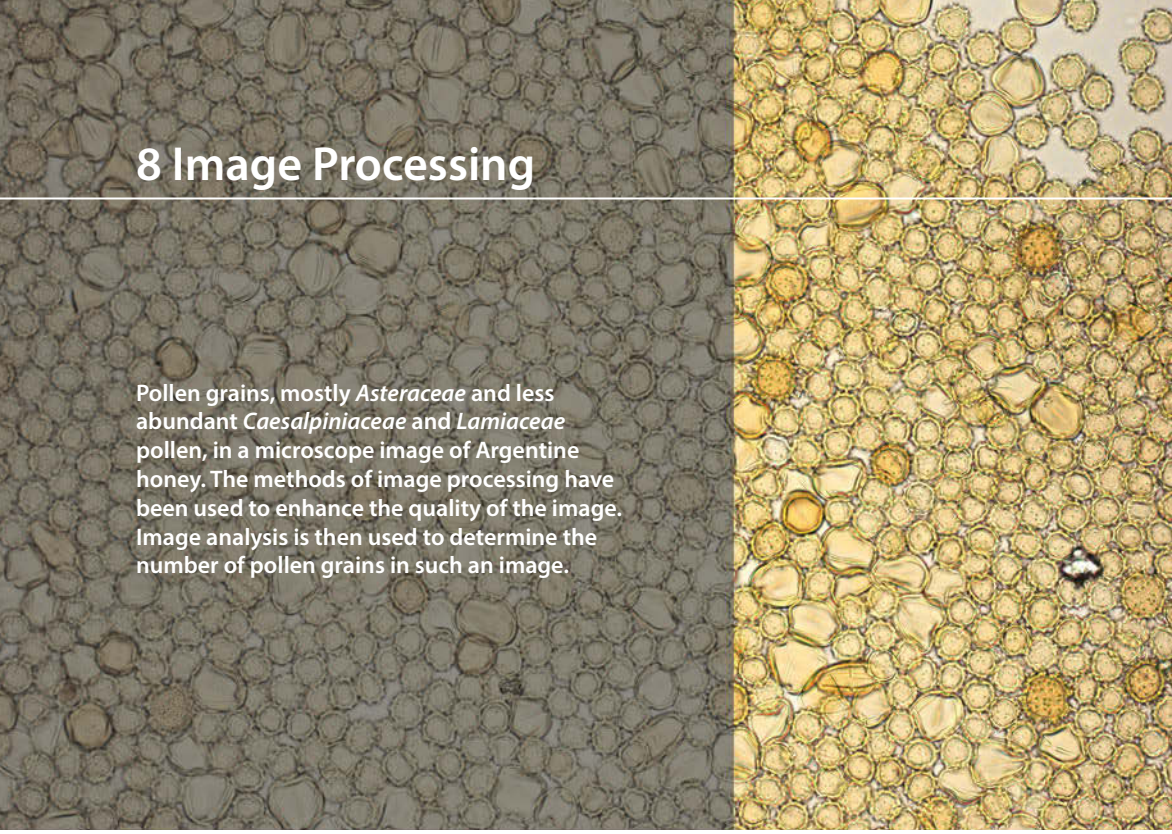
8 Image Processing



Pollen grains, mostly *Asteraceae* and less abundant *Caesalpiniaceae* and *Lamiaceae* pollen, in a microscope image of Argentine honey. The methods of image processing have been used to enhance the quality of the image. Image analysis is then used to determine the number of pollen grains in such an image.

8.1 Introduction

Computer graphics are stored and processed as either vector or raster data. Most of the data types that were encountered in the previous chapter were vector data, i.e., points, lines and polygons. Drainage networks, the outlines of geologic units, sampling locations, and topographic contours are all examples of vector data. In Chapter 7, coastlines are stored in a vector format while bathymetric and topographic data are saved in a raster format. Vector and raster data are often combined in a single data set, for instance to display the course of a river on a satellite image. Raster data are often converted to vector data by digitizing points, lines or polygons. Conversely, vector data are sometimes transformed to raster data.

Images are generally represented as raster data, i.e., as a 2D array of color intensities. Images are everywhere in geosciences. Field geologists use aerial photos and satellite images to identify lithologic units, tectonic structures, landslides and other features within a study area. Geomorphologists use such images to analyze drainage networks, river catchments, and vegetation or soil types. The analysis of images from thin sections, the automated identification of objects, and the measurement of varve thicknesses all make

M.H. Trauth, *MATLAB® Recipes for Earth Sciences*,
DOI 10.1007/978-3-662-46244-7_8

use of a great variety of image processing methods.

This chapter is concerned with the analysis and display of image data. The various ways that raster data can be stored on the computer are first explained (Section 8.2). The main tools for importing, manipulating and exporting image data are then presented in Section 8.3. This knowledge is then used to process and to georeference satellite images (Sections 8.4 to 8.6). On-screen digitization techniques are discussed in Section 8.7. Sections 8.8 and 8.9 deal with importing, enhancing, and analyzing images from laminated lake sediments, including color-intensity measurements on transects across the laminae. Finally, Sections 8.10 to 8.12 deal with automated grain size analysis, charcoal quantification in microscope images, and the detection of objects in microscope images on the basis of their shapes. The Image Processing Toolbox is used for the specific examples throughout this chapter (MathWorks 2014). While the MATLAB User's Guide to the Image Processing Toolbox provides an excellent general introduction to the analysis of images, this chapter provides an overview of typical applications in earth sciences.

8.2 Data Storage

Vector and raster graphics are the two fundamental methods for storing pictures. The typical format for storing *vector data* has already been introduced in the previous chapter. In the following example the two columns in the file *coastline.txt* represent the longitudes and latitudes of the points of a polygon.

```
NaN          NaN
42.892067    0.000000
42.893692    0.001760
NaN          NaN
42.891052    0.001467
42.898093    0.007921
42.904546    0.013201
42.907480    0.016721
42.910414    0.020828
42.913054    0.024642
(cont'd)
```

The `NaN`s help to identify break points in the data (Section 7.2).

The *raster data* are stored as 2D arrays. The elements of these arrays represent variables such as the altitude of a grid point above sea level, the annual rainfall or, in the case of an image, the color intensity values.

```
174 177 180 182 182 182
165 169 170 168 168 170
171 174 173 168 167 170
184 186 183 177 174 176
191 192 190 185 181 181
```

```
189 190 190 188 186 183
```

Raster data can be visualized as 3D plots. The *x* and *y* figures are the indices of the 2D array or any other reference frame, and *z* is the numerical value of the elements of the array (see also Chapter 7). The numerical values contained in the 2D array can be displayed as a pseudocolor plot, which is a rectangular array of cells with colors determined by a colormap. A colormap is an *m*-by-3 array of real numbers between 0.0 and 1.0. Each row defines a red, green, or blue (RGB) color. An example is the above array, which could be interpreted as grayscale intensities ranging from 0 (black) to 255 (white). More complex examples include satellite images that are stored in 3D arrays.

As previously discussed, a computer stores data as bits that have one of two states, represented by either a one or a zero (Chapter 2). If the elements of the 2D array represent the color intensity values of the *pixels* (short for *picture elements*) of an image, 1-bit arrays contain only ones and zeros.

```
0   0   1   1   1   1
1   1   0   0   1   1
1   1   1   1   0   0
1   1   1   1   0   1
0   0   0   0   0   0
0   0   0   0   0   0
```

This 2D array of ones and zeros can be simply interpreted as a black-and-white image, where the value of one represents white and zero corresponds to black. Alternatively, the 1-bit array could be used to store an image consisting of any two different colors, such as red and blue.

In order to store more complex types of data, the bits are joined together to form larger groups, such as bytes consisting of eight bits. Since the earliest computers could only process eight bits at a time, early computer code was written in sets of eight bits, which came to be called bytes. Each element of the 2D array or pixel therefore contains a vector of eight ones or zeros.

```
1    0    1    0    0    0    0    1
```

These 8 bits (or 1 byte) allow 2^8=256 possible combinations of the eight ones or zeros, and are therefore able to represent 256 different intensities, such as grayscales. The 8 bits can be read in the following way, reading from right to left: a single bit represents two numbers, two bits represent four numbers, three bits represent eight numbers, and so forth up to a byte (or eight bits), which represents 256 numbers. Each added bit doubles the count of numbers. Here is a comparison of binary and decimal representations of the number 161:

```
128   64   32   16    8    4    2    1         (value of the bit)
  1    0    1    0    0    0    0    1         (binary)

128 +  0 + 32  + 0 +  0 +  0 +  0 +  1 = 161   (decimal)
```

The end members of the binary representation of grayscales are

```
  0    0    0    0    0    0    0    0
```

which is black, and

```
  1    1    1    1    1    1    1    1
```

which is pure white. In contrast to the above 1-bit array, the 1-byte array allows a grayscale image of 256 different levels to be stored. Alternatively, the 256 numbers could be interpreted as 256 discrete colors. In either case, the display of such an image requires an additional source of information concerning how the 256 intensity values are converted into colors. Numerous global colormaps for the interpretation of 8-bit color images exist that allow the cross-platform exchange of raster images, while local colormaps are often embedded in a graphics file.

The disadvantage of 8-bit color images is that the 256 discrete colorsteps are not enough to simulate smooth transitions for the human eye. A 24-bit system is therefore used in many applications, with 8 bits of data for each RGB channel giving a total of 256^3=16,777,216 colors. Such a 24-bit image is stored in three 2D arrays, or one 3D array, of intensity values between 0 and 255.

```
195  189  203  217  217  221
218  209  187  192  204  206
207  219  212  198  188  190
203  205  202  202  191  201
190  192  193  191  184  190
186  179  178  182  180  169

209  203  217  232  232  236
234  225  203  208  220  220
224  235  229  214  204  205
223  222  222  219  208  216
209  212  213  211  203  206
206  199  199  203  201  187

174  168  182  199  199  203
198  189  167  172  184  185
188  199  193  178  168  172
186  186  185  183  174  185
177  177  178  176  171  177
179  171  168  170  170  163
```

Compared to the 1-bit and 8-bit representations of raster data, 24-bit storage certainly requires a lot more computer memory. In the case of very large data sets such as satellite images and digital elevation models the user should therefore think carefully about the most suitable way to store the data. The default data type in MATLAB is the 64-bit array, which allows storage of the sign of a number (bit 63), the exponent (bits 62 to 52) and roughly 16 significant decimal digits between approximately 10^{-308} and 10^{+308} (bits 51 to 0). However, MATLAB also works with other data types such as 1-bit, 8-bit and 24-bit raster data, to save memory.

The amount of memory required for storing a raster image depends on the data type and the image's dimensions. The dimensions of an image can be described by the number of pixels, which is the number of rows multiplied by the number of columns of the 2D array. Let us assume an image of 729-by-713 pixels, such as the one we will use in the following section. If each pixel needs 8 bits to store a grayscale value, the memory required by the data is 729·713·8=4,158,216 bits or 4,158,216/8=519,777 bytes. This number is exactly what we obtain by typing `whos` in the command window. Common prefixes for bytes are kilo-, mega-, giga- and so forth.

```
bit = 1 or 0 (b)
8 bits = 1 byte (B)
1024 bytes = 1 kilobyte (KB)
1024 kilobytes = 1 megabyte (MB)
1024 megabytes = 1 gigabyte (GB)
1024 gigabytes = 1 terabyte (TB)
```

Note that in data communication 1 kilobit=1,000 bits, while in data storage 1 kilobyte=1,024 bytes. A 24-bit or *true color image* then requires three times the memory required to store an 8-bit image, or 1,559,331 bytes=1,559,331/1,024 kilobytes (KB)≈1,523 KB≈1,559,331/$1{,}024^2$=1.487 megabytes (MB).

However, the dimensions of an image are often given, not by the total number of pixels, but by the length and height of the image and its resolution. The resolution of an image is the number of *pixels per inch* (ppi) or *dots per inch* (dpi). The standard resolution of a computer monitor is 72 dpi although modern monitors often have a higher resolution such as 96 dpi. For instance, a 17 inch monitor with 72 dpi resolution displays 1,024-by-768 pixels. If the monitor is used to display images at a different (lower, higher) resolution, the image is resampled to match the monitor's resolution. For scanning and printing, a resolution of 300 or 600 dpi is enough in most applications. However, scanned images are often scaled for large printouts and therefore have higher resolutions such as 2,400 dpi. The image used in the next section has a width of 25.2 cm (or 9.92 inches) and a height of 25.7 cm (10.12 inches). The resolution of the image is 72 dpi. The total number of

pixels is therefore 72·9.92≈713 in a horizontal direction, and 72·10.12≈729 in a vertical direction.

Numerous formats are available for saving vector and raster data into a file, each with their own particular advantages and disadvantages. Choosing one format over another in an application depends on the way the images are to be used in a project and whether or not the images are to be analyzed quantitatively. The most popular formats for storing vector and raster data are:

- *Compuserve Graphics Interchange Format* (GIF) – This format was developed in 1987 for raster images using a fixed 8-bit colormap of 256 colors. The GIF format uses compression without loss of data. It was designed for fast transfer rates over the Internet. The limited number of colors means that it is not the right format for the smooth color transitions that occur in aerial photos or satellite images. It is, however, often used for line art, maps, cartoons and logos (http://www.compuserve.com).

- *Portable Network Graphics* (PNG) – This is an image format developed in 1994 that is used as an alternative to the GIF. It is similar to the GIF in that it also uses a fixed 8-bit colormap of 256 colors. Alternatively, grayscale images of 1 to 16 bits can be stored, as well as 24 and 48 bit color images. The PNG format uses compression without loss of data, with the method employed being better than that used for GIF images.

- *Microsoft Windows Bitmap Format* (BMP) – This is the default image format for computers running Microsoft Windows as the operating system. However, numerous converters also exist to read and write BMP files on other platforms. Various modifications of the BMP format are available, some of them without compression and others with effective and fast compression (http://www.microsoft.com).

- *Tagged Image File Format* (TIFF) – This format was designed by the Aldus Corporation and Microsoft in 1986 to become an industry standard for image-file exchange. A TIFF file includes an image file header, a directory, and the data in all available graphics and image file formats. Some TIFF files even contain vector and raster versions of the same picture, as well as images at different resolutions and with different colormaps. The main advantage of TIFF files was originally their portability. A TIFF should perform on all computer platforms; unfortunately, however, numerous modifications of the TIFF have evolved in subsequent years, resulting in incompatibilities. The TIFF is therefore now often called the *Thousands of Incompatible File Formats*.

- *PostScript* (PS) and *Encapsulated PostScript* (EPS) – The PS format was developed by John Warnock at PARC, the Xerox research institute. Warnock was also co-founder of Adobe Systems, where the EPS format was created. The PostScript vector format would never have become an industry standard without Apple Computers. In 1985 Apple needed a typesetter-quality controller for the new Apple LaserWriter printer and the Macintosh operating system and adopted the PostScript format. The third partner in the history of PostScript was the company Aldus, the developer of the software PageMaker and now a part of Adobe Systems. The combination of Aldus PageMaker software, the PS format and the Apple LaserWriter printer led to the creation of Desktop Publishing. The EPS format was then developed by Adobe Systems as a standard file format for importing and exporting PS files. Whereas a PS file is generally a single-page format containing either an illustration or a text, the purpose of an EPS file is to also allow the inclusion of other pages, i.e., a file that can contain any combination of text, graphics and images (http://www.adobe.com).

- In 1986 the *Joint Photographic Experts Group* (JPEG) was founded for the purpose of developing various standards for image compression. Although JPEG stands for the committee, it is now widely used as the name for an image compression and a file format. This compression involves grouping pixel values into 8-by-8 blocks and transforming each block with a discrete cosine transform. As a result, all unnecessary high-frequency information is deleted, which makes this compression method irreversible. The advantage of the JPEG format is the availability of a three-channel, 24-bit, true color version. This allows images with smooth color transitions to be stored. The new JPEG-2000 format uses a wavelet transform instead of the cosine transform (Section 5.8) (http://www.jpeg.org).

- *Portable Document Format* (PDF) – The PDF designed by Adobe Systems is now a true self-contained cross-platform document. PDF files contain the complete formatting of vector illustrations, raster images and text, or a combination of all these, including all necessary fonts. These files are highly compressed, allowing a fast internet download. Adobe Systems provides the free-of-charge Acrobat Reader for all computer platforms to read PDF files (http://www.adobe.com).

