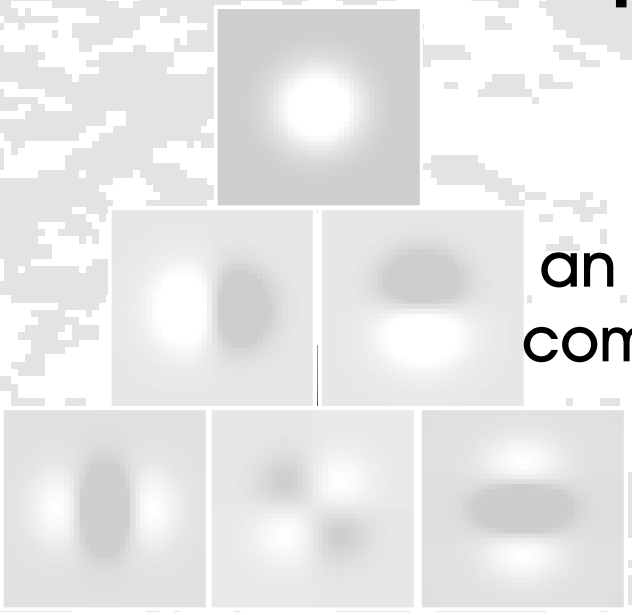
university of amsterdam
computer science department

machine vision

an introduction for computer scientists



rein van den boomgaard
leo dorst

Contents

Preface

This book accompanies the lecture series on Computer Vision at the Computer Science Department of the University of Amsterdam. This version was prepared for the 2004/2005 AI class.

Koen van de Sande provided much of the material for chapter 3.
This version of the notes was made in February 2005.

Chapter 1

Introduction

A large part of the human brain is devoted to visual perception. Recent estimates range from 30 percent to more than 50 percent of the human brain that is devoted to vision related tasks. This serves as an indication of the importance of visual perception for the human brain. It also indicates that mimicking the human visual capabilities with a computer is a daunting task.

This book is *not* about human visual perception although we owe a lot of examples and inspiration to the field of visual perception research.

This book covers the basic principles in low-level *machine vision*. We do so from a computer scientist's point of view. The computational tools used in (low-level) machine vision are the main subject in this book.

Chapter 2

Images

Acquisition, Representation and Manipulation

An image is a mathematical representation of a physical observation as a function over a spatial domain. At every point on the retina the electromagnetic energy is measured. For a computer scientist the physical laws that govern image formation and image observation are just that: *laws*. We can't change them; we have to live with them. Needless to say that this does *not* mean that we need not familiarize ourselves with these physical laws. It is senseless to develop programs that are not in accordance with physical reality. Especially in the field of machine vision the physical (and psycho-physical) description of the sensors is the starting point in utilizing these sensor signals in information systems. Therefore we would like to treat an image as defined over a *continuous* spatial domain, not as a mere collection of pixel values. The fact that we need to discretize an image by sampling should be considered a practical inconvenience that should be banned from our thinking once we have dealt with the matter in the very beginning. Only on the lowest level in an image information system should we be concerned with the fact that the individual pixels make up the discrete representation of the image. Most of our image processing work and image analysis work should regard images as mappings over a continuous domain and treat them accordingly. For a computer scientist this might seem a rather awkward and counterintuitive starting point. He or she is probably familiar with images being large arrays of say 800×600 pixels, each representing a color specified as an RGB triplet.

Fig. 2.1 illustrates the difference between what the human visual brain perceives as an image (the photograph), its mathematical model (the image function visualized as the luminance 'surface') and its discrete representation (the array of numbers). In this chapter we will develop the mathematical tools to think in terms of the mathematical model (the image *function*) when only the image *samples* (the array of numbers) are given.

2.1 Image Formation

Fig. 2.2 shows the reflection of a ray of light at the object surface. The object surface reflects the light in all directions. The ray of light from the surface patch is reflected in the direction of the human eye and projected on the retina; the inner surface of the eye that contains the light sensitive cells (see figure 2.3).

The optical principle of the human eye is the same as for any optical camera, be it a photo

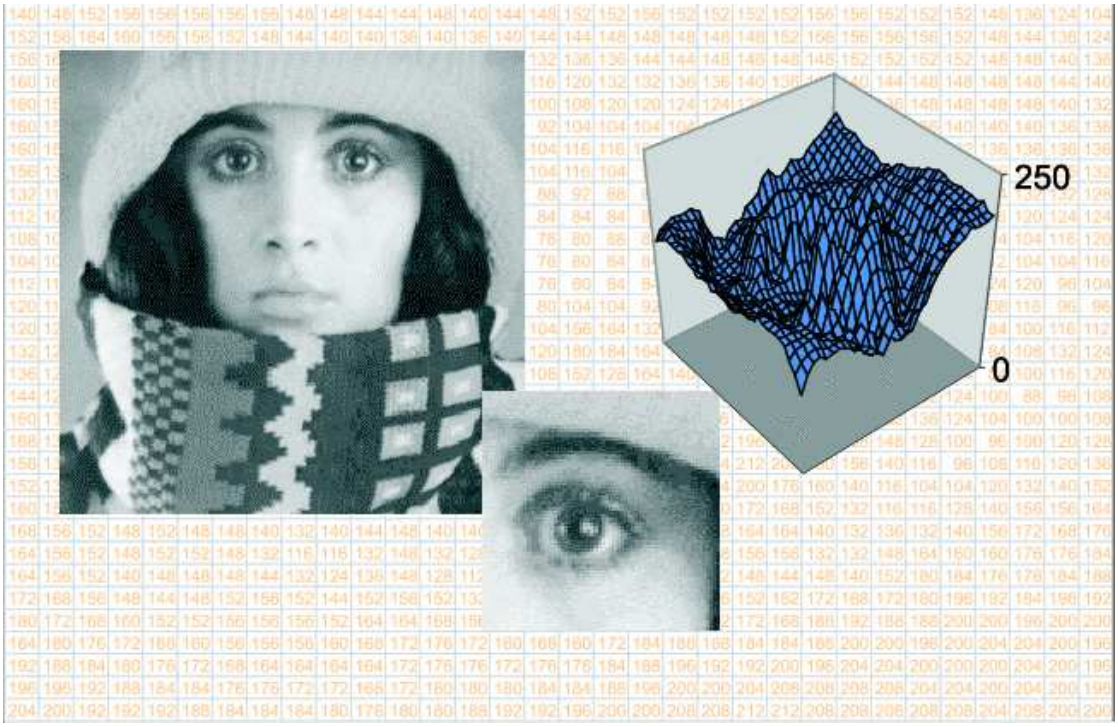


Figure 2.1: **Images: Perception, modeling and representation.** The image on the left and the small detail of the eye are 'real' images to our eye. The two dimensional function surface in the top right carries the same information as the image depicting the eye. In this case however the luminance value corresponds with the vertical height of the functions surface. The array of numbers is a detail of the larger array of numbers that is really represented in the computer.

camera or a video camera. The most simple (but surprisingly accurate) model for such an optical camera is the *pinhole camera*. This is just a box (you can build one yourself about the size of a shoe box) with a small hole (about half a millimeter in diameter; the easiest way to make one is to use aluminum foil for the side where you have to make the hole) and a photosensitive layer on the opposite side (for the homebuilt pinhole camera you can use a translucent piece of paper: "boterhampapier"). See Fig. 2.4 for a sketch of a pinhole camera.

Light reflected from an object travels in a straight line[1], through the pinhole and hits the photosensitive surface. The use of optical lenses is a physical 'trick' to enlarge the hole to get more light into the camera *without* blurring the projected image.

The projection of the 3D world onto the 2D retina of the camera is the cause of many problems in the analysis of the 3D world based on 2D images. In the projection, information about the 3D structure is lost (see Fig. 2.5 showing an image of an 'impossible' 3D object). Reconstruction of the 3D structure from *several* images of the same scene or from a video sequence is the goal in

[1]Straight line optical rays are a useful and accurate approximation of reality in the absence of inhomogeneous media and in the presence of not so heavy objects. In inhomogeneous media spatial differences in the optical characteristics may cause 'bending' of the rays. Rays of light are also bent by very heavy objects (like the sun for instance), an effect that was described by Einstein and only later experimentally verified. Diffraction reveals the wave-like nature of light on a more domestic scale. The 'geometric straight line' model of light propagation is a simplified, but useful, approximation.

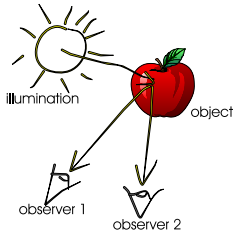


Figure 2.2: **Light reflected at surface.** At the surface of the apple, light is reflected in all directions and two of the rays hit the eye of two observers.

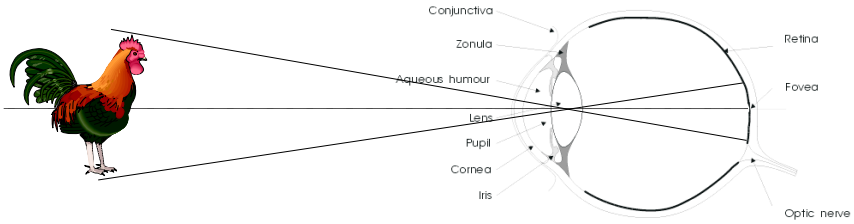


Figure 2.3: **Image formation in the eye.** The object in front of the eye is projected on the retina.

computer vision. In this book we will *not* look at this part of machine (computer) vision. Here we will only be concerned with the interpretation of the 2D image on the retina.

Where the rays of light hit the retina we are able to measure the electromagnetic energy and these measurements, as a function of the position on the retina, provide a representation of the image. If we think of the retina as a plane, all positions on the retina are given by (x, y)-coordinates. This results in a function f whose value $f(x, y)$ at position (x, y) on the retina (or image plane or image domain) is proportional to the amount of measured energy.

It is impossible to measure the energy in *all* locations on the retina. We are forced to measure it in a *finite number* of *sample locations*. The collection of all sample locations is called the *sampling grid* (see Section 2.3.1).

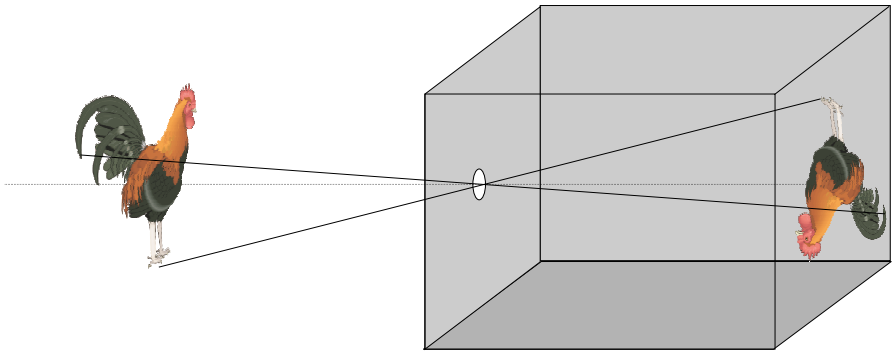


Figure 2.4: **Pinhole camera.** The most simplistic model of an optical camera is a simple box with a hole in it.

Figure 2.5: **The impossible triangle.** The triangle is impossible to make, it is not impossible to see one. . . .

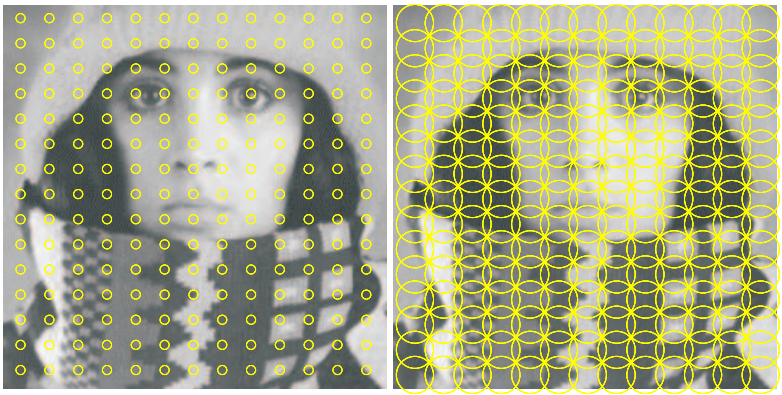


Figure 2.6: **Observing an image.** The image can only be observed in a finite number of locations with a probe of finite spatial (and temporal) size (a). The scale (or size) of the probe should be chosen in accordance with the distance between the sample points (b).

Now let us concentrate on the measurement of the electromagnetic energy in one point on the retina. It is physically impossible to construct a sensor that can observe the electromagnetic energy in a point of 'zero' size. Every sensor has to have a finite spatial (and temporal) extent (see Fig. 2.6.a). The observed (measured) luminance in a particular location on the retina is the integration of the spatial energy density in a small neighborhood (called the *measurement probe*).

The size of the measurement probe should be chosen in accordance with the distances between the individual sample locations. It is useless to have probes that have a size that is smaller then the distance between samples. In that case the measurements in neighboring sample points are completely independent and there is no clue about the stimulus *in between* the sample points. A careful choice of the ratio between the sampling distance and the scale of observation is important to be confident that the observer (the human brain or computer that processes the sampled data) doesn't miss something that happens in between the sample locations (see Fig. 2.6.b).

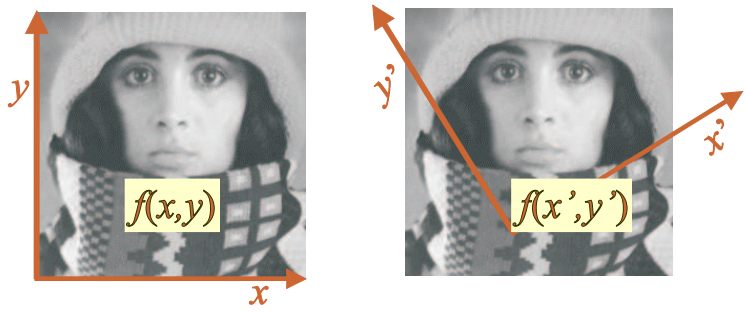


Figure 2.7: **Image domain.** The choice for a basis (and consequently a coordinate frame) is arbitrary.

2.2 Image Definition

An image f is a mapping from a spatial domain E to a range V:

$$f : E \rightarrow V \tag{2.1}$$

i.e. each element $\mathbf{x} \in E$ is mapped onto a value $f(\mathbf{x}) \in V$:

$$\mathbf{x} \in E \mapsto f(\mathbf{x}) \in V. \tag{2.2}$$

The combination of a domain E and a range V forms the *signature* of an image. The collection of all images from E to V is denoted as $\mathrm{Fun}(E, V)$.

Real world image domains without exception have a *continuous* image domain $E = \mathbb{R}^d$. The choice for a *basis* in that space (and with it a *coordinate frame*) can be made arbitrarily (see Fig. 2.7).

The image range V most often is the set of real (positive) numbers, $V = \mathbb{R}$. Each value $f(\mathbf{x}) \in \mathbb{R}$ represents a measurement in the point $\mathbf{x}$.

In this book we do not restrict the notion of images to the 2D representation of visual observations only. The measurements as function of 3D space as acquired with a confocal microscope or a medical CT scanner are images as well.

The derived observations, like the gradient vector in each image point (i.e. the vector in E pointing in the direction in which the measurement value increases maximally), or the Hessian matrix (the geometrical representation of all 2nd order image derivatives) are also called images. These types of images are discussed in a later chapter on the local structure of images.

An important class of images is formed by the *indicator images*. These images are mappings from the spatial domain E to a set of *labels* or *indicator values*. These values are not the outcome of some measurement but indicate the membership to an image region. A label image is often the outcome of an image analysis procedure that assigns to each point in the image domain the label indicating the semantic class the point belongs to.

Binary images are a specialization of a label image. The value $f(\mathbf{x})$ is a boolean value ($V = \{\texttt{TRUE}, \texttt{FALSE}\}$ or $V = \{0, 1\}$). For instance the set of all locations in the spatial domain where the gray value is larger than or equal to some level t results in a binary image g where $\forall x \in E : g(\mathbf{x}) = f(\mathbf{x}) \geq t$. Effectively we have partitioned the image domain E into two regions: the region where $g(\mathbf{x}) = 1$ and the region where $g(\mathbf{x}) = 0$. In general a label image partitions the image domain E into as many regions are there are unique labels (values) in V.

2.3 Image Discretization

To store an image function $f : E \to V$ in computer memory we need to make a discrete representation of it. Image discretization involves two separate processes: discretization of the spatial domain (image *sampling*) and discretization of the image range.

2.3.1 Spatial Sampling

Sampling an image domain leads to a discrete representation of an image as a mapping from the discrete domain $\mathbb{Z}^d$ to the range R.

The grid Λ is the set of sampling points generated as linear combinations of the grid vectors $\mathbf{b}_1, \ldots, \mathbf{b}_d$:

$$\Lambda = \left\{ k_1\mathbf{b}_1 + \cdots k_d\mathbf{b}_d \;\middle|\; k_i \in \mathbb{Z} \right\}$$

Every sample point (given the grid vectors) is uniquely determined with its integer valued indices $k_1, \ldots, k_d$. We collect these indices into a d-dimensional index vector $\mathbf{k} = (k_1 \cdots k_d)^\mathsf{T}$. Let B be the matrix whose columns are the grid basis vectors, i.e.

$$B = (\mathbf{b}_1 \cdots \mathbf{b}_d).$$

Given the index vector $\mathbf{k}$ for a point on the grid, its location in $\mathbb{R}^d$ is given by $B\mathbf{k}$. Thus the grid Λ can be written as: $\Lambda = \left\{ B\mathbf{k} \;\middle|\; \mathbf{k} \in \mathbb{Z}^d \right\}$.

Sampling an image $f : \mathbb{R}^d \to V$ using grid Λ results in a collection of values $f(B\mathbf{k})$ for all $k \in \mathbb{Z}^d$. A sampled image representation thus is a mapping $F : \mathbb{Z}^d \to V$ defined as:

$$\mathbf{k} \in \mathbb{Z}^d \mapsto F(\mathbf{k}) = f(B\mathbf{k}) \in V$$

We always assume that the spatial layout of the sampling points in $\mathbb{R}^d$ is known. Most often the orthonormal basis $B = I$ is used.

The images discussed thus far, including their discrete representation, were unbounded; their domain stretched over the infinite plane (or volume, or ...). This is of course not feasible as we can only represent a finite part of $\mathbb{Z}^d$ within the computer[2]. The *size vector*[3] $\mathbf{s} = (s_1 \cdots s_d)^\mathsf{T}$ defines the number of sample points along each of the d grid vectors. The discrete domain then is not the entire $\mathbb{Z}^d$ but the subset:

$$\mathbb{Z}_\mathbf{s} = \left\{ \mathbf{k} = (k_1 \cdots k_d)^\mathsf{T} \in \mathbb{Z}^d \;\middle|\; 1 \le k_i \le s_i \right\}.$$

It should be carefully noted that using indices in the interval from 1 to s_i is a Matlab peculiarity. Most other image processing software programs use spatial indices in the interval from 0 to $s_i - 1$.

2.3.2 The Problem with Sampling

The choice for a grid for sampling an image (or any other signal for that matter) is not a trivial one. Computationally we would like to reduce the amount of data for efficiency reasons (concerning both storage requirements as well as computational efficiency).

[2] Although it is impossible to represent unbounded images it is common practice in computer vision to develop the theory to work with images using unbounded images. This common practice will be followed in this book as well.

[3] The term 'size vector' reflects the `size` function in Matlab that returns the size of an array. Please note that Matlab mixes the representation of the range with the representation of the domain. For a color image in Matlab we thus have `size(colorimage)` returning something like `256 256 3` where the 3 indicates that the pixel values are 3 element vectors.

The theory of signal sampling (and reconstruction from the samples) is beyond the scope of this book. Here we only show one illustrative example that shows that choosing a sampling grid that is too coarse with respect to the information content of the image does not only result in loss of the small details but even can change the appearance of the image dramatically.

In Fig. 2.8 an image of stripes is shown. The image is sampled in 256×256 points. The second row shows the same image sub sampled on a grid that takes one sample in an original square of 8×8 points. Still the original stripes are clearly visible in the image. The third row subsamples the original image using one sample in an original square of 22×22 points. Perhaps surprisingly what we get looks like a striped image but the stripes are in the wrong orientation and have the wrong thickness.

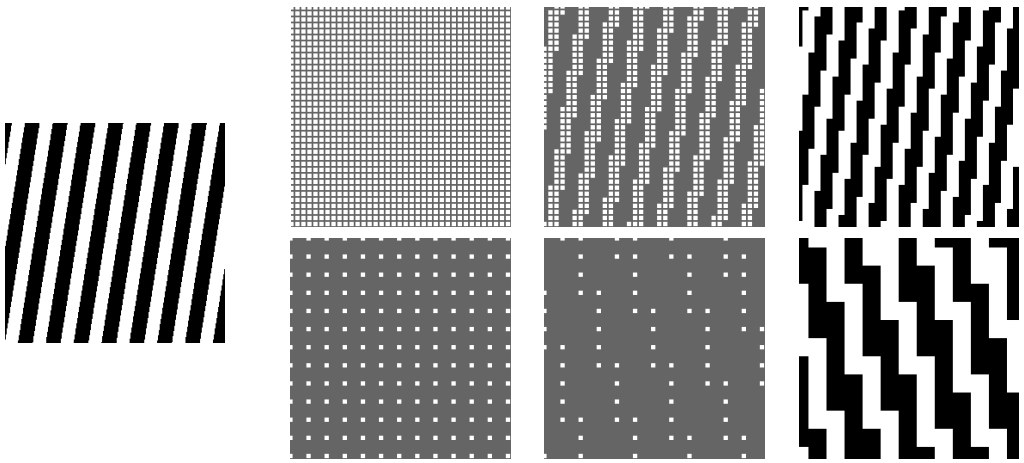


Figure 2.8: **Problems in Image Sampling.** The leftmost image shows the original 256×256 image. The first row on the right shows the sub sampling using a grid with sample distance 8 in both directions. The second row shows the sub sampling using a grid with sample distance 22 in both directions. On both the second and third row, the left image depicts the grid points (represented with 3×3 squares. The middle image shows the sampling result. On the right the samples in the sub sampled image are 'blown-up' such that the entire plane is covered.

All the sampling problems described above are perfectly understandable using standard mathematical theory. Understanding the theory requires knowledge of Fourier analysis, a mathematical tool that is presented in a later stage of the curriculum at the Computer Science department.

2.3.3 Discretizing the Range

The classical grey value image is a mapping from the spatial domain $\mathbb{R}^2$ to the set of luminance values. These luminance values are always positive (there is no such thing as a negative energy that can be observed). Luminances are also restricted to a finite range. There is an energy level that completely saturates the image sensor, such that no larger luminance levels can be observed. Traditionally the luminance range is encoded in 256 levels from 0 to 255, corresponding with an 8 bit discretization of the luminance range. Nowadays grey value luminance ranges are encoded in some sensors in up to 12 bits. Color images are most often represented using an 8 bit encoding of the red, green and blue channel each (making up a 24 bit per pixel encoding of the image).

In this book we will always assume that the image range is properly discretized, meaning that it does not only allow for a faithful representation of the observation but that it also facilitates

the necessary image computations. In practice this is dependent on the both the images (CT images have a luminance range that needs more than 10 bits for proper discretization whereas your cheap web-cam only needs about 6 bits (for each of the 3 color channels)) as well as on the type of computations needed.

In this book we will most often use a floating point representation of the image values. For a grey value image we assume that a grey value of 0 corresponds with black and that a value 1 corresponds with white. The intermediate values correspond with the gradual transition from black to white with all tones of grey.

In Matlab when we read an image from a file it is often an image with integer values in the interval from 0 to 255 (i.e. 8 bit discretization). Such an image can be easily converted to an image with real values in the range from 0 to 1 using the im2double function. See Listing 2.1 for the Matlab code to display some of the images that are in the Matlab distribution. Note the use of curly brackets to define the namelist and also to index the namelist data structure (see the Matlab documentation on cell-arrays).

Listing 2.1: Reading images from disk and converting to real values

```
namelist = {'bonemarr.tif', 'flowers.tif', 'tire.tif',
            'spine.tif', 'saturn.tif', 'rice.tif',
            'lily.tif', 'enamel.tif', 'eight.tif',
            'circuit.tif', 'cameraman.tif', 'bacteria.tif',
            'autumn.tif'};
for i=1:length(namelist)
    a = imread( namelist{i} );  % read the image from file
    a = im2double(a);           % convert the image to doubles
    imshow(a);                  % show the image on the screen
    title( namelist{i} );
    pause;                      % pause and wait until key is pressed
end
```

2.4 Image arithmetic

Consider two 2-dimensional images f and g both in $\mathrm{Fun}(\mathbb{R}^2, \mathbb{R})$ and both sampled on the grid generated by B resulting in discrete representations F and G respectively. The point wise addition of the two images $h = f + g$ is defined by:

$$\forall \mathbf{x} \in \mathbb{R}^2 : h(\mathbf{x}) = (f + g)(\mathbf{x}) = f(\mathbf{x}) + g(\mathbf{x})$$

When we sample the addition of the two images on the same grid B we obtain the discrete image H:

$$H(\mathbf{k}) = F(\mathbf{k}) + G(\mathbf{k})$$

for all $\mathbf{k} \in \mathbb{Z}^2$ (or $\mathbb{Z}_{\mathbf{s}}$ in case of an image sampled in a rectangular subset of the visual plane). The Matlab code for the addition of two images is given in Listing 2.2.

In the above example we have *lifted* a well defined operator (the scalar addition '+') to work on images by applying it pairwise to all pixels of the two images. Consider again the definition of the image addition operator $(f + g)(\mathbf{x}) = f(\mathbf{x}) + g(\mathbf{x})$. The operator $+$ in the left hand side working on images is *defined* in terms of the operator working on scalars in the right hand side.

Listing 2.2: Addition of two images

```
function h = addimages( f, g )
% Addition of two images
s1 = size( f, 1 );
s2 = size( f, 2 );
for k1 = 1:s1
    for k2 = 1:s2
        h(k1,k2) = f(k1,k2) + g(k1,k2);
    end
end
```

The addition of two images in terms of a pixel wise addition of the pixel values is *not* needed in Matlab. Matlab automatically lifts many of the standard arithmetic operators to work on images (i.e. matrices/arrays). The addition of the two images f and g could have been simply written as f+g. Not all arithmetic operators are lifted in Matlab in the way described above. For instance the multiplication f∗g of two matrices f and g is defined as the *matrix multiplication* and not as the element wise multiplication. The element wise multiplication is the Matlab operator .∗ (so, 'dot star'). Also look at the Matlab documentation of the element wise operators .^ and ./ for the element wise power function and the element wise division respectively.

More formally we can repeat this lifting procedure for any scalar function γ. Consider two images $f : E \to V$ and $g : E \to V'$. Let $\gamma : V \times V' \to V''$ be an operator that takes a value from V and a value from V' and produces a value from V''. Such a binary operator γ can be *lifted* to work on images:

$$\forall \mathbf{x} \in E : \gamma(f, g)(\mathbf{x}) = \gamma(f(\mathbf{x}), g(\mathbf{x}))$$

Note that the γ function on the right hand side working on the image values is assumed to exist and that the γ working on images is *defined* through the above equation. An image operator constructed by point wise lifting a value operator to the image domain is called a *point operator*. We will implicitly assume such a lifting construction when we talk about the addition, subtraction, multiplication etc. of images.

We will write γ when operating on image values and images alike. Effectively we are *overloading* the γ-function to work on both values and images. Many of the common binary operators on real numbers are written in an infix notation (like $+$, $-$, etc). We will follow that convention and write $f + g$ to denote the point wise addition of two images.

The lifting construction of a point operator is easily discretized. Let F and G be the discretizations of the images f and g respectively. The discretization H of the the image $h = \gamma(f, g)$ is:

$$H(k) = \gamma(F(k), G(k))$$

This shows that we can define the discrete image operator Γ working on discrete image representations $H = \Gamma(F, G)$. For lifted *point* operators we never again will make the distinction between γ and its discretized version Γ.

Point operators, although very simple, are often used in practical applications of image processing. A simple example is the weighted average of two images. Let f and g be two color images defined on the same spatial domain. A sequence of images that shows a smooth transition from f to g (if rendered as a movie) is obtained by:

$$h_\alpha = (1 - \alpha)f + \alpha g$$

Figure 2.9: **Point wise transition between two images.** From top to bottom the images $(1-\alpha)f + \alpha g$ for $\alpha = 0$, 0.25, 0.5, 0, 75 and 1 are shown.

for α values increasing from 0 to 1. Such a sequence is sketched in Fig. 2.9. The Matlab code to generate the α-blend of two images is given in Listing 2.3[4].

Listing 2.3: Alpha Blending

```
function h = alphablend( f, g, alpha )
% Alpha blend of two images
h = (1-alpha) .* f + alpha .* g;
```

A second example using alpha blending of two images is *unsharp masking*. This technique takes an image f and a blurred version of that image (we call it g). In a later chapter we will learn how to blur an image through 'Gaussian convolution'. See Fig. 2.10 for the original image and its blurred version. It is not too much a surprise that the average of the two images (an α-value of 0.5) is a somewhat less blurred version of the original image. It is perhaps more of a surprise to see that *subtracting* the blurred version from the original one (i.e. choosing a negative α-value) perceptually *sharpens* the image (see the right image in Fig. 2.10).

Not only operators $\gamma : V \times V' \to V''$ are lifted to the image domain. Also *relational operators* can be lifted. In Matlab it is easy to *threshold* an image. The comparison $f(\mathbf{x}) \geq t$ is a Boolean operator that is lifted to the image domain by Matlab automatically. In Fig. 2.11 a thresholded image is shown where the threshold has been selected such that the objects that can be seen in the image have value 1 (depicted in white) and the background has value 0 (depicted in black). The images are obtained with the Matlab code in Listing 2.5. The grey values in the original image

[4]It should be noted that in the code for the alpha blend we could have equally well use f = (1−alpha)∗f + alpha∗g; because alpha is a scalar. However the function using .∗ instead will even work in case the alpha parameter is an image itself (perhaps you can think of some situation where a locally controlled alpha blending is of some use.)

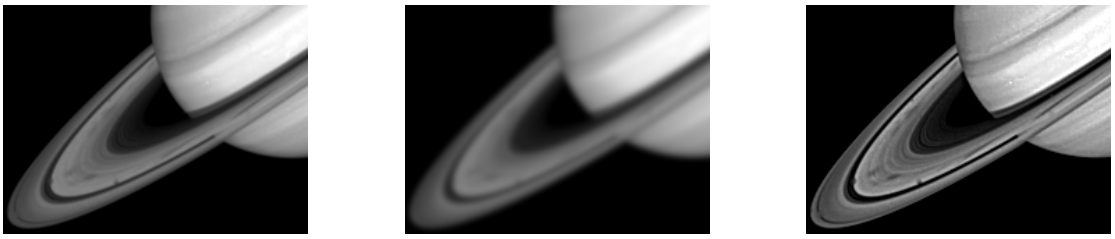


Figure 2.10: **Unsharp image masking.** By subtracting a blurred version(middle) from the original image (left) a perceptually sharper image (right) is obtained.

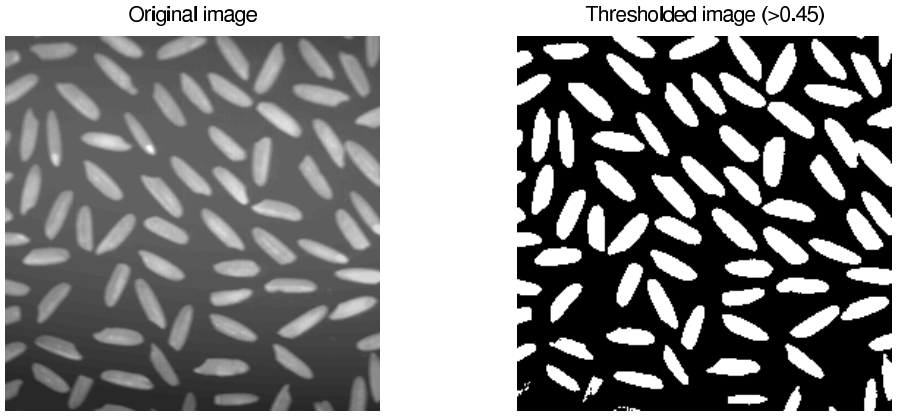


Figure 2.11: **Image Thresholding.** Comparing the image value with a *threshold* value results in a binary image indicating all locations in the image domain where $f(\mathbf{x}) > t$.

Listing 2.4: Unsharp Masking

```
% the code below assumes that the alphablend
% function is available
f = im2double( imread( 'saturn.tif' ) );
%g = gD( f, 2, 0, 0 );
g = filter2( fspecial('gaussian',17 ,3), f );
h = alphablend( f, g, -2 );
subplot(1,3,1); imshow(f);
subplot(1,3,2); imshow(g);
subplot(1,3,3); imshow(h);
```

Listing 2.5: Image Thresholding

```
a = imread( 'rice.tif' );
a = im2double( a );
subplot( 1, 2, 1 );
imshow( a ); title( 'Original image' );
subplot( 1, 2, 2 );
imshow( a>0.45 ); title( 'Thresholded image (>0.45)' );
```

are normalized (through the im2double function to lie within the $[0, 1]$ interval. The threshold is set at 0.45. Note that the rice grains are not perfectly selected due to the uneven illumination of the scene. In a subsequent chapter we will discuss a method (isodata threshold selection) to select an appropriate threshold value automatically.

2.5 Image Interpolation

A discrete image F is obtained by sampling a function $f \in \text{Fun}(\mathbb{R}^d, \mathbb{R})$ in the points on the grid generated with the matrix B:

$$F(\mathbf{k}) = f(B\mathbf{k}).$$

Intuitively it seems impossible to reconstruct a value $f(\mathbf{x})$ for a point $\mathbf{x}$ not on the sampling grid. In general this is true. For an arbitrary function f it is indeed impossible[5] to reconstruct $f(\mathbf{x})$ from the sampled version F.

In practice there is often the need to approximate $f(\mathbf{x})$ from its sampled representation F. The mathematical theory that is concerned with approximately reconstructing a continuous function from a sampled version is called *interpolation*.

We start by looking at the interpolation of sampled one dimensional functions (i.e. functions in one variable). Let $f : \mathbb{R} \to \mathbb{R}$ be a 1D function and let $F : \mathbb{Z} \to \mathbb{R}$ be its sampled version: $F(k) = f(Bk)$. In the 1D case B is a scalar which we assume to be equal to 1.

The simplest method to estimate $f(x)$ for x not being an integer (i.e. not a sample point) is to select the sample point that is closest to x:

$$f(x) \approx f_0(x) = f(\lfloor x + \tfrac{1}{2} \rfloor)$$

[5] It is beyond the scope of these lecture series to delve into the sampling theory that explains that band limited images can be sampled in such a way that from all the sample point values the continuous function can be reconstructed. This is of course the reason that we are able to have digital audio (and video) and not be able to hear (and see) the difference with the original.

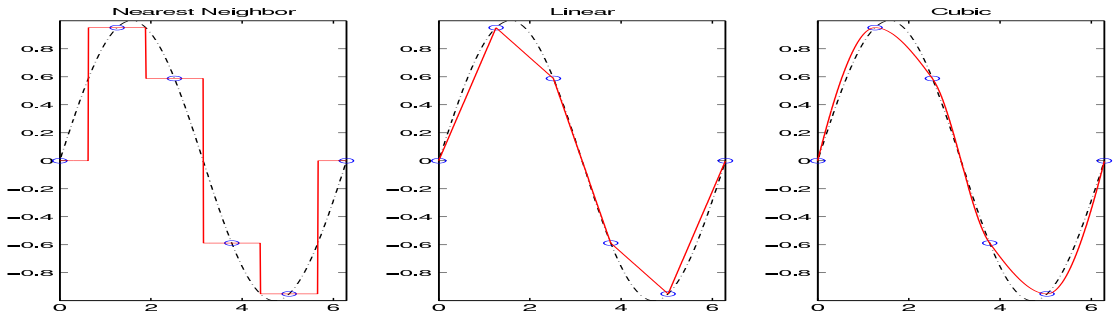


Figure 2.12: **Interpolation.** On the left a function (dashed line) with a few sample points and the nearest neighbor interpolation. In the middle the same function and samples but linearly interpolated and on the right the result of cubic interpolation.

where $\lfloor v \rfloor$ is the *floor* of value v returning the largest integer less than or equal to v. In Fig. 2.12 a 1D function f is given with its sampled version F and the *nearest neighbor interpolation* f_0. Nearest neighbor interpolation is also called *zero order interpolation* due to the fact that the function is locally approximated by a constant function (zero order polynomial).

A better interpolation technique connects two consecutive samples with a straight line. For each interval $x \in [k, k+1]$ the interpolating function is a *first order polynomial* in x:

$$f_1(x) = ax + b.$$

The parameters a and b can be calculated by 'fitting' the straight line to the samples $F(k)$ and $F(k+1)$. This leads to:

$$\begin{aligned} a &= F(k+1) - F(k) \\ b &= (1+k)F(k) - kF(k+1). \end{aligned}$$

Substituting the values of a and b in the expression for f_1 we obtain:

$$f(x) \approx f_1(x) = (1 + k - x)F(k) + (x - k)F(k+1)$$

for all $x \in [k, k+1]$ (note that $k = \lfloor x \rfloor$). See Fig. 2.12 for an example. Note carefully that the interpolating polynomials are different for every interval $[k, k+1]$.

The idea of fitting a polynomial to the data points can be generalized to N-th order polynomials. Fitting an N-th order polynomial to $N+1$ data points is known as *Lagrangian interpolation*. E.g. to fit a cubic polynomial:

$$f_3(x) = ax^3 + bx^2 + cx + d$$

we need 4 data points $F(k-1)$, $F(k)$, $F(k+1)$ and $F(k+2)$ to interpolate the data in the interval $x \in [k, k+1]$.

Instead of fitting the polynomial to the 4 points we can also fit the point to the two points $F(k)$ and $F(k+1)$ and require that the first order derivatives of neighboring interpolating polynomials are equal at the boundaries of their intervals. This leads to the *spline interpolating functions*.

Any one dimensional interpolation technique can be easily generalized to an interpolation method for d-dimensional functions. We consider a two dimensional function $f \in \mathrm{Fun}(\mathbb{R}^2, \mathbb{R})$ sampled in the points $(k, l) \in \mathbb{Z}^2$ to give the discrete representation $F \in \mathrm{Fun}(\mathbb{Z}^2, \mathbb{R})$.

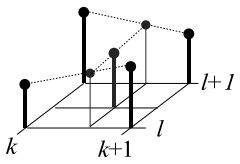


Figure 2.13: **Bilinear Interpolation.** The value at a point in between the 4 sample points is calculated by first interpolating in the k-direction twice, followed by an interpolation in the l-direction.

Assume that $x \in [k, k+1]$ and $y \in [l, l+1]$ (see Fig. 2.13). The value $f(x, l)$ can be approximated using linear interpolation from the values $F(k, l)$ and $F(k+1, l)$:

$$f(x, l) \approx f_1(x, l) = (1 - a)F(k, l) + aF(k + 1, l)$$

where $a = x - k$. Note that $a \in [0, 1]$ because $x \in [k, k + 1]$. Equivalently:

$$f(x, l + 1) \approx f_1(x, l + 1) = (1 - a)F(k, l + 1) + aF(k + 1, l + 1).$$

Now we can use a linear interpolation in the orthogonal direction to approximate $f(x, y)$:

$$f(x, y) \approx f_1(x, y) = (1 - b)f_1(x, l) + bf_1(x, l + 1)$$

where $b = y - l$. Substituting the expressions for $f_1(x, l)$ and $f_1(x, l + 1)$ into the above equation leads to:

$$\begin{aligned} f(x, y) \approx f_1(x, y) \quad = \quad & (1 - a)(1 - b)F(k, l) + \\ & (1 - a)bF(k, l + 1) + \\ & abF(k + 1, l + 1) + \\ & a(1 - b)F(k + 1, l) \end{aligned} \tag{2.3}$$

We thus obtain a 2D interpolation by interpolating in 1D along both directions, hence the name *bilinear interpolation*. Obviously cubic interpolation (or any other 1D interpolating technique) can be generalized to d-dimensional functions this way as well (you just need more points in all directions).

2.6 Geometrical Operators

You have probably seen several images where a face is deformed into a caricature shape of the original image. Probably you have also seen the deformations where one face is slowly deformed into another face. Or, more scary, maybe you have seen the old movies where a human being changes into a wolf.

All these image (video) operations are known under the name of *geometrical image operators*. A geometrical image operator T maps the domain E of an image $f : E \to V$ onto the domain E' of an image $g : E' \to V$. Let $t : E \to E'$ be the associated mapping from E to E' then we define

$$g(t(\mathbf{x})) = f(\mathbf{x}) \tag{2.4}$$

i.e. the value in $t(\mathbf{x}) \in E'$ is the value found at the original point $\mathbf{x} \in E$. A geometrical operator thus changes the position of the points but takes the value along with it.

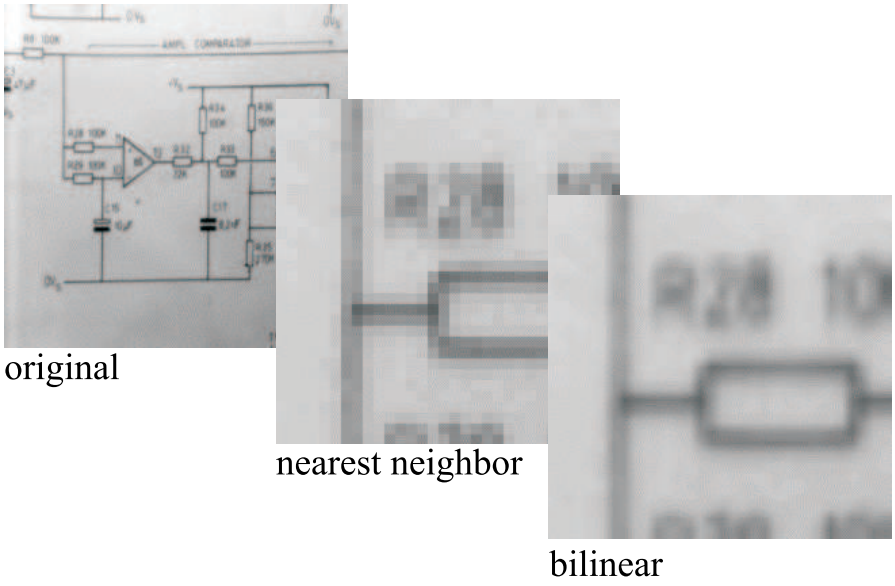


Figure 2.14: **Image scaling.** The effect of the interpolation technique is clearly visible.

In order to come up with an algorithm that takes a discrete representation of an image f and computes a discrete representation of the resultant image g, we assume the mapping t is invertible, i.e. t^{-1} is the mapping that calculates the point $\mathbf{x} \in E$ given the point $\mathbf{y} = t(\mathbf{x}) \in E'$:

$$\mathbf{y} = t(\mathbf{x}) \Longleftrightarrow \mathbf{x} = t^{-1}(\mathbf{y})$$

Let F be the sampled version of f and let G be the sampled version of g. The sampling grid for f and g are represented with the matrices B and C respectively. Let $\mathbf{l}$ be an index vector corresponding with a point on the sampling grid C, then the sample $G(\mathbf{l})$ is given by:

$$G(\mathbf{l}) = g(C\mathbf{l}) = f(t^{-1}(C\mathbf{l})).$$

The point $t^{-1}(C\mathbf{l})$ need not be a sample point on the grid generated by B. Thus we need an interpolation technique to estimate $f(t^{-1}(C\mathbf{l}))$ from its samples F. Let f_{int} denote some interpolation technique then for all sample points of the resulting image G we have:

$$G(\mathbf{l}) = f_{\text{int}}(t^{-1}(C\mathbf{l})). \tag{2.5}$$

Due to the interpolation the discrete image G is not the sampled version of g but only an approximation.

When we zoom into an image by changing the focal length of our camera we really get more details in our image (assuming the optical distortions are negligible). Digital zooming on the other hand can only use interpolation techniques to 'fill in' the missing pixels. Details not captured in the original cannot be made visible of course.

Fig. 2.14 shows an example of digitally zooming in on an image using both nearest neighbor and bilinear interpolation. It is obvious that bilinear interpolation is much better then nearest neighbor interpolation.

Eq.(2.5) is the starting point in defining the algorithm to perform a geometrical image operator. All we have to do is enumerate all indices $\mathbf{l}$ in the domain E' and calculate $G(\mathbf{l}) =$

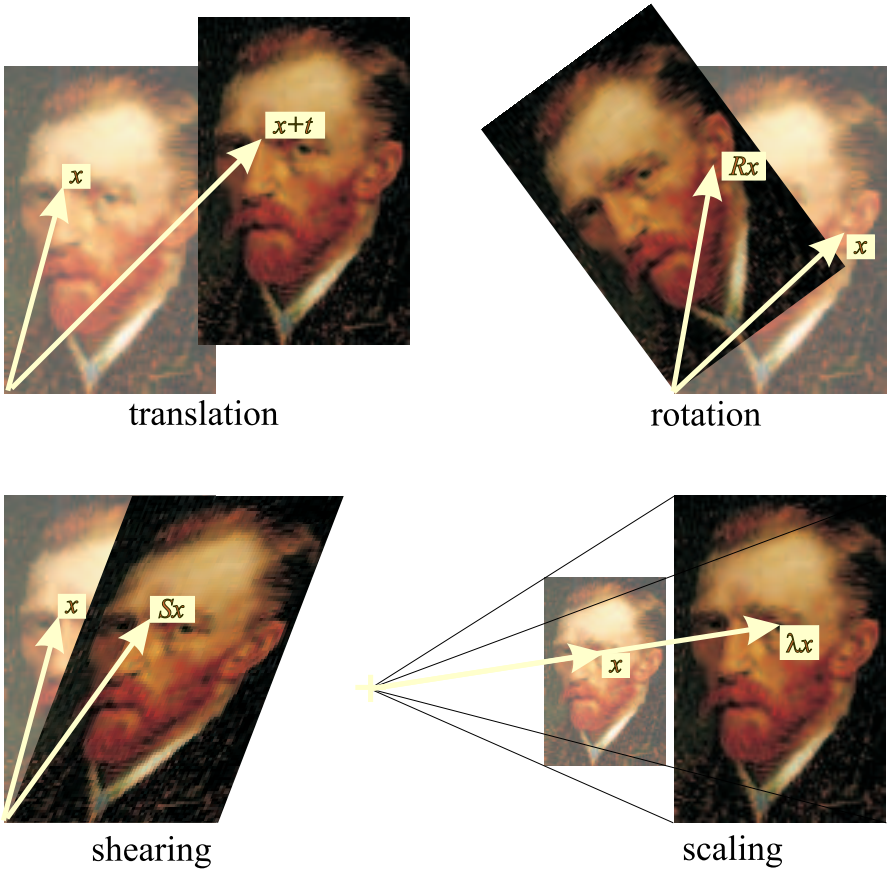


Figure 2.15: **Affine transformations.** From left to right, top to bottom canonical affine transformations are depicted: translation, rotation, shearing and scaling. In general an affine transformation is a composition of the canonical transforms.

$f_{\text{int}}(t^{-1}(C\mathbf{l}))$ using an interpolation technique of your choice. This is called the *backward* algorithm because for each point in the resultant image we calculate where that point came from in the original image[6].

Let us look at some practically important geometric transforms for $2D$-images defined over the real plane. Let $\mathbf{x}$ be a point in the visual domain. We will look at the *affine geometrical operators* of the form:

$$t(\mathbf{x}) = A\mathbf{x} + \mathbf{a}$$

where A is an invertible (non-singular) 2×2-matrix and $\mathbf{a}$ is a vector.The inverse transform is:

$$t^{-1}(\mathbf{y}) = A^{-1}\mathbf{y} - A^{-1}\mathbf{a}$$

i.e. the inverse of an affine transform is an affine transform as well. Depending on the 'shape' of A such an affine transform can express translations, rotations, scalings, shearings and combinations thereof. The canonical affine transformations are sketched in Fig. 2.15. Often, these

[6]Note very carefully that the forward algorithm in which all points on the original grid are enumerated and Eq.(2.4) is used to calculate where that point is going to, in general does not work.

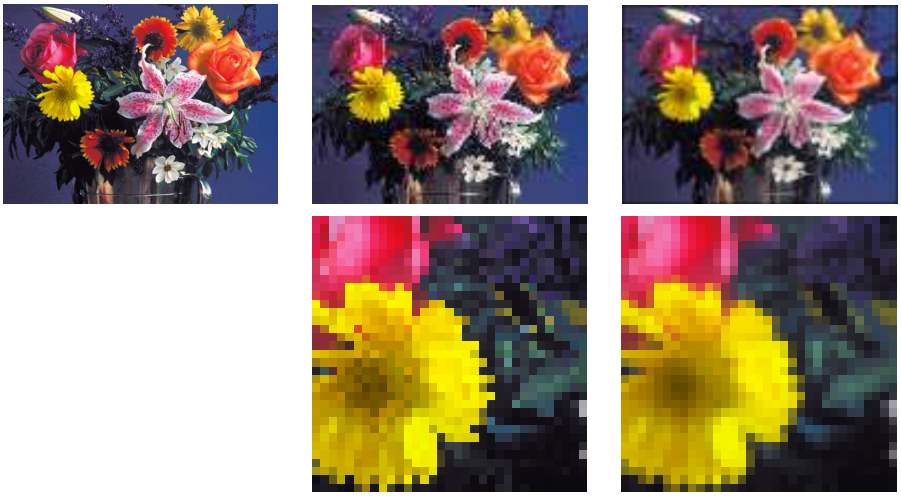


Figure 2.16: **Image Sub Sampling.** Down scaling while keeping the sampling distances equal is equivalent to *sub sampling*. From left to right the original image, the sub sampled image (without smoothing) and the sub sampled image (after smoothing) are shown.

transformations are done using matrices on homogeneous coordinates, since that also includes the useful perspective transformations.

A very frequently used geometric transform is *image scaling*. In a scaling we enlarge or reduce the image in size. The generic form of an image scaling is:

$$t(\mathbf{x}) = \lambda \mathbf{x}$$

where λ is a real scalar. For $\lambda > 1$ the image is enlarged and keeping the sampling distances (the norm of the grid vectors) equal means that we have to approximate the image values in intermediate points based on an interpolation technique.

For $\lambda < 1$ the image is reduced in size and keeping the sampling distances equal implies that we have to 'throw away' many of the known samples. We can do this using the geometrical operation algorithm (based on Eq.(2.5)) but then we are essentially sub sampling the image without enlarging the sampling probes. Better results are obtained by first smoothing the image thereby effectively enlarging the sampling probes (an example is given in Fig. 2.16). Image smoothing operators are discussed in a later chapter.

2.7 The Image Facet Model

2.7.1 A Motivation: Derivatives of Images

In many machine perception tasks estimates of the derivatives of an image (or sound signal) are needed. Consider the image in Fig. 2.17. The grey values along the indicated line are depicted in a separate plot. The borders (edges) of the flower correspond with the positions where there is a large absolute value of the first order derivative.

In Chapter 6 we take a much closer look at the *local structure* of an image expressed in terms of the image derivatives. In that chapter we will introduce the Gaussian derivative model to access the image differential structure.

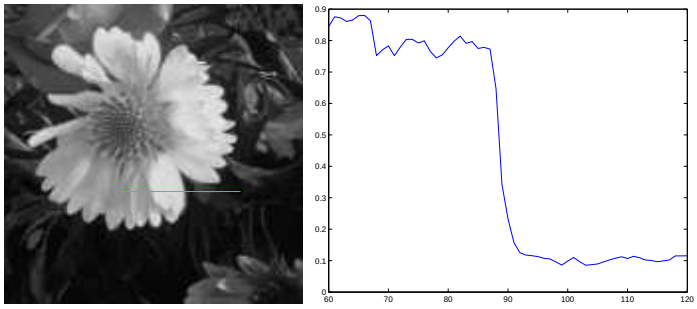


Figure 2.17: **Image derivatives are indicators of change.** On the left an image of some flowers. On the right the grey value profile along the line indicated on the left is shown. Note that the changes from one part of the flower to another are indicated by sudden changes in the grey value, i.e. large absolute derivative values.

The facet model is presented here because it is used quite a lot in image processing and because it nicely illustrates some important mathematical tools such as linear least squares approximations of sampled functions with polynomial functions. It is also useful as an alternative to bilinear interpolation in a local neighborhood of an image.

Let f be an image defined on the continuous $2D$-space. The derivative of f in the $\mathbf{v}$-direction (where $\mathbf{v}$ is a direction vector with norm 1) in a point $\mathbf{x}$ is given as:

$$(\partial_{\mathbf{v}} f)(\mathbf{x}) = \lim_{r \to 0} \frac{f(\mathbf{x} + r\mathbf{v}) - f(\mathbf{x})}{r}$$

i.e. if we move a little in the direction $\mathbf{v}$ then the derivative value is the difference in function value between the point where we started and the value in the point where we moved to, divided by the distance that we have traveled.

It would seem that we would need the capability to compute such differences in all directions; fortunately, that is not the case. If we know the derivatives of f in 2 independent directions, we know them in *all* directions, for derivatives are linear. Typically, we use the two coordinate directions to compute these standard derivatives.

If you already know this (de)composition property of derivates, you can skip some of the following proof, though you may want to remember why precisely it holds.

