

## 2 Introduction to MATLAB

Graphical user interface of MATLAB in typical use. The software comes up with several window panels. The desktop layout includes the Current Folder panel, the Command Window, the Command History panel and the Workspace panel. When using MATLAB several Figure Windows and the Editor are displayed.

### 2.1 MATLAB in Earth Sciences

MATLAB® is a software package developed by The MathWorks, Inc., founded by Cleve Moler, Jack Little and Steve Bangert in 1984, which has its headquarters in Natick, Massachusetts (<http://www.mathworks.com>). MATLAB was designed to perform mathematical calculations, to analyze and visualize data, and to facilitate the writing of new software programs. The advantage of this software is that it combines comprehensive math and graphics functions with a powerful high-level language. Since MATLAB contains a large library of ready-to-use routines for a wide range of applications, the user can solve technical computing problems much more quickly than with traditional programming languages, such as C++ and FORTRAN. The standard library of functions can be significantly expanded by add-on toolboxes, which are collections of functions for special purposes such as image processing, creating map displays, performing geospatial data analysis or solving partial differential equations.

During the last few years MATLAB has become an increasingly popular tool in earth sciences. It has been used for finite element modeling, processing of seismic data, analyzing satellite imagery, and for the generation

of digital elevation models from satellite data. The continuing popularity of the software is also apparent in published scientific literature, and many conference presentations have also made reference to MATLAB. Universities and research institutions have recognized the need for MATLAB training for staff and students, and many earth science departments across the world now offer MATLAB courses for undergraduates. The MathWorks, Inc. provides classroom kits for teachers at a reasonable price, and it is also possible for students to purchase a low-cost edition of the software. This student version provides an inexpensive way for students to improve their MATLAB skills.

The following sections contain a tutorial-style introduction to MATLAB, covering the setup on the computer (Section 2.2), the MATLAB syntax (Sections 2.3 and 2.4), data input and output (Sections 2.5 and 2.6), programming (Sections 2.7 and 2.8), and visualization (Section 2.9). Advanced sections are also included on generating M-files to recreate graphics (Section 2.10), on publishing M-files (Section 2.11), and on creating graphical user interfaces (Section 2.12). The reader is recommended to go through the entire chapter in order to obtain a good knowledge of the software before proceeding to the following chapters of the book. A more detailed introduction can be found in the *MATLAB Primer* (MathWorks 2014a) which is available in print form, online and as PDF file.

In this book we use MATLAB Version 8 (Release 2014b), the Image Processing Toolbox Version 9.1, the Mapping Toolbox Version 4.0.2, the Signal Processing Toolbox Version 6.22, the Statistics Toolbox Version 9.1, the Wavelet Toolbox Version 4.14, and the Simulink 3D Animation Toolbox Version 7.2.

## 2.2 Getting Started

The software package comes with extensive documentation, tutorials and examples. The first three chapters of the book *MATLAB Primer* (MathWorks 2014a) are directed at beginners. The chapters on programming, creating graphical user interfaces (GUIs) and development environments are aimed at more advanced users. Since *MATLAB Primer* provides all the information required to use the software, this introduction concentrates on the most relevant software components and tools used in the following chapters of this book.

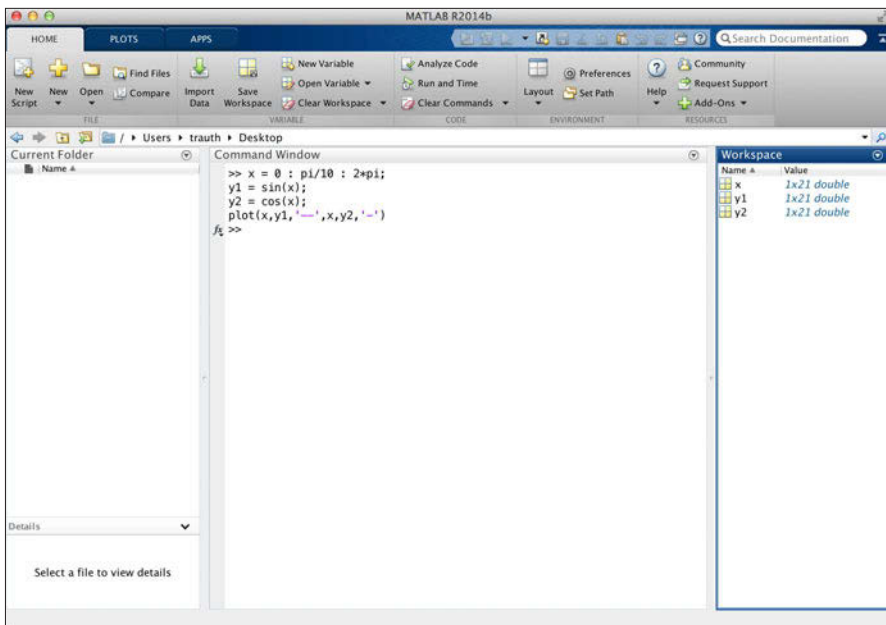
After the installation of MATLAB, the software is launched either by clicking the shortcut icon on the desktop or by typing

```
matlab
```

in the operating system prompt. The software then comes up with several window panels (Fig. 2.1). The default desktop layout includes the *Current Folder* panel that lists the files in the directory currently being used. The *Command Window* presents the interface between the software and the user, i.e., it accepts MATLAB commands typed after the prompt, `>>`. The *Workspace* panel lists the variables in the MATLAB workspace, which is empty when starting a new software session. In this book we mainly use the Command Window and the built-in *Editor*, which can be launched by typing

`edit`

By default, the software stores all of your MATLAB-related files in the startup folder named *MATLAB*. Alternatively, you can create a personal working directory in which to store your MATLAB-related files. You should then make this new directory the working directory using the *Current Folder* panel or the *Folder Browser* at the top of the MATLAB desktop. The software uses a *Search Path* to find MATLAB-related files, which are



**Fig. 2.1** Screenshot of the MATLAB default desktop layout including the *Current Folder* (left in the figure), the *Command Window* (center), the *Workspace* (right) panels. This book uses only the Command Window and the built-in *Editor*, which can be called up by typing `edit` after the prompt. All information provided by the other panels can also be accessed through the Command Window.

organized in directories on the hard disk. The default search path includes only the *MATLAB\_R2014b* directory that has been created by the installer in the applications folder and the default working directory *MATLAB*. To see which directories are in the search path or to add new directories, select *Set Path* from the *Home* toolstrip of the MATLAB desktop, and use the *Set Path* dialog box. The modified search path is saved in a file *pathdef.m* on your hard disk. The software will then in future read the contents of this file and direct MATLAB to use your custom path list.

