings –with, say, using the grey value range criterion– with $(0, 0)$ as the first seed, and subsequent seeds picked from the unsegmented image parts until no pixels remain unsegmented, or

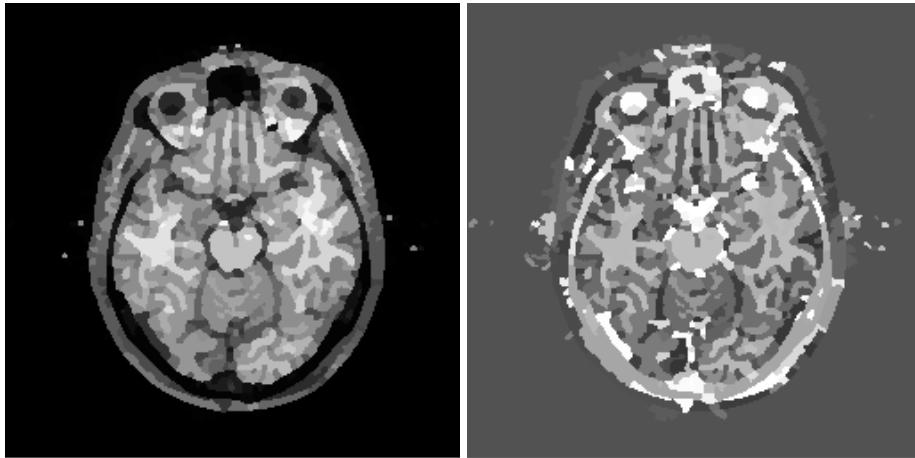


Figure 10.21 Example of an oversegmentation (left) obtained by the watershed of a gradient image of the original. Each segment is visualized here by a grey value equal to the average of the original values of the segment. The right image shows a random grey value remapping of the left image to better display the fact that it is an oversegmentation of the original.

- anything else you can think of.

The oversegmentation can be reduced to a better segmentation by merging adjacent segments. A merging of two adjacent segments can be achieved by removing their common boundary. Two types of criteria exist to judge whether two segments should be merged: those evaluating the edge strength of the common boundary, and those comparing region characteristics of the two segments. Some example heuristics considering edge strength are: remove a common boundary if

- the average edgeness is too low: $\frac{e}{l} < t_1$, where e is the sum of edgeness values of pixels in the common boundary with length l , and t_1 is a threshold,
- the common boundary contains too many weak edge pixels: $\frac{w}{l} > t_2$, with w the number of weak edge pixels (all pixels in the common boundary with edgeness value below a threshold), and t_2 a threshold,
- the common boundary contains too many weak edge pixels compared to the smallest segment size: $\frac{w}{\min\{l_1, l_2\}} > t_3$, where l_1 and l_2 are the respective adjacent segment perimeters, and t_3 is a threshold.

Heuristics considering region characteristics are commonly used combined with edge heuristics, since it is possible that two very similar regions are separated by a strong edge, in which case merging is usually not desired. Example of region heuristics are: remove a common boundary if

- the grey values of the adjacent regions are similar. Various measures can be used to measure this, e.g.,
 - the average grey value of the two segments differs by less than a threshold: $|\bar{g}_1 - \bar{g}_2| < t_1$,
 - the maximum (or minimum) grey value difference found between the two segments is below a threshold: $\max_{(x_1, y_1) \in S_1, (x_2, y_2) \in S_2} |g(x_1, y_1) - g(x_2, y_2)| < t_2$,
 - in addition to one of the above measures: the variance of grey values in each segment must be similar: $|\sigma^2(S_1) - \sigma^2(S_2)| < t_3$, or $|\Delta(S_1) - \Delta(S_2)| < t_4$,
- the texture of the regions is similar. Example of texture measures are treated in a separate section later in this chapter.
- the histogram of the adjacent segments is similar. The similarity can be measured by comparing scalar histogram features such as (central) moments or entropy of the histogram. Note that comparing first and second (central) histogram moments boils down to the same average and variance grey value criteria as described above.

In the above, S_1 and S_2 denote the respective segment regions, g is a grey value, \bar{g}_i is the average grey value of segment i , t_j is a threshold, $\sigma^2(S_i)$ denotes the grey value variance, and $\Delta(S_i)$ is the maximum grey value difference occurring in segment i . Combinations of the criteria above are also possible. The removal of common boundaries is usually done recursively: all common boundaries in an image are considered for removal, those that do not meet the set criteria are removed, and the process is repeated until no more common boundaries are removed. Note that the *order* in which the common boundaries are removed may have an impact on the merging process: suppose we have three segments S_1 , S_2 , and S_3 , with the pairs (S_1, S_2) and (S_2, S_3) adjacent. Then merging S_1 and S_2 to a new segment S_{12} may cause that S_{12} and S_3 are unsuitable for merging, while previously S_2 and S_3 were to be merged according to the criteria set. In practice, it is often a good idea to start by removing the weakest boundary, re-evaluating the boundary strengths, again remove the weakest, etc. .

10.3.2 Splitting and split & merge methods

Where region merging is an agglomerative approach, region splitting is divisive. We mentioned before that this difference makes that the two approaches are not opposites, but fundamentally different problems; the merging of two segments is straightforward, but the splitting of a segment requires that suitable sub-segments are established to split the original segment into. The problem of *how* to split a segment is of course itself a segmentation problem, and we can treat it as such: any segmentation method (such as the ones treated in this chapter) can be applied to the segment to establish sub-segments. Besides the hierarchical level, there is no intrinsic difference. The problem of how to decide if a

segment need splitting can be solved using the same measures of region homogeneity mentioned in the section on region merging, e.g., a segment needs to be split if

- the grey value variance exceeds a threshold, or
- the variance of a texture measure exceeds a threshold, or
- the histogram entropy (or another histogram measure) exceeds a threshold, or
- high edgeness pixels are present.

When a segment needs splitting, a faster approach than starting a segmentation on the segment is to simply split it into four quadrants. This of course has a high probability of placing segment boundaries at unrealistic positions. This problem is reduced by adding merging operations to the segmentation process; recursively splitting and merging the image segments in a way such as the one used in figure 10.22. Especially if different

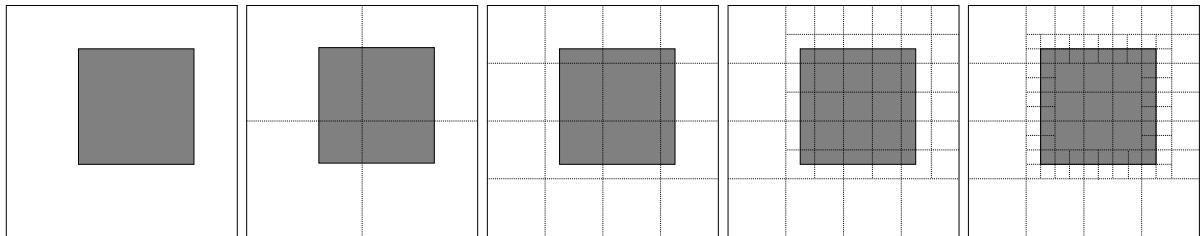


Figure 10.22 Example of split and merge segmentation. Left: original image. From left to right: recursively splitting each segment into four segments when the grey value variance of a segment is non-zero. If we subsequently recursively merge all adjacent segments with the same (average) grey value, we end up with two segments; exactly the desired segments of object and background.

criteria are used for splitting and merging –for example, edge heuristics for the merging and region homogeneity for splitting– the split and merge segmentation may not converge to a stable result, but instead keep alternating between various solutions. This problem can sometimes be avoided by modifying the criteria in the course of the process, but is usually simply ignored by terminating the process after a fixed number of iterations. Another problem of the ‘split into quadrants and merge’ approach is that the resulting segmentation may be very blocky. Figure 10.23 shows examples of using this approach.