Consider the natural basis $(\mathbf{e}_1, \mathbf{e}_2)$ and let $\mathbf{x} = (x\, y)^\mathsf{T}$ and $\mathbf{v} = (\cos(\alpha)\ \sin(\alpha))^\mathsf{T}$ be the coordinate representations with respect to this basis (note that $\mathbf{v}$ is a normalized vector admitting the $(\cos(\alpha)\ \sin(\alpha))^\mathsf{T}$ representation). Then we have:

$$(\partial_{\mathbf{v}} f)(x, y) = \lim_{r \to 0} \frac{f(x + r\cos(\alpha), y + r\sin(\alpha)) - f(x, y)}{r}.$$

For a function f in one variable x we have $f(x + dx) \approx f(x) + dx f_x(x)$ in case $dx \to 0$ (this is just saying that locally a function is approximated with its tangent line). Applying this to our 2D function f while keeping the second argument constant we get:

$$(\partial_{\mathbf{v}} f)(x, y) = \lim_{r \to 0} \frac{1}{r} \left(f(x, y + r\sin(\alpha)) + r\cos(\alpha) f_x(x, y + r\sin(\alpha)) - f(x, y) \right).$$

Now we have two functions f and f_x that we can expand in the same way, this time in the second

argument:

$$\begin{aligned}(\partial_{\mathbf{v}} f)(x,y) &= \lim_{r\to 0}\frac{1}{r}(f(x,y) + r\sin(\alpha) f_y(x,y) + \\ &\quad r\cos(\alpha)(f_x(x,y) + r\sin(\alpha) f_{xy}(x,y)) - f(x,y)).\end{aligned}$$

Rearranging terms leads to:

$$(\partial_{\mathbf{v}} f)(x,y) = \lim_{r\to 0}\frac{1}{r}\left(r\cos(\alpha) f_x(x,y) + r\sin(\alpha) f_y(x,y) + r^2\cos(\alpha)\sin(\alpha) f_{xy}(x,y)\right).$$

or

$$(\partial_{\mathbf{v}} f)(x,y) = \lim_{r\to 0}\cos(\alpha) f_x(x,y) + \sin(\alpha) f_y(x,y) + r\cos(\alpha)\sin(\alpha) f_{xy}(x,y).$$

In the limit for $r \to 0$ the term in r just vanishes and we obtain:

$$(\partial_{\mathbf{v}} f)(x,y) = \cos(\alpha) f_x(x,y) + \sin(\alpha) f_y(x,y). \tag{2.6}$$

Note that for $\mathbf{v} = \mathbf{e}_1$ we have $\partial_{\mathbf{e}_1} f = f_x$ and also $\partial_{\mathbf{e}_2} f = f_y$.

Eq.(2.6) is an important result concerning the differential analysis of two dimensional functions: the derivative in a direction $\mathbf{v}$ can be calculated using the derivatives in x and y[7].

In practice we do not have a continuous representation of f but only its sampled version F with respect to the grid vectors $\mathbf{v}_1, \ldots, \mathbf{v}_d$. Let us look only at the two dimensional case and assume that $\mathbf{v}_i = \mathbf{e}_i$, i.e. the grid vectors are the natural base vectors.

Even in a sample point we do not have the derivative readily available. To calculate the derivative, samples at infinitesimally close points are needed. On a discrete grid these infinitesimally close points are not available and thus have to use an approximation technique. The facet model is one method of providing derivatives.

2.7.2 The Second-Order Facet Model

We now introduce a continuous function that is centered on a grid point. Consider the 2nd order polynomial:

$$g(x,y) = p_1 + p_2 x + p_3 y + p_4 x^2 + + p_5 y^2 + p_6 xy$$

This is a function in two parameters x and y determined by 6 parameters $p_1, \ldots, p_6$. Instead of fitting the polynomial to 6 data points (the minimum number of points needed) we fit the polynomial to 9 points forming the 3×3 neighborhood of a sample point. This is not an interpolation in the sense that the interpolating function passes exactly through the sample point values; the interpolating function only approximates these values.

In this approach, the values of the p_i are valid only based on the small 3×3 neighborhood for which they were designed; for the next neighborhood, other coefficients should be used. We typically use the facet model as an approximation of the image (and its derivatives) only 'within' the central pixel. Let us use i and j as the local coordinates in such a small neighborhood, and X and Y to denote the location of that neighborhood within the image. As we mentioned, the p_i are dependent on the location (X, Y), so it might be better to write them as $p_i(X, Y)$.

Let $(0, 0)$ be the indices of the sample point in which we would like to estimate the image derivatives. Each of the sample point values $F(X + i, Y + j)$ (with $i, j \in \{-1, 0, 1\}$) is then approximated by the polynomial function:

$$F(X+i, Y+j) + e(X+i, Y+j) = p_1 + p_2 i + p_3 j + p_4 i^2 + p_5 j^2 + p_6 ij$$

[7] In principle we can take the derivatives in any two independent directions as the basis to calculate the derivative in an arbitrary direction.

where the term $e(X+i, Y+j)$ indicates that the polynomial fit is only approximative (and we omitted the (X,Y)-dependence on the p_i to unclutter the formula). Let us write $F_{i,j}$ for $F(X+i, Y+j)$ to simplify our notation around this point (X,Y), and similar for $e_{i,j}$. We can then write the above in vector notation as:

$$F_{i,j} + e_{i,j} = \begin{pmatrix} 1 & i & j & i^2 & j^2 & ij \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{pmatrix}$$

For all 9 points in the 3×3 neighborhood of the central point $(0,0)$ we arrive at:

$$\begin{pmatrix} F_{0,0} \\ F_{1,0} \\ F_{1,1} \\ F_{0,1} \\ F_{-1,1} \\ F_{-1,0} \\ F_{-1,-1} \\ F_{0,-1} \\ F_{1,-1} \end{pmatrix} + \begin{pmatrix} e_{0,0} \\ e_{1,0} \\ e_{1,1} \\ e_{0,1} \\ e_{-1,1} \\ e_{-1,0} \\ e_{-1,-1} \\ e_{0,-1} \\ e_{1,-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & -1 & 1 & 1 & 1 & -1 \\ 1 & -1 & 0 & 1 & 0 & 0 \\ 1 & -1 & -1 & 1 & 1 & 1 \\ 1 & 0 & -1 & 0 & 1 & 0 \\ 1 & 1 & -1 & 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{pmatrix}$$

Or in matrix-vector notation:

$$\mathbf{f} + \mathbf{e} = A\mathbf{p}$$

i.e.:

$$\mathbf{e} = \mathbf{f} - A\mathbf{p}.$$

The optimal fitting 2nd order polynomial is obtained by choosing the parameter vector $\mathbf{p}^\star$ that minimizes the norm of the error vector $\mathbf{e}$:

$$\mathbf{p}^\star = \arg\min_{\mathbf{p}} \|\mathbf{e}\| = \arg\min_{\mathbf{p}} \mathbf{e}^\mathsf{T}\mathbf{e}$$

There are two ways to solve this. If you have had vector calculus, you can differentiate with respect to vectors, and the solution would proceed as follows:

- *vector calculus solution*
  Substituting the expression for $\mathbf{e}$ we obtain:

  $$\mathbf{p}^\star = \arg\min_{\mathbf{p}} \left(\mathbf{f}^\mathsf{T}\mathbf{f} - 2\mathbf{p}^\mathsf{T}A^\mathsf{T}\mathbf{f} + \mathbf{p}^\mathsf{T}A^\mathsf{T}A\mathbf{p}\right)$$

  It is not hard to prove that minimizing the above matrix equation boils down to differentiating with respect to the vector $\mathbf{p}$, and setting the result equal to zero:

  $$A^\mathsf{T}A\mathbf{p}^\star = A^\mathsf{T}\mathbf{f}$$

  Solving for $\mathbf{p}^\star$:

  $$\mathbf{p}^\star = \left(A^\mathsf{T}A\right)^{-1} A^\mathsf{T}\mathbf{f}$$

  The matrix $\left(A^\mathsf{T}A\right)^{-1} A^\mathsf{T}$ is called the generalized inverse (or Moore-Penrose inverse) $A^\dagger = (A^\mathsf{T}A)^{-1}A^\mathsf{T}$.

- *projection solution*
  The only values $A\mathbf{p}$ can take are in the subspace $\mathrm{im}(A)$, i.e. the 'image' of the matrix A. The shortest distance between such a vector and $\mathbf{f}$ is achieved by the value for $\mathbf{p}^\star$ such that $\mathbf{f} - A\mathbf{p}^\star$ is perpendicular to $\mathrm{im}(A)$. It should therefore be perpendicular to all elements that span $\mathrm{im}(A)$, i.e. to all columns of A. That implies that $A^\mathsf{T}(\mathbf{f} - A\mathbf{p}^\star) = 0$. The matrix $A^\mathsf{T}A$ is invertible and that produces the same solution as before.

In either case, you get the following linear least squares solution:

$$\mathbf{p}^\star = A^\dagger \mathbf{f}$$

Note that $A^\dagger$ can be calculated without knowing the values of the image data samples. It depends only on the form of the matrix A, which in turn is fully determined by the fact that we use a 3×3, second order facet model:

$$A^\dagger = \frac{1}{36}\begin{pmatrix} 20 & 8 & -4 & 8 & -4 & 8 & -4 & 8 & -4 \\ 0 & 6 & 6 & 0 & -6 & -6 & -6 & 0 & 6 \\ 0 & 0 & 6 & 6 & 6 & 0 & -6 & -6 & -6 \\ -12 & 6 & 6 & -12 & 6 & 6 & 6 & -12 & 6 \\ -12 & -12 & 6 & 6 & 6 & -12 & 6 & 6 & 6 \\ 0 & 0 & 9 & 0 & -9 & 0 & 9 & 0 & -9 \end{pmatrix}$$

We used Matlab to compute this, in the following procedure. The matrix A is constructed in Matlab row by row, which effectively means that for each point (i, j) in the neighborhood we construct the row $(1\ i\ j\ i^2\ j^2\ ij)$: First we construct vectors containing the coordinates within a 3×3 neighborhood.

```
icoords = [0 1 1 0 -1 -1 -1 0 1];
jcoords = [0 0 1 1 1 0 -1 -1 -1];
```

Then we define the function that constructs the A-matrix:

```
function A = matrixA( icoords, jcoords)
A = [];
for k = 1:length(icoords)
  A = [A; rowA(icoords(k),jcoords(k))];
end

function row = rowA( i, j )
row = [ 1 i j i^2 j^2 i*j ];
```

Then the solution to the optimization problem computed above is implemented as:

```
A = matrixA( icoords, jcoords);
Adagger = inv( A' * A ) * A';
```

This results in the matrix $A^\dagger$ above.

According to $A^\dagger$, the optimal parameter p_1 at the location (X, Y) is computed as:

$$\begin{aligned} p_1 \quad = \quad & \frac{1}{9}(5F_{0,0} + 2F_{1,0} - F_{1,1} + \\ & 2F_{0,1} - F_{-1,1} + 2F_{-1,0} - \\ & -F_{-1,-1} + 2F_{0,-1} - F_{1,-1}), \end{aligned}$$

$$\frac{1}{9}\begin{Bmatrix} -1 & 2 & -1 \\ 2 & \underline{5} & 2 \\ -1 & 2 & -1 \end{Bmatrix}$$

$$\frac{1}{6}\begin{Bmatrix} -1 & 0 & 1 \\ -1 & \underline{0} & 1 \\ -1 & 0 & 1 \end{Bmatrix} \quad \frac{1}{6}\begin{Bmatrix} 1 & 1 & 1 \\ 0 & \underline{0} & 0 \\ -1 & -1 & -1 \end{Bmatrix}$$

$$\frac{1}{6}\begin{Bmatrix} 1 & -2 & 1 \\ 1 & \underline{-2} & 1 \\ 1 & -2 & 1 \end{Bmatrix} \quad \frac{1}{6}\begin{Bmatrix} 1 & 1 & 1 \\ -2 & \underline{-2} & -2 \\ 1 & 1 & 1 \end{Bmatrix} \quad \frac{1}{4}\begin{Bmatrix} -1 & 0 & 1 \\ 0 & \underline{0} & 0 \\ 1 & 0 & -1 \end{Bmatrix}$$

Figure 2.18: **Facet Model.** The weights for the sample values are shown for the 6 parameters (p_1 to p_6 from left to right, top to bottom). The parameters are proportional to the estimates of the partial image derivatives (including the 'zero order derivative') f, f_x, f_y, f_{xx}, f_{yy} and f_{xy} in the central point.

and similarly for the others. Remember that the p_i depend on (X, Y), because the image data is taken addressed around the point (X, Y) as $F_{i,j} \equiv F(X + i, Y + j)$.

A convenient way of denoting this operation is to give the pattern of the coefficients of the p_i, omitting the $F_{i,j}$'s:

$$p_1 = \frac{1}{9}\begin{Bmatrix} -1 & 2 & -1 \\ 2 & \underline{5} & 2 \\ -1 & 2 & -1 \end{Bmatrix} \qquad p_2 = \frac{1}{6}\begin{Bmatrix} -1 & 0 & 1 \\ -1 & \underline{0} & 1 \\ -1 & 0 & 1 \end{Bmatrix} \qquad \text{etc.}$$

where the weight at position $(0, 0)$ is underlined to show where the $(0, 0)$ position is. Please note that we use ( ) to denote matrices and vectors and { } to denote fragments of sampled data. We can arrange the weights corresponding with the spatial sampling for all 9 parameters. In Fig. 2.18 these spatial arrangements are shown from top to bottom, left to right.

For the Adagger matrix calculated in Matlab the ordering is again such that one row corresponds with the kernel weights for one of the parameters. The spatial coordinates are stored in the icoords and jcoords vectors. We can use these to construct the 'spatial layout':

```
function p = spatialFacet( Adagger, icoords, jcoords, i, N )
p = zeros(N,N);
offset = floor( (N-1)/2 );
icoords = icoords + offset + 1;
jcoords = jcoords + offset + 1;
for k=1:length(icoords)
  p( icoords(k), jcoords(k) ) = Adagger(i, k);
end
```

All the spatial correlation kernels are then obtained with the following code:

```
icoords = [0 1 1 0 -1 -1 -1 0 1];
jcoords = [0 0 1 1 1 0 -1 -1 -1];

A = matrixA( icoords, jcoords )
Adagger = inv( A'*A) * A'

p1 = spatialFacet( Adagger, icoords, jcoords, 1, 3 )
p2 = spatialFacet( Adagger, icoords, jcoords, 2, 3 )
p3 = spatialFacet( Adagger, icoords, jcoords, 3, 3 )
p4 = spatialFacet( Adagger, icoords, jcoords, 4, 3 )
```

```
p5 = spatialFacet( Adagger, icoords, jcoords, 5, 3 )
p6 = spatialFacet( Adagger, icoords, jcoords, 6, 3 )
```

(There is only one thing different from the spatial layout as shown in Fig. 2.18 where the first coordinate is along the horizontal direction, whereas in Matlab it is along the vertical. This is unfortunately a fact of life when using Matlab for image processing.)

You can use this same principle to compute the coefficients for higher-order facet models, in neighborhoods of different sizes.

2.7.3 Use of the Facet Model for Derivatives

With the help of the fitted polynomial for a facet at the location (X, Y) depending on the $p_i(X, Y)$, it is easy to calculate the following approximations for small deviations (x, y) from this central location:

$$f(X+x, Y+y) \approx p_1 + p_2 x + p_3 y + p_4 x^2 + + p_5 y^2 + p_6 xy \quad (2.7)$$
$$f_x(X+x, Y+y) \approx p_2 + 2p_4 x + p_6 y \quad (2.8)$$
$$f_y(X+x, Y+y) \approx p_3 + 2p_5 y + p_6 x \quad (2.9)$$
$$f_{xx}(X+x, Y+y) \approx 2p_4 \quad (2.10)$$
$$f_{yy}(X+x, Y+y) \approx 2p_5 \quad (2.11)$$
$$f_{xy}(X+x, Y+y) \approx p_6 \quad (2.12)$$

For the derivations, we are most often interested in the approximations in the central point of the 3×3 neighborhood where we have calculated the fitting polynomial, i.e. the point $(x, y) = (0, 0)$. We leave out the argument and obtain:

$$f \approx p_1 \quad (2.13)$$
$$f_x \approx p_2 \quad (2.14)$$
$$f_y \approx p_3 \quad (2.15)$$
$$f_{xx} \approx 2p_4 \quad (2.16)$$
$$f_{yy} \approx 2p_5 \quad (2.17)$$
$$f_{xy} \approx p_6 \quad (2.18)$$

Therefore the coefficients of the facet model can be directly interpreted as estimates of the derivatives at the center of the neighborhood.

As an example, consider the following small part of an image. The grey values increase from 0 to 6 when going from left to right.

·	·	·	·	·	·	·	·	·
·	0	0	1	3	5	6	6	·
·	0	**0**	**1**	**3**	**5**	**6**	6	·
·	0	0	1	3	5	6	6	·
·	·	·	·	·	·	·	·	·

The p_i parameters have to be calculated for all points on the grid. Let us look at the left most position with a box around it. At that point the local 3×3 neighborhood looks like:

0	0	1
0	0	1
0	0	1

To calculate p_1 at this point, the values have to be element wise multiplied with the weights in the p_1 mask, and then summed. The multiplication results in:

$$\begin{matrix} 0 & 0 & -1 \\ 0 & 0 & 2 \\ 0 & 0 & -1 \end{matrix}$$

and the sum of this, giving the optimal p_1 value for this neighborhood, gives $p_1 = 0$. The estimate of the first order derivative at the same point is 0.5.

2.7.4 Correlation

In principle we can calculate the p_i value for all points in an image and store the results in a new image for each p_i, giving 6 'facet coefficient images'. The image operator that takes a weighting kernel (mask) and performs the element wise multiplication of the values in the mask and the values in the image neighborhood and then sums all these values to give the new value for that point, is called the *correlation operator*. Let W be the 3×3 weight mask then the image G resulting from the correlation of image F with kernel W is defined as:

$$G(i,j) = \sum_{k=-1}^{1} \sum_{l=-1}^{1} F(i+k, j+l)\, W(k,l).$$

You can find the correlation between an image and a weighting kernel in Matlab as the function imfilter (). When using the facet model to compute derivatives, the resulting p_i images have a direct meaning, given above.

Figure 2.19: **Solving the Border Problem in Local Image Operators.** From left to right: original image, image with constant border, image with 'nearest border' and image with periodic border.

A careful reader might worry about what happens in case the point (i, j) is on the 'border' of the bounded image domain. Then the point $(i+k, j+l)$ need not be within the bounded image domain. The border problem is indeed a nuisance in vision algorithms. Common choices to deal with the border problem are:

- in case $(i+k, j+l)$ is outside the image domain we take $F(i+k, j+l)$ to be a predefined constant value (zero for instance).
- in case $(i+k, j+l)$ is outside the image domain we take $F(i+k, j+l)$ equal to the value of $F(m, n)$ where (m, n) is the point in the domain that is closest to the point $(i+k, j+l)$ not in the domain.

- in case $(i+k, j+l)$ is outside the image domain we take $F(i+k, j+l)$ equal to the value of $F(m, n)$ where $m = (i+k)\%s_1$ and $n = (j+l)\%s_2$. Here % is the modulo operator. This effectively means that the bounded image is periodically repeated in the infinite space.

These three possibilities to deal with the border problem are illustrated in Fig. 2.19.

The above equation for the correlation of an image F with a kernel W assumes that the kernel is restricted to a 3×3 neighborhood. We can easily abandon this restriction using a kernel W that is defined on the entire (infinite) space and setting $W(k, l) = 0$ for the points outside the original 'effective domain' of the kernel. Note that points $(i+k, j+l)$ where $W(k, l) = 0$ do not contribute at all to the correlation value. This leads to the correlation equation:

$$G(i, j) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} F(i+k, j+l)\, W(k, l).$$

We can generalize the above equation one step further and make it independent on the dimension of the image using index vectors from the domain $\mathbb{Z}^d$ (where d is the dimensionality of the image). This leads to the following definition of the correlation of an image F with kernel W:

$$G(\mathbf{k}) = \sum_{\mathbf{l} \in \mathbb{Z}^d} F(\mathbf{k}+\mathbf{l})\, W(\mathbf{l}). \tag{2.19}$$

We have already seen that the kernel determines what the meaning is of the correlation operator. We have developed several kernels to calculate image derivatives. This correlation operator will turn out to be an important one throughout this book.

2.7.5 The Use of the Facet Model for Interpolation

Now that we have a way to compute reasonable approximations to image values at fractional coordinates, we can of course also use the facet model for interpolation. In that case, you would like to know the value at, say, the real location (X', Y'). You should first compute the neighborhood of which you want to use the facet (this is typically the properly rounded integer part of X' and Y'), and where you are in that local neighborhood (that gives x and y). Use the proper coefficients $p_i(X, Y)$ for that neighborhood, and substitute the values in the equation of the second order polynomial, to give you the interpolated image value. In all its glory, this is:

$$F_{\text{int}}(X+x, Y+y) = p_1(X,Y) + p_2(X,Y)x + p_3(X,Y)y + p_4(X,Y)x^2 + p_5(X,Y)y^2 + p_6(X,Y)xy.$$

If you only need to interpolate for a few values, you could compute the p_i coefficients as and when you need them, by a local correlation of the image with the corresponding filter scheme. If you need almost all values (for instance because you are rotating an image), or many of them repeatedly, then it makes sense to compute the whole 'images' of the p_i coefficients, as explained in the derivative application.

2.8 Exercises

1. **Image Signatures**

   What is the signature of:

   (a) a binary 2D image (*answer:* $f : \mathbb{R}^2 \to \{0, 1\}$)

   (b) a time sequence of 2D images? (*answer:* $f : \mathbb{R}^+ \times \mathbb{R}^2 \to \mathbb{R}$)

(c) a time sequence of color images?

(d) the gradient of a scalar 3D image.

2. **Sampling Grids**

(a) Sketch the points on the grid generated by the vectors $\mathbf{e}_1 = (1\,0)^\mathsf{T}$ and $\mathbf{e}_2 = (0\,1)^\mathsf{T}$.

(b) Also for the grid generated by $\mathbf{e}_1 = (1\ -1)^\mathsf{T}$ and $\mathbf{e}_2 = (1\,1)^\mathsf{T}$.

(c) Also for the grid generated by $\mathbf{e}_1 = (1\,0)^\mathsf{T}$ and $\mathbf{e}_2 = (1\,1)^\mathsf{T}$.

(d) Can you uniquely reconstruct the grid vectors given the grid points?

3. **Hexagonal Sampling Grid**

In this chapter only the rectangular sampling grid was used, as it is the only grid one is likely to encounter in practice. The *hexagonal grid* is known to be advantageous in certain situations. This grid is generated by the vectors $\mathbf{e}_1 = (1\ \ 0)^\mathsf{T}$ and $\mathbf{e}_2 = (1/2\ \ \sqrt{3}/2)^\mathsf{T}$.

(a) Sketch a part of the hexagonal grid.

(b) How many *neighbors* (i.e. other grid points with minimal distance) does each grid point have? How does this compare with the square grid?

(c) Work out the formulas for bilinear interpolation on the hexagonal grid.

(d) Calculate the weight kernels corresponding with the 6 parameters in a least squares 2nd order polynomial estimate of the function in the immediate neighborhood of a point on the hexagonal grid.

4. **Bilinear Interpolation**

Bilinear interpolation of a two dimensional sampled function F results in two dimensional function f_1 that is defined over the continuous domain $\mathbb{R}^2$. Is this function $f_1 : \mathbb{R}^2 \to \mathbb{R}$

(a) continuous?

(b) differentiable?

5. **Multi linear Interpolation**

Give the formula for multi linear interpolation in 3-dimensional space using the standard sampling grid, i.e. you have to generalize Eq.(2.3) to a 3D space.

6. **Matlab Image Interpolation**

In this exercise we only deal with grey value images $f : \mathbb{R}^2 \to \mathbb{R}$.

(a) Write a Matlab function that estimates the value $f(x, y)$ from the 4 surrounding neighbors using bilinear interpolation. Make sure that if someone asks for a value outside the image domain or on the border of the image domain you deal with that in a sensible way.

(b) Use this function to estimate the image values in N points along a line between the points (real values) (x_1, y_1) and (x_2, y_2). Plot these N points in a graph.

7. **Color Image Arithmetic**

   A color image maps each point in the visual domain onto a 3 element vector representing the amount of 'red', 'green' and 'blue' measured in that point. We refer to Chapter 5 for a detailed description of color and color images.

   In the section on image interpolation we have introduced interpolation techniques that take a weighted combination of image values (see Eq.(2.3)).

   For color images using the RGB representation the generalization is simple. The weights in the bilinear interpolation are scalars and multiplying a vector (the color value) with a scalar simply is the element wise multiplication of the vector.

   Redo the previous exercise such that your programs can handle color images as well.

8. **The Facet Model**

   In Section 2.7 we have introduced the facet model based on a 2nd order polynomial fitted to the sample values in a 3×3 neighborhood.

   (a) How can we estimate a third order derivative (say f_{xxx}) in case the facet model we have to use only provides the correlation kernels for a 2nd order polynomial?

   (b) An more direct way is to fit a 3th order polynomial to the sampled data. Can we fit a general 3th order polynomial to the 9 points in a 3×3 neighborhood?

   (c) Redo the section on the facet model now using a 5×5 neighborhood of sample points and fit a 3th order polynomial to the observed data. Your answer should be a collection of weight kernels in spatial layout needed in an image correlation operator to estimate the parameters in the 3rd order polynomial.

Chapter 3

Cameras

Projecting the 3D world on 2D images

When you look around you, the visible 3D world is optically projected onto the retina (photo sensitive layer) in our eye. The luminance (visual energy) is measured in a lot of points on the retina. The collection of all these measurements makes up an image of what we see in much the same way as a collection of pixels on the screen forms an image.

In the projection of the 3D world onto the 2D retina information about the 3D structure is lost. In this chapter we will look at the mathematics to model the camera used to look at the world with a computer. The ultimate goal of this chapter is to gain enough understanding of the basic workings of a camera to be able to reconstruct the 3D world from 2D images.

This will often require you to 'calibrate' the camera, i.e. to shoot some well chosen sample images that permit you to compute focal length and other internal parameters of the camera, as well as the external parameters of location and orientation. This turns the camera into a "3D measurement device". In practice (and in the lab course) we will work with a calibration toolbox, such as Bouguet's [1]. But understanding how such toolboxes work is a good exercise in setting up equations for observations and solving them for the desired parameters, so we treat it in detail.

3.1 The Pinhole Camera

The first 'law' of optics for computer scientists tells us that *light travels in straight lines.*[1] This observation is all that is needed to explain the workings of a pinhole camera.

Imagine a room with only a very tiny hole in the middle of one of the side walls. Imagine that you are standing next to the little hole facing the opposite wall. Outside the room a cow grazes in the field on a very shiny and brigth day. Light is coming into the room only through that very small hole. What would you see in the room?

[1] It is not a real 'law'; there are situations in which light travels along a curved path. Light paths are bent by objects with a large mass (like rays of light from very distant stars are bent by the sun). More close to home we may observe curved light paths in inhomogeneous media. Actually, light is a much more complex phenomenon, and the straight-line approximation, even for those cases, is just an approximate solution to the wave equation; natural light exhibits diffraction phenomena. In every practice of a computer vision scientist we may however ignore these deviations from the stated 'law'.

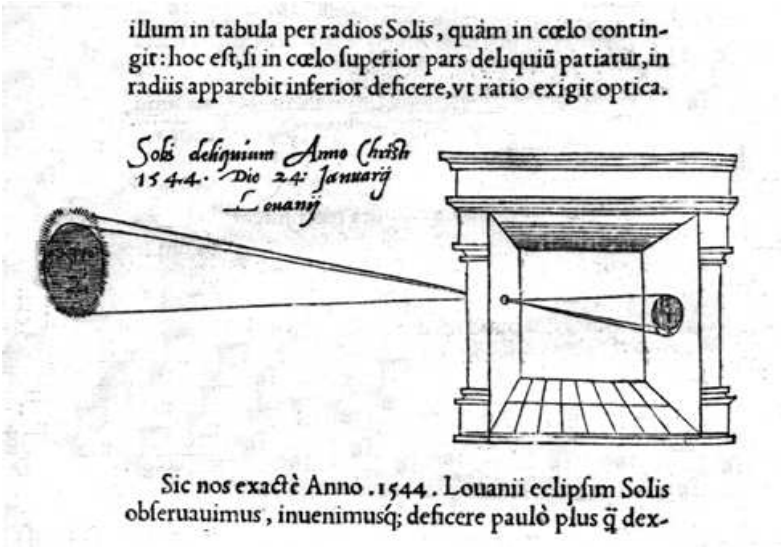


Figure 3.1: **Camera Obscura.**

This construction is an old one: the *camera obscura* used by astronomers to observe sun eclipses and later on by painters before the invention of photography to capture the 3D world as we see it onto a 2D canvas (see Fig. 3.1).

So even a hole can do imaging. Real cameras have lenses, which are a way of collecting more light onto the film, but also lead to side effects such as a limited range in which the image is sharp. In the pinhole cameras, everything is imaged sharp, but with a low intensity. Yet this extremely primitive camera provides us with a mathematical model that is actually a good point of departure for describing real cameras, since it contains the essence of the geometrical transformations between world and image.

In figure 3.2 we have illustrated our model[2]. The origin O is the pinhole of the camera. Point P (with coordinates (X, Y, Z)) lies on a real-world object and is projected onto the image plane I as point P'. The point where the optical axis intersects the image plane is the *principal point* or *image center* C..

We will assume that we know the distance between the pinhole O of the camera and the principal point C; this is the focal distance f of the camera. If we know the real-world coordinates X, Y and Z of point P, then it is possible to calculate the image coordinates x and y of the point P' projected onto the image plane I:

$$x = f\frac{X}{Z}, \qquad y = f\frac{Y}{Z}.$$

We switch to homogeneous coordinates so we can write the above equations in matrix form. Homogenous coordinates are explained in many sources, e.g. [5] or [3]. Basically an extra scaling coordinate w is added to a point and all other coordinates x, y, z need to be divided by w to get normal 3D coordinates out of a 4D homogeneous vector. So to embed a point (x, y, z) into

[2]The real situation is 3D, but since it is the same for X and Y, we decided to show it as a 2D scene where the vertical axis can be either, as that is more instructive.

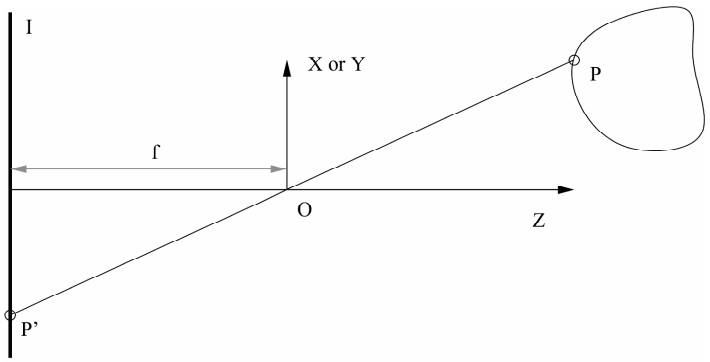


Figure 3.2: **A schematic version of the perspective camera model.** The origin O is the camera pinhole; point P is an object point with its Z component along the horizontal axis (also known as the optical axis); the vertical axis can be its X or Y component. I is the image plane which is perpendicular to the horizontal optical axis and point P' is the projection of P onto the image plane I. f and its double-arrowed line illustrate that the distance between the pinhole O and the principal point C is the focal distance f.

homogeneous coordinates, you mostly use $w = 1$ and map it to:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix},$$

but to interpret homogeneous coordinates you divide the useful ones by the last one:

$$\begin{pmatrix} a \\ b \\ c \\ w \end{pmatrix} \mapsto \begin{pmatrix} a/w \\ b/w \\ c/w \end{pmatrix}.$$

For this reason, we will usually denote such a homogeneous point by

$$\begin{pmatrix} w\,x \\ w\,y \\ w\,z \\ w \end{pmatrix} \quad \text{or} \quad w \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix},$$

so that we can use the (x, y, z) as symbols for the actual coordinates after the homogeneous computation.

Instead of introducing the w, some people use the symbol $\sim$ for equality between homogeneous coordinates in the sense that they are equal in their interpretation. This means that all elements may be multiplied by the same constant factor. Often, this is done to display the homogeneous vector in the simply interpreted normalized form. (We call a homogeneous vector *normalized* when the w coordinate is 1.)

With all this, you can verify that the non-linear projection equations can be written as a linear transformation in the homogeneous coordinate representation:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} wx \\ wy \\ w \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/f & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \sim \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Verify this, this should become basic knowledge (hint: compute w, then divide it out).

3.1.1 Modelling real cameras

Now we extend our model to make it more suitable for real-world usage. We would really like to go to *pixels*, not *millimeters* in the image plane, to make addressing of the image more closely correspond to the way it is stored. So we need to multipy the x and y values by a factor denoting the number of pixels per millimeter, i.e. the reciprocal of the pixel size. But the pixels may not be square in a real camera (for instance because the CCD chip that contains the light-sensitive elements is not exactly orthogonal to the optical axis of the camera), so we have factors s_x and s_y to account for this:

$$x = s_x f \frac{X}{Z}, \qquad y = s_y f \frac{Y}{Z}.$$

Instead of writing $s_x f$ and $s_y f$ all the time, we will combine these factors into single terms f_x and f_y. The same non-orthogonality to the optical axis actually leads to a square becoming a parallellogram, not only with different x and y-dimensions, but also a bit of *shear*, meaning that x changes slightly proportional to y with some proportionality factor s. This causes an off-diagonal term in the projection matrix (see below). Since we can always use the total camera rotation to make the y axis align with a non-skewed direction, only one off-diagonal term is required to model this effect. Often, you find that s is very small, but you may want to confirm that before you decide to put it to zero (and similarly with non-squareness of the pixels).

Another feature of real cameras is that the origin of the coordinate system generally does not lie at the principal point/image center C, but in or near one of the image corners. We introduce u_0 and v_0 to translate the principal point to the right location. For a camera without shear, this would be:

$$x = f_x \frac{X}{Z} + u_0, \qquad y = f_y \frac{Y}{Z} + v_0.$$

In matrix form, all these effects are combined to:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & s & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

This matrix is called the *intrinsic camera matrix* M_{int}, and the parameters it contains are the *intrinsic camera parameters*.

3.1.2 Separating camera frame and world frame

Currently our world frame has its origin at the pinhole of the camera, as it is equal to the camera frame. This is rather inflexible: we do not want the origin of our world coordinate system to

lie at the pinhole. We introduce an intermediate rigid body transformation M_{ext} which converts from the world frame to the camera frame (see [5] or [3]):

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = M_{int} M_{ext} \vec{X} \tag{3.1}$$

Note carefully what the matrix M_{ext} does: if you feed it with the world origin $(0\ 0\ 0\ 1)^\mathsf{T}$ in homogeneous world coordinates, it produces $(t_x, t_y, t_z, 1)^\mathsf{T}$, so those parameters represent the location of the world origin in camera coordinates. Similarly, $(r_{11}, r_{21}, r_{31}, 0)^\mathsf{T}$ represents a homogeneous direction vector (since the last component is zero) in camera coordinates, corresponding to the homogeneous world direction vector $(1, 0, 0, 0)^\mathsf{T}$, i.e. the x-axis direction of the world.

In fields like robotics and computer graphics, you often know the homogeneous matrix telling you where the camera frame is in world coordinates:

$$\begin{pmatrix} R & \mathbf{t} \\ 0^\mathsf{T} & 1 \end{pmatrix},$$

with R the orthogonal matrix of the rotation, and $\mathbf{t}$ the translation vector. Then for M_{ext} you need the inverse. This is:

$$M_{ext} = \begin{pmatrix} R^\mathsf{T} & -R^\mathsf{T}\mathbf{t} \\ 0^\mathsf{T} & 1 \end{pmatrix}$$

Verify this! Such re-representations are a source of errors, so you need to get them right. It really is simple to check which one you need: just transform the point at the origin (as we did above) and check whether the translation you get is in the right direction. That tells you 'what' is relative to 'what', and then you have make sure that you have the same rule for the directional colums of the matrix.

We can clean up the equations a bit, for the last column of M_{int} is all zero, and will not do anything with the last row of the homogeneous coordinate matrix; so we can leave them both out without affecting the result. That gives

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = M_{int} M_{ext} \vec{X} \tag{3.2}$$

as the pinhole camera projection equation.

3.1.3 Pseudo-3D

The structure of projections is already enough to fake a 3-D understanding of a scene. Consider figure 3.3 in which we show a scene containing a calibration cube, consisting of 2 orthogonal chessboards. It seems natural to want to use this as a coordinate frame, and to assign the bottom corner the 3-D coordinates $(0, 0, 0)$, one block up as $(0, 0, 1)$, etcetera. You might expect that you would be able to extend it, and that you could draw cubes in the image as if they were really projections of actual 3D objects, as in the figure on the right. You will explore this in a web exercise, here we give some background. The most important in this section are the linear algebra-based estimation techniques, you will encounter those many times.

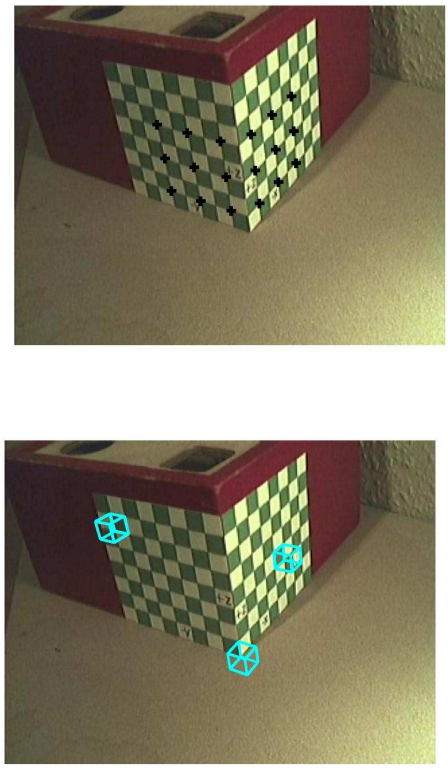


Figure 3.3: **Pseudo-3D drawing in an image with a calibration cube.**

The bottom line of the above pinhole model is that real world points (X, Y, Z) are mapped to pixel coordinates (x, y) via a transformation that is linear when expressed in homogeneous coordinates:

$$w \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = M \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Of course, the m_{ij} are dependent on the actual intrinsic and extrinsic parameters, but there is no way of knowing those from a single image. All we know is correspondences between imposed world coordinates based on the calibration cube coordinate frame, and where those points are in pixel coordinates. So we have 'measured pairs' (x_i, y_i) and (X_i, Y_i, Z_i). Having enough of those can determine the matrix M. Since M is a 3×4 matrix, it has twelve degrees of freedom, but if we multiply it with a constant the homogeneous coordinates for the points are still interpreted the same – so there are really only 11 relevant parameters in M.

When we work out the projection equation explicitly we get the following:

$$\begin{aligned} w_i x_i &= m_{11} X_i + m_{12} Y_i + m_{13} Z_i + m_{14} \\ w_i y_i &= m_{21} X_i + m_{22} Y_i + m_{23} Z_i + m_{24} \\ w_i &= m_{31} X_i + m_{32} Y_i + m_{33} Z_i + m_{34} \end{aligned}$$

This is linear in the unknown parameters m_{ij}, so it makes sense to rewrite it as a transformation on the unknown coefficient vector

$$\mathbf{m} = (m_{11}, m_{12}, \ldots, m_{33}, m_{34})^{\mathsf{T}}.$$

The w_i ocurring on the left, when substituted from its equation, causes terms involving products of $x_1 Y_1$ et cetera. Verify that the final result is:

$$A\mathbf{m} = 0,$$

where

$$A = \begin{pmatrix} X_1 & Y_1 & Z_1 & 1 & 0 & 0 & 0 & 0 & -x_1 X_1 & -x_1 Y_1 & -x_1 Z_1 & -x_1 \\ 0 & 0 & 0 & 0 & X_1 & Y_1 & Z_1 & 1 & -y_1 X_1 & -y_1 Y_1 & -y_1 Z_1 & -y_1 \\ X_2 & Y_2 & Z_2 & 1 & 0 & 0 & 0 & 0 & -x_2 X_2 & -x_2 Y_2 & -x_2 Z_2 & -x_2 \\ 0 & 0 & 0 & 0 & X_2 & Y_2 & Z_2 & 1 & -y_2 X_2 & -y_2 Y_2 & -y_2 Z_2 & -y_2 \\ . & . & . & . & . & . & . & . & . & . & . & . \\ X_n & Y_n & Z_n & 1 & 0 & 0 & 0 & 0 & -x_n X_n & -x_n Y_n & -x_n Z_n & -x_n \\ 0 & 0 & 0 & 0 & X_n & Y_n & Z_n & 1 & -y_n X_n & -y_n Y_n & -y_n Z_n & -y_n \end{pmatrix} \tag{3.3}$$

As you see, each data point pair gives two equations, so 6 pairs should be enough to solve for the 11 unknowns. Using more points would not hurt, they should just make the estimation more accurate.

The need to solve such a set of overdetermined equations encoded in $A\mathbf{m} = 0$ occurs very frequently, so you should know the technique, which we explain in appendix B. The answer is simply that the optimal $\mathbf{m}$ satisfying this equation is the last column of V in the singular value decomposition of A as $A = U\,D\,V^{\mathsf{T}}$. That really is a bit of linear algebra magic worth remembering. (You *must* read and understand the appendix to see how that comes about, but also to find out what is 'optimal' to this solution.)

That then solves the estimation of M, and with that any point in the pseudo-3D coordinates (X, Y, Z) can be mapped simply onto its (x, y) pixel coordinates using the projection equation. Therefore we can draw the pseudo-3D cubes of Fig. 3.3.

3.1.4 Distortion

What we have discussed so far is still a camera using pinhole-geometry for its imaging. This is a reasonable approximation for many good cameras, but in cheap webcams an important effect is image *distortion* due to the cheap lenses. Straight lines are depicted as bent, more so as they are further from the center.

The simplest type of distortion is a radial displacement of vectors. Taking $\mathbf{x} = (x, y)^\mathsf{T}$ as a position vector from the optical center of the image, and $\mathbf{x}_d$ as that vector after distortion, you could use:

$$\mathbf{x}_d = \mathbf{x}\ (1 + k_1\, r^2 + k_2\, r^4),$$

with $r^2 = x^2 + y^2$ and k_1 and k_2 the distortion parameters. This is enough for most purposes and/or cameras. Note that the powers of r have to be for a distortion that is symmetrical for displacements around the center, so this is in fact a second order approximation in reasonable deviations. Depending on the sign of k_1, you can deform a square to a 'barrel' (sides bulging out out) or a 'pillow' (vertices bulging out). Q: Which is which?

The distortion is applied between the intrinsic and extrinsic matrices: $M_{int}\text{distort}(M_{ext}\vec{X})$, but because of its non-linearity it cannot be written as a matrix multiplication.

The distortion used by the Bouguet toolbox is more general, both ellipsoidal and of higher order in r:

$$\begin{aligned} x_d &= x + k_1 r^2 x + k_2 r^4 x + 2k_3 xy + k_4(r^2 + 2x^2) + k_5 r^6 x \\ y_d &= y + k_1 r^2 y + k_2 r^4 y + k_3(r^2 + 2y^2) + 2k_4 xy + k_5 r^6 y \end{aligned} \tag{3.4}$$

with k_1 through k_5 the distortion parameters.

3.2 Camera Calibration

The Bouguet camera calibration toolbox is based on a paper by Zhang [9], which makes good reading, and should be seen as part of these notes. In Appendix B.1 we give some hints on how to read that paper.

It uses a chessboard of known size as a calibration tool. Taking several images of this chessboard leads to many sets of points coupled by the distorted homography of eq.(3.1) and eq.(3.4). Zhang first estimates the intrinsic parameters by assuming the distortion is small, and then uses that as the seed in a clever numerical optimization scheme to find the distortion parameters.

Note in the paper that the method of his Appendix A corresponds to our estimation of M in section 3.1.3 (although he calls the matrix H, for homography), but in a 2D setting. He then proceeds to split this composite M (or H) into the constituent M_{int} and M_{ext}, in a surprisingly simple manner, merely by using the orthogonality constraint on the rotational part of the rigid body motion matrix to set up a homogeneous linear system of equations for the matrix ${M_{int}}^{-T}{M_{int}}^{-1}$. To set that up, he needs at least two views of chessboards. That system can then be solved by the standard SVD techniques, and the result determines M_{int}. Then M_{ext} can be determined from that. But this M_{ext} is not necessarily a rigid body transformation, as it is possible for the base vectors of the X and Y axes to have a length different from 1 (they are slightly scaled then). This happens because the reference points are generally not exact: they are noisy. Here are two solutions to this problem.

- Zhang, in his Appendix C, finds the closest rotation to the rotational part R of the rigid body transformation, by doing an SVD. This is a trick you should know! If $R = U\ \Sigma\ V^\mathsf{T}$, then $U\ V^T$ is the rotation matrix closest to R (in the sense of the Frobenius norm).

- Bouguet, in his toolbox, solves this by taking the M_{ext} we have derived as an initial guess for the real M_{ext}. He then converts the rotational part to Rodrigues coordinates[3]. He then performs a gradient descent to minimize the reprojection error in the reference points on the 3 Rodrigues coordinates and the 3 elements of the translation vector. The Rodrigues coordinates can only describe rotations without scaling and ensure that the final version of matrix M_{ext} is indeed a rigid body transformation.

Bouguet's method is more accurate than Zhang's method, since it also adapts the estimation of the translational part.

Estimating the distortion is a bit less straightforward, since it is a non-linear problem. Zhang solves it iteratively by linearization, using a Levenberg-Marquardt algorithm. Combining these methods, we can determine all intrinsic parameters and the distortion. These are valid for all images we take with this camera (as long as we do not touch focus and zoom, of course). They turn the camera into a geometrical measurement instrument for locations and orientations.

3.3 Voxel coloring and space carving

3.4 Voxel Coloring

Voxel coloring is an algorithm to reconstruct the 3D shape from a number of input photographs with known camera locations. It was originally introduced by Seitz and Dyer in [7]. The reconstruction consists of a regular grid of cube-shaped voxels. Voxels are 'volume elements' in analogy with pixels. Like a pixel, every voxel is associated with a color. See figure 3.4 for an example of a voxel representation. More information about these representations can be found in [5].

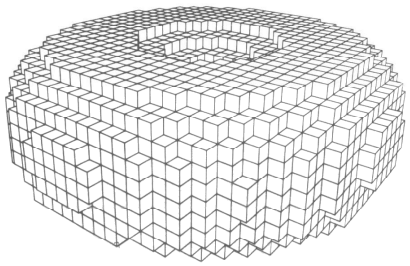


Figure 3.4: **Example of a voxel representation. Picture taken from [5].**

The problem addressed by voxel coloring is the assignment of colors to points in a 3D volume as to be consistent with the input photographs. If a single point has the same color in all images from which it is visible, then it should be given that color. If the colors do not match, then there probably is no point in the 3D volume there and the voxel should be removed. This basic principle underlying voxel coloring is illustrated in figure 3.5.

When rendering the voxel reconstruction from the photo viewpoints, it should be 'identical' to the input photographs (because of limitations of the voxel representation used, they are not

[3]Rodrigues coordinates describe a rotation with just 3 parameters: an unit vector (2 parameters) and a rotation angle around this vector (1 parameter). See [4] for an introduction to Rodrigues coordinates.

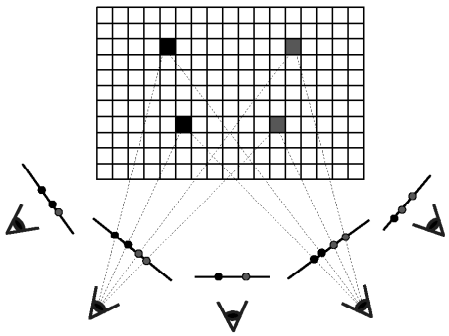


Figure 3.5: **Given a set of input images and a voxel space, we want to assign colors to voxels in a way that is consistent with all images. Picture taken from [8].**

always identical). When a reconstruction matches with the input images, it is *photo-consistent*. However, a photo-consistent reconstruction is not unique, as is illustrated in figure 3.6. The different types of ambiguity are explored in [8].

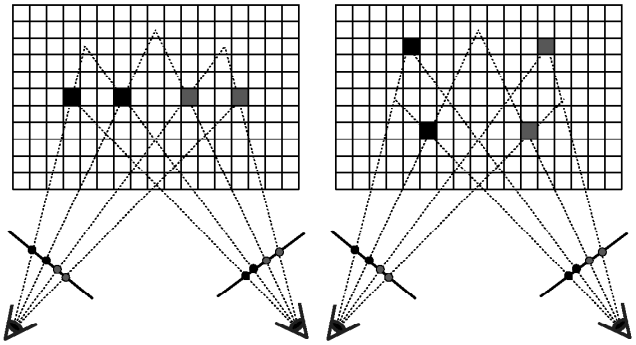


Figure 3.6: **Both voxel colorings appear to be identical from these two viewpoints but have no voxels in common. Picture taken from [8].**

The *photo-hull*, introduced in [6], is the union of all photo-consistent shapes and is thus the maximally photo-consistent shape. Contrary to a photo-consistent reconstruction, the photo-hull is uniquely defined and happens to be the reconstruction created by voxel coloring (this is shown in [6]). If we apply voxel coloring to the case of figure 3.6 now, then the reconstruction is uniquely defined as is shown in figure 3.7.

The working of voxel coloring relies heavily on the ability to compare colors between different images. These colors are only the same if we assume a Lambertian reflection model, which says that all objects in the scene reflect the light equally in all direction. If we do not assume a Lambertian model, then the amount of light reflected would depend on the camera angle being used, which means that a single point in space can have multiple colors.

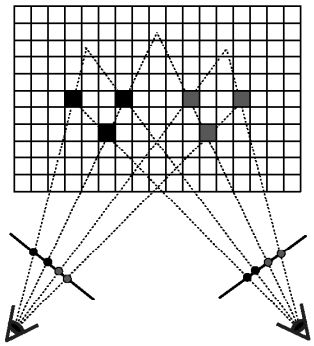


Figure 3.7: **Voxel coloring using the photo-hull. Every voxel has the same color in every reconstruction in which it is contained. Picture taken from [8].**

3.4.1 Occlusion handling

In order to handle occlusion properly, the voxel coloring algorithm traverses the voxel space in a special order. Voxels closer to the camera are visited first. This ensures that a voxel cannot be occluded by an unvisited voxel (since an occluding voxel must be closer to the camera, it has already been visited).

Across all input images, this is known as the *ordinal visibility constraint* (see [7]), which ensures that for scene points P and Q, if P occludes Q, there is some metric which says that $\|P\|$ is smaller than $\|Q\|$ across all input images (thus if P occludes Q, then there are no images possible in which Q is occluded by P).

The ordinal visibility constraint is satisfied when no scene point is contained within the convex hull of camera centers. In case a pinhole camera is assumed, this camera center is equal to the pinhole of the camera. The metric above can be taken as the distance to the convex hull around the camera centers.

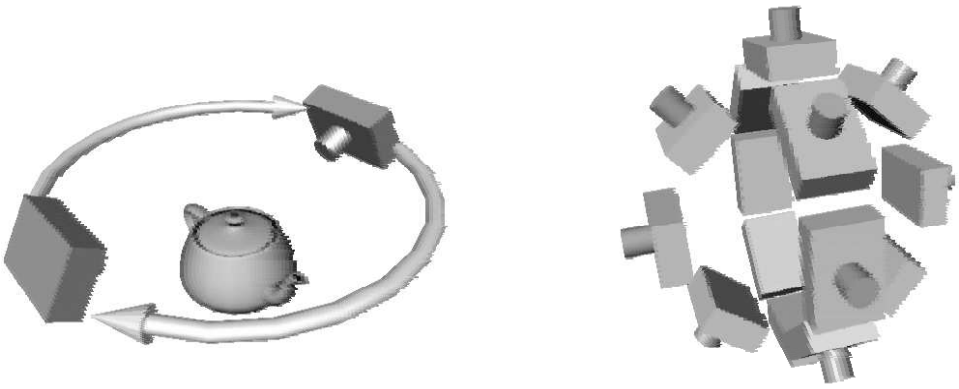


Figure 3.8: **Camera configurations that satisfy the ordinal visibility constraint. Picture taken from [8].**

In figure 3.8 two practical camera setups are shown which satisfy the ordinal visibility constraint. Note that the constraint implies that having two cameras on exactly opposite sides is not allowed. Because of this, not all sides of an object can be reconstructed. There are generalizations of voxel coloring, such as Space Carving [6] and Generalized Voxel Coloring [2], which can both handle arbitrary camera positions.

3.5 Voxel Coloring algorithm

Using the principles described in the previous section, we can now construct the basic voxel coloring algorithm. This algorithm assumes that all voxels V are traversed in an order which satisfies the ordinal visibility constraint (described in section 3.4.1).

```
for every voxel V do
  pixels ← ∅
  for all images I do
    pixels ← pixels ∪ selectUnmarked(projectVoxel(V, I))
  end for
  consistent ← consistencyCheck(pixels)
  if pixels ≠ ∅ and consistent then
    colorVoxel(V, mean(pixels))
    mark(pixels)
  else
    carveVoxel(V)
  end if
end for
```

Every voxel is projected onto the input images, and all pixels that have not yet been marked (as done) are collected. If there are no pixels the voxel projects onto, then the voxel is carved. If the pixel collection is consistently colored, then the voxel is accepted and given the mean color of the collection. The pixels are then marked (to indicate they are done). If the collection does not have a consistent color, then the voxel is carved.

A common optimization is to use a *sprite projection* when projecting a voxel onto an image: all vertices of the voxel are projected and a bounding box around the projected points is used to approximate the shape of the actual voxel projection. Constructing a bounding box around eight projected points is much faster than scan-converting the actual projection, while being more accurate than a simple point projection (which only projects the center of the voxel).

Bibliography

[1] Jean-Yves Bouguet. *Camera calibration toolbox for Matlab.* `http://www.vision.caltech.edu/bouguetj/calib_doc` You will also get a lot of hits searching for 'bouget', apparently many people can't spell.