## 2.3 The Syntax

The name MATLAB stands for *matrix laboratory*. The classic object handled by MATLAB is a *matrix*, i.e., a rectangular two-dimensional *array* of numbers. A simple 1-by-1 array is a *scalar*. Arrays with one column or row are *vectors*, time series or other one-dimensional data fields. An *m*-by-*n* array can be used for a digital elevation model or a grayscale image. Red, green and blue (RGB) color images are usually stored as three-dimensional arrays, i.e., the colors red, green and blue are represented by an *m*-by-*n*-by-3 array.

Before proceeding, we need to clear the workspace by typing

```
clear
```

after the prompt in the Command Window. Clearing the workspace is always recommended before working on a new MATLAB project to avoid name conflicts with previous projects. We can also go a step further, close all Figure Windows using `close all` and clear the content of the Command Window using `clc`. It is therefore recommended that a new MATLAB project should always start with the line

```
clear, close all, clc
```

Entering matrices or arrays in MATLAB is easy. To enter an arbitrary matrix, type

```
A = [2 4 3 7; 9 3 -1 2; 1 9 3 7; 6 6 3 -2]
```

which first defines a variable **A**, then lists the elements of the array in square brackets. The rows of **A** are separated by semicolons, whereas the elements of a row are separated by blank spaces, or alternatively, by commas. After pressing *return*, MATLAB displays the array

```
A =
     2     4     3     7
     9     3    -1     2
```

```

1     9     3     7
6     6     3    -2

```

Displaying the elements of `A` could be problematic for very large arrays such as digital elevation models consisting of thousands or millions of elements. To suppress the display of an array or the result of an operation in general, the line should be ended with a semicolon.

```
A = [2 4 3 7; 9 3 -1 2; 1 9 3 7; 6 6 3 -2];
```

The array `A` is now stored in the workspace and we can carry out some basic operations with it, such as computing the sum of elements,

```
sum(A)
```

which results in the display

```
ans =
    18    22     8    14

```

Since we did not specify an output variable, such as `A` for the array entered above, MATLAB uses a default variable `ans`, short for *answer* or *most recent answer*, to store the results of the calculation. In general, we should define variables since the next computation without a new variable name will overwrite the contents of `ans`.

The above example illustrates an important point about MATLAB: the software prefers to work with the columns of arrays. The four results of `sum(A)` are obviously the sums of the elements in each of the four columns of `A`. To sum all elements of `A` and store the result in a scalar `b`, we simply need to type

```
b = sum(sum(A));
```

which first sums the columns of the array and then the elements of the resulting vector. We now have two variables, `A` and `b`, stored in the workspace. We can easily check this by typing

```
whos
```

which is one the most frequently-used MATLAB commands. The software then lists all variables in the workspace, together with information about their sizes or dimensions, number of bytes, classes and attributes (see Section 2.5 for details about classes and attributes of objects).

```

Name      Size      Bytes  Class      Attributes
A         4x4         128   double

```



Movie  
2.1

```
ans      1x4      32 double
b         1x1       8 double
```



Movie  
2.2

Note that by default MATLAB is case sensitive, i.e., `A` and `a` can define two different variables. In this context, it is recommended that capital letters be used for arrays that have two dimensions or more and lower-case letters for one-dimensional arrays (or vectors) and for scalars. However, it is also common to use variables with mixed large and small letters. This is particularly important when using descriptive variable names, i.e., variables whose names contain information concerning their meaning or purpose, such as the variable `CatchmentSize`, rather than a single-character variable `a`. We could now delete the contents of the variable `ans` by typing

```
clear ans
```

Next, we will learn how specific array elements can be accessed or exchanged. Typing

```
A(3,2)
```

simply yields the array element located in the third row and second column, which is `9`. The array indexing therefore follows the rule (*row, column*). We can use this to replace single or multiple array elements. As an example we type

```
A(3,2) = 30
```

to replace the element `A(3,2)` by `30` and to display the entire array.

```
A =
     2     4     3     7
     9     3    -1     2
     1    30     3     7
     6     6     3    -2
```

If we wish to replace several elements at one time, we can use the *colon operator*. Typing

```
A(3,1:4) = [1 3 3 5]
```

or

```
A(3,:) = [1 3 3 5]
```

replaces all elements of the third row of the array `A`. The colon operator also has several other uses in MATLAB, for instance as a shortcut for entering array elements such as

```
c = 0 : 10
```

which creates a vector, or a one-dimensional array with a single row, containing all integers from 0 to 10. The resultant MATLAB response is

```
c =
    0    1    2    3    4    5    6    7    8    9   10
```

Note that this statement creates 11 elements, i.e., the integers from 1 to 10 and the zero. A common error when indexing arrays is to ignore the zero and therefore expect 10 elements instead of 11 in our example. We can check this from the output of `whos`.

Name	Size	Bytes	Class	Attributes
A	4x4	128	double	
ans	1x1	8	double	
b	1x1	8	double	
c	1x11	88	double	

The above command creates only integers, i.e., the interval between the array elements is one unit. However, an arbitrary interval can be defined, for example 0.5 units. This is later used to create evenly-spaced time vectors for time series analysis. Typing

```
c = 1 : 0.5 : 10
```

results in the display

```
c =
Columns 1 through 6
    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000
Columns 7 through 12
    4.0000    4.5000    5.0000    5.5000    6.0000    6.5000
Columns 13 through 18
    7.0000    7.5000    8.0000    8.5000    9.0000    9.5000
Column 19
   10.0000
```

which autowraps the lines that are longer than the width of the Command Window. The display of the values of a variable can be interrupted by pressing *Ctrl*+*C* (*Control*+*C*) on the keyboard. This interruption affects only the output in the Command Window, whereas the actual command is processed before displaying the result.

