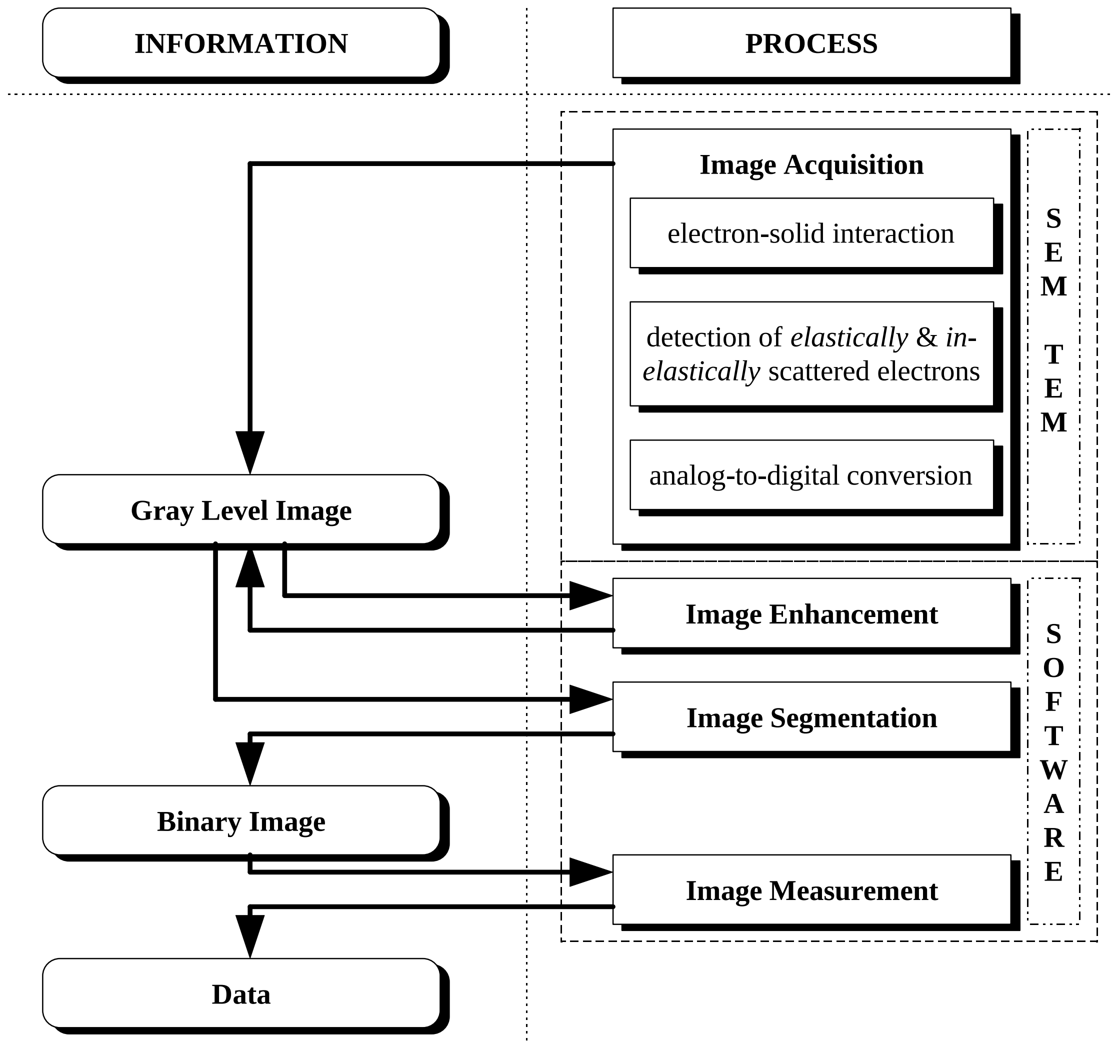
Part 1

IMAGE PROCESSING TECHNIQUES

This part deals with the formation, acquisition and processing of images. Its contents can be best represented as a diagram where the evolution of the considered information (images) and the processes involved are shown.



1.1. Basics of image formation

Since only the images obtained by a scanning electron microscope (SEM) and a transmission electron microscope (TEM) were used in this work and since both techniques are well-established, only a brief introduction is given on the principles and instrumentation of SEM and TEM aiming to show what kind of information is expressed through the images obtained by these techniques.

An image is the optical representation of an object illuminated by a radiation source. The following elements are present in an image formation process: an object, a radiation source (visible light, X-rays, electrons, etc.) and an image formation system. The mathematical model which describes the image formation, depends on the radiation source, on the physics of the radiation-object interaction and on the acquisition system used. A beam of high energy electrons is the radiation source in SEM and TEM. When it strikes a sample a number of phenomena occur simultaneously [1.1.1-1.1.10]. Fig. 1.1.1a, which is taken form [1.1.4], shows different signals generated by the interaction of an electron beam with a solid. The electron interaction volume and the volume from which the different signals are originating are shown in Fig. 1.1.1b which is taken from [1.1.3].

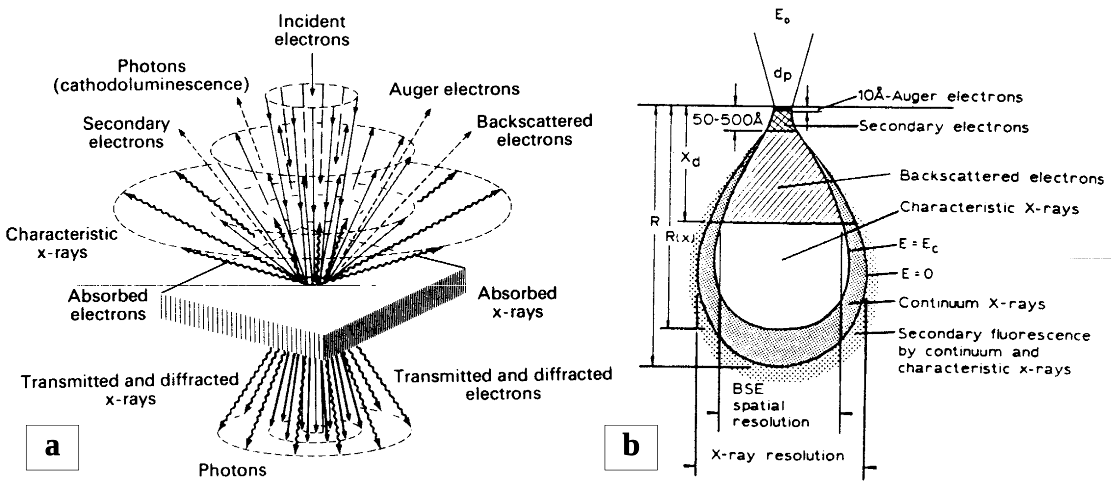


Fig. 1.1.1. a) signals generated by the interaction of an electron beam with a solid; **b)** electron interaction volume and volume from which the different signals are originating.

When electrons enter a material, they interact with the constituent atoms via electrostatic (Coulomb) forces. Most of the primary electrons dissipate their energy as heat but the electron-specimen interaction also yields different types of electrons and electromagnetic waves as a result of *elastic* or *inelastic* scattering events. After elastic scattering, which occurs mainly by interaction of the primary electrons with the electrostatic field of the nucleus, primary electrons change their direction with low energy losses. Inelastic scattering is caused by the interactions of the incident electrons with the nucleus and with the inner- or outer-shell electrons. High-energy electrons can penetrate deep into the atoms and eject inner-shall electrons forming thereby an excited atom. Outer-shell electrons can also undergo single-electron excitation. Due to the ejection of the inner-shell electron a vacancy is formed in the inner electron shell which is filled by an electron from one of the neighboring shells. Owing to this electron transfer, some energy which is equivalent to the energy difference of the two shells, is dissipated. This energy difference is released either as an X-ray or as an Auger electron. The energy of the X-rays and the Auger electrons are specific for the element which produces them.

1.1.1. Image formation in SEM

An image in SEM is obtained by scanning the fine focused electron beam over the surface of the specimen and the simultaneous registration of the signals from the detectors. At each point of the specimen the beam dwells for some fixed time during which the electrons of the beam interact with the specimen. As it was said, a number of phenomena occur as the result of the elastic and inelastic scattering of the primary electrons. If in the case of elastic scattering the scattering angle exceeds 90°, the electron is said to be *backscattered* and may emerge from the specimen close to the point where it entered. The efficiency of elastic scatter events increases with the atomic number of the specimen. A region containing elements with a high atomic number will produce more backscattered electrons than a region with a low atomic number. Therefore *chemical phases* can be recognized in backscattered electron images based on atomic number differences. Since the backscattered electrons are coming from deeper in the specimen (Fig. 1.1.1b), the interaction volume is much larger than the beam diameter and, as a consequence, the resolution in the backscattered image is at most of the order of 200 nm. If in the case of inelastic scattering the final stage of the transitions of the electrons lies above the vacuum level of the solid and if the excited atomic electron has enough energy to reach the surface of the specimen, it may be emitted as a *secondary electron*. The secondary electrons can only escape from a very shallow depth (Fig. 1.1.1b). The intensity of the secondary electron emission is little influenced by the composition of the specimen but is highly dependent on the orientation of the sample surface with the respect to the detector. This makes that they provide important *topographical information* about the surface of the specimen. The low exit depth allows the resolution of the order of 5-20 nm to be reached.

Backscattered electrons are detected by a set of 2 solid state detectors, mounted close above the specimen. Their construction is shown in Fig. 1.1.2a which is taken from [1.1.2]. The electron beam passes through the hole and the backscattered electrons hit the detector and produce a current. The signals which are obtained by both detectors, can be combined into two types of images, topographical and compositional. The topographical image originates from the difference in the incident and backscattered angle and can be obtained by subtracting the signals from both detectors. The compositional image relies on the atomic number dependence of the backscattered electrons and can be obtained by adding the detector signals.

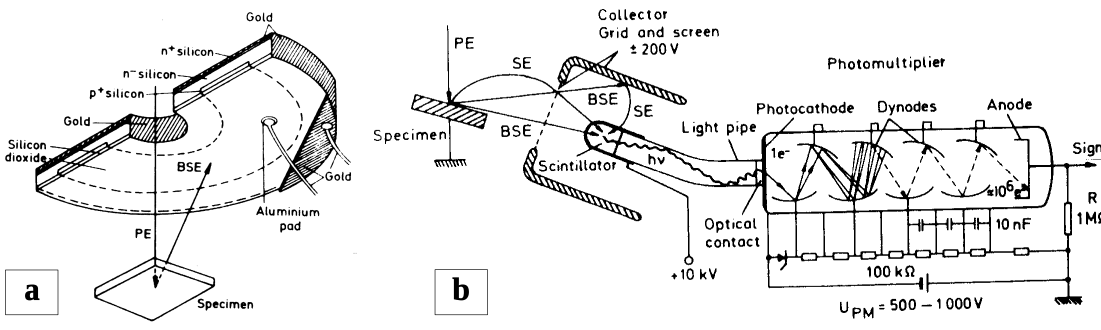


Fig. 1.1.2. a) a circular solid state detector which is used to detect backscattered electrons; **b)** scintillator-photomultiplier combination which is used for recording secondary electrons.

The secondary electrons are detected by a scintillator-photomultiplier combination which is known as the Everhart-Thornley detector. Its construction is shown in Fig. 1.1.2b which is taken form [1.1.2]. The secondary electrons are collected by a grid. The electrons which pass through the collector grid, are accelerated to the scintillator and basically they generate photons interacting with the scintillator. A Faraday cage is placed round the scintillator in order to avoid deflection of the primary beam. The generated photons are converted into an electrical signal.

As it was said, an image in SEM is obtained by scanning the fine focused electron beam over the surface of the specimen and the simultaneous registration of the signal from a detector. Switching between different detectors allows to observe the backscattered or secondary electron images. The signal which was formed in the detector, is suitably amplified and used to modulate the intensity of a cathode ray tube (CRT) which is scanned in synchronism with the electron beam and, thus, a SEM image is formed. At the exit of the head amplifier the video signal is normally proportional to the number of electrons recorded. This signal can be used not only to modulate the intensity of a CRT, but also can be converted to a digital form and the image can be stored digitally. This is done by an analogue-to-digital converter [1.1.2, 1.1.11] which is usually connected to a computer and, therefore, the digitized image can be directly transferred to the computer. A digital image is represented as a two-dimensional data array where each data point is called a picture element or *pixel*. A digitized SEM image consists of pixels where the intensity (range of gray) of each pixel is proportional to the number of the backscattered (in a backscattered electron image) or secondary (in a secondary electron image) electrons, emitted from the corresponding point on the surface of a specimen. Such images are called *gray level images* and usually only 256 levels of gray are used where 0 corresponds to black and 255 corresponds to white. All image processing, described in this work, is done on images of this type.

1.1.2. Image formation in TEM

If a specimen is sufficiently thin and the energy of the incident electrons is high enough, most of the incident electrons will pass through the specimen with little or no energy loss and can be 'visualized' on a fluorescent screen or registered on a photographic plate located below the specimen. The TEM image obtained in this way is called a *conventional electron micrograph* and is, basically, the result of the scattering, diffraction and absorption of the electrons as they pass through the specimen. Different regions of the specimen scatter the electrons by various degrees where the extent of the scattering depends on the local elemental composition and the local thickness and density of the specimen. The TEM is used to investigate ultra-thin samples, typically less then 200 nm. The image resolution depends on the sample thickness and the aberrations of the lenses and is typically of the order of 1-2 nm.

In a conventional transmission electron microscope (CTEM) the image formation process is based primarily on the contrast arising form the elimination of the elastically scattered electrons. Inelastically scattered electrons are indistinguishable from the unscattered electrons and are also imaged into the final image plane. Modern transmission electron microscopes are equipped with the imaging electron energy spectrometer (filter of the prism/mirror/prism type) which allows to work in the electron spectroscopic imaging (ESI) mode so that the electrons of selected energy loss can be visualized. The magnetic field of the prism disperses the electron beam according to the energy of the electrons and the electrons which did not loss energy, are deflected over 90°, reflected by the electrostatic field of the mirror and deflected again by the magnetic field into the optical axis of the microscope. Electrons which have suffered energy losses are slower, so the magnetic field of the prism deflects them at larger angles. After the spectrometer they move outside the optical axis and are caught by the spectrometer slit. Only the unscattered electrons (electrons which suffered no energy loss) reach the final image plane and such imaging is called *zero-loss electron spectroscopic imaging* (ZLESI). Zero-loss filtering not only increases the contrast, also a better comparison with simulated images is possible. To image with electrons which have lost an energy ΔE

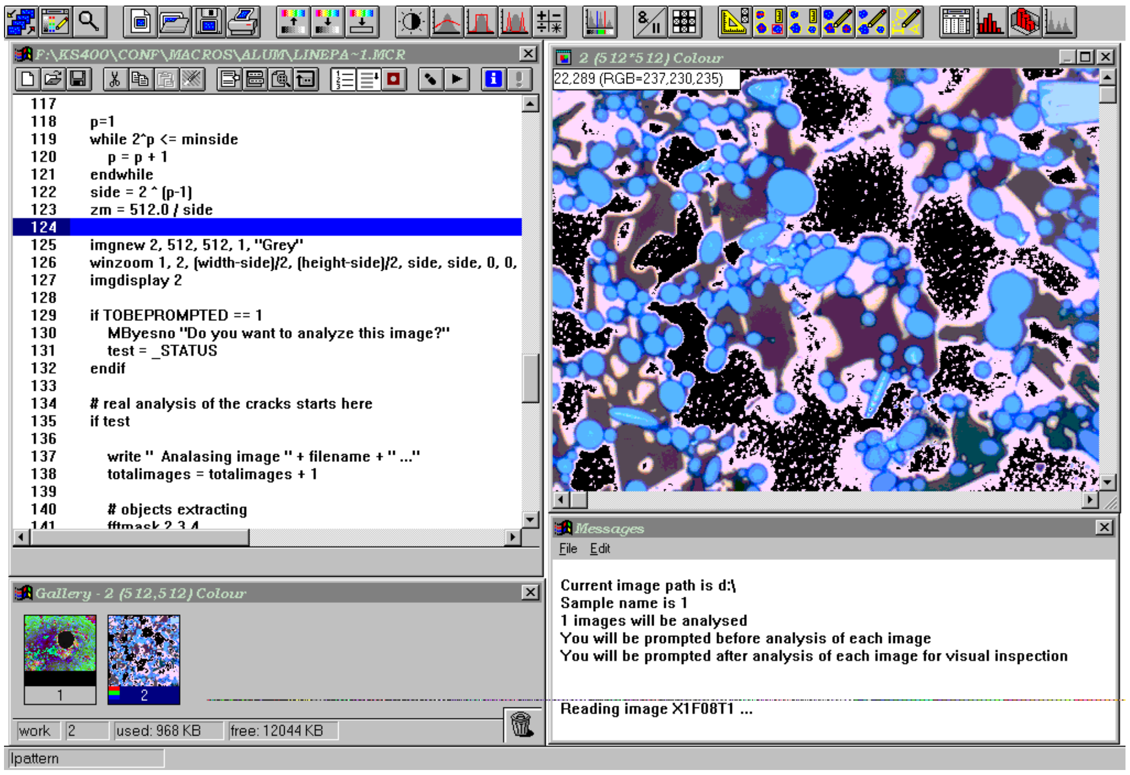
the accelerating voltage E of the primary electron beam is increased to $E + \Delta E$ and the beam electrons which have lost energy ΔE in the specimen, enter the spectrometer with an energy E. They stay on the optical axis of the microscope and pass through the spectrometer slit into the final image plate. Thus, by varying the accelerating voltage, electrons of selected energy loss can be used for imaging. The choice between zero-loss filtering and the use of an energy-loss window depends on the specimen and the information wanted.

1.2. Image processing software

Different commercial general purpose and specialized image processing/analysis software packages are available on the market. For many practical applications commercially available software is the best choice. However, for some of the applications, described in Part 3, no commercial programs are available. Therefore additional software was developed.

1.2.1. KS400

One of commercial image processing software packages used for some of the applications is the *KONTRON Imaging System KS400* [1.2.1]. An example of a typical screen of the system is shown in Fig. 1.2.1.



```

p=1
while 2^p <= minside
   p = p + 1
endwhile
side = 2 ^ (p-1)
zm = 512.0 / side

imgnew 2, 512, 512, 1, "Grey"
winzoom 1, 2, (width-side)/2, (height-side)/2, side, side, 0, 0,
imgdisplay 2

if TOBEPROMPTED == 1
   MByesno "Do you want to analyze this image?"
   test = _STATUS
endif

# real analysis of the cracks starts here
if test

   write "  Analasing image " + filename + " ..."
   totalimages = totalimages + 1

   # objects extracting
   fftmask 2 3 4
```

```
Current image path is d:\
Sample name is 1
1 images will be analysed
You will be prompted before analysis of each image
You will be prompted after analysis of each image for visual inspection

Reading image X1F08T1 ...
```

Fig. 1.2.1. Commercial software *KONTRON Imaging System KS400*

Been very powerful and convenient in use, the KS400 software has one essential disadvantage: it does not allow to have direct access form a macro to single pixels in the image. In practice this means that the functionality of the software is limited by the set of standard functions. In principle, the problem can be overcome by using *Free Programming KONTRON Software Development Kit* which is supplied with KS400. However, the usage of this kit is basically equivalent to writing our own applications except of the implementation of some elements of the user interface. Therefore, image processing software was written instead of writing additional components for KS400.