10.3.3 Pyramid, tree, and scale space methods

Trees are often used as data structures that represent segmentation by splitting and merging. For example, figure 10.24 shows there is an easy one-to-one relationship be-

tween splitting an image recursively into quadrants and a quadtree, as we have seen previously in section 8.3.7. The quadtree is often a more convenient and efficient data structure to use in a segmentation by splitting than the image itself. However, using a tree of this type is limited, because we cannot –for instance– model a merging of two segments on different levels of the pyramid. A more common use of a tree structure is to use it in combination with a multiresolution pyramid (or a scale space): the pyramid level defines the tree level, each pixel in the pyramid defines a tree node, and branches can be formed between nodes in adjacent levels. In this context, a branch is usually called a *link*, the high-level end a *parent*, and the other end a *child*. An example of such a pyramid-tree can be seen in figure 10.25. Ways to construct a pyramid have been discussed in section 9.3.

There are many ways to construct and use a pyramid-tree for segmentation, but all approaches share three steps:

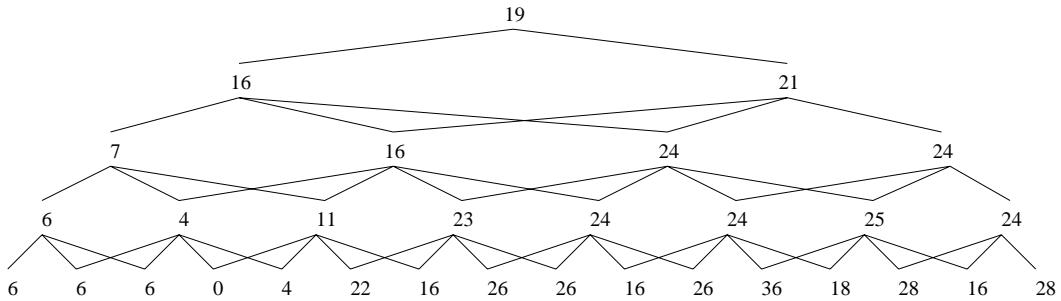
- creating the multiresolution pyramid,
- creating links,
- finding nodes whose leaves (i.e., connected nodes on the base pyramid level) identify a segment. Such nodes are called *roots*.

Many methods iterate between the various steps, e.g., the pyramid may be re-created using the results of the first linking step. In this section, we will show two examples of pyramid-tree segmentation; a classical approach (Burt-pyramid) and an advanced one using scale space techniques.

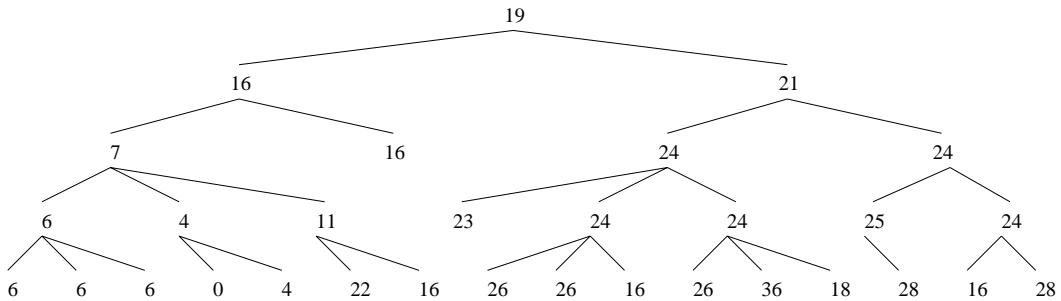
A classical example. The initial pyramid is created by averaging of the grey values of each 4×4 neighborhood. Links are created upwards from the base level of the pyramid. Each child is given exactly one link upwards, to the candidate parent in the 4×4 neighborhood that is closest in grey value. The pyramid is then re-created by assigning to each pixel the average grey value of its children—the base level remains unmodified. The linking and re-creation is iterated to stability. A segmentation is obtained by going down the final pyramid-tree until the number of nodes on the current level equals the pre-defined number of desired segments. These nodes are then assigned to be the segment roots, and the leaves of their sub-trees constitute the various segments.

Example

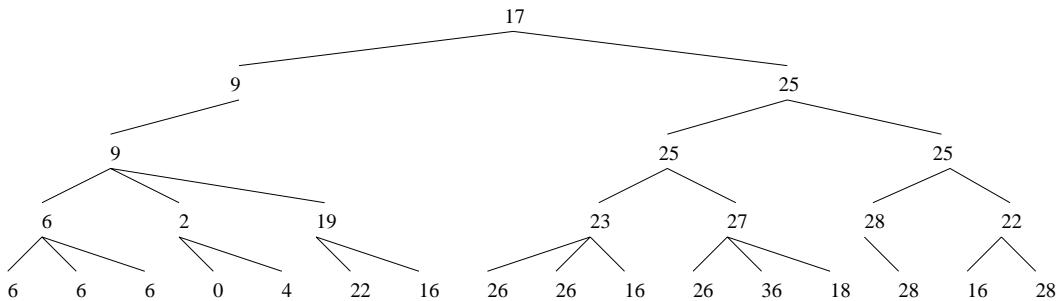
For reasons of clarity, we show here a one-dimensional example of the algorithm described above. We have image data of a noisy edge; the subsequent 16 grey values are $\{6, 6, 6, 0, 4, 22, 16, 26, 26, 16, 26, 38, 18, 28, 16, 28\}$. From this data, we can construct the following pyramid by averaging each 4 grey values. Each pyramid level reduces resolution by a factor of two. Lines show which children were used in the computation of each parent.



Now we link each child to the parent in the 4×4 neighborhood on the current level that is closest in grey value. In case of a tie between candidate parents, we link to the spatially closest parent:



Actually, an additional heuristic was used in the linking process that avoids the crossing of links ('22' and '36' grey values on the base level). When this happens, one of the crossing links is destroyed, and the child is linked to the next closest parent. Now we recompute the pyramid values: each parent is given the average value of its children. Parents without children are deleted from the tree.



Since a repetition of the linking process does not alter the tree, we terminate the process. The segmentation of the image can now be done by bunching all the leaves of the sub-trees at an appropriate pyramid level. For example, if we desire two segments, we take the second level from the top: the sub-tree with root '9' defines one segment, and the sub-tree with root '25' defines the other. Segmentation at lower pyramid levels gives us respectively three, seven, and 16 segments.

A pyramid-tree segmentation using scale space. In this example, the pyramid levels are taken from a scale space built of the original image. The advantage of using scale space is that it is easier to control the reduction of resolution between two pyramid levels. In the pyramid in the previous example, the resolution in each spatial dimension was reduced by a factor of two if we move up a level in the pyramid. In a formula:¹⁴ $N_n = \frac{1}{2}N_{n-1}$, where N_i is the number of pixels in one dimension at level i of the pyramid, the base level being level 0. In terms of the original number of pixels, this becomes $N_n = (\frac{1}{2})^n N_0$. In scale space, the story is a bit more complex: although scaling an image lowers its intrinsic resolution, it is common practice to represent the scaled image using the same number of pixels as in the original image, instead of the more natural number of pixels that is inversely proportional to the scale of the image.¹⁵ This is not very practical in our case; we need a decreasing number of pixels as we rise in the pyramid/scale space. To achieve this, we subsample scaled images. The correct subsampling rate and the correct scale σ_n of each pyramid level are related by the equations:

$$\begin{aligned}\sigma_n &= \varepsilon e^{\tau_0+n\delta} \\ N_n &= N_0 e^{-n\delta},\end{aligned}$$

where δ is a scale sampling parameter and ε is the pixel distance in the original image. The constant τ_0 is used to relate ε to the inner scale of the original image. Often τ_0 is assumed to be zero, which gives us $\sigma_0 = \varepsilon$ for the original image, *i.e.*, the inner scale of the original image equals the pixel size. The example below shows how we can build a pyramid for segmentation based on scale space.