8.3 Importing, Processing and Exporting Images

We first need to learn how to read an image from a graphics file into the workspace. As an example we use a satellite image showing a 10.5 km by 11 km subarea in northern Chile:

```
http://asterweb.jpl.nasa.gov/gallery/images/unconform.jpg
```

The file *unconform.jpg* is a processed TERRA-ASTER satellite image that can be downloaded free-of-charge from the NASA web page. We save this image in the working directory. The command

```
clear

I1 = imread('unconform.jpg');
```

reads and decompresses the JPEG file, imports the data as a 24-bit RGB image array and stores it in a variable `I1`. The command

```
whos
```

shows how the RGB array is stored in the workspace:

```
Name        Size                Bytes  Class    Attributes
I1        729x713x3           1559331  uint8
```

The details indicate that the image is stored as a 729-by-713-by-3 array, representing a 729-by-713 array for each of the colors red, green and blue. The listing of the current variables in the workspace also gives the information *uint8* array, i.e., each array element representing one pixel contains 8-bit integers. These integers represent intensity values between 0 (minimum intensity) and 255 (maximum). As an example, here is a sector in the upper-left corner of the data array for red:

```
I1(50:55,50:55,1)

ans =
   174 177 180 182 182 182
   165 169 170 168 168 170
   171 174 173 168 167 170
   184 186 183 177 174 176
   191 192 190 185 181 181
   189 190 190 188 186 183
```

We can now view the image using the command

```
imshow(I1)
```

which opens a new Figure Window showing an RGB composite of the image (Fig. 8.1). In contrast to the RGB image, a grayscale image needs only a single array to store all the necessary information. We therefore convert the RGB image into a grayscale image using the command `rgb2gray` (RGB to gray):

```
I2 = rgb2gray(I1);
```

The new workspace listing now reads

```
Name        Size                Bytes  Class    Attributes
I1        729x713x3           1559331  uint8
```

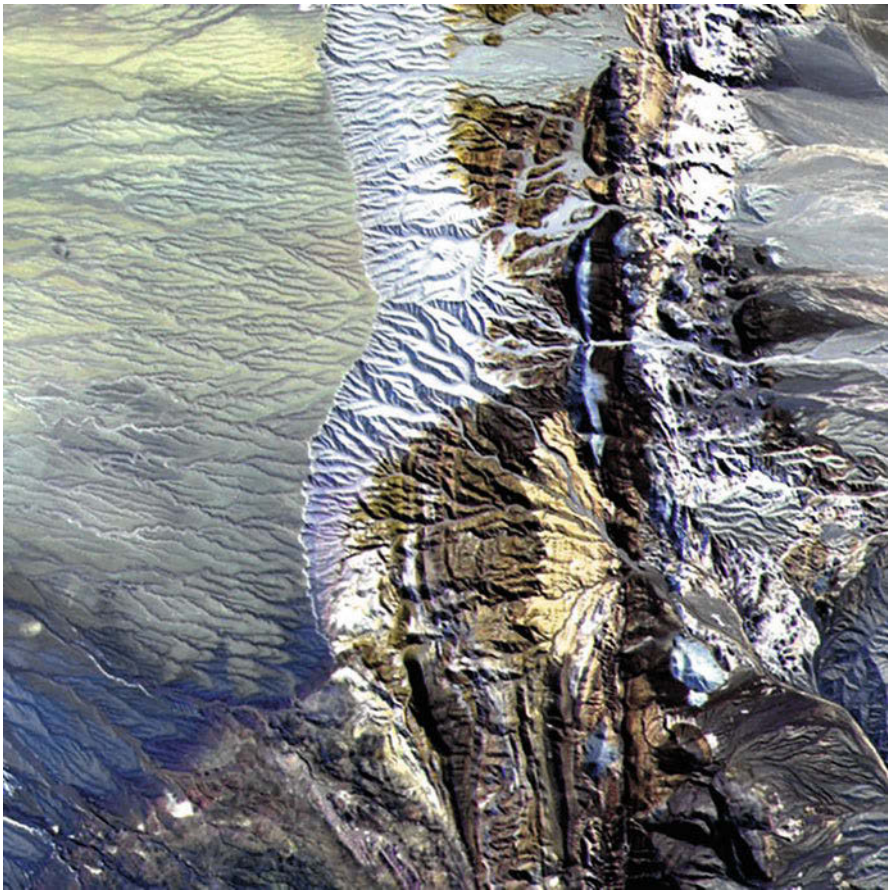


Fig. 8.1 RGB true color image contained in the file *unconform.jpg*. After decompressing and reading the JPEG file into a 729-by-713-by-3 array, MATLAB interprets and displays the RGB composite using the function `imshow`. See detailed description of the image on the NASA TERRA-ASTER webpage: http://asterweb.jpl.nasa.gov. Original image courtesy of NASA/GSFC/METI/ERSDAC/JAROS and U.S./Japan ASTER Science Team.

```
I2         729x713                519777  uint8
ans          6x6                      36  uint8
```

in which the difference between the 24-bit RGB and the 8-bit grayscale arrays can be observed. The variable `ans` for *Most recent answer* was created above using `I1(50:55,50:55,1)`, without assigning the output to another variable. The commands

```
imshow(I2)
```

display the result. It is easy to see the difference between the two images in separate Figure Windows. Let us now process the grayscale image. First, we compute a histogram of the distribution of intensity values.

```
imhist(I2)
```

A simple technique to enhance the contrast in such an image is to transform this histogram to obtain an equal distribution of grayscales.

```
I3 = histeq(I2);
```

We can view the difference again using

```
imshow(I3)
```

and save the results in a new file.

```
imwrite(I3,'unconform_gray.jpg')
```

We can read the header of the new file by typing

```
imfinfo('unconform_gray.jpg')
```

which yields

```
ans =
           Filename: [1x40 char]
        FileModDate: '18-Dec-2013 11:26:53'
           FileSize: 138419
             Format: 'jpg'
      FormatVersion: ''
              Width: 713
             Height: 729
           BitDepth: 8
          ColorType: 'grayscale'
    FormatSignature: ''
    NumberOfSamples: 1
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {}
```

Hence, the command `imfinfo` can be used to obtain useful information (name, size, format, and color type) concerning the newly-created image file.

There are many ways of transforming the original satellite image into a practical file format. The image data could, for instance, be stored as an *indexed color image*, which consists of two parts: a colormap array and a data array. The colormap array is an m-by-3 array containing floating-point values between 0 and 1. Each column specifies the intensity of the red, green and blue colors. The data array is an x-by-y array containing integer elements corresponding to the lines m of the colormap array, i.e., the specific RGB representation of a certain color. Let us transfer the above RGB image into an indexed image. The colormap of the image should contain 16 different colors. The result of

```
[I4,map] = rgb2ind(I1,16);
imshow(I1), figure, imshow(I4,map)
```

saved as another JPEG file using

```
imwrite(I4,map,'unconform_ind.jpg')
```

clearly shows the difference between the original 24-bit RGB image (256^3 or about 16.7 million different colors) and a color image of only 16 different colors. The display of the image uses the default colormap of MATLAB. Typing

```
imshow(I4,map)
cmap = colormap
```

actually retrieves the 16-by-3 array of the current colormap

```
cmap =
    0.0588    0.0275    0.0745
    0.5490    0.5255    0.4588
    0.7373    0.7922    0.7020
    0.3216    0.2706    0.2667
    0.6471    0.6784    0.6157
    0.7961    0.8549    0.9176
    0.4510    0.3922    0.3333
    0.2000    0.1451    0.1451
    0.4824    0.5412    0.5843
    0.4039    0.4078    0.4784
    0.6667    0.7020    0.7451
    0.8980    0.8745    0.7255
    0.2824    0.2902    0.4039
    0.9569    0.9647    0.9608
    0.1765    0.1686    0.2902
    0.5843    0.5843    0.6078
```

We can replace the default colormap by any other built-in colormap. Typing

```
help graph3d
```

lists the available colormaps. As an example we can use

```
imshow(I4,map)
colormap(hot)
```

to display the image with a black-red-yellow-white colormap. Typing

```
edit hot
```

reveals that `hot` is a function creating the *m*-by-3 array containing floating-point values between 0 and 1. We can also design our own colormaps, either by manually creating an *m*-by-3 array or by creating another function similar to `hot`. As an example the colormap `precip.m` (which is a yellow-blue colormap included in the book's file collection) was created to display precipitation data, with yellow representing low rainfall and blue representing high rainfall. Alternatively, we can also use random numbers

```
rng(0)
map = rand(16,3);
imshow(I4,map)
```

Gallery 8.1

to display the image with random colors. Finally, we can create an indexed color image of three different colors, displayed with a simple colormap of full intensity red, green and blue.

```
[I5,map] = rgb2ind(I1,3);
imshow(I5,[1 0 0;0 1 0;0 0 1])
```

Typing

```
imwrite(I4,map,'unconform_rgb.jpg')
```

saves the result as another JPEG file.

8.4 Importing, Processing and Exporting LANDSAT Images

The Landsat project is a satellite remote sensing program jointly managed by the US National Aeronautics and Space Administration (NASA) and the US Geological Survey (USGS), which began with the launch of the Landsat 1 satellite (originally known as the Earth Resources Technology Satellite 1) on 23rd July 1972. The latest in a series of successors is the Landsat 8 satellite, launched on 11th February 2013 (Ochs et al. 2009, Irons et al. 2011). It has two sensors, the Operational Land Imager (OLI) and the Thermal Infrared

Sensor (TIRS). These two sensors provide coverage of the global landmass at spatial resolutions of 30 meters (visible, NIR, SWIR), 100 meters (thermal), and 15 meters (panchromatic) (Ochs et al. 2009, Irons et al. 2011). General information concerning the Landsat program can be obtained from the webpage

```
http://landsat.gsfc.nasa.gov/?page_id=7195
```

Landsat data, together with data from other NASA satellites, can be obtained from the webpage

```
http://earthexplorer.usgs.gov
```

On this webpage we first select the desired map section in the *Search Criteria*, either by entering the coordinates of the four corners of the map or by zooming into the area of interest and selecting *Use Map*. As an example we enter the coordinates 4°42'40.72"N 36°51'10.47"E of the Chew Bahir Basin in the Southern Ethiopian Rift. We then choose *L8 OLI/TIRS* from the *Landsat Archive* as the *Data Set* and click *Results*. Clicking *Results* produces a list of records, together with a toolbar for previewing and downloading data. By clicking the *Show Browse Overlay* button we can examine the images for cloud cover. We find the cloud-free image

```
Entity ID: LC81690572013358LGN00
Coordinates: 4.33915,36.76225
Acquisition Date: 24-DEC-13
Path: 169
Row: 57
```

taken on 24th December 2013. We need to register with the USGS website, log on, and then download the Level 1 GeoTIFF Data Product (897.5 MB), which is then stored on the hard drive in the file *LC81690572013358LGN00.tar.gz*. The *.tar.gz* archive contains separate files for each spectral band as well as a metadata file containing information about the data. We use band 4 (Red, 640–670 nm), band 3 (Green, 530–590 nm) and band 2 (Blue, 450–510 nm), each of which has a 30 m resolution. We can import the 118.4 MB TIFF files using

```
clear

I1 = imread('LC81690602013150LGN00_B4.TIF');
I2 = imread('LC81690602013150LGN00_B3.TIF');
I3 = imread('LC81690602013150LGN00_B2.TIF');
```

Typing

```
whos
```

reveals that the data are in a unsigned 16-bit format `uint16`, i.e., the maximum range of the data is from 0 to 2^{16}=65,536.

```
  I1        7771x7611           118290162  uint16
  I2        7771x7611           118290162  uint16
  I3        7771x7611           118290162  uint16
```

For quantitative analyses these digital number (DN) values need to be converted to radiance and reflectance values, which is beyond the scope of the book. The radiance is the power density scattered from the earth in a particular direction and has the units of watts per square meter per steradian (Wm^{-2} sr^{-1}) (Richards 2013). The radiance values need to be corrected for atmospheric and topographic effects to obtain earth surface reflectance percentages. The Landsat 8 Handbook provides the necessary information on these conversions:

```
https://landsat.usgs.gov/Landsat8_Using_Product.php
```

We will instead use the Landsat 8 data to create an RGB composite of bands 4, 3, and 2 to be used for fieldwork. Since the image has a relatively low level of contrast, we use `adapthisteq` to perform a contrast-limited adaptive histogram equalization (CLAHE) (Zuiderveld 1994). Unlike `histeq` used in the previous section, the `adapthisteq` algorithm works on small regions (or tiles) of the image, rather than on the entire image. The neighboring tiles are then combined using bilinear interpolation to eliminate edge effects.

```
I1 = adapthisteq(I1,'ClipLimit',0.1,'Distribution','Rayleigh');
I2 = adapthisteq(I2,'ClipLimit',0.1,'Distribution','Rayleigh');
I3 = adapthisteq(I3,'ClipLimit',0.1,'Distribution','Rayleigh');
```

Using `ClipLimit` with a real scalar between 0 and 1 limits the contrast enhancement, while higher numbers result in increased contrast; the default value is 0.01. The `Distribution` parameter sets the desired histogram shape for the tiles by specifying a distribution type, such as `Uniform`, `Rayleigh`, or `Exponential`. Using a `ClipLimit` of 0.1 and a `Rayleigh` distribution yields good results. The three bands are concatenated to a 24-bit RGB images using `cat`.

```
I = cat(3,I1,I2,I3);
```

We only display that section of the image containing the Chew Bahir Basin (using axes limits) and hide the coordinate axes. We scale the images to 10% of the original size to fit the computer screen.

```
axes('XLim',[3000 5000],'YLim',[1000 4000],'Visible','Off'), hold on
```

```
imshow(I,'InitialMagnification',10)
```

Exporting the processed image from the Figure Window, we only save the image at the monitor's resolution. To obtain an image of the basins at a higher resolution, we use the command

```
imwrite(I(1000:4000,3000:5000,:),'chewbahirbasin.tif','tif')
```

This command saves the RGB composite as a TIFF-file *chewbahirbasin.tif* (about 36.3 MB) in the working directory, which can then be processed using other software such as Adobe Photoshop.

According to the USGS Landsat webpage, Landsat data are amongst the most geometrically and radiometrically corrected data available. Data anomalies do occasionally occur, however, of which the most common types are listed on the USGS webpage:

```
http://landsat.usgs.gov/science_an_anomalies.php
```

We explore one of these types of anomaly as an example, i.e., artifacts known as *Single Event Upsets* (SEUs) that cause anomalously high values in the image, similar to the *Impulse Noise* (IN) that is also described on the same webpage. These anomalies occur in some, but not all, Landsat images and similarly anomalous high or low values can also occur in other satellite images. We therefore use a part of a Landsat 7 image covering an area in the southern Ethiopian Rift, acquired by the Enhanced Thematic Mapper (ETM+) instrument of that satellite. We can load and display the image using

```
clear

I1 = imread('ethiopianrift_blue.tif');
imshow(I1,'InitialMagnification',200), title('Original Image')
```

The parameter `InitialMagnification` is a numeric scalar that scales the image to, as an example, 200% magnification. The image `I1` shows numerous randomly-distributed anomalously high or low values, as well as a parallel track of paired anomalies in the right half of the image. We first apply a 10-by-10 pixel median filter to the image (see Section 8.8):

```
I2 = medfilt2(I1,[10,10],'symmetric');
imshow(I2,'InitialMagnification',200)
title('Median Filtered Image')
```

Using the option `symmetric` with the function `medfilt2` extends `I2` symmetrically at the image boundaries by reflecting it across its boundaries, instead of padding the image with zeros at the boundaries (by default). The

median-filtered version of the I2 image is, of course, very smooth compared to the original I1 image. We would, however, lose a lot of detail if we used this version of the image. We next subtract the median-filtered image I2 from original image I1, which yields the image I3.

```
I3 = imsubtract(I1,I2);
imshow(I3,'InitialMagnification',200)
title('I1-I2')
```

We then subtract the original image I1 from the median-filtered image I2, which yields the image I4.

```
I4 = imsubtract(I2,I1);
imshow(I4,'InitialMagnification',200)
title('I2-I1')
```

We next replace the original pixels with their median-filtered versions if the difference between the median-filtered image I2 and the original image I1 is great than 10 in both directions (as it is in our example).

```
I5 = I1;
I5(I3>10 | I4>10) = I2(I3>10 | I4>10);
imshow(I5,'InitialMagnification',200)
title('Despeckled Image')
```

The image I5 obtained using this approach is the despeckled version of the image I1. We can also explore the pixel values of both versions of the image (I1 and I5) in a 3D surface plot, using

```
subplot(1,2,1)
I1S = im2double(I1);
surface(I1S), colormap jet, caxis([0 1])
shading interp, view(120,33), axis off
axis([1 size(I1,1) 1 size(I1,2) min(I1S(:)) max(I1S(:))])

subplot(1,2,2)
I5S = im2double(I5);
surface(I5S), colormap jet, caxis([0 1])
shading interp, view(120,33), axis off
axis([1 size(I1,1) 1 size(I1,2) min(I1S(:)) max(I1S(:))])
```

We need convert the image data to class double using im2double in order to be able to display the data using surface. Finally, we can display both images in the same figure window

```
subplot(1,2,1), imshow(I1), title('Original Image')
subplot(1,2,2), imshow(I5), title('Despeckled Image')
```

to see the result of despeckling the image I1 (Fig. 8.2).