MATLAB provides standard arithmetic operators for addition, `+`, and subtraction, `-`. The asterisk, `*`, denotes matrix multiplication involving inner products between rows and columns. For instance, we multiply the matrix `A` with a new matrix `B`

```
B = [4 2 6 5; 7 8 5 6; 2 1 -8 -9; 3 1 2 3];
```

the matrix multiplication is then

```
C = A * B'
```

where `'` is the complex conjugate transpose, which turns rows into columns and columns into rows. This generates the output

```
C =
    69    103    -79    37
    46     94     11    34
    53     76    -64    27
    44     93     12    24
```

In linear algebra, matrices are used to keep track of the coefficients of linear transformations. The multiplication of two matrices represents the combination of two linear transformations into a single transformation. Matrix multiplication is not commutative, i.e., `A*B'` and `B*A'` yield different results in most cases. Similarly, MATLAB allows matrix divisions representing different transformations, with `/` as the operator for right-hand matrix division and `\` as the operator for left-hand division. Finally, the software also allows powers of matrices, `^`.

In earth sciences, however, matrices are often simply used as two-dimensional arrays of numerical data rather than a matrix *sensu stricto* representing a linear transformation. Arithmetic operations on such arrays are carried out element-by-element. While this does not make any difference in addition and subtraction, it does affect multiplicative operations. MATLAB uses a dot, `.`, as part of the notation for these operations.

As an example multiplying `A` and `B` element-by-element is performed by typing

```
C = A .* B
```

which generates the output

```
C =
     8     8    18    35
    63    24    -5    12
     2     3   -24   -45
    18     6     6    -6
```

## 2.4 Array Manipulation

MATLAB provides a wide range of functions with which to manipulate arrays (or matrices). This section introduces the most important functions



for array manipulation, which are used later in the book. We first clear the workspace and create two arrays, `A` and `B`, by typing

```
clear

A = [2 4 3; 9 3 -1]
B = [1 9 3; 6 6 3]
```

which yields

```
A =
     2     4     3
     9     3    -1

B =
     1     9     3
     6     6     3
```

When we work with arrays, we sometimes need to concatenate two or more arrays into a single array. We can use either `cat(dim,A,B)` with `dim=1` to concatenate the arrays `A` and `B` along the first dimension (i.e., along the rows). Alternatively, we can use the function `vertcat` to concatenate the arrays `A` and `B` vertically. By typing either

```
C = cat(1,A,B)
```

or

```
C = vertcat(A,B)
```

we obtain (in both cases)

```
C =
     2     4     3
     9     3    -1
     1     9     3
     6     6     3
```

Similarly, we can concatenate arrays horizontally, i.e., concatenate the arrays along the second dimension (i.e., along the columns) by typing

```
D = cat(2,A,B)
```

or using the function `horzcat` instead

```
D = horzcat(A,B)
```

which both yield

```
D =
```

```

2     4     3     1     9     3
9     3    -1     6     6     3

```

When working with satellite images we often concatenate three spectral bands of satellite images into three-dimensional arrays of the colors red, green and blue (RGB) (Sections 2.5 and 8.4). We again use `cat(dim,A,B)` with `dim=3` to concatenate the arrays `A` and `B` along the third dimension by typing

```
E = cat(3,A,B)
```

which yields

```

E(:,:,1) =
     2     4     3
     9     3    -1

E(:,:,2) =
     1     9     3
     6     6     3

```

Typing

```
whos
```

yields

Name	Size	Bytes	Class	Attributes
A	2x3	48	double	
B	2x3	48	double	
C	4x3	96	double	
D	2x6	96	double	
E	2x3x2	96	double	

indicating that we have now created a three-dimensional array, as the size 2-by-3-by-2 suggests. Alternatively, we can use

```
size(E)
```

which yields

```

ans =
     2     3     2

```

to see that the array has 2 rows, 3 columns, and 2 layers in the third dimension. Using `length` instead of `size`,

```
length(A)
```

yields

```
ans =
     3
```

which tells us the dimension of the largest array only. Hence `length` is normally used to determine the length of a one-dimensional array (or vector), such the evenly-spaced time axis `c` that was created in Section 2.3.

MATLAB uses a matrix-style indexing of arrays with the (1,1) element being located in the upper-left corner of arrays. Other types of data that are to be imported into MATLAB may follow a different indexing convention. As an example, digital terrain models (introduced in Chapter 7.3 to 7.5) often have a different way of indexing and therefore need to be flipped in an up-down direction or, in other words, about a horizontal axis. Alternatively, we can flip arrays in a left-right direction (i.e., about a vertical axis). We can do this by using `flipud` for flipping in an up-down direction and `fliplr` for flipping in a left-right direction

```
F = flipud(A)
F = fliplr(A)
```

yielding

```
F =
     9     3    -1
     2     4     3

F =
     3     4     2
    -1     3     9
```

In more complex examples we can use `circshift(A,K,dim)` to circularly shift (i.e., rotate) arrays by `K` positions along the dimension `dim`. As an example we can shift the array `A` by 1 position along the 2nd dimension (i.e., along the rows) by typing

```
G = circshift(A,1,2)
```

which yields

```
G =
     3     2     4
    -1     9     3
```

We can also use `reshape(A,[m n])` to completely reshape the array. The result is an `m`-by-`n` array `H` whose elements are taken column-wise from `A`. As an example we create a 3-by-2 array from `A` by typing

```
H = reshape(A,[3 2])
```

which yields

```
H =
     2     3
     9     3
     4    -1
```

Another important way to manipulate arrays is to sort their elements. As an example we can use `sort(C,dim,mode)` with `dim=1` and `mode='ascend'` to sort the elements of `C` in ascending order along the first array dimension (i.e., the rows). Typing

```
I = sort(C,1,'ascend')
```

yields

```
I =
     1     3    -1
     2     4     3
     6     6     3
     9     9     3
```

The function `sortrows(C,column)` with `column=2` sorts the rows of `C` according to the second column. Typing

```
J = sortrows(C,2)
```

yields

```
J =
     9     3    -1
     2     4     3
     6     6     3
     1     9     3
```