[2] Bruce Culbertson, Tom Malzbender and Greg Slabaugh. *Generalized Voxel Coloring.* In Proceedings of the ICCV Workshop, Vision Algorithms Theory and Practice, Springer-Verlag Lecture Notes in Computer Science 1883, pages 100-115, 1999.

[3] Leo Dorst, *Introduction to Robotics*, dictaat van de cursus 'Robotica' en het project 'Zoeken, Sturen en Bewegen', Informatica Instituut, Universiteit van Amsterdam, 1992-heden.

[4] Laura Downes and Alex Berg. *CS184: Computing rotations in 3D.* `http://www.cs.berkeley.edu/~ug/slide/pipeline/assignments/as5/rotation.html`

[5] J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesly Publishing Co, section 12.6, 1990.

[6] Kiriakos N. Kutulakos and Steven M. Seitz. *A theory of shape by space carving.* In International Journal of Computer Vision, 38(3): pages 198-218, 2000.

`http://www.phidgets.com`

In Proceedings of the DARPA Image Understanding Workshop, pages 315-321, 1998.

[7] Steven M. Seitz and Charles R. Dyer. *Photorealistic scene reconstruction by voxel coloring.* In Proceedings of the Computer Vision and Pattern Recognition Conference, pages 1067-1073, 1997.

[8] Steven M. Seitz and Charles R. Dyer. *Photorealistic scene reconstruction by voxel coloring.* In International Journal of Computer Vision, 35(2): pages 151-173, 1999.

[9] Z. Zhang, *A Flexible New Technique for Camera Calibration*, Technical Report MSR-TR-98-71, Microsoft Research, 1998. Available on the web at `http://research.microsoft.com/ zhang/calib/`.

Chapter 4

Local Image Statistics

In section 4.1 we briefly introduce some statistical descriptors for a collection of image values. We will not only look at scalar image values (the luminance itself) but we will also consider non-scalar image values like the gradient vectors (to be introduced in a later chapter). In this section we consider finite and countable ensembles of image values resulting from sampling an image domain (or an image region, i.e. a small subset of the image domain).

In section 4.2 we will take the more fundamental route of defining descriptive statistical measures for all image values in a bounded image domain region. For an image $f : \mathbb{R}^d \rightarrow \mathbb{R}$ a region $\mathcal{R} \subset \mathbb{R}^d$ with finite area contains an uncountable number of points. Statistical descriptors then are to be casted in an integral form. These integral descriptors can be faithfully approximated using a sampled representation of the image.

In section 4.3 we look at a simple algorithm from the important field of *statistical pattern recognition*. k-Means clustering provides a way to cluster an ensemble of values into meaningful subsets.

In section 4.4 we follow common practice in image processing to define image operators (i.e. an operator that takes an image as input and produces a new image) by looking at the collection of image values in the neighborhood in a location in an image and calculating a measure for that collection of values. This new value is then assigned to the same location in the 'output' image. This procedure is then repeated for all locations in the image. We discuss some of the well-known of these image operators (often called *filters*) like the *mean* filter, the *variance* filter, the *median* filter, the *percentile* filter, the *minimum* filter and the *maximum* filter.

4.1 Descriptive ensemble statistics

Let $\Omega = \{v_i \mid i = 1, \ldots, N\}$ be a collection or ensemble of (image) values. For the moment we don't care in what way those values are obtained. It could be either by observing a scene in one location at different points in time, or by observing one scene in several points in a scene simultaneously, or... (an endless number of experiments can be set up to obtain such a collection of values.) In fact most of what is described in this section is equally valid for any ensemble of (measurement) values. Once we have such a collection of values there is often a need to

- characterize the entire ensemble using only a few parameters,
- cluster the values in the ensemble into semantically meaningful subsets

In this section we look at some well-known statistical descriptors for ensembles.

4.1.1 Scalar ensembles

Assume that the values v_i in the ensemble $\Omega = \{v_i \mid i = 1, \ldots, N\}$ are real scalars.

Mean value. The mean value μ of the ensemble $\Omega = \{v_i \mid i = 1, \ldots, N\}$ is given by:

$$\mu(\Omega) = \frac{1}{N} \sum_{i=1}^{N} v_i. \tag{4.1}$$

In case the ensemble is known a priori to represent the values obtained from a statistical distribution, the mean or *average* value is an estimator for the *expected* value.

The variance. The variance ν measures the average[1] of the quadratic differences from the mean value in the ensemble:

$$\nu(\Omega) = \frac{1}{N} \sum_{i=1}^{N} (v_i - \mu)^2 . \tag{4.2}$$

Moment analysis. Both the mean and the variance are examples of moment analysis. The p-th order moment μ_p is defined as:

$$\mu_p(\Omega) = \frac{1}{N} \sum_{i=1}^{N} v_i^p,$$

the p-th order *central moment* is denoted as $\bar{\mu}_p$ and is defined as:

$$\bar{\mu}_p(\Omega) = \frac{1}{N} \sum_{i=1}^{N} (v_i - \mu)^p.$$

Note that the mean equals μ_1 and the variance is equal to $\bar{\mu}_2$.

Weighted mean and variation. In case it is known a priori that the values in an ensemble are not equally important, a valuable extension to the mean value is the weighted mean value. Let $\{\alpha_i\}_{i=1,\ldots,N}$ be real positive scalars expressing the relative importance of each measurement. These scalars are then used to *weigh* the values:

$$\mu(\Omega) = \frac{\sum_{i=1}^{N} \alpha_i v_i}{\sum_{i=1}^{N} \alpha_i}.$$

The variation can be generalized in an analogous way.

Median value. The median value of an ensemble (where N is odd) is defined as the value $p_{0.5}$ such that $(N + 1)/2$ of the values in the ensemble are less than the median value. The median value is easier to interpret using a reordering of the values in the ensemble. Let $v_i^\star$ be the reordering such that:

$$v_1^\star \leq v_2^\star \leq \cdots \leq v_N^\star$$

[1] We use the N-normalization in this chapter because we value the geometrical interpretation. The $(N-1)$-normalization should be chosen in case one values the fact that then the variance is an unbiased estimate of the 'underlying' Gaussian distribution

then the median value is given by:

$$p_{0.5}(\Omega) = v^{\star}_{(N+1)/2}.$$

For an even number of values in the ensemble the median value is often defined as the mean of the two central values in the ordered sequence.

The median value and the mean value are closely related. Their use in image processing is also similar: both are often used in case the values in the ensemble are thought to be equal to some unknown values but corrupted with noise. Both the mean and the median serve as a indicators of the 'true value'.

The correspondence between mean and median value becomes more clear if we look at the following characterizations that are very closely related. Let Ω be an ensemble of scalar values, Then the mean value μ is defined as the value that minimizes the sum of all quadratic differences in the ensemble, i.e.:

$$\mu(\Omega) = \arg\min_{m} \left\{ \sum_{i=1}^{N} (v_i - m)^2 \right\} \tag{4.3}$$

(that the mean value minimizes the sum of quadratic differences is easily proved, see exercise 1).

For the median value we have a similar expression, now we minimize the absolute difference instead of the quadratic difference:

$$p_{0.5}(\Omega) = \arg\min_{m} \sum_{i=1}^{N} \{|v_i - m|\}.$$

Due to the quadratic difference the mean emphasizes the measurements that are far away from the mean more than the median does. In case the ensemble contains *outliers*, the median value probably gives better results (see exercise 2).

k-Percentile value. Instead of taking the middle value in the ordered sequence of values, we can take the value p_k such that $100k$ percent of the values are less than or equal to that value.

Weighted median and percentile values. Again let $\{\alpha_i\}_{i=1,\ldots,N}$ indicate the relative importance of the values within the ensemble. The α_i are assumed to be positive integers. Multiplicative weighing is of little use here. Instead we duplicate the value v_i, α_i times, denoted as $\alpha_i \diamond v_i$. The resulting new ensemble is then input to the calculation of the standard median and percentile filters.

Ensemble extremes. The minimum value p_0 and maximum value p_1 in an ensemble are important descriptors. These two descriptors indicate the range of values in the ensemble. For an ensemble that is assumed to be indicative of a true value corrupted with noise the range is a measure of the spread around the true value.

4.1.2 Non-scalar ensembles

The only non-scalar descriptors that will be discussed in this book are the D-dimensional real vectors. These vector ensembles include the gradient vector ensembles, color ensembles and the ensembles of spatial locations in the visual domain.

Let $\Omega = \left\{ \mathbf{v}_i \;\middle|\; i = 1, \ldots, N \right\}$ be an ensemble of vectors. The scalar descriptors based on (central) moment analysis can be simply generalized to ensembles of vectors.

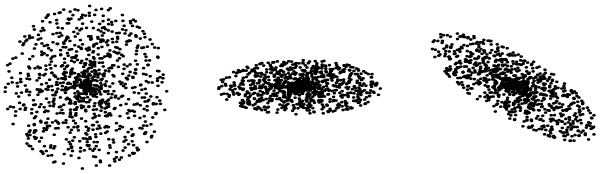


Figure 4.1: **Two dimensional vector clusters.** From left to right: (a) an isotropic cluster (spread in all directions is equal), (b) and (c) elongated clusters (the cluster in (c) is a rotated version of the cluster in (b).

Mean value. The mean value $\mu(\Omega)$ of the ensemble is given by:

$$\mu(\Omega) = \frac{1}{N}\sum_{i=1}^{N}\mathbf{v}_i.$$

To understand the vectorial mean make sure you understand that the mean of two vectors is the vector with endpoint in the location halfway in between the two endpoints of the vectors. The vectorial mean thus boils down to calculating the mean of the individual components of the vectors.

The variance. The generalization of the variance measure ν for scalar ensembles to a variance measure for vector ensembles is not so easily defined. Consider the following variance measure ν_d:

$$\nu_d = \frac{1}{N}\sum_{i=1}^{N}\|\mathbf{v}_i - \mu\|^2 \tag{4.4}$$

where we take the square of the norm of the difference vector $\mathbf{v}_i - \mu$ as a measure of the difference. We write ν_d indicating that this 'variance' is based on a distance measure. Obviously with this measure of variance we can distinguish between the vector ensembles sketched in figure 4.1 (a) and (b) but not between the ensembles in (b) and (c). These last two examples are just rotated versions of the same ensemble and therefore have the same ν_d variance (see exercise 5).

Capturing the 'orientation' of the ensemble clusters is a goal in the analysis of the gradient vector ensembles (see figure **??**). Note that the spread in the gradient vectors is large in one direction and much less in the perpendicular direction. We thus would like a measure of variance that is dependent on the orientation in which we measure the variance.

Consider the ensemble of vectors $\mathbf{v}_i$ again. Let r be a direction vector with norm 1 in which direction we would like to measure the variance. We can 'borrow' the scalar variance measure, Eq.(4.2), in case we first project all values $\mathbf{v}_i$ onto the direction vector $\mathbf{r}$, this results in a *scalar ensemble* with values s_i:

$$s_i = \mathbf{r}^\mathsf{T}\mathbf{v}_i.$$

The mean of the scalar ensemble $\{s_i \mid i = 1, \ldots, N\}$ can be calculated as:

$$\mu_\mathbf{r} = \frac{1}{N}\sum_{i=1}^{N}s_i = \frac{1}{N}\sum_{i=1}^{N}\mathbf{r}^\mathsf{T}\mathbf{v}_i.$$

Because $\mathbf{r}$ is a constant vector we can take it out of the summation:

$$\mu_{\mathbf{r}} = \mathbf{r}^{\mathsf{T}} \left(\frac{1}{N} \sum_{i=1}^{N} \mathbf{v}_i \right) = \mathbf{r}^{\mathsf{T}} \mu,$$

This shows that the scalar mean in the $\mathbf{r}$-direction is the projection of the vectorial mean onto the direction vector.

The variance in the scalar ensemble s_i is:

$$\nu_{\mathbf{r}} = \frac{1}{N} \sum_{i=1}^{N} (s_i - \mu_{\mathbf{r}})^2.$$

Again substituting the expression for the scalar values s_i we obtain:

$$\begin{aligned}
\nu_{\mathbf{r}} &= \frac{1}{N} \sum_{i=1}^{N} (\mathbf{r}^{\mathsf{T}} \mathbf{v}_i - \mathbf{r}^{\mathsf{T}} \mu)^2 \\
&= \frac{1}{N} \sum_{i=1}^{N} (\mathbf{r}^{\mathsf{T}} (\mathbf{v}_i - \mu))^2 \\
&= \frac{1}{N} \sum_{i=1}^{N} (\mathbf{r}^{\mathsf{T}} (\mathbf{v}_i - \mu))(\mathbf{r}^{\mathsf{T}} (\mathbf{v}_i - \mu)) \\
&= \frac{1}{N} \sum_{i=1}^{N} \mathbf{r}^{\mathsf{T}} (\mathbf{v}_i - \mu)(\mathbf{v}_i - \mu)^{\mathsf{T}} \mathbf{r} \\
&= \mathbf{r}^{\mathsf{T}} \left(\frac{1}{N} \sum_{i=1}^{N} (\mathbf{v}_i - \mu)(\mathbf{v}_i - \mu)^{\mathsf{T}} \right) \mathbf{r} \\
&= \mathbf{r}^{\mathsf{T}} C \mathbf{r}
\end{aligned}$$

The matrix C is called the *covariance matrix*[2]. Let $\mathbf{v}_i = (v_{1i} \quad v_{2i})^{\mathsf{T}}$ and $\mu = (\mu_1 \quad \mu_2)^{\mathsf{T}}$ then the covariance matrix is equal to:

$$\begin{aligned}
C &= \frac{1}{N} \sum_{i=1}^{N} \begin{pmatrix} (v_{1i} - \mu_1)^2 & (v_{1i} - \mu_1)(v_{2i} - \mu_2) \\ (v_{1i} - \mu_1)(v_{2i} - \mu_2) & (v_{2i} - \mu_2)^2 \end{pmatrix} \\
&= \begin{pmatrix} \frac{1}{N} \sum_{i=1}^{N} (v_{1i} - \mu_1)^2 & \frac{1}{N} \sum_{i=1}^{N} (v_{1i} - \mu_1)(v_{2i} - \mu_2) \\ \frac{1}{N} \sum_{i=1}^{N} (v_{1i} - \mu_1)(v_{2i} - \mu_2) & \frac{1}{N} \sum_{i=1}^{N} (v_{2i} - \mu_2)^2 \end{pmatrix}
\end{aligned}$$

Notice that C_{11} is the scalar variance in the first element of the value vectors and that C_{22} is the scalar variance in the second element of the value vectors. The $C_{12} = C_{21}$ is the *covariance* (for higher dimensional data vectors there is more than one) needed to calculate the scalar variance in an arbitrary direction $\mathbf{r}$.

The scalar variance in an arbitrary direction $\mathbf{r}$ can thus be calculated as the *quadratic form*

$$\nu_{\mathbf{r}} = \mathbf{r}^{\mathsf{T}} C \mathbf{r}$$

where C, the covariance matrix, is a symmetric positive semi definite matrix (i.e. the variance is always greater than or equal to zero). For such a quadratic form we can always find a direction

[2]The statistical theory that explains the properties of the covariance matrix is treated in a later stage of the computer science studies in Amsterdam.

$\mathbf{r}_1$ such that the scalar variance is maximal and then in the orthogonal direction $\mathbf{r}_2$ the scalar variance is minimal. These directions are given by the *eigenvectors* of the covariance matrix and the scalar variances are equal to the corresponding *eigenvalues*[3].

Let $\mathbf{r}_1$ and $\mathbf{r}_2$ be the two eigenvectors of the covariance matrix C with corresponding eigenvalues λ_1 and λ_2 such that $\lambda_1 \geq \lambda_2$. Without proof we state that:

- the vector values in the ensemble form a cluster of points around the center of gravity μ.
- when projecting all vectors in the ensemble on the first eigenvector most of the variance in the ensemble is preserved: in other words: the cluster of vector values is most elongated in the $\mathbf{r}_1$ direction.
- an eigenvalue $\lambda_2 \approx 0$ denotes that in the $\mathbf{r}_2$-direction there is little variance: i.e. all observation vectors $\mathbf{x}_i$ (almost) lie on the line (in 2-dimensional space) in the $\mathbf{r}_1$-direction through the ensemble mean.

The eigenvalue analysis of the covariance matrix provides a valuable and often used tool to describe the elongation and orientation of point clouds in space. Such analysis can (and *is*) carried out for spatial points, color vectors, gradient vectors etc.

Weighted means and moments. The generalization to weighted means and moments is along the same lines as for the scalar ensembles.

Median, percentile and extremes. For the scalar case these descriptors are based on a (partial) ordering of the values. Scalar values are easily (and intuitively) orderable. Vectorial (and other non-scalar) values are not intuitively orderable and therefore the median value, percentile value and extreme value descriptors have no straightforward generalization to non-scalar values. We will not pursue this subject in this chapter any further.

4.1.3 Frequency distributions and histograms

Instead of representing an ensemble by enumerating all elements $v_1, \ldots, v_N$ in the ensemble, we can also enumerate the *unique* elements together with a count of how many times these unique values occur in the ensemble.

Let $w_1, \ldots, w_M$ be the unique values in the ensemble and let $c_1, \ldots, c_M$ be the counts, then the ensemble of all values is represented as:

$$\{c_1 \diamond w_1, \ldots, c_M \diamond w_M\}$$

where $c_i \diamond w_i$ denotes the c_i duplications of the value w_i. In some cases the plot of c_i as function of w_i provides a valuable overview of all values present in an ensemble and their frequency of occurrence. Consider the ensemble of the values sampled in 256×256 locations in a grey value image as depicted in figure 4.2. Such a plot is called a *histogram*.

Now imagine the following experiment. Add a little amount of noise to each sample value of the image depicted in figure 4.2. The amount of noise can be chosen such that it would not be visible; we would still *see* the same image. Chances are that now all the values (represented as floating point values) will be different. All values are therefore unique and all counts will be one. At first sight this histogram has nothing to do with the original one anymore.

[3]The eigen analysis of the covariance matrix is of course equivalent to the eigen analysis of the Hessian matrix in chapter 6.

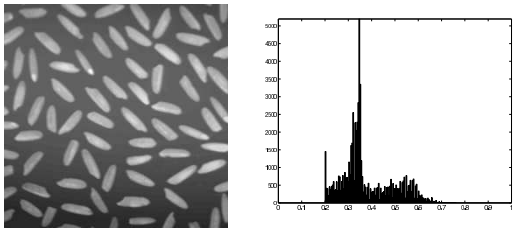


Figure 4.2: **Image histogram.** On the left an image depicting rice grains is shown. On the right the histogram of all image values is sketched.

The histogram of the original integer valued image provides us with a useful overview because the number of possible values is finite and is much smaller than the number of values in the ensemble. For the second (real valued) image the number of possible values is very large compared with the number of values in the ensemble. Due to the random noise probably no two pixels will have the same value.

In machine perception we are most often interested in ensembles of real values: the measurement results. The histogram as the map $h : w_i \mapsto c_i$ (where $\{w_i\}$ is the collection of all unique values in the ensemble) is most often a useless concept.

A standard technique to obtain meaningful histograms is to construct a map $h : w_i \mapsto c_i$ not using all the unique values in the ensemble for the w_i but to sample the range of ensemble values into a finite relatively small set of representative values. Let p_0 and p_1 be the minimum and maximum value in the ensemble. Then we construct the M sample values, for $i = 1, \ldots, M$:

$$w_i = p_0 + (i - \tfrac{1}{2})\frac{p_1 - p_0}{M}.$$

The count c_i is taken to be equal to the number of values v_j such that w_i is the 'closest' value. In fact we have made M *bins* for the range of values. The count c_i equals the number of ensemble values v_j such that:

$$v_j \in (w_i - \frac{p_1 - p_0}{2M}, w_i + \frac{p_1 - p_0}{2M}].$$

In case we want to compare histograms the total range of values should be chosen equal to the union of both ranges. For an arbitrary collection of images we might choose the range that encompasses all possible values within a particular image representation. The number of bins M is an important parameter in the construction of a histogram (see figure 4.3). A choice is hard to make without a priori knowledge concerning the distribution of the values within the ensemble.

In Matlab we can make a histogram of a scalar image $f \in \mathrm{Fun}(E, [0, 1])$ using the histogram function:

```
function [h, bc] = histogram( image, mn, mx, N )
% Construct an image histogram
h = zeros(1,N);
delta = (mx-mn)/N;
for i=1:prod(size(image))
  index = ceil( (image(i)-mn) / delta );
  index = min(N, max(1,index));
  h(index) = h(index)+1;
end
bc = mn + delta/2 + ((1:N)-1)*delta;
```

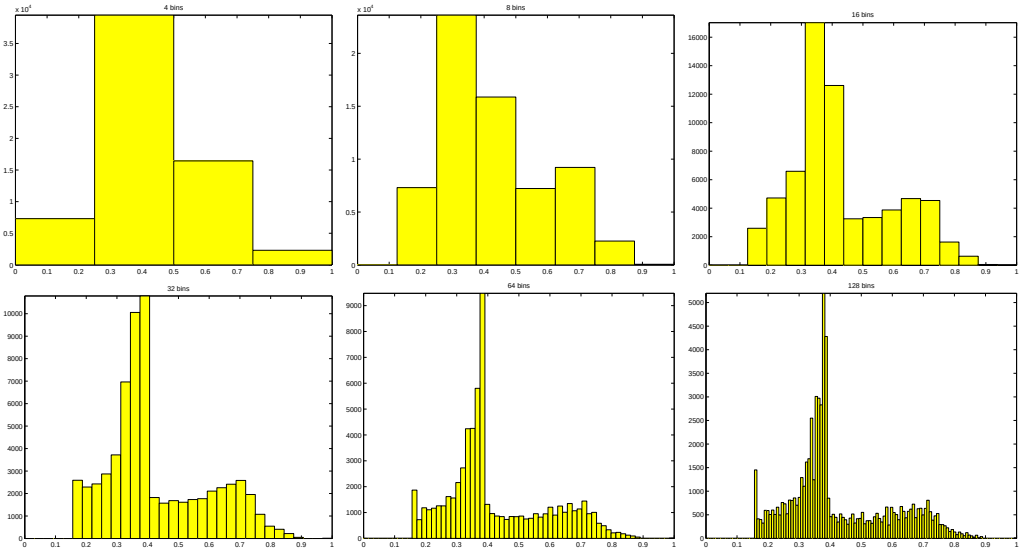


Figure 4.3: **Number of Bins.** From left to right, top to bottom the number of bins in the histogram is 4, 8, 16, 32, 64 and 128. The image of which the histograms are shown is depicted in Fig. 4.2.

This algorithm loops over all elements in the image array (note that we use the single index method that is allowed in Matlab) and for each value determines the associated bin index and increments that bin in the histogram.

There are also a built-in functions to construct histograms. The function imhist is specifically meant to calculate an histogram of all values in an image. We nevertheless do not advise to use this function as its choice for the bin centers is a bit odd. We prefer the hist function that calculates the histogram of all values in a *vector*. The call [h,v]=hist(image(:),N) calculates the same histogram as the call [h,v]=histogram(image,mn,mx,N).

Given a histogram h and vector of bin centers v a nice plot of the histogram is obtained with bar(v,h,1) (see the documentation of the bar function).

4.2 Local image statistics

Consider an image $f : \mathbb{R}^d \to \mathbb{R}$. Even on a 'small' region $\mathcal{R} \subset \mathbb{R}^d$, e.g. a small neighborhood of some location $\mathbf{a}$, the ensemble of all values is infinitely large, let alone all values in the entire image. What then is the motivation for the analysis of the discrete ensembles discussed before? We should be comfortable that what we can calculate so easily has some relation with what we would really like to reason about: image (region) statistics.

For these ensembles of image values taken from regions of the continuous image domain the statistical descriptors discussed in previous sections are integral measures (instead of summations). I.e. we have to integrate an image function $f : E \to B$ over the region $\mathcal{R}$. It can then be shown that a numerical approximation of such integral measures is compatible with the discrete ensemble analysis of previous sections.

In this section we concentrate on the simplest of the statistical descriptors that is discussed in previous sections: the mean value. For an image region we don't have a finite and countable

set of values, instead we look at all values $f(\mathbf{x})$ within an image region $\mathcal{R} \subset E$ (where E is the image domain).

The mean value μ of the image f in the region $\mathcal{R}$ is given by:

$$\mu = \frac{\int_{\mathcal{R}} f(\mathbf{x}) d\mathbf{x}}{\int_{\mathcal{R}} d\mathbf{x}}$$

For a 2D image in the (x, y)-coordinate representation we have:

$$\mu = \frac{\int\int_{(x,y)\in\mathcal{R}} f(x, y) dx dy}{\int\int_{(x,y)\in\mathcal{R}} dx dy}$$

This reduces the task of calculating statistical ensemble descriptors within an image region to the task of numerical approximations of area integrals.

The area integrals have to be approximated because only the discrete representation F of the image f is available. We assume the sampling grid used is generated by the standard basis $\mathbf{e}_1, \ldots, \mathbf{e}_d$. Consider the integral

$$I = \int_{\mathbb{R}^d} f(\mathbf{x}) d\mathbf{x}. \tag{4.5}$$

The most simple approximation based on the sample locations $\mathbf{x_k}$ (see chapter 2 for the definition of the sample locations) uses a nearest neighbor interpolation to approximate the function f. Note that on a hyper-cube region (in 2D this is a square) with lengths 1 (the standard grid vector lengths) centered on a sample location $\mathbf{x_k}$ the approximating function is constant $F(\mathbf{k})$. The hyper-volume of this hyper-cube is 1 and thus the integral over each hyper-cube is equal to the sample value. So we get for the total volume:

$$I \approx \sum_{\mathbf{k}} F(\mathbf{k}) \tag{4.6}$$

i.e. approximating area integrals(in 2D), volume integrals(in 3D) or in general d-dimensional integrals based on a nearest neighbor interpolation technique results in a simple summation of the sampled values. It might even come as a bigger surprise that using a bilinear interpolation results in exactly the same approximating summation of the area integral[4]in Eq.(4.5) (see exercise 6).

Within the nearest neighbor approximation the integral expressions for the statistical descriptors turn out to be equal to their discrete counterpart: the ensemble is just the finite number of image values in the sample locations within the region $\mathcal{R}$. This indicates that the statistical analysis of the discrete ensembles is in good approximation indicative for the statistical descriptors of the 'real' images defined over a continuous space.

4.3 Clustering

Consider the images shown in figure 4.4. On the left a grey value image of is shown with the corresponding grey value histogram. The distinction between the bright cookies and the dark background is evident, both in the image as well as in the histogram. In the histogram we can clearly distinguish two main clusters of grey values centered at grey value 0.15 and 0.6 respectively.

[4]This is only true for the approximation of the area integral taken over the entire image domain. In case only a subset of the image domain is taken as the integration region, different interpolation choices lead to different approximations.

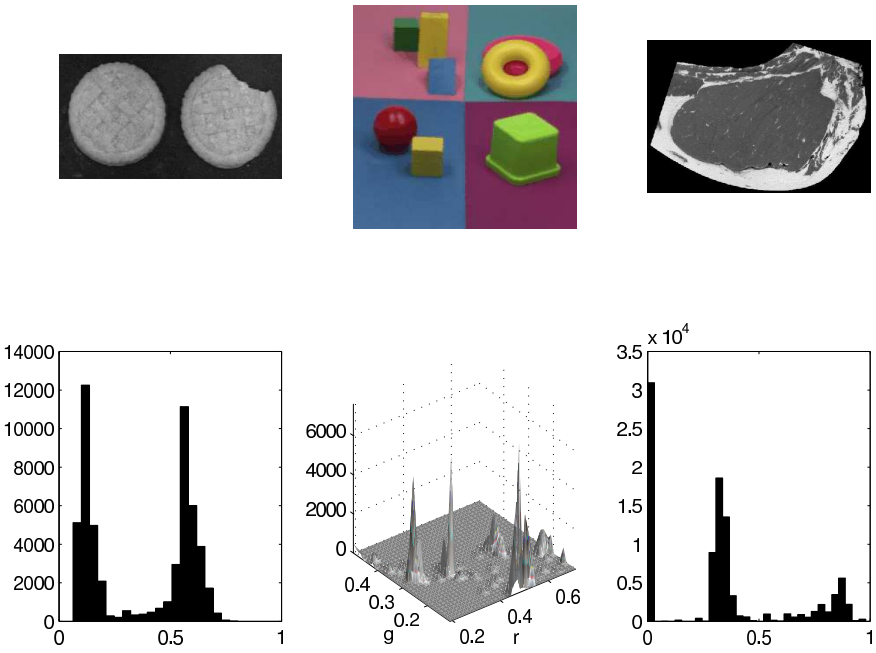


Figure 4.4: **Clusters of image values.** On the left a grey value image and the histogram of all grey values in the image. In the middle a color image with the histogram of chromaticity coordinates (rg). On the right a grey value image of beef and the corresponding histogram.

In the middle an image of colored objects is shown and the histogram of all chromaticity coordinates (normalized red $r = R/(R+G+B)$ and normalized green $g = G/(R+G+B)$). Again we see clusters of image values, this time corresponding to the most pronounced colors in the image. Normalized colors (i.e. chromaticity coordinates) are chosen such that they are invariant with respect to the shading of colors due to the geometry of the objects.

On the right, an image of beef is shown. The human eye immediately distinguishes three important gray values in the image: the black background, the grey meat and the white fat. This can also be seen in the histogram of the image values.

What all these examples have in common is that all the image values cluster around specific values. In practical applications it would be a valuable operational tool if we would have an algorithm to find those cluster centers automatically.

In this section we discuss the *k-means clustering* algorithm. This is a fairly simple clustering algorithm that is nevertheless often useful in practice. It is not within the scope of this book to get into much detail about the theory underlying k-means clustering. We only describe the workings of the algorithms and provide some intuitive explanation. It should be carefully noted that k-means clustering is just one example of clustering, for more elaborate examples we refer to the statistical pattern recognition literature.

Let $\Omega = \{\mathbf{v}_i \mid i = 1 \ldots N\}$ be an ensemble of N values from the d-dimensional real valued vector space $\mathbb{R}^d$. k-Means clustering assumes that a *distance measure* is available, i.e. given two values $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$ we may calculate a distance $d(\mathbf{a}, \mathbf{b})$. For vector values the Euclidean distance is chosen. Note that this distance becomes the absolute difference for scalar values (i.e. $d = 1$).

In k-means clustering we assume that the number of clusters k is known a priori. A cluster is characterized with a cluster center μ_i (for $i = 1, \ldots, k$). Given the cluster centers we can assign each of the values $\mathbf{v}_i$ in the ensemble to the cluster whose center is closest to the value $\mathbf{v}_i$. Let c_i denote the cluster number to which value $\mathbf{v}_i$ belongs, i.e.:

$$c_i = \arg\min_{j=1,\ldots,k} d(\mathbf{v}_i, \mu_j)$$

The classification of all the values in the ensemble can be used to update the cluster centers. The new cluster center μ_j is calculated as the mean of all values in the ensemble that are classified to belong to cluster j:

$$\mu_j = \frac{\sum_{\forall i: c_i = j} \mathbf{v}_i}{\sum_{\forall i: c_i = j} 1}$$

This should be read as: for all $i = 1, \ldots, N$ such that $c_i = j$, sum the values $\mathbf{v}_i$ and divide by the number of values in the ensemble belonging to cluster j.

Given the updated cluster centers we can reclassify the values in the ensemble to see to which cluster they are closest. Then we can recompute the cluster centers etc. etc. This process is iterated until the clustering is stable (the c_i's do not change).

For the 'cookies' image in figure 4.4.(a) the cluster centers are 0.1405 and 0.5790. For the 'beef' image in figure 4.4.(c) the cluster centers found are 0.0009, 0.3391 and 0.8206.

In case an ensemble of values is represented with a histogram, k-means clustering can be made much more efficient by working on the histogram values instead of using the individual values in the ensemble. This is left as an exercise to the reader. It is also nice to point out that the isodata thresholding algorithm from section 4.4 is equivalent to 2-means clustering using a histogram representation of the ensemble.

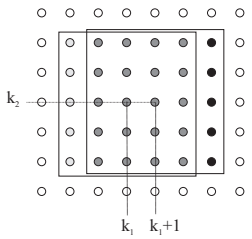


Figure 4.5: **Overlapping neighborhoods.** The neighborhood centered at location $(k_1 + 1, l_1)$ has 20 locations in common with the neighborhood at location (k_1, l_1). This overlap can be often used to devise efficient algorithms.

4.4 Image operators

In this section we look at two types of image operators that are based on the statistical descriptors that we have studied in previous sections in this chapter.

The first type of image operators is conceptually simple. Let γ be some statistical descriptor of a collection of image values and let $N(\mathbf{x}) \subset \mathbb{R}^d$ be a local neighborhood of the location $\mathbf{x}$, then we define the image operator γ_N

$$\gamma_N(f)(\mathbf{x}) = \gamma(\{f(\mathbf{y}) \mid \mathbf{y} \in N(\mathbf{x})\}).$$

I.e. at each location $\mathbf{x}$ we calculate the statistical descriptor γ for the collection of values found in $N(\mathbf{x})$. In principle we can plug into the above equation any of the statistical descriptors that we have studied in this chapter.

Note that in case we have calculated $\gamma_N(f)(k_1, k_2)$ and proceed with the next location in the sampling grid $(k_1 + 1, k_2)$ then the sets $N(k_1, k_2)$ and $N(k_1 + 1, k_2)$ most often have a lot of elements in common and thus the ensemble used to calculate $(\gamma(f))(k_1 + 1, k_2)$ has many values in common with the ensemble in location (k_1, k_2). This is illustrated in figure 4.5. Efficient algorithms to calculate statistical filters are therefore often based on clever use of these 'overlapping' neighborhoods.

Matlab contains a useful function nlfilter that can be used to implement most of the statistical image operators that are described below. As an example consider the local average filter. This operator (often called a *filter* in the image processing jargon) calculates the average of all the values in a local neighborhood. The nlfilter function takes care of enumerating all positions in an image and collecting all values in the neighborhood. Then it calls a function to calculate the new value in a point. This user supplied function takes as argument an array of all the values in the neighborhood. With this function we can calculate the average in every $N \times N$ neighborhood in the image.

```
function r = avgOp( image, N )
r = nlfilter( image, [N N], @average );

function r = average( x )
r = mean( x(:) );
```

The nlfilter function has the disadvantage that it only implements the 'constant value border' to deal with the border problem. In the average filter the constant value of the border shows itself immediately. A second disadvantage of the nlfilter function is that it is slow. For your own health you are advised to run examples only with 128×128 images (or even smaller).

Let us look at some more local neighborhood filters:

Local Average Filter. The local average filter calculates the average image value for all locations in the neighborhood of a location. The uniform convolution and the Gaussian convolution are examples.

For the local average filter (a convolution) Matlab (as all other image processing programs) has an optimized algorithm available. Compare the performance of the above avgOp(image,N) with imfilter (image,ones(N)/(N^2)) function (as a function of N).

Local Variance Filter. The local variance filter measures the variance ν in every image location for all values in the neighborhood $N(\mathbf{x})$. A large local variance in a location in the image is an indication that 'something' is going on there. Such a simple statistical descriptor as the variance however provides no clue as to what is happening there.

The simplest way to implement the local variance filter is like:

```
function r = varOp( image, N )
r = nlfilter( image, [N N], @variance );

function r = variance( x )
r = var( x(:) );
```

A more efficient way to implement the variance filter is to rewrite the definition of the variance. This leads to the following expression:

$$\begin{aligned}
\nu(\Omega) &= \frac{1}{N}\sum_{i=1}^{N}(v_i-\mu)^2 . \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(v_i^2 - 2\mu v_i + \mu^2\right) \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(v_i^2\right) - \frac{2}{N}\mu\sum_{i=1}^{N}v_i + \frac{1}{N}\mu^2\sum_{i=1}^{N}1 \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(v_i^2\right) - 2\mu^2 + \mu^2 \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(v_i^2\right) - \mu^2
\end{aligned}$$

This shows that the variance is equal to the 'mean of the squares' minus the 'square of the mean'. This observation can be used to implement the variance filter with two convolutions (imfilter).

Median and Percentile Filter. Instead of calculating the mean value the median value is calculated in the local neighborhood of every location. A lot of literature is devoted to these *rank order filters*.

Local Minimum and Maximum Filters. In case the ensemble extremes are calculated within a local neighborhood of each of the locations in an image we enter the realm of *mathematical morphology*. This important branch of both practice and theory of image processing and image analysis owes its name to the very elegant and powerful *geometrical interpretation* of the resulting image operators.

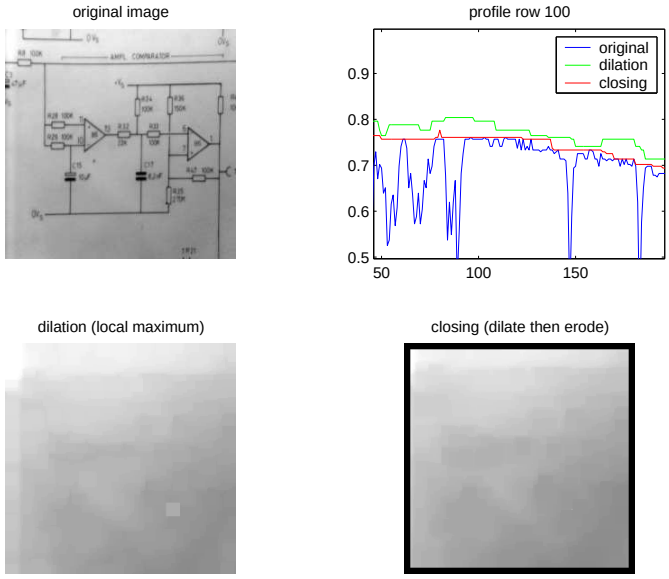


Figure 4.6: **Background estimation using Mathematical Morphology.** From left to right, top to bottom: (a) original image with non-constant illumination. In (c) the dilation (local maximum filter) using a 15×15 neighborhood, In (d) the closing result (first doing a dilation followed by an erosion.

Consider the image in figure 4.6.a showing a drawing with black ink on (not so) white paper. In figure (b) the grey values along a horizontal line through the image domain are sketched. From this one dimensional function but also when just looking at the image it is clear that the paper is much whiter in the top left corner then it is in the bottom right corner.

Now consider the neighborhood N chosen to be large enough that no matter where we locate it in the image the maximal grey value in the neighborhood is from the white paper. The local maximum filter then completely 'wipes' out the black drawing as can be seen in figure (c). If we would take the result of the local maximum filter and apply a local minimum filter to it using the same definition for the local neighborhood an image is obtained that is an approximation of the white paper without the drawing on it. The effect of the local minimum filter is to have grey values that are again comparable in magnitude with the original grey values in the white regions. See figure (d).

In mathematical morphology a local maximum filter is called a dilation and local minimum filter is called an erosion. The intuitive explanation of the working of the combination of local maximum filter (a dilation) followed by a local minimum filter (an erosion) is grounded in a geometrical interpretation of these morphological image operators.

The second type of image operators are the monadic point operators which are based on some statistical analysis of all the image values in the original image. The ensemble of values now consists of all values in the image. The ensemble is completely characterized with its histogram and this then will be the starting point in defining the operators.We assume that the images are scalar images with values in the range $[0, 1]$, i.e. $f \in \mathrm{Fun}(E, \mathbb{R})$.

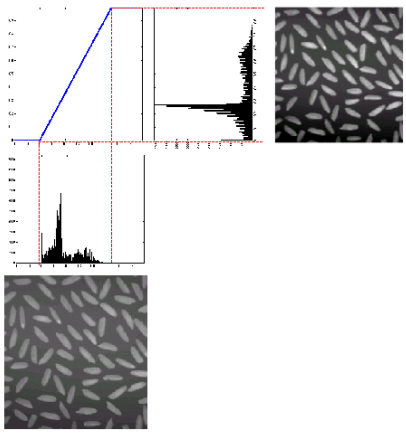


Figure 4.7: **Contrast Stretching.** The image in the lower left is the original image. Just above it the histogram of this image is sketched. The top-left diagram shows the ψ function that maps the input range of values to the output values. The histogram of the resulting image is shown in the top-middle diagram. Note that the histogram is rotated. The independent value (the grey value) runs from bottom to top, the counts run from right to left. The resulting image is shown in the top-right. (In print, the effect is not very dramatic, since the printer tends to extend the dynamic range anyway, to compensate for the printing process.)

Contrast stretching. Let $p_0(f)$ and $p_1(f)$ be the minimum value and maximum value in the image f respectively, then the monadic operator $\psi : \mathbb{R} \mapsto \mathbb{R}$:

$$\psi(v) = \frac{v - p_0(f)}{p_1(f) - p_0(f)}$$

results in an image operator $\psi : \mathrm{Fun}(E, \mathbb{R}) \mapsto \mathrm{Fun}(E, \mathbb{R})$:

$$(\psi f)(\mathbf{x}) = \psi(f(\mathbf{x}))$$

Note that (again) we have overloaded an operator to work on entire images given a function working on image values.

In Fig. 4.7 the operator ψ and its relation to the histogram of ψf given the histogram of f is sketched.

Histogram equalization. Let $f \in \mathrm{Fun}(E, [0, 1])$ be an image with histogram h_f. Let ϕ be a mapping $[0, 1] \to [0, 1]$ then we can lift this value operator to work on images $\phi(f)(x) = \phi(f(x))$. We can construct an order preserving map ϕ such that the histogram h_g of the image $g = \phi(f)$ is constant: $\forall v \in [0, 1] : h_g(v) = 1$. The map that accomplishes this task is the cumulative histogram of the original image:

$$\phi(v) = H_f(v) = \int_{w=0}^{v} h_f(w)dw$$

The proof that the above map indeed results in an image with a constant histogram is easy. Order preservingness of the map ϕ implies that $v_1 \leq v_2$ then also $\phi(v_1) \leq \phi(v_2)$. If we now fix v_2 then if we consider all $v_1 \leq v_2$ it is clear that the number of values with value less

than or equal to v_2 in the original image f should be equal to the number of values in the resultant image with value less than or equal to $\phi(v_2)$. This means: $H_f(v) = H_g(\phi(v))$. Because $h_g(v) = 1$ we have $H_g(\phi(v)) = \phi(v)$ and thus: $\phi(v) = H_f(v)$.

For sampled images this operation can only be done approximately due to the fact that the histogram has to be calculated using a finite number of bins.

In the image processing literature histogram equalization is also often called *histogram normalization*.

Let a be the Matlab array representing a sampled image with grey values in the range from 0 to 1. The histogram of this image is calculated using N bins as [h,v]=histogram(a,0,1,N). The cumulative histogram is calculated as H = cumsum(h). Normalizing the cumulative histogram: H = H/H(end). H now is a sampled representation of the required map ϕ. Evidently we need some kind of interpolation technique to use the sampled representation of the map ϕ in order to process all the values in the image. In Matlab this is very easy using the built-in interp1 interpolation function: b=interp1(v,H,a) results in an approximation of the normalized image.

Histogram equalization is an important image processing operation in practice for the following reason. Consider two images f_1 and f_2 of the same object but taken under two different illumination conditions (say one image taken on a bright and sunny day and the other image taken on a cloudy day). The difference between these images can be approximated with an order preserving mapping γ, i.e. $f_2 = \gamma(f_1)$.

It is not hard to prove that the histogram equalized versions of both images are equal: $\phi_1(f_1) = \phi_2(f_2)$. Here ϕ_1 and ϕ_2 are the histogram equalizing mappings defined above (these mappings are image dependent, that is why we write the subscript to denote the image).

Histogram equalization thus serves as a preprocessing step in vision systems to compensate for the effect of an unknown change in illumination (grey value scaling). For instance in a face recognition system where images of human faces have to be compared with other images of the same face (and most probably taken under different illumination conditions).

Image Thresholding. Consider the 'cookies' image in figure 4.8.(a) and the histogram of this image in (b). The histogram is in accordance with our intuition that there are basically two important grey values in the image: the black background and the bright cookies. Thresholding the image at the grey value in between the two peaks in the histogram will result in an image where the points in the cookies will be 1 and all others will be 0 (i.e. an identification image). An automatic selection of the threshold is given by the *isodata threshold algorithm*.

The threshold t is defined to be that value such that the mean m_1 of all grey values lower than or equal to t and the mean m_2 of all grey values larger than t are symmetrically placed on both sides of t (see figure 4.8.(b)), i.e.:

$$t - m_1(t) = m_2(t) - t$$

or equivalently:

$$t = \frac{1}{2}(m_1(t) + m_2(t)) = M(t).$$

This is only an implicit definition of the threshold value t as it appears on both sides of the above expression. An efficient method to find the value of t such that $t = M(t)$ is based

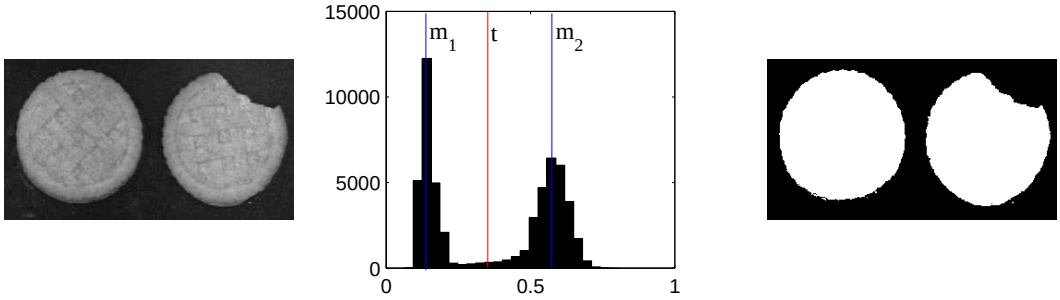


Figure 4.8: **Isodata Thresholding.** From left to right: (a) original image, (b) histogram and threshold t and mean values m_1 and m_2 and (c) thresholded image.

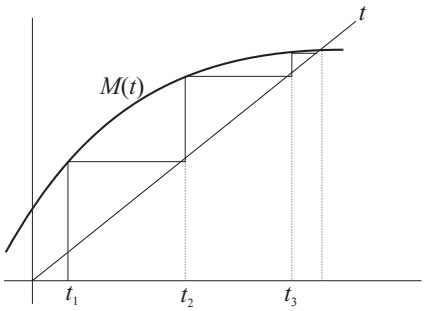


Figure 4.9: **Fixed Point Iteration.** Numerical solution of $t = M(t)$ by functional iteration $t_{i+1} = M(t_i)$.

on what is called *functional iteration* also known as *fixed point iteration* and is illustrated in figure 4.9. Let t_0 be some random initialization and consider the recursive definition:

$$t_{i+1} = M(t_i).$$

For the function M as defined above this process indeed converges (it is not necessarily true for arbitrary functions).

Below the algorithm for isodata threshold calculation is presented in Matlab. This algorithm uses the ensemble of all image values.

```
function t = isodata( im )
% isodata threshold calculation
% this algorithm does the calculation
% using the ensemble of all image values
%
% the image 'im' needs to have values
% in the interval [0,1]
%
% a more efficient algorithm would use
% an histogram (approximative) representation
% of the ensemble of image values
t = 0.5; tprev = 0;
while abs(t-tprev)>0.001
  tprev = t;
  t = (mean( im( im<=t ) ) + mean( im(im>t) ) )/2;
end
```

Most often in practice an algorithm is used that uses an histogram representation of the ensemble of image values. Only for discrete images where there are as many bins in the histogram as there are unique grey values in the image the isodata algorithm using histograms will give the same results as the algorithm presented above (the approximation obtained using a histogram based algorithm is in practice most often good enough).

4.5 Exercises

1. **Mean value.**
   Prove that the definition of the mean as the value that minimizes the sum of the squared differences (see Eq.(4.3)) is equivalent to the well-known definition of the mean value (eq. 4.1). Hint: the minimum is reached where the derivative with respect to m is equal to zero.

2. **Mean versus median value.**
   Consider the following ensemble A of integer values (represented with its histogram)

v	0	1	2	3	4	5	6	7	8	9	10
$h_A(v)$	1	2	4	5	7	8	9	6	5	2	1

   Calculate both the mean and the median value for this ensemble. Also draw the histogram. Do the same for the following ensemble B

v	0	1	2	3	4	5	6	7	8	9	10	31
$h_B(v)$	1	2	4	5	7	7	9	6	5	2	1	1

In this case we assume that the ensembles are repetitive measurements of the same phenomenon and that deviations from the true value are due to noise (either inherently to the phenomenon being observed or due to the measurement device).

Within this interpretation there is good reason to call the value 31 in ensemble B an *outlier*. In the presence of outliers, which of the measures, mean or median, is a better estimate of the true value?

3. **Variance.**
Show that the variance ν of an ensemble $\{v_i\}_{i=1,\ldots,N}$ is equal to:

$$\nu = \text{mean}(v_i^2) - (\text{mean}(v_i))^2$$

where $\text{mean}(v_i^2)$ is the mean of the squares of the values in the ensemble.

4. **Alpha trimmed mean.**
There is a great freedom in defining statistical ensemble descriptors and indeed much of the image processing literature is devoted to the introduction of 'yet-another-local-image-descriptor'. A well-known one is the *alpha trimmed mean*: calculate the mean of all values in the ensemble except for the $\alpha\%$ largest values and the $\alpha\%$ smallest values. The alpha trimmed mean is designed to be a balance between the classical mean and median. What are the values of α for which the alpha trimmed mean behaves like either the mean or the median.

5. **Variance for vector ensembles.**
Show that $\nu_d(R\Omega) = \nu_d(\Omega)$ where we take $R\Omega$ to be the ensemble $R\Omega = \left\{ R\mathbf{v}_i \;\middle|\; i = 1, \ldots, N \right\}$ where R is a rotation matrix. The 'variance' measure ν_d is defined in equation 4.4.

6. **Bilinear interpolation.**
Show that an approximation of

$$I = \int_{\mathbb{R}^d} f(\mathbf{x}) d\mathbf{x}.$$

based on a bilinear interpolation with f sampled on the standard square 2D grid results in:

$$I \approx \sum_{\mathbf{k}} F_{\mathbf{k}}.$$

That is for approximating the above integral in the real 2D space, nearest neighbor interpolation is just as good as bilinear interpolation.

7. **Overlapping neighborhoods.**
Consider a median filter where at each location in the input image F we consider the 25 values in the 5×5-neighborhood and take the 'middle value'. That is, the 25 values are sorted and in the sorted sequence we take the 13th value.

   (a) What sorting algorithm would you use? And what will be the resulting computational complexity of the image operator (order of the number of values in the local neighborhood).

   (b) Can you make use of the fact that the neighborhoods of neighbor locations have a large overlap?

   (c) A well-known median filter algorithm first makes a histogram of the values in the local histogram and from the histogram calculates the median value. The advantage of this approach is that the histogram can be easily updated when moving from a location in the image to the next. Can you think of a way to do this?

Chapter 5

Color Vision

Color is a prime example of machine perception that cannot be properly understood without knowing in what way the human brain perceives color. Color cannot be defined based on physical properties of the stimulus alone; the way in which the human visual system observes the electromagnetic energy (what we call light) is important as well.

In this chapter we will discuss color from a colorimetric point of view. Colorimetry is the science that is aware of the need for a psychophysical description of color but aims at reducing the role of the human observer to being a judge about *color equivalence*. Telling whether two colors are the same or not (in a controlled experimental setting) is a task that is remarkably consistently performed over many different observers with great reproducibility.

Color is an important visual clue in the computer sciences nowadays. Using color in either 'color generating' applications (computer graphics and visualization) as well its use in 'color analysis' applications (computer vision and image processing) can only be based on an understanding of the basics of color science. This chapter aims at providing that basic background material.

5.1 Color Science

A ray of light that hits the photosensitive cells in the retina is physically characterized by its spectral distribution t as a function of the wavelength λ, i.e. $t(\lambda)d\lambda$ is the energy in the beam due to the light with wavelengths in the interval from λ to $\lambda + d\lambda$. In colorimetry, the science of color measurements, we study *beam colors*, i.e. the colors associated with beams of light. This in contrast with *object colors*, the colors associated with objects that reflect the light from an illumination source. When a red apple is illuminated with a beam of white light, the apple only reflects the red components of the light, this makes the apple appear to be red. Illuminating the apple with light containing no red light makes the apple appear black; there is no red light to reflect. Choosing different colored beams to illuminate the 'red' apple its perceived color can change dramatically. Nevertheless we talk about the 'red' apple with good reason. The material properties of the apple make it appear red to the eye when illuminated by white light.

5.1.1 The physics of color

What we call light is electromagnetic radiation with wavelengths in a small part of the entire range; the photo receptors in the retina are sensible to the electromagnetic waves with wavelength in the interval from about 400 to 700 nm only (see Fig. 5.1).

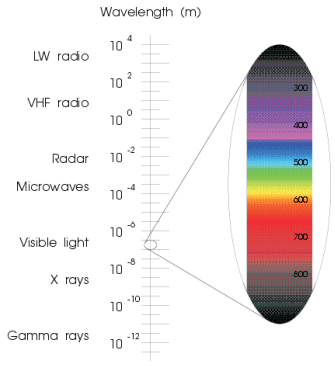


Figure 5.1: **Electromagnetic waves.** Light is only a very small part of the electromagnetic range.

It is the spectral distribution function or *spectrum* of a ray of light that completely characterizes a beam (and thus its color) within the colorimetric context. Because the spectrum t as function of the wavelength λ is a function over the real numbers, the 'space' of all spectra is infinite dimensional.

In this chapter we follow the standard colorimetric practice to represent a spectrum (and any other function of λ) with its sampled version. Let f be a function of λ then we represent f with its sampled version F, where $F(k) = f(\lambda_k)$. Most often the sampling points are taken 5 nm (or 10 nm) apart.

For color spectra it is very advantageous to represent the collection of samples $F(k)$ with a *vector* $\mathbf{f}$ with elements $f_k = F(k)$. We assume that spectral 'vectors' are in a N-dimensional space. With this representation of spectra (and colors to be introduced shortly) as vectors we can utilize the mathematical tools from linear algebra[1].