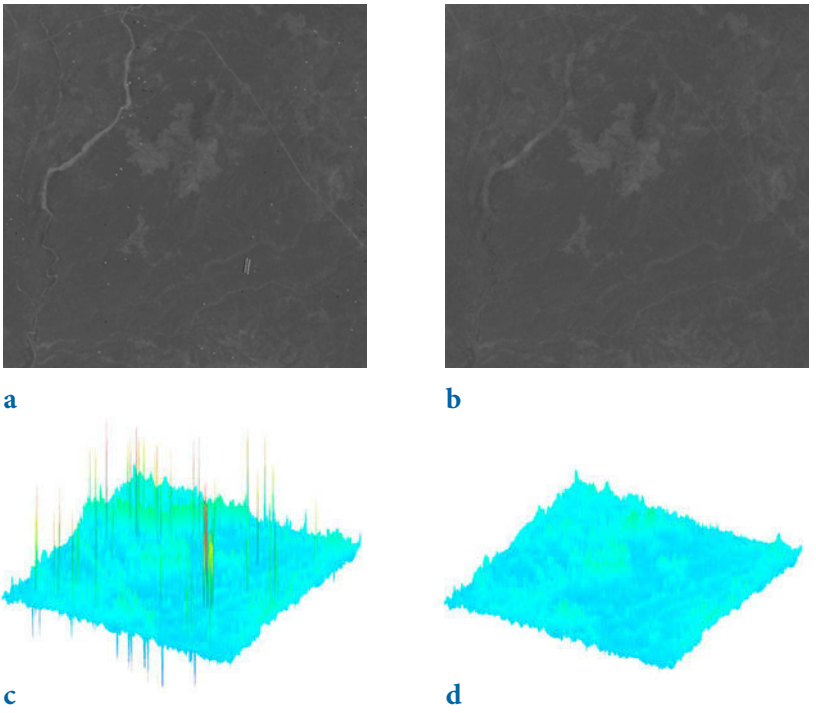


Fig. 8.2 Despeckled section of the blue band of a Landsat image covering the Chew Bahir catchment in Southern Ethiopia; **a** original image, b despeckled image, c surface plots of the original image, and d surface plot of the image after despeckling. Original image courtesy of the Landsat Program of the US National Aeronautics and Space Administration (NASA) and the US Geological Survey (USGS).

8.5 Importing and Georeferencing TERRA ASTER Images

In Section 8.3 we used a processed ASTER image that we downloaded from the ASTER webpage. In this section we will use raw data from this sensor. The ASTER sensor is mounted on the TERRA satellite launched in 1999, part of the Earth Observing System (EOS) series of multi-national NASA satellites (Abrams and Hook 2002). ASTER stands for *Advanced Spaceborne Thermal Emission and Reflection Radiometer*, providing high-resolution (15 to 90 meter) images of the earth in 14 bands, including three visible to near infrared bands (VNIR bands 1 to 3), six short-wave infrared bands (SWIR bands 4 to 9), and five thermal (or long-wave) infrared bands (TIR bands 10 to 14). ASTER images are used to map the surface temperature, emissivity, and reflectance of the earth's surface. The 3rd near infrared band is recorded twice: once with the sensor pointing directly downwards (band 3N, where

N stands for *nadir* from the Arabic word for *opposite*), as it does for all other channels, and a second time with the sensor angled backwards at 27.6° (band 3B, where *B* stands for backward looking). These two bands are used to generate ASTER digital elevation models (DEMs).

The ASTER instrument produces two types of data: Level-1A (L1A) and Level-1B (L1B) data (Abrams and Hook 2002). Whereas the L1A data are reconstructed, unprocessed instrument data, the L1B data are radiometrically and geometrically corrected. Any data that ASTER has already acquired are available; they can be located by searching the Japan Space Systems GDS ASTER/PALSAR Unified Search Site and can be ordered from

```
http://gds.ersdac.jspacesystems.or.jp/?lang=en
```

or from NASA Reverb

```
http://reverb.echo.nasa.gov/reverb/
```

As an example we process an image from an area in Kenya showing Lake Naivasha (0°46'31.38"S 36°22'17.31"E). The Level-1A data are stored in two files

```
AST_L1A_003_03082003080706_03242003202838.hdf
AST_L1A_003_03082003080706_03242003202838.hdf.met
```

The first file (116 MB) contains the actual raw data, whereas the second file (102 KB) contains the header, together with all sorts of information about the data. We save both files in our working directory. Since the file name is very long, we first save it in the `filename` variable and then use `filename` instead of the long file name. We then need to modify only this single line of MATLAB code if we want to import and process other satellite images.

```
filename = 'AST_L1A_003_03082003080706_03242003202838.hdf';
```

The Image Processing Toolbox contains various tools for importing and processing files stored in the hierarchical data format (HDF). The graphical user interface (GUI) based import tool for importing certain parts of the raw data is

```
hdftool('filename')
```

This command opens a GUI that allows us to browse the content of the HDF-file *naivasha.hdf*, obtains all information on the contents, and imports certain frequency bands of the satellite image. Alternatively, the command `hdfread` can be used as a quicker way of accessing image data. The *vnir_Band3n*, *vnir_Band2*, and *vnir_Band1* typically contain much information

about lithology (including soils), vegetation and water on the earth's surface. These bands are therefore usually combined into 24-bit RGB images. We first read the data

```
I1 = hdfread(filename,'VNIR_Band3N','Fields','ImageData');
I2 = hdfread(filename,'VNIR_Band2','Fields','ImageData');
I3 = hdfread(filename,'VNIR_Band1','Fields','ImageData');
```

These commands generate three 8-bit image arrays, each representing the intensity within a certain infrared (IR) frequency band of a 4200-by-4100 pixel image. We are not using the data for quantitative analyses and therefore do not need to convert the digital number (DN) values into radiance and reflectance values. The *ASTER User Handbook* provides the necessary information on these conversions (Abrams and Hook 2002). Instead, we will process the ASTER image to create a georeferenced RGB composite of bands 3N, 2 and 1, to be used in fieldwork. We first use a contrast-limited adaptive histogram equalization method to enhance the contrast in the image by typing

```
I1 = adapthisteq(I1);
I2 = adapthisteq(I2);
I3 = adapthisteq(I3);
```

and then concatenate the result to a 24-bit RGB image using `cat`.

```
naivasha_rgb = cat(3,I1,I2,I3);
```

As with the previous examples, the 4200-by-4100-by-3 array can now be displayed using

```
imshow(naivasha_rgb,'InitialMagnification',10)
```

We set the initial magnification of this very large image to 10%. MATLAB scales images to fit the computer screen. Exporting the processed image from the Figure Window, we only save the image at the monitor's resolution. To obtain an image at a higher resolution, we use the command

```
imwrite(naivasha_rgb,'naivasha.tif','tif')
```

This command saves the RGB composite as a TIFF-file *naivasha.tif* (ca. 52 MB) in the working directory, which can then be processed using other software such as Adobe Photoshop. The processed ASTER image does not yet have a coordinate system and therefore needs to be tied to a geographical reference frame (*georeferencing*). The HDF browser

```
hdftool('naivasha.hdf')
```

can be used to extract the geodetic coordinates of the four corners of the image. This information is contained in the header of the HDF file. Having launched the HDF tool, we select on the uppermost directory called *naivasha.hdf* and find a long list of file attributes in the upper right panel of the GUI, one of which is *productmetadata.0*, which includes the attribute *scenefourcorners*. We collect the coordinates of the four scene corners into a single array `inputpoints`:

```
inputpoints(1,:) = [36.214332 -0.319922];  % upper left corner
inputpoints(2,:) = [36.096003 -0.878267];  % lower left corner
inputpoints(3,:) = [36.770406 -0.400443];  % upper right corner
inputpoints(4,:) = [36.652213 -0.958743];  % lower right corner
```

It is important to note that the coordinates contained in *productmetadata.0* need to be flipped in order to have *x=longitudes* and *y=latitudes*. The four corners of the image correspond to the pixels in the four corners of the image, which we store in a variable named `basepoints`.

```
basepoints(1,:) = [1,1];                   % upper left pixel
basepoints(2,:) = [1,4200];                % lower left pixel
basepoints(3,:) = [4100,1];                % upper right pixel
basepoints(4,:) = [4100,4200];             % lower right pixel
```

The function `fitgeotrans` now takes the pairs of control points, `inputpoints` and `basepoints`, and uses them to infer a spatial transformation matrix `tform`.

```
tform = fitgeotrans(inputpoints,basepoints,'affine');
```

We next determine the limits of the input for georeferencing (i.e., of the original image `naivasha_rgb`) using `size`, which yields `xLimitsIn` and `yLimitsIn`. Adding a value of 0.5 to both `xLimitsIn` and `yLimitsIn` prevents the edges of the image from being truncated during the affine transformation. We then determine the limits of the output (i.e. of the georeferenced image, which is subsequently called `newnaivasha_rgb`) using `outputLimits`, which yields `XBounds` and `YBounds`.

```
xLimitsIn = 0.5 + [0 size(naivasha_rgb,2)];
yLimitsIn = 0.5 + [0 size(naivasha_rgb,1)];
[XBounds,YBounds] = outputLimits(tform,xLimitsIn,yLimitsIn);
```

We then use `imref2d` to reference the image to a world (or global) coordinate system.

```
Rout = imref2d(size(naivasha_rgb),XBounds,YBounds);
```

An `imref2d` object encapsulates the relationship between the intrinsic coordinates anchored to the rows and columns of the image, and the spatial

location of the same row and column locations within a world coordinate system. Finally, the affine transformation can be applied to the original RGB composite `naivasha_rgb` in order to obtain a georeferenced version of the satellite image `newnaivasha_rgb` with the same size as `naivasha_rgb`.

```
newnaivasha_rgb = imwarp(naivasha_rgb,tform,'OutputView',Rout);
```

An appropriate grid for the image can now be computed. The grid is typically defined by the minimum and maximum values for the longitude and latitude. The vector increments are then obtained by dividing the ranges of the longitude and latitude by the array's dimensions and then subtracting one from the results. Note the difference between the MATLAB numbering convention and the common coding of maps used in published literature. The north/south suffix is generally replaced by a negative sign for south,

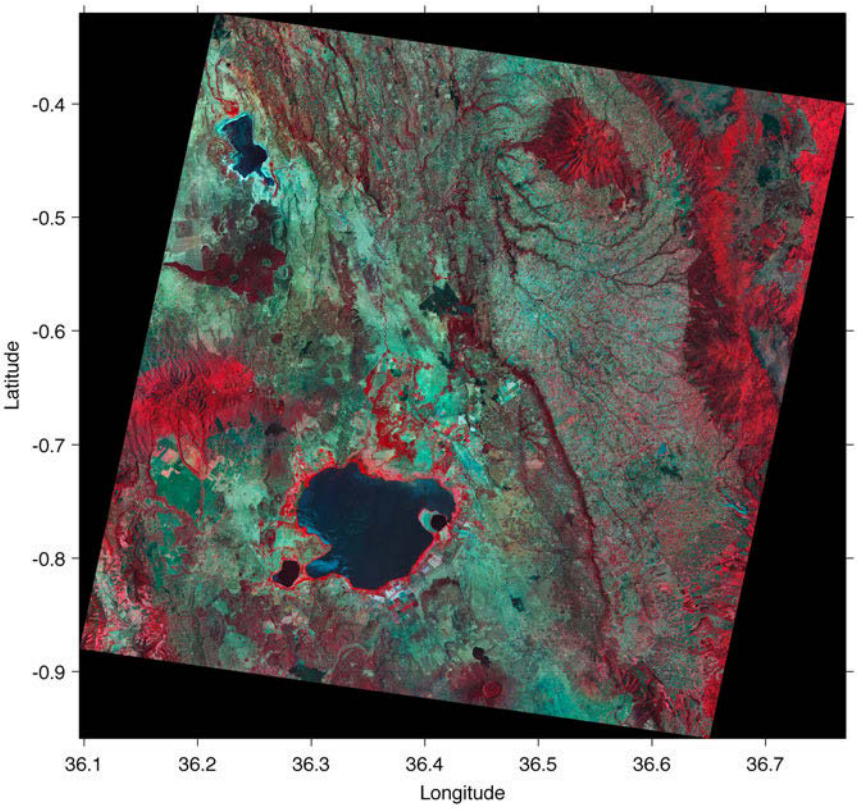


Fig. 8.3 Geoferenced RGB composite of a TERRA-ASTER image using the infrared bands *vnir_Band3n, 2* and *1*. The result is displayed using `imshow`. Original image courtesy of NASA/GSFC/METI/ERSDAC/JAROS and U.S./Japan ASTER Science Team.

whereas MATLAB coding conventions require negative signs for north.

```
X = 36.096003 : (36.770406 - 36.096003)/4100 : 36.770406;
Y = -0.958743 : ( 0.958743 -  0.319922)/4200 : -0.319922;
```

The georeferenced image is displayed with coordinates on the axes and a superimposed grid (Fig. 8.3). By default, the function `imshow` inverts the latitude axis when images are displayed by setting the `YDir` property to `Reverse`. To invert the latitude axis direction back to normal, we need to set the `YDir` property to `Normal` by typing

```
imshow(newnaivasha_rgb,'XData',X,'YData',Y,'InitialMagnification',10)
axis on, grid on, set(gca,'YDir','Normal')
xlabel('Longitude'), ylabel('Latitude')
title('Georeferenced ASTER Image')
```

Exporting the image is possible in many different ways, for example using

```
print -djpeg70 -r600 naivasha_georef.jpg
```

to export it as a JPEG file *naivasha_georef.jpg*, compressed to 70% and with a resolution of 600 dpi.

In the previous example we used the geodetic coordinates of the four corners to georeference the ASTER image. The Image Processing Toolbox also includes functions to automatically align two images that are shifted and/or rotated with respect to each other, cover slightly different areas, or have a different resolutions. We use two ASTER images of the Suguta Valley in the Northern Kenya Rift as an example. The images have been processed in the same way as described for the image of Lake Naivasha and exported as TIFF files using `imwrite`. The image in the file *sugutavalley_1.tif* was taken on 20th February 2003 and the second image in *sugutavalley_2.tif* was taken on 31st August 2003, both just after 8 o'clock in the morning. Lake Logipi, in the center of the images, is much larger in the second image than in the first image. The original images are otherwise almost identical, except for the second image being shifted slightly towards the east. To demonstrate the automatic alignment of the images, the second image has been rotated counterclockwise by five degrees. Furthermore, both images have been cropped to the area of the Suguta Valley, including a small section of the rift shoulders to the west and to the east. We import both images using

```
clear

image1 = imread('sugutavalley_1.tif');
image2 = imread('sugutavalley_2.tif');
```

The size of `image1` is 666-by-329-by-3, while `image2` is slightly smaller: 614-by-270-by-3. We display the images by typing

```
subplot(1,2,1), imshow(image1)
subplot(1,2,2), imshow(image2)
```

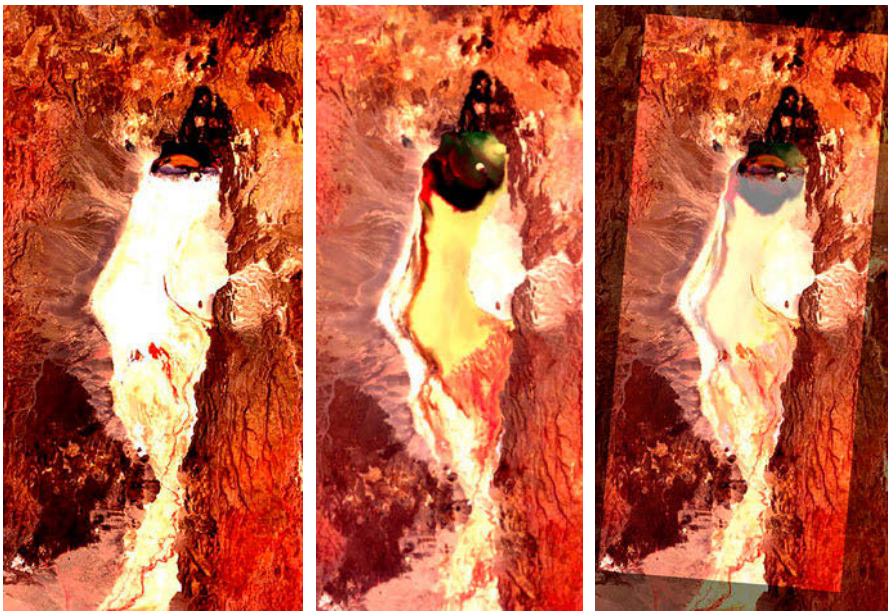
The function `imregconfig` creates optimizer and metric configurations that we transfer into `imregister` to perform intensity-based image registration,

```
[optimizer, metric] = imregconfig('monomodal');
```

where `monomodal` assumes that the images were captured by the same sensor. We can use this configurations to calculate the spatial transformation matrix `tform` using the transformation type `affine`, as in the previous example.

```
tform = imregtform(image2(:,:,1),image1(:,:,1), ...
    'affine',optimizer,metric);
```

This transformation can be applied to `image2` in order to automatically align it with `image1`.



a **b** **c**

Fig. 8.4 Automatically aligned TERRA-ASTER images of the Suguta Valley in the Northern Kenya Rift; **a** first image taken on 20th February 2003, **b** second image taken on 31st August 2003, and **c** overlay of the second image aligned with the first image. Original image courtesy of NASA/GSFC/METI/ERSDAC/JAROS and U.S./Japan ASTER Science Team.

```
image2_reg = imwarp(image2,tform,'OutputView', ...
    imref2d(size(image1)));
```

We can compare the result with the original images using

```
subplot(1,3,1), imshow(image1)
subplot(1,3,2), imshow(image2)
subplot(1,3,3), imshowpair(image1,image2_reg,'blend')

print -djpeg70 -r600 sugutavalley_aligned.jpg
```

As we can see, the second image is now nicely aligned with the first image (Fig. 8.4). The two images can now be used to map changes in the area (e.g., in the size of the lake) between 20th February and 31st August 2003. This script can also be used to automatically align other images, in particular those captured by different sensors.