Example

Suppose we wish to construct a pyramid using scale space where the resolution in each spatial dimension is reduced by a factor of two if we go up a level. Then the following holds (see text above for an explanation): $N_n = (\frac{1}{2})^n N_0$. We also know that $N_n = N_0 e^{-n\delta}$. Some calculation then leads to $\delta = \ln 2$. Substituting this in $\sigma_n = \varepsilon e^{\tau_0+n\delta}$ –where we set ε to 1 (*i.e.*, distances are measured in original pixel lengths), and take τ_0 to be zero– we arrive at $\sigma_n = 2^n$.

We now have almost everything to compute pyramid based on scale space of, say, a 256×256 image: the base of the pyramid is the original image. The first level is created by computing the image at scale $\sigma = 2^1 = 2$, and subsampling it using 128×128 pixels. Level two is created by taking the image at scale $\sigma = 2^2 = 4$, and subsampling it at 64×64 . And so on, until the desired number of pyramid levels is reached.

There is only one snag: scaled images must be computed based on the original image, which we assumed to have scale $\sigma = 1$. We must compensate for this ‘offset’

¹⁴For simplicity, we assume that the original resolution is a power of 2.

¹⁵More accurately, the number of pixels along each spatial axis (*i.e.*, the x -dimension and y -dimension of the image) is inversely proportional to the scale.

scale when computing the pyramid levels. We can do this by using the formula $\sigma_c = \sqrt{\sigma_a^2 + \sigma_b^2}$, where σ_c is the scale of an image with original scale σ_a when convolved with a Gaussian of width σ_b . For example, pyramid level 1, with scale $\sigma = 2$, is computed by applying convolution with a Gaussian with scale $\sigma = \sqrt{3}$ to the original image, because then the final scale σ_1 will be $\sigma_1 = \sqrt{(\sqrt{3})^2 + 1^2} = 2$.

The example above shows how a pyramid can be constructed with a resolution reduction of a factor of two when rising a level in the pyramid. We chose this example because it creates a pyramid similar to the ‘ordinary’ multiresolution pyramid. For many practical applications, however, the reduction by a factor of two per level is too fast to give satisfactory segmentation results. Using the scale space approach, we now have an easy mechanism to set the resolution reduction per level to a lower value: simply set the variable δ to a lower value ($\frac{1}{2} \ln 2$ instead of $\ln 2$ is a much used heuristic value) and evaluate the pyramid as in the example above.

After creating the pyramid, the next steps of creating links and finding roots are basically the same as in the Burt-pyramid example. The only intrinsic difference in the linking process is that we need to establish a set of candidate parents a child may link to. A common procedure is to examine all parents in a circular region directly above the child, with a radius that is proportional to the scale of the parent level. A heuristic value for the proportionality constant is 2.5.

Linking criteria. In our first pyramid example, a child was linked to the parent that was closest in grey value. For a more effective segmentation, it is for many applications beneficial to use other linkage criteria in addition to the grey value difference. Examples include many of the criteria we have seen before:

- gradient magnitude; a large gradient magnitude (i.e., indication of an edge) of a parent should reduce the probability of a child linking to that parent,
- average grey value of the leaves of the sub-tree of a child; this should be close to the grey value of the parent
- spatial distance of child to a potential parent; a large distance should reduce the link probability.

Probabilistic segmentation. Up until now, we linked children to a single parent only. An interesting addition to the segmentation method is to allow children to link to several parents, assigning a *link strength* to each link in such a way that the combined strength of all links leaving a child is one; so-called *probabilistic linking*. Although this addition makes the process and implementation more complex, the amount of information gained over conventional single-linking is considerable: by forcing a child to have

only a single parent, you essentially throw away most of the available local information, reducing it to a single binary link. The cumulative loss of information considering all children and pyramid levels is substantial. This information loss is not present when probabilistic linking is applied.

When a pyramid-tree is probabilistically linked, and we have identified roots that indicate segments in the tree, a leaf of the pyramid (*i.e.*, a base level pixel) may no longer be uniquely connected to a single root. Instead, we can compute the probability of a leaf belonging to a particular root. This probability equals the sum of the edge strength of all paths leading from the root to the leaf via links. The strength of each path equals the product of the link strengths of the links in the path.

Probabilistic segmentation allows for a more realistic image segmentation in many cases. For example, suppose we have three pixels in a row: the left pixel is inside an object, the right one is outside, and the middle one straddles the object boundary. Conventional segmentation techniques will assign the middle pixel either as object or background pixel¹⁶, which is clearly wrong. Probabilistic segmentation will assign the pixel partially to the object, and partially to the background, which is much more realistic. Examples of probabilistic segmentation are shown in figure 10.26 and figure 10.27. Using a probabilistic segmentation is often little more complex than using a conventional one. For example: in non-probabilistic methods, the volume of a segment is determined by counting all pixels contained in the segment. The probabilistic volume is determined by counting each pixel for the fraction that it belongs to the segment. Another example: visualization of the surface of segments obtained by non-probabilistic segmentation often gives a ‘blocky’ result. By making pixels on the surface that have a lower segment probability partially transparent, the visualization is greatly enhanced. An example of this is shown in figure 10.28.

In our example on scale space based pyramid-tree segmentation, we used linear scale space, *i.e.*, the ‘ordinary’ scale space generated by convolving an image with Gaussians. As a final note, we mention that non-linear scale spaces, such as the ones mentioned in section 9.2 (with generating equations shown in table 10.1) can often be used effectively, especially those that preserve strong edges.

10.3.4 Texture measures

In the previous sections, we have come across several measures that are used to decide whether regions should be merged or split. In this section, we treat a useful class of measures based on the *texture* of regions. Texture measures are used for segmentation in exactly the same way as the other measures: regions having a similar measure are

¹⁶or possibly as part of some unrealistic boundary-straddling segment.

candidates for merging, and a region that has high variance of a measure is a candidate for splitting.

Defining ‘texture’ clearly in words is not an easy task. My dictionary tells me texture is *the mode of union or disposition of elementary constituent parts, as in a photograph, or surface of paper, etc.; minute structure or make; structural order*. In the context of image processing, we might think up a definition featuring keywords such as *local pattern, repetition, orientation, graininess*, but perhaps the easiest path to understanding what we mean by a texture is to simply show a number of specific ones; see figure 10.29. Most texture measures are limited in the sense that they only measure a certain aspect of texture, such as graininess or repetitive behavior. Some care is therefore needed when selecting an appropriate texture measure for a certain task; selecting the wrong measure may very well imply that the textures involved cannot be discriminated sufficiently.

Many image processing papers devoted to the quantification and measuring of texture can be found. For the most part, the approaches presented are based on local image statistics, a (Fourier) image transform, or the syntaxis of image patterns. In this section, we present the texture measures most frequently used in practice. Most of these are based on local image statistics.

We have already encountered many measures that can be used to characterize textures in different contexts. For example, the *range* of grey values in a certain neighborhood of a pixel can be used as a measure for discriminating textures. But the range is nothing but the difference of the maximum and minimum grey value in the neighborhood, *i.e.*, the residue of dilation and erosion, *i.e.*, the morphological gradient. In fact, a large number of the measures presented before can be used for discriminating certain types of texture to some degree. Some that can be used are

- range, morphological gradient,
- homogeneity, (see the region heuristics in section 10.3.1),
- histogram features, such as moments or entropy,
- edge density (f_{ww}), or number of edge pixels in a neighborhood,
- edge (gradient) orientation.

Many texture measures can be embedded in a scale space framework, either by using scale space directly in the computation (*e.g.*, by using scaled derivatives) or by applying the texture measures to scaled images. Since many textures exhibit repetitive patterns of a certain size (*i.e.*, a certain scale), there is often a ‘natural’ scale for optimal texture measuring. Note that this scale itself can be used as a measure to discriminate textures.

