7 Processing and Displaying Images in Earth Sciences

7.1 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. Images are generally presented as raster data, i.e., as a 2D array of color intensities. Images are everywhere in geosciences. Field geologists use aerial photographs and satellite images to identify lithologic units, tectonic structures, landslides and other features within a study area. Geomorphologists use such images for the analysis of 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 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 a computer are explained in Section 7.2. The main tools for importing, manipulating and exporting image data are presented in Section 7.3. This information is then used to process and to georeference satellite images (Sections 7.4 and 7.5). On-screen digitization techniques are discussed in Section 7.6. 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 the earth sciences.

7.2 Storing Images on a Computer

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

M. H. Trauth and E. Sillmann, *MATLAB® and Design Recipes for Earth Sciences*,
DOI: 10.1007/978-3-642-32544-1_7, © Springer-Verlag Berlin Heidelberg 2013

```
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 6.2).

In contrast, *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 a 3D plot. The x and y 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 4). Alternatively, 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, blue (RGB) color. An example is the above array that 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, one and zero (Chapter 4). 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. Hence, each element of the 2D array or pixel 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. 8 bits 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 give four numbers, three bits show 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 one-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, the 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 (first bit), the exponent (bits 2 to 12) and roughly 16 significant decimal digits in the range of approximately 10^{-308} to 10^{+308} (bits 13 to 64). However, MATLAB also works with other data types, such as 1-bit, 8-bit and 24-bit raster data, to save memory.

The memory required for storing a raster image depends on the data type and the image's dimension. The dimension 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 729×713 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 $729 \times 713 \times 8 = 4{,}158{,}216$ 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 its length, height, and 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 × 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.

7.3 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 as *unconform_image_vs1_original.jpg*. The command

```
clear

unconform1 = imread('unconform_image_vs1_original.jpg');
```

reads and decompresses the JPEG file, imports the data as a 24-bit RGB image array and stores the data in a variable `unconform1`. The command

```
whos
```

shows how the RGB array is stored in the workspace:

```
Name                Size                    Bytes  Class    Attributes
unconform1        729x713x3               1559331  uint8
```

The details indicate that the image is stored as a 729 × 713 × 3 array representing a 729 × 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:

```
unconform1(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
```

Next, we can view the image using the command

```
imshow(unconform1)
```

which opens a new Figure Window showing an RGB composite of the image.

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):

```
unconform2 = rgb2gray(unconform1);
```

The new workspace listing now reads

```
Name                Size                    Bytes  Class    Attributes
ans                 6x6                        36  uint8
unconform1        729x713x3               1559331  uint8
unconform2        729x713                  519777  uint8
```

in which the difference between the 24-bit RGB and the 8-bit grayscale arrays can be observed. The command

```
imshow(unconform2)
```

displays the result. It is easy to see the difference between the two images in

separate Figure Windows (Fig 7.1). Let us now process the grayscale image. First, we compute a histogram of the distribution of intensity values.

```
imhist(unconform2)
```

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

```
unconform3 = histeq(unconform2);
```

Fig. 7.1 Grayscale image. After converting the RGB image stored in a 729 × 713 × 3 array into a grayscale image stored in a 729 × 713 array, the result is displayed using `imshow`. Original image courtesy of NASA/GSFC/METI/ERSDAC/JAROS and U.S./Japan ASTER Science Team.

We can view the difference again using

```
imshow(unconform3)
```

and save the results in a new file.

```
imwrite(unconform3,'unconform_image_vs2_matlab.jpg')
```

We can read the header of the new file by typing

```
imfinfo('unconform_image_vs2_matlab.jpg')
```

which yields

```
           Filename: 'unconform_image_vs2_matlab.jpg'
        FileModDate: '26-Oct-2011 22:54:36'
           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

```
[x,map] = rgb2ind(unconform1,16);
imshow(unconform1), figure, imshow(x,map)
```

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.