8.6 Processing and Exporting EO-1 Hyperion Images

The Earth Observing-1 Mission (EO-1) satellite is part of the New Millennium Program of the US National Aeronautics and Space Administration (NASA) and the US Geological Survey (USGS), which began with the launch of this satellite on 21st November 2000. EO-1 has two sensors: the Advanced Land Image (ALI) has nine multispectral bands with a 30 m spatial resolution and a panchromatic band with a 10-m resolution, and the hyperspectral sensor (Hyperion) has 220 bands between 430 and 2,400 nm (Mandl et al. 2002, Line 2012). General information about the EO-1 program can be obtained from the webpage

```
http://eo1.gsfc.nasa.gov
```

Hyperion data (together with data from of other NASA satellites) are freely available from the webpage

```
http://earthexplorer.usgs.gov
```

On this webpage we first select the desired map section in the *Search Criteria*, either by entering the coordinates of the four corners of the map or by zooming into the area of interest and selecting *Use Map*. As an example we enter the coordinates 2°8'37.58"N 36°33'47.06"E of the Suguta Valley in the Northern Kenya Rift. We then choose *Hyperion* from the *EO-1* collection as the *Data Set* and click *Results*. Clicking *Results* produces a list of records, together with a toolbar for previewing and downloading data. Clicking the *Show Browse Overlay* button allows us to examine the images for cloud cover. We find the cloud-free image

```
Entity ID: EO1H1690582013197110KF_PF2_01
Acquisition Date: 16-JUL-13
Target Path: 169
Target Row: 58
```

taken on 16th July 2013. As before, we need to register with the USGS website, log on, and then download the radiometrically corrected (but not geometrically corrected) Level 1R (L1R) product (215.3 MB), which is then stored on the hard drive in the file *LEO1H1690582013197110KF_1R.ZIP*. The *.ZIP* archive consists of a metadata file (*.MET*), a Federal Geographic Data Committee (FGDC) metadata file (*.fgdc*), an HDF data set file (*.L1R*), and multiple auxiliary files. The EO-1 User's Guides provides some useful information on the data formats of these files (Barry 2001, Beck 2003). We can import the data from the *EO1H1690582013197110KF.L1R* file using

```
clear

HYP = hdfread('EO1H1690582013197110KF.L1R',...
    '/EO1H1690582013197110KF.L1R', ...
    'Index', {[1 1 1],[1 1 1],[3189 242 256]});
```

The parameter `Index` is a three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set. The value of `start` specifies the position in the file to begin reading. In our example it starts reading at the beginning of the file, i.e., starting from the `[1 1 1]` element. The value of `stride` defines the interval between the values to be read, which in our example is `[1 1 1]`, i.e., every element of the data set is to be read. The value of `edge` specifies the size of the data in the file; in our example the dimensions of the data are 3,189-by-242-by-256. Typing

```
whos
```

shows how the hyperspectral image is stored in the workspace:

```
HYP       3189x242x256            395129856  int16
```

The details indicate that the image is stored as a 3,189-by-242-by-256 array, representing a 3,189-by-256 array for each of the 242 spectral bands. The listing of the current variables in the workspace also gives the information *int16* array, i.e., each array element representing one pixel contains signed 16-bit integers. We need to permute the array to move the bands to the third dimension by typing

```
HYP = permute(HYP,[1 3 2]);
```

We next need to determine the radiance values from the digital number

(DN) values in HYP. The radiance is the power density scattered from the earth in a particular direction and has the units of watts per square meter per steradian (Wm^{-2} sr^{-1}) (Richards 2013). The EO-1 User Guide (v. 2.3) provides the necessary information on these conversions in its *Frequently Asked Questions* (FAQ) section (Beck 2003). According to this document, the radiance HYPR for the visible and near-infrared (VNIR) bands (bands 1 to 70) is calculated by dividing the digital number in HYP by 40. The radiance for the shortwave infrared (SWIR) bands (bands 71 to 242) is calculated by dividing HYP by 80.

```
HYPR(:,:,1:70) = HYP(:,:,1:70)/40;
HYPR(:,:,71:242) = HYP(:,:,71:242)/80;
```

For quantitative analyses, the radiance values HYPR need to be corrected for atmospheric and topographic effects. This correction, which yields earth surface reflectance values (in percentages), is beyond the scope of the book. The EO-1 User Guide (v. 2.3) again explains several methods to convert radiance to reflectance values (Beck 2003). A simple way to convert radiance to reflectance for relatively clear Hyperion images is given in

```
https://eo1.usgs.gov/faq/question?id=21
```

We will instead process the Hyperion image to create a georeferenced RGB composite of bands 29, 23 and 16, to be used in fieldwork. The header file *O1H1690582013197110KF.HDR* contains (among other things) the wavelengths corresponding to the 242 spectral bands. We can read the wavelengths from the file using textscan:

```
fid = fopen('EO1H1690582013197110KF.hdr');
C = textscan(fid,'%f %f %f %f %f %f %f %f',...
    'Delimiter',',','Headerlines',257,'CollectOutput',1)
fclose(fid);
```

This script opens the header file for read only access using fopen and defines the file identifier fid, which is then used to read the text from the file with textscan, and to write it into the array C. The character string %f %f %f %f %f %f %f %f defines the conversion specifiers enclosed in single quotation marks, where %f stands for the double-precision floating-point 64-bit output class. The parameter Headerlines is set to 257, which means that the first 257 lines are ignored when reading the file. If the parameter CollectOutput is 1 (true), textscan concatenates output cells of the same data type into a single array. Function fclose closes the file defined by the file identifier fid. The array C is a cell array, which is a data type with indexed containers called cells. We can easily obtain the wavelengths from C using

```
wavelengths = C{1};
wavelengths = wavelengths';
wavelengths = wavelengths(isnan(wavelengths(:))==0);
```

We can now plot the radiance `HYPR` of the VNIR bands (blue) and SWIR bands (in red) in a single plot.

```
plot(wavelengths(1:60),squeeze(HYPR(536,136,1:60)),'b',...
     wavelengths(71:242),squeeze(HYPR(536,136,71:242)),'r')
```

According to v. 2.3 of the EO-1 User Guide (Beck 2003), Hyperion records 220 unique spectral channels collected from a complete spectrum covering 357 to 2,576 nm. The L1R product has 242 bands but 198 bands are calibrated. Because of an overlap between the VNIR and SWIR focal planes, there are only 196 unique channels. Calibrated channels are 8–57 for the VNIR, and 77–224 for the SWIR. The bands that are not calibrated are set to zero in those channels.

In order to create an RGB composite of bands 29, 23 and 16, we can extract the bands from the radiance values data `HYPR` by typing

```
HYP1 = HYPR(:,:,29);
HYP2 = HYPR(:,:,23);
HYP3 = HYPR(:,:,16);
```

To display the data with `imshow` we need convert the signed integer 16-bit (`int16`) data to unsigned integer 8-bit data (`uint8`). For this purpose, we first obtain an overview of the range of the data using a histogram plot with 100 classes.

```
subplot(1,3,1), histogram(double(HYP1(:)),100), title('Band 29')
subplot(1,3,2), histogram(double(HYP2(:)),100), title('Band 23')
subplot(1,3,3), histogram(double(HYP3(:)),100), title('Band 16')
```

As we see, the radiance values of most pixels from the spectral bands 29, 23 and 16 lie between 0 and 200 Wm^{-2} sr^{-1}. Several functions are available for converting the data from `int16` to `uint8`. The function `im2uint8` rescales the data to the range [0,255] and offsets it if necessary. The function `uint8` simply rounds all values in the range [0,255] to the nearest integer; all values less than 0 are set to 0 and all values greater than 255 are set to 255. The function `mat2gray(A,[AMIN AMAX])` converts an arbitrary array `A` to an intensity image `I` containing values in the range 0 (black) to 1.0 (full intensity or white), where `[AMIN AMAX]` can be used to limit the range of the original data, which is scaled to the range [0,1]. Since most of our radiance values are within the range [0,255], we use `uint8` to convert our data to the `uint8` data type without losing much information.

```
HYP1 = uint8(HYP1);
HYP2 = uint8(HYP2);
HYP3 = uint8(HYP3);
```

Again, displaying the radiance values of the three bands in a histogram using

```
subplot(1,3,1), histogram(single(HYP1(:)),30), title('Band 29')
subplot(1,3,2), histogram(single(HYP2(:)),30), title('Band 23')
subplot(1,3,3), histogram(single(HYP3(:)),30), title('Band 16')
```

reveals that most radiance values are actually within the range [20,80]. Instead of using `histogram` we can also use `imhist` to display the `uint8` data.

```
subplot(1,3,1), imhist(HYP1(:)), title('Band 29')
subplot(1,3,2), imhist(HYP2(:)), title('Band 23')
subplot(1,3,3), imhist(HYP3(:)), title('Band 16')
```

We then use `histeq` to enhance the contrast in the image and concatenate the three bands to a 3,189-by-242-by-3 array.

```
HYP1 = histeq(HYP1);
HYP2 = histeq(HYP2);
HYP3 = histeq(HYP3);

subplot(1,3,1), imhist(HYP1(:)), title('Band 29')
subplot(1,3,2), imhist(HYP2(:)), title('Band 23')
subplot(1,3,3), imhist(HYP3(:)), title('Band 16')

HYPC = cat(3,HYP1,HYP2,HYP3);
```

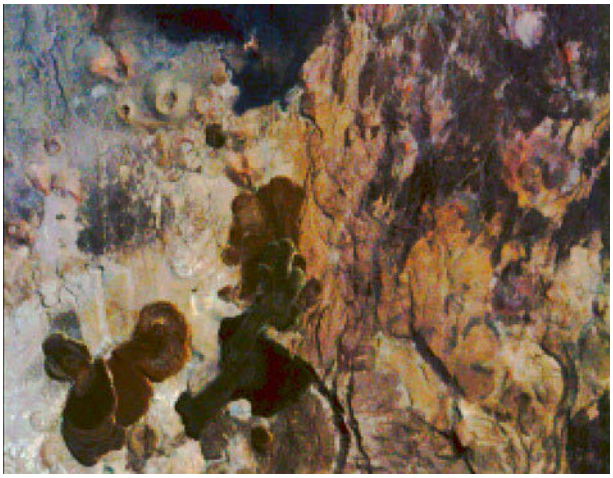


Fig. 8.5 RGB composite of an EO-1 Hyperion image using VNIR bands 29, 23 and 16, showing the Barrier Volcanic Complex in the Suguta Valley of the Northern Kenya Rift. The image was acquired on 16th July 2013. Original image courtesy of NASA EO-1 Mission.

Finally, we can display the entire image using

```
imshow(HYPC)
```

or, as an alternative, that part of the image showing the Barrier Volcanic Complex in the northern Suguta Valley (Fig. 8.5).

```
imshow(HYPC(900:1100,:,:))
```

Exporting the image is possible in many different ways, for example using

```
print -r600 -dtiff barrier.tif
```

to export it as a TIFF file *barrier.tif* with a resolution of 600 dpi.

8.7 Digitizing from the Screen

On-screen digitizing is a widely-used image processing technique. While practical digitizer tablets exist in all formats and sizes, most people prefer digitizing vector data from the screen. Examples of this type of application include the digitizing of river networks and catchment areas on topographic maps, of the outlines of lithologic units on geological maps, of landslide distributions on satellite images, and of mineral grain distributions in microscope images. The digitizing procedure consists of the following steps. The image is first imported into the workspace. A coordinate system is then defined, allowing the objects of interest to be entered by moving a cursor or cross hair onto it and clicking the mouse button. The result is a two-dimensional array of *xy* data, such as longitudes and latitudes of the corner points of a polygon, or the coordinates of the objects of interest in a particular area.

The function `ginput` included in the standard MATLAB toolbox allows graphical input from the screen, using a mouse. It is generally used to select points, such as specific data points, from a figure created by an arbitrary graphics function such as `plot`. The function `ginput` is often used for interactive plotting, i.e., the digitized points appear on the screen after they have been selected. The disadvantage of the function is that it does not provide coordinate referencing on an image. We therefore use a modified version of the function, which allows an image to be referenced to an arbitrary rectangular coordinate system. Save the following code for this modified version of the function `ginput` in a text file *minput.m*.

```
function data = minput(imagefile)
% Specify the limits of the image
xmin = input('Specify xmin! ');
```

```
xmax = input('Specify xmax! ');
ymin = input('Specify ymin! ');
ymax = input('Specify ymax! ');

% Read image and display
B = imread(imagefile);
a = size(B,2); b = size(B,1);
imshow(B);

% Define lower left and upper right corner of image
disp('Click on lower left and upper right corner, then <return>')
[xcr,ycr] = ginput;
XMIN = xmin-((xmax-xmin)*xcr(1,1)/(xcr(2,1)-xcr(1,1)));
XMAX = xmax+((xmax-xmin)*(a-xcr(2,1))/(xcr(2,1)-xcr(1,1)));
YMIN = ymin-((ymax-ymin)*ycr(1,1)/(ycr(2,1)-ycr(1,1)));
YMAX = ymax+((ymax-ymin)*(b-ycr(2,1))/(ycr(2,1)-ycr(1,1)));

% Digitize data points
disp('Click on data points to digitize, then <return>')
[xdata,ydata] = ginput;
XDATA = XMIN + ((XMAX-XMIN)*xdata/size(B,2));
YDATA = YMIN + ((YMAX-YMIN)*ydata/size(B,1));
data(:,1) = XDATA; data(:,2) = YDATA;
```

The function `minput` has four stages. In the first stage the user enters the limits of the coordinate axes as reference points for the image. The image is then imported into the workspace and displayed on the screen. The third stage uses `ginput` to define the upper left and lower right corners of the image. In the fourth stage the relationship between the coordinates of the two corners on the figure window and the reference coordinate system is then used to compute the transformation for all of the digitized points.

As an example we use the image stored in the file *naivasha_georef.jpg* and digitize the outline of Lake Naivasha in the center of the image. We activate the new function `minput` from the Command Window using the commands

```
clear

data = minput('naivasha_georef.jpg')
```

The function first asks for the coordinates of the limits of the x-axis and the y-axis, for the reference frame. We enter the corresponding numbers and press *return* after each input.

```
Specify xmin! 36.1
Specify xmax! 36.7
Specify ymin! -1
Specify ymax! -0.3
```

The function then reads the file *naivasha_georef.jpg* and displays the image. We ignore the warning

```
Warning: Image is too big to fit on screen; displaying at 33%
```

and wait for the next response

```
Click on lower left and upper right corner, then <return>
```

The image window can be scaled according to user preference. Clicking on the lower left and upper right corners defines the dimensions of the image. These changes are registered by pressing *return*. The routine then references the image to the coordinate system and waits for the input of the points we wish to digitize from the image.

```
Click on data points to digitize, then <return>
```

We finish the input by again pressing *return*. The *xy* coordinates of our digitized points are now stored in the variable `data`. We can now use these vector data for other applications.

8.8 Image Enhancement, Correction and Rectification

This section introduces some fundamental tools for image enhancement, correction and rectification. As an example we use an image of varved sediments deposited around 33 kyrs ago in a landslide-dammed lake in the Quebrada de Cafayate of Argentina (25°58.900'S 65°45.676'W) (Trauth et al. 1999, 2003). The diapositive was taken on 1st October 1996 with a film-based single-lens reflex (SLR) camera. A 30-by-20 cm print was made from the slide, which has been scanned using a flatbed scanner and saved as a 394 KB JPEG file. We use this as an example because it demonstrates some problems that we can solve with the help of image enhancement (Fig. 8.6). We then use the image to demonstrate how to measure color-intensity transects for use in time series analysis (Section 8.9).