Although a multitude of texture measures can be defined, it is very hard to find a single measure that performs well in all practical situations. In practice, several scalar texture measures are often used simultaneously to discriminate between different textures. This

is often much more effective than using a single measure. For example, measures like average grey value of a region, variance of grey values, and size of the texture pattern can each separately be used to discriminate textures, but it is obvious that very different textures may have a similar average grey value or variance or pattern size, decreasing the discriminative power of the separate measures. If we use all three measures simultaneously, we can discriminate textures even if one or two of the measures have similar values. A simple way to combine several texture measures x_1, x_2, \dots, x_n is to combine them into a vector $(x_1, x_2, \dots, x_n)^T$, and define the difference in texture of two regions to be the distance of the respective texture measure vector heads of the regions. To make sure the separate texture measures x_i all have a similar influence on the final result, it is common to put $\frac{x_i - \bar{x}_i}{\sigma_{x_i}^2}$ in the vector instead of x_i . Here, \bar{x}_i denotes the average value of x_i , and $\sigma_{x_i}^2$ denotes the variance of x_i . This ensures no single texture measure has a disproportional influence on the texture vector distance of two regions.

Autocorrelation. The correlation of an image with translated versions of itself can give us –amongst others– information on the pattern size of a texture. This *autocorrelation* is computed by multiplication of each grey value $f(x, y)$ in a certain region with the translated grey value $f(x + k, y + l)$, and then taking the sum of these multiplications over all the pixels in the region. In practice, it is often convenient to use a slightly more complex formula, which computes a value that is normalized between 0 and 1: the *autocorrelation coefficient* $r(k, l)$ (with translation variables k and l) of a region R –which may be the whole image– is defined by

$$r(k, l) = \frac{\sum_{(x,y) \in R} f(x, y)f(x + k, y + l)}{\sum_{(x,y) \in R} f^2(x, y)}.$$

Example autocorrelation functions can be seen in figure 10.30. A repetitive pattern will cause peaks in the autocorrelation function, with the inter-peak distance equal to the pattern size. If peaks cannot be distinguished with any precision, the slope of the function at $(0, 0)$ may still give information: a relatively small slope, *i.e.*, the function decreases slowly as we move away from $(0, 0)$ indicates a large pattern size, while a large slope indicates a small pattern. The shape of the autocorrelation function, characterized by a vector of suitable scalar shape measures obtained from it, can itself be used for a texture measure. Typically, moments (especially second order) computed from the autocorrelation function are included in the vector, but other information, such as derivatives or even samples of the autocorrelation function at fixed locations, may be included as well. Shape vectors can be used for texture discrimination as described earlier in this section.

The Hurst coefficient. The idea of grey value *range* of a neighborhood as a texture measure can be extended to using neighborhoods of various sizes by using the *Hurst*

coefficient.¹⁷ For a certain pixel (x, y) , the Hurst coefficient is computed as follows: for each distance d , find the range $r_{(x,y)}(d)$, i.e., the range of grey values of all pixels at a distance d from (x, y) . Now, make a plot of $r_{(x,y)}(d)$ on double logarithmic paper¹⁸, and fit a line through the result. The slope of this line equals the Hurst coefficient. Figure 10.31 shows examples of Hurst coefficient images computed from fabric images. The practical computation of the Hurst coefficient uses only practical values for the distance d . For example, suitable values for d are 1 (neighbors to the left or right, above or below), $\sqrt{2}$ (diagonal neighbors), 2 (neighbors once removed to the left or right, above or below), $\sqrt{5}$ (neighbors a horse jump removed), etc. A maximum value for d is defined, which is effectively the neighborhood size around a pixel in which the Hurst coefficient is computed.

Co-occurrence matrices. The entries of a co-occurrence matrix $P_{\alpha,d}(g, h)$ equal how many times the grey value pair (g, h) occurs at a distance d in the orientation α in an image.

Example

Consider this image

0	1	2
2	1	0
0	0	2

Suppose we set distance to $d = 1$ and orientation to $\alpha = 0$, i.e., we consider only directly neighboring pixel pairs in the direction of the x -axis. We see that the pair $(0, 1)$ occurs twice ($0 - 1$ on the top row and $1 - 0$ on the second row). So the entry of $P_{0,1}(0, 1)$ equals two. The pair $(0, 0)$ also occurs twice, so $P_{0,1}(0, 0) = 2$. Note that we count this pattern once for *each* pixel with value 0 that has a neighbor in direction 0 at distance 1, so the occurrence of $0 - 0$ on the third row counts for two occurrences. The full co-occurrence matrix is:¹⁹

$$P_{0,1} = \begin{pmatrix} 2 & 2 & 1 \\ 2 & 0 & 2 \\ 1 & 2 & 0 \end{pmatrix}$$

The matrix constructed as in the example above is always symmetrical, because both patterns aligned with the orientation α and those 'reversely' aligned ($\alpha + 180^\circ$) to it are counted as the same pattern ($0 - 1$ is the same pattern as $1 - 0$). It is also possible to

¹⁷The Hurst coefficient is closely related to the perhaps better known *fractal dimension* D , which is related to the Hurst coefficient H by $D = C - H$, where C is a constant.

¹⁸In other words, make a plot of $\log r_{(x,y)}(\log d)$.

¹⁹where we index the matrix image-fashion for convenience instead of mathematical matrix-fashion.

construct a matrix where patterns aligned with and opposite to the orientation are considered to be different. This results in an asymmetric matrix. The asymmetric variant of the co-occurrence matrix is less common than the symmetric version in the context of texture measurement. In practice, co-occurrence matrices are often only computed for choices of d and α that are convenient for digital images. For example, α is usually chosen at 90° or 45° increments, and d is usually an integer or a multiple of $\sqrt{2}$ for diagonal orientations α . For textures where the orientation α is not critical, it is common to use co-occurrence matrices $P_d(g, h)$ that are independent of orientation, *i.e.*, they have entries listing the number of times a grey value pair (g, h) occurs at a distance d , regardless of the orientation α of the pair. Such a matrix can be computed by computing the co-occurrence matrices $P_{\alpha_i, d}$ for a number of orientations α_i , and then taking their sum: $P_d(g, h) = \sum_i P_{\alpha_i, d}(g, h)$.

Co-occurrence matrices capture the characteristics of local patterning well, especially if the texture patterns are relatively small. Several scalar measurements obtained from co-occurrence matrices have been shown to be effective for texture discrimination. These measures are usually computed from normalized versions $P_{\alpha, d}^*$ (*i.e.*, the sum of the entries is one) of the co-occurrence matrices $P_{\alpha, d}$. Some useful ones are

- energy (or uniformity): $\sum_g \sum_h (P_{\alpha, d}^*(g, h))^2$.
A uniform region will cause $P_{\alpha, d}^*$ to have relatively large entries. Regions with random grey values will show the opposite: no large entries, but many small ones. The square in this formula attaches a large weight to large entries, *i.e.*, this measure is large for uniform regions.
- contrast (or inertia): $\sum_g \sum_h (g - h)^2 P_{\alpha, d}^*(g, h)$.
This measure attaches a large weight to frequently occurring pairs of high contrast (large $|g - h|$).
- maximum probability: $\max_{(g, h)} P_{\alpha, d}^*(g, h)$.
The probability range is also sometimes used: $\max_{(g, h)} P_{\alpha, d}^*(g, h) - \min_{(g, h)} P_{\alpha, d}^*(g, h)$.
- entropy: $-\sum_g \sum_h P_{\alpha, d}^*(g, h) \log P_{\alpha, d}^*(g, h)$.

Typically, the texture of a region is characterized by computing several of the above measures of several co-occurrence matrices of the region obtained for different distances d and orientations α . The obtained measurement vector has been shown to have a good texture discriminating power for many practical applications.

Fourier techniques. Since many texture patterns exhibit distinct periodicity and orientation, Fourier techniques can offer a suitable framework to characterize such patterns. Similar to the techniques above, a texture is commonly characterized by a vector of scalar texture measurements. In this context, each measurement is usually obtained from a masked version of the Fourier magnitude image of a region. Common mask shapes are rings (for pattern size related measurements) and wedges (for orientation related measurements), such as the ones in figure 10.32. The measurement itself is usually

energy, *i.e.*, the sum of squared values in the masked Fourier image, or the maximum value occurring in the masked Fourier image.