7.4 Processing and Printing Satellite Images

In the previous section, we used a processed ASTER image that we downloaded from the ASTER web page. The original ASTER raw data contain a lot more information and higher resolution than the free-of-charge image stored in *unconform.jpg*. The ASTER instrument produces two types of data, Level-1A and Level-1B. Whereas the L1A data are reconstructed, unprocessed instrument data, L1B data are radiometrically and geometrically corrected. Each ASTER data set contains 15 data arrays representing the intensity values from 15 spectral bands (see the ASTER-web page for more detailed information) and various other items of information such as location, date and time. The raw satellite data can be purchased from the USGS online store:

```
https://wist.echo.nasa.gov/wist-bin/api/ims.cgi
```

On this webpage we can select a discipline/topic (e.g., *Land: ASTER*), and then choose from the list of data sets (e.g., *DEM, Level 1A* or *1B* data), define the search area, and click *Start Search*. The system now needs a few minutes to list all relevant data sets. A list of data sets, including various types of additional information (cloud coverage, exposure date, latitude and longitude), can be obtained by clicking on *List Data Granules*. Furthermore, a low resolution preview can be accessed by selecting *Image*. Having purchased a particular data set, the raw image can be downloaded using a temporary FTP-access. As an example, we process an image from an area in Kenya showing Lake Naivasha. The data are stored in two files

```
naivasha_data.hdf
naivasha_data.hdf.met
```

The first file, which has a size of 111 MB, contains the actual raw data, whereas the second file, which has a size of 100 KB, contains the header, together with all sorts of information about the data. We save both files in our working directory. 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('naivasha_data.hdf')
```

This command opens a GUI that allows us to browse the content of the HDF-file *naivasha_data.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. An image such as that used in the previous section is typically achieved by computing an RGB composite from the *vnir_Band3n*, *2* and *1* in the data file. We first read the data, as follows:

```
clear

I1 = hdfread('naivasha_data.hdf','VNIR_Band3N',...
   'Fields','ImageData');
I2 = hdfread('naivasha_data.hdf','VNIR_Band2',...
   'Fields','ImageData');
I3 = hdfread('naivasha_data.hdf','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 4200×4100 pixel image. The *vnir_Band3n*, *2* and *1* 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

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

As with the previous examples, the $4200 \times 4100 \times 3$ array can now be displayed using

```
imshow(naivasha_rgb);
```

MATLAB scales the 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 (Fig. 7.2), we use the command

```
imwrite(naivasha_rgb,'naivasha_image_vs1_matlab.tif','tif')
```

This command saves the RGB composite as a TIFF-file of about 50 MB in the working directory, which can then be processed with other software such as Adobe Photoshop.

7.5 Georeferencing Satellite Images

The processed ASTER image does not yet have a coordinate system, and the image therefore needs to be tied to a geographical reference frame (*georeferencing*). The raw data can be loaded and transformed into an RGB composite by typing

```
clear
```