Array manipulation also includes the comparison of arrays, for example by checking whether elements in `A(i,j)` are also found in `B` using `ismember`. Typing

```
A, B
```

```
K = ismember(A,B)
```

yields

```
A =
     2     4     3
     9     3    -1

B =
     1     9     3
     6     6     3
```

```
K =
    0     0     1
    1     1     0
```

The array  $L(i,j)$  is zero if  $A(i,j)$  is not in  $B$ , and one if  $A(i,j)$  is in  $B$ . We can also locate elements within  $A$  for which a statement is true. For example we can locate elements with values less than zero and replace them with `NaN`s by typing

```
L = A;
L(find(L<0)) = NaN
```

or, more briefly

```
L(L<0) = NaN
```

which yields

```
L =
    2     4     3
    9     3    NaN
```

This is very useful when working with digital elevation models, where values below sea level are not relevant. Alternatively, we can replace data voids other than `NaN`s such as `-32768`, which are often used with digital terrain models (Section 7.3 to 7.5). We can then determine which elements of an array are `NaN`s by typing

```
M = isnan(L)
```

which yields

```
M =
    0     0     0
    0     0     1
```

where `NaN`s are indicated by ones and non-`NaN` values are indicated by zeros. Which of the elements in array  $A$  are unique can be determined by typing

```
N = unique(A)
```

which yields

```
N =
   -1
    2
    3
    4
    9
```

The value of 3 occurs twice in `A` and the number of elements in `N` is therefore one less than in `A`.

## 2.5 Data Structures and Classes of Objects

The default data type or *class* in MATLAB is *double precision* or `double`, which stores data in a 64-bit array of floating-point numbers. Such floating-point numbers are approximations of real numbers that allow a maximum range of values in a limited numbers of bits. A double-precision array allows the sign of a number to be stored (bit 63), together with the exponent (bits 62 to 52), and roughly 16 significant decimal digits (bits 51 to 0). Typing

```
clear

realmin('double')
realmax('double')
```

yields the smallest and largest positive floating-point number in double precision

```
ans =
    2.2251e-308

ans =
    1.7977e+308
```

The actual number of floating point numbers is therefore limited by the number of bits available, in contrast to real numbers. The difference between 1.0 and the next largest double-precision number can be calculated using the floating-point relative accuracy `eps` by typing

```
eps(1.0)
```

which yields

```
ans =
    2.2204e-16
```

The round-off error depends on the value of the real number; it is, for example, different for 5.0, as we can see by typing

```
eps(5.0)
```

which yields

```
ans =
    8.8818e-16
```

For real numbers there is, by definition, no such gap between consecutive numbers. The use of a finite number of floating-point numbers is limited by the number of available bits due to the finite precision arithmetic of a computer. There are countless examples available with which to demonstrate this, but we will restrict ourselves to the simple example of the sine of  $\pi$ . Typing

```
sin(pi)
```

yields

```
ans =  
1.2246e-16
```

and not, as would be expected, zero. Since `pi` is only the nearest floating-point value to  $\pi$ , the sine of `pi` is not exactly zero but a value very close to zero.

Let us now look at some examples of arrays in order to familiarize ourselves with the different data types in MATLAB. For the first example we create a 3-by-4 array of random numbers with double precision by typing

```
clear  
  
rng(0)  
A = rand(3,4)
```

We use the function `rand` that generates uniformly distributed pseudorandom numbers within the open interval  $[0,1]$ . To obtain identical data values, we use `rng(0)` to reset the random number generator by using the integer 0 as *seed* (see Chapter 3 for more details on random number generators and types of distributions). Since we did not use a semicolon here we get the output

```
A =  
0.8147    0.9134    0.2785    0.9649  
0.9058    0.6324    0.5469    0.1576  
0.1270    0.0975    0.9575    0.9706
```

By default, the output is in a scaled fixed point format with 5 digits, e.g., 0.8147 for the `(1,1)` element of `A`. Typing

```
format long
```

switches to a fixed point format with 16 digits for double precision. Recalling `A` by typing

```
A
```

yields the output

```
A =
Columns 1 through 2
    0.814723686393179    0.913375856139019
    0.905791937075619    0.632359246225410
    0.126986816293506    0.097540404999410

Columns 3 through 4
    0.278498218867048    0.964888535199277
    0.546881519204984    0.157613081677548
    0.957506835434298    0.970592781760616
```

which autowraps those lines that are longer than the width of the Command Window. The command `format` does not affect how the computations are carried out, i.e., the precision of the computation results remains unchanged. The precision is, however, affected by converting the data type from *double* to 32-bit *single precision*. Typing

```
B = single(A)
```

yields

```
B =
    0.8147237    0.9133759    0.2784982    0.9648885
    0.9057919    0.6323593    0.5468815    0.1576131
    0.1269868    0.0975404    0.9575068    0.9705928
```

Although we have switched to `format long`, only 8 digits are displayed. The command `whos` lists the variables `A` and `B` with information on their sizes or dimensions, number of bytes, and classes

Name	Size	Bytes	Class	Attributes
A	3x4	96	double	
B	3x4	48	single	

The default class `double` is used in all MATLAB operations in which the physical memory of the computer is not a limiting factor, whereas `single` is used when working with large data sets. The double-precision variable `A`, whose size is 3-by-4 elements, requires  $3 \cdot 4 \cdot 64 = 768$  bits or  $768/8 = 96$  bytes of memory, whereas `B` requires only 48 bytes and so has half the memory requirement of `A`. Introducing at least one complex number to `A` doubles the memory requirement since both real and imaginary parts are double precision, by default. Switching back to `format short` and typing

```
format short
A(1,3) = 4i + 3
```

yields



```

A =
Columns 1 through 2
    0.8147 + 0.0000i    0.9134 + 0.0000i
    0.9058 + 0.0000i    0.6324 + 0.0000i
    0.1270 + 0.0000i    0.0975 + 0.0000i

Columns 3 through 4
    3.0000 + 4.0000i    0.9649 + 0.0000i
    0.5469 + 0.0000i    0.1576 + 0.0000i
    0.9575 + 0.0000i    0.9706 + 0.0000i

```

and the variable listing is now

Name	Size	Bytes	Class	Attributes
A	3x4	192	double	complex
B	3x4	48	single	

indicating the class `double` and the attribute `complex`.