1.2.2. Image processing/analysis software developed

Software was written implementing new techniques of image processing and analysis. Some applications were written for the *SPARK Station 20* running the *UNIX Solaris 2.4* operation system. Other programs were developed for *IBM-compatible PC* running the *Microsoft Windows 3.x/95* operation systems.

The software written for UNIX consists of a set of low-level image processing/analysis functions including

- Sun raster file image (RAS) reading/writing;
- automatic and manual image thresholding;
- gray-scale and binary morphology;
- fractal analysis of contours using 'hand and dividers' method;
- fractal analysis of percolation networks;
- image correlation.

Some of these functions were later incorporated into a PGT-IMIX software used to control a JEOL JSM-6300 scanning electron microscope. All C codes, given in this Part, are taken from this package. They were compiled by the *cc 3.0.1* compiler for the *UNIX Solaris 2.4* operation system on a *SPARK Station 20.*

The software written for Windows operation system includes a number of specialized programs. One of them is *Image Processing System* a typical view of which is shown in Fig. 1.2.2. The software is written in C++ using MFC and developed with Visual C/C++ v. 1.5 for the Windows 3.1 operation system. The system works with images stored using Windows bitmap image file format (BMP) and allows to perform

- different image manipulations: rotating, flipping, resizing, cropping;
- histogram modifications: brightness/contract adjustment, autoscaling and equalization;
- image filtering: median, mean, shrink and swell filtering, matrix filtering;
- binarization: manual, autothreshold using clustering and correlation criteria;
- binary morphology: erosion, dilation, thinning, watershed segmentation.

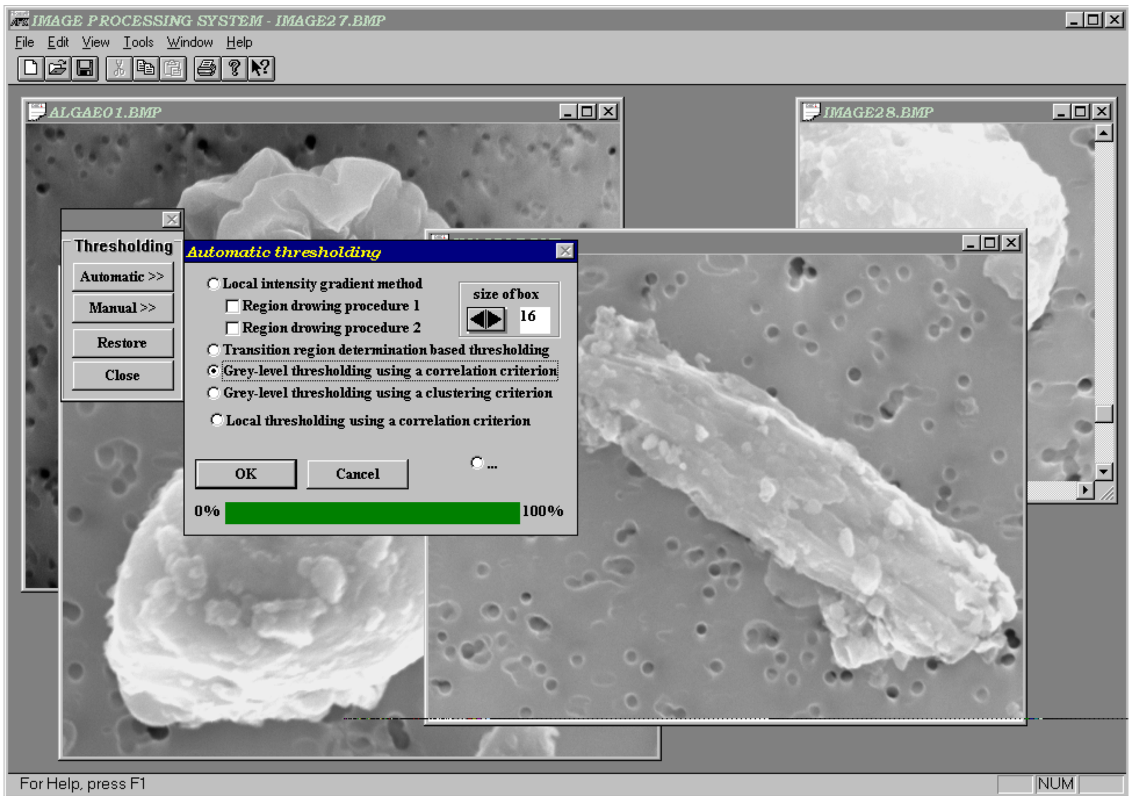


IMAGE PROCESSING SYSTEM - IMAGE27.BMP
File Edit View Tools Window Help
ALGAE01.BMP
IMAGE28.BMP
Thresholding
Automatic >>
Manual >>
Restore
Close
Automatic thresholding
Local intensity gradient method
Region drawing procedure 1
Region drawing procedure 2
Transition region determination based thresholding
Grey-level thresholding using a correlation criterion
Grey-level thresholding using a clustering criterion
Local thresholding using a correlation criterion
OK
Cancel
0%
100%
For Help, press F1
NUM

( ) Local intensity gradient method
☐ Region drawing procedure 1
☐ Region drawing procedure 2
( ) Transition region determination based thresholding
[✓ Grey-level thresholding using a correlation criterion]
( ) Grey-level thresholding using a clustering criterion
( ) Local thresholding using a correlation criterion

size of box: 16

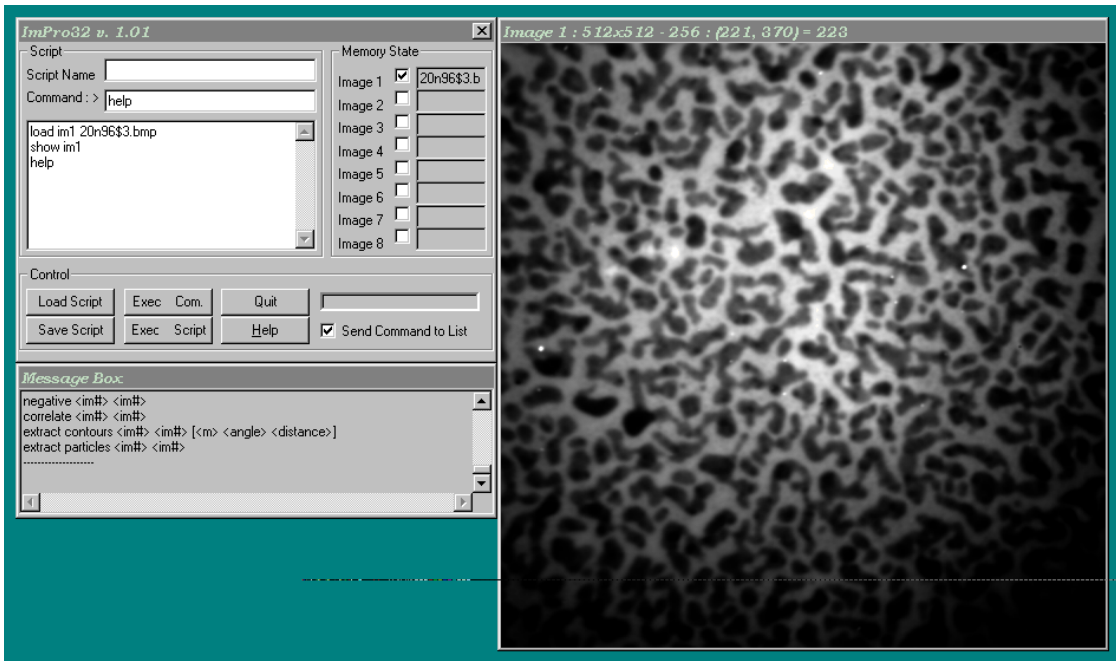
Fig. 1.2.2. Multi-purpose image processing and analysis software *Image Processing System*.

The program is able to show different kind of information about the image such as its gray level histogram, horizontal and vertical line profiles, etc. It also includes some advanced image analysis functions such as Fourier analysis and correlation analysis. All the functions are available from pull down menus as is standard in a Windows application.

Another program, a typical view of which is shown in Fig. 1.2.3, is called *ImPro32*. It is written in C++ using MFC and developed with Visual C/C++ v. 4.0 for the Windows 95 operation system. The program uses a 'command line' user interface. It allows to record different single commands as a script which than can be run in the 'execute script' mode. Among simple image manipulation and processing functions the program has two advanced functions which allow to extract objects and their contours from binary images and to perform a segmentation of overlapping objects of known shape.

To perform a fractal analysis of microscopic objects a program called *Fractals* was developed.

Its typical view is shown in Fig. 1.2.4. The software was written in C++ using MFC and developed with Visual C/C++ v. 1.5 for the Windows 3.1 operation system. The program has two different fractal analysis methods for contours: 'hybrid' and 'box counting', and the 'covering set' method of fractal analysis of the interior part of objects. It has a typical Windows-style user interface with pull down menu.



Command : > help

```
load im1 20n96$3.bmp
show im1
help
```

Image 1 [✓] 20n96$3.b

[✓] Send Command to List

```
negative <im#> <im#>
correlate <im#> <im#>
extract contours <im#> <im#> [<m> <angle> <distance>]
extract particles <im#> <im#>
--------------------
```

Fig. 1.2.3. Specialized image processing and analysis software *ImPro32.*

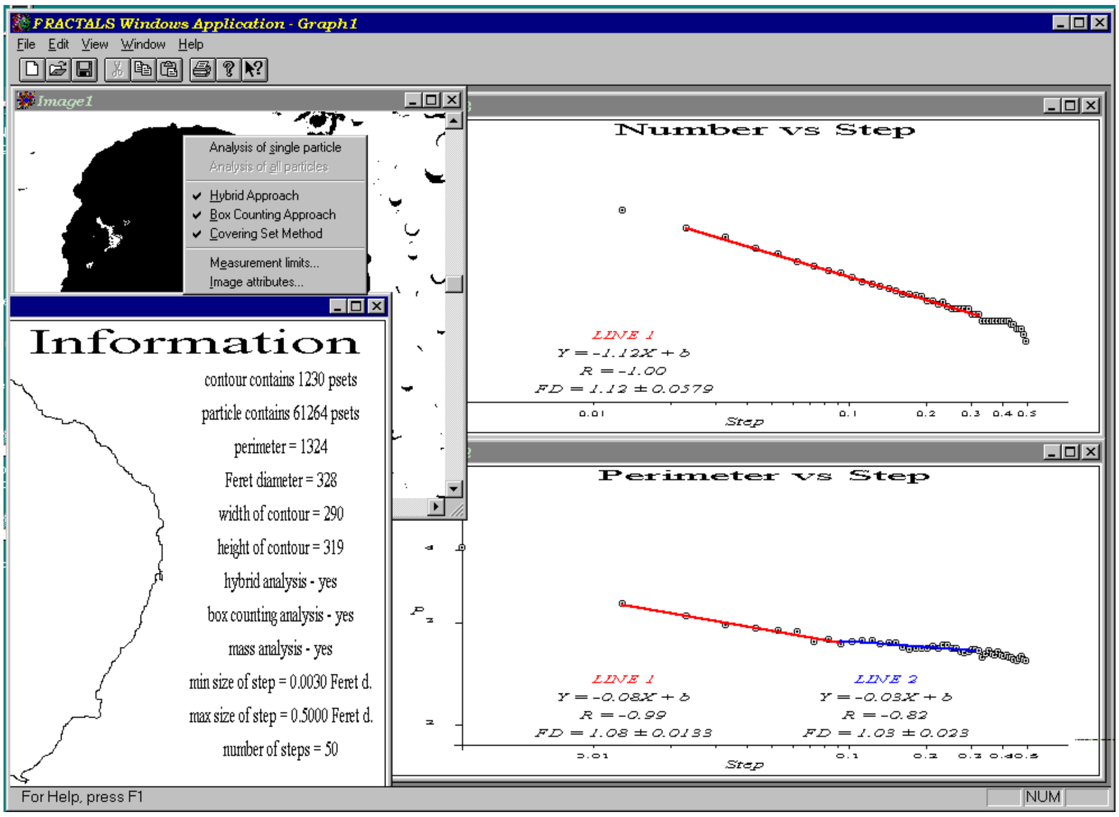
An essential feature of this program is the ability to perform a visual inspection of Richardson plot (more details about this subject are given in Parts 2 and 3) in a very convenient way. The program allows to choose straight line intervals just by pointing with the mouse to appropriate points on the Richardson plot. After this the program recalculates the corresponding slope(s) of the chosen line(s), the corresponding fractal dimension(s) and the uncertainty.

For some of our applications it was necessary to simulate different objects having knows fractal properties. To perform such modeling a program called *Simul*, was developed. Its typical view is shown in Fig. 1.2.5. The software was written in C++ using MFC and developed with Visual C/C++ v. 2.0 for the Windows 3.1 operation system. Different simulation models are implemented including

- two models of simulation for percolation networks;

- simulation of diffusion-limited particle-cluster aggregation;
- simulation of diffusion-limited deposition.

The parameters for each simulation model can be set within the corresponding dialog boxes. With this software it is possible to visualize different steps of the simulation, to apply additional rules to the simulation models, etc.



Information

contour contains 1230 psets
particle contains 61264 psets
perimeter = 1324
Feret diameter = 328
width of contour = 290
height of contour = 319
hybrid analysis - yes
box counting analysis - yes
mass analysis - yes
min size of step = 0.0030 Feret d.
max size of step = 0.5000 Feret d.
number of steps = 50

Fig. 1.2.4. Specialized image analysis software *Fractals.*

1.2.3. Image processing with MathCAD and MatLab.

The MatLab [1.2.2] and MathCAD [1.2.3] environments are ideally suited to image processing. In particular, MatLab's matrix-oriented language is well suited for manipulating images, which are nothing more than visual renderings of matrices. The result is a very easy and economical way of expressing image processing operations. In addition both programs have Image Processing Toolboxes which provide a powerful and flexible environment for image processing and analysis. Both programs were used to perform different calculations on images, for example, MathCAD was used to generate all examples of contour functions discussed in Part 2.

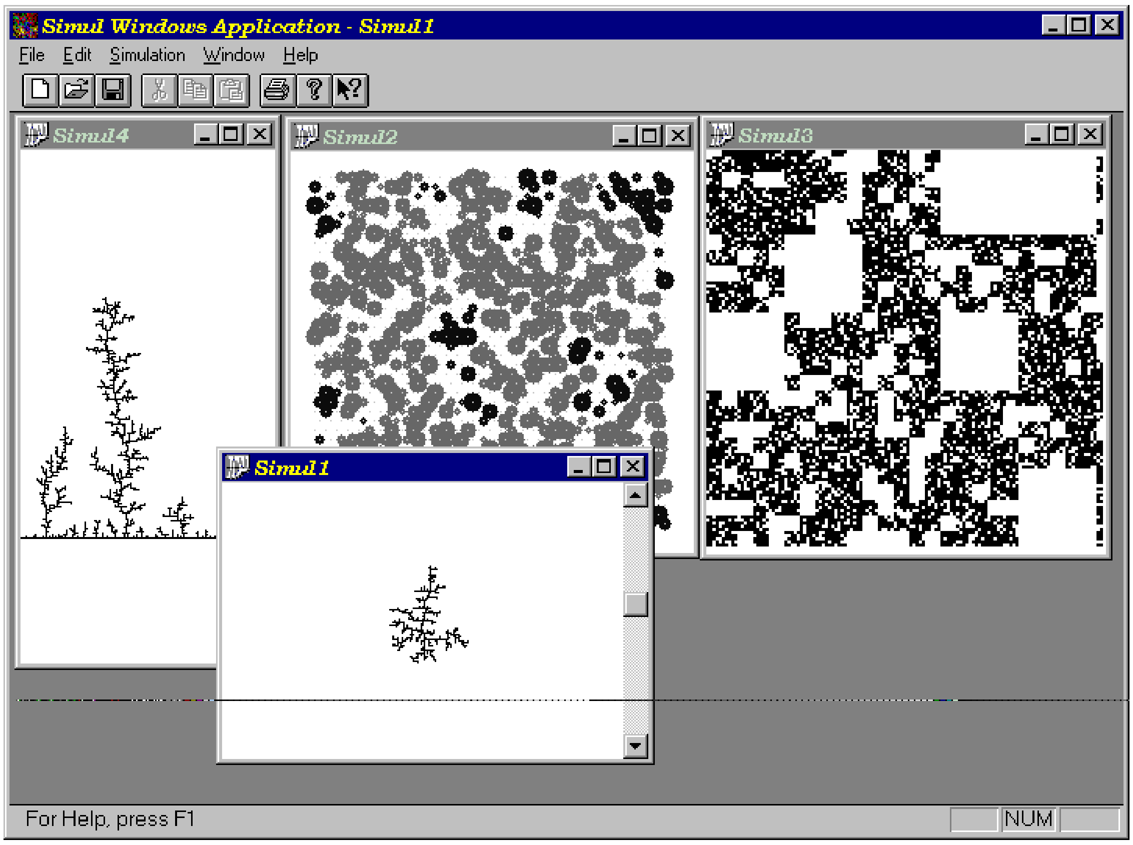

Simul Windows Application - Simul1
File Edit Simulation Window Help
Simul4
Simul2
Simul3
Simul1
For Help, press F1
NUM

Fig. 1.2.5. Specialized fractal simulation software *Simul.*

There are several advantages of using MathCAD and MatLab for image analysis. One of them is the ability to have direct access to any portion of available information what in general is not possible with many commercial image analysis systems. With these programs it is possible to stop any calculations at any time, change a portion of the calculation procedure and then restart the calculations from the point which was affected by the changes without recompiling the code as it usually happens with programming in C, or even restarting the calculations from the beginning. Such ability is very useful for research and the development of new techniques. However, the main disadvantage of these programs is the relatively slow computational speed compared to compiled C code. It is caused by the need of the code to be translated first into a machine code and only then to be executed. Therefore, complex image processing applications can be better implemented using high level programming languages such as C or C++, rather than using software like MatLab or MathCAD.

1.3. Image storage and manipulation

An important issue in image processing is the image storage problem. Many image file formats were developed over the past decades aiming to represent images in a compact and economical way with the ability to work with such representation on different platforms. Since the bitmap file format is a standard image file format for all Microsoft Windows operation systems (Windows 3.x/95/NT) and the raster file format is a standard image file format for all Sun's versions of UNIX operation system and computers running these two operation systems were used, only these two image file formats are used and described here. Since the Windows bitmap file format is well documented and examples of different applications which are able to read and write images using this file format, can be found in the literature [1.3.1-1.3.5], only a brief description is given here. The Sun raster file format is not that well documented. Therefore it is shown in some details how images, presented in this format, can be retrieved, stored and manipulated. All the examples of different image processing functions given in this Part are based this image file format.