10.4 Clustering techniques

Clustering techniques is the collective name for methods that attempt to group together measurements points ('patterns'). For an off-topic example, let's say we measure the weight and length of a population of rabbits. We can then plot the measurements of each rabbit together as shown in figure 10.33. When looking at the figure, it will be clear that three clusters can be identified –there are very likely three types of rabbit in the population. The object of clustering techniques is to identify such clusters in data. The relation to segmentation will be clear; when viewing the rabbit plot as an image, the most intuitive segmentation would be to divide the image into segments equal to the three clusters and the background.

Clustering techniques are often formulated for data of arbitrary dimension (not just two as in our rabbit example), but many clustering methods can readily be applied to two- or three-dimensional images. The images best suited for applying clustering techniques to are those similar in appearance to the rabbit plot, *i.e.*, of a very sparse nature, the dark pixels forming quasi-coherent clouds.

The rabbit data consists of a list of pairs of measurements. This data representation is the most natural for clustering techniques. If we wish to apply clustering to a binary image, we therefore use a similar representation: each pixel with value one is included in a data list of coordinate pairs. For grey-valued images there may not be a natural conversion to a list of data points. For some types of grey-valued images, it is possible to consider the grey value as the number of data points measured at a certain location.

Example

This binary image:

0	1	0
0	1	1
0	0	1

can be represented as a set of data points by listing the coordinates of the 'one' pixels:

$$\{(1, 0), (1, 1), (2, 1), (2, 2)\}.$$

The following grey valued image can similarly be represented by listing all coordinates a number of times equal to the grey value:

0	1	2
0	1	1
0	0	3

$$\{(1,0), (1,1), (2,0), (2,0), (2,1), (2,2), (2,2), (2,2)\}.$$

The heart of most clustering techniques is the *distance matrix* D . An entry $D(i,j)$ gives the distance between two data points i and j . If there are N data points, then D is an $N \times N$ symmetric matrix.

Example

Given this list of data points:

$$\{(1,1), (3,1), (2,2), (2,4), (2,5), (3,5)\},$$

then the 6×6 symmetric distance matrix D is

$$\begin{pmatrix} 0 & 2 & \sqrt{2} & \sqrt{10} & \sqrt{17} & 2\sqrt{5} \\ 2 & 0 & \sqrt{2} & \sqrt{10} & \sqrt{17} & 4 \\ \sqrt{2} & \sqrt{2} & 0 & 2 & 3 & \sqrt{10} \\ \sqrt{10} & \sqrt{10} & 2 & 0 & 1 & \sqrt{2} \\ \sqrt{17} & \sqrt{17} & 3 & 1 & 0 & 1 \\ 2\sqrt{5} & 4 & \sqrt{10} & \sqrt{2} & 1 & 0 \end{pmatrix}$$

where an entry $D(i,j)$ gives the distance between the i -th and j -th data point in the list.

In the example above, we used the Euclidean distance between data points. Other types of distance may also be employed. In a more general vein, the matrix may also be a *dissimilarity matrix*, *i.e.*, its entries give the dissimilarity between two data points, which may not equal the distance between them.

Although many different clustering techniques exist, we will discuss in this section only two basic approaches; an agglomerative one and a partitional²⁰ one. We will see that these general techniques bear a great resemblance to techniques we have seen in different contexts.

²⁰'Partitional' could be replaced 'divide' as used in reference to other segmentation methods here, but the clustering literature usually reserves 'divide' for a different type of clustering method.

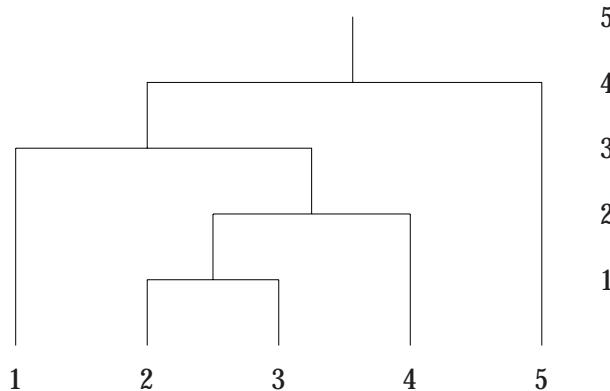
Agglomerative clustering. This approach starts out by calling each data point a separate cluster, and then proceeds to merge appropriate clusters into single clusters. The dissimilarity matrix is used to decide which clusters are to be merged; the smallest entry gives the data points that are least dissimilar and hence are the most likely candidates to be merged. After merging, the next smallest entry is selected, supplying two new merging candidates, *etc.*

Example

Given five data points with the following dissimilarity matrix:

$$D(i, j) = \begin{pmatrix} \times & 7 & 9 & 3 & 10 \\ & \times & 1 & 8 & 4 \\ & & \times & 2 & 5 \\ & & & \times & 6 \\ & & & & \times \end{pmatrix},$$

(where we have omitted the lower half because there is no additional information because of the matrix symmetry, and left out the diagonal which has only zeros). The distance between data points 2 and 3 is smallest (1), so the data points 2 and 3 will be merged first. The next smallest entries are 2, 3, and 4, so the next merging candidates are respectively the data point pairs 3 and 4; 1 and 4; and 2 and 5. The respective merges are visualized easiest in a tree structure:



The numbers at the bottom of the tree signify the five data points, the numbers to the right indicate the level of the tree. The tree shows that data points 2 and 3 are merged first (tree level 1), then their cluster is merged with data point 4 (level 2), then with 1 (level 3), and finally with 5 (level 4).

We can use the tree level to obtain the desired number of clusters in an image. For example, if we want two clusters, we select level 4, defining the two clusters: (1, 2, 3, 4) and (5). Level three defines three clusters: (1), (2, 3, 4) and (5).

The type of tree used in the above example is usually called a *dendrogram*. An algorithm to perform the above described clustering is:

Algorithm:

Given n data points and a dissimilarity matrix D :

1. Initialize by defining each data point to be a separate cluster. Set an iteration counter $c = 0$.
 2. Select the smallest entry in D : $D(a, b) = \min_{(i,j)} D(i, j)$.
 3. Merge the clusters a and b .
 4. Update D :
 - (a) Update row b and column b by setting $D(k, b) = \min\{D(k, a), D(k, b)\}$ and $D(b, k) = D(k, b)$, both for all possible k , $k \neq a, k \neq b$.
 - (b) Delete row a and column a
 5. Increase c by one, and go to step 2 if $c < n$.
-

This algorithm will merge clusters if there are member data points that are closest considering *all* data point pairs. A variation on this algorithm is to merge two clusters if the *largest* distance between its member data points is the smallest when considering all clusters. This can be achieved by a simple modification of the above algorithm: the ‘min’ in the update step 4 must be replaced by ‘max’.

Partitional clustering. The most frequently used form of partitional clustering is to divide the data points into clusters in such a way that the total distance of data points to their respective cluster centers is minimal. An algorithm to achieve this is called *K-means clustering*, and we have already seen this algorithm in a one-dimensional guise earlier in this chapter.

Algorithm: K-means clustering

This algorithm minimizes the total distance of data points to the cluster center of the cluster they are assigned to. Note that it does not require the actual computation of distances.

1. Select the number of desired clusters k . Arbitrarily (or better: intelligently) place the k cluster centers at different initial locations in the image.
2. Assign each data point to the cluster whose center is closest.
3. Recompute the cluster centers; the cluster center should be at the average coordinates (center of gravity) of the data points that make up the cluster.