MATLAB also works with even smaller data types such as 1-bit, 8-bit and 16-bit data, in order to save memory. These data types are used to store digital elevation models or images (see Chapters 7 and 8). For example  $m$ -by- $n$  pixel RGB true color images are usually stored as three-dimensional arrays, i.e., the three colors are represented by an  $m$ -by- $n$ -by-3 array (see Chapter 8 for more details on RGB composites and true color images). Such multi-dimensional arrays can be generated by concatenating three two-dimensional arrays representing the  $m$ -by- $n$  pixels of an image. First, we generate a 100-by-100 array of uniformly distributed random numbers in the range  $[0,1]$ . We then multiply the random numbers by 255 to get values between 0 and 255.

```

clear

rng(0)
I1 = 255 * rand(100,100);
I2 = 255 * rand(100,100);
I3 = 255 * rand(100,100);

```

The command `cat` concatenates the three two-dimensional arrays (8 bits each) into a three-dimensional array (3·8 bits=24 bits).

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

Since RGB images are represented by integer values between 0 and 255 for each color, we convert the 64-bit double-precision values to unsigned 8-bit integers using `uint8` (Section 8.2). The function `uint8` rounds the values in `I` to the nearest integer. Any values that are outside the range  $[0,255]$  are assigned to the nearest endpoint (0 or 255).

```
I = uint8(I);
```

Typing `whos` then yields

Name	Size	Bytes	Class	Attributes
I	100x100x3	30000	uint8	
I1	100x100	80000	double	
I2	100x100	80000	double	
I3	100x100	80000	double	

Since 8 bits can be used to store 256 different values, this data type can be used to store integer values between 0 and 255, whereas using `int8` to create signed 8-bit integers generates values between  $-128$  and  $+127$ . The value of zero requires one bit and there is therefore no space left in which to store  $+128$ . Finally, `imshow` can be used to display the three-dimensional array as a true color image.

```
imshow(I)
```

We next introduce *structure arrays* as a MATLAB data type. Structure arrays are multi-dimensional arrays with elements accessed by textual field designators. These arrays are data containers that are particularly helpful in storing any kind of information about a sample in a single variable. As an example we can generate a structure array `sample_1` that includes the image array `I` defined in the previous example as well as other types of information about a sample, such as the name of the sampling location, the date of sampling, and geochemical measurements, stored in a 10-by-10 array.

```
sample_1.location = 'Plougasnou';
sample_1.date = date;
sample_1.image = I;
sample_1.geochemistry = rand(10,10);
```

The first layer of the structure array `sample_1` contains a character array, i.e., a two-dimensional array of the data type `char` containing a character string. We can create such an array by typing

```
location = 'Plougasnou';
```

We can list the size, class and attributes of a single variable such as `location` by typing

```
whos location
```

and learn from

Name	Size	Bytes	Class	Attributes
location	1x10	20	char	

that the size of this character array `location` corresponds to the number of characters in the word *Plougasnou*. Character arrays are 16-bit arrays, i.e.,  $2^{16}=65,536$  different characters can be stored in such arrays. The character string `location` therefore requires  $10 \cdot 16=160$  bits or  $160/8=20$  bytes of memory. In addition, the second layer `datum` in the structure array `sample_1` contains a character string generated by `date` that yields a string containing the current date in `dd-mm-yyyy` format. We access this particular layer in `sample_1` by typing

```
sample_1.date
```

which yields

```
ans =
    27-Jun-2014
```

as an example. The third layer of `sample_1` contains the image created in the previous example, while the fourth layer contains a 10-by-10 array of uniformly-distributed pseudorandom numbers. All layers of `sample_1` can be listed by typing

```
sample_1
```

resulting in the output

```
sample_1 =
    location: 'Plougasnou'
      date: '06-Oct-2009'
    image: [100x100x3 uint8]
  geochemistry: [10x10 double]
```

This represents a list of the layers `location`, `date`, `image` and `geochemistry` within the structure array `sample_1`. Some variables are listed in full, whereas larger data arrays are only represented by their size. In the list of the layers within the structure array `sample_1`, the array `image` is characterized by its size `100x100x3` and the class `uint8`. The variable `geochemistry` in the last layer of the structure array contains a 10-by-10 array of double-precision numbers. The command

```
whos sample_1
```

does not list the layers in `sample_1` but the name of the variable, the bytes and the class `struct` of the variable.

Name	Size	Bytes	Class	Attributes
sample_1	1x1	31546	struct	

MATLAB also has *cell arrays* as an alternative to structure arrays. Both classes or data types are very similar and are containers of different types and sizes of data. The most important difference between the two is that the containers of a structure array are *named fields*, whereas a cell array uses *numerically-indexed cells*. Structure arrays are often used in applications where the organization of the data is particularly important. Cell arrays are often used when processing large data sets in count-controlled loops (Section 2.7).

As an example of cell arrays we use the same data collection as in structure arrays, with the layers of the structure array as the cells in the cell array. The cell array is created by enclosing the location name *Plougasnou*, the date, the image `I` and the 10-by-10 array of uniformly-distributed pseudorandom numbers in curly brackets.

```
C = {'Plougasnou' date I rand(10,10)}
```

Typing

```
C
```

lists the contents of the cell array

```
C =  
Columns 1 through 2  
'Plougasnou' '27-Jun-2014'  
Columns 3 through 4  
[100x100x3 uint8] [10x10 double]
```

which contains the location name and date. The image and the array of random numbers are too large to be displayed in the Command Window, but the dimensions and class of the data are displayed instead. We access a particular cell in `C`, e.g., the cell 2, by typing

```
C{2}
```

which yields

```
ans =  
27-Jun-2014
```

We can also access the other cells of the cell array in a similar manner.

## 2.6 Data Storage and Handling

This section deals with how to store, import, and export data with MATLAB. Many of the data formats typically used in earth sciences have to be converted

before being analyzed with MATLAB. Alternatively, the software provides several import routines to read many binary data formats in earth sciences, such as those used to store digital elevation models and satellite data.