1.3.1. Windows bitmap file format

Windows bitmap files are stored in a device-independent bitmap (DIB) format. The term 'device independent' means that the bitmap specifies pixel color in a form independent of the method used by a display to represent color. Each bitmap file contains a bitmap-file header, a bitmap-information header, a color table, and an array of bytes that defines the bitmap bits. The file has the following form:

```
BITMAPFILEHEADER    bmfh;
BITMAPINFOHEADER    bmih;
RGBQUAD             aColors[];
BYTE                aBitmapBits[];
```

The bitmap-file header contains information about the type, size, and layout of a device-independent bitmap file. The header is defined as a *BITMAPFILEHEADER* structure:

```
typedef struct tagBITMAPFILEHEADER {     /* bmfh */
        UINT     bfType;          /* Specifies the type of file. This member must be BM.  */
        DWORD    bfSize;          /* Specifies the size of the file, in bytes. */
        UINT     bfReserved1;     /* Reserved; must be set to zero. */
        UINT     bfReserved2;     /* Reserved; must be set to zero. */
        DWORD    bfOffBits;       /* Specifies the byte offset from the BITMAPFILEHEADER
                                     structure to the actual bitmap data in the file. */
} BITMAPFILEHEADER;
```

The bitmap-information header, defined as a *BITMAPINFOHEADER* structure, specifies the dimensions, compression type, and color format for the bitmap:

```
typedef struct tagBITMAPINFOHEADER {    /* bmih */
        DWORD     biSize;               /* Specifies the number of bytes required by the
                                           BITMAPINFOHEADER structure. */
        LONG      biWidth;              /* Specifies the width of the bitmap, in pixels. */
        LONG      biHeight;             /* Specifies the height of the bitmap, in pixels. */
        WORD      biPlanes;             /* Specifies the number of planes for the target
                                           device. This member must be set to 1. */
        WORD      biBitCount;           /* Specifies the number of bits per pixel. This value
                                           must be 1, 4, 8, or 24. */
        DWORD     biCompression;        /* Specifies the type of compression for a comp-
                                           ressed bitmap. It can be one of the following
                                           values: BI_RGB, BI_RLE8 and BI_RLE4*/
        DWORD     biSizeImage;          /* Specifies the size, in bytes, of the image. */
        LONG      biXPelsPerMeter;      /* Specifies the horizontal resolution, in pixels per
                                           meter, of the target device for the bitmap. */
        LONG      biYPelsPerMeter;      /* Specifies the vertical resolution… */
        DWORD     biClrUsed;            /* Specifies the number of color indexes in the color
                                           table actually used by the bitmap. */
        DWORD     biClrImportant;       /* Specifies the number of color indexes that are
                                           considered important for displaying the bitmap. */
} BITMAPINFOHEADER;
```

Here *biCompression* Specifies the type of compression for a compressed bitmap. It can be one of the following values:

- *BI_RGB*: specifies that the bitmap is not compressed.
- *BI_RLE8*: specifies a run-length encoded format for bitmaps with 8 bits per pixel.
- *BI_RLE4*: specifies a run-length encoded format for bitmaps with 4 bits per pixel.

The *biSizeImage* specifies the size, in bytes, of the image. It is valid to set this member to zero if the bitmap is in the *BI_RGB* format. The *biXPelsPerMeter* and *biYPelsPerMeter* specifie the horizontal and vertical resolution, in pixels per meter, of the target device for the bitmap. An application can use these values to select a bitmap from a resource group that best matches the characteristics of the current device. The *biClrUsed* specifies the number of color indexes in the color table actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the *biBitCount* member. If the *biClrUsed* member is nonzero, it specifies the actual number of colors that the graphics engine or device driver will access if the *biBitCount* member is less than 24. If *biBitCount* is set to 24, *biClrUsed* specifies the size of the reference color table used to optimize performance of Windows color palettes. If the bitmap is a packed bitmap the *biClrUsed* member must be set to zero or to the actual size of the color table. The *biClrImportant* specifies the number of color indexes that are considered important for displaying the bitmap. If this value is zero, all colors are important.

The *biBitCount* member of the *BITMAPINFOHEADER* structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. These members can have any of the following values:

- *1*: bitmap is monochrome and the color table contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the color table. If the bit is set, the pixel has the color of the second entry in the table.
- *4*: bitmap has a maximum of 16 colors. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
- *8*: bitmap has a maximum of 256 colors. Each pixel in the bitmap is represented by a 1-byte index into the color table. For example, if the first byte in the bitmap is 0x1F, the first pixel has the color of the thirty-second table entry.
- *24*: bitmap has a maximum of 2^{24} colors. Each 3-byte sequence in the bitmap array represents the relative intensities of red, green, and blue, respectively, for a pixel.

The color table, defined as an array of *RGBQUAD* structures, contains as many elements as there are colors in the bitmap:

```
typedef struct tagRGBQUAD {     /* rgbq */
        BYTE        rgbBlue;            /* Specifies the intensity of blue in the color. */
        BYTE        rgbGreen;           /* Specifies the intensity of green in the color. */
        BYTE        rgbRed;             /* Specifies the intensity of red in the color. */
        BYTE        rgbReserved;        /* Not used; must be set to zero. */
} RGBQUAD;
```

The color table is not present for bitmaps with 24 color bits because each pixel is represented by 24-bit red-green-blue (RGB) values in the actual bitmap data array. The colors in the table should appear in order of importance. This helps a display driver render a bitmap on a device that cannot display as many colors as there are in the bitmap.

The bitmap bits, immediately following the color table, consist of an array of *BYTE* values representing consecutive rows, or 'scan lines', of the bitmap. Each scan line consists of consecutive bytes representing the pixels in the scan line, in left-to-right order. The number of bytes representing a scan line depends on the color format and the width, in pixels, of the bitmap. If necessary, a scan line must be zero-padded to end on a 32-bit boundary. However, segment boundaries can appear anywhere in the bitmap. The scan lines in the bitmap are stored from bottom to up. This means that the first byte in the array represents the pixels in the lower-left corner of the bitmap and the last byte represents the pixels in the upper-right corner.

1.3.2. Sun raster file format

A raster file is composed of three parts: first, a header containing 8 integers; second, a (possibly empty) set of colormap values; and third, the pixel image, stored a line at a time, in increasing y order. It can be defined by the following structure:

```
struct rasterfile {
        struct rasterheader header;
        /* color map follows for ras_maplength bytes, followed by image */
        unsigned char       *ras_red;
        unsigned char       *ras_green;
        unsigned char       *ras_blue;
        /* image data */
        unsigned char       *ras_matrix;
};
```

The image is layered out in the file in the same way as in memory. Each line of the image is rounded up to the nearest 16 bits. The header is defined by the following structure:

```
struct rasterheader {
        int     ras_magic;          /* magic number */
        int     ras_width;          /* width (pixels) of image */
        int     ras_height;         /* height (pixels) of image */
        int     ras_depth;          /* depth (1, 8, or 24 bits) of pixel */
        int     ras_length;         /* length (bytes) of image */
        int     ras_type;           /* type of file; see RT_* below */
        int     ras_maptype;        /* type of colormap; see RMT_* below */
        int     ras_maplength;      /* length (bytes) of following map */
};
```

The *ras_magic* field always contains the following constant:

```
#define RAS_MAGIC 0x59a66a95
```

The *ras_width*, *ras_height*, and *ras_depth* fields contain the image's width and height in pixels, and its depth in bits per pixel, respectively. The depth is either 1 or 8, corresponding to standard frame buffer depths. The *ras_length* field contains the length in bytes of the image data. For an unencoded image, this number is computable from the *ras_width*, *ras_height*, and *ras_depth* fields, but for an encoded image it must be explicitly stored in order to be available without decoding the image itself. The length of the header and of the (possibly empty) color-map values are not included in the value of the *ras_length* field; it is only the image data length.

The following *ras_type*'s are supported:

```
        /* Sun supported ras_type's */
#define RT_OLD              0           /* Raw pixrect image in 68000 byte order */
#define RT_STANDARD         1           /* Raw pixrect image in 68000 byte order */
#define RT_BYTE_ENCODED     2           /* Run-length compression of bytes */
#define RT_FORMAT_RGB       3           /* XRGB or RGB instead of XBGR or BGR */
#define RT_FORMAT_TIFF      4           /* tiff <-> standard raster file */
#define RT_FORMAT_IFF       5           /* iff (TAAC format) <-> standard raster file */
#define RT_EXPERIMENTAL     0xffff      /* Reserved for testing */
```

For historical reasons, files of type *RT_OLD* will usually have a 0 in the *ras_length* field, and software expecting to encounter such files should be prepared to compute the actual image data length if needed.

The *ras_maptype* and *ras_maplength* fields contain the type and length in bytes of the colormap values, respectively. The following *ras_maptype*'s are supported:

```
        /* Sun registered ras_maptype's */
#define RMT_RAW             2
        /* Sun supported ras_maptype's */
#define RMT_NONE            0           /* ras_maplength is expected to be 0 */
#define RMT_EQUAL_RGB       1           /* red[ras_maplength/3],green[],blue[] */
```

If *ras_maptype* is not *RMT_NONE* and the *ras_maplength* is not 0, then the colormap values are the *ras_maplength* bytes immediately after the header. These values are either uninterpreted bytes (usually with the *ras_maptype* set to *RMT_RAW*) or the equal length red, green and blue vectors, in that order (when the *ras_maptype* is *RMT_EQUAL_RGB*). In the latter case, the *ras_maplength* must be three times the size in bytes of any one of the vectors.

The following function can be used to create a new raster file (of the type *RT_STANDARD* & *RMT_EQUAL_RGB*) in memory:

```
struct rasterfile* NewRasterFile(width, height, maplength)
        int                 width;          /* width of the image to be created */
        int                 height;         /* height of the image to be created */
        int                 maplength;      /* number of colors to be used */
{
        struct rasterfile*  localfile;
        int                 index;

        localfile = (struct rasterfile *)malloc(sizeof(struct rasterfile));
        if (localfile == NULL) return localfile;

        localfile->ras_red          = NULL;
        localfile->ras_green        = NULL;
        localfile->ras_blue             = NULL;
        localfile->ras_matrix           = NULL;

        if (width * height != 0) {
                maplength = (maplength > 256) ? 256 : maplength;

                localfile->header.ras_magic       = RAS_MAGIC;      /* magic number */
                localfile->header.ras_width       = width;          /* width (pixels) of image */
                localfile->header.ras_height      = height;         /* height (pixels) of image */
                localfile->header.ras_depth       = 8;              /* depth of pixel */
                localfile->header.ras_length      = width * height; /* length (bytes) of image */
                localfile->header.ras_type        = RT_STANDARD;        /* type of file */
                localfile->header.ras_maptype     = RMT_EQUAL_RGB;      /* type of colormap */
                localfile->header.ras_maplength = maplength * 3; /* length (bytes) of following map */

                localfile->ras_red    = (unsigned char *)calloc(localfile->header.ras_maplength/3, 1);
                localfile->ras_green= (unsigned char *)calloc(localfile->header.ras_maplength/3, 1);
                localfile->ras_blue  = (unsigned char *)calloc(localfile->header.ras_maplength/3, 1);
                localfile->ras_matrix = (unsigned char *)calloc(localfile->header.ras_length, 1);

                if ((localfile->ras_red == NULL) || (localfile->ras_green == NULL) ||
                        (localfile->ras_blue == NULL) || (localfile->ras_matrix == NULL))
                                DeleteRasterFile(localfile);
                else {
                        for (index = 0; index < localfile->header.ras_maplength/3; index++) {
                                *(localfile->ras_red + index) = (unsigned char)index;
                                *(localfile->ras_green + index) = (unsigned char)index;
                                *(localfile->ras_blue + index) = (unsigned char)index;
                        }
                }
        }

        return localfile;
}
```

The following function can be used to delete a raster file from memory:

```
void DeleteRasterFile(localfile)
        struct rasterfile*      localfile;
{
        if (localfile != NULL) {
                if (localfile->ras_red != NULL)         free(localfile->ras_red);
                if (localfile->ras_green != NULL)       free(localfile->ras_green);
                if (localfile->ras_blue != NULL)        free(localfile->ras_blue);
                if (localfile->ras_matrix != NULL)      free(localfile->ras_matrix);
                free(localfile);
                localfile = NULL;
        }
        return;
}
```

The following function can be used to read a raster file (of the type *RT_STANDARD* &

RMT_EQUAL_RGB) from disk to memory:

```
struct rasterfile* ReadRasterFile(filename)
        char*                   filename;
{
        struct rasterfile*      localfile = NULL;
        FILE*                   diskfile;
        int                     width, height;
        int                     length, maplength;

        if ((diskfile = fopen(filename, "r")) == NULL) return localfile;        /* cannot open file */
        if ((localfile = NewRasterFile(0, 0, 0)) == NULL) {
                fclose(diskfile);
                return localfile;                               /* cannot create new raster file */
        }
        if (fread(&localfile->header, sizeof(struct rasterheader), 1, diskfile) != 1) {
                fclose(diskfile);
                DeleteRasterFile(localfile);
                return localfile;                               /* cannot read rasheader */
        }

        localfile->header.ras_magic     = swaplong(&localfile->header.ras_magic);
        localfile->header.ras_width     = swaplong(&localfile->header.ras_width);
        localfile->header.ras_height    = swaplong(&localfile->header.ras_height);
        localfile->header.ras_depth     = swaplong(&localfile->header.ras_depth);
        localfile->header.ras_length    = swaplong(&localfile->header.ras_length);
        localfile->header.ras_type      = swaplong(&localfile->header.ras_type);
        localfile->header.ras_maptype   = swaplong(&localfile->header.ras_maptype);
        localfile->header.ras_maplength = swaplong(&localfile->header.ras_maplength);

        if ((localfile->header.ras_magic != RAS_MAGIC) ||
            (localfile->header.ras_type != RT_STANDARD) ||
            (localfile->header.ras_maptype != RMT_EQUAL_RGB) ||
            (localfile->header.ras_depth != 8)) {
                fclose(diskfile);
                DeleteRasterFile(localfile);
                return localfile;                               /* not supported type of raster file */
        }

        width = localfile->header.ras_width;
        height = localfile->header.ras_height;
        length = localfile->header.ras_length;
        maplength = localfile->header.ras_maplength/3;

        DeleteRasterFile(localfile);
        if ((localfile = NewRasterFile(width, height, maplength)) == NULL) {
                fclose(diskfile);
                return localfile;                               /* cannot create new raster file */
        }

        if ((fread(localfile->ras_red, 1, maplength, diskfile) != maplength) ||
           (fread(localfile->ras_green, 1, maplength, diskfile) != maplength) ||
            (fread(localfile->ras_blue, 1, maplength, diskfile) != maplength)) {
                fclose(diskfile);
                DeleteRasterFile(localfile);
                return localfile;                               /* cannot read color table */
        }

        if (fread(localfile->ras_matrix, 1, length, diskfile) != length) {
                fclose(diskfile);
                DeleteRasterFile(localfile);
                return localfile;                               /* cannot read image */
        }

        fclose(diskfile);
        return localfile;
}

long swaplong(l) long* l; {
        unsigned char* b; unsigned long l_swapped;
        b = (unsigned char *)l; l_swapped = 256L*256L*256L*(long)b[0] + 256L*256L*(long)b[1] +
        256L*(long)b[2] + (long)b[3];
        return (long)l_swapped;
}
```

The following function can be used to write a raster file to disk:

```
typedef int BOOL;
#define TRUE        (BOOL)1
#define FALSE       (BOOL)0

BOOL WriteRasterFile(filename, localfile)
        char*                   filename;
        struct rasterfile*      localfile;
{
        FILE*                   diskfile;
        int                     ras_length, ras_maplength;
        BOOL                    flag = FALSE;

        if (localfile == NULL) return FALSE;                            /* no data */
        if ((diskfile = fopen(filename, "w")) == NULL) return FALSE;    /* cannot open file */

        ras_length              = localfile->header.ras_length;
        ras_maplength           = localfile->header.ras_maplength/3;

        localfile->header.ras_magic     = swaplong(&localfile->header.ras_magic);
        localfile->header.ras_width     = swaplong(&localfile->header.ras_width);
        localfile->header.ras_height    = swaplong(&localfile->header.ras_height);
        localfile->header.ras_depth     = swaplong(&localfile->header.ras_depth);
        localfile->header.ras_length    = swaplong(&localfile->header.ras_length);
        localfile->header.ras_type      = swaplong(&localfile->header.ras_type);
        localfile->header.ras_maptype   = swaplong(&localfile->header.ras_maptype);
        localfile->header.ras_maplength = swaplong(&localfile->header.ras_maplength);
        if (fwrite(&localfile->header, sizeof(struct rasterheader), 1, diskfile) == 1)
                if ((fwrite(localfile->ras_red, 1, ras_maplength, diskfile) == ras_maplength) &&
                    (fwrite(localfile->ras_green, 1, ras_maplength, diskfile) == ras_maplength) &&
                    (fwrite(localfile->ras_blue, 1, ras_maplength, diskfile) == ras_maplength) &&
                    (fwrite(localfile->ras_matrix, 1, ras_length, diskfile) == ras_length))
                         flag = TRUE;                                        /* file is saved */

        localfile->header.ras_magic       = swaplong(&localfile->header.ras_magic);
        localfile->header.ras_width       = swaplong(&localfile->header.ras_width);
        localfile->header.ras_height      = swaplong(&localfile->header.ras_height);
        localfile->header.ras_depth       = swaplong(&localfile->header.ras_depth);
        localfile->header.ras_length      = swaplong(&localfile->header.ras_length);
        localfile->header.ras_type        = swaplong(&localfile->header.ras_type);
        localfile->header.ras_maptype     = swaplong(&localfile->header.ras_maptype);
        localfile->header.ras_maplength = swaplong(&localfile->header.ras_maplength);

        fclose(diskfile);
        return flag;
}
```

The following function can be used to make a copy of a raster file in memory:

```
struct rasterfile* CopyRasterFile(originalfile)
        struct rasterfile*      originalfile;
{
        struct rasterfile*      copyfile;
        int                     index;

        int width = originalfile->header.ras_width;
        int height = originalfile->header.ras_height;
        int length = originalfile->header.ras_length;
        int maplength = originalfile->header.ras_maplength/3;

        if ((copyfile = NewRasterFile(width, height, maplength)) == NULL) return copyfile;

        for (index = 0; index < maplength; index++) {
                *(copyfile->ras_red + index) = *(originalfile->ras_red + index);
                *(copyfile->ras_green + index) = *(originalfile->ras_green + index);
                *(copyfile->ras_blue + index) = *(originalfile->ras_blue + index);
        }

        for (index = 0; index < length; index++)
                *(copyfile->ras_matrix + index) = *(originalfile->ras_matrix + index);

        return copyfile;
}
```

To have a direct access to the pixels inside of raster file the following 4 functions can be used:

```
BOOL SetPset(localfile, ras_x, ras_y, ras_color)
        struct rasterfile*      localfile;
        int                     ras_x, ras_y;
        unsigned char           ras_color;
{
        if (localfile == NULL) return FALSE;
        if ((ras_x < 0) || (ras_x >= localfile->header.ras_width)) return FALSE;
        if ((ras_y < 0) || (ras_y >= localfile->header.ras_height)) return FALSE;
        if (ras_color >= localfile->header.ras_maplength/3) return FALSE;

        *(localfile->ras_matrix + ras_y * localfile->header.ras_width + ras_x) =
        (unsigned char)ras_color;

        return TRUE;
}

BOOL GetPset(localfile, ras_x, ras_y, ras_color)
        struct rasterfile*      localfile;
        int                     ras_x, ras_y;
        unsigned char*          ras_color;
{
        if (localfile == NULL) return FALSE;
        if ((ras_x < 0) || (ras_x >= localfile->header.ras_width)) return FALSE;
        if ((ras_y < 0) || (ras_y >= localfile->header.ras_height)) return FALSE;

        *ras_color =  *(localfile->ras_matrix + ras_y * localfile->header.ras_width + ras_x);

        return TRUE;
}

void SetPsetFast(localfile, ras_x, ras_y, ras_color)
        struct rasterfile*      localfile;
        unsigned int            ras_x, ras_y;
        unsigned char           ras_color;
{
        *(localfile->ras_matrix + ras_y * localfile->header.ras_width + ras_x) =
        (unsigned char)ras_color;
}

void GetPsetFast(localfile, ras_x, ras_y, ras_color)
        struct rasterfile*      localfile;
        unsigned int            ras_x, ras_y;
        unsigned char*          ras_color;
{
        *ras_color = *(localfile->ras_matrix + ras_y * localfile->header.ras_width + ras_x);
}
```

All the codes given here were compiled with the *cc 3.0.1* compiler under *UNIX Solaris 2.4* operation system on a *SPARK Station 20.*

1.4. Image enhancement

One of the very important topics in image processing is image enhancement. Image enhancement involves a collection of techniques that are used to improve the visual appearance of an image, or to convert the image to a form which is better suited for human or machine interpretation. There is no general theory of image enhancement because there is no general standard for the quality of an image. Therefore, different classes of techniques were developed over the past decades [1.4.1-1.4.8]. Some of them, which are most frequently used in practice, are discussed below.