-
4. Go to step 2 until no more changes occur or a maximum number of iterations is reached.
-

A drawback of this algorithm is that the number of desired clusters needs to be set beforehand. An addition to the above algorithm is therefore to embed it into an outer loop that modifies the number of clusters, or, more precisely, removes irrelevant clusters, and splits existing clusters where appropriate. After this modification, the entire new algorithm is iterated until stable. A criterion for removal of a cluster is when the number of contained data points falls below a set threshold. When this happens, the data points are probably outliers and can usually be safely ignored in the rest of the process. A criterion for splitting a cluster can be when a distance can be found of a contained data point to the cluster center that is relatively large compared to the average distance of the data point to *all* cluster centers. The splitting can then be implemented by setting this data point to be the center of a new cluster.

In a manner similar to pyramid-tree segmentation, the algorithm can be extended to be probabilistic, *i.e.*, data points can belong to several clusters simultaneously, or –more precisely– are assigned a probability distribution giving the probabilities of the data point belonging to particular clusters. This probabilistic variant is usually called *fuzzy clustering*.

10.5 Matching

If we want to locate an object in an image, and we have available an example of what it should look like (a *template*), we can find this object by matching the template to various image locations until we have found the object. For an example, see figure 10.34. The most straightforward way of determining whether a template ‘fits’ would be to place the template at a certain image location, and see whether the grey values of the template and the underlying image image grey values all match. However, because there will generally be some differences between the image and template values because of noise and other artifacts, this is not a very practical method. More useful is a quantitative measure of fit such as

$$M_1(p, q) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} (g(x, y) - f(x + p, y + q))^2,$$

where f is the image, g the $M \times N$ template, and the variables p and q determine the location of the template in the image, see figure 10.35. This measure will be small if

the template is similar to the part of the image under investigation; then all grey value differences $g(x, y) - f(x + p, y + q)$ are small and the sum M_1 will be small. The location of optimal template fit is found by minimizing M_1 to p and q .

Besides the squared form M_1 , another often used measure is M_2 , which uses the actual grey value differences instead of their squares:

$$M_2(p, q) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} |g(x, y) - f(x + p, y + q)|.$$

M_2 puts less weight on relatively large grey value differences than M_1 . But perhaps the most commonly used measure is the *cross correlation* M_3 , which is defined by

$$M_3(p, q) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} g(x, y) f(x + p, y + q).$$

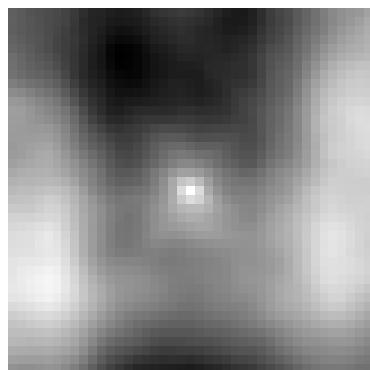
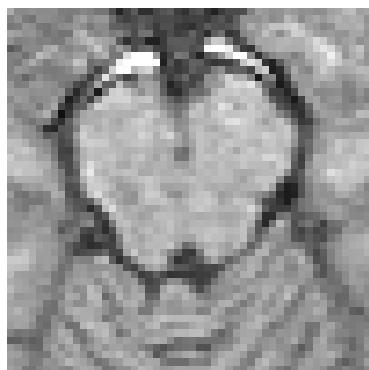
The maximum of M_3 with respect to p and q is assumed to indicate the location in the image where the template fits best. There is a strong relation between the measures M_1 and M_3 , which is most obvious if we extend the sums in the measure M_1 to the entire image f (and pad the template g with zeros); then the measure M_3 appears literally between constants if we expand the square:

$$\begin{aligned} M_1^*(p, q) &= \sum_{x=0}^{d_x-1} \sum_{y=0}^{d_y-1} (f(x, y) - g(x - p, y - q))^2 \\ &= \underbrace{\sum \sum f^2(x, y)}_{\text{constant}} - 2 \underbrace{\sum \sum f(x, y)g(x - p, y - q)}_{=M_3} + \underbrace{\sum \sum g^2(x - p, y - q)}_{\text{constant}} \end{aligned}$$

where d_x and d_y are the dimensions of the image f .

-normalized cc

-example



40x40 around optimal position,

optimal pos obviously at image center

-other measures

-other transforms

-patt recogn

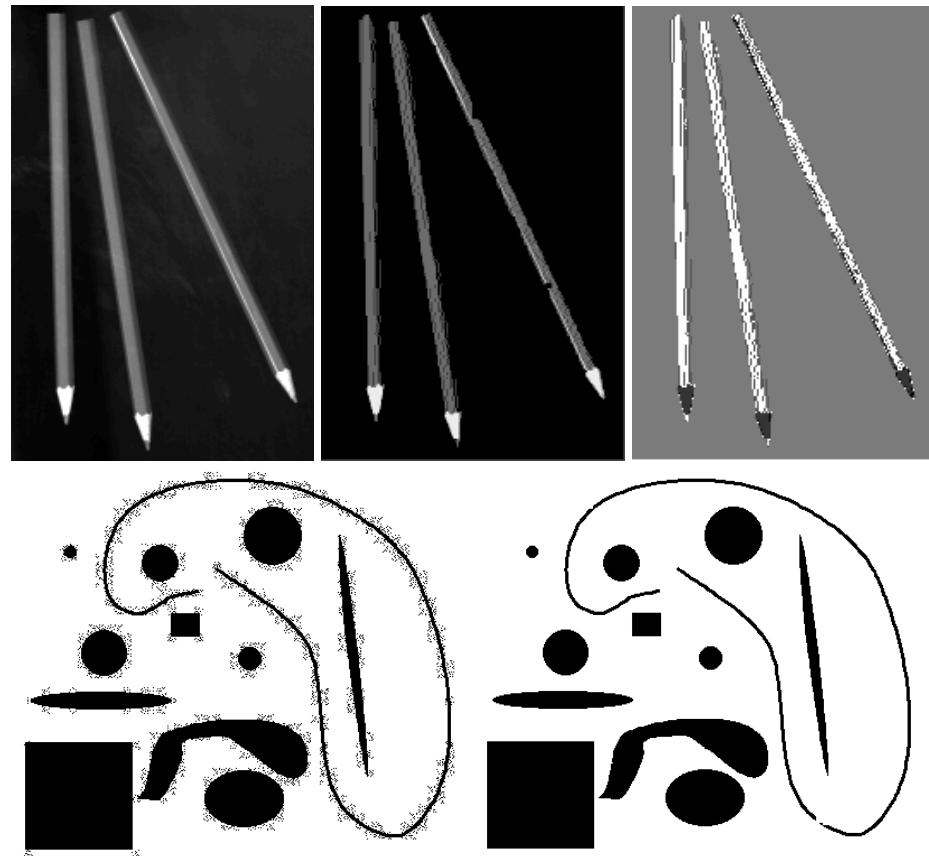


Figure 10.23 Examples of using a split and merge technique to segment images. The initial segmentation was the entire image, *i.e.*, one segment. Recursive splitting of segments into four quadrants and merging of neighboring segments was applied, both using grey value variance as a criterion. The process was iterated until stable. Top row, left: original image. Middle: after the segmentation process, with each segment assigned a grey value equal to the mean of the original grey values of the segment pixels. Right: after random remapping of the grey values to show individual segments. Bottom row, left: original binary image containing compression artifacts. Right: after segmentation, with each segment represented by the mean original grey value.

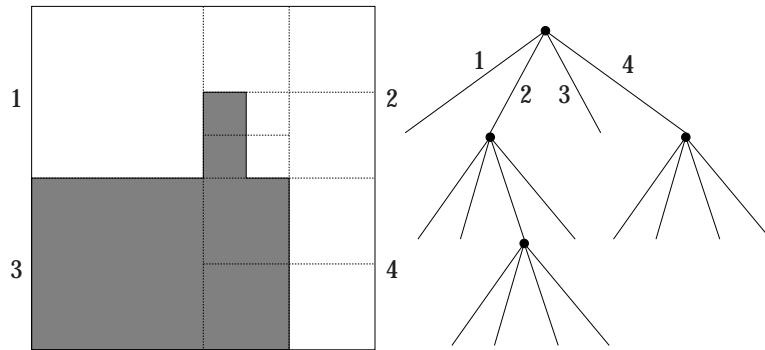


Figure 10.24 Example of modeling a recursive splitting of an image by a quadtree. The numbers show the correspondence between image quadrants and tree branches.