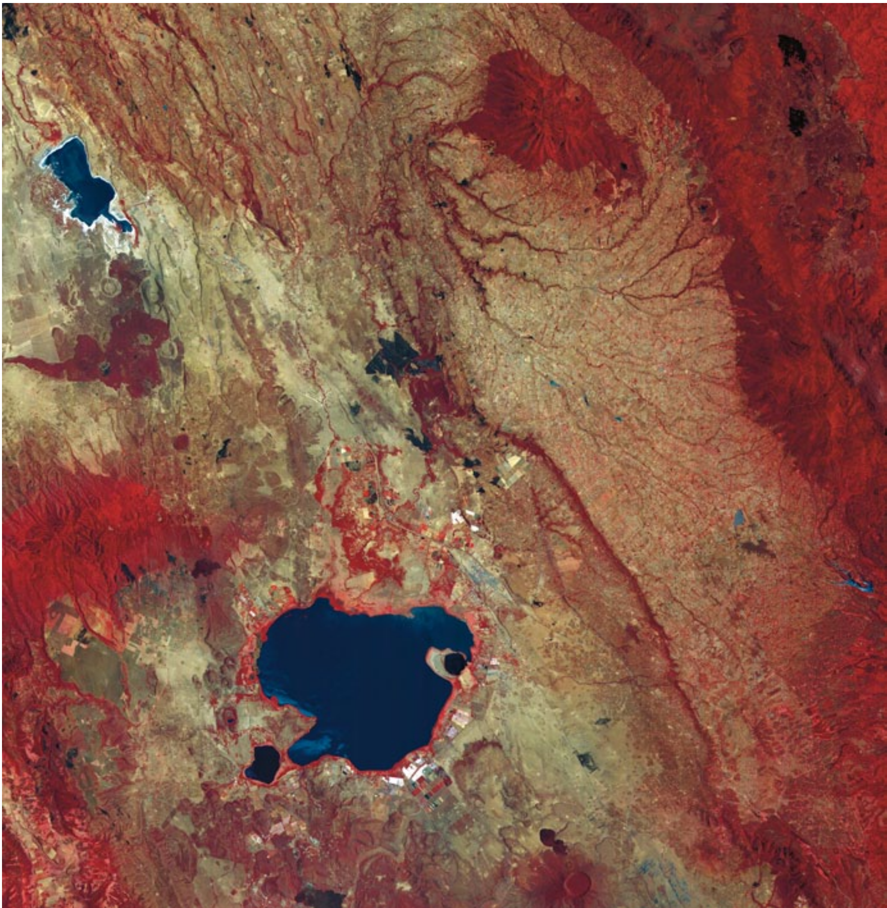

Fig. 7.2 RGB composite of a TERRA-ASTER image using the spectral infrared bands *vnir_Band3n*, *2* and *1*. The result is displayed using `imshow`. Original image courtesy of NASA/GSFC/METI/ERSDAC/JAROS and U.S./Japan ASTER Science Team.


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

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

The HDF browser

```
hdftool('naivasha_data.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_data.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* that contains the following information:

```
upperleft  = [-0.319922, 36.214332];
upperright = [-0.400443, 36.770406];
lowerleft  = [-0.878267, 36.096003];
lowerright = [-0.958743, 36.652213];
```

These two-element vectors can be collected into a single array `inputpoints`. Subsequently, the left and right columns can be flipped in order to have *x=longitudes* and *y=latitudes*.

```
inputpoints(1,:) = upperleft;
inputpoints(2,:) = lowerleft;
inputpoints(3,:) = upperright;
inputpoints(4,:) = lowerright;
inputpoints = fliplr(inputpoints);
```

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,:) = [1,4200];
basepoints(2,:) = [1,1];
basepoints(3,:) = [4100,4200];
basepoints(4,:) = [4100,1];
```

The function `cp2tform` now takes the pairs of control points, `inputpoints` and `basepoints`, and uses them to infer a spatial transformation matrix `tform`.

```
tform = cp2tform(inputpoints,basepoints,'affine');
```

This transformation can be applied to the original RGB composite `naivasha_rgb` in order to obtain a georeferenced version of the satellite image `newnaivasha_rgb`.

```
[newnaivasha_rgb,x,y] = imtransform(naivasha_rgb,tform);
```

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 longitude and latitude range by the array dimension and by subtracting one from the result. Note the difference between the MATLAB numbering convention

and the common coding of maps used in the literature. The north/south suffix is generally replaced by a negative sign for south, whereas MATLAB coding conventions require negative signs for north.

```
X = 36.096003 : (36.770406-36.096003)/8569 : 36.770406;
Y =  0.319922 : ( 0.958743- 0.319922)/8400 :  0.958743;
```