1.4.1. Gray level histogram modifications

A first-order global characteristic of an image is its *gray level histogram*. The gray level histogram of an image is a chart, listing all of the gray levels that are used in the image on the horizontal axis and indicating the number of pixels having each level on the vertical axis. The gray level histogram of the image shown in Fig. 1.4.1a is shown as right inset. Gray level images usually consist of 256 levels of gray, so, the horizontal axis of the histogram runs from 0 to 255. The vertical axis varies in scale depending on the number of pixels in the image and the distribution of the gray level values.

A set of techniques, called *gray level histogram modifications*, is used to improve the visual appearance of an image. The techniques are based on remapping the gray levels within an image by applying a transformation function. The function can either be linear or nonlinear. The brightness of an image can be changed by applying the following function: $g(x,\ y) = f(x,\ y) + C$ where C is a constant. When $C > 0$ the resulting image will be brighter and when $C < 0$ the resulting image will be darker. An example of the application of such a transformation ($C = 125$) to the image in Fig. 1.4.1a is shown in Fig. 1.4.1b. As seen from the histogram (shown as right inset in Fig. 1.4.1b) all the gray levels in the original histogram are shifted to the right by 125, but the histogram still has the same shape. The contrast of an image can be changed by applying the following transformation function: $g(x,\ y) = K \cdot f(x,\ y)$ where K is a constant. When $K > 1$ the resulting image has a higher contrast comparing to the original image, and when $K < 1$ the resulting image has a lower contrast. Finally, the following transformation function is often used: $g(x,\ y) = K \cdot f(x,\ y) + C$. Here the variable K adjusts the contrast and the variable C adjusts the brightness of the image. A common linear transformation function

$$g(x,\ y) = \frac{255}{\max f(x,\ y) - \min f(x,\ y)} \cdot \left(f(x,\ y) - \min f(x,\ y) \right),$$

is called *autoscaling*. An example of the autoscaled image is given in Fig. 1.4.1c. As seen from the histogram shown as right inset in Fig. 1.4.1c, the gray level histogram of the modified image is shifted to the left and rescaled over the range of all available gray levels compared to the gray level histogram of the original image shown in Fig. 1.4.1a.

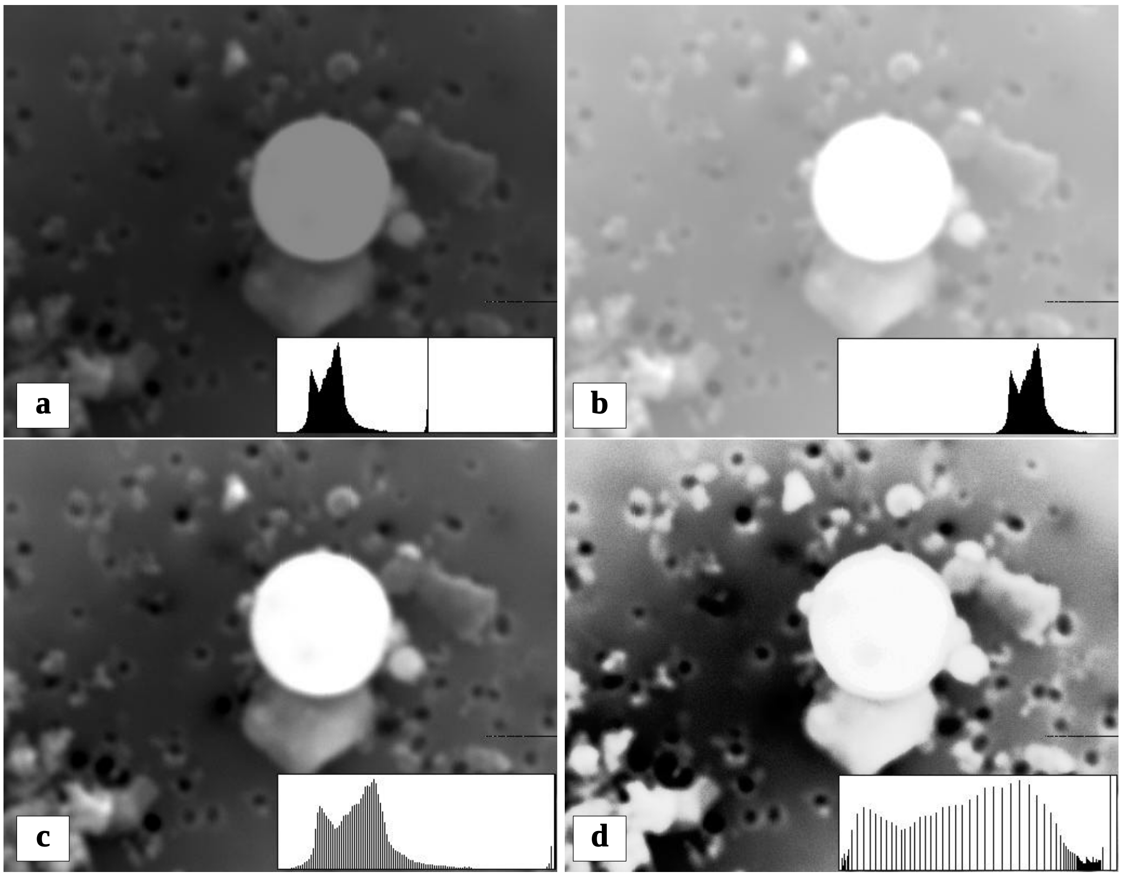


Fig. 1.4.1. a) original image; **b)** the same image after brightness correction; **c)** results of autoscaling of the gray level histogram; **d)** image after histogram linearization.

Using different values of K and C different transformation curves can be obtained. If the transformed gray level values exceed the acceptable gray level range (usually from 0 to 255) the output gray levels are set to the minimal and maximal allowable gray levels.

There are images in which nearly the full possible range of levels is used, but most of the levels are associated with only a few pixels. To improve the visual 'quality' of such images the *histogram equalization* is often applied. Gray level values are transformed relative to one another with the goal to make the resulting gray level histogram 'as flat as possible' and to space the peaks evenly over the range all gray levels. The following transformation function can be applied to perform the histogram equalization:

$$g(x,\ y) = \frac{m-1}{\sum_{j=0}^{m} n_j} \cdot \sum_{j=0}^{f(x,\ y)} n_j$$

where m is the total number of gray levels in the original image f and n_j is the number of pixels having particular gray level j.

Descriptions of many other histogram transformations can be found in the literature [1.4.1-1.4.8]. For example, a histogram linearization is performed with the aim to 'linearize' the histogram of transformed image. An example of an image having a linearized gray level histogram is shown in Fig. 1.4.1d.

1.4.2. Smoothing of noisy images

Noise is the random variation in the pixel content caused by the acquisition, digitization and transmission process. An example of a 'noisy' image is shown in Fig. 1.4.2a. As seen from the image, pixels which are affected by noise often appear markedly different from their neighbor pixels. Noise cannot be eliminated altogether, it can be reduced by a process called *smoothing*. The simples way of noise reduction is to analyze a few nearest neighbor pixels of each pixel in the image and to see if the difference between the gray value of the pixel and the gray values of its nearest neighbors is essential. Often smoothing by averaging is applied. Here each pixel is replaced by the average of its neighbor pixels. Another effective way of noise reduction is median filtering when each pixel is replaced by the median of its neighbor pixels. The image, shown in Fig. 1.4.2b, is obtained by the median filtering of the image shown in Fig. 1.4.2a.

There is a class of spatial filters called *low-pass filters*, which are often applied for image smoothing. Their application is based on the *discrete convolution* of the original image with a special mask. The discrete convolution of an image f with a mask h produces an image g:

$$g(x,y) = \sum_{i=-v}^{v} \sum_{j=-u}^{u} h(i,j) * f(x+i,\ y+j)$$

where h is a $2v+1$ by $2u+1$ matrix. The output $g(x,\ y)$ at a point $(x,\ y)$ is given by a weighted sum of input pixels around $(x,\ y)$ where the weights are given by the $h(i,\ j)$. In practice this equation is implemented as a series of shift-multiply-sum operations. The values of h are referred to as the filter kernel.

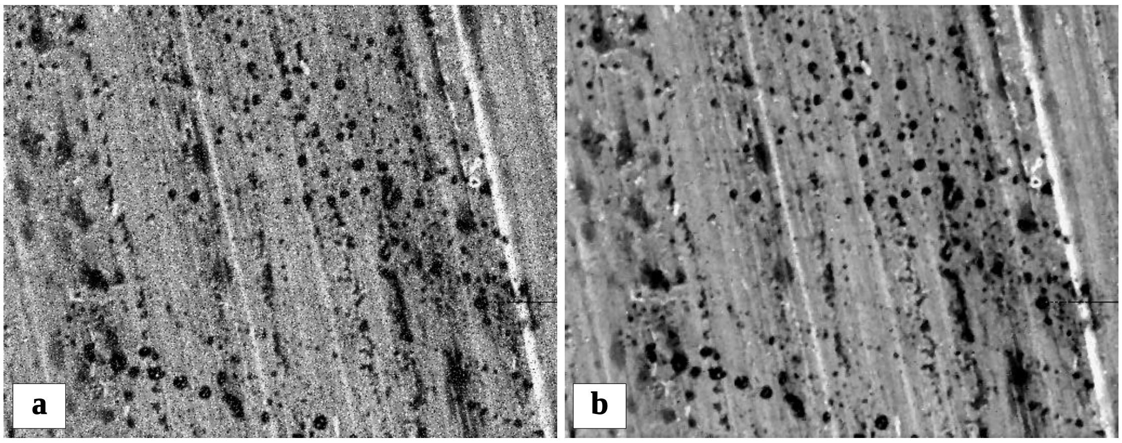


Fig. 1.4.2. a) image with 20% random noise added; **b)** the same image after applying the median filter.

C code, implementing the discrete convolution technique is given below.

```
struct rasterfile* DiscreteConvolution(image, mask, masksizeX, masksizeY)
        struct rasterfile*      image;
        float*                  mask;
        int                     masksizeX, masksizeY;
{
        struct rasterfile*      newimage = NULL;
        int                     x, y, i, j, u, v;
        float                   S;
        unsigned char           pixelvalue;

        /* is image valid? */
        if (image == NULL) return newimage;
        if ((newimage = CopyRasterFile(image)) == NULL) return newimage;

        v = (masksizeX - 1 ) / 2; u = (masksizeY - 1 ) / 2;

        /* convolution */
        for (x = 0; x < image->header.ras_width; x++)
                for (y = 0; y < image->header.ras_height; y++) {
                        S = 0.0F;
                        for (i = -v; i <= +v; i++)
                                for (j = -u; j <= +u; j++) {
                                        if (GetPset(image, x+i, y+j, &pixelvalue))
                                                S += pixelvalue * mask[(j+u) * masksizeX + i];
                                }
                        pixelvalue = (S < image->header.ras_maplenght/3) ?
                                        ((S > 0) ? (unsigned char)S : (unsigned char)0) :
                                        (unsigned char)(image->header.ras_maplenght/3 - 1);
                        SetPsetFast(image, x, y, pixelvalue);
                }
        return newimage;
}

/* an example of a mask of the size 3x3 */
float mask[] = { 1/9, 1/9, 1/9,    1/9, 1/9, 1/9,    1/9, 1/9, 1/9 };
```

C code 1.4.1. The function *DiscreteConvolution* requires an image, a mask and its size and returns an image which is the result of the discrete convolution of the original image with the mask.

The effect of the convolution depends on the type of filter kernel used. To produce a smoothing effect the following filters can be used:

$$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}, \begin{pmatrix} \frac{1}{10} & \frac{1}{10} & \frac{1}{10} \\ \frac{1}{10} & \frac{1}{5} & \frac{1}{10} \\ \frac{1}{10} & \frac{1}{10} & \frac{1}{10} \end{pmatrix}, \begin{pmatrix} 0 & \frac{1}{8} & 0 \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ 0 & \frac{1}{8} & 0 \end{pmatrix}, \begin{pmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{pmatrix}.$$

An interesting application of a smoothing is a *shading correction*. An example of an image for which a shading correction is necessary, is shown in Fig. 1.4.3a.

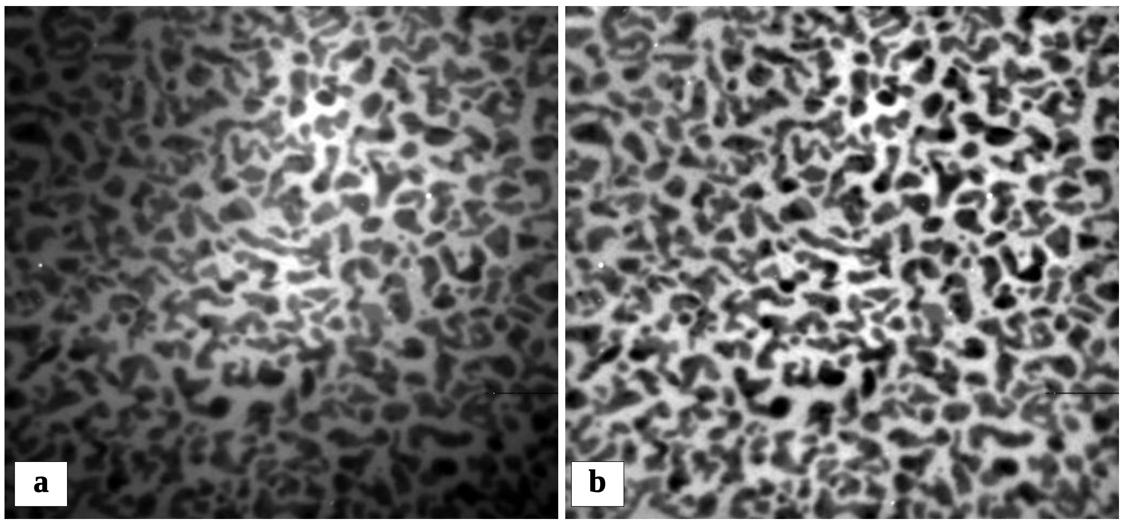


Fig. 1.4.3. Shading correction **a)** original image; **b)** the same image after shading correction.

Shading correction can be done in the following way. First, the original image is smoothed applying a few times (9 for the given example) a low-pass filter of a large size (49x49 for the given example). Second, the difference between the original and the smoothed images is calculated and the resulting image is autoscaled. The result of such a sequence of operations is the shadow-corrected image shown in Fig. 1.4.3b.

Since noise reduction in images is a very important topic in practical image analysis a lot of different methods of smoothing were developed. Their description can be found in the literature [1.4.1-1.4.8].

1.4.3. Sharpening

Similarly to smoothing there is a class of spatial filters called *high-pass filters*, which are often applied for image sharpening. Their application is also based on the discrete convolution of the original image with a special mask. To produce a sharpening effect the following filters can be used:

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}, \begin{pmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{pmatrix}.$$

An example of an image where a sharpening may be necessary is shown in Fig. 1.4.4a. The resulted image with enhanced edges is shown in Fig. 1.4.4b. Another way of making an image sharper is to apply edge detection filters such as the Laplace, Roberts, Sobel filters [1.4.1-1.4.8], and to add the found edges to the original image.

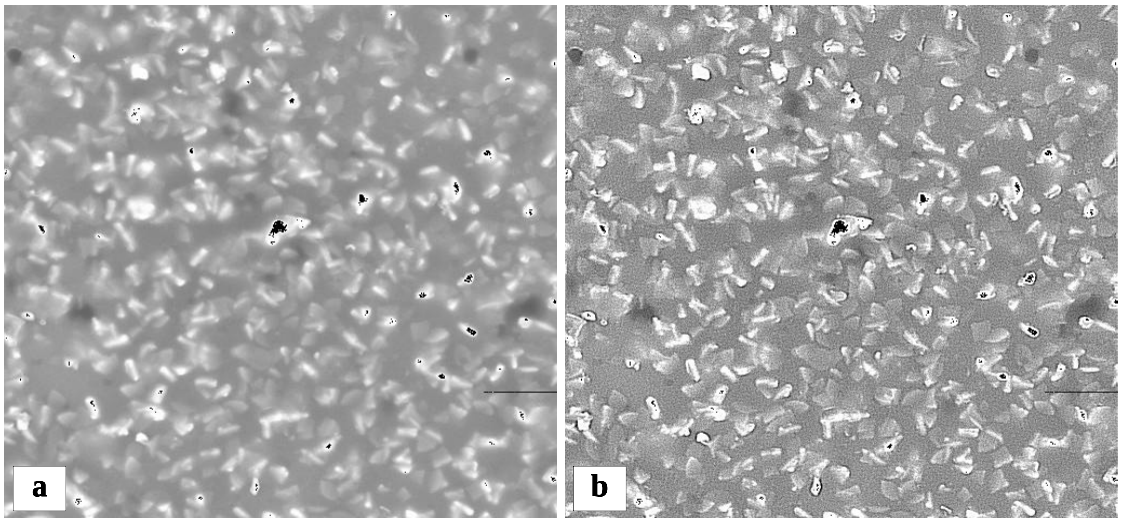


Fig. 1.4.4 a) original image; **b)** the same image after applying a sharpening filter.