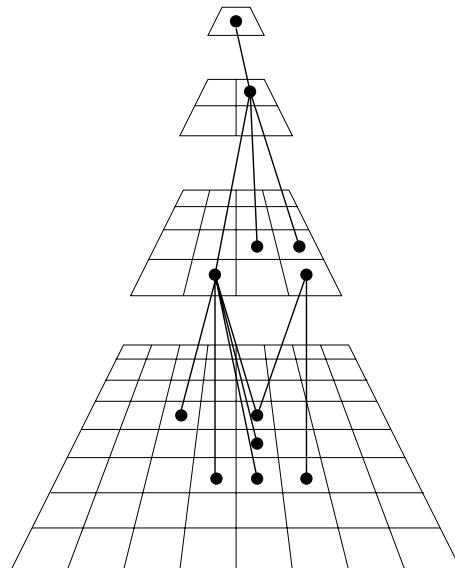


Figure 10.25 Example of a multiresolution pyramid-tree. The original 8×8 image rests at the bottom of the pyramid. Each level above is formed by reducing the resolution in each dimension by a factor of two. Each pixel represents a node in the tree, and links can be formed between nodes in adjacent levels. Some example links are drawn.

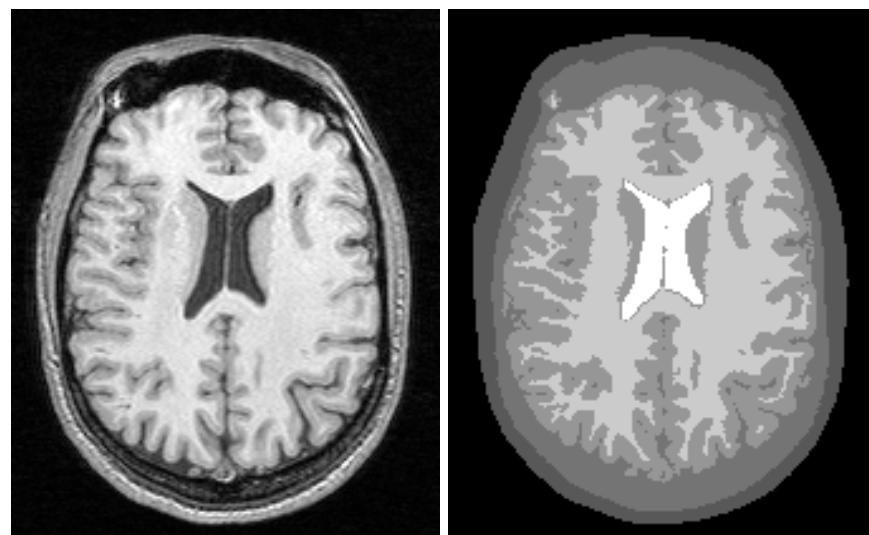


Figure 10.26 Example of probabilistic segmentation. Left: original image. Right: after probabilistic pyramid-tree segmentation (6 segments).

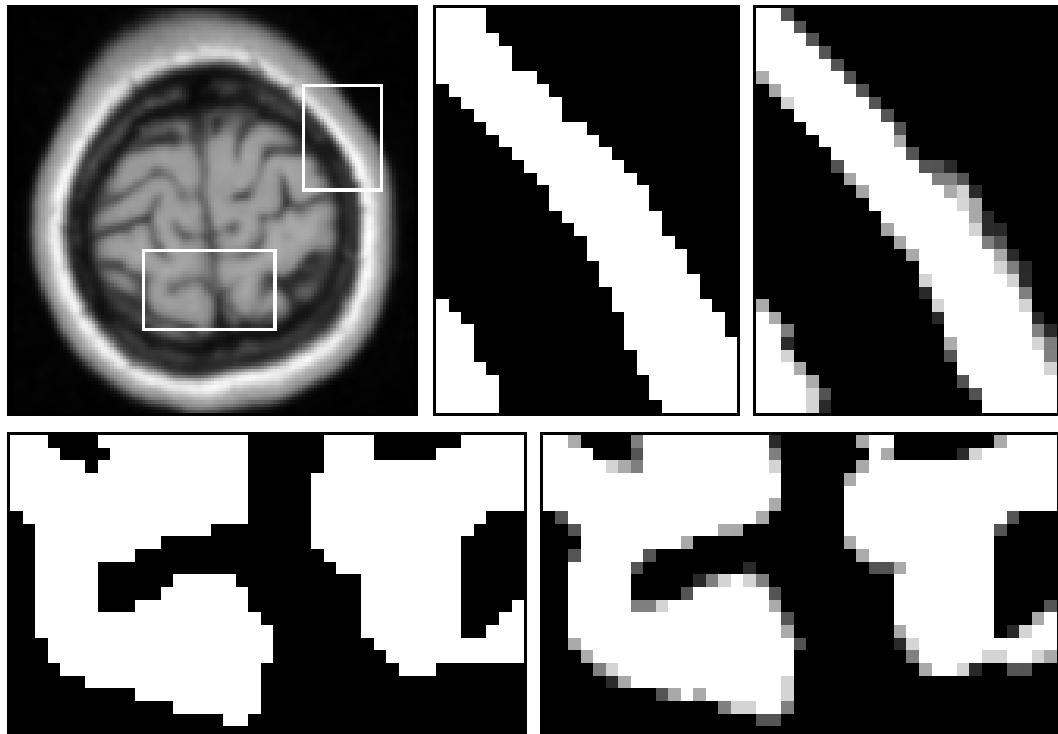


Figure 10.27 Example of the advantages of probabilistic pyramid-tree segmentation. Top left: original image. Middle: zoomed result of segmentation using single-parent linking, showing the problems when pixels are forced to belong exclusively to a single segment. Right: result of probabilistic segmentation, where the grey value of pixels indicates the probability they belong to the object segment (probability one corresponds to white, zero to black.) Bottom row: same images, zoomed in on a different image area. The zoom areas are indicated by the boxes in the original image. Note that the probabilistic segmentation is much more realistic. The conventional segmentation shows an unlikely jagged boundary.

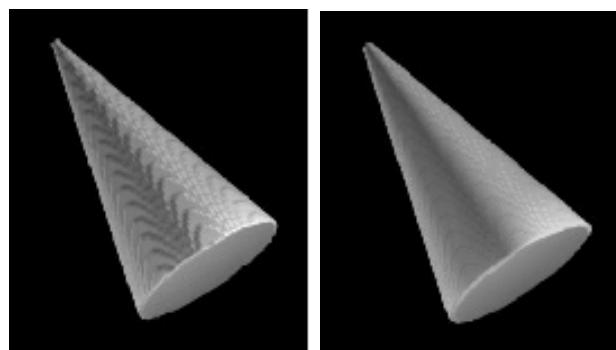


Figure 10.28 Example of the advantages of probabilistic segmentation. Left: visualization of a segment where each pixel belongs either fully to the object, or fully to the background. Right: probabilistic visualization. Pixels are made transparent proportional to the probability that they belong to the background. Note the much smoother looking surface.



Figure 10.29 Examples of various kinds of texture.