The georeferenced image is displayed with coordinates on the axes and a superimposed grid (Fig. 7.3).

```
imshow(newnaivasha_rgb,'XData',X,'YData',Y), axis on, grid on
xlabel('Longitude'), ylabel('Latitude')
title('Georeferenced ASTER Image')
```

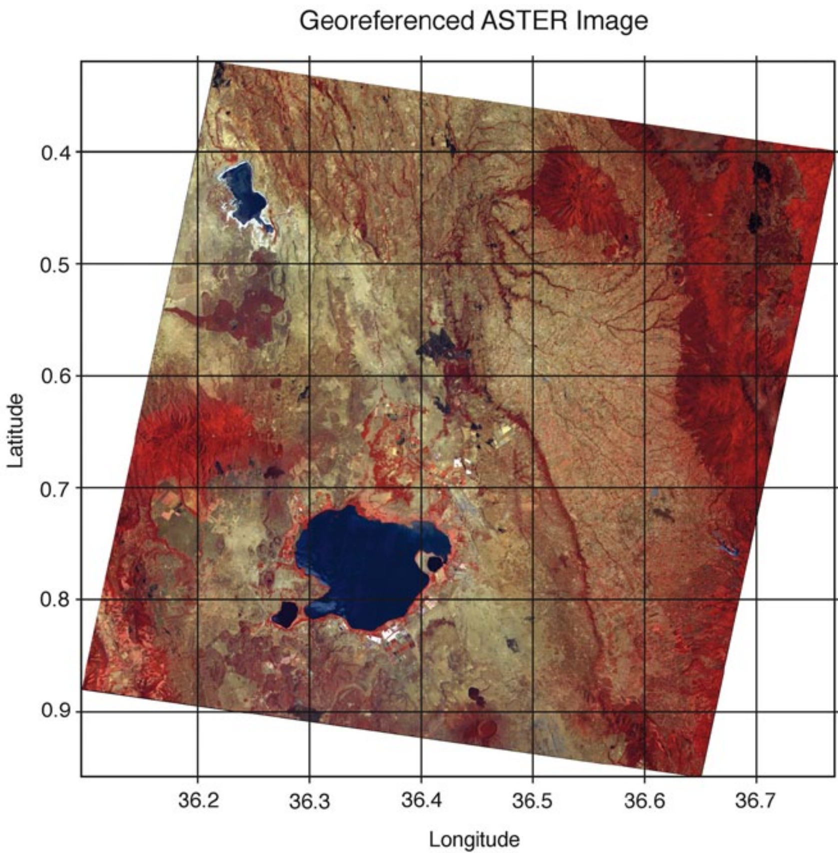

Fig. 7.3 Geoferenced RGB composite of a TERRA-ASTER image using the infrared bands *vnir_Band3n, 2* and *1*. The result is displayed using `imshow`. Original image courtesy of NASA/GSFC/METI/ERSDAC/JAROS and U.S./Japan ASTER Science Team.

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

```
print -djpeg70 -r600 naivasha_georef_vs1_matlab.jpg
```

to export it as a 1 MB JPEG file compressed to 70 %, with a resolution of 600 dpi. Alternatively, we can save it as a TIFF file with a resolution of 600 dpi

```
print -dtiff -r600 naivasha_georef_vs1_matlab.tif
```

which has a size of about 25 MB.

7.6 Digitizing from the Screen: From Pixel to Vector

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 for this type of application include digitizing river networks and catchment areas on topographic maps, the outlines of lithologic units on geological maps, the distribution of landslides on satellite images, and the distribution of mineral grains in a microscope image. The digitizing procedure consists of the following steps. Firstly, the image is 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 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 using a mouse on the screen. 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. Next, the image is 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_vs1_matlab.jpg')
```

The function first asks for the coordinates for the limits of the x- and 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_vs1_matlab.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 dimension 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.

Recommended Reading

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

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

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

The Mathworks (2010) Image Processing Toolbox – User's Guide. The MathWorks, Natick, MA