1.5. Image segmentation

Segmentation is one of the key problems in image processing. A popular method used for image segmentation is thresholding. After thresholding a *binary image* is formed where all object pixels have one gray level and all background pixels have another - generally the object pixels are 'black' and the background is 'white'. The best threshold is the one that selects all the object pixels and maps them to 'black'. Various approaches for the automatic selection of the threshold have been proposed [1.5.1-1.5.10].

Thresholding can be defined as mapping of the gray scale into the binary set $\{0,\ 1\}$:

$$S(x,\ y) = \begin{cases} 0, & if \quad g(x,\ y) < T(x,\ y) \\ 1, & if \quad g(x,\ y) \geq T(x,\ y) \end{cases}$$

where $S(x,\ y)$ is the value of the segmented image, $g(x,\ y)$ is the gray level of the pixel $(x,\ y)$ and $T(x,\ y)$ is the threshold value at the coordinates $(x,\ y)$. In the simplest case $T(x,\ y)$ is coordinate independent and a constant for the whole image. It can be selected, for instance, on the basis of the gray level histogram. When the histogram has two pronounced maxima, which reflect gray levels of object(s) and background, it is possible to select a single threshold for the entire image. A method which is based on this idea and uses a correlation criterion to select the best threshold, is described below. Sometimes gray level histograms have only one maximum. This can be caused, e.g., by inhomogeneous illumination of various regions of the image. In such case it is impossible to select a single thresholding value for the entire image and a local binarization technique (described below) must be applied. General methods to solve the problem of binarization of inhomogeneously illuminated images, however, are not available.

Segmentation of images involves sometimes not only the discrimination between objects and the background, but also separation between different regions. One method for such separation is known as watershed segmentation [1.5.11-1.5.14] the basic principles of which are described below.

1.5.1. Global thresholding using a correlation criterion

Concerning the thresholding problem, let g represents the possible gray values in the original image. These values are characterized by the below- and above-threshold means $\mu_0(T)$ and $\mu_1(T)$ of the original image, given by

$$\mu_0(T) = \sum_{g=0}^{T} g p_g \left(\sum_{g=0}^{T} p_g \right)^{-1} \text{ and } \mu_1(T) = \sum_{g=T+1}^{n} g p_g \left(\sum_{g=T+1}^{n} p_g \right)^{-1}$$

where $g = 0, 1, \ldots, n$ are gray values and T $(0 < T < n)$ is the threshold level. The probability distribution p_g of gray values g is given by $p_g = f_g / N$ where N is the total number of pixels in the image and f_g is the number of pixels having gray value g. The expected values become

$$E_x = \sum_{g=0}^{n} g p_g\,,\; E_y(T) = \sum_{g=0}^{T} \mu_0(T) p_g + \sum_{g=T+1}^{n} \mu_1(T) p_g\,,\; E_{xx} = \sum_{g=0}^{n} g^2 p_g\,,$$

$$E_{xy}(T) = \sum_{g=0}^{T} g\mu_0(T) p_g + \sum_{g=T+1}^{n} g\mu_1(T) p_g \text{ and } E_{yy}(T) = \sum_{g=0}^{T} \mu_0^2(T) p_g + \sum_{g=T+1}^{n} \mu_1^2(T) p_g\,.$$

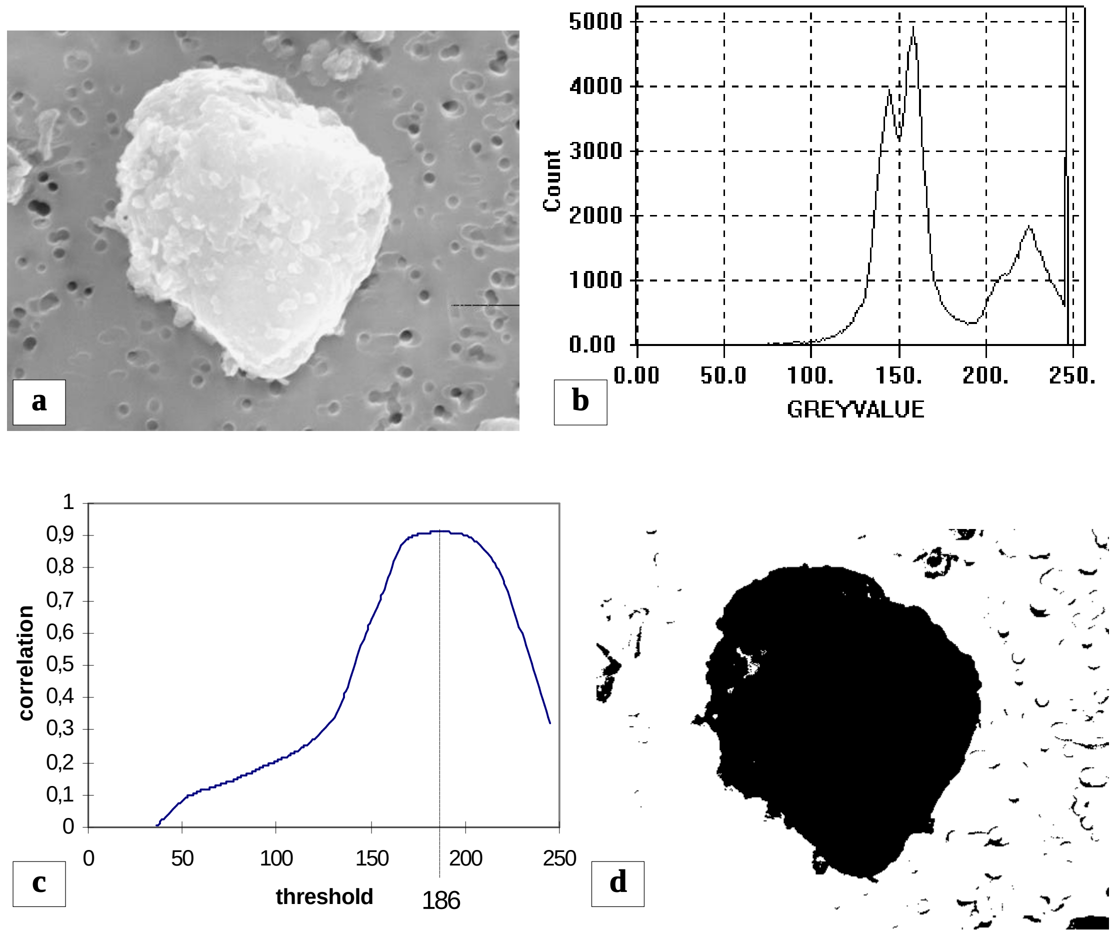


Fig. 1.5.1. The results of applying the correlation criterion based technique **a)** secondary electron SEM image of an urban dust particle; **b)** it gray level histogram; **c)** correlation as a function of the threshold level; **d)** the resulting bilevel image for a threshold of 186.

The variances are given by

$$V_x = E_{xx} - \left(E_x\right)^2,\ V_y(T) = E_{yy}(T) - \left(E_y(T)\right)^2.$$

Actually, E_x, E_{xx} and V_x are independent of the threshold T as they are obtained from the original image. The correlation coefficient given by

$$\rho_{xy}(T) = \frac{E_{xy}(T) - E_x E_y(T)}{\sqrt{V_x V_y(T)}},$$

is now a function of the thresholding level. The optimal value of T corresponds to the value that maximizes the correlation between the original and the bilevel images. This value is found by iteration.

The result of applying the technique to a secondary electron SEM image of an urban dust particle is shown in Fig. 1.5.1. As seen from the example, the correlation method selected a meaningful threshold.

A C implementation of this technique is given below.

```
struct rasterfile* AutoThreshold(image)
        struct rasterfile*      image;
{
        /* local variables */
        struct rasterfile*      newimage = NULL;
        unsigned char           thresh, cmaplength, color, g, T;
        unsigned int            x, y;
        unsigned int*           ColorTable;
        float                   Pg, Ex, Exx, Ey, Eyy, Exy, Vx, Vy, m0, m1, sPg, Rxy, maxRxy;
        unsigned int            length, max_length;

        /* is image valid? */
        if (image == NULL) return FALSE;
        if ((newimage = CopyRasterFile(image)) == NULL) return newimage;

        cmaplength = image->header.ras_maplength / 3;

        /* first, calculate the gray level histogram */
        ColorTable = (unsigned int*)calloc(cmaplength, sizeof(unsigned int));

        if (ColorTable == NULL) {
                DeleteRasterFile(newimage);
                return newimage;
        }

        max_length = image->header.ras_width * image->header.ras_height;

        for (length = 0; length < max_length; length++)
                ColorTable[*(image->ras_matrix + length)] += 1;

        /* calculate threshold independent variables */
        Ex = Exx = maxRxy = Ey = Exy = Eyy = 0;
```

```
for (g = 0; g <= cmaplength; g++) {
        Pg = (float)ColorTable[g] / (image->header.ras_width * image->header.ras_height);
        Ex += g * Pg;
        Exx += g * g * Pg;
}                                               /* Ex, Exx - Ok. */

Vx = Exx - Ex * Ex;                             /* Vx - Ok. */

for (T = 0; T < cmaplength; T++) {
        /* calculate threshold dependent variables */
        m0 = 0; sPg = 0;

        for (g = 0; g <= T; g++) {
                Pg = ColorTable[g] / (image->header.ras_width * image->header.ras_height);
                sPg += Pg;
                m0 += g * Pg;
        }

        if (sPg != 0) m0 /= sPg;                /* m0 - Ok. */
        m1 = 0; sPg = 0;

        for (g = (T + 1); g <= cmaplength; g++) {
                Pg = ColorTable[g] / (image->header.ras_width * image->header.ras_height);
                sPg += Pg;
                m1 += g * Pg;
        }

        if (sPg != 0) m1 /= sPg;                /* m1 - Ok. */

        for (g = 0; g <= T; g++) {
                Pg = ColorTable[g] / (image->header.ras_width * image->header.ras_height);
                Ey += m0 * Pg;
                Exy += g * m0 * Pg;
                Eyy += m0 * m0 * Pg;
        }

        for (g = (T + 1); g <= cmaplength; g++) {
                Pg = ColorTable[g] / (image->header.ras_width * image->header.ras_height);
                Ey += m1 * Pg;
                Exy += g * m1 * Pg;
                Eyy += m1 * m1 * Pg;
        }                                       /* Ey, Exy, Eyy - Ok. */

        Vy = Eyy - Ey * Ey;                     /* Vy - Ok. */

        /* calculate correlation and find the largest value */
        if (Vx*Vy >= 0) {
                Rxy = sqrt(Vx*Vy);
                if (Rxy != 0.0) Rxy = (Exy - Ex * Ey) / Rxy;
                if (Rxy >= maxRxy) {
                        maxRxy = Rxy;
                        thresh = T;             /* thresh - Ok. */
                        }
                }
        }

        /* finally, apply the best threshold */
        for (length = 0; length < max_length; length++)
                if (*(image->ras_matrix + length) >= thresh)
                        *(newimage->ras_matrix + length) = (unsigned char)(cmaplength - 1);
                else
                        *(newimage->ras_matrix + length) = (unsigned char)0;

        free(ColorTable);
        return TRUE;
}
```

C code 1.5.1. The function *AutoThreshold* requires an image and returns the thresholded image.

1.5.2. Local binarization using discrete convolution

This method of binarization is based on the application of the discrete convolution filtering technique that produces a transformed image which can be easy thresholded using 1 as the threshold value. It was found that in order to produce a 'binarization' effect the following kernels can be used:

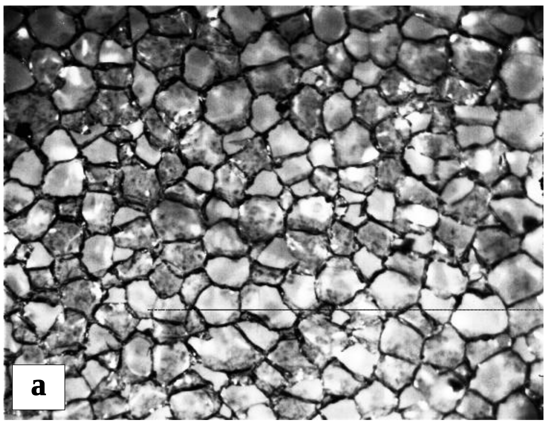
$$\begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -2 & -2 & -2 & -1 \\ -1 & -2 & 32+p & -2 & -1 \\ -1 & -2 & -2 & -2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}, \begin{pmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -2 & -2 & -2 & -2 & -2 & -1 \\ -1 & -2 & -3 & -3 & -3 & -2 & -1 \\ -1 & -2 & -3 & 80+p & -3 & -2 & -1 \\ -1 & -2 & -3 & -3 & -3 & -2 & -1 \\ -1 & -2 & -2 & -2 & -2 & -2 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}.$$

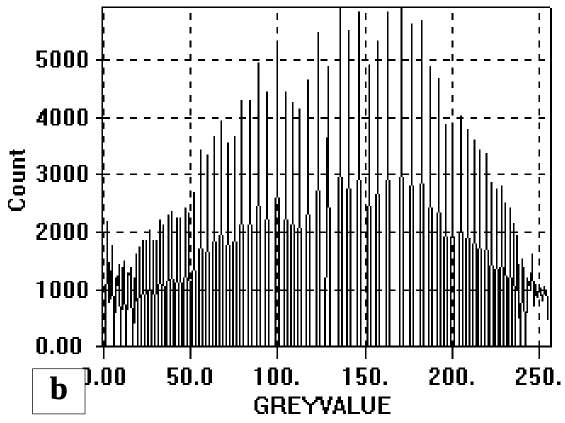
They are constructed according to the following rule: the elements of the kernel are equal to -1 if they belong to the first or last rows or the first or last columns, -2 if they are not defined yet and belong to the second or last but one rows or the second or last but one columns, and so on. The central element of the kernel equals minus the sum of all elements of the kernel plus an additional parameter p. The size of the kernel can be different, but for many applications kernels having the size 5x5 or 7x7 give good results.

For decision making about a suitable numerical value of parameter p, the coefficient of correlation between original and convoluted images is used. It is calculated as follows:

$$r(p) = \frac{Cov(f,\ g(p))}{\sqrt{Var(f) \cdot Var(g(p))}}$$

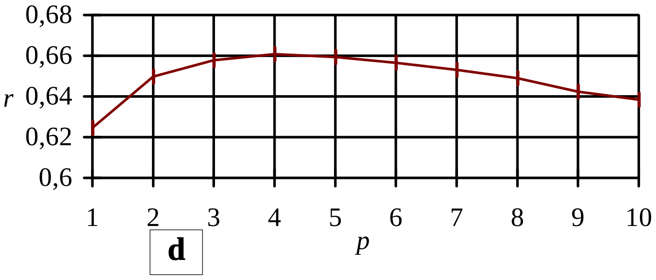
where f and $g(p)$ are the original gray level image and the image convoluted using a value p for the parameter, respectively.

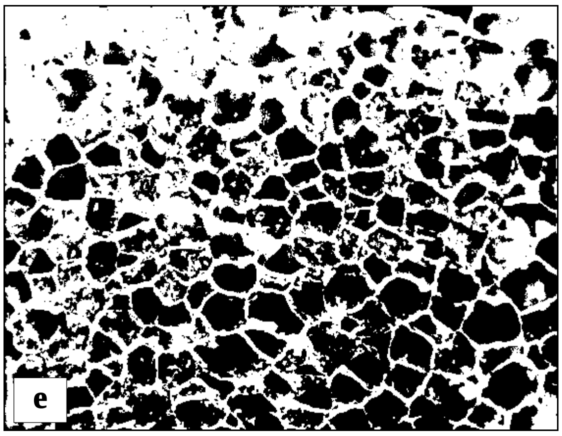




$$\begin{pmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -2 & -2 & -2 & -1 \\ -1 & -2 & 32+p & -2 & -1 \\ -1 & -2 & -2 & -2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

c





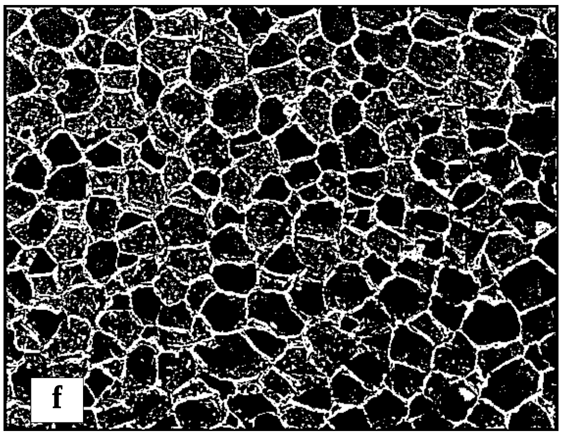
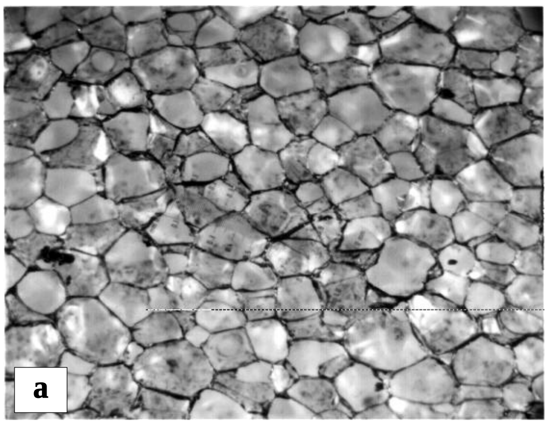
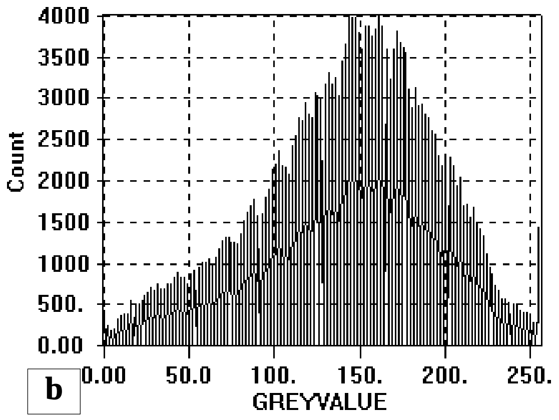


Fig. 1.5.2. a) original image; **b)** its gray level histogram; **c)** convolution kernel; **d)** correlation as a function of the parameter p; **e)** image binarized by global thresholding using the correlation criterion; **f)** image binarized by the local binarization technique.