A computer generally stores data as *binary digits* or *bits*. A bit is analogous to a two-way switch with two states, on = 1 and off = 0. The bits are joined together to form larger groups, such as bytes consisting of 8 bits, in order to store more complex types of data. Such groups of bits are then used to encode data, e.g., numbers or characters. Unfortunately, different computer systems and software use different schemes for encoding data. For instance, the characters in the widely-used text processing software Microsoft Word differ from those in Apple Pages. Exchanging binary data is therefore difficult if the various users use different computer platforms and software. Binary data can be stored in relatively small files if both partners are using similar systems of data exchange. The transfer rate for binary data is generally faster than that for the exchange of other file formats.

Various formats for exchanging data have been developed during recent decades. The classic example for the establishment of a data format that can be used with different computer platforms and software is the American Standard Code for Information Interchange (ASCII) that was first published in 1963 by the American Standards Association (ASA). As a 7-bit code, ASCII consists of  $2^7=128$  characters (codes 0 to 127). Whereas ASCII-1963 was lacking lower-case letters, in the ASCII-1967 update lower-case letters as well as various control characters such as escape and line feed, and various symbols such as brackets and mathematical operators, were also included. Since then, a number of variants appeared in order to facilitate the exchange of text written in non-English languages, such as the expanded ASCII containing 255 codes, e.g., the Latin-1 encoding.

The simplest way to exchange data between a certain piece of software and MATLAB is using the ASCII format. Although the newer versions of MATLAB provide various import routines for file types such as Microsoft Excel binaries, most data arrive in the form of ASCII files. Consider a simple data set stored in a table such as

SampleID	Percent C	Percent S
101	0.3657	0.0636
102	0.2208	0.1135
103	0.5353	0.5191
104	0.5009	0.5216
105	0.5415	-999
106	0.501	-999

The first row contains the names of the variables and the columns provide the percentages of carbon and sulfur in each sample. The absurd value -999

indicates missing data in the data set. Two things have to be changed to convert this table into MATLAB format. First, MATLAB uses `NaN` as the representation for *Not-a-Number* that can be used to mark missing data or gaps. Second, a percent sign, `%`, should be added at the beginning of the first line. The percent sign is used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in the code.

```
%SampleID  Percent C      Percent S
101         0.3657        0.0636
102         0.2208        0.1135
103         0.5353        0.5191
104         0.5009        0.5216
105         0.5415        NaN
106         0.501         NaN
```

MATLAB will ignore any text appearing after the percent sign and continue processing on the next line. After editing this table in a text editor, such as the *MATLAB Editor*, it can be saved as ASCII text file *geochem.txt* in the current working directory (Fig. 2.2). The MATLAB workspace should first be cleared by typing

```
clear
```

after the prompt in the Command Window. MATLAB can now import the data from this file with the `load` command.

```
load geochem.txt
```

MATLAB then loads the contents of the file and assigns the array to a variable `geochem` specified by the filename *geochem.txt*. Typing

```
whos
```

yields

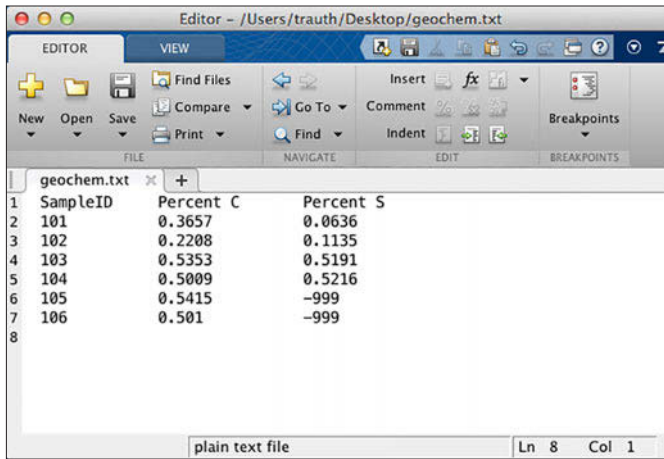
Name	Size	Bytes	Class	Attributes
geochem	6x3	144	double	

The command `save` now allows workspace variables to be stored in a binary format.

```
save geochem_new.mat
```

*MAT-files* are double-precision binary files using *.mat* as extension. The advantage of these binary MAT-files is that they are independent of the computer platforms running different floating-point formats. The command

```
save geochem_new.mat geochem
```



**Fig. 2.2** Screenshot of MATLAB Editor showing the content of the file *geochem.txt*. The first line of the text needs to be commented by a percent sign at the beginning of the line, followed by the actual data array. The `-999` values need to be replaced by `NaNs`.

saves only the variable `geochem` instead of the entire workspace. The option `-ascii`, for example

```
save geochem_new.txt geochem -ascii
```

again saves the variable `geochem`, but in an ASCII file named *geochem\_new.txt* in a floating-point format with 8 digits:

```
1.0100000e+02  3.6570000e-01  6.3600000e-02
1.0200000e+02  2.2080000e-01  1.1350000e-01
1.0300000e+02  5.3530000e-01  5.1910000e-01
1.0400000e+02  5.0090000e-01  5.2160000e-01
1.0500000e+02  5.4150000e-01      NaN
1.0600000e+02  5.0100000e-01      NaN
```

In contrast to the binary file *geochem\_new.mat*, this ASCII file can be viewed and edited using the MATLAB Editor or any other text editor.

Such data files, especially those that are produced by electronic instruments, can look much more complicated than the example file *geochem.txt* with a single header line. In Chapters 7 and 8 we will read some of these complicated and extensive files, which are either binary or text files and usually have long headers describing the contents of the files. At this point, let us have a look at a variant of text files that contains not only one or more header lines but also unusual data types such as date and time, in a non-decimal format. We use



Movie  
2.3

the function `textscan` to perform this task. The MATLAB workspace should first be cleared by typing

```
clear
```

after the prompt in the Command Window. MATLAB can now import the data from the file *geochem.txt* using the `textscan` command.

```
fid = fopen('geochem.txt');
C = textscan(fid, '%u %f %f', 'Headerlines', 1, 'CollectOutput', 1);
fclose(fid);
```