5.1.2 The perception of color

Within the retina of the human eye two basic classes of photo receptors can be found: the *rods* and *cones*. The rods are used in low light level conditions. The rods are not capable of distinguishing different colors; only light dark differences are detectable.

The second class of photo receptors are the cones. These are concentrated in the center of the visual field (the fovea). There are three types of cones. These contain photosensitive pigments with different spectral absorptances. Let t be the spectral distribution of the beam incident on the retina, the response of the i-th type of cone with sensitivity s_i is given as:

$$c_i = \int s_i(\lambda)t(\lambda)d\lambda.$$

In Fig. 5.2 the three sensitivity functions are shown. They are also known as the S, M and L sensitivity curves as they roughly correspond with the short, medium and long wavelength electromagnetic waves.

[1] Using *linear* algebra is of course only a good choice for modeling a physical phenomenon in case the phenomenon behaves in a linear way. This is indeed the case for color measurement in the human eye as will become clear in this chapter.

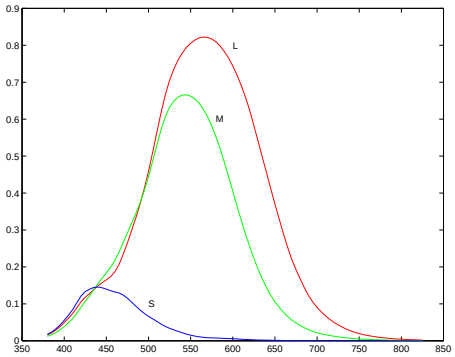


Figure 5.2: **Cone sensitivity curves.** The sensitivity curves for the three cone types are shown. The nomenclature S(hort), M(edium) and L(ong) is indicative of the ordering of the peak sensitivities of the cones.

Within the vector representation of spectra and sensitivity functions an approximation of the above integral expression becomes:

$$c_i = \mathbf{s}_i \cdot \mathbf{t}$$

i.e. the inner product of the sensitivity and the spectrum[2]. The inner product can be written as the following operator: $c_i = \mathbf{s}^\mathsf{T}\mathbf{t}$. Combining the three responses c_i for $i = 1, 2, 3$ in one *tristimulus response vector* $\mathbf{c}$ we get:

$$\mathbf{c} = S^\mathsf{T}\mathbf{t}$$

where S is the *sensitivity matrix* whose columns are the 3 sensitivity vectors $S = (\mathbf{s}_1\, \mathbf{s}_2\, \mathbf{s}_3)$. The matrix S^T is the representation of a mapping from the N-dimensional *spectral space* onto the 3D tristimulus vector space.

The tristimulus vector $\mathbf{c}$ is the input to the brain for any beam of uniform colored light that hits the retina. Thus two beams with spectra $\mathbf{t}_1$ and $\mathbf{t}_2$ appear to be equally colored in case

$$S^\mathsf{T}\mathbf{t}_1 = S^\mathsf{T}\mathbf{t}_2$$

Carefully note that this doesn't mean that the two spectra are equal. The mapping S^T from a high (N) dimensional space onto a 3 dimensional space does not have an inverse and therefore $S^\mathsf{T}\mathbf{t}_1 = S^\mathsf{T}\mathbf{t}_2$ does not necessarily imply that $\mathbf{t}_1 = \mathbf{t}_2$. Spectra that give rise to the same color tristimulus values are called *metameric spectra*. Metamerism is both a cause of a lot of difficulties in color science but also of great practical importance.

5.1.3 Chromaticity diagrams

It is an *empirical fact* that color response is linear. I.e. when we mix two beams of light with spectra $\mathbf{t}_1$ and $\mathbf{t}_2$ then the mixed beam has a spectrum $\mathbf{t}_1 + \mathbf{t}_2$ and the color response is indeed $\mathbf{c}_1 + \mathbf{c}_2$. This is also in accordance with the linear model $\mathbf{c} = S^\mathsf{T}\mathbf{t}$ for we have $S^\mathsf{T}(\mathbf{t}_1 + \mathbf{t}_2) = S^\mathsf{T}\mathbf{t}_1 + S^\mathsf{T}\mathbf{t}_2 = \mathbf{c_1} + \mathbf{c_2}$, but this is obviously not a proof of the linearity of the photo-receptors in the retina. It is just an indication that we have chosen an appropriate model.

A second empirical fact is that the color of a beam does not perceptually change when its intensity is changed. Thus a beam with spectrum $\mathbf{t}$ and a beam with spectrum $a\mathbf{t}$ only differ in perceived brightness, not in color.

[2]It should be noted that the integral expression for the cone response is also an inner product in the Hilbert vector space of spectral functions.

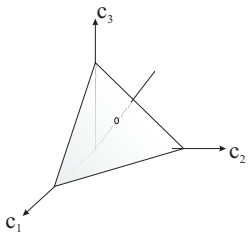


Figure 5.3: **Chromaticity plane.** All the color tristimulus responses on the half line from the origin are perceived as the same color, only the intensities are different. Therefore a color (disregarding its intensity) can be uniquely characterized as the intersection of the half line with a chromaticity plane.

Consider a half line emanating from the origin in the tristimulus space. All points on this line can be written as $a\mathbf{c}_0$ where $\mathbf{c}_0$ is a point on this line. All the points on the line have the same color, only the perceived brightness is different. Therefore the color may be characterized uniquely with the intersection of the half line with a plane. In Fig. 5.3 we have sketched the plane $c_1 + c_2 + c_3 = 1$ as an example of such a *chromaticity plane*. Please note that the chromaticity plane is not necessarily a plane of equal intensity beams!

We don't need three coordinates to identify a color in the chromaticity plane. Just two coordinates, say c_1 and c_2 are enough (why?). The plot of colors in a two dimensional view of the chromaticity plane is called a *chromaticity diagram*.

There is nothing special or magical about chromaticity diagrams. Instead of defining a chromaticity plane we could equally well take any parameterizable surface such that all half lines intersect the surface only once. The disadvantage of such a choice is that additive color mixtures are not easily represented as linear mixtures in the chromaticity plane anymore.

5.1.4 Color matching experiments

Consider again the *color match* of the two beams $\mathbf{t}_1$ and $\mathbf{t}_2$, i.e. $S^\mathsf{T}\mathbf{t}_1 = S'\mathbf{t}_2$. Let A be any non-singular 3×3 matrix then $S^\mathsf{T}\mathbf{t}_1 = S^\mathsf{T}\mathbf{t}_2 \Leftrightarrow A^\mathsf{T}S^\mathsf{T}\mathbf{t}_1 = A^\mathsf{T}S^\mathsf{T}\mathbf{t}_2$ (see Exercise 3). This shows that for color matching experiments *any linear combination of the sensitivity curves of the human eye will lead to the same color matches.* Any matrix $C = SA$ is called a color matching matrix.

In fact color matching matrices were the basis of colorimetry long before the human sensitivity matrix could be measured directly. Understanding color matching experiments is therefore crucial in understanding the international standards in color representation.

In a color matching experiment (see Fig. 5.4) an observer looks at a circular field composed of two halves. From one half of the field the arbitrary beam with spectrum $\mathbf{t}$ is visible and from the other half the mixture with spectrum $\mathbf{u}$ of three fixed lights (the primaries) is visible. Let the spectra from the three primaries at full intensity be $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$. The three primaries should be *colorimetric independent*, i.e. the associated tristimulus vectors $S^\mathsf{T}\mathbf{p}_1$, $S^\mathsf{T}\mathbf{p}_2$ and $S^\mathsf{T}\mathbf{p}_3$ should be linearly independent.

The observer can control the attenuation factor a_i of each of the primaries. This results in an observed spectrum $\mathbf{u} = a_1\mathbf{p}_1 + a_2\mathbf{p}_2 + a_3\mathbf{p}_3$. The observer is asked to control the attenuation factors such that the color in both halves of the circular field appear to be equal. Such a color match requires that:

$$\begin{aligned} S^\mathsf{T}\mathbf{t} &= S^\mathsf{T}\mathbf{u} \\ &= S^\mathsf{T}(a_1\mathbf{p}_1 + a_2\mathbf{p}_2 + a_3\mathbf{p}_3). \end{aligned}$$

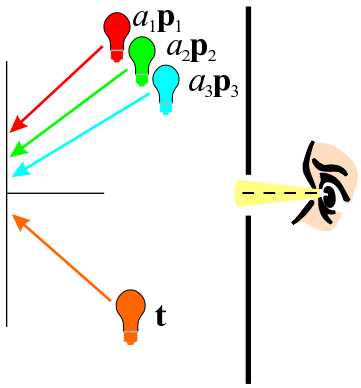


Figure 5.4: **Color matching.** In a color matching experiment the observer sees a (circular) field of two halves. In one half the test light (spectrum $\mathbf{t}$) is shown. In the other half a mixture of three primaries is shown. The observer is asked to set the attenuation of the three primary beams to obtain a perceptual match of the two colors.

Collecting the three primaries in one matrix $P = (\mathbf{p}_1\, \mathbf{p}_2\, \mathbf{p}_3)$ and writing the attenuation factors in a vector $\mathbf{a} = (a_1\, a_2\, a_3)'$, the color match equation can be written as:

$$S^\mathsf{T}\mathbf{t} = S^\mathsf{T}P\mathbf{a}.$$

Because $S^\mathsf{T}P$ is invertible (see Exercise 4) we get:

$$\mathbf{a} = (S^\mathsf{T}P)^{-1}S^\mathsf{T}\mathbf{t} \tag{5.1}$$

showing that *for any spectrum* $\mathbf{t}$ *we can find the attenuation factors* $\mathbf{a}$ *to construct a color match using any three (colorimetric independent) primaries.*

Some test lights to be matched lead to negative attenuation factors according to Eq.(5.1). Negative attenuation factors are of course a physical impossibility. The color matching experiment is not lost though. If we shift the primary beam with the negative factor to the side of the unknown light source we have a positive factor again. The observed colors will change because we add a second source to the test beam. A color matching experiment is concerned with color equivalence and not with color appearance.

Note that the matrix $C^\mathsf{T} = (S^\mathsf{T}P)^{-1}S^\mathsf{T}$ needed to calculate $\mathbf{a}$ given the spectrum $\mathbf{t}$ is of the form AS^T where A is a 3×3 non-singular matrix. It has already been argued that such a matrix leads to equivalent color matches as the human sensitivity matrix.

The solution for the attenuation factors $\mathbf{a}$ to match a color beam $\mathbf{t}$ in the form given in Eq.(5.1) is dependent on the human sensitivity matrix S, for we have $\mathbf{a} = C^\mathsf{T}\mathbf{t} = (S^\mathsf{T}P)^{-1}S^\mathsf{T}\mathbf{t}$. The matrix $C^\mathsf{T} = (S^\mathsf{T}P)^{-1}S^\mathsf{T}$, however, can be measured directly without knowing S explicitly. The human observer ('who has the matrix S in his brain') is needed to judge color equivalence.

Let $\mathbf{t} = \mathbf{\Delta}_i$ be the spectrum of a *monochromatic* beam of wavelength λ_i (these monochromatic beams *can* be realized in practice). The monochromatic beam $\mathbf{\Delta}_i$ has a unit vector

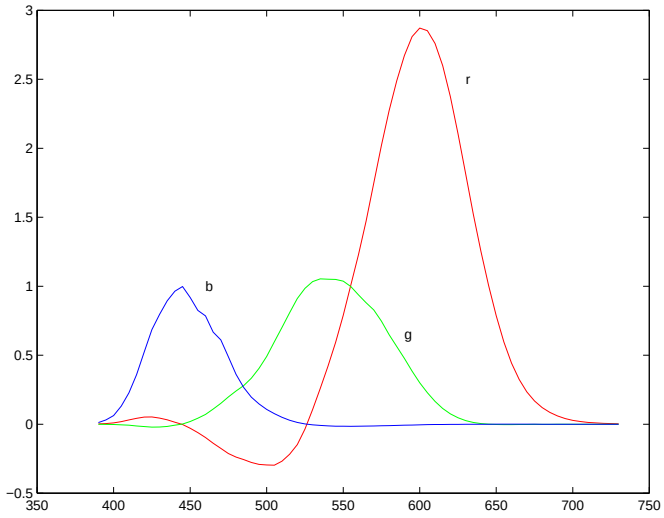


Figure 5.5: **Color matching functions.** In this experiment monochromatic beams of wavelengths 435.8, 546.1 and 700 nm are used.

representation:

$$\mathbf{\Delta}_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

where the 1 element is at index i in the vector.

Then $\mathbf{a} = C^{\mathsf{T}}\mathbf{t}$ is the i-th row of the color matching matrix C. Thus by repeating the color matching experiment for all monochromatic beams the entire color matching matrix C can be determined experimentally. The color matching matrix is dependent on the chosen primary beams.

A very well known color matching experiment has been done using three monochromatic beams (wavelengths 435.8, 546.1 and 700 nm) resulting in the color matching matrix $C_{\bar{r}\bar{g}\bar{b}}$ whose columns are the functions $\bar{r}$, $\bar{g}$ and $\bar{b}$ as sketched in Fig. 5.5. Each curve plots the intensity of the primary beam as a function of the wavelength of the monochromatic beam to be matched. Note that over a large range of wavelengths the relative intensity of the 'red' beam is negative indicating that a color match was only possible in case the red beam was shifted to the side of the monochromatic beam to be matched.

5.1.5 The XYZ color model

The CIE (Committee Internationale de l'Eclairage) in 1931 adopted the color matching experiment with the three monochromatic beams resulting in the three matching functions in Fig. 5.5. However they decided that negative values in the color matching matrix was a bad choice. A derived color matching matrix $C_{\bar{x}\bar{y}\bar{z}}$ was defined such that:

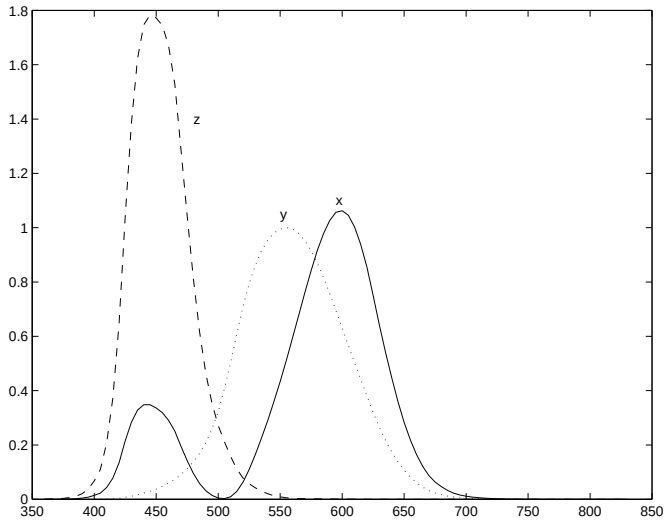


Figure 5.6: **The CIE 1931 color matching functions.**

- it contains only positive values (calculations with negative values were error prone in the pre-computer era), and
- the Y-response is approximately equal to the perceived brightness of the beam (see Fig. 5.6).

Because the perceived brightness of a beam is a subjective qualification lacking scientific rigor, there is not much in favor for the choice made in 1931. Unfortunately we are stuck with it. It has served us as a standard for long and will probably continue to do so in the future.

The one thing that makes the $C_{\bar{x}\bar{y}\bar{z}}$ color matching matrix a troublesome one, especially for scientists new to the field (and for their teachers), is that there are no physically realizable primary beams that will directly lead to the $C_{\bar{x}\bar{y}\bar{z}}$ color with a color matching experiment as described in the previous section. One can therefore often read that the XYZ color model corresponds with 'imaginary primaries'. There is of course nothing magical, let alone imaginary, about the XYZ color model, the CIE just played with the 'mathematical freedom' to select an arbitrary non-singular matrix A to obtain an equivalent color matching matrix CA from a color matching matrix C that followed from a real physical color matching experiment (with 'real' primaries).

Given a spectrum $\mathbf{t}$, the corresponding color specified in the XYZ-color system is:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = C^{\mathsf{T}}_{\bar{x}\bar{y}\bar{z}}\mathbf{t}$$

In Section 5.1.3 the notion of a chromaticity plane and chromaticity diagram were explained. A well known chromaticity plane in the XYZ-space, is the one defined by:

$$X + Y + Z = 1.$$

The corresponding chromaticity coordinates are:

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad z = \frac{Z}{X+Y+Z}. \tag{5.2}$$

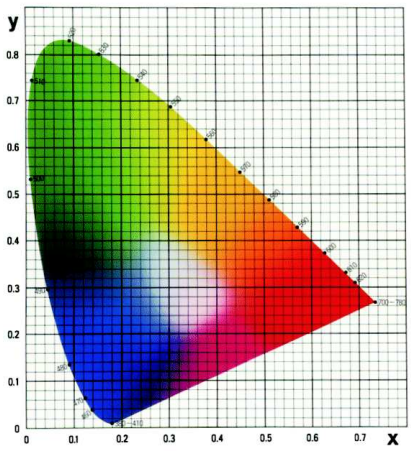


Figure 5.7: **The xy-chromaticity diagram.**

Obviously any two of these coordinates are sufficient for the description of color (disregarding the luminance). The standard plot is the xy-chromaticity diagram (see Fig. 5.7).

Any spectrum is a positive linear combination of all monochromatic beams. This implies that all colors in the xy-plane are within the convex hull of all monochromatic colors. Plotting the (x, y)-coordinates of all *positive* monochromatic spectra leads to the well-known horse shoe shaped locus in the diagram (see Exercise 9). All colors thus lie within the convex hull of the horse shoe shape. The gap between the two extremes of the visible spectrum is called the purple gap (note that purple is not the color of any monochromatic beam: you won't find the color purple in the rainbow.).

The xy-chromaticity diagram is of great help to understand some notions and concepts in colorimetry:

The color white. There is a common saying that black and white are no colors. Black is just the absence of any light whatsoever. But the notion of white is a troublesome one. In fact the color 'white' is a color like any other. There are unfortunately many white colors defined. The light of the sun is often called white, so is the color of the D65 illuminant (another CIE standard). And many more 'whites' are known. Although it is a subjective choice, there are many notions in color science that refer to the color white (therefore the white color has to be specified in quantitative colorimetric applications[3]).

Color mixing in the xy-diagram Additive color mixing is easily described in the xy-chromaticity diagram. Given a color with coordinates (x_1, y_1) and a second color with coordinates (x_2, y_2) then the additive mixture is given as $(x_1 + x_2, y_1 + y_2)$.

Complementary colors. Two colors c_1 and c_2 are called complementary in case they mix to form white. In the xy-diagram they lie on a line through the white point on opposite sides.

Dominant wavelength. For a color c, its dominant wavelength is the wavelength of the monochromatic beam c_D, on the horse shoe curve on the line from the white point through the

[3]For a computer CRT monitor the white point is specified in the xy chromaticity diagram.

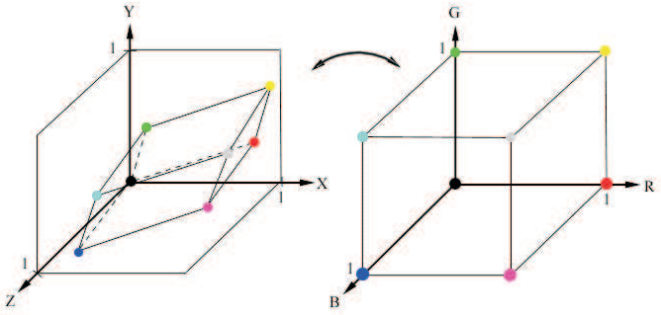


Figure 5.8: **The RGB cube as a subspace of the XYZ space.**

color c (see Exercise 7).

5.1.6 Color reproduction

An important practical consequence of the color matching experiment is that many colors can be 'made' using just three arbitrary (but colorimetric independent) primary color beams. This is exactly the way a CRT (cathode ray tube) monitor operates.

A color CRT (cathode ray tube) monitor works with three independent electron beams (these are not visible to the human eye) directed at the surface of the monitor. That surface is coated with phosphors that emit light when hit by an electron beam. A CRT uses three types of phosphors, one emitting red light, one emitting green light and the third emitting blue light.

Let $\mathbf{r}$, $\mathbf{g}$ and $\mathbf{b}$ be the spectra of the three phosphors of a color monitor. Each of the three beams is attenuated with factors R, G and B. The resulting spectrum of the mixed beam is:

$$\mathbf{t} = R\mathbf{r} + G\mathbf{g} + B\mathbf{b} = \begin{pmatrix} \mathbf{r} & \mathbf{g} & \mathbf{b} \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} = P_{\text{RGB}} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Let $\begin{pmatrix} X & Y & Z \end{pmatrix}$ be a color that we would like to reproduce on the monitor, in that case we should have:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = C^{\mathsf{T}}_{\bar{x}\bar{y}\bar{z}} P_{\text{RGB}} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

leading to the attenuation factors:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = (C^{\mathsf{T}}_{\bar{x}\bar{y}\bar{z}} P_{\text{RGB}})^{-1} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}.$$

In Fig. 5.8 the XYZ color space is sketched on the left, with the three primary colors indicated. The RGB 'cube', visualized within the XYZ space, is not a orthonormal cube of course. The RGB color cube is sketched in Fig. 5.9 indicating the different colors.

Note that only positive factors $R, G, B \in [0, 1]$ are allowed. This means that not all colors can be reproduced, only the colors that are the convex combinations of the three primary beams can be generated. When plotting the colors of the three primaries in the chromaticity diagram only the colors within the resulting triangle can be reproduced. This is called the *gamut* of the color reproduction device.

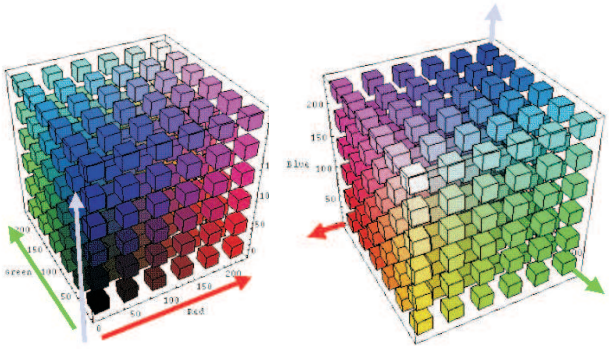


Figure 5.9: **The RGB color cube.**

Not all color reproduction devices have the same gamut. This implies that an image that can be faithfully reproduced on one device (i.e. all colors are within the gamut) might result in an *out of gamut error* for another device. Out-of-gamut colors then need to be mapped on colors that can be reproduced (this is called *gamut mapping*).

A simple way to map an out of gamut color to the physically realizable gamut is to draw the line from the white point to the wanted color. The intersection of the line with the gamut is provides a color that is in the gamut. Note that in this way we are mixing the wanted (unrealizable) color with white and thus the selected color on the boundary of the gamut has about the same color appearance as the unrepresentable color.

5.1.7 Color recording

Color reproduction requires just three colorimetric independent primaries. No special relation to the human spectral color subspace was needed. For color recording the situation is quite different.

The principle of any tristimulus color recording device is very much like the photo receptors in the human retina. Three independent sensors with three different sensitivity curves measure the electromagnetic energy. Let $\mathbf{r}$, $\mathbf{g}$ and $\mathbf{b}$ be the sensitivities (vectors) for the three color recording channels. The three measurements then are:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = S_{\text{RGB}}^{\mathsf{T}} \mathbf{t}$$

where $S_{\text{RGB}} = (\mathbf{r}\,\mathbf{g}\,\mathbf{b})$ is the RGB sensitivity matrix of color recording device and R, G and B are the responses of the three sensors.

Let S_{SML} be the sensitivity matrix associated with human color vision. In previous sections we just wrote S now we use S_{SML} to distinguish the 'eye sensitivity matrix' from an 'artificial' color recording sensitivity matrix. We will use the term 'camera' from now on as a generic term for any color recording device.

Obviously for a color camera we want that whenever two beams $\mathbf{t}_1$ and $\mathbf{t}_2$ lead to the same color sensation for the human eye, i.e. $S_{\text{SML}}^{\mathsf{T}}\mathbf{t}_1 = S_{\text{SML}}^{\mathsf{T}}\mathbf{t}_2$, these two beams are also indistinguishable for the camera, i.e. $S_{\text{RGB}}^{\mathsf{T}}\mathbf{t}_1 = S_{\text{RGB}}^{\mathsf{T}}\mathbf{t}_2$. Also for two beams that do have a different color for the eye, we would like that the camera is also capable of making a distinction. In short we would

like that the equivalence classes of beams for the human eye and for the color camera are the same, i.e. for all beams $\mathbf{t}_1$ and $\mathbf{t}_2$:

$$S_{\text{SML}}^{\mathsf{T}}\mathbf{t}_1 = S_{\text{SML}}^{\mathsf{T}}\mathbf{t}_2 \Longleftrightarrow S_{\text{RGB}}^{\mathsf{T}}\mathbf{t}_1 = S_{\text{RGB}}^{\mathsf{T}}\mathbf{t}_2.$$

This leads to the condition that the color recording sensitivity vectors are linear combinations of the eye sensitivity vectors:

$$S_{\text{RGB}} = S_{\text{SML}}A$$

with A a 3×3 non-singular matrix. A color recording device that satisfies this requirement is called *colorimetric*.

Let S_{RGB} be the sensitivity matrix of a colorimetric color recording device. How can we relate the recorded response R, G and B to the XYZ color model? Because both $C_{\bar{x}\bar{y}\bar{z}}$ and S_{RGB} are matrix multiplications of the eye sensitivity matrix with a 3×3 non-singular matrix we have that:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{\text{XYZ}\rightarrow\text{RGB}} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

where $M_{\text{XYZ}\rightarrow\text{RGB}}$ is a 3×3 color conversion matrix. Because there is no unique RGB color model, you can find several of these XYZ to RGB conversion matrices in the literature.

In practice it is nearly impossible to design colorimetric color camera's. In the next subsection where we discuss Cohen's matrix it will be shown that non-colorimetric camera's in principle lead to color matches different from the human eye.

5.1.8 Cohen's matrix

In the previous subsections we have seen that matrices of the form $S_{\text{SML}}A$, with S_{SML} the eye sensitivity matrix and A any 3×3 non-singular matrix, are *colorimetric equivalent* as they all lead to equivalent color matches. What all matrices of the form $S_{\text{SML}}A$ have in common is that their column vectors span the same 3-dimensional subspace of the high dimensional spectral space. In this section we look at Cohen's matrix as a coordinate independent way of describing this *color subspace*.

Consider the following situation. In laboratory A scientists have measured the color matching matrix C_A using the primaries $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$. In lab B the color matching matrix C_B is measured using the primaries $\mathbf{q}_1$, $\mathbf{q}_2$ and $\mathbf{q}_3$. How do the color matching matrices relate? Do both labs agree on which beams match in color?

The answer to the second question is *yes* (i.e. in case the standard observers in both labs are indeed standard), both labs do agree on the equivalence of color beams. But their numerical tristimulus response vectors are certainly different.

Let $P = (\mathbf{p}_1\,\mathbf{p}_2\,\mathbf{p}_3)$ be the primary matrix of lab A and let $Q = (\mathbf{q}_1\,\mathbf{q}_2\,\mathbf{q}_3)$ be the primary matrix of lab B. For lab A the color matching matrix turns out to be $C_A^{\mathsf{T}} = (S_{\text{SML}}^{\mathsf{T}}P)^{-1}S_{\text{SML}}^{\mathsf{T}}$ (see Eq.(5.1)), for lab B we get $C_B^{\mathsf{T}} = (S_{\text{SML}}^{\mathsf{T}}Q)^{-1}S_{\text{SML}}^{\mathsf{T}}$. Note that for both color matching matrices the columns span the same subspace of the spectral space.

This 3D subspace of spectral space is an important one: for any spectrum $\mathbf{t}$ we can find a *unique* metameric spectrum $\mathbf{t}^\star$ within the subspace. This spectrum is called the *fundamental spectrum* and can be written as a linear combination of the three column vectors in the sensitivity matrix S_{SML}:

$$\mathbf{t}^\star = S_{\text{SML}}\mathbf{a}$$

with $\mathbf{a}$ the 3D vector of the scalar multiples of the three sensitivity vectors. Because $\mathbf{t}^\star$ is a metamer of $\mathbf{t}$ we should have a color match: $S_{\text{SML}}^{\mathsf{T}}\mathbf{t} = S_{\text{SML}}^{\mathsf{T}}\mathbf{t}^\star$. This leads to $\mathbf{a} = (S_{\text{SML}}^{\mathsf{T}}S_{\text{SML}})^{-1}S_{\text{SML}}^{\mathsf{T}}\mathbf{t}$

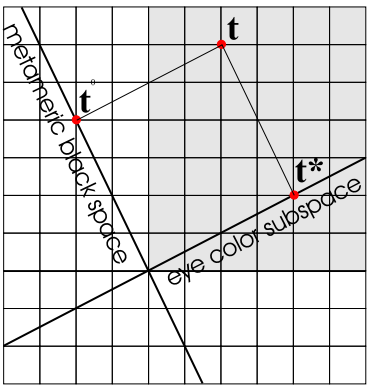


Figure 5.10: **A simplified view on spectral space.** The eye color subspace is spanned by the vector $(2\,1)^\mathsf{T}$. Cohen's matrix Π projects a spectrum $\mathbf{t}$ onto the fundamental spectrum $\mathbf{t}^\star$. The orthogonal complement of the eye color space is called the *metameric black space*. The space of all physical realizable spectra (with all positive elements) is rendered in grey.

and thus to $\mathbf{t}^\star = S_{\text{SML}}(S_{\text{SML}}^\mathsf{T} S_{\text{SML}})^{-1} S_{\text{SML}}^\mathsf{T}\mathbf{t}$. The operator

$$\Pi = S_{\text{SML}}(S_{\text{SML}}^\mathsf{T} S_{\text{SML}})^{-1} S_{\text{SML}}^\mathsf{T} \tag{5.3}$$

is called *Cohen's matrix*, and maps a spectrum $\mathbf{t}$ onto its fundamental spectrum $\mathbf{t}^\star$. Obviously if $\mathbf{t}$ is a fundamental spectrum itself (i.e. lies in the 3D color subspace of spectral space) we should have that it is mapped onto itself (remember that we have said that the fundamental spectrum is unique). This means that the operator Π should be idempotent, i.e. $\Pi^2 = \Pi$ (such an operator is called a projection operator, see also Exercise 12). *The projection operator thus serves as a basis independent way of characterizing the 3D color subspace of spectral space.*

Returning to our example of the two labs A and B. Although both will find different color matching matrices C_A and C_B respectively, both will have the same projection matrix.

In Fig. 5.10 a simplified view on the projection in spectral space is sketched. Here we have taken spectral space to be 2 dimensional and we take the eye color subspace to be one dimensional (with basis vector $(2\,1)^\mathsf{T}$). Cohen's matrix in this case is:

$$\begin{aligned}
\Pi &= S_{\text{SML}}(S_{\text{SML}}^\mathsf{T} S_{\text{SML}})^{-1} S_{\text{SML}}^\mathsf{T} \\
&= \begin{pmatrix} 2 \\ 1 \end{pmatrix} \left(\begin{pmatrix} 2 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \right)^{-1} \begin{pmatrix} 2 & 1 \end{pmatrix} \\
&= \frac{1}{5} \begin{pmatrix} 4 & 2 \\ 2 & 1 \end{pmatrix}
\end{aligned}$$

The projection on the metameric black space is done with the operator $I - \Pi$ where I is the identity matrix in spectral (N-dimensional) space. With Cohen's matrix we can write any spectrum $\mathbf{t}$ as a sum of the fundamental spectrum $\mathbf{t}^\star = \Pi\mathbf{t}$ and a vector in metameric black space $\mathbf{t}^0 = (I - \Pi)\mathbf{t}$. Note that we can replace the metameric black space vector with any other spectrum in the black space without changing the color response.

When comparing color recording devices, a solid way of doing so is to compare the associated projection matrices. In practice they will *not* be equal to Cohen's matrix for the human eye as few artificial color recording devices are colorimetric.

In Fig. 5.11 again the simplified 2D view on spectral space is sketched. A non-colorimetric camera subspace is also sketched. Note that the metameric spectra leading to the same color

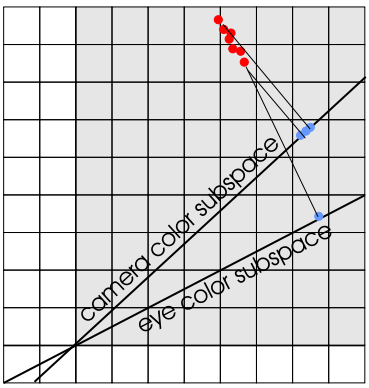


Figure 5.11: **A non-colorimetric camera.** The camera color subspace is not the same as the eye color subspace. Therefore spectra that are metamers for the human eye are distinguishable to the camera.

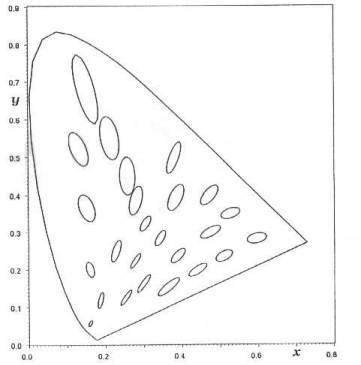


Figure 5.12: **MacAdam ellipses.** see text.

sensation for the human eye are projected in the camera color subspace onto different spectra meaning that the recorded colors in the camera are different. Whereas the human eye cannot distinguish the spectra, the camera does make a distinction. The same applies for spectra that are metameric for the camera; these appear as different colors to the human eye.

5.1.9 Perceptual color differences

Although the XYZ color model serves us very well as a standard for judging color equivalence as a basis for a standard for color recording, for color reproduction and for color communication, it has a severe drawback when talking about the perceptual difference of two *different* colors. Let (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2) be two different colors. The Euclidean distance measure

$$\Delta = \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2 + (Z_1 - Z_2)^2}$$

is known *not* to result in perceptually uniform differences in color perception.

In Fig. 5.12 for some colors (the centers of the ellipses) the colors that are just noticeably different are indicated (all colors on the ellipse contour). The experiment showing the just

noticeable different colors was first performed by MacAdam, the ellipses are therefore called the *MacAdam ellipses*.

The CIE proposed the $L^\star u^\star v^\star$ color model to be perceptually uniform:

$$L^\star = \begin{cases} 116\left(\frac{Y}{Y_n}\right)^{\frac{1}{3}} - 16 & : \ \frac{Y}{Y_n} > 0.008856 \\ 903\frac{Y}{Y_n} & : \ \frac{Y}{Y_n} \leq 0.008856 \end{cases}$$

here $L^\star$ is called *lightness*, the perceptual response to luminance (Y) and Y_n is the luminance of the white reference. Lightness perception is thus roughly logarithmic. Two patches of equal color are distinguishable in case the lightness differs by more than about one percent.

Given a perceptual lightness indication $L^\star$ we need two color components 'perpendicular' to it to describe the color of the beam. The CIE has chosen these such that in the resulting color parameterization the Euclidean distance measure roughly corresponds with perceptual differences. First we calculate u' and v' (these are part of an obsolete standard):

$$u' = \frac{4X}{X + 15Y + 3Z} \qquad v' = \frac{9Y}{X + 15Y + 3Z}$$

and from here we calculate

$$u^\star = 13L^\star(u' - u'_n) \qquad v^\star = 13L^\star(v' - v'_n)$$

where again the underscore n denotes that the u_n and v_n are calculated for the reference white. Note that the $u^\star$, $v^\star$ 'chromaticities' are translated to have the reference white in the center.

5.2 Further Reading

Colorimetry. B.A. Wandell, *The Foundations of Color Measurement and Color Perception*, `http://white.stanford.edu/~brian/refs.html`, SID Color Tutorial Notes.

A well-written tutorial from a well-known researcher in the field. Down loadable from the Internet.

Colorimetry. J.J. Koenderink, *Color* (lecture notes), Department of Physics, Utrecht University.

Color. G. Wyszecki and W.S. Styles, *Color Science*, John Wiley and Sons, New York, 1982.

The 'bible' for color science. Not recommended unless you are looking for something specific or you are knowledgeable in the field.

Color Management Systems. `www.color.org`

CIE. The "Commission International de l'Eclairage" that defined the standards in the field, `http://www.cie.co.at/cie/home.html`.

Colorimetric Data. Looking for the XYZ colormatching matrix? Here, `http://www-cvrl.ucsd.edu/`, you will find all you need. In case you would like to import the data in Matlab, you should look at the function dlmread.

Poynton's Gamma and Colour FAQs. Here you find a lot of information on color and gamma related non-linearities (especially the technical stuff). Very useful! `http://www.inforamp.net/~poynton/Poynton-colour.html`

Beyond trichromacy. Although for us, human beings, color is a three dimensional phenomenon, for many other animals it is more than that. Look at the *Evolution of photo-receptors* at the www-site `http://www.univie.ac.at/Vergl-Physiologie/www/research/morphretframe.html` to learn that reptilian retinas have 5 different 'color' channels.

Color Systems. For some beautiful pictures of old and modern color systems see `http://www.uni-mannheim.de/fakul/psycho/irtel/colsys/`. You can also find an interesting web based way to learn whether you have calibrated your monitor correctly.

5.3 Exercises

1. **Chromaticity plane.**
   Consider the chromaticity plane in tristimulus space of the vectors $\mathbf{c} = (c_1\, c_2\, c_3)^\mathsf{T}$ defined with: $c_1 + c_2 + c_3 = 1$. Show that the normalized color coordinates:
   $$\bar{c}_1 = \frac{c_1}{c_1 + c_2 + c_3} \quad \bar{c}_2 = \frac{c_2}{c_1 + c_2 + c_3} \quad \bar{c}_3 = \frac{c_3}{c_1 + c_2 + c_3}$$
   are the coordinates on the chromaticity plane. Also show that any two of the normalized coordinates provide a unique characterization of the color.

2. **Chromaticity coordinates (chromaticities).**
   Instead of defining a plane, we can also define a surface as the basis to define chromaticity coordinates. Consider the coordinates:
   $$\bar{c}_1 = \frac{c_1}{\sqrt{c_1^2 + c_2^2 + c_3^2}} \quad \bar{c}_2 = \frac{c_2}{\sqrt{c_1^2 + c_2^2 + c_3^2}} \quad \bar{c}_3 = \frac{c_3}{\sqrt{c_1^2 + c_2^2 + c_3^2}}.$$
   What is the chromaticity surface in this case?

3. **Color match.**
   Prove that $S^\mathsf{T}\mathbf{l}_1 = S^\mathsf{T}\mathbf{l}_2 \Leftrightarrow AS^\mathsf{T}\mathbf{l}_1 = AS^\mathsf{T}\mathbf{l}_2$ in case A is non-singular.

4. **Color match.**
   Show that $S^\mathsf{T}P$ in Eq.(5.1) is indeed non-singular.

5. **The color white.**
   (a) White is a color just like any other. Regrettably there is no unique definition on which color we call white. Search on the Internet to find different definition of the color white and plot the chromaticity values in the xy-chromaticity diagram. (Hint: look for "D65", "Color temperature", "white", etc).
   (b) It is sometimes said that white light has a uniform spectrum. Show that this is not necessarily true by mixing any color (thus also a white color) with just two monochromatic beams. Sketch the construction in the xy-chromaticity diagram.

6. **Rendering the Spectral Colors.**
   (a) Faithful rendering of spectral colors is by definition impossible using a color rendering device based on a finite number of primary colors. Why?
   (b) In the article *Color Rendering of Spectra*, J. Walker (`http://www.fourmilab.ch/documents/specrend`) describes a C-program to approximately render the spectrum. Rewrite their program in Matlab. Make sure that you utilize the color reproduction capabilities of the monitor your program is running on.

7. **Dominant wave-length characterization of colors.**
Instead of describing a color with its XYZ coordinates (or any other choice for a color basis) we can parameterize color in many more ways. One of them characterizes a color by its dominant wave-length (also termed 'hue').

   Consider a spectrum $\mathbf{u} = \alpha + \beta\mathbf{\Delta}_j$ where α and β are constants and Δ_j is the monochromatic beam of wavelength λ_j with total energy β. The dominant wave length equals λ_0.

   (a) The spectrum $\mathbf{u} = \alpha + \beta\mathbf{\Delta}_j$ is a mixture of a monochromatic beam and a beam of equal intensity over all wavelengths (white light). Can you plot the resulting color in the xy-chromaticity diagram?

   (b) Calculate the parameters α, β and j such that the beam $\mathbf{u} = \alpha + \beta\mathbf{\Delta}_j$ results in a color sensation equivalent to a beam characterized by X, Y and Z.

   (c) What happens for the purples?

   (d) Comment on the perceptual meaning of the dominant wavelength λ_j and the quotient β/α. Compare this with the well-known Hue, Saturation and Intensity representation of color.

8. **XYZ-cone.**
Given a spectrum $\mathbf{t}$, the XYZ-color is given as $\begin{pmatrix} X & Y & Z \end{pmatrix}^\mathsf{T} = C_{\bar{x}\bar{y}\bar{z}}\mathbf{t}$. All elements in $\mathbf{t}$ are positive, as well as all elements in the color matching matrix $C_{\bar{x}\bar{y}\bar{z}}$, thus all XYZ color vectors have positive elements. We thus have to consider only the positive first octant of the XYZ space. But not all XYZ triples in the first octant correspond with physically realizable colors.

   Let $\mathbf{\Delta}_i$ be the monochromatic beam of wavelength λ_i. Then any beam $\mathbf{t}$ can be written as a linear combination of the monochromatic beams:

$$\mathbf{t} = \sum_i t_i \mathbf{\Delta}_i.$$

   The XYZ color triple corresponding with the beam $\mathbf{t}$ than is linear combinations of the XYZ triples of the monochromatic beams:

$$\begin{aligned}
\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} &= C^\mathsf{T}_{\bar{x}\bar{y}\bar{z}}\mathbf{t} \\
&= C^\mathsf{T}_{\bar{x}\bar{y}\bar{z}}\left(\sum_i t_i \Delta_i\right) \\
&= \sum_i t_i C_{\bar{x}\bar{y}\bar{z}}\mathbf{\Delta}_i \\
&= \sum_i t_i \begin{pmatrix} X_i \\ Y_i \\ Z_i \end{pmatrix}
\end{aligned}$$

   where $(X_i\, Y_i\, Z_i)$ is the i-th row from the matrix $C_{\bar{x}\bar{y}\bar{z}}$.

   (a) Show that the convex hull of all half lines $a(X_i\, Y_i\, Z_i)^\mathsf{T}$ with $a > 0$ and for all i defines the cone of all physically realizable spectra.

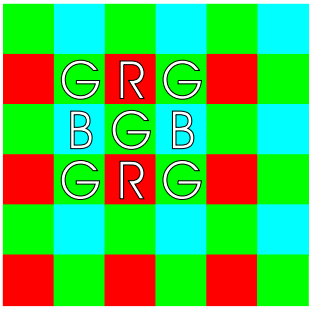


Figure 5.13: **Bayer color mosaic in a one CCD color camera.**

(b) Write a Matlab program to visualize this cone[4].

9. **The locus of the monochromatic beams in the xy-plane.**
Write a Matlab program to display the locus of all monochromatic beams in the xy chromaticity diagram.

10. **Monochromatic beams.**
Explain that a monochromatic beam cannot be matched with the mixture of other beams.

11. **Color mosaicing.**
'Cheap' video camera's are often equipped with just one array of photosensitive elements. In order to record a color RGB response the sensitivity of the individual elements are coated with one of three color filters mimicking colorimetric sensitivity curves. Fig. 5.13 shows a small part of a one-chip RGB CCD layout. This particular mosaic of the RGB sensors is a *Bayer* mosaic. The G sensitivity curve is close to the luminance response of the human eye.

(a) Explain that for every pixel we have only one scalar measurement.

(b) Why do you think there are more G sensors than R or B sensors?

(c) Construct a simple linear interpolation algorithm that at each sample location results in a R, G and B value.

(d) Write a Matlab program that implements the *demosaicing*. As test input you can take a 3 channel RGB image and generate the mosaic image yourself. Compare your algorithm with the original image.

(e) Can you suggest some improvements to reduce the smoothing effect inherent to the interpolation technique?

12. **Cohen's matrix.**

(a) Prove that the linear operator $\Pi = S_{\text{SML}}(S_{\text{SML}}^{\mathsf{T}} S_{\text{SML}})^{-1} S_{\text{SML}}^{\mathsf{T}}$ is a projection. I.e. prove that $\Pi^2 = \Pi$.

(b) Show that a color matching matrix C of the form $S_{\text{SML}} A$ with A a non-singular 3×3 matrix, leads to the same projection operator Π.

[4]The nicest visualization of this cone will be used as illustration in an upcoming update of this chapter

13. **An orthonormal color matching matrix.**
Let $S_{\text{SML}}\mathbf{a}$ be a spectrum obtained as a mixture of the three human sensitivity vectors. The color response is given by $\mathbf{c} = S_{\text{SML}}^{\mathsf{T}} S_{\text{SML}}\mathbf{a}$. Can you construct an orthonormal color matching matrix C such that $\mathbf{c} = C^{\mathsf{T}}C\mathbf{a} = \mathbf{a}$? Look at the function orth in Matlab. Plot the vectors of the orthonormal C matrix. (Hint: orthogonalization of S_{SML} is equivalent to orthogonalization of $C_{\bar{x}\bar{y}\bar{z}}$).

14. **Metameric black-space.**

    (a) What is the dimension of the metameric black space?

    (b) Enumerate all spectra in the black space that are physically realizable.

Chapter 6

The Local Structure of Images

When you look at images, even abstract depictions of things you have never seen before, you will find that your attention is drawn to certain *local* features: dots, lines, edges, crossings etc. In a sense, these are 'prototypical pieces' of an image, and we can describe every image as a field of such local elements. You may group these to larger objects, and recognize those, but viewed as a bottom-up process perception starts with local structure.

In this chapter, we present some rather basic mathematical tools to describe and detect this local structure. Staying true to the theme of this course, we will use only those tools which can describe this local structure without being too specifically dependent on a particular instantiation of image coordinates or pixels.

6.1 An image, locally

An image $f : \mathbb{R}^2 \to V$ has a value $f(\mathbf{a})$ at every location $\mathbf{a}$. We need to find away of characterizing the *local structure*, i.e. what the image looks like near the location $\mathbf{a}$. So, taking a set of small translations $\mathbf{x}$, how can we characterize the image values $f(\mathbf{a}+\mathbf{x})$ as a *structure*, exhibiting the local coherence and prototype? In this section, we completely focus on the local structure of scalar-valued images, and call those values *gray values*.

6.1.1 Linear approximation

If the translation $\mathbf{x}$ is small enough, and the image smooth enough, so that the image values do not vary too suddenly from one location to the next, then $f(\mathbf{a}+\mathbf{x})$ will vary *linearly* with $\mathbf{x}$, in two senses:

- Moving twice as far in the $\mathbf{x}$-direction will make the image function rise just about twice as much as moving to $\mathbf{x}$, and the same for any multiple $\alpha\mathbf{x}$. In formula:

$$f(\mathbf{a}+\alpha\mathbf{x}) - f(\mathbf{a}) \approx \alpha\,(f(\mathbf{a}+\mathbf{x}) - f(\mathbf{a})\,)$$

- Changes always add up: moving over $\mathbf{x}$ and then over $\mathbf{y}$ gives the identity

$$f(\mathbf{a}+\mathbf{x}+\mathbf{y}) - f(\mathbf{a}) = f(\mathbf{a}+\mathbf{x}) - f(\mathbf{a}) + f((\mathbf{a}+\mathbf{x})+\mathbf{y}) - f(\mathbf{a}+\mathbf{x}). \tag{6.1}$$

  Note that this formula looks at the local image at two locations $\mathbf{a}$ and $\mathbf{a}+\mathbf{x}$. If the function is smooth, then to a good approximation both differences can be evaluated at the location

a. So then:

$$f(\mathbf{a}+\mathbf{x}+\mathbf{y}) \approx f(\mathbf{a}) + (f(\mathbf{a}+\mathbf{x}) - f(\mathbf{a})\,) + (f(\mathbf{a}+\mathbf{y}) - f(\mathbf{a})\,)\,. \tag{6.2}$$

Realize the difference between eq.(6.1) and eq.(6.2)!

Both conditions combined mean that $f(\mathbf{a}+\mathbf{x}) - f(\mathbf{a})$ is in good approximation a linear function of the translation $\mathbf{x}$. It can therefore be written as the inner product of $\mathbf{x}$ with some vector $\mathbf{g}$:

$$f(\mathbf{a}+\mathbf{x}) - f(\mathbf{a}) \approx \mathbf{x} \cdot \mathbf{g}(\mathbf{a}) \tag{6.3}$$

since that is the most general form of a scalar linear function of a vector variable (Write it out in components: it is $x\,\mathbf{g}_x + y\,\mathbf{g}_y$, the general linear function, as a 'gray value landscape' it is a tilted plane through the origin $(\mathbf{a}, f(\mathbf{a}))$ as in Figure 6.1c.) We write $\mathbf{g}(\mathbf{a})$, since in general the linear function will be different at different places in the image, and therefore $\mathbf{g}$ is dependent on the location $\mathbf{a}$. You can see this in Figure 6.1.

Obviously, the next question is: how do we find $\mathbf{g}(\mathbf{a})$ if all we know is the image f? Several ideas could come to mind:

- We could use the local image data $f(\mathbf{a}+\mathbf{x})$ in a best-fit procedure to find $\mathbf{g}(\mathbf{a})$ at each location $\mathbf{a}$. You would naturally tend to do this if you had the image data available as a set of discrete locations, such as an array of pixels.
- If we had an analytical function f describing the image, we could find $\mathbf{g}(\mathbf{a})$ using the derivatives of f.

At first sight, the second method looks infeasible, since very few images would seem to be describable as nice functions, and this is certainly unlikely for naturally occurring images. Yet both methods are quite feasible, and the second is actually more general and leads to a good framework for feature detectors.

6.1.2 The 1-jet

A mathematician would differentiate eq.(6.3) with respect to the vector $\mathbf{x}$, and find immediately that she should set: $\mathbf{g}(\mathbf{a}) \approx \boldsymbol{\nabla} f(\mathbf{a})$, with $\boldsymbol{\nabla} f(\mathbf{a})$ the *gradient* (vector derivative) of f. But since you are probably not familiar with differentiation of scalar functions with respect to vectors, we'll have to reach this result more indirectly, using the coordinate representation of the location by (x, y)-coordinates. This will also give an explicit expression for that gradient, and motivate its use.

We will need to denote both the global coordinates of the point we are studying, and the local coordinates of points nearby. We do that compactly using the following convention: (X, Y) will denote global coordinates, and (x, y) the local coordinates.

In these coordinates, eq.(6.3) becomes

$$f(\mathbf{a}_X + x, \mathbf{a}_Y + y) - f(\mathbf{a}_X, \mathbf{a}_Y) \approx x\,\mathbf{g}_x(\mathbf{a}_X, \mathbf{a}_Y) + y\,\mathbf{g}_y(\mathbf{a}_X, \mathbf{a}_Y) \tag{6.4}$$

Realize again what this means: taking the variation of the function f in the direction $(x\ y)^\mathsf{T}$, we find a linear function in x and y. It only depends on where we are in the image (which is the location $\mathbf{a}$). But if it is a 2-dimensional linear function, we can determine it fully by looking at the variation in two independent directions. How those are chosen does not matter; since we have the image represented in (x, y)-coordinates, it makes sense to look at the variation in the x-direction and y-direction.

We can obtain an equation for the x-variation by itself, through setting $y = 0$:

$$f(\mathbf{a}_X + x, \mathbf{a}_Y) - f(\mathbf{a}_X, \mathbf{a}_Y) \approx x\, \mathbf{g}_x(\mathbf{a}_X, \mathbf{a}_Y)$$

This equation views f as a function of x for constant $\mathbf{a}_Y$, so as a 1-dimensional scalar function of a scalar variable x. You know how to manipulate such functions from your calculus class.

Let us rewrite this function f in terms of a Taylor-series around the point $\mathbf{a}_X$ (remember, for now $\mathbf{a}_Y$ is just a constant parameter):

$$f(\mathbf{a}_X + x, \mathbf{a}_Y) \approx f(\mathbf{a}_X, \mathbf{a}_Y) + x f'(\mathbf{a}_X, \mathbf{a}_Y) + \tfrac{1}{2} x^2 f''(\mathbf{a}_X, \mathbf{a}_Y) + \cdots \tag{6.5}$$

where f' is the derivative of f with respect to the first coordinate x, and $f'' = (f')'$, the second derivative. For the full 2-variable function, this *partial derivative with respect to* x is written as $\frac{\partial f}{\partial x}(\mathbf{a})$, or in a convenient shorthand $f_x(\mathbf{a})$. For small x, the term with x^2 is negligible relative to the term with x, so substituting this in eq.(6.4), we find:

$$\mathbf{g}_x(\mathbf{a}) \approx f_x(\mathbf{a}).$$

By similar reasoning for the y-variation we find $\mathbf{g}_y(\mathbf{a}) \approx f_y(\mathbf{a})$. We take these approximate equalities, valid to the first order, as the definition of $\mathbf{g}$. So we set:

$$\mathbf{g}(\mathbf{a}) = \begin{pmatrix} f_x(\mathbf{a}) \\ f_y(\mathbf{a}) \end{pmatrix} \equiv \boldsymbol{\nabla} f(\mathbf{a})$$

This defines the *gradient (vector)* of f, denoted by $\boldsymbol{\nabla} f$ (sometimes pronounced 'nabla f'), in terms of the (x, y)-coordinates. But it should not matter how we have computed $\boldsymbol{\nabla} f$, for the definition eq.(6.3) makes no reference to coordinates. That means that $\mathbf{g}$ must have a geometrical meaning, independent of the coordinate system. This meaning is depicted in Figure 6.3: $\mathbf{g} = \boldsymbol{\nabla} f$ is a vector in the direction of steepest ascent of f, it is the direction in which f increases most in value. The amount by which it grows is indicated by its magnitude.