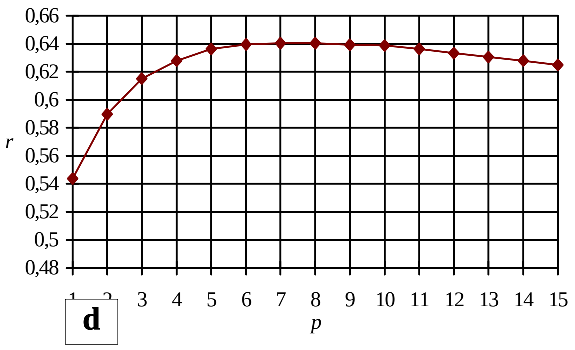
The correlation coefficient $r(p)$ reaches a maximum for a certain value of p. The parameter p corresponding to the maximal correlation is used as a suitable one for the given image.

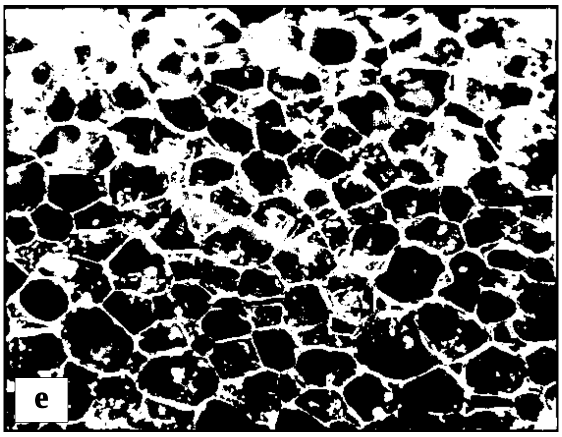




c

$$\begin{pmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -2 & -2 & -2 & -2 & -2 & -1 \\ -1 & -2 & -3 & -3 & -3 & -2 & -1 \\ -1 & -2 & -3 & 80+p & -3 & -2 & -1 \\ -1 & -2 & -3 & -3 & -3 & -2 & -1 \\ -1 & -2 & -2 & -2 & -2 & -2 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$





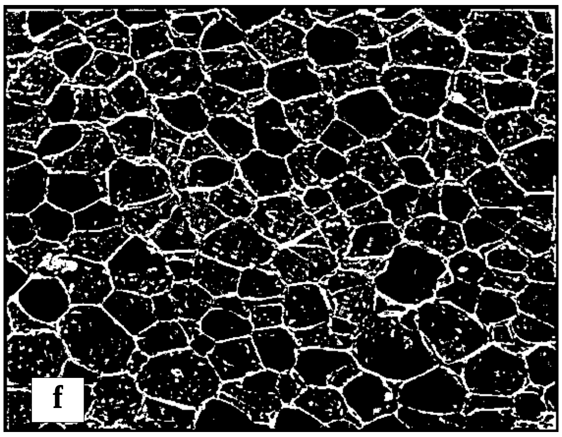


Fig. 1.5.3. a) original image; **b)** its gray level histogram; **c)** convolution kernel; **d)** correlation as a function of the parameter p; **e)** image binarized by global thresholding using the correlation criterion; **f)** image binarized by the local binarization technique.

Figs 1.5.2a and 1.5.3a represent two images taken from the surface of a material with a cell structure. Both images have inhomogeneously illuminated regions; the upper left corner of both images is much darker than the central part. The gray level histogram of both images (Figs 1.5.2b and 1.5.3b, respectively) has no pronounced maxima related to 'objects' and/or background. A single threshold value selection for the whole image does not allow to perform good segmentation (Figs 1.5.2e and 1.5.3e, respectively). The proposed technique gives binary images (Figs 1.5.2f and 1.5.3f) of much better quality than those obtained by the global binarization method.

A C implementation of this technique is given below.

```
struct rasterfile* LocalThreshold(image, masksize)
        struct rasterfile*      image;
        int                     masksize;
{
        /* local variables */
        struct rasterfile*      tmpimage = NULL, newimage = NULL;
        float*                  mask;
        int                     p, i;
        float                   correlation, maxcorrelation = 0.0;

        /* is image valid? */
        if (image == NULL) return newimage;

        /* make an appropriate mask */
        mask = MakeMask(masksize);
        if (mask == NULL) return newimage;

        p = mask[masksize * (masksize - 1) / 2 + (masksize - 1) / 2];

        /* perform convolution & calculate correlation */
        for (i = 1; i < masksize * 2; i++) {
                mask[masksize * (masksize - 1) / 2 + (masksize - 1) / 2] = p + i;
                tmpimage = DiscreteConvolution(image, mask, masksize, masksize);
                if (tmpimage == NULL) break;

                correlation = ImageCorrelation(image, tmpimage);

                if (correlation > maxcorrelation) {
                        maxcorrelation = correlation;
                        if (newimage != NULL) DeleteRasterFile(newimage);
                        newimage = CopyRasterFile(tmpimage);
                }
                DeleteRasterFile(tmpimage);
        }

        free(mask);
        return newimage;
}
```

```
/* MakeMask is used to make a mask of a given size according to the rules described above */
float* MakeMask(masksize)
        int                 masksize;
{
        float*              mask;
        int                 i, j;
        float               summ = 0.0F;

        /* is the size valid? */
        if (masksize % 2 == 1) masksize--;
        masksize = (masksize < 3) ? 3 : ((masksize > 15) ? 15 : masksize );

        /* getting memory for the mask */
        mask = (float*)calloc(masksize * masksize, sizeof(float));
        if (mask == NULL) return mask;

        /* filling in the mask */
        for (i = 0; i < (masksize+1)/2; i++)
                for (j = 0; j < (masksize+1)/2; j++) {
                        mask[i * masksize + j] = -(((i < j) ? i : j) + 1);
                        mask[(masksize - 1 - i) * masksize + j] = -(((i < j) ? i : j) + 1);
                        mask[i * masksize + (masksize - 1 - j)] = -(((i < j) ? i : j) + 1);
                        mask[(masksize - 1 - i) * masksize + (masksize - 1 - j)] = -(((i < j) ? i : j) + 1);
                }

        /* counting the mask's summ */
        for (i = 0; i < masksize; i++)
                for (j = 0; j < masksize; j++)
                        summ += mask[i * masksize + j];

        /* filling in the central element */
        mask[masksize * (masksize - 1) / 2 + (masksize - 1) / 2] = -summ;
        return mask;
}

/* ImageCorrelation is used to calculate the coefficient of correlation between two images */
float ImageCorrelation(image1, image2)
        struct rasterfile*  image1;
        struct rasterfile*  image2;
{
        float               expected_image1, expected_image2;
        float               covariance, correlation;
        float               variance_image1, variance_image2;
        float               k_image1, k_image2, max_k1_k2;

        if (image1 == NULL) return -2.0;
        if (image2 == NULL) return -2.0;

        k_image1 = (float)(image1->header.ras_maplength/3 - 1);
        k_image2 = (float)(image2->header.ras_maplength/3 - 1);

        max_k1_k2 = (k_image1 > k_image2) ? k_image1 : k_image2;
        k_image1 = max_k1_k2 / k_image1;
        k_image2 = max_k1_k2 / k_image2;

        expected_image1 = ExpectedValue(image1, k_image1);
        expected_image2 = ExpectedValue(image2, k_image2);

        variance_image1 = Variance(image1, expected_image1, k_image1);
        variance_image2 = Variance(image2, expected_image2, k_image2);

        covariance = Covariance(image1, expected_image1, k_image1,
                                image2, expected_image2, k_image2);

        correlation = covariance / sqrt(variance_image1 * variance_image2);
        return correlation;
}

/* ExpectedValue is used by ImageCorrelation function */
float ExpectedValue(image, k_image)
        struct rasterfile*      image;
        float                   k_image;
{
        float                   expected = 0.0;
        unsigned int            x, y;
        unsigned char           color;

        for (x = 0; x < image->header.ras_width; x++)
                for (y = 0; y < image->header.ras_height; y++) {
                        GetPsetFast(image, x, y, &color);
                        expected = expected + (float)color * k_image;
                }

        expected /= (image->header.ras_width * image->header.ras_height);
        return expected;
}

/* Variance is used by ImageCorrelation function */
float Variance(image, expected_value, k_image)
        struct rasterfile*      image;
        float                   expected_value;
        float                   k_image;
{
        float                   variance = 0.0;
        unsigned int            x, y;
        unsigned char           color;

        for (x = 0; x < image->header.ras_width; x++)
                for (y = 0; y < image->header.ras_height; y++) {
                        GetPsetFast(image, x, y, &color);
                        variance += ((color * k_image - expected_value) *
                                    (color * k_image - expected_value));
                }

        variance /= (image->header.ras_width * image->header.ras_height);
        return variance;
}

/* Covariance is used by ImageCorrelation function */
float Covariance(image1, expected_value1, k_image1, image2, expected_value2, k_image2)
        struct rasterfile*      image1;
        float                   expected_value1, k_image1;
        struct rasterfile*      image2;
        float                   expected_value2, k_image2;
{
        float                   covariance = 0.0;
        unsigned int            x, y;
        unsigned char           color1, color2;

        for (x = 0; x < image1->header.ras_width; x++)
                for (y = 0; y < image1->header.ras_height; y++) {
                        GetPsetFast(image1, x, y, &color1);
                        GetPsetFast(image2, x, y, &color2);
                        covariance += ((color1 * k_image1 - expected_value1) *
                                        (color2 * k_image2 - expected_value2));
                }

        covariance /= (image1->header.ras_width * image1->header.ras_height);
        return covariance;
}
```

C code 1.5.2. The function *LocalThreshold* requires an image and the size of the mask to be used and returns the thresholded image.

1.5.3. Segmentation based on watershed transform

The method of segmentation based on the use of watershed lines was developed in the framework of mathematical morphology. Consider an image f as a topographic surface and define the catchment basins and the watershed lines in terms of a flooding process. Imagine that each cavity of the surface is pierced and the surface is plunged into a lake with a constant vertical speed. The water entering through the holes floods the surface. The moment that the floods filling two distinct catchment basins start to merge, a dam is erected in order to prevent mixing of the floods. The union of all dams defines the watershed lines of the image f.

There are different computer implementations of watershed algorithms. Basically, they can be divided into two groups: algorithms which simulate the flooding process and procedures aiming at direct detection of the watershed points. A modification of the watershed algorithm for the case of binary images is described in the section 1.6.

1.6. Processing of binary images

Binary images, as described above, consist of groups of 'black' and 'white' pixels selected on the basis of some properties. Usually it is assumed that an object (a particle) consists of 'black' pixels whereas 'white' pixels represent the background. An object in a binary image can be detected by the *bottom-to-top left-to-right procedure* as shown in Fig. 1.6.1a. On a *rectangular grid* a pixel is said to be *four-* or *eight-connected* when it has the same properties as one of its nearest four or eight neighbors as shown in Fig. 1.6.1b. Usually, one considers eight-connectivity for an object and four-connectivity for the background [1.6.1], otherwise there are some difficulties as shown in Fig. 1.6.1c. The *boundary* is a linked edge that characterizes the shape of an object. Two different types of boundaries can be found if one operates on 4- or 8-connected neighborhoods. In the case of 4-connectivity a point of an object is considered to belong to the object's boundary if at least one of its 4-connected neighbor points belongs to the background. In the case of 8-connectivity a point of an object is considered to belong to the object's boundary if at least one of its 8-connected neighbor points belongs to the background. In all applications described in this work, the latter type of boundary is used.

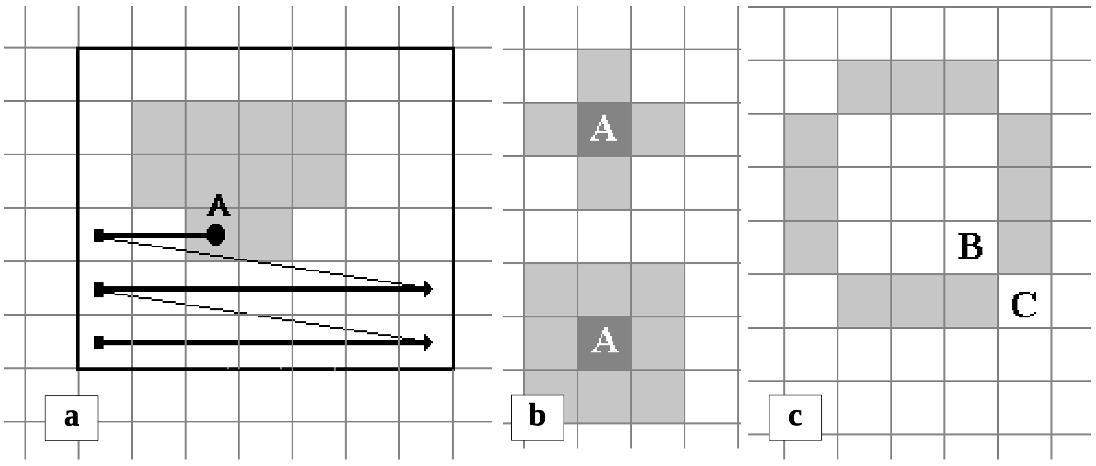


Fig. 1.6.1. a) object detection bottom-to-top left-to-right procedure; **b)** connectivity on a rectangular grid: pixel A and its 4-connected and 8-connected neighbors; **c)** connectivity paradox: “Are B and C connected?”.

1.6.1. Image enhancement

Methods for image enhancement, mentioned for the case of gray level images, can be applied to the binary images too. Of course, they should be slightly adapted in order to be used with 'black' and 'white' gray values. However, there are some specific techniques which are designed specifically for binary images. Some of them are described below.

Binary morphology

Many binary image enhancement procedures are based on morphological operations [1.6.2] such as *erosion* and *dilation*. The simplest kind of erosion, sometimes referred to as classical erosion, is to remove any foreground pixel touching another pixel which is a part of the background. Instead of removing pixels from objects a complementary operation known as dilation can be used to add pixels. The classical dilation adds any background pixel which touches another pixel that is already part of an object.

An *ultimate erosion* erodes binary objects without deleting them. An *ultimate dilation* dilates binary objects but does not connect them.

The combination of an erosion followed by a dilation is called an *opening*. The combination of a dilation followed by an erosion is called a *closing*. Binary closing can be used, for example, to fill small holes and enhance edges as shown in Fig. 1.6.2. Many morphological filters are based on these two operations.

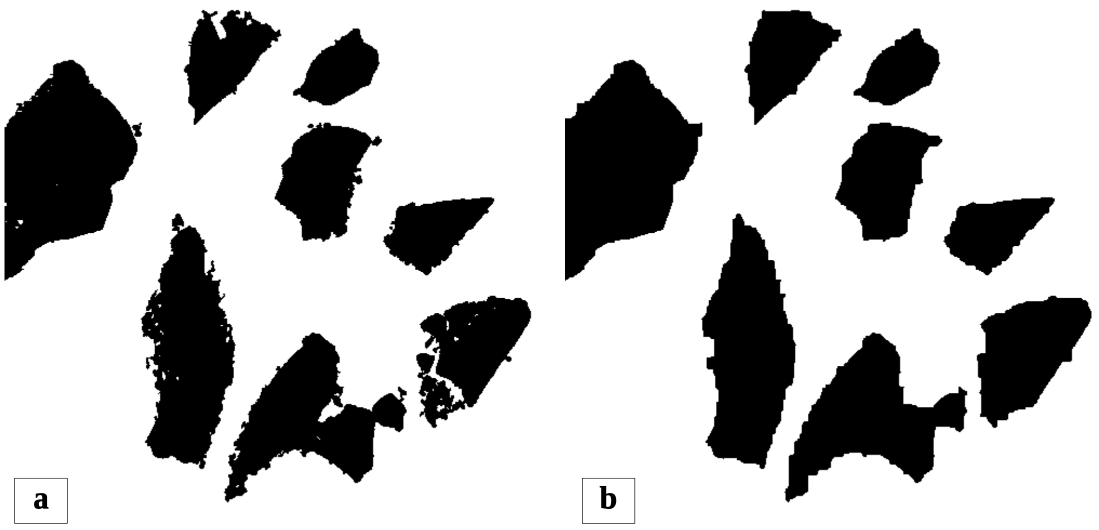


Fig. 1.6.2. a) a binary image of objects having some holes inside and noisy boundaries; **b)** the same image after binary opening when erosion and dilation were applied 4 times.

A combination of simple morphological operations can be used to perform very complicated tasks. For example, a combination of an erosion followed by an ultimate dilation can be used to perform a touching objects segmentation. An example is shown in Fig. 1.6.3.

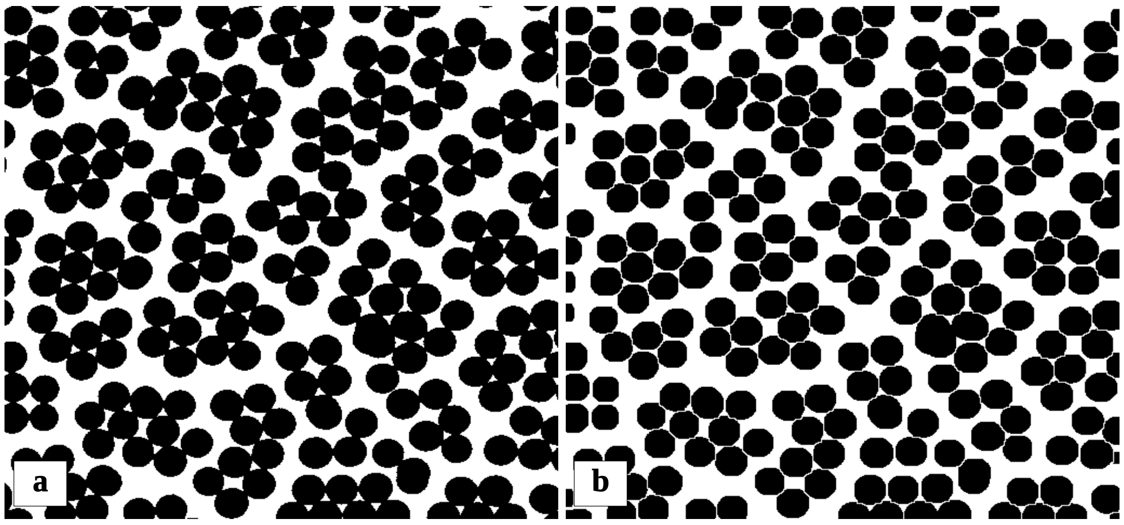


Fig. 1.6.3. a) a binary image of touching objects; **b)** the same image after sequential application of binary erosion (4 times) and ultimate binary dilation (also 4 times).

The difference between dilated and eroded image is called a *morphological gradient*. The difference between original binary image and its binary opening is called the *top-hat transform*. The difference between the binary closing of a binary image and the original binary image is called the *black top-hat transform*. One possible application of such transforms is contour detection in binary images.