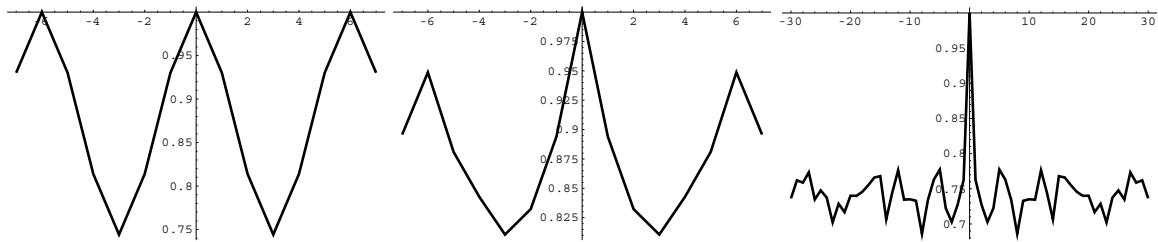


Figure 10.30 Plots of the autocorrelation coefficient, as a function of x -translation. The region used was a 42×1 rectangular window. Left: autocorrelation of the window containing the values $\{1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2\}$, i.e., containing a pattern of length 6. Note that the peaks of the correlation function occur at a distance of 6 pixels. Middle: using the same window, but with added noise: $\{1, 5, 5, 4, 4, 6, 2, 4, 6, 7, 3, 2, 2, 2, 5, 7, 5, 2, 5, 2, 5, 2, 7, 7, 6, 2, 3, 3, 7, 6, 6, 6, 5, 2, 6, 7, 5, 6, 4, 3, 5, 6, 5, 5, 2, 3, 6, 7, 6, 4\}$. Note that the peaks of the function still occur at a six pixel interval. Right: autocorrelation function of a window containing random values. The symmetry of all three plots is caused by using padding with the window itself to obtain unknown pixel values.

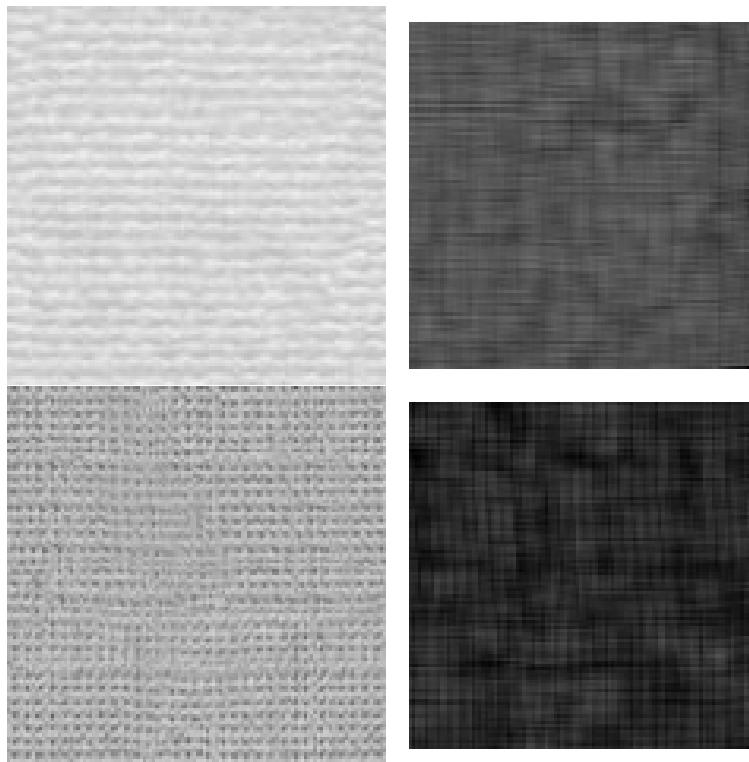


Figure 10.31 Hurst coefficients for two types of fabric. Left: original 120×120 images. Right: Hurst coefficient computed using a neighborhood of size 15. The white border stems from the fact that only neighborhoods that fit inside the image boundary were used.

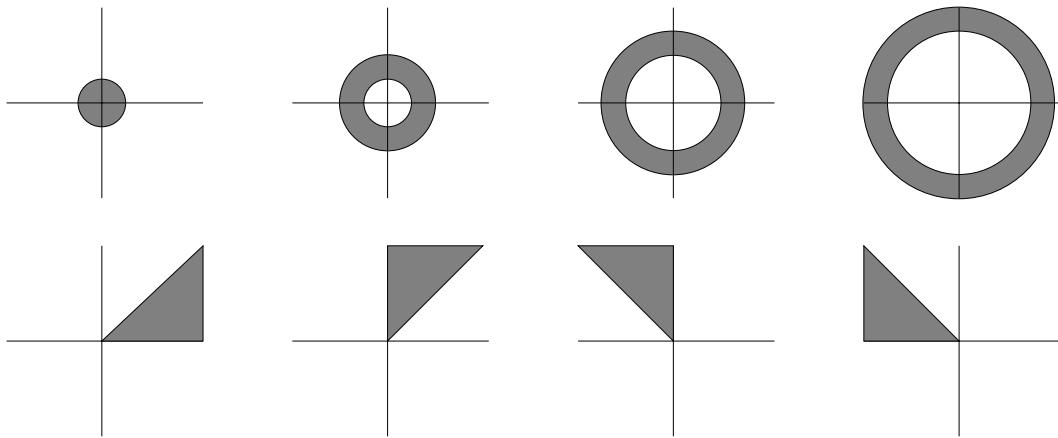


Figure 10.32 Example of masks used in Fourier texture description. Top row: ring-shaped masks, which let pass only a certain range of frequencies (band-pass). Bottom row: wedge shaped masks, which let pass only those frequencies directed along a limited range of angles. Typically, measurements obtained from all masked images are combined into a single texture-characterizing vector.

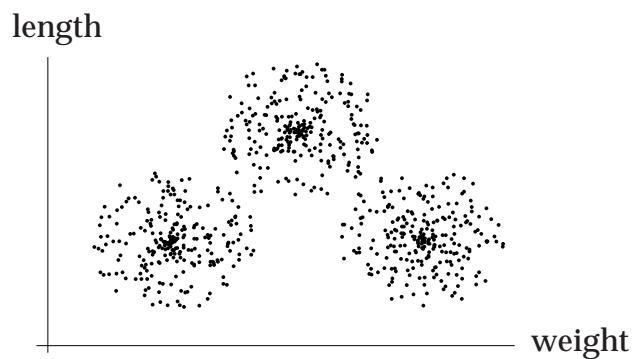


Figure 10.33 Finding clusters of measurement data points is often similar to image segmentation.

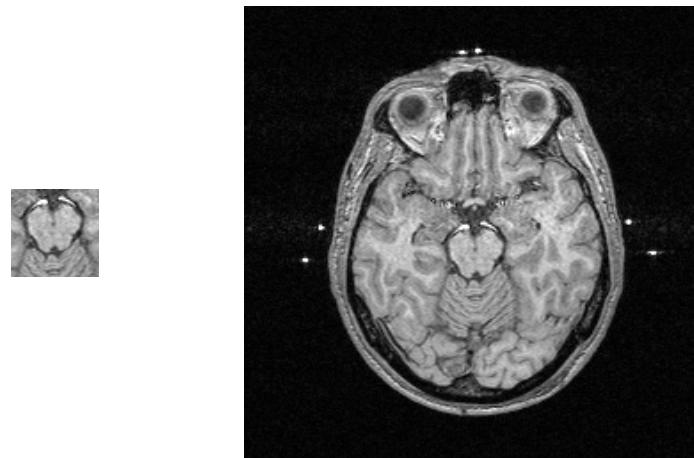


Figure 10.34 Image matching. The left image gives a template of the object we wish to locate in the right image. This is achieved by matching of the template to various locations in the right image.

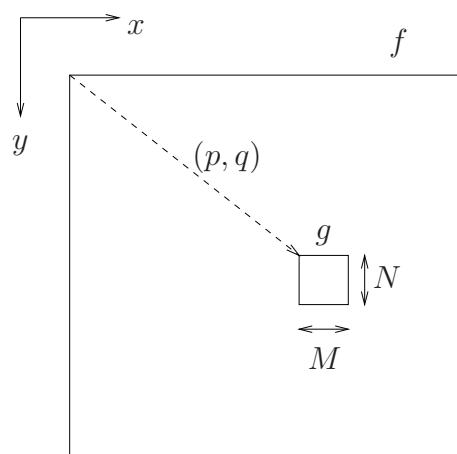


Figure 10.35 Definitions for image matching.