We can read and decompress the file *varves_original.jpg* by typing

```
clear

I1 = imread('varves_original.jpg');
```

which yields a 24-bit RGB image array `I1` in the MATLAB workspace. Typing

```
whos
```

yields

```
Name         Size                 Bytes  Class    Attributes
I1        1096x1674x3           5504112  uint8
```

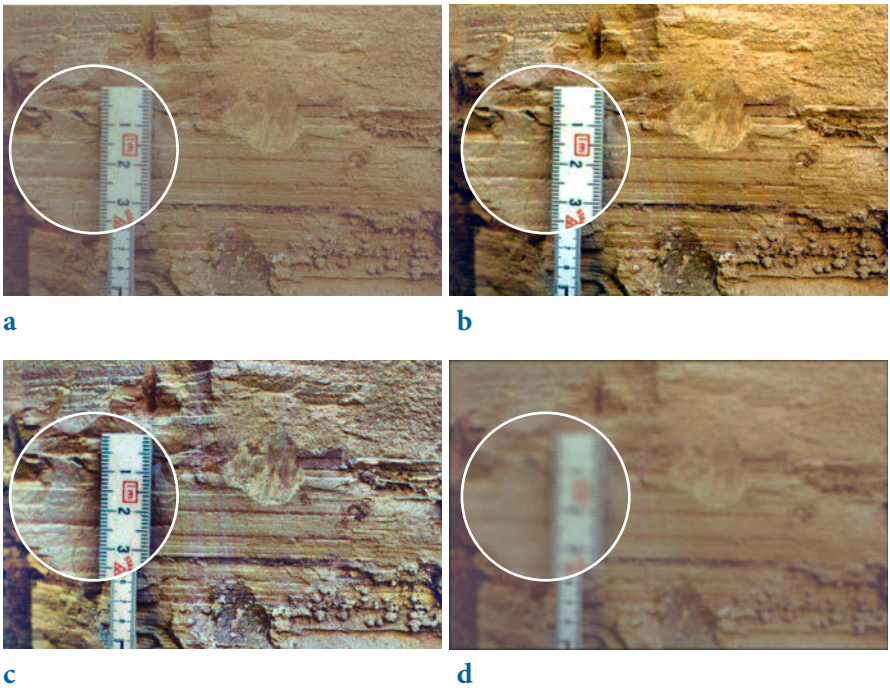


Fig. 8.6 Results of image enhancements; **a** original image, **b** image with intensity values adjusted using `imadjust`, Gamma=1.5, **c** image with contrast enhanced using `adapthisteq`, **d** image after filtering with a 20-by-20 pixel filter with the shape of a Gaussian probability density function with a mean of zero and a standard deviation of 10, using `fspecial` and `imfilter`.

revealing that the image is stored as an `uint8` array of the size 1,096-by-1,674-by-3, i.e., 1,096-by-1,674 arrays for each color (red, green and blue). We can display the image using the command

```
imshow(I1)
```

which opens a new Figure Window showing an RGB composite of the image. As we see, the image has a low level of contrast and very pale colors, and the sediment layers are not exactly horizontal. These are characteristics of the image that we want to improve in the following steps.

First, we adjust the image intensity values or colormap. The function `imadjust(I1,[li; hi],[lo ho])` maps the values of the image `I1` to new values in `I2`, such that values between `li` and `hi` are adjusted to values between `lo` and `ho`. Values below `li` and above `hi` are clipped, i.e., these values are adjusted to `lo` and `ho`, respectively. We can determine the range of the pixel

values using

```
lh = stretchlim(I1)
```

which yields

```
lh =
    0.3255    0.2627    0.2784
    0.7020    0.7216    0.7020
```

indicating that the red color ranges from 0.3255 to 0.7020, green ranges from 0.2627 to 0.7216, and blue ranges from 0.2784 to 0.7020. We can utilize this information to automatically adjust the image with `imadjust` by typing

```
I2 = imadjust(I1,lh,[]);
```

which adjusts the ranges to the full range of [0,1], and then display the result.

```
imshow(I2)
```

We can clearly see the difference between the very pale image `I1` and the more saturated image `I2`. The parameter `gamma` in `imadjust(I1,[li;hi],[lo;ho], gamma)` specifies the shape of the curve describing the relationship between `I1` and `I2`. If `gamma<1` the mapping is weighted toward higher (brighter) output values. If `gamma>1` the mapping is weighted toward lower (darker) output values. The default value of `gamma=1` causes linear mapping of the values in `I1` to new values in `I2`.

```
I3 = imadjust(I1,lh,[],0.5);
I4 = imadjust(I1,lh,[],1.5);

subplot(2,2,1), imshow(I1), title('Original Image')
subplot(2,2,2), imshow(I2), title('Adjusted Image, Gamma=1.0')
subplot(2,2,3), imshow(I3), title('Adjusted Image, Gamma=0.5')
subplot(2,2,4), imshow(I4), title('Adjusted Image, Gamma=1.5')
```

We can use `imhist` to display a histogram showing the distribution of intensity values for the image. Since `imhist` only works for two-dimensional images, we examine the histogram of the red color only.

```
subplot(2,2,1), imhist(I1(:,:,1)), title('Original Image')
subplot(2,2,2), imhist(I2(:,:,1)), title('Adjusted Image, Gamma=1.0')
subplot(2,2,3), imhist(I3(:,:,1)), title('Adjusted Image, Gamma=0.5')
subplot(2,2,4), imhist(I4(:,:,1)), title('Adjusted Image, Gamma=1.5')
```

The result obtained using `imadjust` differs from that obtained using `histeq` (which we used in Section 8.3 to enhance the contrast in the image). The function `histeq(I1,n)` transforms the intensity of image `I1`, returning in

I5 an intensity image with n discrete levels. A roughly equal number of pixels is ascribed to each of the n levels in I5, so that the histogram of I5 is approximately flat. Histogram equalization using histeq has to carried out separately for each color, since histeq only works for two-dimensional images. We use n=50 in our exercise, which is slightly below the default value of n=64.

```
I5(:,:,1) = histeq(I1(:,:,1),50);
I5(:,:,2) = histeq(I1(:,:,2),50);
I5(:,:,3) = histeq(I1(:,:,3),50);

subplot(2,2,1), imshow(I1), title('Original Image')
subplot(2,2,3), imhist(I1(:,:,1)), title('Original Image')
subplot(2,2,2), imshow(I5), title('Enhanced Image')
subplot(2,2,4), imhist(I5(:,:,1)), title('Enhanced Image')
```

The resulting image looks quite disappointing and we therefore use the improved function adapthisteq instead of histeq. This function uses the contrast-limited adaptive histogram equalization (CLAHE) by Zuiderveld (1994). Unlike histeq and imadjust, the algorithm works on small regions (or tiles) of the image, rather than on the entire image. The neighboring tiles are then combined using bilinear interpolation to eliminate edge effects.

```
I6(:,:,1) = adapthisteq(I1(:,:,1));
I6(:,:,2) = adapthisteq(I1(:,:,2));
I6(:,:,3) = adapthisteq(I1(:,:,3));

subplot(2,2,1), imshow(I1), title('Original Image')
subplot(2,2,3), imhist(I1(:,:,1)), title('Original Image')
subplot(2,2,2), imshow(I6), title('Enhanced Image')
subplot(2,2,4), imhist(I6(:,:,1)), title('Enhanced Image')
```

The result looks slightly better than that obtained using histeq. However, all three functions for image enhancement, imadjust, histeq and adapthisteq, provide numerous ways to manipulate the final outcome. The *Image Processing Toolbox – User's Guide* (MathWorks 2014) and the excellent book by Gonzalez and others (2009) provide more detailed introductions to the use of the various parameters available and the corresponding values of the image enhancement functions.

The Image Processing Toolbox also includes numerous functions for 2D filtering of images. Many of the methods we have looked at in Chapter 6 for one-dimensional data also work with two-dimensional data, as we have already seen in Chapter 7 when filtering digital terrain models. The most popular 2D filters for images are Gaussian filters and median filters, as well as filters for image sharpening. Both Gaussian and median filters are used to smooth an image, mostly with the aim of reducing the amount of noise.

In most examples the signal-to-noise ratio is unknown and adaptive filters (similar to those introduced in Section 6.10) are therefore used for noise reduction. A Gaussian filter can be designed using

```
h = fspecial('gaussian',20,10);
I7 = imfilter(I1,h);
```

where `fspecial` creates predefined 2D filters, such as moving average, disk, or Gaussian filters. The Gaussian filter weights `h` are calculated using `fspecial('gaussian',20,10)`, where `20` corresponds the size of a 20-by-20 pixel filter following the shape of a Gaussian probability density function with a standard devation of `10`. Next, we calculate `I8`, which is a median-filtered version of `I1`.

```
I8(:,:,1) = medfilt2(I1(:,:,1),[20 20]);
I8(:,:,2) = medfilt2(I1(:,:,2),[20 20]);
I8(:,:,3) = medfilt2(I1(:,:,3),[20 20]);
```

Since `medfilt2` only works for two-dimensional data, we again apply the filter separately to each color (red, green and blue). The filter output pixels are the medians of the 20-by-20 neighborhoods around the corresponding pixels in the input image.

The third filter example deals with sharpening an image using `imsharpen`.

```
I9 = imsharpen(I1);
```

This function calculates the Gaussian lowpass filtered version of the image that is used as an unsharp mask, i.e., the sharpened version of the image is calculated by subtracting the blurred filtered version from the original image. The function comes with several parameters that control the ability of the Gaussian filter to blur the image and the strength of the sharpening effect, and a threshold specifying the minimum contrast required for a pixel to be considered an edge pixel and sharpened by unsharp masking. Comparing the results of the three filtering exercises with the original image

```
subplot(2,2,1), imshow(I1), title('Original Image')
subplot(2,2,2), imshow(I7), title('Gaussian Filter')
subplot(2,2,3), imshow(I8), title('Median Filter')
subplot(2,2,4), imshow(I9), title('Sharpening Filter')
```

clearly demonstrates the effect of the 2D filters. As an alternative to these time-domain filters, we can also design 2D filters with a specific frequency response, such as the 1D filters described in Section 6.9. Again, the book by Gonzalez and others (2009) provides an overview of 2D frequency-selective filtering for images, including functions used to generate such filters. The authors also demonstrate the use of a 2D Butterworth lowpass filter in image

processing applications.

We next rectify the image, i.e., we correct the image distortion by transforming it to a rectangular coordinate system using a script that is similar to that used for georeferencing satellite images in Section 8.5. This we achieve by defining four points within the image, which are actually at the corners of a rectangular area (which is our reference area). We first define the *upper left*, *lower left*, *upper right*, and *lower right* corners of the reference area, and then press return. Note that it is important to pick the coordinates of the corners in this particular order. In this instance we use the original image `I1`, but we could also use any other enhanced version of the image from the previous exercises. As an example we can click the left side of the ruler at 1.5 cm and at 4.5 cm, where two thin white sediment layers cross the ruler, for use as the upper-left and lower-left corners. We then choose the upper-right and lower-right corners, further to the right of the ruler but also lying on the same two white sediment layers,

```
imshow(I1)
basepoints = ginput
```

and click return which yields

```
basepoints =
  517.0644  508.9059
  511.5396  733.5792
  863.2822  519.9554
  859.5990  739.1040
```

or any similar values. The image and the reference points are then displayed in the same figure window.

```
close all
imshow(I1)
hold on
line(basepoints(:,1),basepoints(:,2),...
       'LineStyle','none',...
       'Marker','+',...
       'MarkerSize',48,...
       'Color','b')
hold off
```

We arbitrarily choose new coordinates for the four reference points, which are now on the corners of a rectangle. To preserve the aspect ratio of the image, we select numbers that are the means of the differences between the x- and y-values of the reference points in `basepoints`.

```
dx = (basepoints(3,1)+basepoints(4,1))/2- ...
     (basepoints(1,1)+basepoints(2,1))/2
```

```
dy = (basepoints(2,2)+basepoints(4,2))/2- ...
     (basepoints(1,2)+basepoints(3,2))/2
```

which yields

```
dx =
  347.1386

dy =
  221.9109
```

We therefore choose

```
inputpoints(1,:) = [0 0];
inputpoints(2,:) = [0 dy];
inputpoints(3,:) = [dx 0];
inputpoints(4,:) = [dx dy];
```

The function `fitgeotrans` now takes the pairs of control points, `inputpoints` and `basepoints`, and uses them to infer a spatial transformation matrix `tform` using the transformation type `projective`.

```
tform = fitgeotrans(inputpoints,basepoints,'projective');
```

We next need to estimate the spatial limits for the output, `XBounds` and `YBounds`, corresponding to the projective transformation `tform`, and a set of spatial limits for the input `xLimitsIn` and `yLimitsIn`.

```
xLimitsIn = 0.5 + [0 size(I1,2)]
yLimitsIn = 0.5 + [0 size(I1,1)]

[XBounds,YBounds] = outputLimits(tform,xLimitsIn,yLimitsIn)
```

Then we use `imref2d` to reference the image to world coordinates.

```
Rout = imref2d(size(I1),XBounds,YBounds)
```

An `imref2d` object `Rout` encapsulates the relationship between the intrinsic coordinates anchored to the rows and columns of the image and the spatial location of the same row and column locations in a world coordinate system. Finally, the projective transformation can be applied to the original RGB composite `I1` in order to obtain a rectified version of the image (`I10`).

```
I10 = imwarp(I1,tform,'OutputView',Rout);
```

We now compare the original image `I1` with the rectified version `I10`.

```
subplot(2,1,1), imshow(I1), title('Original Image')
subplot(2,1,2), imshow(I10), title('Rectified Image')
```

We see that the rectified image has black areas at the corners. We can remove these black areas by cropping the image using `imcrop`.

```
I11 = imcrop(I10);

subplot(2,1,1), imshow(I1), title('Original Image')
subplot(2,1,2), imshow(I11), title('Rectified and Cropped Image')
```

The function `imcrop` creates displays of the image with a resizable rectangular tool that can be interactively positioned and manipulated using the mouse. After manipulating the tool into the desired position, the image is cropped by either double clicking on the tool or choosing *Crop Image* from the tool's context menu. The result of our image enhancement experiment can now be used in the next section to analyze the colors of individual sediment layers.

8.9 Color-Intensity Transects Across Varved Sediments

High-resolution core logging has, since the early 1990s, become popular as an inexpensive tool for investigating the physical and chemical properties of marine and lacustrine sediments. During the early days of nondestructive core logging, magnetic susceptibility and grayscale intensity transects were measured on board research vessels to generate a preliminary stratigraphy of marine cores, since the cyclic recurrence of light and dark layers seemed to reflect glacial-interglacial cycles during the Pleistocene. Paleolimnologists adopted these techniques to analyze annually-layered (varved) lake sediments and to statistically detect short-term variabilities such as the 11 year sunspot cycle, the 3-7 year El Niño cycle, or the 78 year Gleissberg cycle. Modern multi-sensor core loggers are now designed to log a great variety of physical and chemical properties using optical scanners, radiograph imaging, and x-ray fluorescence elemental analyzers.

As an example we explore varved sediments deposited around 33 kyrs ago in a landslide-dammed lake in the Quebrada de Cafayate of Argentina (Trauth et al. 1999, 2003). These lake sediments have been intensively studied for paleoclimate reconstructions since they document episodes of a wetter and more variable climate that eventually fostered mass movements in the NW Argentine Andes during the Late Pleistocene and Holocene. Aside from various sedimentological, geochemical and micropaleontological analyses, the colors of the sediments have been studied as a proxy for rainfall intensities at the time of deposition. Color-intensity transects were analyzed to detect interannual variations in precipitation caused by the El Niño/Southern Oscillation (ENSO, 3–7 year cycles) and the Tropical Atlantic Sea-Surface Temperature Variability (TAV, 10–15 year cycles) using linear and nonlinear

methods of time-series analysis (e.g., Trauth et al. 2000, Marwan et al. 2003).