This script opens the file *geochem.txt* for *read only* access using `fopen` and defines the file identifier `fid`, which is then used to read the text from the file using `textscan` and to write it into the cell array `C`. The character string `%u %f %f` defines the conversion specifiers enclosed in single quotation marks, where `%u` stands for the 32-bit unsigned integer output class and `%f` stands for a 64-bit double-precision floating-point number. The parameter `Headerlines` is set to 1, which means that a single header line is ignored while reading the file. If the parameter `CollectOutput` is 1 (i.e., is true), `textscan` concatenates output cells with the same data type into a single array. The function `fclose` closes the file defined by `fid`. The array `C` is a cell array, which is a data type with indexed containers called cells (see Section 2.5). The advantage of this data type is that it can store data of various types and sizes, such as character strings, double-precision numbers, and images in a single variable such as `C`. Typing

```
C
```

yields

```
C =
    [6x1 uint32]    [6x2 double]
```

indicating that `C` contains a 6-by-1 32-bit unsigned integer array, which is the sample ID, and a 6-by-1 double-precision array, which represents the percentages of carbon and sulfur in each sample. We can access the contents of the cells in `C` by typing

```
data1 = C{1}
data2 = C{2}
```

which yields

```
data1 =
    101
    102
```



```

103
104
105
106

data2 =
    0.3657    0.0636
    0.2208    0.1135
    0.5353    0.5191
    0.5009    0.5216
    0.5415         NaN
    0.5010         NaN

```

We now concatenate the two cells into one double-precision array `data`. First, we have to change the class of `C{1}` into `double` or the class of the entire array `data` will be `uint32`. Typing

```

data(:,1) = double(C{1})
data(:,2:3) = C{2}

```

yields

```

101.0000    0.3657    0.0636
102.0000    0.2208    0.1135
103.0000    0.5353    0.5191
104.0000    0.5009    0.5216
105.0000    0.5415         NaN
106.0000    0.5010         NaN

```

The format of the data is as expected.

The next examples demonstrate how to read the file `geophys.txt`, which contains a single header line but also the date (in an *MM/DD/YY* format) and time (in an *HH:MM:SS.SS* format). We again use `textscan` to read the file,

```

clear

fid = fopen('geophys.txt');
data = textscan(fid,'%u %f %f %f %s %s','Headerlines',1);
fclose(fid);

```

where we skip the header, read the first column (the sample ID) as a 32-bit unsigned integer (`uint32`) with specifier `%u`, the next three columns *X*, *Y*, and *Z* as 64-bit double-precision floating-point numbers (`double`) with specifier `%f`, and then the date and time as character strings with specifier `%s`. We then convert the date and time to serial numbers, where a serial date number of 1 corresponds to *Jan-1-0000*. The year *0000* is merely a reference point and is not intended to be interpreted as a real year.

```
data_date_serial = datenum(data{5});
data_time_serial = datenum(data{6});
```

Finally, we can convert the date and time serial numbers into a data and time array by typing

```
data_date = datevec(data_date_serial)
data_time = datevec(data_time_serial)
```

which yields

```
data_date =
    2013    11    18     0     0     0
    2013    11    18     0     0     0
    2013    11    18     0     0     0
    2013    11    18     0     0     0
    2013    11    18     0     0     0

data_time =
    1.0e+03 *
    2.0130    0.0010    0.0010    0.0100    0.0230    0.0091
    2.0130    0.0010    0.0010    0.0100    0.0230    0.0102
    2.0130    0.0010    0.0010    0.0100    0.0230    0.0504
    2.0130    0.0010    0.0010    0.0100    0.0240    0.0051
    2.0130    0.0010    0.0010    0.0100    0.0240    0.0233
```

The first three columns of the array `data_date` contain the year, month and day. The fourth to sixth columns of the array `data_time` contain the hour, minute and second.

We can also write data to a formatted text file using `fprintf`. As an example we again load the data from `geochem.txt` after we have commented out the first line and have replaced `-999` with `NaN`. Instead of using `load geochem.txt`, we can type

```
clear

data = load('geochem.txt');
```

to load the contents of the text file into a double-precision array `data`. We write the data to a new text file `geochem_formatted.txt` using `fprintf`. Since the function `fprintf` writes all elements of the array `data` to the file in column order we need to transpose the data before we save it.

```
data = data';
```

We first open the file using the permission `w` for *writing*, and discard the existing contents. We then write `data` to this file using the formatting operators `%u` for unsigned integers and `%6.4f` for fixed-point numbers with a field width of six characters and four digits after the decimal point. The

control character `\n` denotes a new line after each line of three numbers.

```
fid = fopen('geochem_formatted.txt','w');
fprintf(fid,'%u %6.4f %6.4f\n',data);
fclose(fid);
```

We can view the contents of the file by typing

```
edit geochem_formatted.txt
```

which opens the file *geochem\_formatted.txt*

```
101    0.3657    0.0636
102    0.2208    0.1135
103    0.5353    0.5191
104    0.5009    0.5216
105    0.5415         NaN
106    0.5010         NaN
```

in the MATLAB Editor. The format of the data is as expected.

## 2.7 Control Flow

Control flow in computer science helps to control the order in which computer code is evaluated. The most important kinds of control flow statements are count-controlled loops such as `for` loops and conditional statements such as `if-then` constructs. Since in this book we do not deal with the programming capabilities of MATLAB in any depth, the following introduction to the basics of control flow is rather brief and omits certain important aspects of efficient programming, such as the pre-allocation of memory prior to using `for` loops, and instructions on how the use of `for` loops can be avoided by vectorization of the MATLAB code. This introduction is instead limited to the two most important kinds of control flow statements: the aforementioned `for` loops and the `if-then` constructs. Readers interested in MATLAB as a programming environment are advised to read the more detailed chapters on control flow in the MATLAB documentation (MathWorks 2014a and c).