In local (x, y) coordinates, the structure of the image around the location $\mathbf{a} = (\mathbf{a}_X, \mathbf{a}_Y)$ is now described to first order as:

$$f(\mathbf{a}_X + x, \mathbf{a}_Y + y) \approx f(\mathbf{a}_X, \mathbf{a}_Y) + x\, f_x(\mathbf{a}_X, \mathbf{a}_Y) + y\, f_y(\mathbf{a}_X, \mathbf{a}_Y)$$

Knowing only 3 numbers: $f(\mathbf{a})$, $f_x(\mathbf{a})$, $f_y(\mathbf{a})$ we can therefore describe the local image around $\mathbf{a}$, for all locations near it. Since this uses derivatives of order 1, this triple is known as *the* 1*-jet of* f *at* $\mathbf{a}$. Note that those 3 numbers are usually different at all locations at which you want to describe the image locally.

Figure 6.1 gives a local image and two terms of its Taylor approximation (ignore the fourth column for now). Together, they give a first order approximation (a plane in gray-values) to the original The same concepts are depicted for a more complicated function in Figure 6.2: you can see how the linear patches begin to form a reasonable approximation to the original (ignore the fourth column for now).

So 'from function to image', we can apply these ideas easily. The converse, 'from image to function' will need some work, especially since we want to work with discretized images (how do you determine the partial derivative of a bunch of pixels on a discrete grid?), and it is therefore postponed till Section 6.2.

6.1.3 The 2-jet

The 1-jet describes the image locally to first order, i.e. in linear dependence of the translation vector $\mathbf{x}$. This describes the local 'steepness' of the image (when viewed as a gray value landscape).

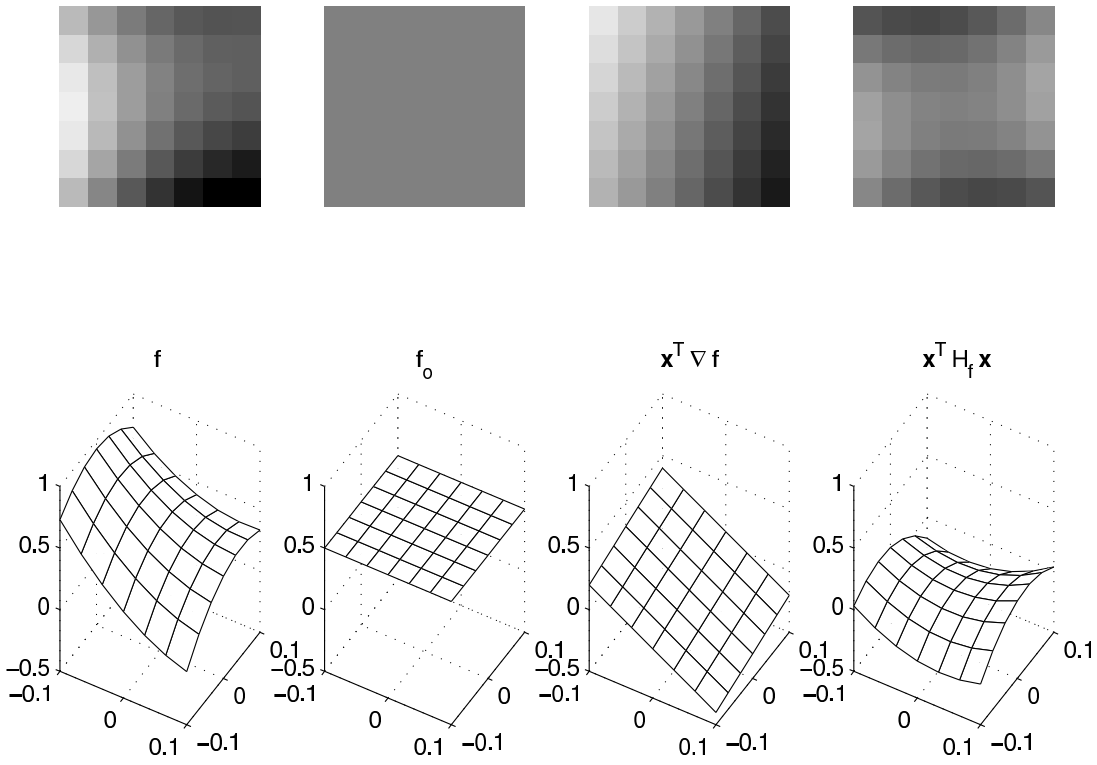


Figure 6.1: **Local Taylor approximations of a neighborhood** f: original f, zeroth order term f_0 (this has constant gray value), first order term $\mathbf{x}^\mathsf{T}\boldsymbol{\nabla} f$, second order term $\mathbf{x}^\mathsf{T} H_f \mathbf{x}$ (those two are depicted with 0 represented by grey).

Most functions (and images) are course not usually exactly linear; subtracting the first order approximation from the original will show up deviations. Those deviations are of the second order: they can be described in (x, y)-coordinates as a function of the form:

$$\begin{aligned} f(\mathbf{a}_X + x, \mathbf{a}_Y + y) - f(\mathbf{a}_X, \mathbf{a}_Y) - x f_x(\mathbf{a}_X, \mathbf{a}_Y) - y f_y(\mathbf{a}_X, \mathbf{a}_Y) \approx \\ \approx \tfrac{1}{2} x^2 \, h_1(\mathbf{a}_X, \mathbf{a}_Y) + x\, y \, h_2(\mathbf{a}_X, \mathbf{a}_Y) + \tfrac{1}{2} y^2 \, h_3(\mathbf{a}_X, \mathbf{a}_Y), \end{aligned} \tag{6.6}$$

in which all combinations of two of the coordinates x and y occur, weighted with some coefficients which need to be determined (where we have introduced some factors of $\frac{1}{2}$ in their definition, to make later formulas simpler).

As in the case of the 1-jet, we can determine good estimations for them by differentiation; but since they are of the second order, we have to differentiate twice. We can in fact expand the function f in a second order Taylor series, the 2-dimensional counterpart of eq.(6.5). This

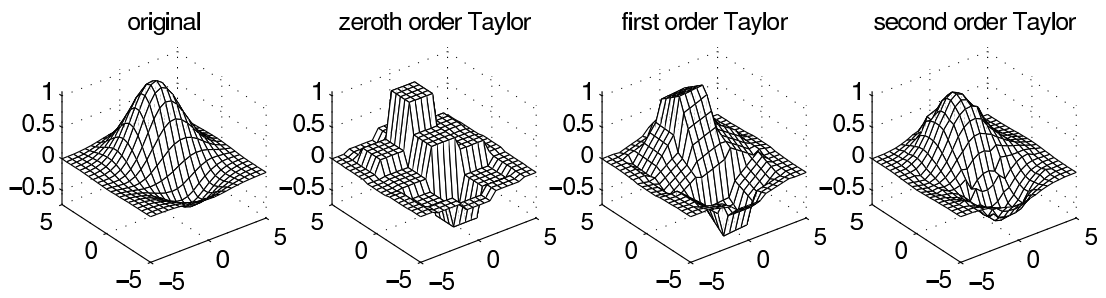


Figure 6.2: **Approximations of an image function** through Taylor approximations in local neighborhoods: zeroth order, first order, second order.

describes f locally to the second order at $\mathbf{a}$:

$$\begin{aligned} f(\mathbf{a}_X + x, \mathbf{a}_Y + y) \approx \\ \approx \quad & f(\mathbf{a}_X, \mathbf{a}_Y) \\ & + x f_x(\mathbf{a}_X, \mathbf{a}_Y) + y f_y(\mathbf{a}_X, \mathbf{a}_Y) \\ & + \tfrac{1}{2} x^2 \tfrac{\partial^2 f}{\partial x^2}(\mathbf{a}_X, \mathbf{a}_Y) + xy \tfrac{\partial^2 f}{\partial x \partial y}(\mathbf{a}_X, \mathbf{a}_Y) + \tfrac{1}{2} y^2 \tfrac{\partial^2 f}{\partial y^2}(\mathbf{a}_X, \mathbf{a}_Y). \end{aligned} \tag{6.7}$$

Comparison with eq.(6.6) gives us a rationale for chosing the h-parameters:

$$h_1 = \frac{\partial^2 f}{\partial x^2} \equiv f_{xx}, \quad h_2 = \frac{\partial^2 f}{\partial x \partial y} \equiv f_{xy}, \quad h_3 = \frac{\partial^2 f}{\partial y^2} \equiv f_{yy}.$$

We now see that we need 6 numbers to describe the local structure of the image at $\mathbf{a}$, namely the values of f, f_x, f_y, f_{xx}, f_{xy} and f_{yy} at location $\mathbf{a}$. Using those, we can describe the image locally over a larger neighborhood than with the 1-jet. This is illustrated in the fourth columns of Figure 6.1 and 6.2.

It is often more convenient to write the approximation compactly in vector notation. To write a quadratic form in terms of vectors, use the following trick which you may remember from linear algebra:

$$a\,x^2 + 2b\,x\,y + c\,y^2 = (x\ y) \begin{pmatrix} a & b \\ b & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Therefore we need to introduce a matrix of partial second derivatives to describe eq.(6.7) in coordinate-free form

$$H_f(\mathbf{a}) \equiv \begin{pmatrix} \frac{\partial^2 f}{\partial x^2}(\mathbf{a}) & \frac{\partial^2 f}{\partial x \partial y}(\mathbf{a}) \\ \frac{\partial^2 f}{\partial x \partial y}(\mathbf{a}) & \frac{\partial^2 f}{\partial y^2}(\mathbf{a}) \end{pmatrix} \equiv \begin{pmatrix} f_{xx}(\mathbf{a}) & f_{xy}(\mathbf{a}) \\ f_{xy}(\mathbf{a}) & f_{yy}(\mathbf{a}) \end{pmatrix}$$

This is called the *Hessian* of the function f in (x, y)-coordinates, at location $\mathbf{a}$.[1] With it, we can rewrite eq.(6.7) much more compactly:

$$f(\mathbf{a} + \mathbf{x}) \approx f(\mathbf{a}) + \mathbf{x}^\mathsf{T} \boldsymbol{\nabla} f(\mathbf{a}) + \tfrac{1}{2} \mathbf{x}^\mathsf{T} H_f(\mathbf{a}) \mathbf{x}$$

to second order (we have replaced the inner product $\mathbf{x} \cdot \boldsymbol{\nabla} f(\mathbf{a})$ by $\mathbf{x}^\mathsf{T} \boldsymbol{\nabla} f(\mathbf{a})$; check that this is the same!). But such a compact rewriting does not make much sense if you cannot work with it

[1] Ludwig Otto Hesse, 1811-1874.

directly, so for your convenience we will mostly compute with the explicit form of eq.(6.7). If you continue in machine perception, you should learn to manipulate vector expressions directly – it will save you a lot of work, in the long run. Figure 6.1 demonstrates the shapes of the different terms in a local neighborhood.

The derivatives that occur in this expression will have to be determined numerically from the image, and for a discrete image such differentiation would appear to be difficult. However, think back to the facet model: we can make a finite computation on a small neighborhood of a pixel to determine a good approximation to its derivatives. We will treat a slightly different method later on, but for now be assured that we can make these Taylor series operational.

6.1.4 The 0-jet

There is of course also a 0-jet description of the image. It describes the image in a local neighborhood of $\mathbf{a}$ as:

$$f(\mathbf{a}+\mathbf{x}) \approx f(\mathbf{a}).$$

So all local neighborhoods are viewed as constant (a different constant for every neighborhood, but still...). This is not very helpful to determine the local structure: only the intensity matters. This view of an image is depicted in Figure 6.1 and 6.2 as f_0.

In a sense, a pixelated image is the zeroth-order description of the reality it depicts. But the pixel's value is not the 0-jet of the real f: that would require sampling at a point. Instead, the value of a pixel is *f integrated over the aperture s of the pixel* (the aperture is the region of reality containing the points which have influenced the value of that pixel; the area of which the light-rays hit the pixel after passing through the camera). So, assuming image formation by weighted averaging over the aperture, the value is:

$$f(i,j) = \int\int f * s(x,y)\, dx\, dy,$$

Therefore a pixel image is *never* the 0-jet of reality (there are no infinitely thin probes).

Figure 6.2 shows the effect of approximating an image by its local Taylor series.

6.1.5 The gradient gauge

We have seen the specific expression for the gradient if (x,y)-coordinates are used: $\boldsymbol{\nabla} f = (f_x\ f_y)^{\mathsf{T}}$. These coefficients of the gradient vector change when we rotate the image under the camera, keeping $\mathbf{a}$ fixed, because x and y are the camera coordinates. But the local structure at $\mathbf{a}$ is of course not that different – just rotated.

So if we would want to describe 'what happens' at a certain location in the image, we cannot just look at the components of the 1-jet or 2-jet: they depend too much on how the image is placed under the camera. We should try to derive a set of numbers at each location $\mathbf{a}$ which describe the essential structure *independent of how the image is positioned*.

At locations where the gradient is not equal to zero (or, practically speaking, small), this can be done by using the gradient to establish a local direction that 'turns with the image'. We then express the second order structure in terms of coordinates that turn with the gradient, and that gives the desired characterization of the 2-jet using numbers that are always the same at the same locations.

How do we do this? The gradient direction in (x,y)-coordinates is $(f_x\ f_y)^{\mathsf{T}}$, so the unit vector in the gradient direction is:

$$\mathbf{e}_w \equiv \frac{1}{\sqrt{f_x^2+f_y^2}} \begin{pmatrix} f_x \\ f_y \end{pmatrix} \tag{6.8}$$

where we normalized by the norm of the gradient:

$$|\nabla f| = \sqrt{f_x^2 + f_y^2}$$

Figure 6.3 shows the gradient direction at selected points of the image; at those points that

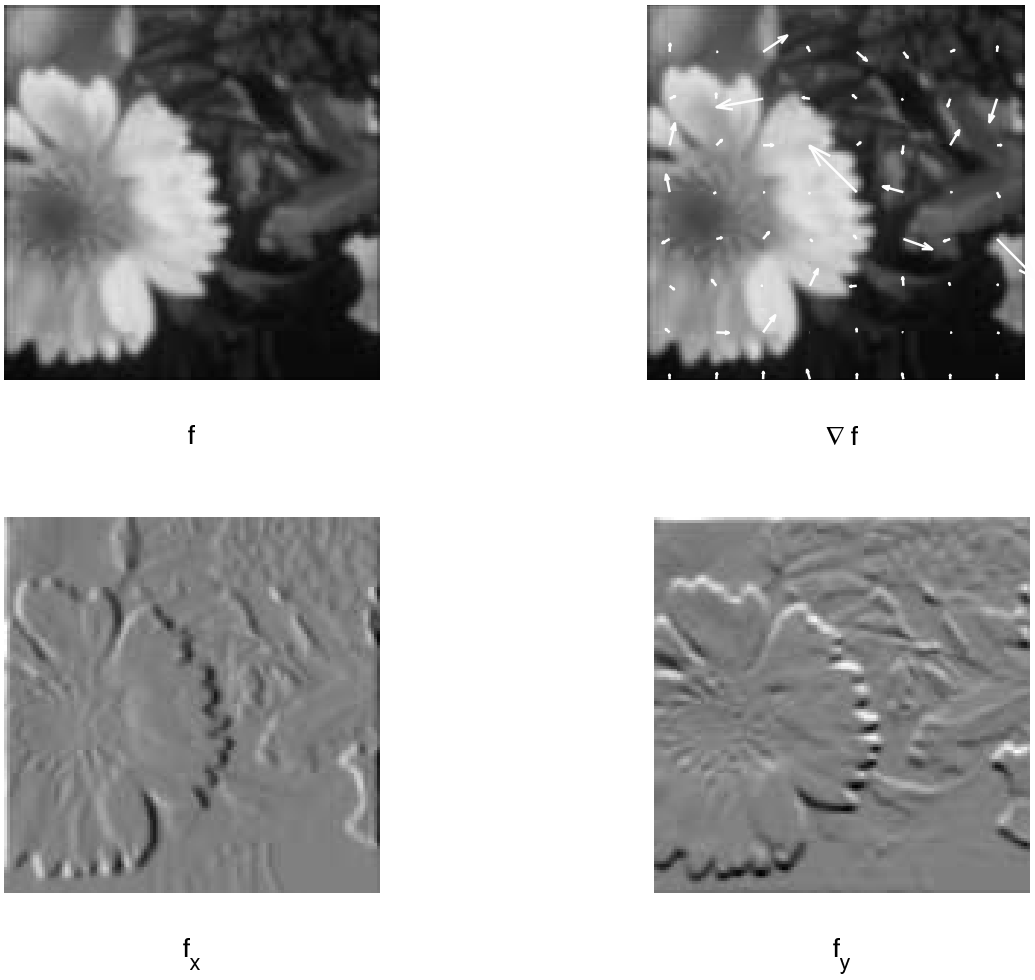


Figure 6.3: **Gradient.** An image f and its gradient: a vector-valued image of vectors ∇f (only a few are indicated). The components of ∇f on the (x, y)-frame are depicted as the gray-valued images f_x and f_y, with gray denoting zero, positive values bright and negative values dark.

is the direction of the $\mathbf{e}_w$-vector. In literature, eq.(6.8) is called the w-direction, for historical reasons. Perpendicular to it, we establish a v-direction, by turning it clockwise over 90 degrees (make sure you understand that this vector is precisely that!):

$$\mathbf{e}_v \equiv \frac{1}{\sqrt{f_x^2 + f_y^2}} \begin{pmatrix} f_y \\ -f_x \end{pmatrix}.$$

These two directions form our new coordinate frame. We compute the coordinates of any vector $(x\ y)^\mathsf{T}$ relative to this frame by a coordinate transformation R which is the appropriate rotation; so $\mathbf{v} = R\mathbf{x}$, in coordinates

$$\begin{pmatrix} v \\ w \end{pmatrix} = \frac{1}{\sqrt{f_x^2 + f_y^2}} \begin{pmatrix} f_y & -f_x \\ f_x & f_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

You see that the gradient vector $(f_x\ f_y)^\mathsf{T}$ now has different coordinates in the (x, y)-frame and in the (v, w)-frame

$$\boldsymbol{\nabla} f = \begin{pmatrix} f_x \\ f_y \end{pmatrix}_{(x,y)} = \begin{pmatrix} 0 \\ \sqrt{f_x^2 + f_y^2} \end{pmatrix}_{(v,w)}.$$

So indeed, $\mathbf{e}_w$ points in the $\boldsymbol{\nabla} f$-direction. You should realize that this is valid at *all* locations in the image, because v and w are local directions, depending on the location $\mathbf{a}$, since f_x and f_y are locally varying functions. A more complete way would be to write:

$$\boldsymbol{\nabla} f(\mathbf{a}) = \begin{pmatrix} 0 \\ \sqrt{(f_x(\mathbf{a}))^2 + (f_y(\mathbf{a}))^2} \end{pmatrix}_{(v(\mathbf{a}),w(\mathbf{a}))},$$

but we are sure you appreciate the shorthand. The local image to first order is now described using the gradient gauge as:

$$f(\mathbf{a} + v\,\mathbf{e}_v + w\,\mathbf{e}_w) \approx f(\mathbf{a}) + w\,f_w(\mathbf{a}). \tag{6.9}$$

Here is a slightly different way of understanding what has happened to the coordinates of $\boldsymbol{\nabla} f$. The first order differential structure at $\mathbf{a}$ as a function of the displacement $\mathbf{x}$ was $\mathbf{x}^\mathsf{T}\boldsymbol{\nabla} f(\mathbf{a})$, when expressed in the (x, y)-frame. In the (v, w)-frame, it should still give the same values for displacement that mean the same. So take an arbitrary vector $\mathbf{v}$ in the (v, w)-frame. Since R is a rotation, $\mathbf{v} = R\mathbf{x}$ yields $\mathbf{x} = R^\mathsf{T}\mathbf{v}$, so the vector $\mathbf{x}$ corresponds to it in the (x, y)-frame. For that vector, the first order approximation of the variation of f is $\mathbf{x}^\mathsf{T}\boldsymbol{\nabla} f$. For $\mathbf{v}$ it is therefore $\mathbf{x}^\mathsf{T}\boldsymbol{\nabla} f = (R^\mathsf{T}\mathbf{v})^\mathsf{T}\boldsymbol{\nabla} f = \mathbf{v}^\mathsf{T}(R\boldsymbol{\nabla} f)$. Therefore the gradient of the image in terms of (v, w)-coordinates is

$$\begin{pmatrix} f_v \\ f_w \end{pmatrix} = R(\boldsymbol{\nabla} f) = \frac{1}{\sqrt{f_x^2 + f_y^2}} \begin{pmatrix} f_y & -f_x \\ f_x & f_y \end{pmatrix} \begin{pmatrix} f_x \\ f_y \end{pmatrix} = \begin{pmatrix} 0 \\ \sqrt{f_x^2 + f_y^2} \end{pmatrix}$$

As the vectors rotate, so do the gradients at each location – which are after all vectors themselves.

The changes to the Hessian are more subtle, but can be computed by the same method, most clearly using the vector notation $\mathbf{x}^\mathsf{T} H_f \mathbf{x}$ for the second order term. Again, since R is a rotation, $\mathbf{v} = R\mathbf{x}$ yields $\mathbf{x} = R^\mathsf{T}\mathbf{v}$, so:

$$\mathbf{x}^\mathsf{T} H_f \mathbf{x} = (R^\mathsf{T}\mathbf{v})^\mathsf{T} H_f (R^\mathsf{T}\mathbf{v}) = \mathbf{v}^\mathsf{T}(R H_f R^\mathsf{T})\mathbf{v}$$

Therefore the new Hessian on the gradient-based frame is RH_fR^T:

$$\begin{aligned} &\begin{pmatrix} f_{vv} & f_{vw} \\ f_{vw} & f_{ww} \end{pmatrix} = \\ &= \frac{1}{f_x^2 + f_y^2} \begin{pmatrix} f_y & -f_x \\ f_x & f_y \end{pmatrix} \begin{pmatrix} f_{xx} & f_{xy} \\ f_{xy} & f_{yy} \end{pmatrix} \begin{pmatrix} f_y & f_x \\ -f_x & f_y \end{pmatrix} \\ &= \frac{1}{f_x^2 + f_y^2} \begin{pmatrix} f_x^2 f_{yy} - 2f_x f_y f_{xy} + f_y^2 f_{xx} & (f_y^2 - f_x^2) f_{xy} + f_x f_y (f_{xx} - f_{yy}) \\ (f_y^2 - f_x^2) f_{xy} + f_x f_y (f_{xx} - f_{yy}) & f_x^2 f_{xx} + 2f_x f_y f_{xy} + f_y^2 f_{yy} \end{pmatrix} \end{aligned} \tag{6.10}$$

What have we gained by this? If we now describe the local structure at **a** by the 2-jet expressed in the (v, w)-coordinates, i.e. by the numbers $f, f_v, f_w, f_{vv}, f_{vw}, f_{ww}$ at **a**, then that characterization is not dependent on how we put the picture under the camera: the numbers will always be the same (or, practically speaking, close). This permits us to classify the local structure of the image directly based on their values. Look ahead to Figure 6.11 and 6.12 to see that this is going to be very useful!

6.1.6 Edge detection

The gradient gauge coordinates help you think about the local structure and convert that into the proper formulas. An example of this is edge detection. In the (v, w)-coordinates, we might say that the central location is at an edge when

- $f_w \gg 0$ – this *detects* edge-like points as points of high gradient
- f_{ww} passes through 0 (i.e. changes sign) – this *localizes* the edge at the inflection point along it; for f_{ww} is the derivative of the edge *along its rise*, which is zero at an inflection point.

These two properties go into the *Canny edge detector*, which we ask you to implement in the lab course. The result should look similar to the 'Canny' subimage of Figure 6.11. The Canny edge detector actually finds the locations where $f_w^2 f_{ww}$ passes through 0 while $f_w \gg 0$. Why would this be preferable? (Hint: study eq.(6.10)!)

When you implement it, you should not perform the gradient gauge explicitly: just use f_x etcetera computed in image coordinates in the expressions for f_w and f_{ww} to evaluate this detector. But thinking about image structure and deriving formulas, the gauge coordinates save a lot of work – and they lead to rotation-invariant detectors. The Canny edge detector is just one example of the kinds of capabilities the gauge coordinates give us.

6.1.7 Isophotes

When thinking about the local image structure, it is often helpful to think about the 'level lines' of the same intensity. These are called *isophotes*, and in the gradient gauge locally given by the equation

$$f(v, w) = \text{constant}.$$

The isophote passing through the location we are considering can be seen as giving w as a function of v. We denote the isophote therefore by $W(v)$, implicitly defined as a function of v by $f(v, W(v)) = \text{constant}$, and satisfying $W(0) = 0$ (since it passes through $(0, 0)$ locally) and $W_v(0) = 0$ since in the gradient gauge the function f has no local first-order variation in the v-direction: the image is constant to the first order in that direction (see also eq.(6.9)).

To characterize a corner, we could use the *isophote curvature*. In terms of the local coordinates, this should be proportional to W_{vv}, the second derivative of the isophote in the v-direction.

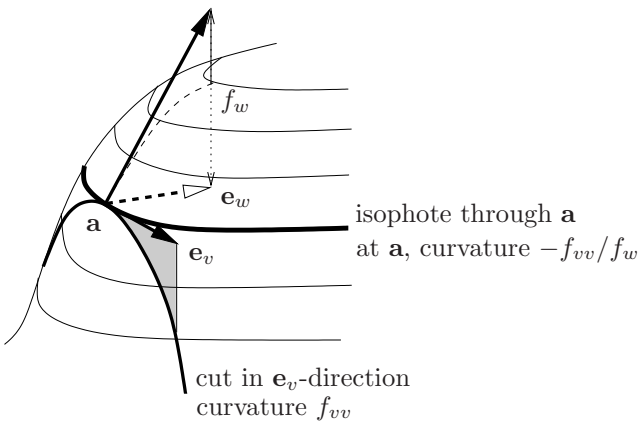


Figure 6.4: **The local neighborhood and the isophote through its center point.**

So let us compute that quantity:

$$\begin{aligned} f(v, W(v)) &= \text{constant} \\ \textit{differentiate to } v &\downarrow \\ f_v + f_w W_v &= 0 \\ \textit{differentiate to } v &\downarrow \\ f_{vv} + f_{vw} W_v + f_{wv} W_v + f_{ww} W_v^2 + f_w W_{vv} &= 0 \\ \textit{use } W_v(0) = 0 &\downarrow \\ f_{vv} + f_w W_{vv} &= 0 \end{aligned}$$

So we find the isophote curvature:

$$W_{vv} = -\frac{f_{vv}}{f_w}. \tag{6.11}$$

Note how straightforward the derivation of eq.(6.11) is using the gauge coordinates: in the expression for the second order derivative, most terms were 0 because we use the gauge coordinates.

To make a corner detector, you would really like to evaluate this only when the gradient is high, to make sure that you are at least on a steep edge rather than a local extremum. Other considerations ('invariance under affine mappings' and 'avoiding the instability of the division') lead to the specific form of the corner detector we will use:

$$-f_{vv} f_w^3. \tag{6.12}$$

This stabilizes the three divisions by f_w which are required – two in the definition of f_{vv}, and one from eq.(6.11)). Figure 6.11 gives an example of the corner detector at work (at the part labelled $-f_{vv} f_w^3$), at a certain spatial scale in the image; more about that later.

6.1.8 Prototypical 2-jet images

So far, we can classify the local images based on the structure of their 2-jet in the (v, w)-coordinates.

- straight edge: $f_w = |\nabla f| \gg 0$, $f_{vv} \approx 0$

- bright rounded corner: $f_w = |\nabla f| \gg 0$, $f_{vv} \ll 0$
- dark rounded corner: $f_w = |\nabla f| \gg 0$, $f_{vv} \gg 0$

Figure 6.1, and your own variations on it, should convince you that this is correct. It leads to part of the local classification in Figure 6.5.

6.1.9 The curvature gauge

If at some location $\mathbf{a}$ the gradient is small, the first order structure in the approximation is less important, and the local image at $\mathbf{a}$ is characterized by its second order structure. At those locations, the gradient gauge cannot be made (there is no reliable gradient to determine its direction), and other local coordinates are more suitable for understanding what is going on. An often used gauge is now the *curvature gauge*. It is a coordinate system on which the Hessian becomes diagonal. Those coordinates are traditionally denoted as (p, q)-coordinates.

Expressed in (x, y)-coordinates, the two eigenvectors of the Hessian are:

$$\mathbf{p} = \begin{pmatrix} f_{xx} - f_{yy} - \sqrt{(f_{xx} - f_{yy})^2 + 4f_{xy}^2} \\ 2f_{xy} \end{pmatrix}$$

$$\mathbf{q} = \begin{pmatrix} f_{xx} - f_{yy} + \sqrt{(f_{xx} - f_{yy})^2 + 4f_{xy}^2} \\ 2f_{xy} \end{pmatrix}$$

These are perpendicular to each other (verify that!), a consequence of the symmetry of the Hessian matrix, so by normalization they give us a coordinate frame $(\mathbf{e}_p, \mathbf{e}_q)$. We now choose our (p, q)-coordinate system in these axes. Using them to produce a rotation diagonalizes the Hessian to:

$$\begin{pmatrix} f_{xx} + f_{yy} - \sqrt{(f_{xx} - f_{yy})^2 + 4f_{xy}^2} & 0 \\ 0 & f_{xx} + f_{yy} + \sqrt{(f_{xx} - f_{yy})^2 + 4f_{xy}^2} \end{pmatrix}$$

When the gradient is negligibly small, the local image at $\mathbf{a}$ in (p, q)-coordinates is now:

$$f(\mathbf{a} + p\,\mathbf{e}_p + q\,\mathbf{e}_q) = f(\mathbf{a}) + \tfrac{1}{2}p^2 f_{pp}(\mathbf{a}) + \tfrac{1}{2}q^2 f_{qq}(\mathbf{a})$$

(This curvature gauge can of course also be performed when the first order terms are not small. But since they then dominate, that expression is not very useful since it does not help in classification of the local neighborhood.)

6.1.10 Prototypical second order images

In terms of the curvature gauge, we can classify the second order images around $\mathbf{a}$, as in Figure 6.5. Note that p is in the direction of the smallest eigenvalue of H_f. The relative magnitudes, and the signs of f_{pp} and f_{qq}, are now enough to classify the local image structure.

- dark blob on bright background: $f_w \approx 0$, $f_{pp} \approx f_{qq} \gg 0$
- dark bar on bright background: $f_w \approx 0$, $f_{pp} \approx 0$, $f_{qq} \gg 0$
- bright blob on dark background: $f_w \approx 0$, $f_{pp} \approx f_{qq} \ll 0$
- bright bar on dark background: $f_w \approx 0$, $f_{pp} \ll 0$, $f_{qq} \approx 0$

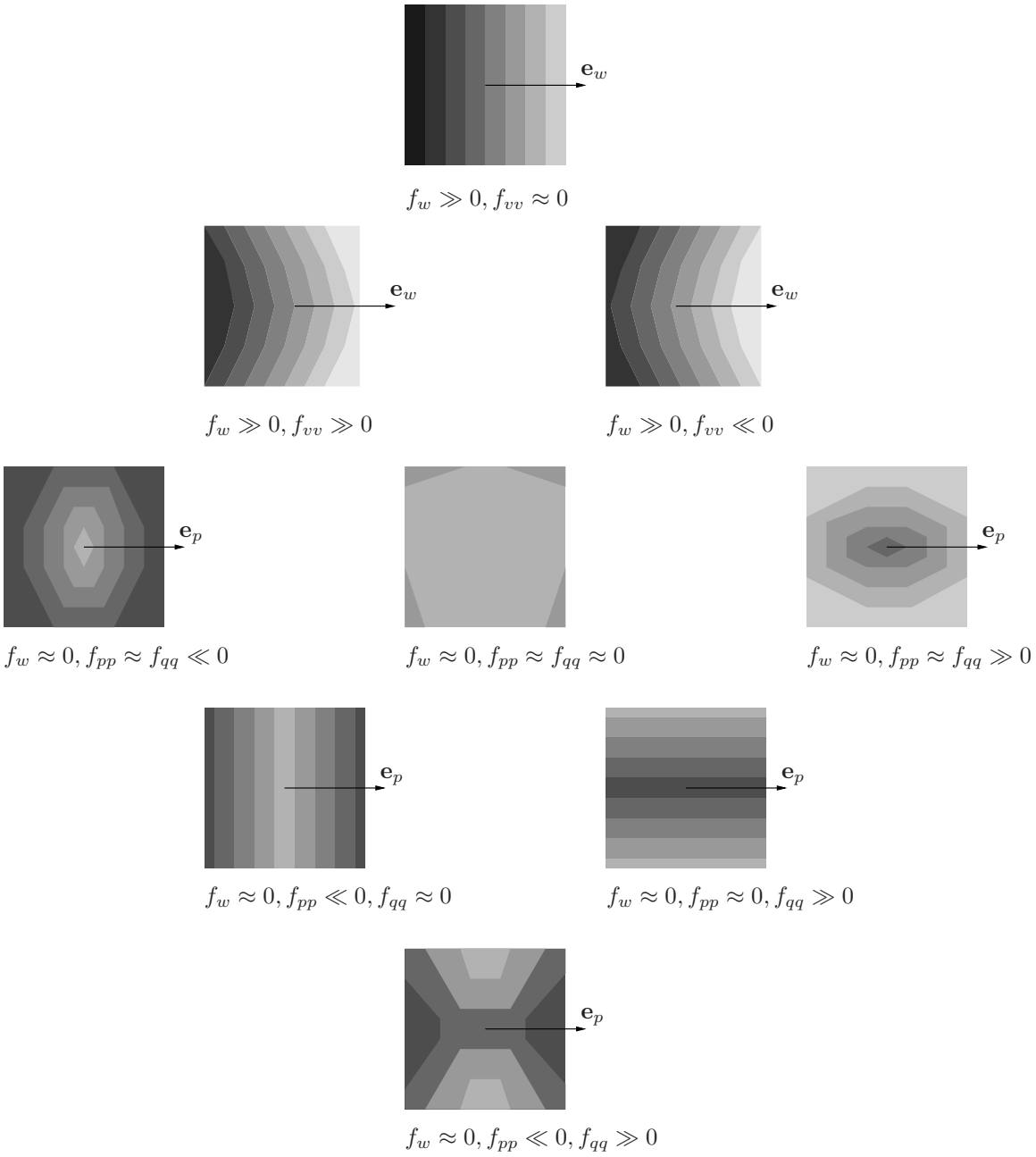


Figure 6.5: **Classification of local neighborhoods.**

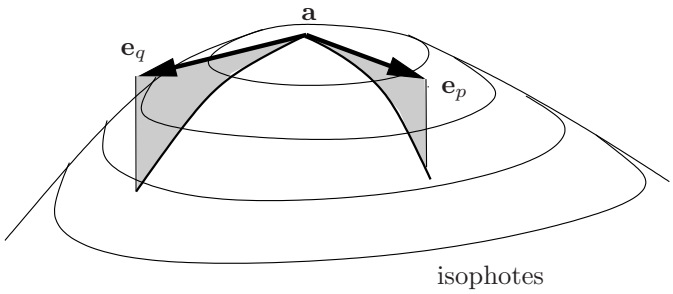


Figure 6.6: The curvature gauge is applied in the absence of a gradient.

- locally constant image patch: $f_w \approx 0$, $f_{pp} \approx 0$, $f_{qq} \approx 0$
- saddle point: $f_w \approx 0$, $f_{pp} \ll 0$, $f_{qq} \gg 0$

(Note that the 'constant' patch is constant to the second order only.) You can recognize local structures that we would characterize as 'blobs' and 'bars'. That characterization can now be translated into f_{pp} and f_{qq}. Therefore, once you have built an f_{pp}-filter and an f_{qq}-filter, you can use them to build blob and bar detectors, equally reliable independent of the orientation of your image.

But of course, we can only build them once we have shown how to compute those derivatives...

6.1.11 N-jets

So far, we have only used the local image structure up to the second order. Obviously, we can continue to higher orders and describe more advanced local images. An example could be T-junctions where 3 bars meet, which require at least a third order structure to classify. The principle is the same: approximate the local image by a truncated Taylor series of order N, and if the terms of order 1 to $(N-1)$ are small, the terms of order N describe the local behavior.

Vice versa, this is an organizing principle to give a proper and robust solution to your detection of local image structure: characterize it in terms of local derivatives of the right order, in a conveniently chosen local coordinate frame for its description; then relate that coordinate frame to the (x, y)-derivatives derivatives of the appropriate order, and of appropriate scale (see next section).

6.2 Changes at an appropriate scale

6.2.1 Naive derivatives

So we desperately need to compute derivatives of images. Since images typically come to us as a collection of discrete pixels with certain values, this is not just a straightforward application of the mathematical definition

$$\partial_{\mathbf{x}} f(\mathbf{a}) = \lim_{r \to 0} \frac{f(\mathbf{a} + r\mathbf{x}) - f(\mathbf{a})}{r}, \tag{6.13}$$

for we cannot perform the limiting process this demands.

In the early days of image processing, people would do the closest thing to taking a limit, on the grid. So for the derivative in the x-direction ($\mathbf{x} = (1\ 0)^{\mathsf{T}}$ in eq.(6.13)), they would subtract

two neighboring image values to get an estimate:

$$f_x(i,j) \approx f(i+1,j) - f(i,j).$$

You see that this involves the two pixel values at locations $(i+1,j)$ and (i,j), so we should probably say that it is an approximation of the value of the f_x-image at $(i+\frac{1}{2},j)$. Or we could take instead as an approximation:

$$f_x(i,j) \approx \tfrac{1}{2}(f(i+1,j) - f(i-1,j)),$$

which has the advantage that it is localized more at (i,j), but which does not take the value of $f(i,j)$ itself into account (do you understand why we have to divide by 2?!).

Both these ways of estimating the derivative are very sensitive to noise (small random variations in the image) so that the results often convey more about the noise-structure of the local neighborhood than about the actual variation of the image at that location. Yet others have patched that up by taking the average of a number of these derivatives at subsequent lines:

$$\begin{aligned} f_x(i,j) \quad \approx \quad & \tfrac{1}{6}(\ f(i+1,j-1) - f(i-1,j-1) \\ & \quad + f(i+1,j) - f(i-1,j) \\ & \quad + f(i+1,j+1) - f(i-1,j+1)\ ) \end{aligned} \tag{6.14}$$

This is actually the estimate of the derivative f_x as it is used in the facet model, see Section 2.7. That connection is a bit of a surprising afterthought; in their basic construction, these methods are rather *ad hoc* (although if you know what you are doing, on black-and-white images full of contrast and relatively free of noise these may be good enough; they are typically used in simple industrial vision tasks). We need a structural way of addressing this problem of taking derivatives, since they are central to our method of describing the local image structure.

6.2.2 Convolution

Before we go into this, we need to treat a very convenient shorthand for the linear combination of pixels from a neighborhood, providing a new image. This is the *convolution operation.*

Take some coefficient scheme, say

$$g = \begin{Bmatrix} 0 & 0 & 0 \\ 0 & \underline{0} & 1 \\ 0 & 0 & 0 \end{Bmatrix} \tag{6.15}$$

and an image $f(i,j)$. To compute the convolution of the image with the scheme, *point-mirror it through its origin* (indicated by the underlined number) to obtain

$$\begin{Bmatrix} 0 & 0 & 0 \\ 1 & \underline{0} & 0 \\ 0 & 0 & 0 \end{Bmatrix}$$

place that that over the image with the center (underlined) at location (i,j); multiply the image values with the coefficient values; and add all the results. That provides a number which is the result at position (i,j) of the result image.

As you write this out in two `for`-loops, you see that you are actually implementing the following formula:

$$\begin{aligned}(f*g)(i,j) &= \sum_{k'=-1}^{1}\sum_{\ell'=-1}^{1} f(i+k',j+\ell')\,g(-k',-\ell') \\ &= \sum_{k=-1}^{1}\sum_{\ell=-1}^{1} f(i-k,j-\ell)\,g(k,\ell) \qquad (6.16)\end{aligned}$$

(Make sure you understand the rewriting: what is the relationship between k and k'?) In the case above, this operation leads to the value that was in f at $(i-1,j)$ now being placed at (i,j). Obviously, this is done for all locations (i,j) in the image. The net result is that we have shifted the image over 1 pixel to the right, so over the vector $(x\ y) = (1\ 0)^{\mathsf{T}}$: $f*g$ is the shifted image. The original scheme for g showed the only non zero-entry at the location $(1\ 0)^{\mathsf{T}}$, so that is nicely consistent. (It is for that reason that the point-mirroring was introduced – it always strikes one as strangely involved, but it actually helps in the interpretation. The same operation without the point mirroring – which mean the minuses in eq.(**??**) are changed to plusses – is also useful: it is called a *correlation*.)

Eq.(6.16), generalized to arbitrarily sized schemes g, is the formula for the convolution:

$$(f*g)(i,j) = \sum_k \sum_\ell f(i-k,j-\ell)\,g(k,\ell) \qquad (6.17)$$

of which the continuous-space equivalent is:

$$(f*g)(x,y) = \int\int f(x-x',y-y')\,g(x',y')dx'\,dy'.$$

Convolution is a *linear operation*, in the sense that $(f+h)*g = f*g+h*g$ and $(\alpha f)*g = \alpha(f*g)$. A bit surprisingly, it is also *symmetrical*: $f*g = g*f$ (prove that, it's very simple, but the point-mirroring is essential to make this true!) and *associative*: $f*(g*h) = (f*g)*h$ (also not hard).

The big surprise is that *any* translation invariant linear operation on an image can be written as a convolution, see Appendix A. That makes it universal, and it is the preferred way of specifying linear operations. For instance, the derivative approximation of eq.(6.14) is the convolution of the image with:

$$g = \tfrac{1}{6}\begin{Bmatrix} 1 & 0 & -1 \\ 1 & \underline{0} & -1 \\ 1 & 0 & -1 \end{Bmatrix} \qquad (6.18)$$

Another operation is the *uniform filter* (a.k.a. *(moving) average filter*), characterized by:

$$\tfrac{1}{9}\begin{Bmatrix} 1 & 1 & 1 \\ 1 & \underline{1} & 1 \\ 1 & 1 & 1 \end{Bmatrix}$$

6.2.3 Performing convolutions

The definition of the convolution contains two $\sum$ signs, so that means two `for`-loops in an implementation. But that gives only *one* resulting point; to do it for every point in the image requires two more `for`-loops. So convolution is an expensive operation, especially for a large kernel on a large image. Sometimes the kernel is *separable*, and then you can perform it first in the i-direction and then in the j-direction; but you are rarely this lucky.

In Matlab, use the `conv2`-command for the convolution – it contains the nested `for`-loops implicitly. Matlab 'knows' how to separate kernels (Exercise: take the uniform filter at various window sizes, and plot the timing results as a function of the size).

The convolution result is not quite of the same extent as the original image, and it may not really be properly defined on its whole range. For instance, if your image has coordinates in the range $i \in [1 : N_i]$ and $j \in [1 : N_j]$, and the kernel coordinates run from -1 to 1, then computation of $(f * g)(1, 1)$ requires $f(0, 0)$, which is undefined. It is common to take such undefined elements to be equal to zero, so that the computation can proceed. This is called *zero padding*. As a consequence you can get a result from a convolution of an $(N_i \times N_j)$-sized image and an $(M_i \times M_j)$-sized kernel that can be non-zero over $(N_i + M_i - 1) \times (N_j + M_j - 1)$ pixels; yet only the inner $(N_i - M_i + 1) \times (N_j - M_j + 1)$ pixels have not been achieved by any zero-padding and are therefore really valid outcomes.

Since one would normally like $(N_i \times N_j)$-sized image as a result (to compare it more easily to the original image), the convolution is often forced to return a result of that size, but you should then realize that the borders are not quite to be trusted. In Matlab, study the options to `conv2` to achieve each of those results.

6.2.4 Scale selection through Gaussian convolution

We are interested in derivatives, since they give us the local image structure containing potentially interesting parts of the image such as blobs or bars. But since we cannot predict how big the blobs or bars might be that are relevant to our image analysis, we would really like to determine those at a specifiable *scale*. That means we should have derivatives that contain a scale parameter. The scale should be related in a convenient way to the spatial dimensions we want to detect: so that a scale of 10 times the sampling distance will indeed lead to the detection of features of just about that size.

It turns out that under very general demands, such as: 'there should be no preferential directions or locations in the image', there is a unique way of constructing an image by convolution to be used for the analysis at a scale σ. In d-dimensional images, you *must* do the convolution using a Gaussian convolution scheme, given (as a continuous function) by:

$$G_\sigma(\mathbf{x}) \equiv \frac{1}{(\sigma\sqrt{2\pi})^d}\, e^{-\frac{\mathbf{x}\cdot\mathbf{x}}{2\sigma^2}}$$

This is often called *the Gaussian kernel*; it is depicted in Figure 6.7. Note that it falls off rapidly, so that you can get good approximations with relatively small kernels. The size of the kernel chosen of course depends on σ, since that is a measure of the extent of the Gaussian. At a distance of 0, σ, 2σ and 3σ from the center, the value of the kernel are 0.1592, 0.0965, 0.0215 and 0.0018, respectively. It is common to take the kernel about 2.5σ wide, and make a discrete approximation within that by simply sampling the continuous Gaussian, and rescaling. The rescaling *must* be done since more important than the actual values of the kernel is the fact that it should be *normalized* for integration (or in the discrete case: summation): the sum of the coefficients should add to 1. This guarantees that it does not lead to a change in value when applied to a constant image. When sampling, you should not really take a grid size of less than σ.

Convolution by G_σ gives you the image at scale σ, with the proper blurring. You may show that:

$$G_\sigma * G_\tau = G_{\sqrt{\sigma^2+\tau^2}}$$

so that you can use previously blurred images to produce even coarser ones at a moderate extra cost. The fact that 'rescale of a rescaled image is again a rescaled image' is of course a fundamental requirement of any sensible rescaling worth the name. It can be shown that under very general conditions, 'convolution by a Gaussian' is the only linear rescaling operation with this property. It is a lot less arbitrary than it seems!

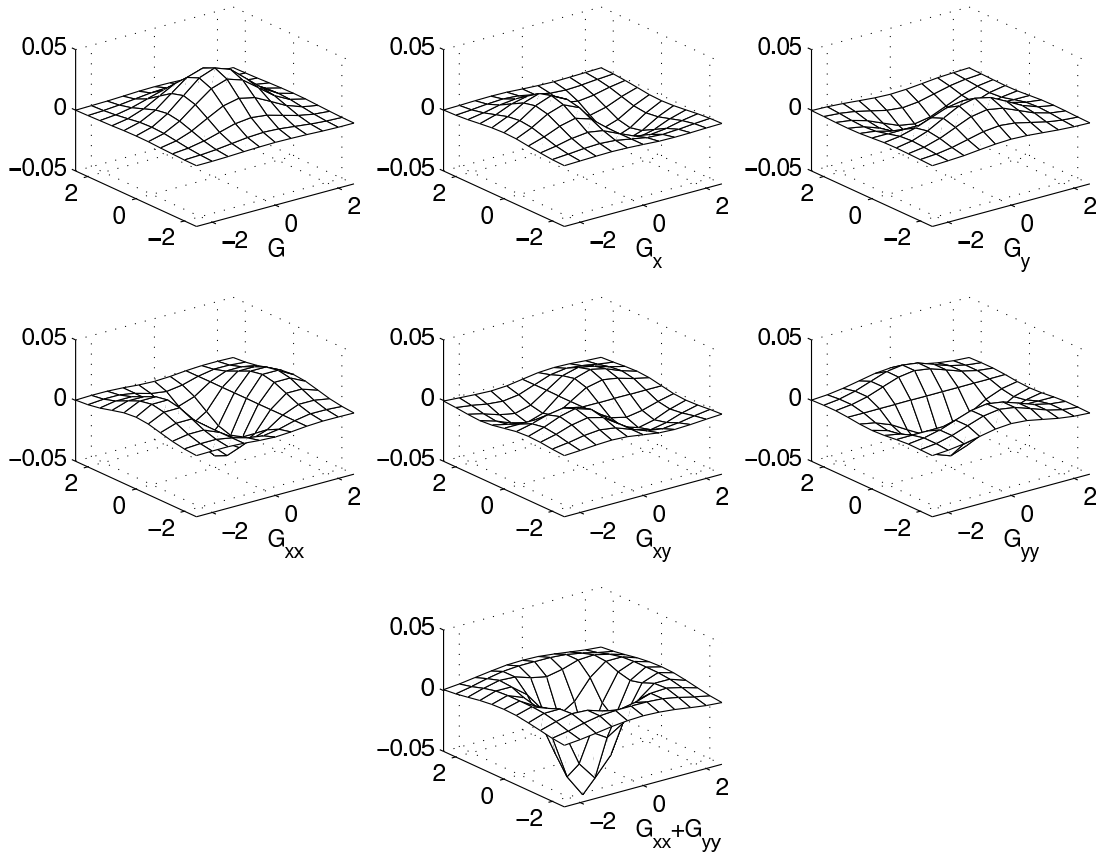


Figure 6.7: **Gaussian derivatives**: the convolution kernels, in units of σ (i.e. $\sigma = 1$).

6.2.5 Gaussian derivatives

If you want to determine the derivative of an image f at scale σ, you must take the derivative of the image considered at that scale, which is $f * G_\sigma$. But since taking the derivative is a translation invariant linear operator, it can be written as a convolution; and convolutions are associative and commutative, so symbolically (with D denoting the kernel of the differentiation – which we will not specify since it is rather tricky!)

$$\partial(f * G_\sigma) = D * (f * G_\sigma) = f * (D * G_\sigma) = f * (\partial G_\sigma).$$

In words: the derivative of an image at scale σ can be computed by convolution with the derivative of a Gaussian kernel of that scale.

So to compute f_x at scale σ, we convolve with:

$$\frac{\partial G_\sigma}{\partial x} = -\frac{x}{\sigma^2} G_\sigma$$

(verify the correctness of this formula!) and to compute f_{xx} we convolve with

$$\frac{\partial^2 G_\sigma}{\partial x^2} = (\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2}) G_\sigma$$

etcetera – you can compute the other kernels yourself. They are depicted in Figure 6.7.

This solves the differentiation problem, since those are finite kernels without any problems about limits in a neighborhood that is smaller than the grid size. And it is right that differentiation should involve a scale: they are used for the analysis of observations, and observations always need a specification of the scale at which you should consider the data.

In a discrete implementation, you should still be a bit careful when making a discrete version of these derivative kernels. It is important that the derivatives are centered at the same location as the original Gaussian, and that they integrate to zero (since they should not respond to a constant image). Both are best done by constructing them as a coefficient scheme of odd size (from 3×3 onwards). Also, to properly represent the higher order derivatives of a Gaussian you need a larger extent than of the Gaussian itself. These issues are explored in a lab course exercise.

Figure 6.7 shows the kernels for the Gaussian of $\sigma = 1$ and some of its derivates to second order, over a range of 2.5σ. (Compare the figure for G_x with eq.(6.18)...) Note that the higher order derivatives indeed tend to be less negligible at the border of the discretization, so they need a bigger kernel to be represented to the same accuracy.

We have also depicted $G_{xx}+G_{yy}$. It is rotationally symmetric, so apparently this combination of derivatives does not depend on the orientation. This is the trace of the Hessian (since it is the trace, it is indeed invariant – remember that from linear algebra!?).

6.2.6 Applying the derivatives

Now we can execute our derivatives to determine the local image structures. In Figure 6.8 we show f_x, f_y, f_{xx}, f_{xy}, f_{yy}, the basic elements of the local structure. We have indicated the derivates by rescaling, to make them be visible; light pixels are positive, dark pixels are negative. You can see clearly that the derivatives are dependent on an alignment with the image coordinates. Only if the orientation of the image was carefully chosen may these quantities have a meaning.

Figure 6.9 show those derivatives computed at a larger scale – note that only coarser edges now show up.

But we are of course mostly interested in rotationally covariant image features, so in f_w, f_{vv} etc. These are based on the computations using f_x, f_y, f_{xx} etc. You cannot compute them by single convolutions, since the formulas defining them are non-linear: for instance, $f_w = \sqrt{f_x^2 + f_y^2}$, so you make it from $f_x^\sigma = f * (\partial_x G_\sigma)$, and f_y by computing those and then making the non-linear combination. The Laplacian is the only exception: since it is a linear combination of derivates, you can make it by a single convolution with the prefabricated kernel $(G_{xx} + G_{yy})$. This is easily shown: $f_{xx} = f * G_{xx}$ and $f_{yy} = f * G_{yy}$, so that

$$\text{laplacian} f = \text{trace} H_f = f_{xx} + f_{yy} = f * G_{xx} + f * G_{yy} = f * (G_{xx} + G_{yy}).$$

It is a pretty good blob detector, at its scale: a strong negative response is indicative of a light blob in the image.