The El Paso section in the Quebrada de Cafayate contains well-developed annual layers in most parts of the profile (Fig. 8.7). The base of each of these mixed clastic and biogenic varves consists of reddish silt and clay, with a sharp lower boundary. Towards the top of the varves, reddish clay and silt are gradually replaced by light-brown to greenish clay. The change from reddish hues correlates with a slight decrease in grain size. This clastic portion of the varves is capped by a thin layer of pure white diatomite. Diatomites are sediments comprised mainly of porous siliceous skeletons of single-cell algae, i.e. diatoms. This internal structure of the laminae is typical of annual-layered sediments. The recurrence of these layers and the distribution of diatoms, together with the sediment coloration and provenance, all provide additional evidence that rhythmic sedimentation in this region was controlled by a well-defined annual cycle. The provenance of the sediments contained in the varved layers can be traced using index minerals characteristic of the various possible source areas within the catchment. A comparison of the mineral assemblages in the sediments with those of potential source rocks within the catchment area indicates that Fe-rich Tertiary sedimentary rocks

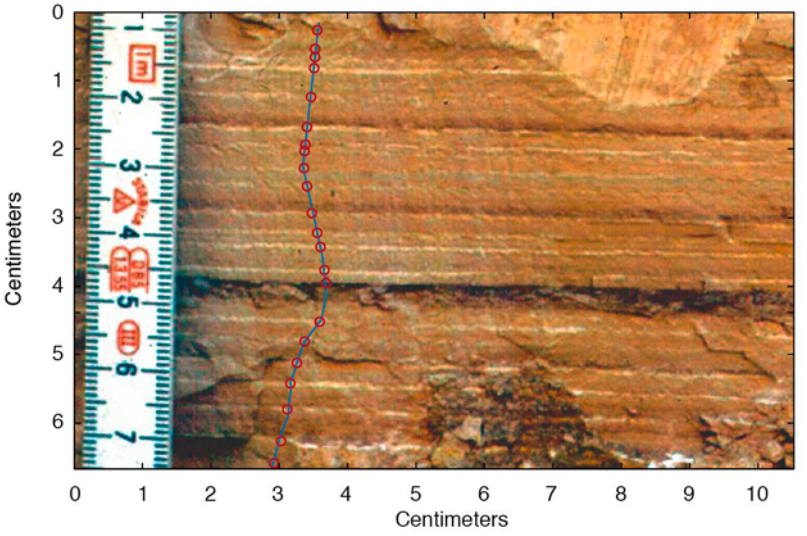


Fig. 8.7 Photograph of varved lake sediments from the Quebrada de Cafayate in the Santa Maria Basin, with cyclic occurrences of intense dark-red coloration reflecting enhanced precipitation and sediment input during ENSO- and TAV-type periodicities (350 cm above the base of the El Paso section). The solid blue line denotes the course of the digitized color-intensity transect. The red circles note the position of the diatomite layers, representing annual layers.

exposed in the Santa Maria Basin were the source of the red-colored basal portion of the varves. In contrast, metamorphic rocks in the mountainous parts of the catchment area were the most likely source of the drab-colored upper part of the varves.

In nearly every second to fifth, and every tenth to fourteenth varve, the varve thickness increases by a factor of 1.5 to 2 and the basal reddish coloration is more intense, suggesting a greater fluvial input of Fe-rich Tertiary material. Exceptionally well-preserved sections containing 70–250 varves were used for more detailed statistical analysis of the observed cyclicities (see Chapter 5). High-quality photographs from these sections were scanned and subjected to standardized color and illumination corrections. Pixel-wide representative red-color intensities were subsequently extracted from transects across the images of these varves. The resolution of these time series was of the order of ten intensity values per varve.

We will now analyze a 22-year package of varved lake sediments from the Quebrada de Cafayate as an example (Fig. 8.6). The photo was taking during a field expedition in the late 1990s using an analog camera. It was then scanned and the contrast levels adjusted to heighten details using standard photo processing software. We import the image from the file *varves_original.tif* as a 24-bit RGB image array and store the data in a variable `I`.

```
clear

I = imread('varves_original');
```

We display the image using `imshow` and turn the axis labeling, tick marks and background back on.

```
imshow(I), axis on
```

The image is scaled to pixel indices or coordinates, so we first need to scale the image to a centimeter scale. While keeping the figure window open we use `ginput` to count the number of pixels per centimeter. The function `ginput` provides a crosshair with which to gather an unlimited number of points forming a polygon, until the *return* key is pressed. We place the crosshair at 1 cm and 6 cm on the scale in the image and click to gather the pixel coordinates of the 5-cm interval.

```
[x,y] = ginput;
```

The image is `size(I,2)=1830` pixels wide and `size(I,1)=1159` pixels high. We convert the width and the height of the image into centimeters using the conversion `5/sqrt((y(2,1)-y(1,1))^2+(x(2,1)-x(1,1))^2)`, where `5` corres-

ponds to the 5 cm interval equivalent to the `sqrt((y(2,1)-y(1,1))^2+(x(2,1)-x(1,1))^2)` pixels gathered using `ginput`.

```
ix = 5 * size(I,2) / sqrt((y(2,1)-y(1,1))^2+(x(2,1)-x(1,1))^2);
iy = 5 * size(I,1) / sqrt((y(2,1)-y(1,1))^2+(x(2,1)-x(1,1))^2);
```

We can now display the image using the new coordinate system where `ix` and `iy` are the width and height of the image, respectively (in centimeters).

```
imshow(I,'XData',[0 ix],'YData',[0 iy]), axis on
xlabel('Centimeters'), ylabel('Centimeters')
```

We now digitize the color-intensity transect from the top of the image to bottom. The function `improfile` determines the RGB pixel values `C` along line segments defined by the coordinates `[CX,CY]`.

```
[CX,CY,C] = improfile;
```

The scaled image and the polygon are displayed in the same figure window. The three color-intensity curves are plotted in a separate window.

```
imshow(I,'XData',[0 ix],'YData',[0 iy]), hold on
plot(CX,CY), hold off

figure
plot(CY,C(:,1),'r',CY,C(:,2),'g',CY,C(:,3),'b')
xlabel('Centimeters'), ylabel('Intensity')
```

The image and the color-intensity profiles are on a centimeter scale. To detect the interannual precipitation variability, as recorded in the color intensity of the sediments, we need to convert the length scale to a time scale. We use the 22 white diatomite layers as time markers to define individual years in the sedimentary history. We again use `ginput` to mark the diatomite layers from top to bottom along the color-intensity transect and store the coordinates of the laminae in the new variable `laminae`.

```
imshow(I,'XData',[0 ix],'YData',[0 iy]), hold on
plot(CX,CY), hold off
laminae = ginput;
```

To inspect the quality of the age model we plot the image, together with the polygon and the marked diatomite layers.

```
imshow(I,'XData',[0 ix],'YData',[0 iy])
hold on
plot(CX,CY)
plot(laminae(:,1),laminae(:,2),'ro')
xlabel('Centimeters'), ylabel('Centimeters')
hold off
```

We define a new variable `newlaminae` that contains the vertical y-component of `laminae` as the first column and the years 1 to 22 (counting backwards in time) as the second colum. The 22 years are equivalent to the length of `laminae`. The function `interp1` is used to interpolate the color-intensity transects over an evenly-spaced time axis stored in the variable `age`.

```
newlaminae(:,1) = laminae(:,2);
newlaminae(:,2) = 1 : length(laminae(:,2));
age = interp1(newlaminae(:,1),newlaminae(:,2),CY);
```

We complete the analysis by plotting the color-intensity curves on both a length and a time scale for comparison (Fig. 8.8).

```
subplot(2,1,1), plot(CY,C(:,1),CY,C(:,2),CY,C(:,3))
xlabel('Centimeters'), ylabel('Intensity'), title('Color vs. Length')
subplot(2,1,2), plot(age,C(:,1),age,C(:,2),age,C(:,3))
xlabel('Years'), ylabel('Intensity'), title('Color vs. Age')
```

The interpolated color-intensity transects can now be further analyzed using the time-series analysis tools. The analysis of a representative red-color intensity transect across 70–250 varves during the project described

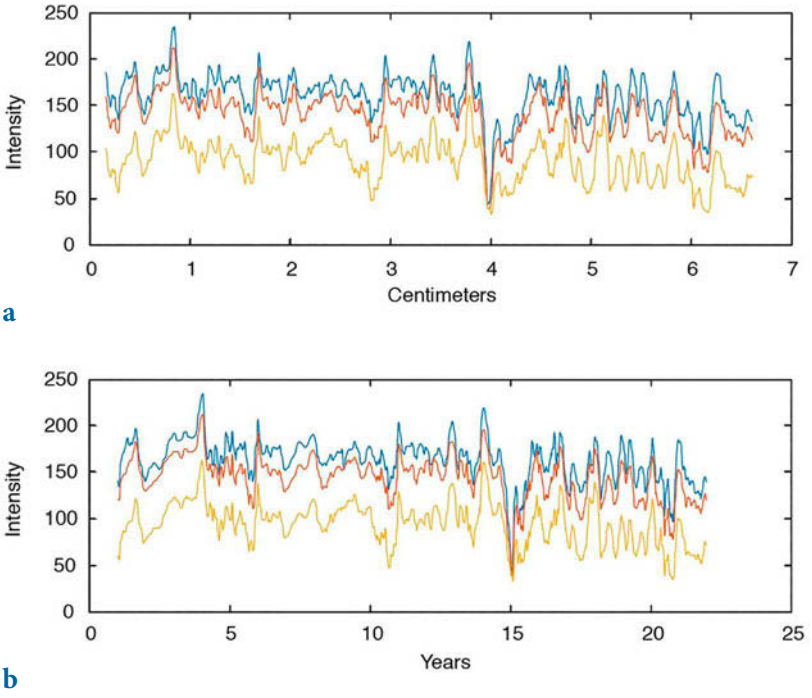


Fig. 8.8 Color-intensity curves (red, green and blue) plotted against **a** depth and **b** age.

above revealed significant peaks at about 13.1, 3.2, 2.2 and 1.0 yrs, suggesting both ENSO and TAV influences in the area at around 33,000 years ago (see Chapter 5 and Fig. 5.1).

8.10 Grain Size Analysis from Microscope Images

Identifying, measuring and counting particles in an image are the classic applications of image analysis. Examples from the geosciences include grain size analysis, counting pollen grains, and determining the mineral composition of rocks from thin sections. For grain size analysis the task is to identify individual particles, measure their sizes, and then count the number of particles per size class. The motivation to use image analysis is the ability to perform automated analyses of large sets of samples in a short period of time and at relatively low costs. Three different approaches are commonly used to identify and count objects in an image: (1) region-based segmentation using the watershed segmentation algorithm, (2) object detection using the Hough transform, and (3) thresholding using color differences to separate objects. Gonzalez, Woods and Eddins (2009) describe these methods in great detail in the 2nd edition of their excellent book, which also provides numerous MATLAB recipes for image processing. The book has a companion webpage at

```
http://www.imageprocessingplace.com/
```

that offers additional support in a number of important areas (including classroom presentations, M-files, and sample images) as well as providing numerous links to other educational resources. We will use two examples to demonstrate the use of image processing for identifying, measuring, and counting particles. In this section we will demonstrate an application of watershed segmentation in grain size analysis and then in Section 8.9 we will introduce thresholding as a method for quantifying charcoal in microscope images. Both applications are implemented in the MATLAB-based RADIUS software developed by Klemens Seelos from the University of Mainz (Seelos and Sirocko 2005). RADIUS is a particle-size measurement technique, based on the evaluation of digital images from thin sections that offers a sub-mm sample resolution and allows sedimentation processes to be studied within the medium silt to coarse sand size range. It is coupled with an automatic pattern recognition system for identifying sedimentation processes within undisturbed samples. The MATLAB code for RADIUS can be downloaded from

```
http://www.particle-analysis.info/
```

The following example for object segmentation illustrates the segmentation, measuring, and counting of objects using the watershed segmentation algorithm (Fig. 8.9). We first read an image of coarse lithic grains of different sizes and store it in the variable `I1`. The size of the image is 284-by-367 pixels and, since the width is 3 cm, the height is 3 cm·284/367=2.32 cm.

```
clear

I1 = imread('grainsize.tif');
ix = 3; iy = 284 * 3 / 367;
imshow(I1,'XData',[0 ix],'YData',[0 iy])
title('Original Image')
```

Here, `ix` and `iy` denote the coordinate axes used to calibrate the image `I1` to a centimeter scale. The true number of objects counted in this image is 236, including three grains that overlap the borders of the image and will therefore be ignored in the following exercise. We reject the color information of the image and convert `I1` to grayscale using the function `rgb2gray`.

```
I2 = rgb2gray(I1);
imshow(I2,'XData',[0 ix],'YData',[0 iy])
title('Grayscale Image')
```

This grayscale image `I2` has a relatively low level of contrast. We therefore use the function `imadjust` to adjust the image intensity values. The function `imadjust` maps the values in the intensity image `I2` to new values in `I3`, such that 1% of the data is saturated at low and high intensities. This increases the contrast in the new image `I3`.

```
I3 = imadjust(I2);
imshow(I3,'XData',[0 ix],'YData',[0 iy])
title('Adjusted Intensity Values')
```

We next determine the background of the `I3` image, which means basically the texture of the black foil on which the grains are located. The function `imopen(im,se)` determines objects in an image `im` with a specific shape `se` (a flat structuring element such as a circular disk) and size (expressed as a specific number of pixels), as defined by the function `strel`. We then produce a background-free image, `I4`.

```
I4 = imopen(I3,strel('disk',1));
imshow(I4,'XData',[0 ix],'YData',[0 iy])
title('No Background')
```

We subtract the background-free image `I4` from the original grayscale image `I3` to observe the background `I5` that has been eliminated.

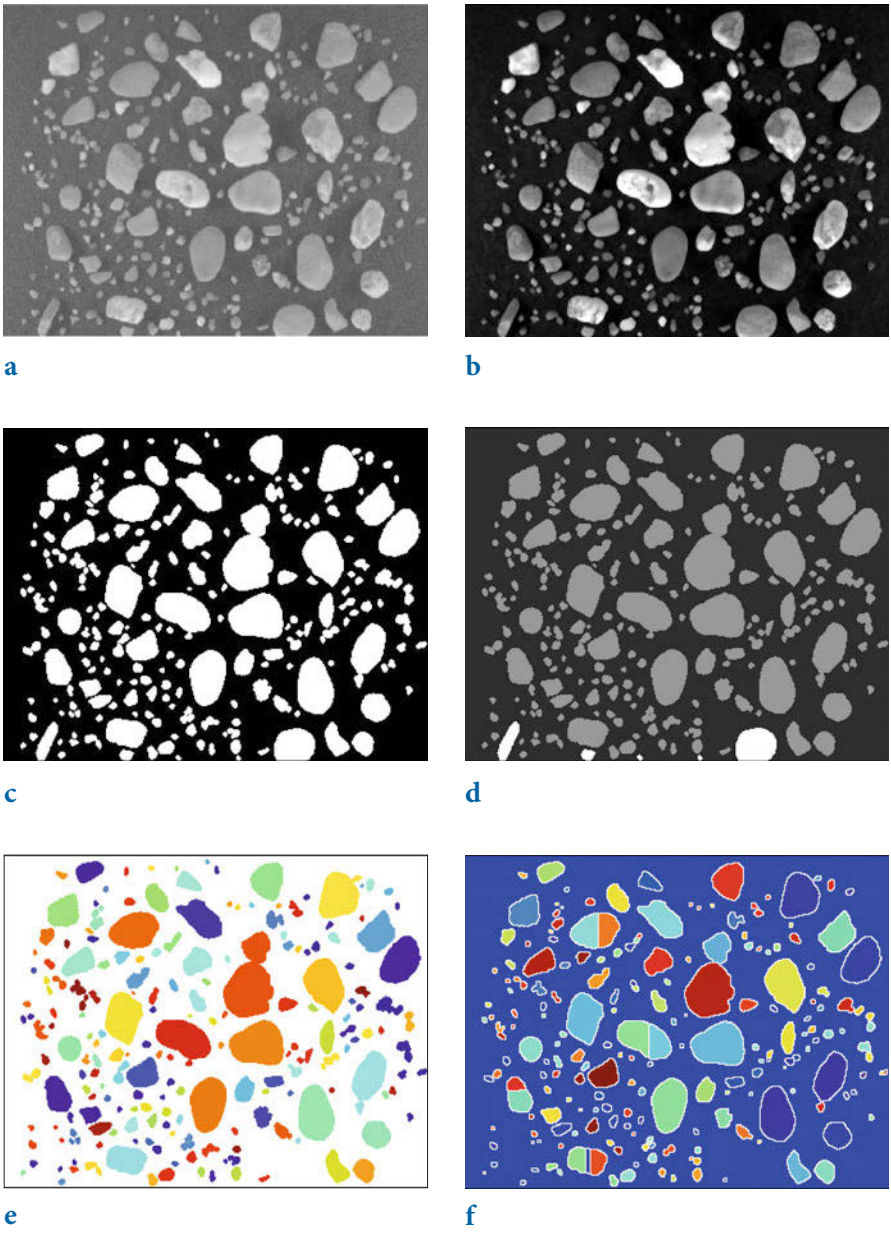


Fig. 8.9 Results from automated grain size analysis of a microscope image; **a** original grayscale image, **b** image after removal of background, **c** image after conversion to binary image, **d** image after eliminating objects overlapping the image border, **e** image with objects detected by tracing the boundaries of connected pixels, and **f** image with objects detected using a watershed segmentation algorithm.

```
I5 = imsubtract(I3,I4);
imshow(I5,'XData',[0 ix],'YData',[0 iy])
title('Background')
```