A C code for binary erosion and dilation is given below.

```
struct rasterfile* BinaryErosion(image, background, foreground, connectivity)
        struct rasterfile*      image;
        int                     background, foreground, connectivity;
{
        /* few local variables */
        int                     ras_x, ras_y, kx, ky, test;
        struct rasterfile*      temp_image = NULL;
        unsigned char           pixel;

        /* are the image and the connectivity type valid? */
        if (image == NULL) return temp_image;
        if ((connectivity != 4) && (connectivity != 8)) return temp_image;

        /* making new image */
        if ((temp_image = CopyRasterFile(image)) == NULL) return temp_image;

        for (ras_x = 0; ras_x < image->header.ras_width; ras_x++)
        for (ras_y = 0; ras_y < image->header.ras_height; ras_y++) {
                GetPsetFast(image, ras_x, ras_y, &pixel);
                if (pixel != foreground) continue;

                /* check 4 or 8 neighbor pixels */
                test = 0; kx = -1;
                while ((kx < 2) && !test) {
                        if ((ras_x + kx >= 0) && (ras_x + kx < image->header.ras_width)) {
                                ky = -1;
                                while ((ky < 2) && !test) {
                                        if ((connectivity == 4) && (kx * ky != 0)) goto next;
                                        if ((ras_y+ky >= 0) && (ras_y+ky < image->header.ras_height)) {
                                                GetPsetFast(image, (ras_x+kx), (ras_y+ky), &pixel);
                                                if (pixel == background) test = 1;
                                        }
                                        next: ++ky;
                                }
                        }
                        ++kx;
                }

                /* should the pixel be removed? */
                if (test) SetPsetFast(temp_image, ras_x, ras_y, background);
        }

        return temp_image;
}

struct rasterfile* BinaryDilation(image, background, foreground, connectivity)
        struct rasterfile*      image;
        int                     background, foreground, connectivity;
{
        return BinaryErosion(image, foreground, background, connectivity);
}
```

C code 1.6.1. The functions *BinaryErosion* and *BinaryDilation* require a binary image, colors of the background and the foreground and the type of the connectivity (4 or 8). They return eroded and dilated binary images, respectively.

Shrink and swell filters

Shrink and swell filters [1.6.3] are often used to remove noise and fill small holes. A *shrink filter* is used to remove noise from the background. The entire image is scanned and each foreground pixel is examined. If at least k neighbors are background pixels, this pixel is changed to a background pixel. A *swell filter* is used to fill the holes in the foreground. It works in a similar way as the shrink filter. The entire image is scanned and each background pixel is examined. If at least k neighbors are foreground pixels, this pixel is changed to a foreground pixel. In both cases the value of k is the filter index and values between 5 and 8 are typically used.

1.6.2. Contour following techniques

A contour following technique is used to obtain a sequential list of the contour coordinates or its chain encode. An example of a binary image of a particle, its contour coordinates and chain encode are shown in Fig. 1.6.4.

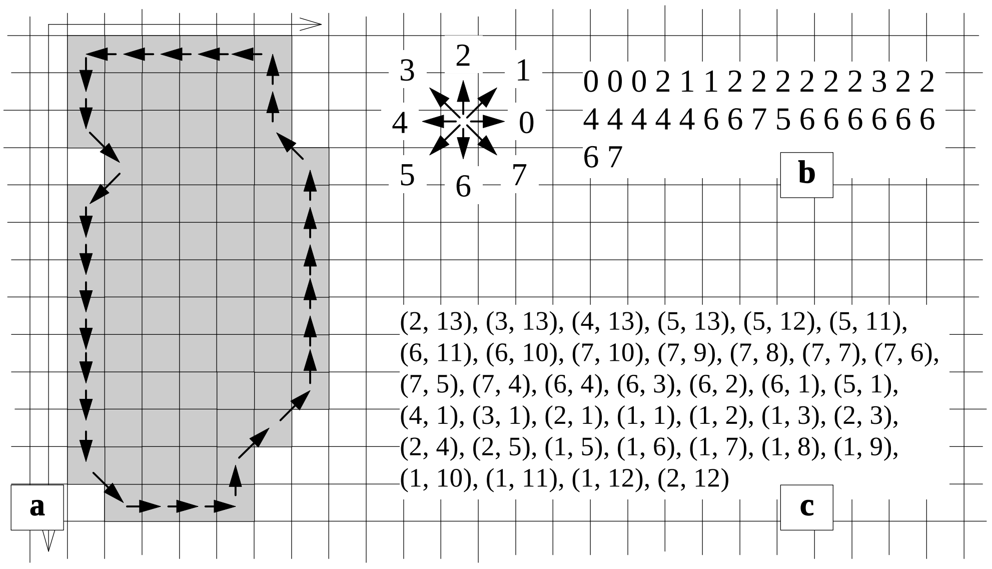


Fig. 1.6.4. Chain code (**b**) and a connected list of the contour coordinates (**c**).

Chain representation of a contour is desirable in those applications which require the contours of particles to be stored. In this case, chain encoding allows to reduce the space needed to record the information. Otherwise a list of the contour coordinates is more suited for an on-line analysis such as fractal or Fourier analysis. Having a sequential list of the contour coordinates $\{(x_i,\ y_i)\}$ the perimeter of the particle can be calculated as

$$P = \sum_i \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}\ .$$

Different contour following techniques are reported in the literature [1.6.1, 1.6.4, 1.6.5], three of them are described below. It was found that one of the most frequently used techniques, *'turtle' procedure*, has an essential drawback and, therefore, its usage in practice should be avoided. So-called *crack following technique* was found to be most effective and simple for computer implementation.

'Turtle' procedure

The algorithm operates on 4-connected neighbors. The turtle starts from a boundary point: if the current pixel belongs to the object, it turns right and advances one pixel; if the current pixel belongs to the background, it turns left and advances one pixel (Fig 1.6.5). The algorithm terminates when the turtle returns to the starting boundary point.

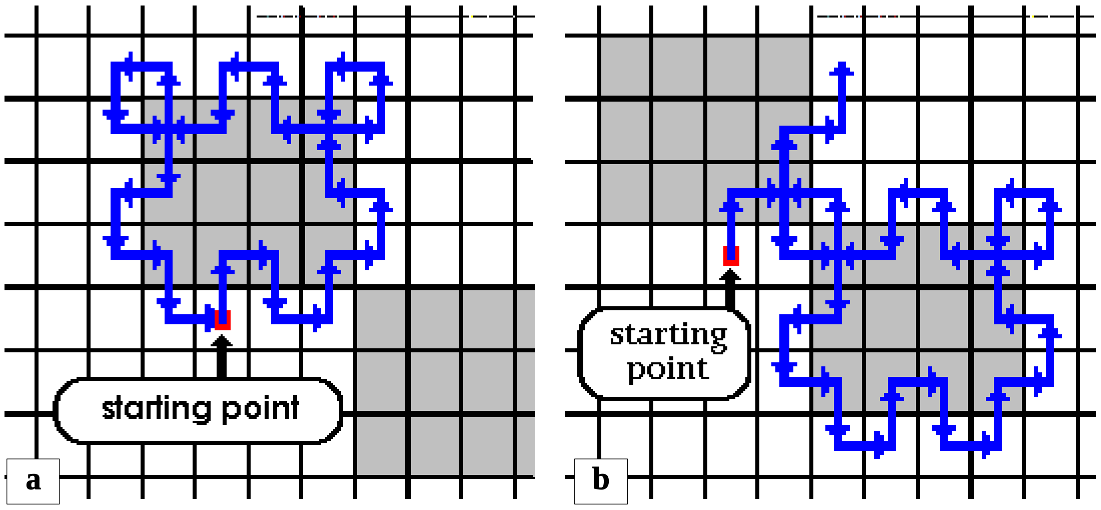


Fig. 1.6.5. Turtle procedure. Sometimes different contours can be tracked starting from different border points.

The turtle procedure has two disadvantages:

1. Certain loops occur during boundary following as illustrated in Fig. 1.6.5. An additional processing of the list of the contour coordinates obtained by this method should be done.
2. The method is starting point dependent as shown in Figs 1.6.5a and 1.6.5b. It is hardly possible to avoid this dependency.

Crack following

The algorithm operates on 4-or 8-connected neighbors. Suppose, we are staying on the border between the object and the background where point L belongs to the object and point R belongs to the background as shown in Fig. 1.6.6a. This pair of the points defines one of the *cracks* on the border. We are facing two other points L_1 and R_1 which are 4-adjacent to L and R, respectively. Then the following rules define the next crack on the border (see Fig. 1.6.6b):

1. $L_{new} = L_1$, $R_{new} = R_1$ if L_1 belongs to the object and R_1 does not belong to the object;
2. $L_{new} = L$, $R_{new} = L_1$ if L_1 and R_1 do not belong to the object;
3. $L_{new} = R_1$, $R_{new} = R$ if L_1 and R_1 belong to the object.

The fourth rule is different in the cases of 4- and 8-connectivity:

4a. $L_{new} = L$, $R_{new} = L_1$ if L_1 does not belong to the object and R_1 belongs to the object;

4b. $L_{new} = R_1$, $R_{new} = R$ if L_1 does not belong to the object and R_1 belongs to the object.

The algorithm terminates when we come back to the initial pair of points.

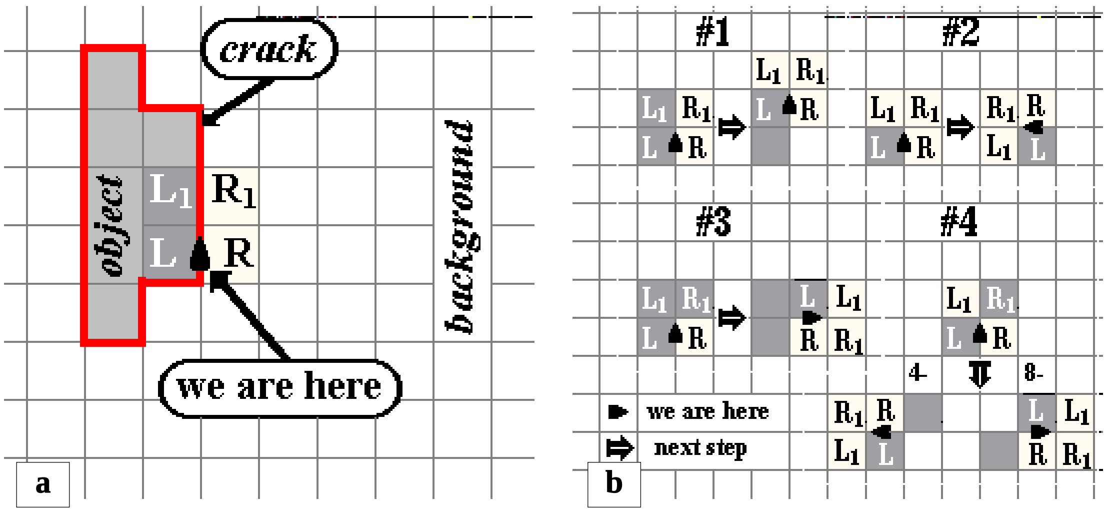


Fig. 1.6.6. a) definition of the crack and some important points; **b)** illustrations of the rules for the crack following algorithm.

A C implementation of the crack following technique is given below.

```
struct i_point {int x; int y};

long ExtractContour(image, the_x, the_y, prtcolor, bgcolor, cntcolor, contour, internalParticle)
        struct rasterfile*    image;              /* raster image */
        int                   the_x, the_y;       /* a point on the contour */
        unsigned char         prtcolor;           /* particle color */
        unsigned char         bgcolor;            /* background color */
        unsigned char         cntcolor;           /* contour color */
        struct i_point*       contour;            /* contour's coordinates (to be found) */
        BOOL*                 internalParticle;   /* does particle touch a border of the image? */
{
        /* internal variables */
        struct i_point        starting, this;     /* */
        unsigned char         pointcolor, l_p_color, r_p_color;
        char                  direction = 0;
```

```
    long                pindex = 0L, perimeter1 = 0L, perimeter2 = 0L;
    BOOL                status = TRUE;

    /* rotation table */
    struct i_point l_shift[4], r_shift[4];
    /* 0: */ l_shift[0].x = +1;   l_shift[0].y = 0;    r_shift[0].x = +1;   r_shift[0].y = -1;
    /* 1: */ l_shift[1].x = 0;    l_shift[1].y = +1;   r_shift[1].x = +1;   r_shift[1].y = +1;
    /* 2: */ l_shift[2].x = -1;   l_shift[2].y = 0;    r_shift[2].x = -1;   r_shift[2].y = +1;
    /* 3: */ l_shift[3].x = 0;    l_shift[3].y = -1;   r_shift[3].x = -1;   r_shift[3].y = -1;

    /* initialization */
    starting.x = this.x = the_x;
    starting.y = this.y = the_y;

    /* is image valid? */
    if (image == NULL) return -1L;

    /* crack following */
    do {
        if (!GetPset(image, this.x+l_shift[direction].x, this.y+l_shift[direction].y, &l_p_color)) {
            l_p_color = bgcolor;
            status = FALSE;
        }
        if (!GetPset(image, this.x+r_shift[direction].x, this.y+r_shift[direction].y, &r_p_color)) {
            r_p_color = bgcolor;
            status = FALSE;
        }
        if ((r_p_color == prtcolor) || (r_p_color == cntcolor)) {
            perimeter1 += 1;
            this.x += r_shift[direction].x;
            this.y += r_shift[direction].y;
            direction = (direction == 0) ? 3 : (direction - 1);
            SetPsetFast(image, this.x, this.y, cntcolor);
            contour[pindex].x = this.x;
            contour[pindex].y = this.y;
            pindex++;
        }
        else if ((l_p_color == prtcolor) || (l_p_color == cntcolor)) {
            perimeter2 += 1;
            this.x += l_shift[direction].x;
            this.y += l_shift[direction].y;
            SetPsetFast(image, this.x, this.y, cntcolor);
            contour[pindex].x = this.x;
            contour[pindex].y = this.y;
            pindex++;
        }
        else
            direction = (direction == 3) ? 0 : (direction + 1);
    } while ((this.x != starting.x) || (this.y != starting.y) || (direction != 0));
    *internalParticle = status;
    return (long)(1.4142*perimeter1 + perimeter2);
}
```

C code 1.6.2. The function *ExtractContour* requires an image, the coordinates of a point on the border, the particle, background and border colors, allocated memory for the traced contour (to be found) and the pointer to the Boolean variable which after processing contains TRUE if the particle does not coincide with a border of the image. The function returns the perimeter.

Border following

This algorithm operates on 4-or 8-connected neighbors. Suppose, we found a boundary point P and defined the eight neighbors R_1, R_2, …, R_8 of P in clockwise direction starting from the right 4-adjacent to P point as shown in the Fig. 1.6.7a. Then the following rules define a new point of the border (Fig. 1.6.7b).

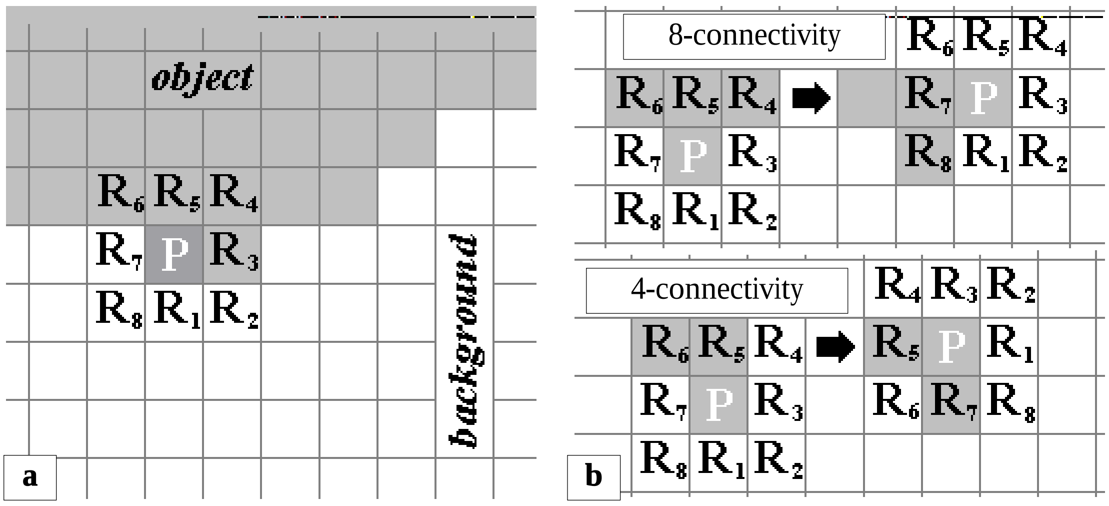


Fig. 1.6.7. a) definition of the boundary point P and its eight neighbors R_1, R_2, …, R_8; **b)** an illustration of the rules used for the border following algorithm.

If 8-connectivity is used: let R_i be the first of R's that belongs to the object, then $P_{new} = R_i$ and $R_{1(new)} = R_{i-1}$ (if $i-1=0$ we define $i-1=8$).

If 4-connectivity is used: let R_i be the first 4-neighbour of P (the first of the R_1, R_3, R_5 or R_7) that belongs to the object, then if R_{i-1} does not belong to the object, take $P_{new} = R_i$ and $R_{1(new)} = R_{i-1}$; if R_{i-1} belongs to the object, take $P_{new} = R_{i-1}$ and $R_{1(new)} = R_{i-2}$ (if $i-2=-1$ we define $i-2=7$).

The algorithm terminates when we come to the initial P again and the current R_1 coincides with the initial R_1.

1.6.3. Perimeter estimation by different yardsticks

The problem of perimeter estimation is considered in fractal analysis of object profiles. Since fractal analysis is used for some of the applications described in Part 3, the algorithm to perform a perimeter approximation using yardsticks of different size is described here.

For a given yardstick size λ the perimeter is determined as follows. Starting at some arbitrary contour point $(x_s,\ y_s)$ the next point on the contour $(x_n,\ y_n)$ in clockwise direction is located which has a distance $d_j = \sqrt{(x_s - x_n)^2 + (y_s - y_n)^2}$ as close as possible to λ. This point is then used to locate the next point on the contour that satisfies this condition. The process is repeated until the initial starting point is reached. The perimeter is the sum of all distances d_j including the distance between the last located point and the starting point. The average yardstick size is the sum of all distances d_j (excluding the distance between the last located point and the starting point) divided by the number of points found.