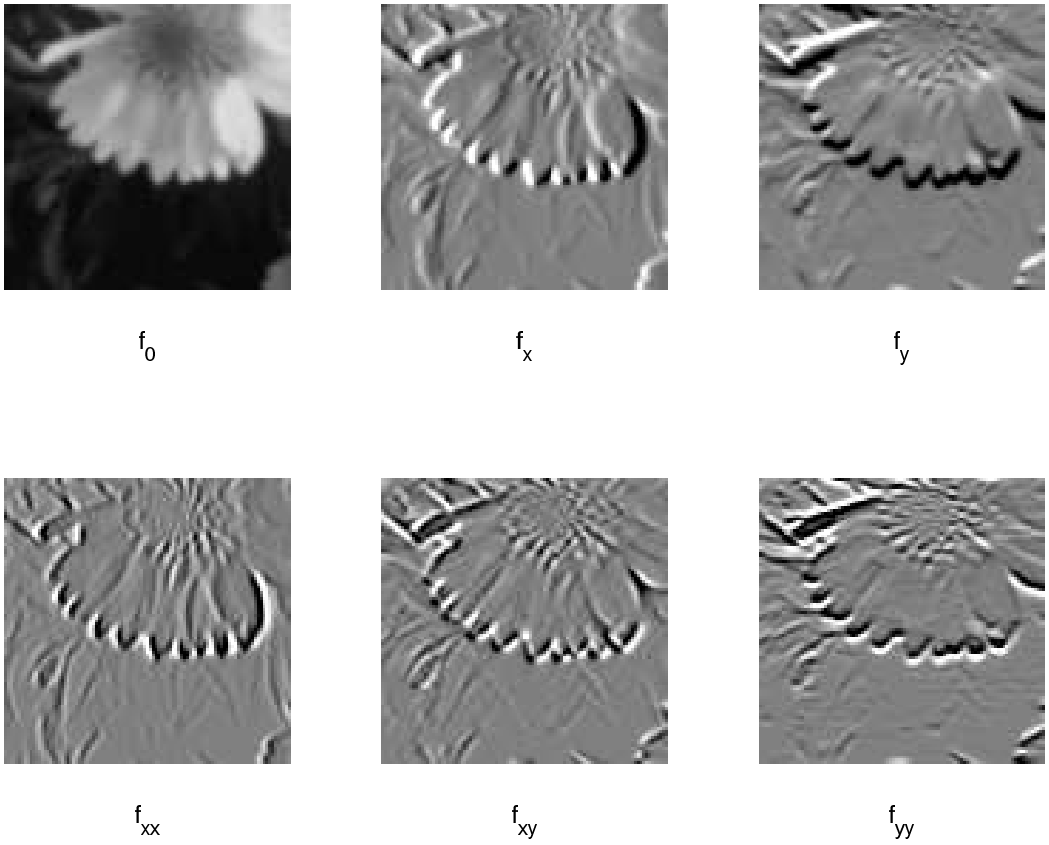


Figure 6.8: **Derivatives.** The derivatives along the (x, y)-directions, at a scale of 1.

Figure 6.11 and 6.12 show the invariants on a tell-tale detail of the figure, at two different scales. The gradient norm (which is f_w) is a good edge detector, and f_{vv} where the gradient is high was supposed to be good at detecting corners; we have therefore grouped those in the product $f_w^2 f_{vv}$ (which turns out to be even an affine invariant). Note that the scale is important in determining the detail: especially in the corner detector this is rather dramatic. We leave the blob and bar detectors to a lab course exercise.

6.2.7 Zero crossings

Often it is enough to look at the significant zero crossings of a detector, rather than at its precise values, to localize some local features. For instance the zero crossings of an appropriate second derivative denotes the inflection points of edges. Two choices are common: f_{ww}, and the *Marr edge detector* based on the Laplacian $f_{xx} + f_{yy}$.

> **Marr edge detector.** Marr introduced an 'edge detector' on the zero crossings of the image Laplacian $f_{xx}+f_{yy}$. Let us compare this to the Canny edge detector, which localizes the edges at the zero crossings of f_{ww}. We rewrite $f_{xx} + f_{yy} = f_{ww} + f_{vv} = f_{ww} - \kappa f_w$, with κ the isophote curvature. So at straight edges, the two agree, but

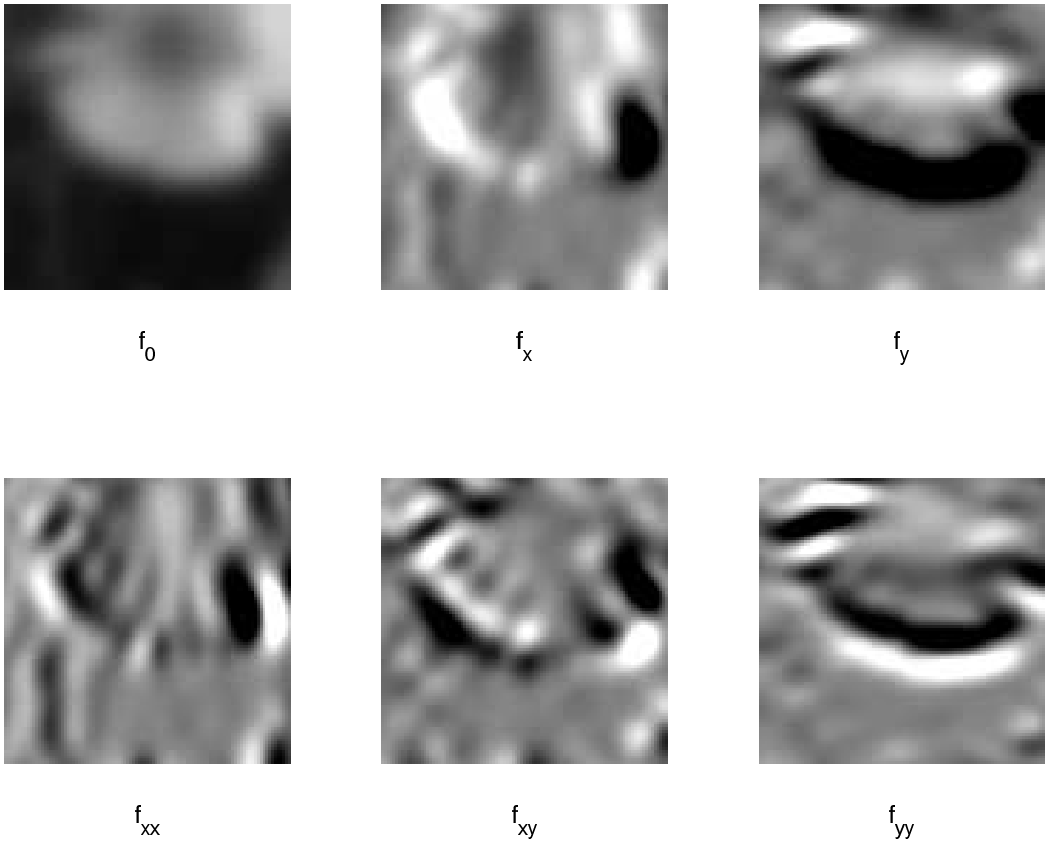


Figure 6.9: **Derivatives.** The derivatives along the (x, y)-directions, at a scale of 5.

when there is significant curvature they differ. Which of them cuts corners?

We have indicated the zero crossings of the Laplacian in Figure 6.11 and 6.12. But you should make sure that the zero-crossings are significant: for the Laplacian you really only have this meaning of approximate 'edge location' if the gradient is simultaneously high.

You can visualize the zero-crossings of any function f by applying to it a function that emphasizes the transition from light to dark, the *sigmoid*-function (so named because it is s-shaped, see Figure 6.13):

$$\sigma(x) = \frac{1 - e^{-x}}{1 + e^{-x}},$$

You can control the width by dividing x first, to scale it; this is not the same as the spatial scale σ, since you are now giving meaning to significant differences in *gray values*, not spatial extent!

Alternatively, you can use `if`-statements to discriminate the zero-crossings. The latter is faster, the sigmoid method former has nicer analytic properties if you need to do additional computations.

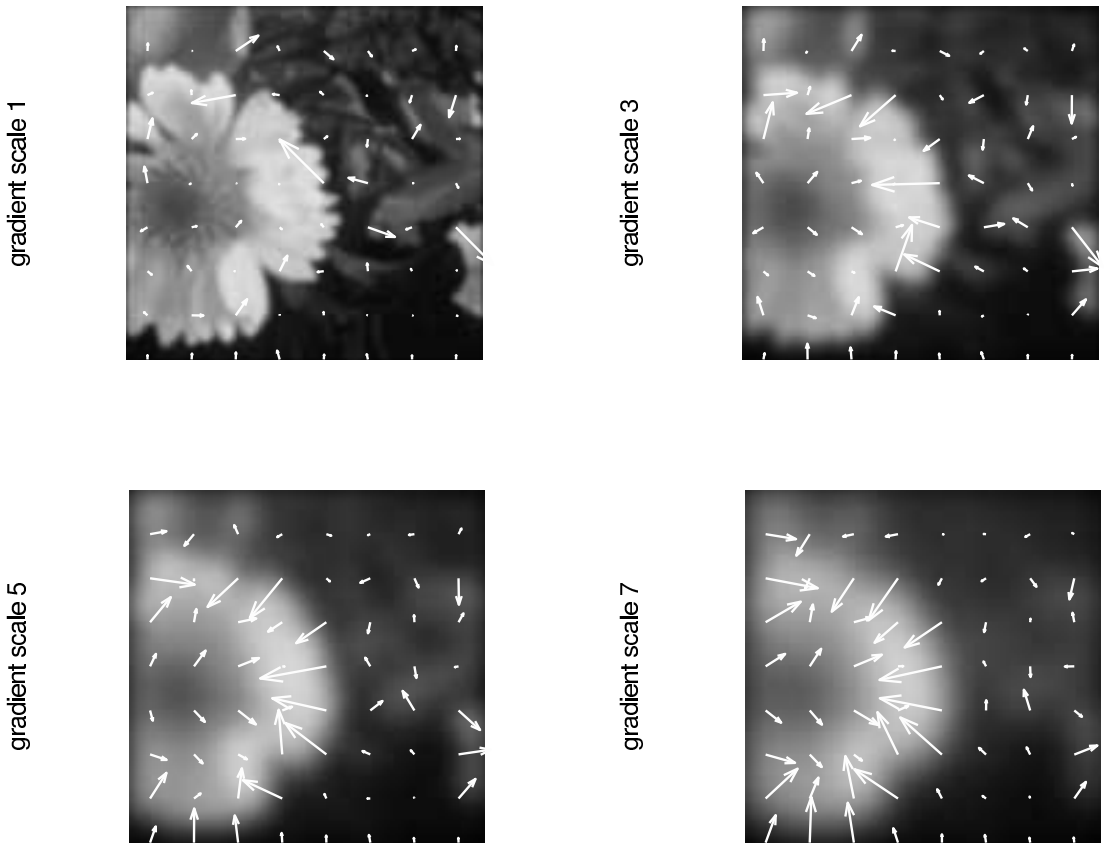


Figure 6.10: **Gradient vector.** The gradient vector ∇f at different resolutions of an image.

6.2.8 Summary: local structure

So here is the method we have for detecting certain local structures:

- Think about your detection problem locally, in terms of gradients, curvatures and higher order geometric properties.
- Convert that thinking immediately into a sensible formula using gauge coordinates: gradient gauge or curvature gauges, expressing the detector in f_w, f_{vw} etcetera. (Example: the corner detector $f_w^3 f_{vv}$.)
- Convert the result into the local image coordinates, using the standard expressions for f_w etcetera (such as $f_w = \sqrt{f_x^2 + f_y^2}$).
- Select the right scale for the problem at hand.
- Implement the detector formula using the Gaussian derivatives for that scale in image coordinates (so $f_x = f * (\partial_x G_\sigma)$, f_{xy} etcetera).

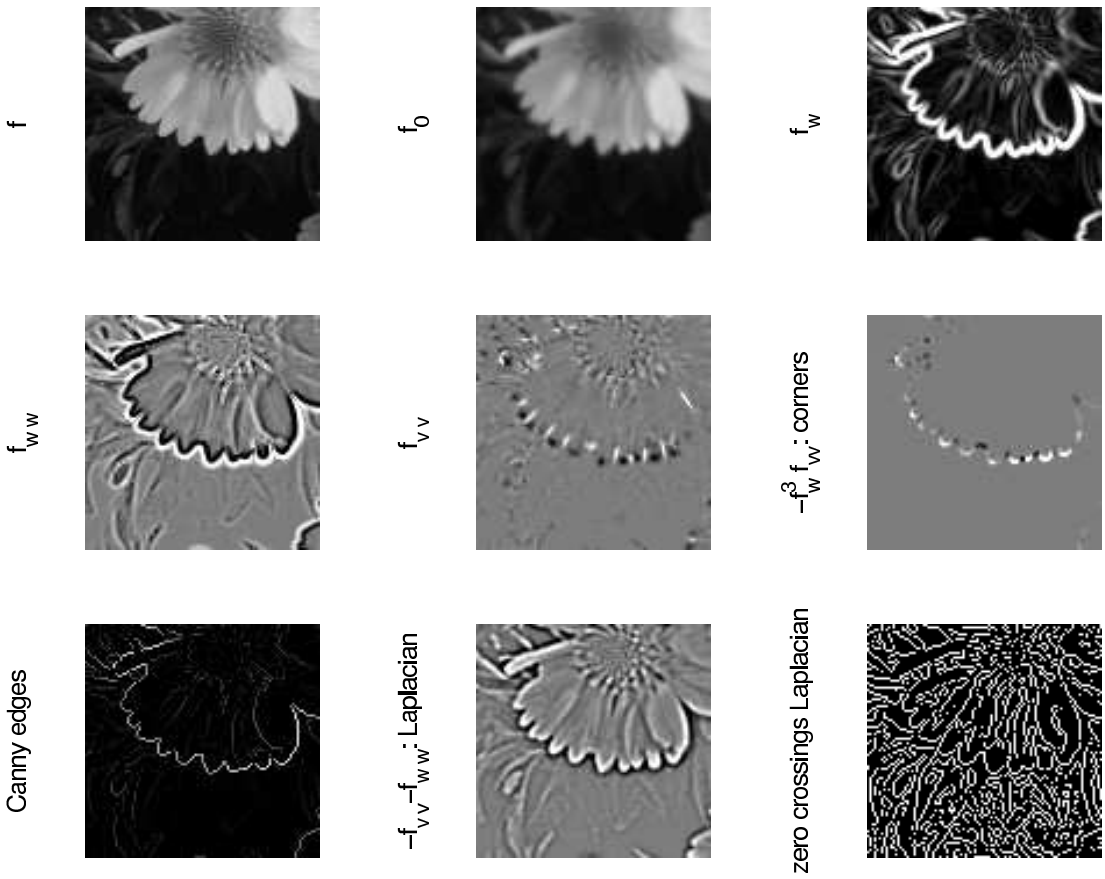


Figure 6.11: **Local invariants.** Some local invariants at scale $\sigma = 1$.

- The strength of the response (i.e. the intensity of the resulting image) gives your detected features and their relevance.

This methodology has mainly been developed by Koenderink and his co-workers in Utrecht University in The Netherlands. It is also helpful in analyzing biological systems of perception, giving a successful way of modeling biophysical data.

6.3 Exercises

1. **Local fit.**
   In Section 6.1.1 we discussed two ways of fitting a plane locally to an image. Perform the fit in the first method, in a least-square-sense, in a 3×3 neighborhood, so fit a plane $\alpha\, x + \beta\, y$ to the local image data. How do α and β depend on the pixel values in the neighborhood? (Use Matlab!)

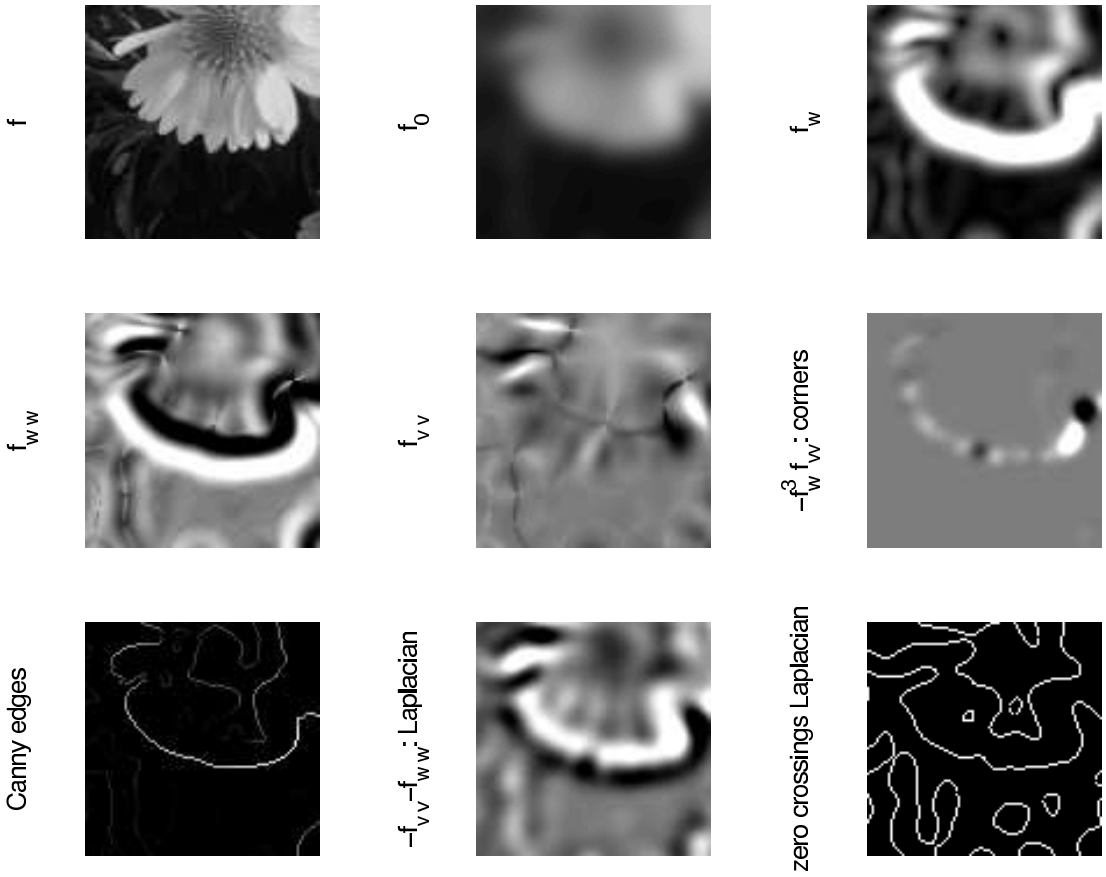


Figure 6.12: **Local invariants.** Some local invariants at scale $\sigma = 5$.

2. **1-jet**
   Compute the 1-jet of the image $f(X, Y) = X^2 + Y^3$ at the location $\mathbf{a} = (1\ 2)^\mathsf{T}$. Use it to estimate the value $f(1.1, 2.2)$. Compare this to the true value.

3. **1-jet**
   When does the approximation of the previous exercise begin to differ more than 1% of the true value? (Give a separate estimation for the X and Y values at which this happens. Use what you know about Taylor-series!)

4. **Gradient**
   Compute the gradient of $f(X, Y) = \frac{1}{2}(X^2 + Y^2)$. When is it zero? Sketch the function and interpret the result.

5. **Gradient**
   When is the gradient of the function $f(X, Y) = (X + Y)^2$ equal to zero? Sketch the function, and interpret the result as though it were an image.

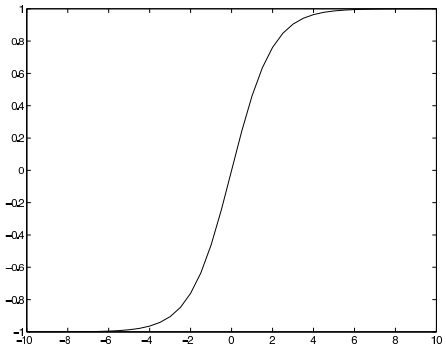


Figure 6.13: **Sigmoid.** The sigmoid function.

6. **Gradient**
   What is the gradient of the function $f(X,Y) = e^{-\frac{1}{2}(X^2+Y^2)}$? What is its direction, and what is its magnitude, at the location $\mathbf{a}$? Write down a linear approximation to f at $\mathbf{a}$, in vector form. (You should be able to factor out $f(\mathbf{a})$.)

7. **Gradient**
   Write a Matlab program to compute and draw the gradients of the image $F(i,j) = i^2 + j^3$ at all discrete locations. (Hint: use `quiver()` for your drawing.)

8. **2-jet**
   Compute the 2-jet of the image $f(X,Y) = X^2 + Y^3$ at the location $\mathbf{a} = (1\ 2)^\mathsf{T}$. Use it to estimate the value $f(1.1, 2.2)$. Compare this to the true value.

9. **2-jet**
   When does the approximation of the previous exercise begin to differ more than 1% of the true value? (Give a separate estimation for the X and Y values at which this happens. Use what you know about Taylor-series!) Compare this to the 1-jet result.

10. **Hessian**
    Compute the Hessian of $f(X,Y) = \frac{1}{2}(X^2 + Y^2)$. What is special about it?

11. **Hessian**
    What is the Hessian at the locations where the gradient of the image $f(X,Y) = (X+Y)^2$ equals zero? Sketch the image, and interpret the result.

12. **Separability**
    The uniform filter is separable. Show this, and implement it that way using `for`-loops.

13. **Separability**
    Check (by doing an experiment on computational times for different window sizes) whether Matlab uses separability when it performs the uniform filter.

14. **Convolution**
    Apart from computing a convolution to produce an $N_i \times N_j$ result by zero-padding, people sometimes just take 'the closest point in the image that is well-defined'. Give the re-addressing formula. Can you imagine why they do this?

15. **Convolution**
    Yet others 'mirror' the image f in its borders to obtain values for the pixels the convolution requires outside the image (so $f(1,2)$ is used for $f(0,2)$, etcetera). How would this compare to the previous trick?

16. **Gaussian convolution**
    Prove that the Gaussian convolution is separable.

17. **Laplacian**
    Prove that the Laplacian is rotationally symmetric. Is the Laplacian separable?

Chapter 7

The Geometry of Visual Observations

Consider the problem of finding the orange/red ball in the left image shown in Fig. 7.1. For the human visual system this is a trivial task. Programming the computer to do the same reveals the complexity of such a seemingly simple task.

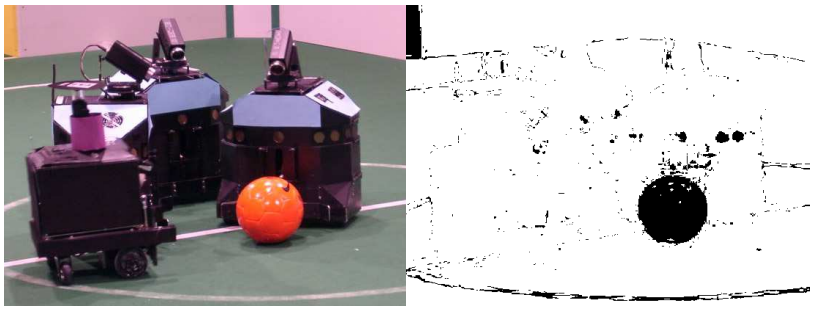


Figure 7.1: **Locating the ball.** On the left the color image is shown. On the right all points where the color corresponds with the orange/red ball are depicted in black.

In the right image of figure Fig. 7.1 all sample points where the color (see Chapter 5) is distinctive for the orange/red ball are rendered in black. Please note that no pixel wise selection of colors is possible that only selects the orange/red points belonging to the ball. We will always select the points in the top left corner as well and also a lot of points on the edge of two color regions will be selected[1].

The human visual system doesn't have much difficulty with detecting the ball shape in the image where we have labeled all the pixels that have an orange/red color in the original image. The *shape* of the cluster of 'ball points' makes it clear to us which points make up the ball.

We are able to *group* the ball points into a semantic meaningful cluster of image points. In this chapter some of the tools needed to develop the computational tools for grouping visual clues into semantic meaningful clusters will be discussed.

The result of a local clue analysis (like the simple color selection that lead to the image in Fig. 7.1) most often is a *label image* (see Chapter 2). A label image defines a partitioning of the

[1] Why is is that points on the edges are sometimes 'reddish'?

visual space into a collection of sets. The sets are subsets of the continuous real plane (for 2D images). In practice the image plane is represented as a discrete set of sample points.

In Section 7.2 we discuss some tools for *discrete geometry* from a mathematical and computational point of view. The (mathematical) language of choice for the analysis of the geometry of (sampled) sets is *mathematical morphology*.

7.1 Sets

There is a one-to-one correspondence between subsets of E and the Boolean functions over E. Let $\mathcal{A} \subseteq E$ then the 'equivalent' Boolean function $a : E \rightarrow \{0, 1\}$ is given by

$$a(\mathbf{x}) = \begin{cases} 1 & : \quad \mathbf{x} \in \mathcal{A} \\ 0 & : \quad \mathbf{x} \notin \mathcal{A} \end{cases}$$

The function a is called the *indicator function* of the set $\mathcal{A}$. The relation between sets and their indicator functions is sketched in Fig. 7.2. This *isomorphism* of subsets and Boolean functions

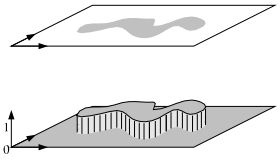


Figure 7.2: **Sets and Indicator Functions.** Both the set (shape) in the plane are shown (top figure) as well as the indicator function (bottom figure).

allows us to develop our notions and intuitions using the set formalism. Later on we can safely use the results for Boolean (binary) images as well.

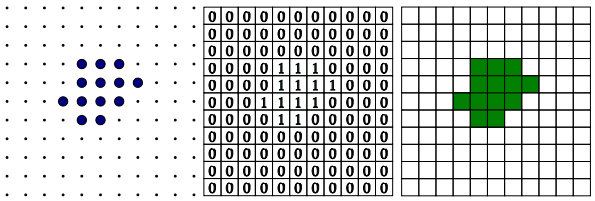


Figure 7.3: **Visualizing Discrete Sets.** From left to right: enumerating and visualizing the points in the set, visualizing the indicator function and visualizing an 'interpolated' continuous version of the set.

In image processing sets (shapes) most often are the result of some image processing operator (like the decision whether a point has the red/orange color of the ball in Fig. 7.1. In practice we thus deal with *discrete sets*, sets defined as subsets of $\mathbb{Z}^d$. In visualizing these discrete sets there are several options. We can enumerate and visualize all the points in the set (see Fig. 7.3.a). It is also possible to visualize the indicator function of the set (see Fig. 7.3.b). Finally we can 'interpolate' the discrete set to render a set in the continuous plane. The easiest way to do so is with a nearest neighbor interpolation, i.e. each sample point corresponds with a square centered on that point (see Fig. 7.3.c).

7.1.1 Set Operators

Let $\mathcal{A}, \mathcal{B}, \ldots, \mathcal{Z}$ be subsets of the space E (for now it suffices to think of E as being the real plane $\mathbb{R}^d$ but the definitions are valid for arbitrary domains). The fundamental set operators (illustrated in Fig. 7.4) is:

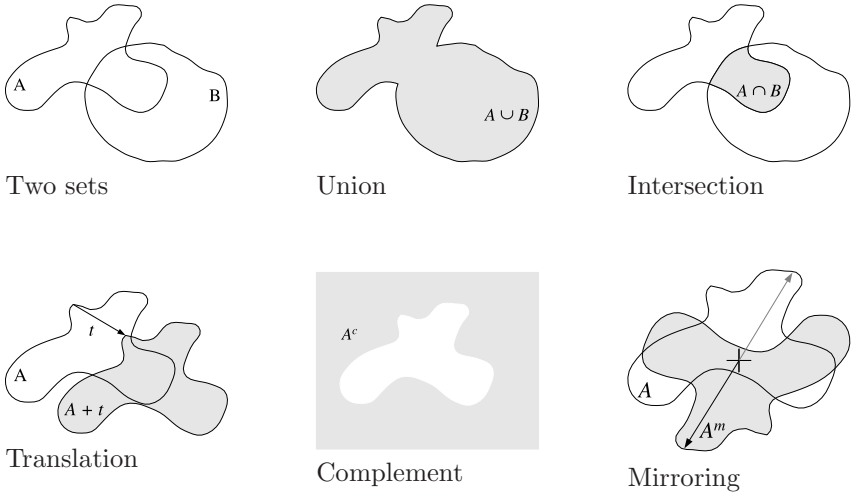


Figure 7.4: **Set operators.** The basic set operators are illustrated in this figure.

Set union. The union of two sets $\mathcal{A}$ and $\mathcal{B}$ is defined by:

$$\mathcal{A} \cup \mathcal{B} = \{\mathbf{x} \in E \mid \mathbf{x} \in \mathcal{A} \quad \text{or} \quad \mathbf{x} \in \mathcal{B}\}$$

Set intersection. The intersection of two sets $\mathcal{A}$ and $\mathcal{B}$ is defined by:

$$\mathcal{A} \cap \mathcal{B} = \{\mathbf{x} \in E \mid \mathbf{x} \in \mathcal{A} \quad \text{and} \quad \mathbf{x} \in \mathcal{B}\}$$

Set complement. The set complement is defined by

$$\mathcal{A}^{\mathsf{c}} = \{\mathbf{x} \in E \mid \mathbf{x} \notin E\}$$

Set translation. A set $\mathcal{A}$ can be translated over a vector $\mathbf{t}$ resulting in the set $\mathcal{A} + \mathbf{t}$ defined by:

$$\mathcal{A} + \mathbf{t} = \{\mathbf{x} + \mathbf{t} \mid \mathbf{x} \in \mathcal{A}\}$$

i.e. the 'addition' of a vector to a set boils down to the addition of the vector to all elements in the set.

Set mirroring. Mirroring all points in a set $\mathcal{A}$ in the origin results in the set $\mathcal{A}^\star$ defined by:

$$\mathcal{A}^\star = \{-\mathbf{x} \mid \mathbf{x} \in \mathcal{A}\}.$$

Set inclusion. A set $\mathcal{A}$ is a *subset* of set $\mathcal{B}$ in case

$$\mathcal{A} \subseteq \mathcal{B} \Longleftrightarrow (\forall \mathbf{x} \in \mathcal{A} \rightarrow \mathbf{x} \in \mathcal{B})$$

For planar sets the inclusion relation is easily visualized: in case $\mathcal{A} \subseteq \mathcal{B}$ then $\mathcal{A}$ completely fits within the set $\mathcal{B}$.

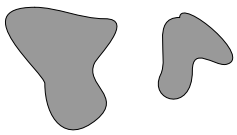


Figure 7.5: **Sets in the plane.** Depicted is *one* subset of the plane containing *two* connected components.

7.1.2 Connected Sets

Consider the set $\mathcal{A}$ depicted in Fig. 7.5. Intuitively it is clear that we have two 'objects' depicted although it is only one set. What is an 'object' then?

Each of the two objects in the figure is surrounded entirely by the background. It is impossible to draw a line without lifting your pencil from the paper, from a point within the left object to a point in the right object and not write in the 'background' (the white paper). In other words: the two objects are not *connected*.

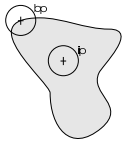


Figure 7.6: **Border and interior points.** Note that the size of the circular neighborhoods are taken to be infinitesimally small

Consider a 2D set $\mathcal{A}$ defined on the continuous plane $\mathbb{R}^2$. An ϵ-neighborhood of a point $\mathbf{x} \in \mathcal{A}$ is the infinitesimal small disk with radius $\epsilon \to 0$ centered at $\mathbf{x}$. If the ϵ-neighborhood contains only points from $\mathcal{A}$ and none of $\mathcal{A}^c$ we say that $\mathbf{x}$ is an *interior point* of $\mathcal{A}$. If the ϵ-neighborhood contains points from both $\mathcal{A}$ and $\mathcal{A}^c$ we say that $\mathbf{x}$ is a *border point* of $\mathcal{A}$ (see Fig. 7.6). The set of all border points of $\mathcal{A}$ is denoted as $\partial\mathcal{A}$ and is also called the *contour* of $\mathcal{A}$ or *boundary* of $\mathcal{A}$.

A *connected path* in $\mathbb{R}^2$ from $\mathbf{x}$ to $\mathbf{y}$ is a collection of points $\mathbf{p}(t)$ with $t \in [0, 1]$ such that $\mathbf{p}(0) = \mathbf{x}$ and $\mathbf{p}(1) = \mathbf{y}$. Furthermore the mapping $\mathbf{p}$ should be a continuous mapping, i.e. if $|t_1 - t_2| \to 0$ then also $\|\mathbf{p}(t_1) - \mathbf{p}(t_2)\| \to 0$. Again it should be possible to draw the path with a pencil without ever lifting the pencil from the paper. A *simply connected path* is a connected path that does not self intersect.

With this definition of connected paths we are able to define connected sets. A set $\mathcal{A}$ in $\mathbb{R}^2$ is a *connected set* if for any two points $\mathbf{x}$ and $\mathbf{y}$ in $\mathcal{A}$ there is a connected path $\mathbf{p}$ from $\mathbf{x}$ to $\mathbf{y}$ which is entirely within $\mathcal{A}$. Consider again the two sets in Fig. 7.5. It is clear that the left and right 'blob' are connected objects but their union (i.e. the set $\mathcal{A}$) is not.

The connectivity of a set is characterized by the number and arrangement of holes and disconnected subsets a set has, represented by the *connectivity graph*. Fig. 7.7 shows a set $\mathcal{A}$ and its connectivity graph. A connectivity graph provides a convenient way of enumerating all connected components (blobs) in a binary image.

The notions introduced in this subsection (neighborhoods, interior and border points, etc) are not trivially and not even uniquely formalized for discrete sets. Before you read on, how would you define the border $\partial\mathcal{A}$ of a discrete set $\mathcal{A}$ (and what about the property that $\partial\mathcal{A} = \partial\mathcal{A}^c$?).

The discrete sets we will consider from now on, are all on a square sampling grid generated by the standard orthonormal basis (see Chapter 2 for the definition of sampling grids).

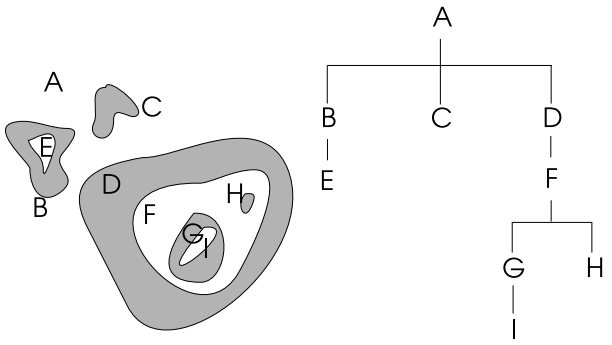


Figure 7.7: **Connectivity graph.** The first level in the graph denotes the background, the second level gives all objects in the background. The third level corresponds with all "holes" in the objects, etc, etc. The connected components of the set $\mathcal{A}$ are denoted as $\mathcal{A}_i$ and the connected background sets are denoted as $\mathcal{B}_i$.

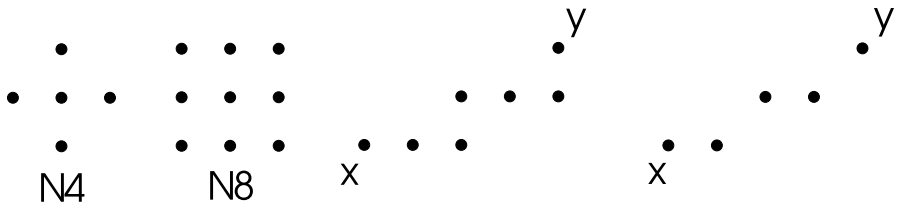


Figure 7.8: **Connectivity on the Square Grid.** From left to right: the 4-connected neighborhood, the 8-connected neighborhood, a 4 connected line and an 8-connected line.

For continuous sets (i.e. subsets of the continuous plane) the notion of an (infinitesimally small) ϵ-neighborhood was used to define the connectivity of a set. The smallest neighborhoods that can be used on the square sampling grid are the 4-connected and 8-connected neighborhoods (see Fig. 7.8).

A 4(8)-connected discrete path between the sample points $\mathbf{x}$ and $\mathbf{y}$ is a list of N sample points $\mathbf{k}(i)$ such that $\mathbf{k}(1) = \mathbf{x}$ and $\mathbf{k}(N) = \mathbf{y}$. Furthermore $\mathbf{k}(i+1)$ should be in the 4(8)-connected neighborhood of the point $\mathbf{k}(i)$.

> A discrete set $\mathcal{A}$ then is 4(8)-connected if for all $\mathbf{x}$ and $\mathbf{y}$ in $\mathcal{A}$ a 4(8)-connected path exists that connects the two points.

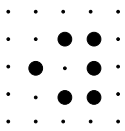


Figure 7.9: **Connectivity of object and background.** This simple object illustrates the need to define object and background connectivity differently.

In Fig. 7.9 a simple set $\mathcal{A}$ is depicted. If we define the foreground set (the heavy dots) to be 8-connected the set $\mathcal{A}$ consists of 1 connected component. As we can make a loop within

$\mathcal{A}$ around a point not in $\mathcal{A}$, the set has a hole in it. However if we also take the background set $\mathcal{A}^c$ to be 8-connected as well we have the strange contradiction that the background is 1 simple connected object. A comparable contradiction arises when we take both $\mathcal{A}$ and $\mathcal{A}^c$ to be 4-connected. To prevent these contradictions we have:

> if $\mathcal{A}$ is considered 8-connected, then $\mathcal{A}^c$ should be taken as 4-connected, equivalently
> if $\mathcal{A}$ is considered 4-connected, then $\mathcal{A}^c$ should be taken as 8-connected.

Note that notions that are so familiar and intuitive in continuous space are not necessarily valid in the discrete space. As an example consider two one pixel thick lines that are not parallel. In the continuous plane two such lines have one point in common. In the discrete space it is possible that two 8 connected lines cross and have none, one or even more points in common.

The interior of a discrete set is the set of all points where the ϵ-neighborhood fits entirely within the set. For a discrete set we have to specify the neighborhood to be $\mathcal{N}_4$ or $\mathcal{N}_8$. The interior of a 4-connected discrete set $\mathcal{A}$ is the set of points $\mathbf{x}$ where $\mathcal{N}_4 + \mathbf{x}$ completely fits within in $\mathcal{A}$, i.e. $\mathcal{N}_4 + \mathbf{x} \subseteq \mathcal{A}$. This set will be denoted as $\mathcal{A} \ominus \mathcal{N}_4$ (in a later section this 'notation' proves to be a well defined operator (erosion) specified by the set $\mathcal{N}_4$ working on the set $\mathcal{A}$ to produce a new set $\mathcal{A} \ominus \mathcal{N}_4$). We have defined the contour of a set $\mathcal{A}$ as the set of all points where the ϵ-neighborhood does not fit. I.e. the contour is the original set minus its interior[2]. The contour of the 4-connected discrete set $\mathcal{A}$ thus is:

$$\mathcal{A} \setminus \mathcal{A} \ominus \mathcal{N}_4.$$

It might come as a surprise that the contour defined in this way is *8-connected*. Equivalently the contour of an 8-connected set is 4-connected.

Given the discrete definitions of connectivity we could specify an algorithm to enumerate all points in a discrete set $\mathcal{A}$ that are connected to a given point $\mathbf{x}$. Instead of doing so now, we will first give a basic introduction to mathematical morphology because that will provide us with the tools to derive and describe our algorithms for *connected component analysis* in a more concise and elegant way.

7.2 Mathematical Morphology

Mathematical morphology provides a 'language' to reason about subsets of the visual plane and to define useful operators that transform sets into other sets. We have already seen an example. We have denoted the subset of $\mathcal{A}$ of all points $\mathbf{x}$ where $\mathcal{N}_4 + \mathbf{x}$ fits within $\mathcal{A}$ as $\mathcal{A} \ominus \mathcal{N}_4$. In the morphological language this is called the *erosion* (or *Minkowski subtraction*) of the sets $\mathcal{A}$ and $\mathcal{N}_4$. The erosion is one of the basis morphological set operators.

In this section a (very) brief introduction to mathematical morphology is presented. We restrict ourselves to the *structural* morphological operators. The structural morphological operators define set operators using *structuring elements*. These are small sets (like the set $\mathcal{N}_4$ used in the erosion to find the interior of a discrete set) that are used in the operator and correspond with the shapes we are looking for in the original set. Mathematical morphology thus provides a set oriented language for geometrical relations between sets and geometrical transformation of these sets.

[2] For sets in $\mathbb{R}^2$ the contour is defined as the difference of the *closing* of the set and its *interior*. For discrete sets the notion of open and closed sets is not essential.

7.2.1 Erosions and Dilations

The erosion and dilation are the basic spatial neighborhood operators in mathematical morphology. The dilation of two sets $\mathcal{A}$ and $\mathcal{B}$ can be interpreted as the *vectorial addition of the two sets* and is denotes as $\mathcal{A} \oplus \mathcal{B}$:

$$\mathcal{A} \oplus \mathcal{B} = \{\mathbf{x} + \mathbf{y} \mid \mathbf{x} \in \mathcal{A}, \mathbf{y} \in \mathcal{B}\} \tag{7.1}$$

i.e. the dilation (or Minskowski addition) of two sets is the additions of all pairs of points, one from $\mathcal{A}$ and the other from $\mathcal{B}$.

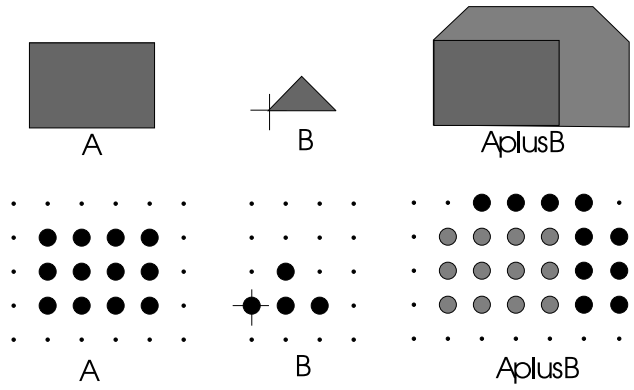


Figure 7.10: **Dilation.** In the top row the dilation of set in $\mathbb{R}^2$ is shown and in the bottom row the dilation of a discrete set is shown. The origin is indicated by a small cross.

For a more geometrical interpretation, fix a point $\mathbf{x} \in \mathcal{A}$. This point will generate the points $\mathbf{x}+\mathbf{y}$ for all $\mathbf{y} \in \mathcal{B}$. Or equivalently: we translate the set $\mathcal{B}$ to position $\mathbf{x}$. All the points in $\mathcal{B}+\mathbf{x}$ are in $\mathcal{A} \oplus \mathcal{B}$. We have to repeat this for all $\mathbf{x} \in \mathcal{A}$ and take the *union* of all translated versions of $\mathcal{B}$:

$$\mathcal{A} \oplus \mathcal{B} = \bigcup_{\mathbf{x} \in \mathcal{A}} \mathcal{B} + \mathbf{x} \tag{7.2}$$

Definition Eq.(7.2) of the dilation is helpful in geometrically constructing the dilation of two given sets, see Fig. 7.10 for examples (for subsets in the continuous space and for sampled (discrete) sets).

From Eq.(7.1) it follows that the dilation is a commutative operator, i.e.

$$\mathcal{A} \oplus \mathcal{B} = \mathcal{B} \oplus \mathcal{A}.$$

In practice the set $\mathcal{A}$ in the dilation $\mathcal{A} \oplus \mathcal{B}$, corresponds with a binary image and the set $\mathcal{B}$ usually is a small set that defines the way in which the image $\mathcal{A}$ is to be transformed. The set $\mathcal{B}$ 'structures' the set $\mathcal{A}$ and is called the *structuring element*.

Eq.(7.2) leads to the following algorithm for a dilation.

Listing 7.1: Dilation (write formalism)

```
function c = dilation( a, b )
% dilation of binary image a with structuring element b
% using the 'write formalism'
c = a; % initialize the result image
for each point (i,j) in the image domain:
    if a(i,j)==1
```

```
        for each point (k,l) in the s.e. b:
            c( i+k, j+l ) = 1
```

In words: loop over all points (i,j) in the input image a. If a(i,j)==1 the point (i,j) is in the set $\mathcal{A}$. In that case set c(k,l)=1 for all points (k,l) in the neighborhood of (i,j) (as defined by the structuring element b).

The dilation algorithm has a structure that is different from the structure of the neighborhood operators that we have encountered thus far. Previously discussed local neighborhood operators (e.g. the convolution) loop over all points in a local neighborhood relative to position (i, j) of the input image. From all these values the result value is calculated and is written to the corresponding point (i, j) in the output image. This is called the 'read formalism'.

For the dilation we can formulate a 'read formalism' algorithm as well. The dilation $\mathcal{A} \oplus \mathcal{B}$ can be rewritten as:

$$\begin{aligned} \mathcal{A} \oplus \mathcal{B} &= \{\mathbf{x} \mid \exists \mathbf{y} \in \mathcal{A} : \mathbf{y} \in \mathcal{B}^\star + x\} \\ &= \{\mathbf{x} \mid (\mathcal{B}^\star + \mathbf{x}) \cap \mathcal{A} \neq \emptyset\} \end{aligned}$$

Eq.(7.3) leads to a different (but equivalent) algorithm for the dilation.

Listing 7.2: Dilation (read formalism)

```
function c = dilation( a, b )
% dilation of binary image a with s.e. b
% using the 'read formalism'
c = a; % initialize the result image
for each point (i,j) in the image:
    r = 0;
    for each point (k,l) in the s.e. b:
        if a( i-k, j-l ) == 1
            r = 1
    c(i,j) = r
```

In words: loop over all points (i, j) and check whether at least one of the points $(i - k, j - l)$ in the neighborhood of (i, j) (defined by the mirrored set $\mathcal{B}^\star$) is in the set $\mathcal{A}$ (i.e. a(i−k,j−l)==1).

From the definition and examples of the dilation it will be clear that the dilation tends to enlarge (or 'dilate') the sets. In fact we can prove that in case the origin $\mathbf{O}$ is within the structuring element $\mathcal{B}$ we have:

$$\mathbf{O} \in \mathcal{B} \rightarrow \mathcal{A} \oplus \mathcal{B} \supseteq \mathcal{A}$$

In most practical uses of dilations the origin is indeed within the structuring element and in that case the dilation only 'adds' points to the original set.

If we have a set operator that enlarges the sets it might be handy to have an operator that shrinks sets as well. This is the *erosion* operator:

$$\mathcal{A} \ominus \mathcal{B} = \{\mathbf{x} \mid B + \mathbf{x} \subseteq \mathcal{A}\} \tag{7.3}$$

i.e. the erosion $\mathcal{A} \ominus \mathcal{B}$ is the set of all points $\mathbf{x}$ such that the translated structuring element $\mathcal{B} + \mathbf{x}$ completely fits within the set $\mathcal{A}$.

In Fig. 7.11 examples of erosions in both the continuous space and discrete space are shown.

Note the symmetry between the erosion and dilation. In the dilation every point $\mathbf{x} \in \mathcal{A}$ is replaced with the structuring element $\mathcal{B} + \mathbf{x}$. In the erosion on the other hand every occurrence of

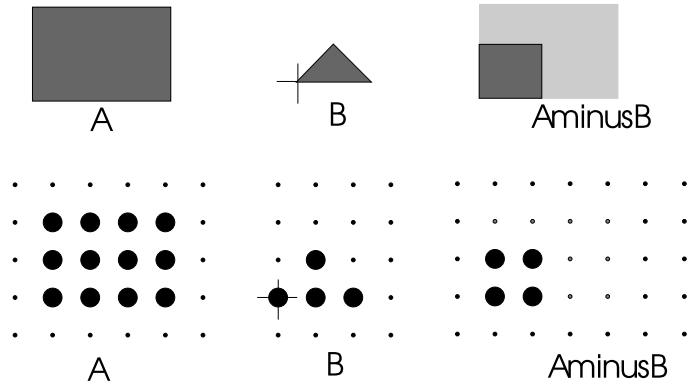


Figure 7.11: **Erosion.** In the top row the erosion of set in $\mathbb{R}^2$ is shown and in the bottom row the erosion of a discrete set is shown.

the structuring element $\mathcal{B}+\mathbf{x}$ is replaced with $\mathbf{x}$ in the resulting set. This symmetry is illustrated in Fig. 7.12.

The symmetry between erosion and dilation can also be formulated algebraically. These operators turn out to be *dual operators with respect to complementation*. It is not so hard to prove that:

$$\mathcal{A} \oplus \mathcal{B} = (\mathcal{A}^{\mathsf{c}} \ominus \mathcal{B}^{\star})^{\mathsf{c}} \tag{7.4}$$

In words this duality tells us that dilation of the object is equivalent with erosion of the background (albeit with a mirrored structuring element).

The erosion rightfully deserves its name as it removes points from the original set in case the origin is element of the structuring element:

$$\mathbf{O} \in \mathcal{B} \rightarrow \mathcal{A} \ominus \mathcal{B} \subseteq \mathcal{A}.$$

We have already seen a practical example of an erosion. The erosion $\mathcal{A} \ominus \mathcal{N}_4$ results in all points $\mathbf{x} \in \mathcal{A}$ such that $\mathcal{N}_4 + \mathbf{x} \subseteq \mathcal{A}$, i.e. the interior points of $\mathcal{A}$ are selected. The set difference $\mathcal{S} \setminus (\mathcal{A} \ominus \mathcal{N}_4)$ finds all the boundary (border) points in the set $\mathcal{A}$.

Although a dilation enlarges a set and an erosion shrinks a set, *erosion and dilation are not inverse operators*, i.e. in general $(\mathcal{A} \oplus \mathcal{B}) \ominus \mathcal{B} \neq \mathcal{A}$.

7.2.2 Properties of erosions and dilations

In this subsection we state some properties of the (structural) erosions and dilations introduced in the previous subsection.

The dilation is an *associative* operator, i.e.

$$(\mathcal{A} \oplus \mathcal{B}) \oplus \mathcal{C} = \mathcal{A} \oplus (\mathcal{B} \oplus \mathcal{C}).$$

In practice $\mathcal{A}$ is the image under study and $\mathcal{B}$ and $\mathcal{C}$ are relatively small sets (the structuring elements). The above property can be interpreted as: instead of dilating an image twice in succession (first with $\mathcal{B}$ then with $\mathcal{C}$) we can do one dilation using $\mathcal{B} \oplus \mathcal{C}$ as structuring element.

To obtain efficient algorithms for dilations it is often wise to build a large structuring element by successive dilations using smaller structuring elements. An example is the dilation using a large square structuring element. This dilation is equal to two successive dilations, one using a horizontal line, the second one using a vertical line.

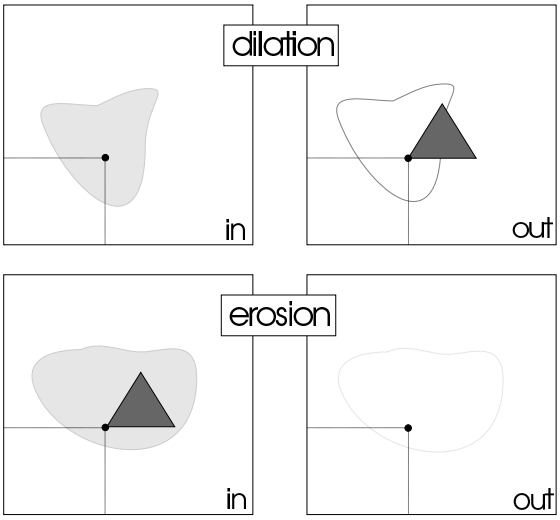


Figure 7.12: **Erosion-Dilation.** In the dilation we replace every object point with the structuring element (top row) and in the erosion we replace each occurrence of the structuring element with a point.

The erosion is *not* an associative operator, instead we have:

$$(\mathcal{A} \ominus \mathcal{B}) \ominus \mathcal{C} = \mathcal{A} \ominus (\mathcal{B} \oplus \mathcal{C}).$$

The proof is simple using the duality property of the erosion and dilation.

Another property that is of great practical consequence in developing efficient algorithms for erosions and dilations is the following one.

When dilating a set $\mathcal{A}$ with a structuring element $\mathcal{B}$ that is a simply connected set (one connected component with no holes in it) that contains the origin, it is intuitively clear that only the points on the boundary of the set $\mathcal{A}$ will generate the points that need to be added to $\mathcal{A}$ to obtain $\mathcal{A} \oplus \mathcal{B}$:

$$\mathcal{A} \oplus \mathcal{B} = \mathcal{A} \cup (\partial\mathcal{A} \oplus \mathcal{B}) \tag{7.5}$$

Evidently finding the boundary points of the set $\mathcal{A}$ requires an erosion itself, but in case the structuring elements is relatively large or in case the dilation needs to be iterated several times, keeping track of which points in a set need to processed is an often used mechanism to arrive at efficient algorithms.

7.2.3 Reconstruction

In this section we return to the problem of connectivity analysis task to find the connected component of a set that contains a given point $\mathbf{x}_0$.

The dilation using $\mathcal{N}_4$ or $\mathcal{N}_8$ can be used to find the subset of a set $\mathcal{M}$ that is connected to a given point $\mathbf{x}_0$. The set with only the point $\mathbf{x}_0$ is denoted as $\{\mathbf{x}_0\}$. Assume the $\mathbf{x}_0 \in \mathcal{M}$. Finding the connected component of $\mathcal{M}$ that contains $\mathbf{x}_0$ can be done by iteratively dilating the 'seed' while restricting the dilated set to be subset of the 'mask' $\mathcal{M}$:

$$\begin{aligned} \mathcal{S}^0 &= \{\mathbf{x}_0\} \\ \mathcal{S}^{i+1} &= \mathcal{S}^i \oplus \mathcal{N}_c \cap \mathcal{M} \end{aligned}$$

After a finite number of iterations stability is reached in the sense that $\mathcal{S}^{i+1} = \mathcal{S}^i$. We write $\mathcal{S}^\infty$ to denote the stable set.

There is no need to restrict the seed set $\mathcal{S}^0$ to only one point. With the same algorithm we can find the connected component(s) of $\mathcal{M}$ that are connected to a point (or points) in the seed $\mathcal{S}^0$. This is called the *reconstruction of the set $\mathcal{M}$ from the seed (or markers) $\mathcal{S}$.* The reconstruction is denoted as $\gamma_c(\mathcal{M}, \mathcal{S})$ and is defined by:

$$\gamma_c(\mathcal{M}, \mathcal{S}) = \mathcal{S}^\infty \tag{7.6}$$

where

$$\begin{aligned} \mathcal{S}^0 &= \mathcal{S} \\ \mathcal{S}^{i+1} &= \mathcal{S}^i \oplus \mathcal{N}_c \cap \mathcal{M} \end{aligned}$$

The algorithm for the reconstruction that results from a straightforward implementation of the above 'iterate-until-stability' equations is a rather inefficient one. In every iteration, every point in the seed set $\mathcal{S}^i$ is dilated. Eq.(7.5) shows that we only need to dilate the points in the boundary of the seed set. Furthermore the set of points that need to be dilated can be updated quite easily in the process circumventing the need to determine the boundary points using an erosion.