The function `im2bw` converts the background-free image `I4` to a binary image `I6` by thresholding. If the threshold is 1.0 the image is all black, corresponding to the pixel value of 0. If the threshold is 0.0 the image is all white, corresponding to a pixel value of 1. We manually change the threshold value until we get a reasonable result and find 0.2 to be a suitable threshold.

```
I6 = im2bw(I4,0.2);
imshow(I6,'XData',[0 ix],'YData',[0 iy])
title('Binary Image')
```

We next eliminate objects in `I6` that overlap the image border, since they are actually larger than shown in the image and will result in false estimates. We eliminate these using `imclearborder` and generate image `I7`.

```
I7 = imclearborder(I6);
himage1 = imshow(I6,'XData',[0 ix],'YData',[0 iy]); hold on
set(himage1, 'AlphaData', 0.7);
himage2 = imshow(imsubtract(I6,I7),'XData',[0 ix],'YData',[0 iy]);
set(himage2, 'AlphaData', 0.4);
title('Image Border'), hold off
```

We then trace the boundaries using `bwboundaries` in a binary image where non-zero pixels belong to an object and zero pixels are background. By default, the function also traces the boundaries of holes in the `I7` image. We therefore choose the option `noholes` to suppress the tracing of the holes. Function `label2grb` converts the label matrix `L` resulting from `bwboundaries` to an RGB image. We use the colormap `jet`, the zerocolor `w` for `white`, and the color order `shuffle` (which simply shuffles the colors of `jet` in a pseudorandom manner).

```
[B,L] = bwboundaries(I7,'noholes');
imshow(label2rgb(L,@jet,'w','shuffle'),...
   'XData',[0 ix],'YData',[0 iy])
title('Define Objects')
```

The function `bwlabeln` is used to obtain the number of connected objects found in the binary image. The integer 8 defines the desired connectivity, which can be either 4 or 8 in two-dimensional neighborhoods. The elements of `L` are integer values greater than or equal to 0. The pixels labeled 0 are the background. The pixels labeled 1 make up one object, the pixels labeled 2 make up a second object, and so on.

```
[labeled,numObjects] = bwlabeln(I7,8);
```

```
numObjects
```

In our example the method identified 192 grains, which is significantly less than the 236 grains counted manually, reduced by the three objects that overlap the borders of the image. Visual inspection of the color-coded image generated by `bwboundaries` reveals the reason for the underestimated number of grains. Two large grains in the middle of the image have been observed as being connected, giving a single, very large grain in the final result. Reducing the disk size with `strel` from `disk=1` to `disk=5` can help separate connected grains. Larger disks, on the other hand, reduce the number of grains because smaller grains are lost by filtering. We now determine the areas of each of the grains.

```
graindata = regionprops(labeled,'basic');
grainareas= [graindata(:).Area];
objectareas = 3^2 * grainareas * 367^(-2);
```

We then find the maximum, minimum and mean areas for all grains in the image, in cm^2.

```
max_area = max(objectareas)
min_area = min(objectareas)
mean_area = mean(objectareas)
```

The connected grain in the middle of the image has a size of 0.16 cm^2, which represents the maximum size of all grains in the image. Finally, we plot the histogram of all the grain areas.

```
clf
e = 0 : 0.0005 : 0.15;
histogram(objectareas,e)
xlabel('Grain Size in Millimeters^2')
ylabel('Number of Grains')
axis([0 0.1 0 30])
```

Several methods exist that partly overcome the artifact of connected grains in grain size analyses. The most popular technique for region-based segmentation is the watershed segmentation algorithm. Watersheds in geomorphology are ridges that divide areas contributing to the hydrological budget of adjacent catchments (see Section 7.10). Watershed segmentation applies to grayscale images the same methods used to separate catchments in digital elevation models. In this application, the grayscale values are interpreted as elevations in a digital elevation model, where the watershed then separates the two objects of interest.

The criterion commonly used to identify pixels that belong to one object is the nearest-neighbor distance. We use the distance transform performed

by `bwdist`, which assigns to each pixel a number that is the distance between a pixel and the nearest non-zero pixel in `I7`. In an image in which objects are identified by pixel values of zero and the background by pixel values of one, the distance transform has zeros in the background areas and non-zero values that increase progressively with increasing distances from the edges of the objects. In our example however, the objects have pixel values of one and the background has pixels with with values of zero. We therefore have to apply `bwdist` to the complement of the binary image `I7` instead of to the image itself.

```
D = bwdist(~I7,'cityblock');
```

The function `bwdist` provides several methods for computing the nearest-neighbor distances, including *Euclidean distances*, *cityblock distances*, *chessboard distances* and *quasi-Euclidean distances*. We choose the *cityblock* option in this particular example, but other methods might be more appropriate for separating objects in other images. The distance matrix now contains positive non-zero values in the object pixels and zeros elsewhere. We then complement the distance transform, and ascribe a value of `-Inf` to each pixel that does not belong to an object.

```
D = -D;
D(~I7) = -Inf;
```

We compute the watershed transform for the distance matrix, and display the resulting label matrix.

```
L2 = watershed(D);
imshow(label2rgb(L2,@jet,'w','shuffle'),...
   'XData',[0 ix],'YData',[0 iy])
title('Watershed Segmentation')
```

After having displayed the results from watershed segmentation, we determine the number of pixels for each object using the recipe from above, except for index `i` running from `2` to `max(objects)` since the value `1` denotes the background and `0` denotes the boundaries of the objects. The first true object is therefore marked by the value of `2`.

```
objects = sortrows(L2(:),1);
max(objects)
clear objectsizes
for i = 2 : max(objects)
    clear individualobject
    individualobject = objects(objects == i);
    objectsizes(i) = length(individualobject);
end
objectsizes = objectsizes';
```

```
objectsizes = sortrows(objectsizes,1);
objectsizes = objectsizes(objectsizes~=0);
```

We have now recognized 205 objects, i.e., more objects than were identified in the previous exercise without watershed segmentation. Visual inspection of the result, however, reveals some oversegmentation (due to noise or other irregularities in the image) in which larger grains are divided into smaller pieces. On the other hand, very small grains have been eliminated by filtering the image with the morphological structuring element `strel`. We scale the object sizes. The area of one pixel is $(3\ \text{cm}/367)^2$.

```
objectareas = 3^2 * objectsizes * 367^(-2);
```

We now determine the areas for each of the grains. We again find the maximum, minimum and mean areas for all grains in the image, in cm^2.

```
max_area = max(objectareas)
min_area = min(objectareas)
mean_area = mean(objectareas)
```

The largest grain in the center of the image has a size of 0.09 cm^2, which represents the maximum size of all grains in the image. Finally, we plot the histogram of all the grain areas.

```
clf
e = 0 : 0.0005 : 0.15;
histogram(objectareas,e)
xlabel('Grain Size in Millimeters^2'),...
   ylabel('Number of Grains')
axis([0 0.1 0 70])
```

As a check of the final result we digitize the outline of one of the larger grains and store the polygon in the variable `data`.

```
figure
imshow(I1,'XData',[0 ix],'YData',[0 iy])
data = ginput;
```

We close the polygon by copying the first row of coordinates to the end of the array. We then display the polygon on the original image.

```
data(end+1,:) = data(1,:)

imshow(I1,'XData',[0 ix],'YData',[0 iy]), hold on
plot(data(:,1),data(:,2)), hold off
```

The function `polyarea` yields the area of the large grain.

```
polyarea(data(:,1),data(:,2))
```

```
ans =
    0.0951
```

The calculated area corresponds approximately to the result from the grain size analysis. If oversegmentation is a major problem when using segmentation to count objects in an image, the reader is referred to the book by Gonzalez, Woods and Eddins (2009) that describes marker-controlled watershed segmentation as an alternative method to avoid oversegmentation.

8.11 Quantifying Charcoal in Microscope Images

Quantifying the composition of substances in geosciences, such as the mineral composition of a rock in thin sections, or the amount of charcoal in sieved sediment samples, is facilitated by the use of image processing methods. Thresholding provides a simple solution to segmenting objects within an image that have different coloration or grayscale values. During the thresholding process, pixels with an intensity value greater than a threshold value are marked as object pixels (e.g., pixels representing charcoal in an image) and the rest as background pixels (e.g., all other substances). The threshold value is usually defined manually through visual inspection of the image histogram, but numerous automated algorithms are also available.

As an example we analyze an image of a sieved lake-sediment sample from Lake Nakuru, Kenya (Fig. 8.10). The image shows abundant light-gray oval ostracod shells and some mineral grains, as well as gray plant remains and black charcoal fragments. We use thresholding to separate the dark charcoal particles and count the pixels of these particles after segmentation. After having determined the number of pixels for all objects distinguished from the background by thresholding, we use a lower threshhold value to determine the ratio of the number of pixels representing charcoal to the number of pixels representing all particles in the sample, i.e., to determine the percentage of charcoal in the sample.

We read the image of size 1500-by-1500 pixels and assume that the width and the height of the square image are both one centimeter.

```
clear

I1 = imread('lakesediment.jpg');
ix = 1; iy = 1;
imshow(I1,'XData',[0 ix],'YData',[0 iy]), axis on
xlabel('Centimeter'), ylabel('Centimeter')
title('Original Image')
```

The RGB color image is then converted to a grayscale image using the function `rgb2gray`.

```
I2 = rgb2gray(I1);
imshow(I2,'XData',[0 ix],'YData',[0 iy]), axis on
xlabel('Centimeters'), ylabel('Centimeters')
title('Grayscale')
```

Since the image contrast is relatively low, we use the function `imadjust` to adjust the image intensity values. The function `imadjust` maps the values in the intensity image `I1` to new values in `I2`, such that 1% of the data is saturated at low and high intensities of `I2`. This increases the contrast in the new image `I2`.

```
I3 = imadjust(I2);
imshow(I3,'XData',[0 ix],'YData',[0 iy]), axis on
xlabel('Centimeters'), ylabel('Centimeters')
title('Better Contrast')
```

We next determine the background of the lithic grains, which basically means the texture of the black foil on which the grains are located. The function `imopen(im,se)` determines objects in an image `im` below a certain pixel size and a flat structuring element `se`, such as a disk with a radius of 5 pixels generated by the function `strel`. The variable `I4` is the background-free image resulting from this operation.

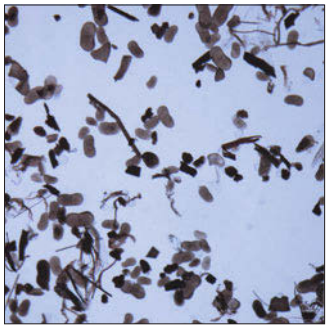
```
I4 = imopen(I3,strel('disk',5));
imshow(I4,'XData',[0 ix],'YData',[0 iy]), axis on
xlabel('Centimeters'), ylabel('Centimeters')
title('W/O Background')
```

We subtract the background-free image `I4` from the original grayscale image `I3` to observe the background `I5` that has been eliminated.

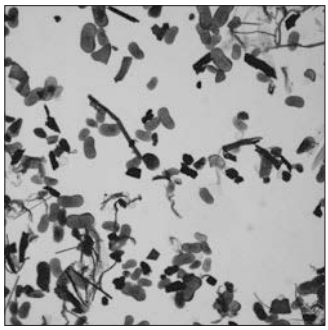
```
I5 = imsubtract(I3,I4);
imshow(I5,'XData',[0 ix],'YData',[0 iy]), axis on
xlabel('Centimeters'), ylabel('Centimeters')
title('Background')
```

The function `im2bw` converts the `I4` image to a binary image (`I6`) by thresholding. If the threshold is 1.0 the image is all black, corresponding to a pixel value of 0. If the threshold is 0.0 the image is all white, corresponding to a pixel value of 1. We manually change the threshold value until we get a reasonable result. In our example a threshold of 0.03 gives good results for identifying charcoal fragments.

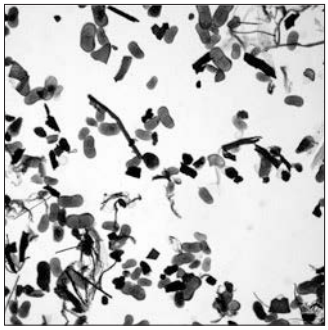
```
I6 = im2bw(I4,0.03);
imshow(I6,'XData',[0 ix],'YData',[0 iy]), axis on
xlabel('Centimeters'), ylabel('Centimeters')
title('Only Charcoal')
```



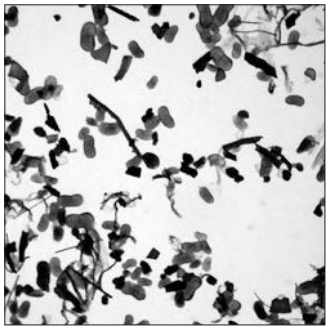
a



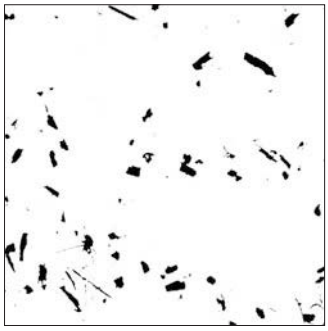
b



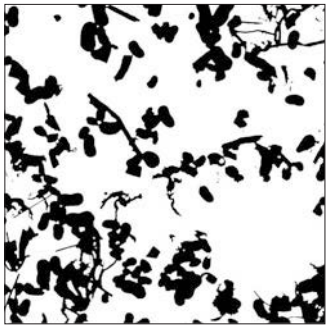
c



d



e



f

Fig. 8.10 Display of results from automatic quantification of charcoal in a microscope image; **a** original color image, **b** grayscale image, **c** image after enhancement of contrast, **d** image after removal of background, **e** image after thresholding to separate charcoal particles, and **f** image after thresholding to separate all objects.

Since we know the size of a pixel we can now simply count the number of pixels to estimate the total amount of charcoal in the image. Finally, we compute the area of all objects, including charcoal.

```
I7 = im2bw(I4,0.6);
imshow(I7,'XData',[0 ix],'YData',[0 iy]), axis on
xlabel('Centimeters'), ylabel('Centimeters')
title('All Objects')
```

We are not interested in the absolute areas of charcoal in the image but in the percentage of charcoal in the sample.

```
100*sum(sum(I6==0))/sum(sum(I7==0))

ans =
   13.4063
```

The result suggests that approximately 13% of the sieved sample is charcoal. As a next step, we could quantify the other components in the sample, such as ostracods or mineral grains, by choosing different threshold values.

8.12 Shape-Based Object Detection in Images

The counting of objects within images on the basis of their shapes is a very time-consuming task. Examples of where this is carried out for round objects include the counting of planktonic foraminifera shells to infer past sea-surface temperatures, of diatom frustules to infer past chemical composition of lake water, and of pollen grains to determine assemblages that can be used to reconstruct regional air temperature and precipitation. Linear objects that are determined include faults in aerial photos and satellite images (to derive the present-day stress field of an area) and annual layers (varves) in thin sections (to establish an annually-resolved sedimentary history).

The Hough transform, named after the related 1962 patent of Paul VC Hough, is a popular technique with which to detect objects within images, based on their shapes. The Hough transform was orginally used to detect linear features, but soon after being patented it was generalized to identify objects of any shape (Duda and Hart 1972, Ballard 1981). The book by Gonzalez and others (2009) contains a comprehensive introduction to the Hough transform and its applications for detecting objects within images. According to their introduction to the method, the Hough transform is performed in two steps. In the first step an edge detector is used to extract edge features, such as distinct sediment layers or the outlines of pollen grains, from an image (Fig. 8.13). In the second step lines (or objects of any other shape) that trace these edge features are identified. The Image Processing

Toolbox (MathWorks 2014) contains functions that use the Hough transform to detect lines or circular objects.

The classic Hough transform is used to detect lines in images. After applying an edge detector of any kind we end up with a binary image that has black pixels on the edges and white pixels in between. We next describe the lines through a given black pixel by the Euklidean distance ρ between the line and the origin, and by the angle θ of the vector from the origin to the closest point on the line (Fig. 8.11 a):

$$\rho = x\cos\theta + y\sin\theta$$

The family of all lines passing through this particular pixel (x_i,y_i) of an edge feature are displayed as a sinusoidal curve in the (θ,ρ) parameter space (Fig. 8.11 b). The intersection point (θ',ρ') of two such sinusoidal curves corresponds to the line that passes through two different pixels, (x_1,y_1) and (x_2,y_2), of an edge feature. Next, we search for n points (x_i,y_i) in the Hough transform where many lines intersect, since these are points defining the line tracing an edge feature. Detecting circles instead of lines works in a similar manner, using the coordinates of the center of the circle and its radius instead of ρ and θ.