A C code for this technique is given below.

```
float PerimeterEstimation(contour, n, yardstick)
        struct i_point*     contour;            /* contour's coordinates */
        int                 n;                  /* number of points of the contour */
        float*              yardstick;          /* yardstick to be used to estimate the perimeter */
{
        /* local variables */
        int                 starting_index = 0, current_index, last_index, index, nsteps = 0;
        float               distance = 0.0F, lastdistance, perimeter = 0.0F, step = 0.0F, temp;
        BOOL                AddLastLine;

        /* is the contour valid? */
        if (contour == NULL) return 0.0F;

        /* index */
        current_index = starting_index;
        last_index = current_index;
        index = current_index + 1;

        /* perimeter estimation */
        while (index != starting_index) {
                lastdistance = distance;
                distance = SQR(contour[current_index]->x - contour[index].x);
                distance += SQR(contour[current_index]->y - contour[index].y);
                distance = sqrt(distance);

                if (distance < *yardstick) {
                        lasti_ndex = index;
                        index = (index + 1) % n;
                        AddLastLine = TRUE;
                        continue;
                }
                if (ABS(distance - *yardstick) < ABS(lastdistance - *yardstick)) {
                        temp = distance;
                        current_index = index;
                        last_index = index;
                        index = (index + 1) % n;
                }
                else {
                        temp = lastdistance;
                        current_index = last_index;
                }

                perimeter += temp;
                step += temp;
                nsteps++;
                distance = 0.0F;
                AddLastLine = FALSE;
        }

        /* take into account the last part of the contour */
        if (AddLastLine == TRUE) {
                distance = SQR(contour[current_index]->x - contour[index].x);
                distance += SQR(contour[current_index]->y - contour[index].y);
                distance = sqrt(distance);
                perimeter += distance;
        }

        /* average yardstick used to estimate the perimeter */
        *yardstick = step / nsteps;
        return perimeter;
}
```

C code 1.6.3. The function *PerimeterEstimation* requires a list of contour's coordinates, the number of points on the contour and the yardstick size to be used to estimate the perimeter. It returns the estimated perimeter and the average yardstick size used to perform the estimation.

1.6.4. Contour filling and object labeling

One of the most common problems in computer analysis of images is finding the interior of a region when its contour is given. In particle analysis this problem arises, for example, when one needs to generate the particle's profile when its contour (a list of contour coordinates or its chain code) is given, when there is a need to pick out one of the particles visible on an image, to calculate the particle's area, etc. To solve these problems the following recursive algorithm was developed.

Suppose, the contour of an object is drawn and a point P inside the object's profile which is not yet marked as belonging to the interior of the object, is given (Fig. 1.6.8a). Moving in the horizontal direction to the left the contour point L which is the nearest to P, is found. In the same manner, moving in the horizontal direction to the right, the other contour point R which is the nearest to P, is located. It is evident that all points between L and R belong to the interior of the object and they can be already labeled. Next, all points just below and above of the labeled line belong either to the contour or to the interior of the object. Now one of these points which belongs to the interior of the object but is not yet marked, is considered as a point P inside of the object's profile and the entire procedure is repeated from the beginning. The algorithm allows to fill the interior for an object. With small modifications it can be used to pick out the object or to eliminate it. Also this procedure is used to calculate the area of an object.

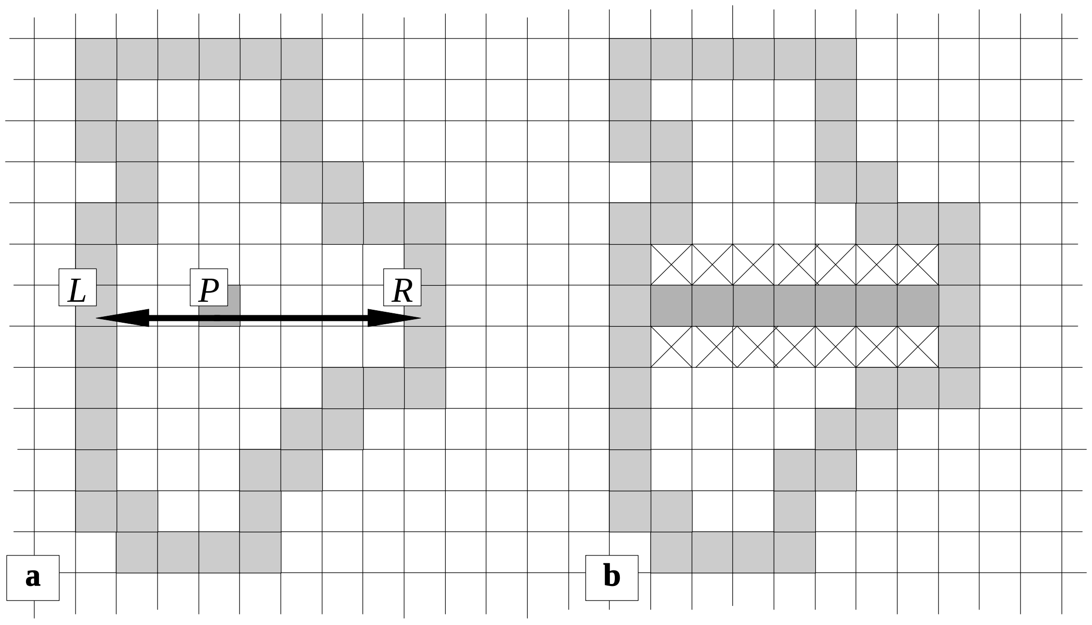


Fig. 1.6.8. a) a point P inside of the object and its two nearest contour points L and R in horizontal direction; **b)** the situation after the first call of the recursive filling procedure. The points, marked as 'x' are used to call the function recursively.

A C code for this algorithm is given below.

```
long FillObject(image, the_x, the_y, oldcolor, newcolor)
        struct rasterfile*      image;                  /* a raster image */
        int                     the_x, the_y;           /* a point inside of the particle */
        unsigned char           oldcolor;               /* present color of the particle's interior */
        unsigned char           newcolor;               /* color to be used to fill the interior */
{
        /* local variables */
```

```
    int                 area = 0;
    int                 lx, rx, x;
    unsigned char       pointcolor;

    /* is the given point a correct one? */
    if (!GetPset(image, the_x, the_y, &pointcolor) || (pointcolor != oldcolor)) return 0L;

    /* initialization */
    lx = rx = the_x;

    /* find the nearest contour point form the left side */
    while (GetPset(image, lx, the_y, &pointcolor) && (pointcolor == oldcolor))
        --lx;
    ++lx;

    /* find the nearest contour point form the right side */
    while (GetPset(image, rx, the_y, &pointcolor) && (pointcolor == oldcolor))
        ++rx;
    --rx;

    /* mark the points between two found */
    for (x = lx; x <= rx; x++)
        SetPsetFast(image, (unsigned int)x, (unsigned int)the_y, newcolor);

    /* area calculation */
    area = rx - lx + 1;

    /* check points below the labeled line */
    if (GetPset(image, the_x, the_y+1, &pointcolor))
        for (x = lx-1; x <= rx+1; x++)
            area += FillParticle(image, x, the_y+1, oldcolor, newcolor);

    /* check points above the labeled line */
    if (GetPset(image, the_x, the_y-1, &pointcolor))
        for (x = lx-1; x <= rx+1; x++)
            area += FillParticle(image, x, the_y-1, oldcolor, newcolor);

    return area;
}
```

C code 1.6.4. The function *FillObject* requires an image, the coordinates of a point inside of the object, the color of the interior of the object and new color to be used to fill the interior. The function returns the object's area.

1.6.5. Watershed segmentation of touching objects

A common difficulty in measuring images occurs when objects are touching and, therefore, cannot be separately identified, counted or measured. One method for separation such objects is based on morphological erosion and dilation as shown above. Another method is based on a variant of the watershed segmentation algorithm, especially designed for the binary images [1.6.6]. The first step in the procedure is to create a distance map in which the gray level is the measure of how far the corresponding binary pixel is from the background. In a second step, the segmentation occurs along watersheds in the distance map.

To construct the distance map a few methods are available. The simplest one uses sequential application of morphological erosion. The features in the image are sequentially eroded and after each iteration all remaining pixels have their corresponding gray level value incremented by one. When the process is completed the desired distance map has been constructed. An example of such a map obtained for an image of touching objects (Fig. 1.6.9a) is shown in Fig. 1.6.9b. The map was obtained using 8-connectivity to perform the erosion. Now, imagine that the gray level values of each pixel in the distance map correspond to a physical elevation. Then the objects are represented as mountain peaks and the total number of peaks corresponds to the total number of objects in the image. As a first step in the watershed segmentation the peaks are labeled as belonging to the resulting binary image and they form the nuclei for subsequent feature growth by thickening. Thickening is performed in steps starting at a height (brightness) value one less than the highest peak. At each height pixels having this value, are candidates for thickening. They become a part of the binary image if they do not create a new junction between any neighboring pixels. The operation is than repeated at each next lover brightness level until no changes take place. Examples of watershed segmented images are shown in Figs 1.6.9c and 1.6.9d. The difference between these images is that different distance maps were used. For the image shown in Fig. 1.6.9c 8-connectivity was used to generate the distance map (shown in Fig. 1.6.9b) whereas for the image shown in Fig. 1.6.9d 4-connectivity was used. In both cases some imperfections are observed. However for many practical applications the method was found to be appropriate.

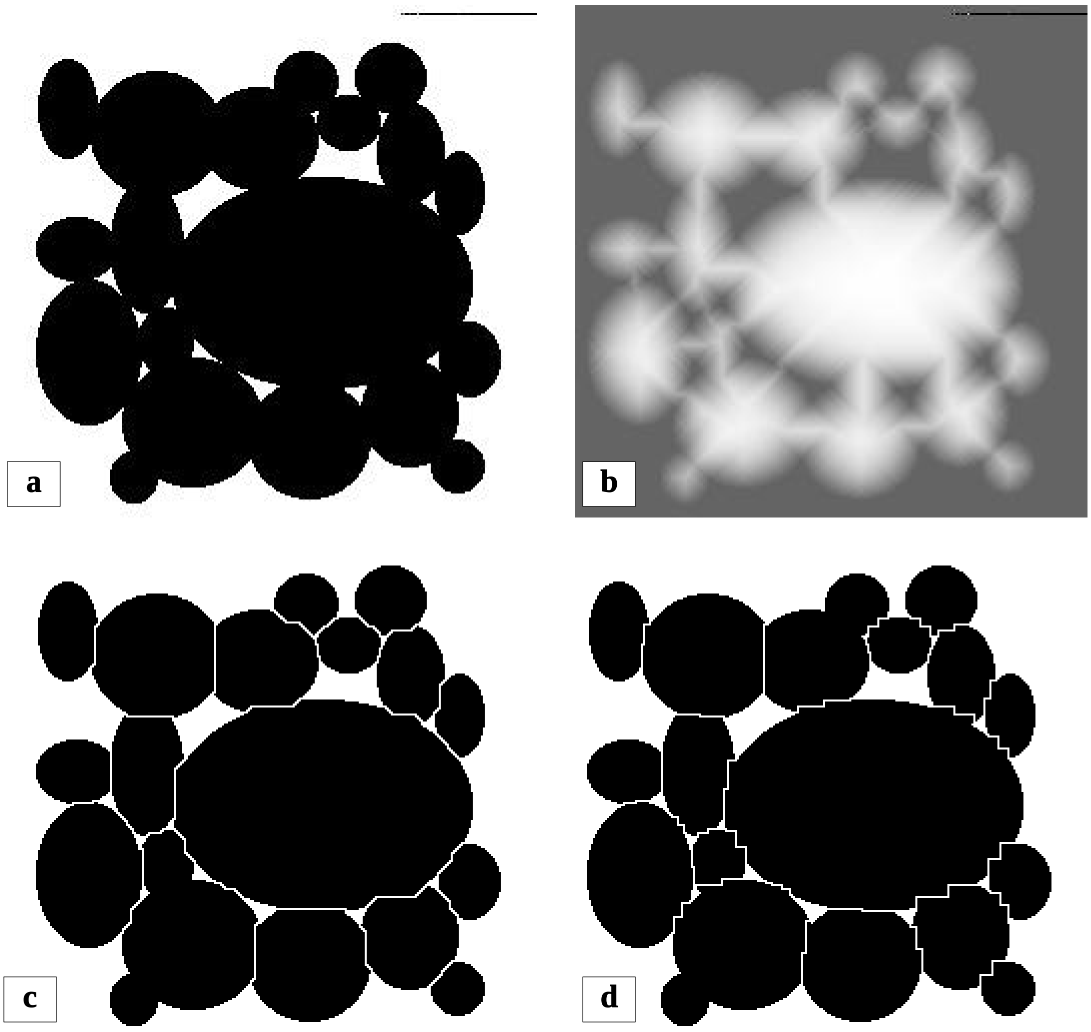


Fig. 1.6.9. a) a binary image of touching objects; **b)** the distance map obtained for the binary image using the 8-connectivity rule; **c)** watershed segmentation performed using this distance map; **d)** watershed segmentation performed using the distance map obtained with the 4-connectivity rule.

References

[1.1.1] K. Kiss, *Problem Solving with Microbeam Analysis*, Elsevier, Budapest, 1988.

[1.1.2] L. Reimer, *Scanning Electron Microscopy: Physics of Image Formation and Microanalysis*, Springer-Verlag, Berlin, 1985.

[1.1.3] J. Goldstein, D. Newbury, P. Echlin, D. Joy, C. Fiori and E. Lifshin, *Scanning Electron Microscopy and X-ray Microanalysis*, Plenum Press, New York, 1984.

[1.1.4] L. Murr, *Electron and Ion Microscopy and Microanalysis*, McGraw-Hill, New York, 1984.

[1.1.5] J. Goldstein and H. Yakowitz, eds, *Practical Scanning Electron Microscopy: Electron and Ion Microprobe Analysis*, Plenum Press, New York, 1975.

[1.1.6] S. Reed, *Electron Microprobe Analysis*, Cambridge University Press, London, 1975.

[1.1.7] O. Wells, A. Boyde, E. Lifshin and A. Rezanowich, *Scanning Electron Microscopy*, McGraw-Hill, Inc., New York, 1974.

[1.1.8] D. Holt, M. Muir, P. Grant and I. Boswarva, *Quantitative Scanning Electron Microscopy*, Academic Press, London, 1974.

[1.1.9] L. Reimer, ed., *Energy-Filtering Transmission Electron Microscopy*, Springer, Berlin, 1995.

[1.1.10] R. Egerton, *Electron Energy-Loss Spectroscopy in the Electron Microscope*, 2nd ed., Plenum Press, New York, 1996.

[1.1.11] J. Russ, *Computer-Controlled Microscopy: The Measurement and Analysis of Images*, Plenum Press, New York, 1990.

[1.2.1] Kontron Elektronik GmbH - Bildanalyse, Oskar vin Miller str. 1, 85365 München.

[1.2.2] The MathWorks Inc., 24 Prime Park Way, Natick, Mass. 01760-1500.

[1.2.3] MathSoft Inc., 101 Main Street, Cambridge, Massachusetts, 02142, USA.

[1.3.1] S. Stanfield; R. Arvesen and A. Light; *Visual C++ How-To: the Definitive MFC Problem Solver*, Wait Group Press, Corte Madera, 1995.

[1.3.2] M. Blaszczak, *The Revolutionary Guide to WIN32 Programming Using Visual C++*, Wrox Press, Birmingham, 1995.

[1.3.3] *Mastering Microsoft Visual C++4: In-Dept, Interactive Training for Experienced Developers*, Microsoft Corporation, Redmond, 1995.

[1.3.4] P. Brill, *Visual C++4*, Easy Computing, Brussels, 1996.

[1.3.5] M. Andrews, *Learn Visual C++ Now: Teach Yourself Microsoft Visual C++ in the Quick and Easy Way*, Microsoft Press, Redmond, 1996.

[1.4.1] A. Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall International, Englewood Cliffs, 1989.

[1.4.2] I. Pitas, *Digital Image Processing Algorithms*, Prentice Hall Inc., New York, 1993.

[1.4.3] J. Russ, *Computer-Controlled Microscopy: The Measurement and Analysis of Images*, Plenum Press, New York, 1990.

[1.4.4] J. Parker, *Practical Computer Vision Using C*, John Wiley & Sons, New York, 1993.

[1.4.5] H. Myler and A. Weeks, *Computer Imaging Recipes in C*, Prentice Hall International, Englewood Cliffs, 1993.

[1.4.6] S. Bow, *Pattern Recognition and Image Preprocessing*, Marcel Dekker, New York, 1990.

[1.4.7] C. Lindley, *Practical Image Processing in C*, John Wiley & Sons, Inc., New York, 1991.
[1.4.8] T. Pavlidis, *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, 1982.

[1.5.1] P. Sahoo, S. Soltani, A. Wong and Y. Chen, *A survey of thresholding techniques*, Computer Vision, Graphics, and Image Processing, vol. 41, pp. 233-260, 1989.
[1.5.2] A. Brink, *Grey-level thresholding of images using a correlation criterion*, Pattern Recognition Letters, vol. 9, pp. 335-341, 1989.
[1.5.3] A. Perez and R. Gonzalez, *An iterative thresholding algorithm for image segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-9, no. 6, pp. 742-751, 1987.
[1.5.4] J. Kittler and J. Illingworth, *On threshold selection using clustering criteria*, IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-15, pp. 652-655, 1985.
[1.5.5] Y. Zhang and J. Gerbrands, *Transition region determination based thresholding*, Pattern Recognition Letters, vol. 12, pp. 13-23, 1991.
[1.5.6] W. Snyder, G. Bilbro, A. Logenthiran and S. Rajala, *Optimal thresholding - a new approach*, Pattern Recognition Letters, vol. 11, pp. 803-810, 1990.
[1.5.7] R. Whatmough, *Automatic threshold selection from a histogram using the "exponential hull"*, CVGIP: Graphical Models and Image Processing, vol. 53, no. 6, pp. 592-600, 1991.
[1.5.8] B. Chanda, B. Chaudhuri and D. Dutta Majumder, *On image enhancement and threshold selection using the graylevel co-occurrence matrix*, Pattern Recognition Letters, vol. 3, pp. 243-251, 1985.
[1.5.9] J. Parker, *Gray level thresholding in badly illuminated images*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 13, no. 8, pp. 813-819, 1991.
[1.5.10] V. Kindratenko, B. Treiger and P. Van Espen, *Binarization of inhomogeneously illuminated images*, Lecture Notes in Computer Science, vol. 974, pp. 483-487, 1995.
[1.5.11] J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, London, 1988.
[1.5.12] E. Dougherty, ed., *Mathematical Morphology in Image Processing*, Marcel Dekker, Inc., New York, 1992.
[1.5.13] F. Meyer, *Topographic distance and watershed lines*, Signal Processing, vol. 38, pp. 113-125, 1994.
[1.5.14] L. Vincent and P. Soille, *Watersheds in digital space: an effecient algorithm based on immersion simulations*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 13, no. 6, pp. 583-597, 1991.

[1.6.1] A. Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall International, Englewood Cliffs, 1989.
[1.6.2] J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, London, 1988.
[1.6.3] Y. Xie, *Particle Classification with a Combination of Chemical Composition and Shape Index Utilizing Artificial Neural Network*, Ph.D. Dissertation, Clarkson University, 1994.
[1.6.4] I. Pitas, *Digital Image Processing Algorithms*, Prentice Hall Inc., New York, 1993.
[1.6.5] A. Rosenfeld and A. Kak, *Digital Picture Processing*, 2nd ed., Academic Press Inc., London, 1982.
[1.6.6] J. Russ, *Computer-Controlled Microscopy: The Measurement and Analysis of Images*, Plenum Press, New York, 1990.