According to Eq.(7.5) we have:

$$\begin{aligned} \mathcal{S}^{i+1} &= \mathcal{M} \cap (\mathcal{S}^i \cup \partial\mathcal{S}^i \oplus \mathcal{N}) \\ &= (\mathcal{M} \cap \mathcal{S}^i) \cup (\mathcal{M} \cap \partial\mathcal{S}^i \oplus \mathcal{N}) \\ &= \mathcal{S}^i \cup (\mathcal{M} \cap \partial\mathcal{S}^i \oplus \mathcal{N}) \end{aligned}$$

i.e. in each iteration of the reconstruction algorithm we add the points in the dilation of the boundary. It is a bit more complex to prove that the boundary need not be calculated in each iteration. In fact we may dilate the difference $\Delta\mathcal{S}^i = \mathcal{S}^i \setminus \mathcal{S}^{i-1}$ instead of the boundary:

$$\mathcal{S}^{i+1} = \mathcal{S}^i \cup (\mathcal{M} \cap \Delta\mathcal{S}^i \oplus \mathcal{N})$$

Based on the above equation a much more efficient reconstruction algorithm can be specified. Now we keep track of the points to be dilated (i.e. the points in the set $\Delta\mathcal{S}^i$ using a queue data structure (FIFO queue). Because a FIFO queue is used we do not have to keep track of separate queues for $\Delta\mathcal{S}^i$ for each value of i.

Listing 7.3: Reconstruction (Queue based)

```
function r = reconstruct( m, s )
% 4 connected binary reconstruction of the mask m from seed s
%
N4ij = [ 1 0; 0 1; -1 0; 0 -1 ]; % the 4 connected neighbors
M = size(m,1); N = size(s,2);
r = s;
[i,j] = find( r ); % find all object points in the seed
q = initQueue;
q = putOnQueue( q, [i j] ); % and put them on the queue

while ~emptyQueue(q)
    [q,ij] = getFromQueue(q);
    for k=1:size(N4ij,1)
        in = ij(1) + N4ij(k,1);
```

```
        jn = ij(2) + N4ij(k,2);
        if inImage(size(m), in, jn ) & r(in,jn)==0 & m(in,jn)==1
            r(in,jn) = 1;
            q = putOnQueue(q, [in jn] );
        end
    end
    %imshow(r); drawnow;
end


function r = inImage( s, i, j )
r = i>=1 & i<=s(1) & j>=1 & j<=s(2);

function r = emptyQueue(q)
r = size(q,1)==0;

function q = initQueue
q = [];

function q = putOnQueue(q, elt)
q = [q; elt];

function [q, elt] = getFromQueue(q)
elt = q(1,:);
q = q(2:end,:);
```

7.2.4 Openings and closings

Due to the fact that erosions and dilations are not inverse operators the sequence of first eroding and then dilating results in a non-trivial change in the original set. In fact the resulting operator is a very important operator that is called an *opening* and is denoted as $\mathcal{A} \circ \mathcal{B}$ and defined by:

$$\mathcal{A} \circ \mathcal{B} = (\mathcal{A} \ominus \mathcal{B}) \oplus \mathcal{B}$$

The opening has a very nice geometrical interpretation as the set that is obtained as the union of all translates of the structuring element that fit entirely within the set $\mathcal{A}$, i.e.:

$$\mathcal{A} \circ \mathcal{B} = \bigcup_{\mathcal{B}+\mathbf{x} \subseteq \mathcal{A}} \mathcal{B} + \mathbf{x}$$

With an opening we thus can 'approximate' the original set using the structuring element as the only building block. The approximation is also required to be a subset of the original set.

The structural (or morphological) opening $\mathcal{A} \circ \mathcal{B}$ is also an opening in the algebraic sense. I.e. it is an operator that has the following three properties:

$$\begin{aligned} (i) \quad & \mathcal{A}_1 \subseteq \mathcal{A}_2 \rightarrow \mathcal{A}_1 \circ \mathcal{B} \subseteq \mathcal{A}_2 \circ \mathcal{B} \\ (ii) \quad & \mathcal{A} \circ \mathcal{B} \subseteq \mathcal{A} \\ (iii) \quad & (\mathcal{A} \circ \mathcal{B}) \circ \mathcal{B} = \mathcal{A} \circ \mathcal{B} \end{aligned}$$

i.e. the opening (i) preserves the ordering of sets, the opening (ii) is a subset of the original set and (iii) the opening is *idempotent*. The fact that the opening is idempotent means that applying

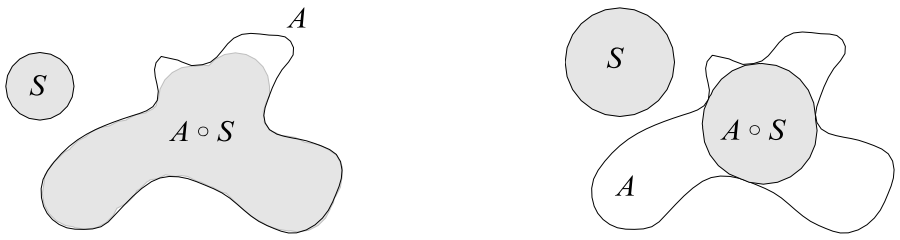


Figure 7.13: **Opening.** The outline of the set $\mathcal{A}$ is drawn and the opening of $\mathcal{A}$ with respect to structuring element $\mathcal{S}$ is indicated in grey (on the left). On the right the opening with respect to a larger structuring element.

an opening once is enough, repeating it will not change the result (that is of course quite different with the erosion and dilation).

In case we first dilate and then erode, we construct the closing:

$$\mathcal{A} \bullet \mathcal{B} = (\mathcal{A} \oplus \mathcal{B}) \ominus \mathcal{B}$$

This operation is most easily interpreted using the duality with respect to the complement property:

$$\mathcal{A} \bullet \mathcal{B} = (\mathcal{A}^{\mathsf{c}} \circ \mathcal{B}^{\star})^{\mathsf{c}}$$

In the closing the background is approximated as the union of all translated structuring elements $\mathcal{B}^{\star}$ that fit completely within the background. The closing has the following three properties:

$$\begin{aligned} &(i) && \mathcal{A}_1 \subseteq \mathcal{A}_2 \rightarrow \mathcal{A}_1 \bullet \mathcal{B} \subseteq \mathcal{A}_2 \bullet \mathcal{B} \\ &(ii) && \mathcal{A} \bullet \mathcal{B} \supseteq \mathcal{A} \\ &(iii) && (\mathcal{A} \bullet \mathcal{B}) \bullet \mathcal{B} = \mathcal{A} \bullet \mathcal{B} \end{aligned}$$

The example with which we started this chapter may serve as a nice example of using openings and closings to analyze the shapes found in an image. A sequence of a closing to fill the small holes in the ball and of an opening to remove the relatively small details in the background results in a near perfect shape representing only the ball.

7.3 Shape

7.3.1 Representation of Shape

In a previous section we have seen two ways to represent a two dimensional set (or shape): either as the collection of points that are within the set, or as its indicator function. For the purpose of shape measurements there are more ways to represent shape.

Both representation methods we have used before, explicitly encode the interior of the sets, i.e. its area (or volume in 3D). Another way to represent a set is to encode its boundary. In the continuous plane $\mathbb{R}^2$ the boundary of a set $\mathcal{S}$ is the set of all points whose ϵ-neighborhood contains both points in $\mathcal{S}$ and $\mathcal{S}^{\mathsf{c}}$.

On the discrete square grid using either $\mathcal{N}_4$ or $\mathcal{N}_8$ as the ϵ-neighborhood, this leads to 'two-pixel-thick' boundaries. In practice most often the 'inner' boundary of a set $\mathcal{A}$ is chosen $\partial\mathcal{A} = \mathcal{A} \setminus \mathcal{A} \ominus \mathcal{N}_x$.

Instead of enumerating all points in the boundary of a set, it is customary to encode a boundary with *Freeman vectors*. These vectors point from one point on the (discrete) contour to

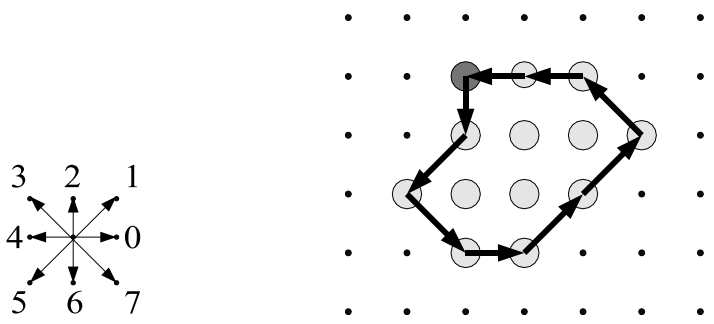


Figure 7.14: **Freeman Codes.** On the left the encoding of the Freeman vectors. On the right a set with the Freeman encoded contour. The code string (starting in the dark grey shaded point) is 6 5 7 0 1 1 3 4 4.

the next point. The contour is effectively polygonally approximated using a fixed set of possible line elements. Fig. 7.14 shows the standard encoding of the 8 possible vectors in a Freeman encoding. Also shown is a discrete set with its associated Freeman code string. For a simply connected object the Freeman code string provides a complete characterization of the shape. Its position on the plane is lost though.

Freeman codes provide a powerful way of representing discrete shapes on the plane. It is not without disadvantages: besides the fact that due to the fact that we code the inner boundary there is a bias to make objects appear smaller than they are in reality, the representation depends on the discretization grid. If we would make a picture of the same shape with a more refined camera (doubling the resolution say) the number of Freeman codes in the representation would roughly double as well. Comparing rectangles independent of their size is therefore not a trivial task based on a Freeman code representation.

Starting with a Freeman encoding of a shape we can make a more resolution independent representation by approximating the boundary with straight lines. An optimal algorithm to subdivide a Freeman code string into a resolution independent polygonal approximation is not a trivial task. Here we present a simple algorithm for the polygonal approximation of curves with a start point and an endpoint (open curves).

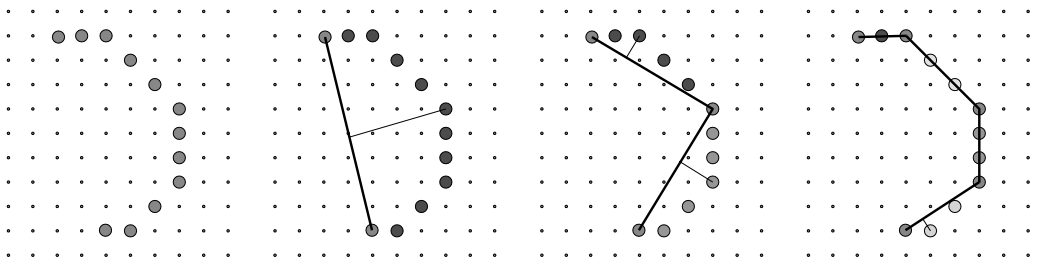


Figure 7.15: **Polygonal Approximation of Curves.** In a few number of steps the curved (discrete) line is approximated by straight lines.

Let $L = \{p_1, \ldots, p_N\}$ be the collection of N points on a discrete curve. The sequence of points is given by the Freeman convention. The sub code string starting p_i and ending in p_j is denoted as $[p_i, p_j]$. Assume we are subdividing this interval $[p_i, p_j]$ then we look for the point p_k with $i \leq k \leq j$ such that the distance from p_k to the straight line connecting p_i and p_j is maximal. In case the distance is larger than a predefined constant δ then the curve $[p_i, p_j]$ is subdivided

in the two curves $[p_i, p_k]$ and $[p_k, p_j]$. Then we apply the same procedure to the two curves just obtained. We do this recursively until no curve has a point on it with distance larger than δ from the straight line connecting its start and end point. The recursive algorithm is evidently started by subdividing the entire curve. The recursive subdivision is illustrated in Fig. 7.15.

The recursive straight line approximation described above is based on the a priori assumption that the contour is indeed build from straight line segments (or at least can be accurately approximated with straight lines). Less a priori knowledge is assumed in case the boundary is approximated with a parameterized curve (e.g. a spline curve).

Another representation of sets that is of great practical importance is the *run length encoding* of sets. This representation codes the consecutive pixels in a row that are in the set. Such a 'run' of pixels is encoded with the start position and the end position. The encoding is thus independent of the length of the run. Especially for long runs this encoding becomes very efficient. A well-known example of run length encoding in practice is the fax standard. Black and white scans of documents are run length encoded.

7.3.2 Shape Measurements

Given a silhouette of an apple. How would you compare this shape with the silhouette of another apple (to see whether the apples are of the same variety for instance) or how would you tell that one shape is an apple and the other one is a pear? Or to state the question boldly: what *is* the shape of an object. We are not going to answer this question here because we believe that there is no answer. The question is ill-posed.

All we can do is say things like: the apple silhouette is almost round, or the apple silhouette is about 20 square centimeters big, or ... etc. I.e. we can do a lot of measurements on the shapes we study and we can only 'know' the shapes through these quantities.

Given the fact that only an operational definition for shape is possible, it is no surprise that many, many different shape measurements are known and used in practice. In this section we discuss very few of all possible shape measures.

Area and Length Measurements

What could be more simple than to calculate the area of a set, the length of its perimeter and its bounding box? Of these simple shape measures the last two are surprisingly difficult to answer.

The area of a set is indeed a simple measure. We have seen the area already in the chapter on statistical measures where we have seen that the integral over a (indicator) function is approximated with the sum of its discrete representation. The area $A(\mathcal{S})$ is therefore nicely approximated[3] with the count of the number of samples in the set (just count the pixels) assuming that the area of the elementary square around each sample point is equal 1.

The length of the perimeter of a set $L(\mathcal{S})$ is a troublesome notion. Not in the continuous plane of course, but once the set is discretized and we aim for estimating the length of the perimeter given a discrete representation of the boundary (the Freeman code string). If we measure the length of the polygonal approximation we obtain:

$$L(\mathcal{S}) = n_e + n_0\sqrt{2}$$

where n_e is the number of even Freeman codes in the string and n_0 is the number of odd Freeman code strings. Note that the odd codes correspond with the diagonal steps of length $\sqrt{2}$ in a square sampling grid of length 1.

[3]The term "nicely approximated" can be given a more objective meaning: the area measure is converging to the real area in case the sampling distance is made smaller and smaller

Because of the "staircase" approximation of smooth lines in the continuous plane it should not be too surprising that the above formula to calculate the perimeter of a set in general will give an over estimation. To correct for this a better estimator is:

$$L(\mathcal{S}) = 0.9481(n_e + n_o\sqrt{2})$$

Still a better estimator than the one above is possible:

$$L(\mathcal{S}) = 0.980n_e + 1.406n_o - 0.091n_c$$

where n_c is the number of corners in the Freeman polygonal approximation. The math needed to arrive at these results is beyond the scope of this lecture series. The starting point in the math analysis is to look for all possible sets in the continuous plane that through sampling will result in the given discrete boundary. A probability weighted average over all possible continuous perimeter length values then is the most probable outcome for the discrete perimeter length.

An inherent problem with measuring length of discrete curves is that curves in the continuous plane are sets of zero area (volume). These sets cannot be faithfully represented on the discrete grid. Therefore more modern length estimators are based on ways to cast length measurements in terms of area or volume measurements.

It is not hard to see that for smooth sets on the continuous plane(i.e. sets with boundary curvature less than or equal to d along the entire boundary) we have for small d:

$$L(\mathcal{S}) \approx A\left(\mathcal{S} \setminus \mathcal{S} \ominus d\mathcal{B}\right)$$

where $d\mathcal{B}$ is the disk of radius d. Unfortunately for good approximations the radius d should be rather small. But in that case the disk $d\mathcal{B}$ cannot be faithfully sampled on the discrete grid and the above approximation is a poor one again. Effectively we are then just counting the pixels in the discrete boundary: an estimator that performs worse than the worst estimator based on the Freeman code string presented above.

The basic idea of measuring area instead of length is not lost though. The set $\mathcal{S}$ of which we want to measure the perimeter length, is most often the level set (i.e. the result of thresholding) of some scalar (grey value) image. It proves that we are able to do the measurements in the scalar image instead of in the result of thresholding the image.

Moment Analysis

Let $\mathcal{S}$ be a sampled set with indicator function s. The area (or volume in 3D) of the set is given by:

$$A(\mathcal{S}) = \sum_{\mathbf{i,j}} s(i,j)$$

assuming the sampling grid has sampling distances equal to one. This is the zero order moment of the ensemble of all the points in the set as we have already encountered in a previous chapter.

The two first order moments of the set $\mathcal{S}$ provide a indication of what the position of the set is in the plane:

$$\mathbf{p}(\mathcal{S}) = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \frac{1}{A(\mathcal{S})} \begin{pmatrix} \sum_{i,j} i\, s(i,j) \\ \sum_{i,j} j\, s(i,j) \end{pmatrix}$$

The vector $\mathbf{p}$ is called the *center of mass* of the set $\mathcal{S}$. And for good reason: if you would cut the shape of the set out of sheet of metal and balance it on the top of your finger you would have to put your finger in the center of mass to keep it in perfect balance[4] The center of mass is

[4]It should be noted that the center of mass need not be a point in the set itself.

of course nothing else then the mean of the vectorial ensemble of all point coordinates $(i\ j)^\mathsf{T}$ in the set $\mathcal{S}$.

The covariance matrix of this ensemble can be used to calculate an orientation of the set $\mathcal{S}$. In the notation using the indicator function s we get:

$$C = \frac{1}{A(\mathcal{S})} \begin{pmatrix} \sum_{i,j}(i-p_1)^2\, s(i,j) & \sum_{i,j}(i-p_1)(j-p_2)\, s(i,j) \\ \sum_{i,j}(i-p_1)(j-p_2)\, s(i,j) & \sum_{i,j}(j-p_2)^2\, s(i,j) \end{pmatrix}$$

The eigenvectors of this matrix are indicators of the orientation of the set $\mathcal{S}$ and the relative magnitude of the eigenvalues are indicative of the elongation of the set.

7.4 Exercises

1. **Set operators.**

   (a) The set containing all elements in $\mathcal{A}$ but not in $\mathcal{B}$ is denoted as $\mathcal{A} \setminus \mathcal{B}$. Express the *set difference* in terms of the fundamental set operators.

   (b) The set containing all elements that are in either $\mathcal{A}$ or in $\mathcal{B}$ but not in both is called the *symmetric set difference* of $\mathcal{A}$ and $\mathcal{B}$ and is often denoted as $\mathcal{A}\Delta\mathcal{B}$ Express the *symmetric set difference* in terms of the fundamental set operators.

   (c) Let a and b be the Boolean indicator functions of the sets $\mathcal{A}$ and $\mathcal{B}$ respectively. Every set operator corresponds with a Boolean function. Fill in the table below:

Set Operator	Boolean Operator	Matlab Operator
$\mathcal{A} \cup \mathcal{B}$	OR	or(A,B)
$\mathcal{A} \cap \mathcal{B}$		
$\mathcal{A}^c$		
$\mathcal{A} \setminus \mathcal{B}$		
$\mathcal{A}\Delta\mathcal{B}$		

2. **Set inclusion.**

   Given two binary images a and b being the indicator functions of the sets $\mathcal{A}$ and $\mathcal{B}$, give a (one-liner) Matlab expression that implements the inclusion relation $\mathcal{A} \subseteq \mathcal{B}$. Test your solution on the four canonical possibilities: $\mathcal{A}$ and $\mathcal{B}$ completely disjoint, $\mathcal{A}$ and $\mathcal{B}$ partly overlapping, $\mathcal{A}$ a subset of $\mathcal{B}$ and $\mathcal{B}$ a subset of $\mathcal{A}$. (Hint: look at the documentation of the Matlab functions `all` and `any`).

3. **Morphological operators in GIS.**

   Consider a geographical information system containing maps of some rural area with a lot of camping grounds and a river that flows through the area. Let $\mathcal{C}$ be the set of connected components each of them corresponding with a camping ground. Let $\mathcal{R}$ be the set corresponding with the river.

   - Give a morphological expression to find all camping grounds that are within d kilometers from the river.
   - Most often the spatial data in a GIS is represented with polygons. Algorithms for erosion and dilation working on polygonal spatial data structures are not easy to specify. Can you think of an algorithm to do a dilation with a disk shaped structuring element? (Hint: the border of a set is 'translated' parallel to itself).

4. **Dilation Algorithms.**

   Compare the two dilation algorithms Algorithm 7.1 and Algorithm 7.2. Which one has lowest computational complexity? Simply count the number of pixels that have to be addressed in computer memory. Is it dependent on the image contents?

Chapter 8

Grouping

8.1 Grouping Regions: Magic Wand

The basic idea in grouping points into meaningful subsets of the spatial domain is to start with a seed point and then find all points that are connected to that point and also satisfy a homogeneity constraint, i.e. all points in the region should have the same perceptual properties.

The simplest form of grouping points into regions is *region growing* as discussed in this section. In Section 8.2 we discuss the *watershed segmentation* algorithm that is closely related to region growing but is a much more powerful technique in practice.

We consider only grey value images $f : \mathbb{R}^2 \to \mathbb{R}$ in this section (the extension to color images is discussed in one of the exercises). The goal now is to find the *connected region* $\mathcal{A} \subset \mathbb{R}^2$ that contains the *seed point* $\mathbf{x}_0 \in \mathbb{R}^2$ such that for all points $\mathbf{x} \in \mathcal{A}$ the grey value difference compared with the grey value in the seed point is less than or equal to a threshold value Δ, i.e. $|f(\mathbf{x}) - f(\mathbf{x}_0)| \leq \Delta$.

An algorithm to find the subset $\mathcal{A}$ is easy to formulate. First we construct the subset $\mathcal{B} \subset \mathbb{R}^2$ of all points with a grey value difference less than or equal to Δ:

$$\mathcal{B} = \left\{ \mathbf{x} \in \mathbb{R}^2 \;\middle|\; |f(\mathbf{x}) - f(\mathbf{x}_0)| \leq \Delta \right\}.$$

From this set $\mathcal{B}$ we have to select the subset $\mathcal{A}$ of all points that are connected to the point $\mathbf{x}_0$. This is exactly what the *reconstruction operator* γ_c (discussed in Section 7.2.3) does:

$$\mathcal{A} = \gamma_c(\mathcal{B}, \{\mathbf{x}_0\}).$$

It was argued there that the reconstruction operator for sampled sets is based on iterating conditional dilations until stability. The structuring element reflects the chosen connectivity of the set $\mathcal{A}$ (i.e. either the 4 connectivity or the 8-connectivity in case of a rectangular sampling grid).

Reconstruction can be efficiently implemented as a queue based algorithm. Algorithm 8.1 combines the queue based reconstruction algorithm (see Algorithm 7.3)with the construction of the set $\mathcal{B}$ of all points with a grey value difference less than or equal to Δ. The queue functions are the same as used before.

Region growing is a simple segmentation technique. Most (if not all) commercial image processing programs (like Adobe Photoshop and Paint Shop Pro) implement region growing under the name *magic wand*. The simplicity comes at a price though. The results of region growing can be less than satisfactory due to the facts that:

Listing 8.1: Region Growing (Magic Wand)

```
function lbl = magicwand( im, i0, j0, delta )
% region growing (4 connected) from point (i0,j0)
%
N4ij = [ 1 0; 0 1; -1 0; 0 -1 ]; % the 4 connected neighbors

lbl = zeros(size(im));           % initialize label image

q = initQueue;                   % initialize the queue
q = putOnQueue(q, [i0 j0] );     % put the starting point on the queue

lbl(i0,j0)=1;                    % by definition in the labeled set

while ~emptyQueue(q)
    % get the first point (i,j) from the queue
    [q, ij] = getFromQueue(q);

    for k=1:size(N4ij,1)
        in = ij(1) + N4ij(k,1);
        jn = ij(2) + N4ij(k,2);
        if inImage(size(im), in, jn ) & ...
         lbl(in,jn)==0 & ...
         abs( im(in,jn) - im(ij(1),ij(2)))<=delta
            lbl(in,jn) = 1;
            q = putOnQueue(q, [in jn] );
        end
    end
end
```

- the result is very sensitive to the choice of a seed point,
- the result is very sensitive to the choice of the Δ value, and
- the result is not *locally stable* in the sense that only a very small 'bridge' between two regions of similar grey values will select both regions.

Many generalizations of region growing techniques have been proposed in the image processing literature.

8.2 Grouping Regions: Watershed Segmentation

Watershed segmentation is a segmentation technique that was developed in a mathematical morphological framework and is used very often in practical image processing application.

In this chapter a very brief introduction to the watershed segmentation algorithm is given. A somewhat simplified algorithm is presented that is much like the region growing algorithm that has been introduced in a previous section. For a more comprehensive introduction to the watershed segmentation procedure described within the morphological framework we refer to the work of Vincent and the book of Soille.

In the queue based algorithm for region growing we grow the seed point until no more points to be added to the region satisfy the homogeneity constraint. Watershed segmentation starts

with as many seeds as the number of objects we would like to distinguish in an image. Not only the actual objects should be marked but also the background should be marked.

Then we start to grow all seeds. Instead of using a homogeneity constraint that stops regions from growing, in the watershed algorithm regions are grown until they touch each other. Regions are not allowed to 'overtake' points already assigned to a region. When the image domain is completely filled the segmentation procedure stops. In the growing process we also keep track of the region that points belong to. The result of the growing process is a partitioning of the image domain where we have one region per original seed.

Whereas in the region growing algorithm we didn't need a way to prioritize which points to 'grow', in the watershed algorithm there are several competing regions and therefore we need to be able to tell which point is allowed to grow first. We do so by introducing a *dissimilarity measure* $d : \mathbb{R}^2 \to \mathbb{R}^+$. The value $d(\mathbf{x})$ is proportional to the dissimilarity of the point $\mathbf{x}$ with its surrounding neighbors. Of all the points that are allowed to grow, the one with the lowest dissimilarity measure has highest priority.

An obvious choice for a dissimilarity measure is the edge strength defined as the gradient norm f_w. For a large edge strength the points in the neighborhood are quite dissimilar. Using Gaussian derivatives of a suitable scale also allows us to 'zoom in' on the level of detail that we are interested in.

In comparison with the region growing algorithm the ordering of points to grow according to the dissimilarity measure forces us to replace the first-in-first-out queue with a *priority queue*. Algorithm 8.2 defines the priority queue data type[1]

Listing 8.2: Priority Queue

```
function r = emptyPQueue( q )
% test whether PQueue is empty
r = size(q,1)==0;

function q = initPQueue
% initialize a PQueue
q = [];

function q = putOnPQueue( q, elt)
% put an element on the PQueue
% such that the elt's are sorted according to elt(1)
q = [q; elt];
q = sortrows( q, 1 );

function [q, elt] = getFromPQueue( q )
% retrieve a triple (i,j,v) from the queue
% this is guaranteed to be the triple with
% lowest v-value
elt = q(1,:);
q = q(2:end,:);
```

Assume that an image marker has value marker(i,j)=0 in case the point (i,j) is *not* a seed

[1]It should be noted that the given Matlab implementation is a very simplistic one. It would be much better to implement a *priority heap*. Unfortunately Matlab is not specifically suited for implementing abstract data types beyond the arrays. Low-level image processing in an interpreted language is of course not a good idea anyway. The code only serves as educational tools.

point and value marker(i,j)=v for seed points where v is the identifying label for the seed point (and thus for the regions that are grown from the seeds).

Listing 8.3: Watershed Segmentation

```
function lbl = mywatershed( d, marker )
% watershed from markers
% d: image of dissimilarity measure
% marker: image containing initial seeds
M = size(d,1); N = size(d,2);

% first get seed points from marker image
% and put them on the queue (with highest priority)
q = initPQueue;
[i, j] = find( marker );
lbl = marker;
q = putOnQueue( q, [zeros(size(i)) i j] );

% get points from queue until it is empty
while ~emptyPQueue(q)
    [q, pij] = getFromPQueue( q );
    p = pij(1); i = pij(2); j = pij(3);

    % loop over all points (k,l) in the neighborhood of (i,j)
    for k=1:size(N4ij,1)
        in = i + N4ij(k,1);
        jn = j + N4ij(k,2);
        if inImage(size(im), in, jn ) & lbl(in,jn)==0
            lbl(in,jn) = lbl(i,j);
            q = putOnQueue(q, [p in jn] );
        end
    end
end
```

The watershed segmentation code in Algorithm 8.3 can be used to segment an image in case the markers (seed points) are given. Obviously just as in the region growing algorithm we can specify these interactively (note that the watershed algorithm needs markers for the objects *and* for the background). The markers should have a label value that is distinctive for the objects to be found in the image.

The presented Matlab function for the watershed segmentation is not very efficient (to say the least). The priority queue implementation is very naive; the entire queue is sorted to insert only one element. A *heap* algorithm would be a far better choice. Even with excellent choices of the data structures and of the algorithms to be used, any implementation in the interpreted Matlab language is bound to be slow. Too slow for practical purposes.

Consider again the image showing the four coins. With a marker in each of the coins and only one marker in the background a watershed segmentation using the presented algorithm results in the image in Fig. 8.1. The Matlab code to generate these images is shown in Algorithm 8.4.

It would of course be advantageous to have an automated way of selecting useful markers for the watershed segmentation. The classical marker selection method is to select all local minima in the dissimilarity image as markers. This most often results in *over segmentation* due to the very many local minima. Only with the selection of the prominent minima, the watershed

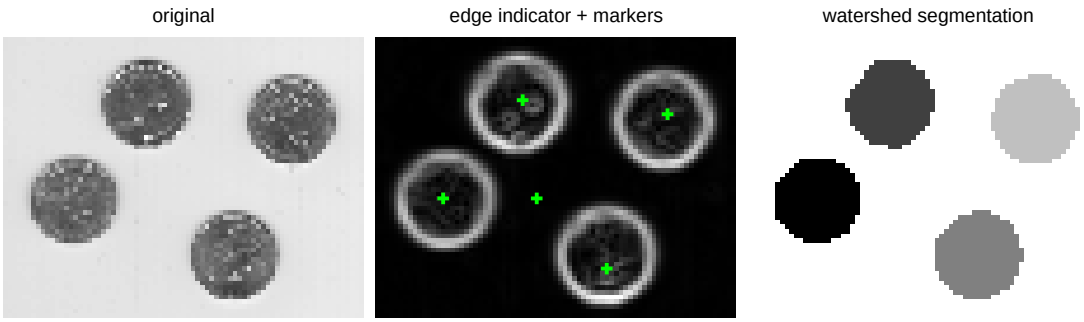


Figure 8.1: **Watershed Segmentation.** On the left the original image is shown. In the middle the edge strength indicator (the gradient norm) with the selected markers. And on the right the resulting segmentation.

segmentation algorithm becomes the robust and often used tool for image segmentation that it is.

8.3 Grouping Lines: Hough Transform

After you have applied some detector of local structure to an image, you end up with an image of 'detection strengths'. For instance, after you have applied a Canny edge detector to an image you get pixels where edges reside, as in Figure 8.2a. You can clearly see the straight lines that form the edges of the blocks. But this image is *not* an image of edges; it is just an image of candidate edge points (and their intensity may be interpreted loosely as 'the likelihood that the point belongs to some edge'. We are not yet capable of determining, say, the length of the top-right edge of some block: that edge is not yet a data structure we can operate on.

Somehow, we need to collect these candidate points into lines. This is a specific instance of a general problem: given a set of points, collect them into groups matching a prescribed model. In our case, the desired model is a straight line segment, but we could also have asked for circles or other shapes. We need a general method to solve such problems, for they occur often.

Permutational methods are out if the question. There are very many points in the edge image of Figure 8.2a which are non-zero, and to match all of those with each other makes for a huge number of combinations (any two points determine a line, and then you have to see whether a sufficient number of other points are close to this line – for each point pair). So we need a way of structuring the search preventing such combinatorics. We do this through a clever method based on a parametrization known as the *Hough transform*. Let us call it 'searching by voting'.

8.3.1 The parameter space of a Hough transform

The Hough transform method is going to work as follows: we are going to transform the image with edge points of Figure 8.2a to a parameter domain of straight lines – this gives the vague curvy lines in Figure 8.2c. Then in that domain we select significantly present parameter combinations (denoted by the circles in that figure), and transform those back. This gives the carriers of the edge segments as in Figure 8.2b. In a further processing step we can then select the significant portions of those lines.

In this section we focus on the core of the Hough method: the parameter space selection. We do this by giving some examples of such spaces.

Listing 8.4: Watershed Segmentation Example

```
% watershedfigure
a=imread('eight.tif'); a=im2double(a);
a=imresize(a,1/4);  % else it would take forever
b=a>0.8; g = sqrt(gD(a,1,1,0).^2+gd(a,1,0,1).^2); s =
zeros(size(a));
s(35,35)=6; % background marker
s(35,15)=2; s(14,32)=3; s(50,50)=4; s(17,63)=5;
w=mywatershed(g,s);

subplot('position',[0,0,1/3.1,1]); imshow(a); title('original');

subplot('position',[1/3,0,1/3.1,1]); imshow(g,[]); hold on;
plot(35,35,'+g','LineWidth',2);
plot(15,35,'+g','LineWidth',2); % note reversal of i,j coordinates
plot(32,14,'+g','LineWidth',2); plot(50,50,'+g','LineWidth',2);
plot(63,17,'+g','LineWidth',2); hold off; title('edge indicator + markers');

subplot('position',[2/3,0,1/3.1,1]); imshow(w,[]);
title('watershed segmentation');
```

1. *lines: point-to-*line *Hough transform*
   For instance, if you are searching for (infinite) lines, then you might want to characterize them using the familiar line equation in (x, y)-coordinates:

   $$y = a\,x + b \tag{8.1}$$

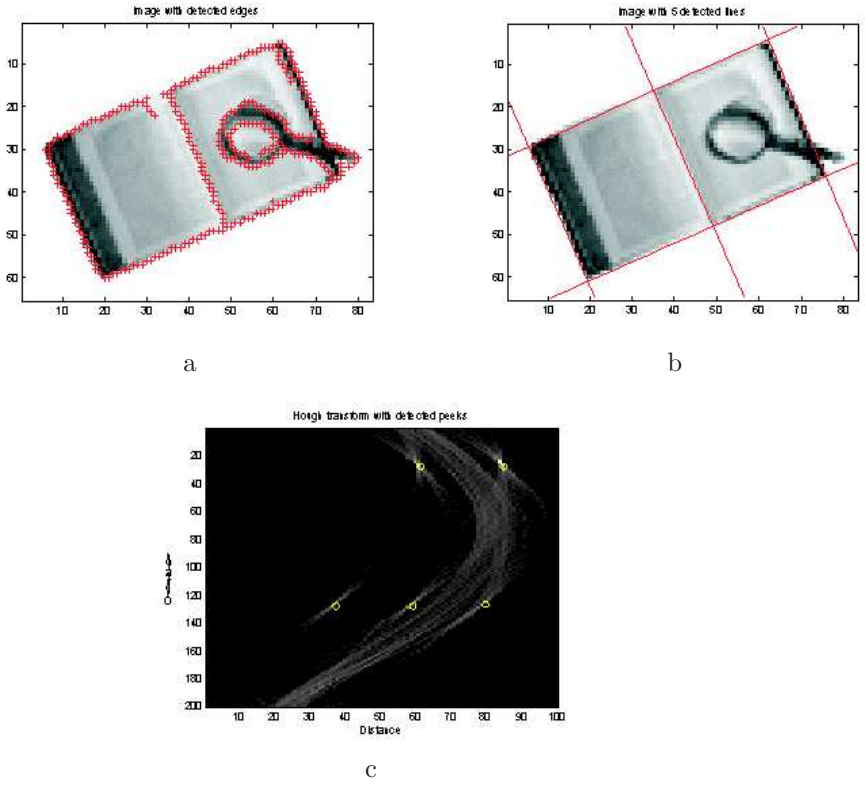
   Note that a line is characterized by a (the slope) and b (called the 'intercept' since it is the value of y where the line intercepts the y-axis). A tuple (a, b) therefore codes for a line.

   If we vary a and b slightly, we get other lines, close to the original line. Therefore the tuples (a, b) span a space of points, each characterizing a line in (x, y)-space. We will use sans serif font for the (a, b)-space entities, and regular font for (x, y)-space entities. So a point represents a line. Conversely, a point (x_0, y_0) in (x, y)-space gives by eq.(8.1) a linear relationship between a and b: this is a line in (a, b)-space, see Figure 8.3(column 1). All points on this line are lines going through the point (x_0, y_0).

   If have a lot of points in (x, y)-space lying on a line, those will be represented in a lot of lines going through a single point, see Figure 8.3(column 2). We will make use of that in step 2 (Section 8.3.2).

   This representation of a line is very straightforward, but it has a disadvantage: it is not *isotropic*, i.e. not all directions of lines are equally tractable. Lines that are (almost) along the y-axis require huge a-values. Often people divide the parameter domain in parts representing lines with orientations up to 45 degrees using the $y = a\,x + b$ formula; and for lines with orientations larger than 45 degrees they switch to a second parameter domain using $x = a'\,y + b'$. This is a bit clumsy; it is better to consider a more isotropic line representation that can handle all lines in the same way. The following two representations have that property.

2. *lines: point-to-*circle *Hough transform*
   Another way of representing a line is by using a support vector. A vector $\mathbf{s}$ in 2-dimensional



a b c

Figure 8.2: **Hough transform method:** the detection of lines (b) in a image of edge points (a) indicated by the crosses (better visible in the color version of this picture), using the point-to-sine Hough transform (c). Images from the site `www.s2.chalmers.se/~ghassan/phd/illus/linehough/`, which also contains Matlab code for Hough transforms.

space determines a line perpendicular to it, passing through its tip, see Figure 8.3(column 3). This is a unique line, determined by the equation:

$$\mathbf{x} \cdot \mathbf{s} = \mathbf{s} \cdot \mathbf{s} \tag{8.2}$$

We have therefore 'parametrized' the line by a vector $\mathbf{s}$. (Again we use sans serif for the representing entities.) The vectors characterizing lines in this way form a space. (We cannot really call it a vector space, since we do not have a meaning for the sum of two such representative vectors; what is the sum of two lines?)

All lines going through a fixed point $\mathbf{z}$ are represented by a circle of vectors in the representative space. This is most easily seen by writing $\mathbf{z} \cdot \mathbf{s} = \mathbf{s} \cdot \mathbf{s}$ out into components:

$$z_1 s_1 + z_2 s_2 = s_1^2 + s_2^2$$

so that

$$(s_1 - \tfrac{1}{2} z_1)^2 + (s_2 - \tfrac{1}{2} z_2)^2 = \tfrac{1}{4}(z_{01}^2 + z_{02}^2)$$

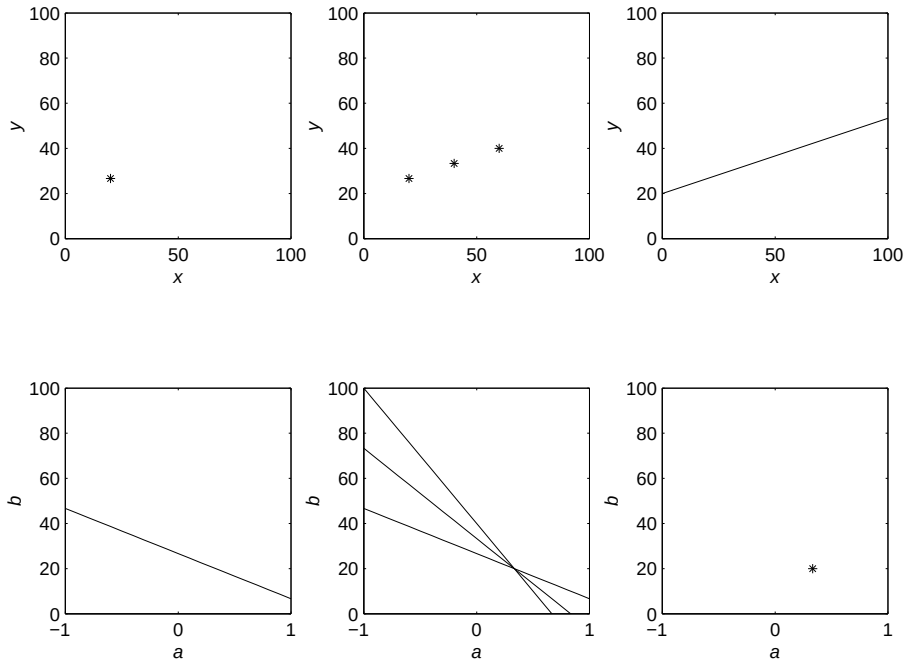


Figure 8.3: **Hough transform:** point-to-line.

and this is the equation of a circle in the (s_1, s_2)-plane. We have drawn it in Figure 8.4(column 1); verify that you understand what center and radius are, by observing that the resulting equation can be rewritten as $|\mathbf{s} - \frac{1}{2}\mathbf{z}| = |\frac{1}{2}\mathbf{z}|$. (Can you prove this directly from eq.(8.2)? That is the better way to get it!)

If we have a lot of points on the same line, this should be represented by a lot of circles through the same point in the representation space; the vector to this point, and the corresponding line have been sketched in Figure 8.4(column 3). There is a second point all circles pass through: the point at the origin. (Question: Why – what does this point represent, and why is that in common to all lines?)

3. *lines: point-to-*sine *Hough transform*
Yet another possibility is to characterize this vector $\mathbf{s}$ by its norm $r \equiv |\mathbf{s}|$ and its angle ϕ relative to some fixed direction, as $\mathbf{s} = (r\cos\phi, r\sin\phi)$. Then you describe lines as points in (r, ϕ)-space. All lines through a point $\mathbf{z}$, defined by $\mathbf{z} \cdot \mathbf{s} = \mathbf{s} \cdot \mathbf{s}$, are now represented by sinusoidal functions in (r, ϕ)-space:

$$z_1 \cos\phi + z_2 \sin\phi = r$$

so, setting $\alpha = \text{atan2}(z_2, z_1)$,

$$r = |\mathbf{z}| \cos(\phi - \alpha).$$

This is the sinusoidal representation depicted in Figure 8.5(column 1). If you now have points lying on a line, they generate a bunch of sinusoids. These sinusoids intersect in two points, as you see in Figure 8.5(column 2). Either of these points is a representation of the original line (see column 3) – make sure you understand why.

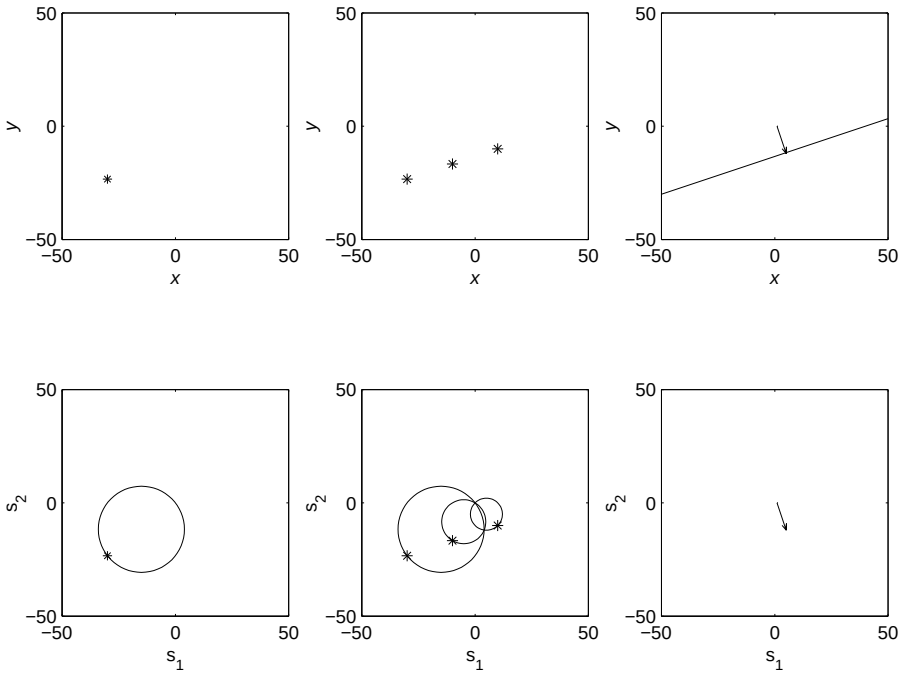


Figure 8.4: **Hough transform:** point-to-circle.

(Question: It is customary in this Hough transform to choose your image coordinate origin in the center of the image. What is the maximum value r can have? Make sure you take the transformation domain big enough – has that been done in Figure 8.5?)

If you want to get a better feeling for this Hough transform, try the applet at `www.vision.ee.ethz.ch/~jhug/MoseIadar`

4. *circles: point-to-*cone *Hough transform*
   If the shape you want to detect is a circle rather than a line, you need a parametrization of the circle. An obvious one is through its *center* $\mathbf{c}$ and *radius* ρ. The circle equation is

$$|\mathbf{x} - \mathbf{c}| = \rho$$

   or in coordinates:

$$(x_1 - c_1)^2 + (x_2 - c_2)^2 = \rho^2 \tag{8.3}$$

   There are *three* real-valued parameters in this equation, so a circle has a 3-dimensional parameter space. In (c_1, c_2, ρ)-space, the circle is represented as a point. The set of circles through a specific point in (x, y)-space is found by considering eq.(8.3) for fixed (x_1, x_2). It is then the equation of a cone in (c_1, c_2, ρ)-space: for every ρ you get a circle in (c_1, c_2) with center (x_1, x_2); see Figure 8.6. If you are searching for circles with a certain radius R, you only need to consider the plane determined by $\rho = R$.

So a parametrization is not really fully determined by the object you want to detect, there is some freedom of choice. Each has its own advantages and disadvantages. For lines, the point-to-sine method is used most.

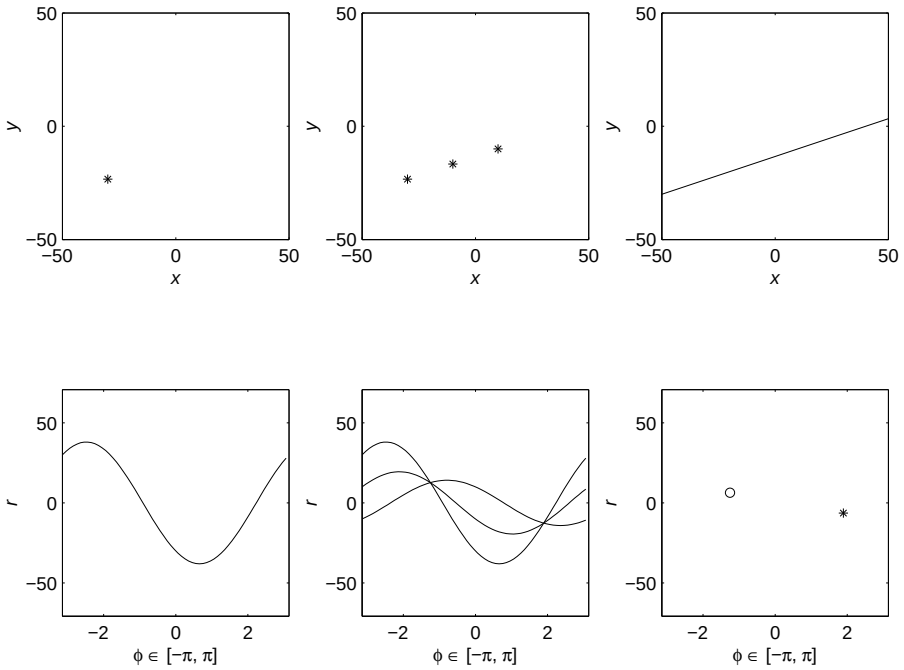


Figure 8.5: **Hough transform:** point-to-sine.

8.3.2 The Hough transform method

We now describe the steps in the Hough transform method in some detail. We need a general term for the shape in the parameter space that is the transformation of a point using the appropriate formulas; let us call that a 'Hough-stamp'.

Step 1: Edge point image

First you need an image containing the candidate points which need to be collected into boundaries. Typically, this image is the outcome of some edge detector. A common input is the image formed by applying the Canny edge detector (see Chapter 4).

Step 2: voting

Now transform each point in the edge image to the parameter space. There, the line, circle, sinusoid or cone of points corresponding to the image point can be considered as a 'vote' for those points. In general, each point in the edge image leads to a Hough-stamp in the parameter space, weighted by the intensity of the edge point (since that can be considered as a measure of how likely the point is to be an actual edge point). Doing this for all edge points and accumulating the results should lead to particular points standing out in the parameter space.

To perform this voting in a computer program, you should divide the parameter space into bins, and accumulate the 'hits' of the Hough-stamps. There is a practical issue here of density: the Hough-stamp in parameter space due to a single point is very thin and so represents little density. So how should you fill the bins? A good solution is to remember that an edge point is really a pixel, in itself representing a certain collection of points, in a region about the size of

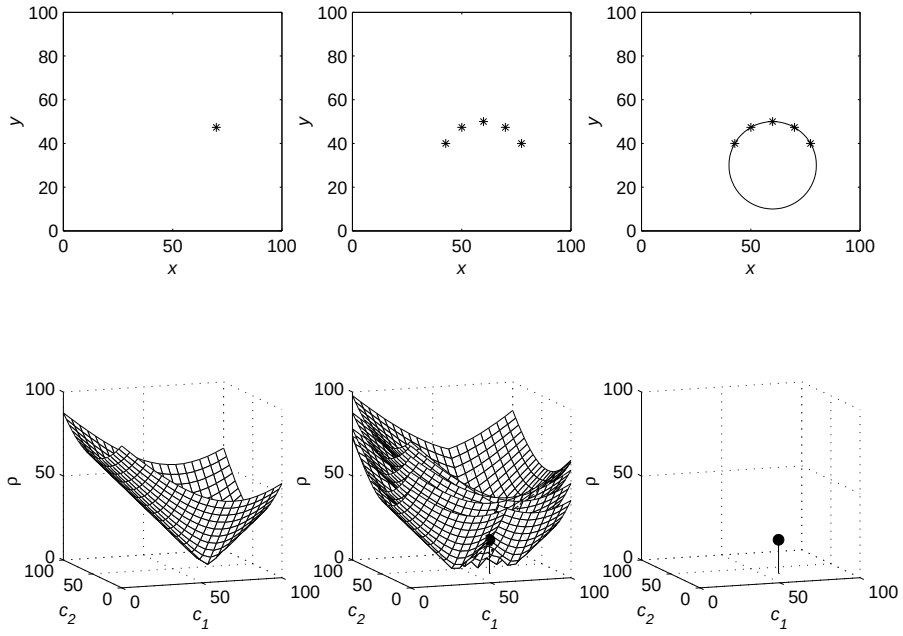


Figure 8.6: **Hough transform:** point-to-cone.

the pixel separation (or more, depending on the scale used to compute the edges). So we should really count the transformation of this ensemble of points; that leads to a distribution around the *Hough stamp* permitting accumulation.

This 'thickening' again depends on the actual transformation, and you could compute it precisely. For instance, in the case of the point-to-line Hough transform, it leads to a hyperbolic widening of the line, by a certain profile which depends on the sensitivity area of the original pixel. In a good approximation, you can make it a Gauss-like profile of width around the line which grows with the a-parameter (see exercise below).

The Matlab code for this type of Hough transform then is:

```
function ht = hough (im, a, b, scale)
% Hough transform for lines-to-lines
%  im  - input image of potential edge points
%  a,b - output image indices
%  scale - size of sensitivity region of input pixel
[x,y]      = find(im);   % all non-zero points
N          = size(x); [dummy,sa] = size(a); [dummy,sb] = size(b);
ht = zeros(sb,sa); for i=1:N
    ht = ht + writeline( a, b, x(i), y(i), im(x(i),y(i)) );
end


function line = writeline( a, b, x0, y0, intensity)
% Gaussian-weighted line of proper width around Hough stamp
[A,B] = meshgrid(a,b); line = intensity * 2./cosA .*
exp(-((A*x0+B-y0)./cosA).^2);
```

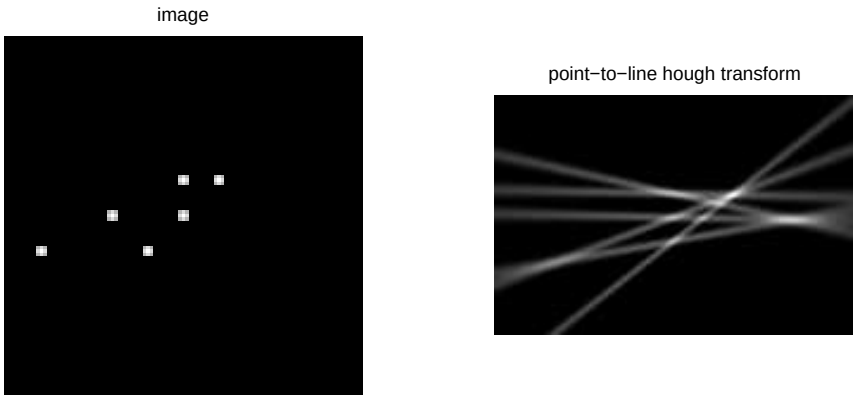


Figure 8.7: **Hough transform**: the thickening of the Hough-stamp for the point-to-line Hough transform.

This is illustrated in Figure 8.7. For the more general Hough transform, you need to replace the `writeline` function by the appropriate Hough stamp in its own parameter space.

Step 3: point detection

We now need to find the points in parameter space which are being voted for most: those are the parameter combinations that actually occur in the edge image to form the lines, circles or other shapes for which we have constructed the parameter space.

So you typically select the bins in the parameter space which have filled above a certain threshold level. Sometimes you known approximately how many boundary elements you expect, and make this determine your selection. So if the image has been set up to contain only a hexagon which you want to detect, you can just select the 6 fullest bins as the relevant points. (If the resolution was very fine-grained, these may not represent different lines, so some care is required.)