For our a first example we use these functions to detect the thin layers of pure white diatomite within varved sediments exposed in the Quebrada de Cafayate of Argentina, which have already been used as examples in previous sections (Trauth et al. 1999, 2003) (Fig. 8.12). The quality of the image is not perfect, which is why we can not expect optimal results. We first read the cropped version of the laminated sediment from Section 8.8 and store it in the variable `I1`. The size of the image is 1,047-by-1,691 pixels, consisting of three colors (red, green and blue).

```
clear

I1 = imread('varves_cropped.tif');
imshow(I1,'InitialMagnification',30)
```

We reject the color information of the image and convert `I1` to grayscale using the function `rgb2gray`.

```
I2 = rgb2gray(I1);
imshow(I2,'InitialMagnification',30)
```

We then use `adapthisteq` to perform a contrast-limited adaptive histogram equalization (CLAHE), in order to enhance the contrast in the image (Zuiderveld 1994).

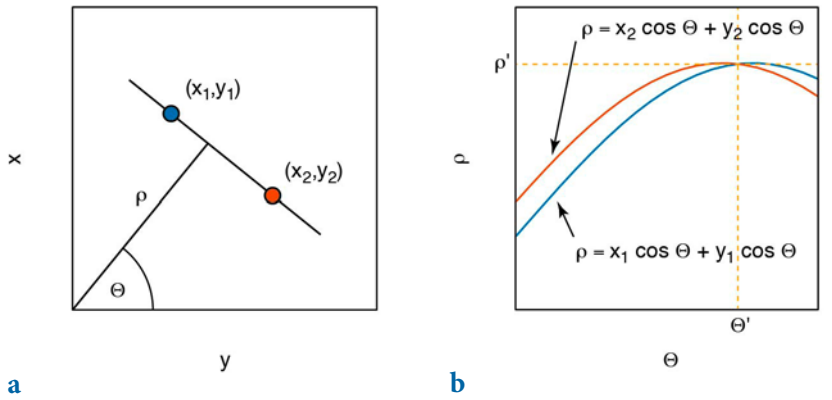


Fig. 8.11 The concept of the Hough transform: **a** parametrization of lines in the *xy*-plane, and **b** sinusoidal curves in the $p\theta$ parameter space, with the point of intersection corresponding to the line that passes through two different pixels of an edge feature (modified from Gonzales et al. 2009).

```
I3 = adapthisteq(I2,'ClipLimit',0.1,'Distribution','Rayleigh');
imshow(I3,'InitialMagnification',30)
```

Here, `ClipLimit` limits the contrast enhancement using a real scalar from 0 to 1, with higher numbers resulting in greater contrast; the default value is 0.01. The `Distribution` parameter defines the desired histogram shape for the tiles by specifying a distribution type, such as `Uniform`, `Rayleigh` and `Exponential`. Using a `ClipLimit` of 0.1 and a `Rayleigh` distribution yields good results. Using the function `im2bw` then converts the `I3` image to a binary image (`I4`) by thresholding. If the threshold is 1.0 the image is all black, corresponding to the pixel value of 0. If the threshold is 0.0 the image is all white, corresponding to a pixel value of 1. We manually change the threshold value until we get a reasonable result and find 0.55 to be a suitable threshold.

```
I4 = im2bw(I3, 0.55);
imshow(I4,'InitialMagnification',30)
```

The function `hough` implements the Hough transform, `houghpeaks` finds the high-count accumulator cells in the Hough transform, and `houghlines` extracts lines in the original image, based on the other two functions. We determine the n=15 lines corresponding to the first 15 maxima of the Hough transform and store fifteen of the lines (lines 1 to 5, 6 to 10, and 11 to 15) in three separate variables `lines1`, `lines2` and `lines3`.

```
[H,theta,rho] = hough(I4);
peaks  = houghpeaks(H,15);
```

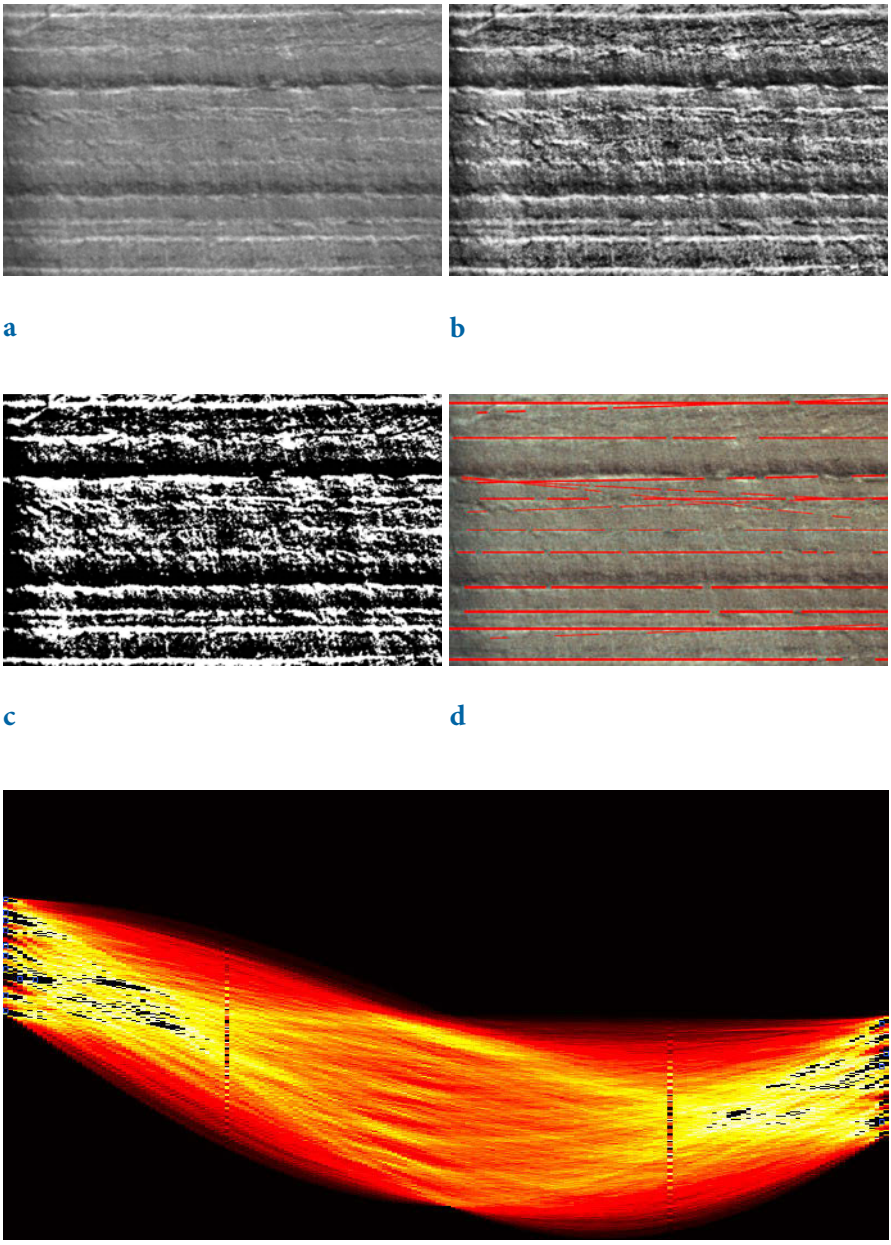


Fig. 8.12 Automated detection of thin layers of pure white diatomite within varved sediments exposed in the Quebrada de Cafayate of Argentina, using `houghlines` (Trauth et al. 1999, 2003); **a** grayscale image, **b** enhanced image, **c** binary image, **d** image with diatomite layers marked by red lines, and **e** Hough transform of the image.

```
lines1 = houghlines(I4,theta,rho,peaks(1:5,:));
lines2 = houghlines(I4,theta,rho,peaks(6:10,:));
lines3 = houghlines(I4,theta,rho,peaks(11:15,:));
```

We then display the Hough transform and mark the 15 maxima of the Hough transform with blue squares.

```
imshow(imadjust(mat2gray(H)),[], ...
    'XData',theta, ...
    'YData',rho, ...
    'InitialMagnification','fit')
colormap(hot), axis square, hold on
plot(theta(peaks(:,2)),rho(peaks(:,1)), ...
    'LineStyle','none', ...
    'Marker','s', ...
    'Color','b')
xlabel('\theta')
ylabel('\rho')
title('Hough Transform')
```

The variables `lines1`, `lines2` and `lines3` can now be used to display the lines on the image, with the line thickness decreasing from `lines3` to `lines1` depending on the rank of the lines in the Hough transform.

```
imshow(I1,'InitialMagnification',30), hold on
for k = 1:length(lines1)
xy = [lines1(k).point1; lines1(k).point2];
plot(xy(:,1),xy(:,2),'LineWidth',3,'Color',[1 0 0]);
end
hold on
for k = 1:length(lines2)
xy = [lines2(k).point1; lines2(k).point2];
plot(xy(:,1),xy(:,2),'LineWidth',2,'Color',[1 0 0]);
end
for k = 1:length(lines3)
xy = [lines3(k).point1; lines3(k).point2];
plot(xy(:,1),xy(:,2),'LineWidth',1,'Color',[1 0 0]);
end
```

The result shows that the clearly recognizable white layers are well detected whereas the less pronounced layers are not identified. The method also mistakenly marks non-existing lines on the image because of the low quality of the image. Using a better quality image and carefully adjusting the parameters used with the Hough transform will yield better results.

In the second example the Hough transform is used to automatically count pollen grains in a microscope image of Argentine honey (Fig. 8.13). The quality of the image is again not perfect, which is why we can not expect optimum results. In particular, the image of three-dimensional objects was taken with a very large magnification, so it is slightly blurred. We first read the pollen image and store it in the variable `I1`. The size of the image is 968-

by-1,060 pixels of three colors (red, green and blue). Since the image is relatively large, we reduce the image size by a factor of two.

```
clear

I1 = imread('pollen.jpg');
I1 = I1(1:2:end,1:2:end,:);
imshow(I1,'InitialMagnification',100)
```

We reject the color information of the image and use the red color only.

```
I2 = I1(:,:,1);
imshow(I2,'InitialMagnification',100)
```

Next, we use `adapthisteq` to perform a contrast-limited adaptive histogram equalization (CLAHE) in order to enhance the contrast in the image (Zuiderveld 1994).

```
I3 = adapthisteq(I2);
imshow(I3,'InitialMagnification',100)
```

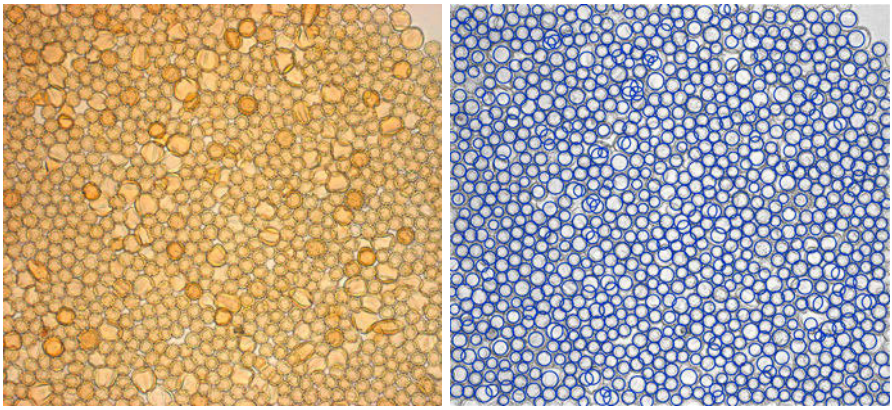
The function `imfindcircles` implements the Hough transform and extracts circles from the original image.

```
[centers,radii] = imfindcircles(I3,[12 20],...
    'Method','TwoStage',...
    'ObjectPolarity','Bright',...
    'Sensitivity',0.92,...
    'EdgeThreshold',0.20);
num = length(centers);
nstr = ['Number of Pollen Grains: ',num2str(num)];
```

Herein we use the `TwoStage` detection method for a two-stage circular Hough transform, following the procedure described by Yuen et al. (1990) and Davies (2005). The object polarity is set to `bright` as we are looking for bright rather than dark objects in our image. The sensitivity of `0.92` and the edge threshold of `0.20` are found by trial and error. We then use `viscircles` to display the circles on the grayscale image `I2`

```
imshow(I2,'InitialMagnification',100)
viscircles(centers, radii,'EdgeColor','b')
title(nstr)
```

using the output `centers` and `radii` from `imfindcircles`. The edge color of the circles in the graphics is set to `b` for blue. The result shows that we have counted 884 pollen grains with the method. The algorithm identifies the majority of the objects, but some are not recognized and some of the larger objects are mistakenly identified as two or more pollen grains. Using a better



a **b**

Fig. 8.13 Automated detection of pollen grains (mostly *Asteraceae* and less abundant *Caesalpiniaceae* and *Lamiaceae* pollen) in a microscope image of Argentine honey, using `imfindcircles` (original image courtesy K Schittek and F Flores); **a** original RGB image, **b** pollen grains detected using the Hough transform.

quality image and carefully adjusting the parameters used with the Hough transform will yield better results. Plotting the histogram of the pollen radii using `histogram`

```
histogram(radii)
```

reveals that most of the grains have a radius of around 15 pixels.

Recommended Reading

Abrams M, Hook S (2002) ASTER User Handbook – Version 2. Jet Propulsion Laboratory and EROS Data Center, Sioux Falls

Ballard DH (1981) Generalizing the Houghtransform to detectarbitraryshapes. Pattern Recognition 13:111–122

Barry P (2001) EO-1/Hyperion Science Data User's Guide. TRW Space, Defense & Information Systems, Redondo Beach, CA

Beck R (2003) EO-1 User Guide. USGS Earth Resources Observation Systems Data Center (EDC), Sioux Falls, SD

Campbell JB (2002) Introduction to Remote Sensing. Taylor & Francis, London

Davies ER (2005) Machine Vision: Theory, Algorithms, Practicalities – 3rd Edition. Morgan Kauffman Publishers, Burlington MA

Duda RO, Hart PE (1972) Use of the Hough transform to Detect Lines and Curves in Pictures. Communications of the ACM 15:11–15

Francus P (2005) Image Analysis, Sediments and Paleoenvironments – Developments in Paleoenvironmental Research. Springer, Berlin Heidelberg New York

Gonzalez RC, Woods RE, Eddins SL (2009) Digital Image Processing Using MATLAB – 2nd Edition. Gatesmark Publishing, LLC

Hough PVC (1962) Method and Means for Recognizing Complex Patterns. US Patent No. 3069654

Irons J, Riebeek H, Loveland T (2011) Landsat Data Continuity Mission – Continuously Observing Your World. NASA and USGS (available online)

Jensen JR (2013) Remote Sensing of the Environment: Pearson New International Edition. Pearson, London

Lein JK (2012) Environmental Sensing – Analytical Techniques for Earth Observation. Springer, Berlin Heidelberg New York

Mandl D, P Cruz, S Frye, Howard, J (2002) A NASA/USGS Collaboration to Transform Earth Observing-1 Into a Commercially Viable Mission to Maximize Technology Infusion. SpaceOps 2002, Houston, TX, October 9–12, 2002

Marwan N, Trauth MH, Vuille M, Kurths J (2003) Nonlinear time-series analysis on present-day and Pleistocene precipitation data from the NW Argentine Andes. Climate Dynamics 21:317–326

MathWorks (2014) Image Processing Toolbox – User's Guide. The MathWorks, Inc., Natick, MA

Ochs B., Hair D, Irons J, Loveland T (2009) Landsat Data Continuity Mission, NASA and USGS (available online)

Richards JA (2013) Remote Sensing Digital Image Analysis – 4th Edition. Springer, Berlin Heidelberg New York

Seelos K, Sirocko F (2005) RADIUS – Rapid Particle Analysis of digital images by ultra-high-resolution scanning of thin sections. Sedimentology 52:669–681

Trauth MH, Bookhagen B, Marwan N, Strecker MR (2003) Multiple landslide clusters record Quaternary climate changes in the NW Argentine Andes. Palaeogeography Palaeoclimatology Palaeoecology 194:109–121

Trauth MH, Alonso RA, Haselton KR, Hermanns RL, Strecker MR (2000) Climate change and mass movements in the northwest Argentine Andes. Earth and Planetary Science Letters 179:243–256

Trauth MH, Strecker MR (1999) Formation of landslide-dammed lakes during a wet period between 40,000–25,000 yr B.P. in northwestern Argentina. Palaeogeography Palaeoclimatology Palaeoecology 153:277–287

Yuen HK, Princen J, Illingworth J, Kittler J (1990) Comparative study of Hough transform methods for circle finding. Image and Vision Computing 8:71–77

Zuiderveld K (1994) Contrast Limited Adaptive Histograph Equalization. Graphic Gems IV. San Diego: Academic Press Professional, 474–485