Step 4: transforming back

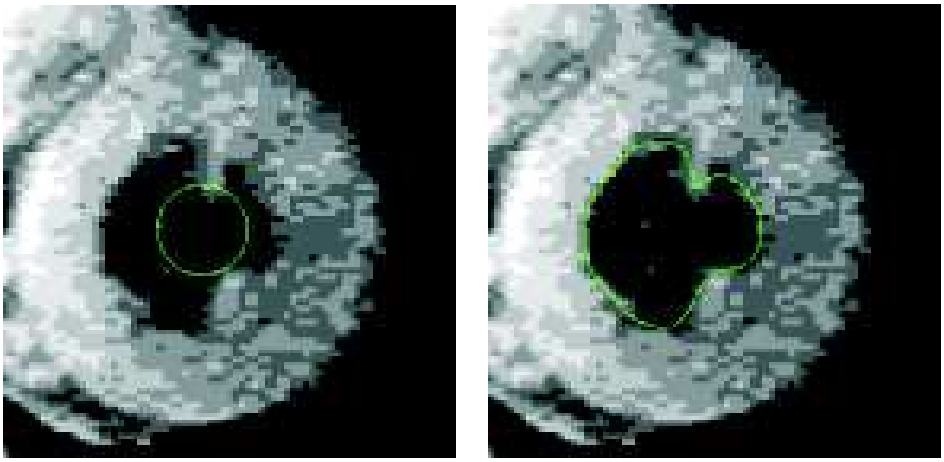
The points found in step 3 correspond to ideal matches of whole lines (or circles, or other shapes); you can now transform them back and draw those in the original (edge) image. The basic equation for your representation gives you that transform immediately; for instance, having found a point (a_0, b_0) in the point-to-line Hough transform gives the line $y = a_0\,x + b_0$ in the image, by eq.(8.1).

Step 5: selection of boundary points

When your transform the points back, you get *whole* lines, circles, etc. It rarely happens that these are fully present in the edge image, often you only have parts of them present: line segments, circular arcs. So you typically use the shapes found as a guide to select which lines etc. occur, and therefore which points should be considered to be part of the interesting boundaries.

Step 6: better fit

Now that we know which edge points need to form each line segment (or other elementary shape), it is a good idea to do an accurate fit with those points to that shape. You might use a least



a b

Figure 8.8: **Snakes:** from an initial estimate the aorta can be found. Images from the site `http://iacl.ece.jhu.edu/projects/gvf/`.

squares fitting method to do this. After this, you finally have your boundaries in a parametrized representation.

8.4 Grouping Boundaries: Snakes

A contour of an object in a 2-dimensional image is a closed curve indicating the separation between the object and the 'background'. Obviously once you would have the contour, you could focus your attention fully on the object, so determining the contour is an important intermediate step in understanding and analyzing an image. Unfortunately we have seen many examples in which the contour of an object seems hard to determine on purely local information only. The basic reason is that most objects are not easily distinguishable from their background: the imaging properties are very similar – indeed, the background itself probably contains objects of interest as well. Only in exceptional artificial setups in industrial inspection can one arrange the imaging so that the objects really have different imaging properties and are easy to separate from the background – for instance in shape inspection by backlighting them.

One can often give an approximate estimate of the contour, either automatically on the basis of the collected local structure, or because one expects the object in a certain region (for instance because you are following an object in a video sequence and knew where it was in the previous frame), or, as a last resort, by human input (such as drawing a coarse circle around the object). In those cases, we would like a method that takes this approximate contour and automatically adjusts it to become a better indication of the object boundary. An example is Figure 8.8, where a good fit for an aorta is found based on a coarse initial estimate.

Such methods are called *deformable contour methods*, or *active contours*, and they are popularly known as *snakes*, since they writhe and wriggle when they adjust their shape.

8.4.1 Contours

Since a contour denotes a separation between regions, it does not really consist of pixels. Rather it is a continuous curve between pixels. Pixels represent some energy over a finite region, but

a contour should be infinitely thin. Of course, when you draw it to your screen, your display program will make pixels to denote approximately where it is, and it is easy to mistake that for the actual contour. It would be somewhat more accurate to draw the separation induced by the contour by masking out all pixels in the background, leaving only the inside showing.

So in our 2-dimensional images, a contour is a closed curve. Let us briefly specify tools used in the mathematical description of curves, since we will need those to develop our algorithms for snakes.

A curve is a collection of locations $\mathbf{p}$ indexed by a single continuous parameter t. Think of walking along the curve, then $\mathbf{p}(t)$ is your location at the time t. You can walk along the same curve with different speeds, so we have to be a bit careful that we do not take the parameter t too seriously – it is a bit too specific for the description of the curve itself. For instance, if you draw the curve $(t, t^2)'$ with $t \in [0, 1]$, or the curve $(\sin(t), \sin^2(t))'$ with $t \in [0, \pi/2]$, you find that they are the same.

In order to prevent annoying special cases, we do demand that you traverse the curve without tracing back your steps, proceeding in the direction you chose for your first step. And it is also convenient to agree on that direction, to make it a proper boundary: let us always make it such that the object 'mass' is on your left hand side. So the curve $(\cos(t), \sin(t))'$ then denotes a boundary around a circular object, but $(\cos(-t), \sin(-t))'$ denotes a circular hole in an object.

For a closed curve you will find the same points $\mathbf{p}(t)$ after a certain time t, depending on your speed. (The same points may also come up along a curve when it intersects itself, but since those curves cannot very well denote boundaries we will try to exclude such curves from our descriptions.)

At any time, you can determine your velocity – remember that velocity is a vector, having both a magnitude (usually called the *speed*) and a direction (called the *bearing* or *heading*). This is just another example of the derivative: take the difference between two points a little time apart, and take the limit when that time goes to zero:

$$\dot{\mathbf{p}}(t) \equiv \frac{d\mathbf{p}}{dt}(t) \equiv \lim_{h \to 0} \frac{\mathbf{p}(t+h) - \mathbf{p}(t)}{h}$$

We will call a curve *smooth* when this velocity is a continuous function of t; so a smooth curve does not have kinks. We can approximate a curve with kinks by a smooth curve to any degree of accuracy, so if we limit ourselves to smooth curves (they are easier to work with) this is not a limitation in practice.

8.4.2 The energy of a snake

A *snake* is a contour curve which can reshape itself. The way a snake moves is partly inspired by the physics of the motion of a loop of rather stiff elastic material, falling into a ditch. Forgetting the ditch for the moment, the loop tries to take on a circular shape: it wants to shrink (because it is elastic) and it wants to straighten out (because it is stiff). So if you bring it out of this natural shape, it starts to move to achieve its ideal shape.

In physics, things move because a force acts on them; apparently, the loop's deviation from the ideal shape generates a force at each point. Such forces are vectors: the point tends to move in a certain direction, with a certain acceleration. If there are multiple forces working on points, there is a nice trick to combine them: rather than adding the vectors, one adds energies. Many forces can be considered as (minus) the gradient of a special function, i.e. as the direction and magnitude of the steepest decrease of this function. That function is the *energy*. So motion takes place to minimize energy.

To specify the motion of a snake, you have to specify its energy function; then the 'energy-minimization-algorithm' will make the snake move to minimize this energy. Obviously, there is a lot of freedom in choosing the energy, and some of those choices are very specific for the problem at hand. But in general, there are four recognizable components,

$$E_{total} = \alpha\, E_{elastic} + \beta\, E_{bending} + \gamma\, E_{image} + \delta\, E_{constraint} \tag{8.4}$$

The numbers α, β γ, δ are weighting factors to control the relative importance of each of the contributions. We now discuss the various terms.

- *elastic energy*
  Thinking of the snake as a rubber band, the property that it wants to shrink is described by the *elastic energy*

  $$E_{elastic} = \int |\dot{\mathbf{p}}(t)|^2 dt.$$

  This leads to a force that tends to straighten and shorten the snake. (In the literature, some people call this the 'continuity' term of the energy because it takes spikes out of the snake; they presumably mean it in the sense of 'continuity of the derivative'.)

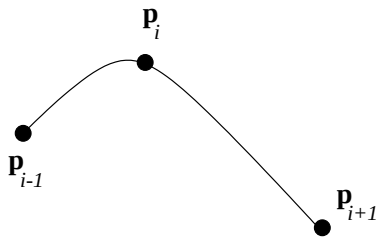


Figure 8.9: **Contour approximation**: a double-quadratic approximation of a contour.

If we have a snake given by a sequence of discrete points $\mathbf{p}_i$ (with $i = 1, \cdots, n$, we need to have a discrete equivalent of this formula. We can give this immediately from the physical intuition: the elastic energy of a spring between two points is $k\,|\mathbf{p}-\mathbf{q}|^2$ (with k being called *Hooke's constant*). So the total elastic energy is:

$$E_{elastic} = k \sum_{i=1}^{n} |\mathbf{p}_i - \mathbf{p}_{i-1}|^2.$$

For each point triple $\mathbf{p}_{i-1}, \mathbf{p}_i, \mathbf{p}_{i+1}$, the contribution would be minimal if $\mathbf{p}_i$ were on the straight line connection $\mathbf{p}_{i-1}$ and $\mathbf{p}_{i+1}$, but a closed curve cannot be drawn completely straight. A decent solution is to make an iterative procedure in which each point $\mathbf{p}_i$ moves some way towards that line connecting its neighbors. See exercise at the end of this chapter!

Figure 8.10 gives this kind of evolution for a polygon: you see that it rapidly becomes elliptical and shorter, whatever its original shape may have been.

- *bending energy*
  The bending energy is about the straightness of the local contour; it is proportional to the

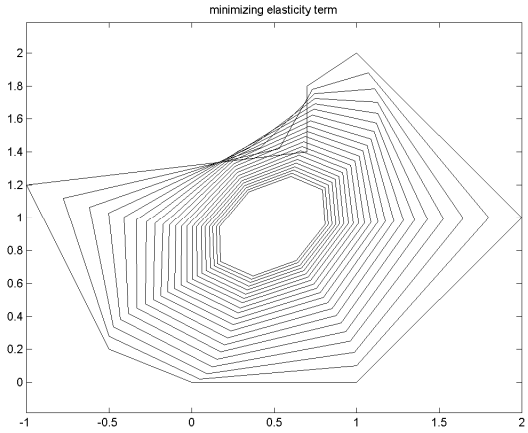


Figure 8.10: **Contour evolution**: minimizing the elastic energy. For simplicity we have depicted the snake determined by the points by a polygon.

curvature of the curve, which is the rate of change of the unit tangent vector. So minimizing curvature involves the second derivative, and leads to an energy of the form:

$$E_{bending} = \int |\ddot{\mathbf{p}}(t)|^2 dt.$$

Of all closed contours, this is minimal for a circle, as you would expect from the analogy with physics. So minimization should lead to making the contour more circular; a point moves to lie on a circle determined by 3 neighboring points.

In the discrete case, we need to have a measure of the second derivative to compute

$$E_{bending} = \kappa \sum_{i=1}^{n} |\ddot{\mathbf{p}}_i|^2.$$

So we need a sensible local model of the snake near the point $\mathbf{p}_i$ to estimate the second derivative at i. Here is an example of such a model: locally, between three points, you could approximate the snake by a continuous curve that is quadratic in both its components, as in Figure 8.9, so:

$$\mathbf{p}(t) = \mathbf{p}_i + \tfrac{1}{2}(\mathbf{p}_{i+1} - \mathbf{p}_{i-1})\, t + \tfrac{1}{2}(\mathbf{p}_{i-1} - 2\mathbf{p}_i + \mathbf{p}_{i+1})\, t^2 \tag{8.5}$$

You can easily verify that $\mathbf{p}(-1) = \mathbf{p}_{i-1}$, $\mathbf{p}(0) = \mathbf{p}_i$ and $\mathbf{p}(1) = \mathbf{p}_{i+1}$. This model of the snake provides us a way to motivate sensible formulas for the first derivative at the point $\mathbf{p}_i = \mathbf{p}(0)$, since

$$\dot{\mathbf{p}}_i \approx \dot{\mathbf{p}}(0) = \tfrac{1}{2}(\mathbf{p}_{i+1} - \mathbf{p}_{i-1}), \tag{8.6}$$

and

$$\ddot{\mathbf{p}}_i \approx \ddot{\mathbf{p}}(0) = \mathbf{p}_{i+1} - 2\mathbf{p}_i + \mathbf{p}_{i-1}). \tag{8.7}$$

Conversely, if you are using such approximations to the first and second derivative at $\mathbf{p}_i$, then you can motivate that by saying that you are thinking of the continuous curve $\mathbf{p}(t)$ as if it runs like eq.(8.5).

An explicit local solution might be to move the point $\mathbf{p}_i$ so that it goes towards a circle fitted to its neighbors; but it is more customary to solve the minimization of this term in combination with the others. We will get to that in section 8.4.3.

- *image-based energy*
  If you use the snake to help you find a contour, there should be an energy related to where the edge of the object is. If you have an object that has a clear brightness difference with the background, you can use (minus) the norm of the image gradient (or its square) as such a measure:

$$E_{image} = -\int |(\nabla f)(\mathbf{p}(s))|^2 ds = -\int |(\nabla f)(\mathbf{p}(t))|^2 \, |\dot{\mathbf{p}}(t)| dt.$$

  As before, the notation $(\nabla f)(\mathbf{p})$ means: the gradient of the image f, evaluated at the location $\mathbf{p}$. The first expression uses the 'arc length' parametrization s, and weights the image gradient properly over the length of the curve; the second is the equivalent expression in an arbitrary parametrization: note that $|\dot{\mathbf{p}}(t)|dt$ is the length of the piece of curve traversed in a time dt. (Q: What would the value of $|\frac{d\mathbf{p}}{ds}(s)|$ be?)

  If you plot $-|\nabla f|^2$ as a function of position in the image, you see that this function is a 'ditch' for the snake to fall into. But if you use this energy in combination with the previous two, the snake will also try to minimize its internal energy (elastic and bending) so it will not quite follow the deepest part of the ditch after it is done. (If that minimum would be a good approximation to the contour, you would not need snakes!)

  To estimate the local image gradient at a location $\mathbf{p}_i$, we can obviously use the techniques earlier in the course: a facet model, or the Gaussian derivatives.

- *constraint energy*
  As people started implementing snakes according to those three energies above, they found that it offered not quite enough control over where the snake moved to. So many introduced an artificial energy in which they packaged the 'constraints', using it for instance to make sure that the snake would not move away from sections of the contour they were pretty sure about.

  You can made such an energy that pulls you to a certain 'anchor' location $\mathbf{P}$ for instance by imagining springs attached between $\mathbf{P}$ and the snake points $\mathbf{p}_i$, and adding the energy of those springs: $\sum |\mathbf{P} - \mathbf{p}_i|^2$ to the total energy. That is indeed one of the techniques people use to design constraint energies, sometimes selecting which part of the snake should be pulled to which anchors. It is all a bit arbitrary and patchy, but some of those methods are applicable to a wider class of problems.

8.4.3 Energy minimization

So a snake consists of points $\mathbf{p}(t)$, and we need to somehow move those such that the total energy over the snake is minimal. We discuss this in a discrete implementation, in which we have used a set of discrete point $\mathbf{p}_i$ to characterize the snake. These points characterize the contour. For instance, we can use the points to make a polygon, or a piecewise double-parabolic curve as in eq.(8.5), or a spline (but be careful: you cannot just take the control points of a spline if those points are not *on* the curve, since we will use them to estimate the energy *along* the curve!)

The initial guess for the contour is then given by this circular sequence $\mathbf{p}_i$, with $i \in [1, n]$. The energy is now a sum over the energy of this sequence. We approximate the integrals by sums, as before. Instead of optimizing the total sum, we are now going to optimize the contribution of each point through a local displacement. This is a *greedy algorithm*. It is *not* exact, i.e. it is not actually a minimization of E.

So for a point $\mathbf{p}_i$, we consider its local neighborhood and in that some possible locations $\mathbf{q}_j$ that the point might move to. We compute the energy change which we would get if the point

were to move to $\mathbf{q}_j$ but all other points would remain the same. (This is wrong! You would want to know what the new internal energy is after all the points have moved – so this is a simplification. If the motion of the points is small relative to their distance, it is a reasonable approximation; but if not, then not.) Then among all those $\mathbf{q}_j$, we pick the one that minimizes the energy most, and that is the new position of the point $\mathbf{p}_i$.

> Several variations of this method have been designed. For instance, if you would limit the $\mathbf{p}_i$ and $\mathbf{q}_j$ to be locations on the pixel grid, you could take a 3×3 neighborhood around $\mathbf{p}_i$ and take the best of those.
>
> Or, if you allow a real-valued position for $\mathbf{p}_i$ and $\mathbf{q}_i$, you could use a limited number of $\mathbf{q}_j$ along the grid directions to estimate the local energy function using the facet model or a Taylor approximation, and then move the point $\mathbf{p}_i$ in the 'minus-gradient' direction of this estimated function.

Doing this for all points gives the new contour sequence. You have some control over the relative importance of the various energy terms by changing the weighting parameters $\alpha, \beta, \gamma, \delta$ in the total energy.

An example is Figure 8.8. Here a snake is being fit mostly to the gradient image, it follows the desired contour of the aorta in all its detail. Another example is Figure 8.11. Here the desired object is not really determined by the black/white gradient image (indicated in (b)); so this time a snake needs to be used which emphasizes the internal elastic and bending energy more. The result in (c) is not bad, but not perfect – but then, this was a hard problem.

When you look into the large literature about snakes you find that many people have modified this basic outline, to get snakes that better corresponded to the kind of contours they wanted to find in their application. For instance, in many applications, your desired contour might have sharp corners (for instance because you are looking at man-made objects); the bending energy term would try to lead the snake smoothly around those, so you need to give those some special treatment to keep it in place around them. Some do it with the constraint energy, others just take those points out of the minimization routine.

8.4.4 The bottom line

Some concluding remarks:

- *a common language*
  Although they are not a perfect description of the contour tightening process, snakes do provide a common terminology and a framework for the design of techniques permitting exchange of methods and algorithms.

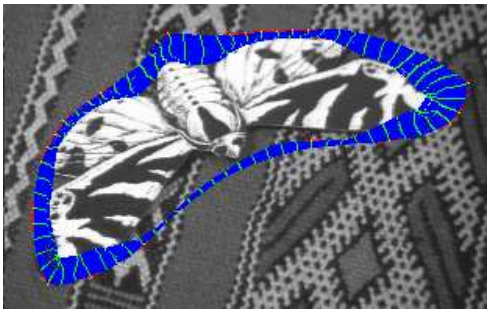
- *initial estimate*
  The initial estimate is crucial to the quality of the final contour and the resulting segmentation.

- *parselmouth*
  You 'talk' to snakes through the energy functions and their relative magnitudes (the α etc. in eq.(8.4)). The optimization procedure is very sensitive to the relative importance of these various forms of energy.

- *optimization*
  We have just given an indication of a rather naive optimization procedure through a greedy algorithm; if you really need snakes, look for more accurate optimization methods under the keywords *steepest descent*, *conjugate gradient*, *variational calculus*.

a

b



c

Figure 8.11: **Snakes:** how a snake catches a moth. (a) original image with first snake (white dots); (b) norm-of-gradient image; (c) snake interations to final loose noose. You may check the color version for clarity: images from the site `www.cogs.susx.ac.uk/users/davidy/teachvision/vision7.html#heading11`.

- *what contour*
  In many applications, the precise small variations of the contour are what you need automatic procedure to produce, since they are too cumbersome to denote by hand. In those cases, subtle problem-dependent energy functions or algorithmic modifications may be required.

- *3-D*
  Some methods can be extended to the contour surfaces of 3-dimensional objects – but it is not easy.

8.5 Exercises

1. **Region growing for color images.**

   In Section 8.1 a region growing algorithm for grey value images is given. Define the region growing algorithm for color images in case the color difference of the color $f(\mathbf{x}) = (r(\mathbf{x}), g(\mathbf{x}), b(\mathbf{x}))^\mathsf{T}$ and the color at the seed point $\mathbf{x}_0$ is defined as:

   $$d(\mathbf{x}, \mathbf{y}) = \max\left(|r(\mathbf{x}) - r(\mathbf{x}_0)|, |g(\mathbf{x}) - g(\mathbf{x}_0)|, |b(\mathbf{x}) - b(\mathbf{x}_0)|\right).$$

   (a) Show that the set $\mathcal{B}$ of all points $\mathbf{x}$ in the image domain with $d(\mathbf{x}_0, \mathbf{x}) \leq \epsilon$ is given by

the *intersection* of the sets:

$$\{\mathbf{x} \mid |c_i(\mathbf{x}) - c_i(\mathbf{x}_0)| \leq \epsilon\}$$

for $c_i = \{r, g, b\}$.

(b) Adapt the Algorithm 8.1 to work with the given definition of 'color differences'.

2. **Region growing based on Hue differences.**

   Instead of the defining color differences using all three components of the color in some cases it is advantageous to select a measure of color differences that is based only on the hue. In Matlab the function rgb2hsv transforms a color image in the rgb-representation into the equivalent color image in the hsv-representation. Use this color transformation and the scalar region growing algorithm to segment the yellow flower in the image flowers.tif.

3. **Hough**
   (Not easy!) Show that the weighting factor for the point-to-line Hough transformation needs to grow with a as in the code given above. For simplicity, assume that each pixel in the original edge image represents a circular sensitivity area of width `scale` and uniform density.

4. **Hough for line segments**
   We could have taken a Hough transformation representing line *segments* (rather than whole lines) immediately and voted for those. This would allow you to skip step 5 in the Hough method.

   Think this through: how many parameters would such a space of line segments have; what consequences would that have for the speed of the algorithm? Under what circumstances might you prefer it to the method in the text of taking whole lines and limiting their scope later?

5. **Snakes**
   Show that the movement toward the middle of the line segment between $\mathbf{p}_{i-1}$ and $\mathbf{p}_{i+1}$ is a movement in the direction of the second derivative of eq.(8.7). How far should you move to have a stable algorithm? Can you guarantee stability of your algorithm? Implement this, and experiment! See also Figure 8.9.

Appendix A

Linear Image Operators

In this appendix we will consider discrete image operators only, i.e. operators working on discrete representations obtained by sampling a continuous function. This restriction has been made to circumvent the necessity to introduce Dirac functions. In the discrete case we have that the integrals from the continuous image models become simple summations. Nevertheless most of what is shown in this appendix is true for the continuous case as well.

Translation invariance. An image operator O is called translation invariant[1] in case O commutes with the translation operator T. This is expressed in the following commutative diagram:

$$\begin{array}{ccc} f & \xrightarrow{T} & Tf \\ {\scriptstyle O}\downarrow & & \downarrow{\scriptstyle O} \\ Of & \xrightarrow[T]{} & TOf = OTf \end{array}$$

A translation over vector t is denoted with the translation operator T_t. We thus have:

$$(T_t f)(\mathbf{x}) = f(\mathbf{x} - \mathbf{t})$$

The Kronecker delta function. The Kronecker delta function is the discrete elementary pulse for linear operators:

$$\delta(\mathbf{x}) = \begin{cases} 1 & : \quad \mathbf{x} = 0 \\ 0 & : \quad \mathbf{x} \neq 0 \end{cases}$$

Image as a sum of pulses. Using the Kronecker delta function any discrete image $f : E \to V$ can be written as the sum of translated image pulses:

$$f(\mathbf{x}) = \sum_{\mathbf{y} \in E} f(\mathbf{y})(T_{\mathbf{x}}\delta)(\mathbf{y})$$

[1]Strictly speaking translation invariance doesn't make sense for bounded images, only for the infinite visual domain is the translation operator consistently defined. Nevertheless translation invariance is an often used assumption. In approximation it is also valid for bounded images.

Linear image operators. An image operator L is called a linear operator in case:

$$L\left(\sum_{i\in\mathcal{I}}\alpha_i f_i\right)=\sum_{i\in\mathcal{I}}\alpha_i L(f_i)$$

where $\{f_i\}_{i\in\mathcal{I}}$ is a collection of images and $\mathcal{I}$ is some index set.

Convolution. An important theorem in linear systems theory is that any linear, translation invariant operator is a convolution. The proof is remarkably simple. Using the Kronecker delta function we may write an image f as:

$$f=\sum_{\mathbf{y}\in E} f(\mathbf{y})T_{\mathbf{y}}\delta.$$

Applying the linear, translation invariant operator L on this image we obtain:

$$L(f)=L\left(\sum_{\mathbf{y}\in E} f(\mathbf{y})T_{\mathbf{y}}\delta\right).$$

Because L is linear we can apply it to the images $\delta_{\mathbf{y}}$ and then take the sum:

$$L(f)=\sum_{\mathbf{y}\in E} f(\mathbf{y})L(T_{\mathbf{y}}\delta).$$

Note that L is assumed to be a translation invariant operator, thus:

$$L(f)=\sum_{\mathbf{y}\in E} f(\mathbf{y})T_{\mathbf{y}}L(\delta).$$

Note that $L(\delta)$ is a function as well. For obvious reasons it is called the *impulse response function*. Let $w=L(\delta)$:

$$L(f)=\sum_{\mathbf{y}\in E} f(\mathbf{y})T_{\mathbf{y}}w=\sum_{\mathbf{y}\in E} f(\mathbf{y})(T_{\mathbf{y}}w)(\mathbf{x})=\sum_{\mathbf{y}\in E} f(\mathbf{y})w(\mathbf{x}-\mathbf{y}).$$

This is called the *convolution sum* of the functions f and w, denoted as $f*w$

Convolution recipe. In image processing and all other signal processing application areas the linear operators and thus the concept of convolution is of great importance. A recipe to calculate $(f*w)(x)$ given a function f and the *convolution kernel* w is:

1. mirror the function w in the origin giving function $w^\star(\mathbf{x})=w(-\mathbf{x})$.
2. shift the function to position $\mathbf{x}$. This gives function $T_{\mathbf{x}}(w^\star)$.
3. multiply $T_{\mathbf{x}}(w^\star)$ pointwise with the original function f: $f(\mathbf{y})(T_x(w^\star))(\mathbf{y})$, and
4. take the sum over the entire domain:

$$(f\star w)(\mathbf{x})=\sum_{\mathbf{y}\in E} f(\mathbf{y})(T_{\mathbf{x}}(w^\star))(\mathbf{y})$$

Note that we write $T_{\mathbf{x}}(w^\star)$ to indicate that we first mirror in the origin followed by a translation. These two operators do not commute: $T_{\mathbf{x}}(w^\star)\neq(T_{\mathbf{x}}w)^\star$.

Commutative. The convolution is a commutative operator: $f * g = g * f$.

The importance and use of the commutative property lies mainly in its use in the proofs of properties of the convolution. In practice we *do* make a distinction between the function f and the kernel g; the kernel in practice is a parameter to the image operator that takes one image and produces a new image (namely the convolution with g).

Associative. The convolution is an associative operator: $(f * g) * h = f * (g * h)$. The associativity of the convolution operator can be proven directly using the definition of the convolution.

Support of convolution kernels. The *effective domain* or *support* of a convolution kernel w is the subset of E for which it is non-zero:

$$S_w = \{\mathbf{x} \in E \mid w(\mathbf{x}) \neq 0\}$$

In an algorithm to implement the convolution we only have to take the points in the effective domain into consideration as outside the domain the contribution to the convolution sum is equal to zero.

In this appendix we will use the following notational convention for 2D convolution kernels: i) we only give the values $w(x)$ for x in the effective domain and ii) we write them down in a spatial layout corresponding with the discretization grid.

Consider the discrete convolution $f * w$ where the kernel is:

$$w(k,l) = \begin{cases} \frac{1}{9} & : \quad -1 \leq k,l \leq 1 \\ 0 & : \quad \text{elsewhere} \end{cases}$$

The kernel is graphically depicted as:

$$w = \frac{1}{9} \begin{Bmatrix} 1 & 1 & 1 \\ 1 & \underline{1} & 1 \\ 1 & 1 & 1 \end{Bmatrix}$$

where we have underlined the value in the origin. The common factor 1/9 is put in front of the kernel. All values not shown in the kernel are assumed to be zero. Please note that a kernel is NOT a matrix.

The convolution:

$$f * \frac{1}{9} \begin{Bmatrix} 1 & 1 & 1 \\ 1 & \underline{1} & 1 \\ 1 & 1 & 1 \end{Bmatrix}$$

thus calculates the local average in a 3×3 neighborhood around all the points in image f.

First order derivative in the horizontal direction. The convolution needed to calculate the first order derivative of an image f based on a facet model within a 3×3 neighborhood is:

$$f * \frac{1}{6} \begin{Bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{Bmatrix}$$

All the other 3×3 kernels to approximate the derivatives up to order 2 are given in section 2.7.

Domain decomposition. The associativity of the convolution is a very important property of translation invariant linear operators. It allows us to build large convolution kernels from small ones.

The computational complexity of a straightforward implementation of the convolution operator is linear in the number of points in the support of the kernel. Therefore, if we can reduce the number of pixels in the support we can make computation of a convolution more efficient.

As an example consider the sequence of 2 uniform convolutions:

$$\left(f * \frac{1}{9}\begin{Bmatrix} 1 & 1 & 1 \\ 1 & \underline{1} & 1 \\ 1 & 1 & 1 \end{Bmatrix}\right) * \frac{1}{9}\begin{Bmatrix} 1 & 1 & 1 \\ 1 & \underline{1} & 1 \\ 1 & 1 & 1 \end{Bmatrix} = f * \frac{1}{81}\begin{Bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & \underline{9} & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{Bmatrix}.$$

Note that a convolution with a 3×3-kernel in general requires 9 multiplications and additions per pixel in the image. A sequence of n of these convolutions would require $9n$ operations. The resulting kernel is of size $(2n+1) \times (2n+1)$ and would require $(2n+1)^2$ operations is used directly in a convolution operator.

Dimensional decomposition. Instead of decomposing a large d-dimensional convolution kernel into several d-dimensional convolution kernels with a small spatial support, it is in certain cases possible to decompose a d-dimensional kernel into d one dimensional kernels. This is called *dimensional decomposition*.

Let $w(x_1, \ldots, x_d)$ be the values from a d-dimensional kernel. This kernel is dimensional decomposable in case there exist d one dimensional functions $w_1, \ldots, w_d$ such that:

$$w(x_1, \ldots, x_d) = w_1(x_1)w_2(x_2)\cdots w_d(x_d).$$

We then have that:

$$f * w = (\cdots((f * w_1) * w_2)\cdots * w_d)$$

Here we have misused the notation a bit. Strictly speaking the w_i's are one dimensional functions. Nevertheless we use them in d-dimensional convolutions. The following example may illustrate that the function w_i is directed along the i-th base vector in the d-dimensional space.

Consider the 2-dimensional kernel:

$$w = \frac{1}{25}\begin{Bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & \underline{1} & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{Bmatrix}$$

For this kernel we have:

$$\begin{aligned} w &= w_1 * w_2 \\ &= \frac{1}{5}\begin{Bmatrix} 1 & 1 & \underline{1} & 1 & 1 \end{Bmatrix} * \frac{1}{5}\begin{Bmatrix} 1 \\ 1 \\ \underline{1} \\ 1 \\ 1 \end{Bmatrix} \end{aligned}$$

In general the convolution with a $n \times n$ kernel has a computational complexity of order n^2 whereas the convolution using the dimensionally decomposition (if it exists) has a computational complexity of order n.

Gaussian Convolutions. The Gaussian kernel (in 2D):

$$G^s(x,y) = \frac{1}{2\pi s^2} \exp\left(-\frac{x^2+y^2}{2s^2}\right)$$

can be rewritten as:

$$G^s(x,y) = \frac{1}{s\sqrt{2\pi}} \exp\left(-\frac{x^2}{2s^2}\right) \frac{1}{s\sqrt{2\pi}} \exp\left(-\frac{y^2}{2s^2}\right)$$

This means that indeed the Gaussian 2D convolution can be decomposed in 2 one dimensional convolutions. I.e. first a convolution along all the rows in an image, followed by a convolution along all columns in the resulting image.

The Gaussian kernel has a lot of special properties. Here we mention only one: *the Gaussian kernel is the unique rotationally symmetric kernel that can be dimensionally decomposed.*

Appendix B

Least squares solution of homogeneous equations

This note is taken from section A.3.4.2 in the book *Multiple View Geometry in Computer Vision* by R. Hartley and A. Zisserman. A recommended book in case more information on 3D vision is needed.

We seek a solution to the homogeneous equation $Ax = 0$ where A is an $m \times n$ matrix with $m > n$. We look for a non-trivial solution $x \neq 0$. The column rank of A should therefore be less then n.

If A results from observation and is known to have a non-trivial solution in theory, in practice it will often be full rank due to noise and other measurement errors. Therefore we will look for an approximate solution: minimize $\|Ax\|$.

Again we look for non-trivial solutions. This can be done with the constraint $\|x\| = 1$ (note that this is allowed since $Ax = 0$ is equivalent with $Awx = 0$ for scalar w). Thus:

$$\text{minimize:} \quad \|Ax\| \qquad \text{subject to:} \quad \|x\| = 1 \tag{B.1}$$

Let $A = UDV^\mathsf{T}$ be the SVD decomposition of A, so

$$\text{minimize:} \quad \|UDV^\mathsf{T}x\| \qquad \text{subject to:} \quad \|x\| = 1$$

Because U is orthogonal and therefore norm preserving we have:

$$\text{minimize:} \quad \|DV^\mathsf{T}x\| \qquad \text{subject to:} \quad \|x\| = 1$$

Also V is orthogonal (and thus V^T as well) and thus:

$$\text{minimize:} \quad \|DV^\mathsf{T}x\| \qquad \text{subject to:} \quad \|V^\mathsf{T}x\| = 1$$

Set $y = V^\mathsf{T}x$:

$$\text{minimize:} \quad \|Dy\| \qquad \text{subject to:} \quad \|y\| = 1$$

Now D is a diagonal matrix, and we get the smallest outcome for $\|Dy\|$ by picking up the smallest element in D. Since the SVD gives D moreover as a diagonal matrix with its diagonal elements in descending order, taking $y = (0, 0, \ldots, 0, 1)^\mathsf{T}$ is the one that picks up this smallest element. Therefore this is the solution, but we still need to translate it back to x. Because $y = V^\mathsf{T}x$ we have $x = Vy$ and thus $x = V(0, 0, \ldots, 0, 1)^\mathsf{T}$, i.e. the last column of the matrix V is the solution of the minimization in equation B.1.

B.1 Appendix: How to read Zhang's report

B.1.1 How to read a paper

Whenever you have a scientific paper that is not quite in your field (yet), it is useful to read Abstract, Introduction, and Conclusions first. Together, these should provide a roadmap for the paper, and help prevent you getting lost in it.

Then, read the paper through once, trying to understand it, but don't spend too much effort on things you don't get immediately. Often these will become clear later on. There may be various reasons for this:

- The paper may be badly written, using things before they are properly introduced, or using inconvenient symbols or concepts. Also, the main line and the side tracks are not always clearly separated.
- The paper may use strange abstractions for very concrete things, without you (or the author) actually needing them at the abstract level. This is sometimes done to impress.
- There may be excursions that are interesting to specialists, but not required for understanding the paper at the level you need it.
- The author may assume common knowledge you don't have (yet).
- The paper may be the wrong paper to read for what you want to know; but when you see the author refer to other material in context you may be able to home in on the right information anyway.

The first pass of reading will help you understand the structure and identify the various difficulties you will have in understanding the paper, sometimes even already resolving them.

When you have decided that this is really the paper you want to read, and where the interesting bits are for you, you read it again. In this second pass, you should really make an effort to understand everything relevant (you will have identified the relevant parts in pass one). Follow the reasoning in detail, fill in the blanks using the references, some background material (including internet search for mathematical terms you may not know). This is not a linear pass, you have to deconstruct the paper! This pass should be very rewarding, you are increasing your knowledge considerably in doing this. (Always scout the references for useful literature you may not yet know!) Make notes in the margin, or attach your notes to the paper. You don't want this effort to get lost, so that you might have to do it again... At the end of this pass, you should know the relevant things well enough to be able to explain them to others.

After you have read and understoof it and have thought about it for a few days or weeks, go back a third time. You will have the satisfaction of seeing it all fall into place, and often some of the obscure or more abstract bits are now suddenly clear, deepening your understanding of the subject and the field. Summarize the salient points in 5 or so bullets on the title page, so that when you see the paper again you will remember what you liked about it (or didn't).

In brief, personalize the paper so that it becomes part of your ever-expanding knowledge.

B.1.2 Specific comments on Zhang's report

Zhang's report is rather well-written and should not be too hard to read.

Zhang could have said explicitly beforehand that (3) and (4) only depend on $\mathbf{A}^{-T}\mathbf{A}^{-1}$ so that it is sufficient to determine that – which is what he does in 3.1 – as long as you can retrieve

A from it – which is done using Appendix B. This is one of those structural things that you find out on the first pass of reading.

Also on first reading you find that section 2.4 is difficult, using unfamiliar terms, but you will also have found that it is not essential: it is an explanation of something he derives in different terms, and if those are unfamiliar to you it will clearly not help you. So you can skip 2.4 without losing much.

You should recognize the technique of 3.1 at first glance, turning the equations for **B** into a set of linear equations (9), and note how he just mentions in passing what the solution is. This is a technique we have treated, and this proves it is just part of the normal toolkit of a vision researcher/practitioner.

In section 3.2, saying that the maximum likelihood estimator 'can be obtained' by mininizing the functional is a bit strange: it is *defined* as the one minimizing the functional, so obviously it can be obtained by actually doing it...

Also in section 3.2, there is a reference to the *Rodrigues formula*. This is one of those things you can look up with Google. You should try to find a practical form of it, like:

$$\mathbf{R} = \cos\phi\ \mathbf{I} + (1 - \cos\phi)\mathbf{n}\,\mathbf{n}^T + \sin\phi\,\mathbf{n}^\times$$

(Relate this to **r** and find out what $\mathbf{n}^\times$ means, and you're done.)

In 3.3, we meet an old friend in (13). But the reference to Levenberg-Marquardt and `Minpack` is new. First reading will have shown you that you can treat it as a black box for now. If you want to know more, find out what it is via the internet; and if you find it cropping up in more and more papers in your field of interest, it will pay to make a closer study of it. You will need to understand all your tools eventually.

Isn't it nice of Zhang to give a summary in 3.4? If he had not done that, this is the place where you would have done it yourself, in the margin.

Chapter 4: is it a main issue, or an aside? You decide.

Chapter 5 on the experimental results: such chapters are always included in papers about methods that work. If you're going to use the method, they are very useful to read, since they give both qualitative and quantitative understanding, and of course they tell you how good the method is when used in the capable hands of the author, on his own datasets. That also helps you understand how general the method is, and whether it is applicable to your datasets.

In many papers of 'work in progress', the tests are incomplete, the understanding of the limitations preliminary, et cetera. Be on the lookout for places where the author says so himself; but that is often only part of the story. In Zhang's case, can we trust his judgement? You decide! His observations on page 16 are certainly very useful.

It is friendly of him to supply so many details in the appendices. This is often the place to find them, since putting them there keeps the flow of the paper clean. In your first pass, you will have decided which of the appendices you want to read as part of the main paper. The method described in his Appendix C is actually quite common now, and worth knowing. How would you summarize it?

Appendix C

Image Processing in Matlab

C.1 Exploring Matlab

Matlab is a very powerful program for technical and scientific numerical programming. It is so popular (and expensive) that several Matlab clones are available nowadays. Both commercially as well as freeware. Search on the Internet for 'Matlab' and 'clone' and you will find 'SciLab', 'Octave', 'O-matrix' and several others.

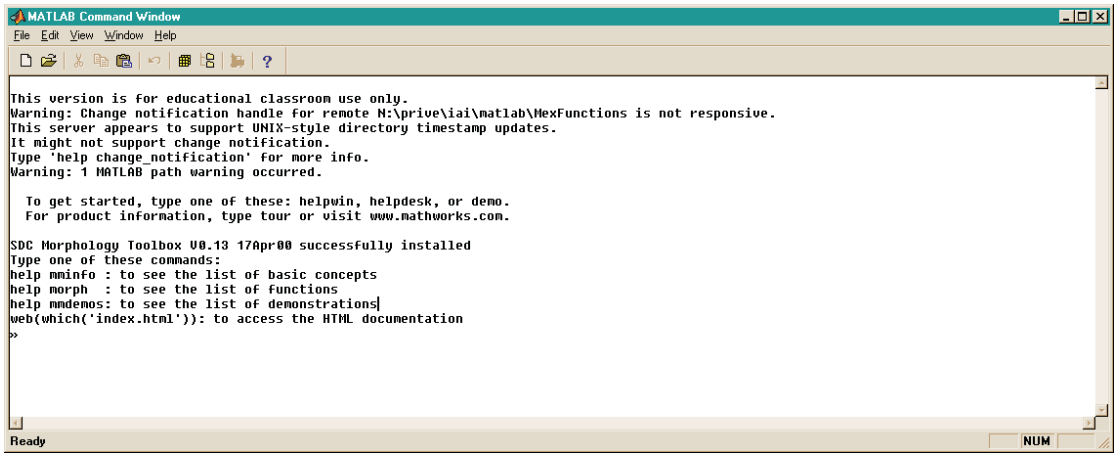
The best way to explore Matlab is by using it. The command demo brings up a GUI from which you can choose various basic tutorials and demonstration programs.

The help files are quite extensive. There is a plain ASCII text help system as well as a HTML based help system (the 'help desk'). You are advised to read the 'getting started' tutorial that is available on all Matlab installations.

C.2 Getting started

Image processing in Matlab is fun. It is simple, yet powerful for many of the tasks encountered in scientific image processing. This short introduction to Matlab and to image processing using Matlab is intended to help you get started enjoying the experience yourself.

We assume that you have Matlab 6.0 up and running on your PC. We also assume that the image processing toolbox is available. Your screen should look something like:



To get started right away, type in at the prompt (do not forget the semicolon!):

```
a = imread('kids.tif');
```

this will read the image in file 'kids.tif' (an image that is part of the Matlab distribution) and store the pixel data in the Matlab variable a. Note that in Matlab there is no need to declare variables before using them. Assigning a value to a variable has the consequence that the variable becomes of the type of what was assigned to it. In this case the variable a is an array of bytes (one of the possibilities to represent images in Matlab).

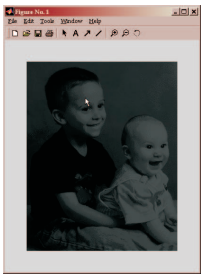
Within Matlab the command whos can be used to display the names of all variables and some associated information:

```
whos
  Name      Size          Bytes  Class
  a       400x318       127200  uint8 array
Grand total is 127200 elements using 127200 bytes
```

Here we have used the typographic convention that all output from Matlab is preceded with a vertical line on the left.
Here we see that there is only one variable a and that it is a two dimensional 400×318 array of uint8's (i.e. unsigned bytes). Matlab assumes that a 2D array of uint8's is an image where the value corresponds with black and the value 255 corresponds with white. The image a can be shown on the screen with the command:

```
imshow(a);
```

A separate window should appear on your screen looking something like:



Indeed the image should look rather dark. This is easily understood when we look at the minimum and maximum grey value present in the image:

```
minval=min(min(a))
  minval = 0

maxval=max(max(a))
  maxval = 63
```

showing that this image only utilizes the very low (dark) grey values (remember that grey value 255 is white and 0 is black). Read the on-line Matlab documentation for the function max to understand why you have to nest the function call to max twice[1].

Matlab can display images represented as an uint8 array but that is more or less all it can do with unsigned integers. To do some image arithmetic we first convert the image to a double (floating point) representation.

[1] A dimension independent way of calculating the maximum value in a d-dimensional array is (max(a(:)). Here we use the Matlab notation a(:) to denote the 1D vector of all elements in a.

```
b = im2double(a);
```

This function automatically scales the image in such a way that now the range of grey values runs from 0 (black) to 1 (white).

Exercise. Now print the minimum and maximum grey value in the image b.

We can change the grey values in b in such a way that the entire dynamic range is utilized (from 0 to 1 in this case):

```
c = 1/maxval * b;
```

Display the image c to see what happened. You can also check by calculating the maximal value for this image.

Worth mentioning are the zoom capabilities. For the 5.3 Matlab version there are magnifier buttons visible in the figure windows. Pressing the '+' magnifier button will allow you to zoom in by clicking with the left button and zoom out with the right button. Matlab 5.2 users can achieve the same using the command zoom.

Remember:

- using double images in Matlab is much easier.
- convert images to doubles using im2double (or do the conversion to the $[0, 1]$ interval yourself).
- when you change the pixel values of an image you need to display it again. Images on display are just snapshots of the values at the time the image was displayed
- the 'help' button on the Matlab window command window brings up the help browser. All Matlab functions are documented.

C.3 Array Indexing

Let us read another image from file:

```
a = imread('flowers.png');
```

this is a color image which is obvious when you display it (using the imshow command). For now we only process gray value images, so:

```
a = rgb2gray(a);
```

will result in a grey value image. Changing the resulting uint8 image to the double format:

```
a = im2double(a);
```

Display this image and show the coordinate axes:

```
imshow(a); axis on;
```

(note that we can put two statements on one line if they are separated by a semicolons.) This will change the display to:

A surprise in using Matlab for image processing is the choice for the coordinate axes. In Matlab images are represented using array data structures. The standard array indexing technique is used to access the individual pixels in the image. The value of a pixel at index (i,j) is obtained in Matlab as:

```
a(50,300)
  ans =
      0.3608
```

The answer from Matlab is appropriately assigned to the variable ans and displayed on the screen. Note that in the above command no semicolon was typed. Without ending a command with a semicolon, Matlab assumes that you want to display the result from a command (besides the 'side effects' a command might have, like rendering an image in a separate window). Ending the line with a semicolon prevents Matlab from displaying the result. Thus please end your image processing commands with a semicolon, else a very large number of pixel values will be displayed in the command window (pressing control-c will end this).

The value a(50,300) cannot only be obtained, but it can also be changed:

```
a(50,300) = 1;
```

Displaying the image again:

```
imshow(a);
```

shows that indeed something has changed: there is a small white spot visible somewhere. Convince yourself that it is at the location that you expected. Experiment with changing some pixel values to understand the coordinate system that Matlab is using. Is a(0,0) a valid indexing in the array?

In Matlab everything is an array. And because arrays are central to the Matlab universe it is easy to work with them. A vector of N numbers is represented in Matlab as a $1 \times N$ array and can be easily constructed from the command line (and in programs):

```
v=[ 8 5 1 9 ]
  v =
      8     5     1     9
```

The individual elements in the vector can be obtained and assigned to as v(1), v(2), v(3) and v(4). Besides indexing in an array with just numbers, also indexing with vectors is possible. As an example, define the vector:

```
x = [ 1 3 ];
```

and use this to index into the vector v:

```
v(x)
```

```
ans =
    8     1
```

You can also use this indexing construction to assign several elements from the array in one command:

```
v(x) = [ 11 34 ]
```

```
v =
    11     5    34     9
```

These indexing constructions are so often used in Matlab that there is a special constructor to make arrays with a series of scalars: the colon construction:

```
1:10
```

```
ans =
    1  2  3  4  5  6  7  8  9  10
```

Of course the start and end values can be chosen arbitrarily. A step value can also be specified:

```
100:-15:10
```

```
ans =
   100    85    70    55    40    25    10
```

For two dimensional arrays the indexing constructions are very handy indeed. For example to find the 3x3 matrix of values around a pixel with indices (50, 300) we can simply write:

```
a(49:51, 299:301)
```

```
ans =
    0.3451    0.3569    0.3922
    0.3529    1.0000    0.3686
    0.3373    0.3412    0.3569
```

Remember:

- ending a command with a semi-colon prevents the answer to be printed on the screen.
- typing CONTROL-C stops the printing of all image values (or other large data arrays) when you have omitted the semi-colon.
- indexing in Matlab starts at 1
- indexing in Matlab uses the matrix indexing scheme (row,column)
- a sequence (vector) of numbers can be easily generated with start:step:end
- index vectors (matrices) can be used to index a matrix

C.4 Displaying images and other plots

We assume that in a is the image from the last section. In case we take a scalar for one of the indices and let the other index run from 1 to the maximum we get a vector of values being either the pixel values in a row or column:

```
line = a( 128, :);
```

Please note that in an indexing construction the colon without any numbers stands for 1:end where end is the maximum allowed for the array involved. In case you didn't type the semicolon at the end of the previous command you will have seen a lot of numbers printed on your screen. There is however a better way to look at one dimensional data: the plot command.

```
plot(line)
```

In the graphical window where a was displayed you should now be able to see a grey value profile, i.e. the grey values in the 128th row plotted as a function of the column index. Note that in Matlab there is no need to have just one command per line. The previous two commands can be combined:

```
plot( a( 128, :));
```

Most often it is convenient to show both the image and the image profile. You can do that in Matlab in several ways. The 'difficult' way is to display both the image and the profile in one window (look at the subplot command documentation). It is easier to create two windows. Every figure command creates a new graphical (figure) window. All image rendering and plot commands will use the 'actual' window (the last one created or the last one used).

```
imshow(a); figure; plot( a(128,:) );
```

C.5 Programming in Matlab

Often you will find yourself typing the same sequences of commands over and over again. Then it's time to collect these commands in a function.

Consider the function nbh that returns a (2N+1) square neighborhood centered at the pixel at coordinates (i,j) in a given image im. The coordinates are easily generated:

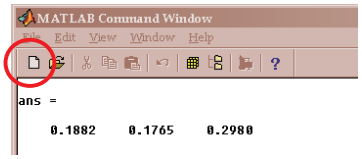
```
N = 3; i = 128; j = 164; % these will be the parameters
                         % to our function
iind = i + (-N:+N);
jind = j + (-N:+N);
```

Note that we have used a new language construction here. We have added a constant to a vector (array). The constant then is added to all elements from the vector. We cannot use im(iind, jind) to get the pixel values for all values of i and j . In case we are close to the border the neighborhood would not be (entirely) within the image and an error (index out of range) will be issued by the Matlab interpreter. Let imax and jmax be the maximum index in both axes:

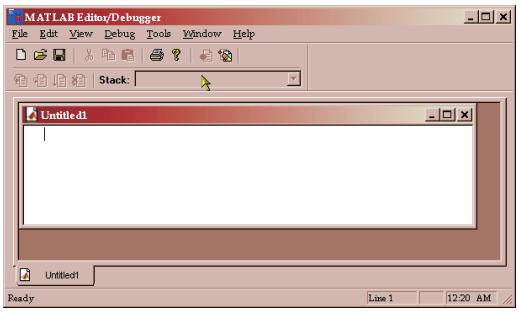
```
im = imread('bonemarr.tif'); % bone marrow image
im = im2double(im);
[imax, jmax] = size(im);
iind = max( min(iind, imax ), 1 );
jind = max( min( jind, jmax ), 1 );
```

Carefully note that a Matlab function can return multiple values as the size function does . Also note that again a scalar is 'distributed' over an array. The imax parameter in the min function is compared with each of the elements in the iind vector. Convince yourself that the above code fragment takes care of the 'border problem'. Let us cast this little piece of code in a small function.

In the command window press the button to start editing a new function:



This will open a text editor window:



Type in the following function:

```
function nbharray = nbh( image, i, j, N )
% nbh: get the 2N+1 squared neighborhood
% of the pixel (i,j) in 'image'
[imax, jmax] = size(image);
iind = i + (-N:+N);
jind = j + (-N:+N);
iind = max( min( iind, imax ), 1 );
jind = max( min( jind, jmax ), 1 );
nbharray = image( iind, jind );
```

Notice that in Matlab you don't have to explicitly return a value, just assign it to the 'return variable' (in the above function nbharray). Save this function in a directory that you have selected for this purpose. Give the file the same name as the function with extension .m, in this case nbh.m. Make sure that you add this directory to the path along which Matlab searches for files. You can do this by clicking the 'pathbrowser button'

on the Matlab tool bar. Now you are ready to use the function:

```
nbh(im,1,1,3)
```

```
ans =
   0.8392  0.8392  0.8392  0.8392  0.8627  0.8745  0.8510
   0.8392  0.8392  0.8392  0.8392  0.8627  0.8745  0.8510
   0.8392  0.8392  0.8392  0.8392  0.8627  0.8745  0.8510
   0.8392  0.8392  0.8392  0.8392  0.8627  0.8745  0.8510
   0.8549  0.8549  0.8549  0.8549  0.8588  0.9059  0.8902
   0.8588  0.8588  0.8588  0.8588  0.8784  0.9137  0.9059
   0.8706  0.8706  0.8706  0.8706  0.8784  0.9294  0.9137
```

You may not have noticed, but by now we are capable of writing our own convolution operation using a uniform kernel: enumerate all points `i,j` in the image and at each point calculate the average of the neighborhood pixel values.

```
function r = slowuniform( image, N )
[imax, jmax] = size(image);
r = image; % not strictly needed
           % but only here for 'efficiency'
np = (2*N+1)^2;
for i=1:imax
    for j = 1:jmax
        r(i,j) = sum(sum( nbh( image, i, j, N )))/np;
    end
end
```

Make sure you try the function (just to see that it is worthy of its name).

It should be noted that the syntax of a for-loop in Matlab is `for <varname> = <array>`. So instead of writing `for i=1:imax` we could have written `range = 1:imax; for i=range ...`.

Remember:

- one function per file works OK.
- a function name and its file name must be the same
- store your own functions in your own directory and set the Matlab path to it (use the menu item 'set path' in the file menu or click the 'path browser button').

C.6 Image Processing in Matlab

By now you certainly found out that writing your own image processing function in *interpreted* Matlab code is very inefficient. Rule 1 in Matlab is that `for` or any other loop over the individual pixels is to be avoided. Either use a function available in the image processing toolbox or try to use generic array processing functions. In case everything else fails you could also write your own functions in C and use them from within Matlab as if it were a standard function. You need a C-compiler to do so.

The Matlab version used in the practical has the 'image processing toolbox' available. Browse the help files to see the functions in this toolbox. During the practical course you will be pointed to several of the functions from the toolbox to solve image processing problems efficiently.