The `for` loops, as the first example of a MATLAB language statement, execute a series of commands between `for` and `end` a specified number of times. As an example we use such a loop to multiply the elements of an array `A` by 10, round the result to the nearest integer, and store the result in `B`.

```
clear

rng(0)
A = rand(10,1)
for i = 1 : 10
    B(i,1) = round(10 * A(i));
```

```
end
B
```

which yields

```
A =
    0.8147
    0.9058
    0.1270
    0.9134
    0.6324
    0.0975
    0.2785
    0.5469
    0.9575
    0.9649

B =
     8
     9
     1
     9
     6
     1
     3
     5
    10
    10
```

The result is as expected. We can expand the experiment by using a nested `for` loop to create a 2D array `B`.

```
rng(0)
A = rand(10,3)
for i = 1 : 10
    for j = 1 : 3
        B(i,j) = round(10 * A(i,j));
    end
end
B
```

which yields

```
A =
    0.8147    0.1576    0.6557
    0.9058    0.9706    0.0357
    0.1270    0.9572    0.8491
    0.9134    0.4854    0.9340
    0.6324    0.8003    0.6787
    0.0975    0.1419    0.7577
    0.2785    0.4218    0.7431
    0.5469    0.9157    0.3922
    0.9575    0.7922    0.6555
    0.9649    0.9595    0.1712
```

```

B =
     8     2     7
     9    10     0
     1    10     8
     9     5     9
     6     8     7
     1     1     8
     3     4     7
     5     9     4
    10     8     7
    10    10     2

```

This book tries to make all of the recipes independent of the actual dimensions of the data. This is achieved by the consistent use of `size` and `length` to determine the size of the data instead of using fixed numbers such as the `30` and `3` in the above example (Section 2.4).

```

rng(0)
A = rand(10,3)
for i = 1 : size(A,1)
    for j = 1 : size(A,2)
        B(i,j) = round(10 * A(i,j));
    end
end
B

```

When working with larger data sets with many variables one might occasionally wish to automate array manipulations such as those described in Section 2.4. Let us assume, for example, that we want to replace all `NaNs` in all variables in the memory with `-999`. We first create a collection of four variables, each of which contains a single `NaN`.

```

clear

rng(0)
A = rand(3,3); A(2,1) = NaN
BC = rand(2,4); BC(2,2) = NaN
DE = rand(1,2); DE(1,1) = NaN
FG = rand(3,2); FG(2,2) = NaN

```

We list the variables in the workspace using `whos` and store this list in `variables`.

```
variables = whos;
```

We then use a `for` loop to store the content of each variable in `v` using `eval` and then locate the `NaNs` in `v` using `isnan` (Section 2.4) and replace them with `-999`. The function `eval` executes a MATLAB expression stored in a text string. We assign the value of `v` to the variable in the base workspace and then clear the variables `i`, `v` and `variables`, which are no longer needed.

```

for i = 1 : size(variables,1)
    v = eval(variables{i});
    v(isnan(v)==1) = -999;
    assignin('base',variables{i},v);
    eval(variables{i})
end

clear i v variables

```

Comparing the variables before and after the replacement of the NaNs with -999 reveals that the script works well and that we have successfully manipulated our data.

The second important statements to control the flow of a script (apart from `for` loops) are `if-then` constructs, which evaluate an expression and then execute a group of instructions if the expression is true. As an example we compare the value of two scalars `A` and `B`.

```

clear

A = 1
B = 2
if A < B
    disp('A is less than B')
end

```

which yields

```
A is less than B
```

The script first evaluates whether `A` is less than `B` and, if it is, displays the message `A is less than B` in the Command Window. We can expand the `if-then` construct by introducing `else`, which provides an alternative statement if the expression is not true.

```

A = 1
B = 2
if A < B
    disp('A is less than B')
else
    disp('A is not less than B')
end

```

which yields

```
A is less than B
```

Alternatively, we can use `elseif` to introduce a second expression to be evaluated.

```
A = 1
```

```

B = 2
if A < B
    disp('A is less than B')
elseif A >= B
    disp('A is not less than B')
end

```

The `for` loops and `if-then` constructs are extensively used in the following chapters of the book. For other aspects of programming, please refer to the MATLAB documentation (MathWorks 2014a and c).

## 2.8 Scripts and Functions

MATLAB is a powerful programming language. All files containing MATLAB code use `.m` as an extension and are therefore called *M-files*. These files contain ASCII text and can be edited using a standard text editor. However, the built-in Editor color-highlights various syntax elements such as comments in green, keywords such as *if*, *for* and *end* in blue, and character strings in pink. This syntax highlighting facilitates MATLAB coding.

MATLAB uses two types of M-files: scripts and functions. Whereas scripts are a series of commands that operate on data in the workspace, functions are true algorithms with input and output variables. The advantages and disadvantages of both types of M-file will now be illustrated by an example. We first start the Editor by typing

```
edit
```

This opens a new window named *untitled*. Next, we generate a simple MATLAB script by typing a series of commands to calculate the average of the elements of a data array `x`.

```

[m,n] = size(x);
if m == 1
    m = n;
end
sum(x)/m

```

The first line of the `if-then` construct yields the dimensions of the variable `x` using the command `size`. In our example `x` should be either a column vector, i.e., an array with a single column and dimensions `(m,1)`, or a row vector, i.e. an array with a single row and dimensions `(1,n)`. The `if` statement evaluates a logical expression and executes a group of commands if this expression is true. The `end` keyword terminates the last group of commands. In the example the `if-then` construct picks either `m` or `n` depending on whether `m==1` is false or true. Here, the double equal sign `'=='` makes element by element

comparisons between the variables (or numbers) to the left and right of the equal signs and returns an array of the same size, made up of elements set to logical `1` where the relationship is true and to logical `0` where it is not true. In our example `m==1` returns `1` if `m` equals `1` and `0` if `m` equals any other value. The last line of the `if-then` construct computes the average by dividing the sum of elements by `m` or `n`. We do not use a semicolon here in order to allow the output of the result. We can now save our new M-file as *average.m* and type

```
clear

x = [3 6 2 -3 8];
```

in the Command Window to define an example array `x`. We then type

```
average
```

without the extension *.m* to run our script and obtain the average of the elements of the array `x` as output.

```
ans =
    3.2000
```

After typing

```
whos
```

we see that the workspace now contains

Name	Size	Bytes	Class	Attributes
ans	1x1	8	double	
m	1x1	8	double	
n	1x1	8	double	
x	1x5	40	double	

The listed variables are the example array `x` and the output of the function `size`, `m` and `n`. The result of the operation is stored in the variable `ans`. Since the default variable `ans` might be overwritten during one of the succeeding operations, we need to define a different variable. Typing

```
a = average
```

however, results in the error message

```
??? Attempt to execute SCRIPT average as a function.
```

We can obviously not assign a variable to the output of a script. Moreover, all variables defined and used in the script appear in the workspace; in our example these are the variables `m` and `n`. Scripts contain sequences of



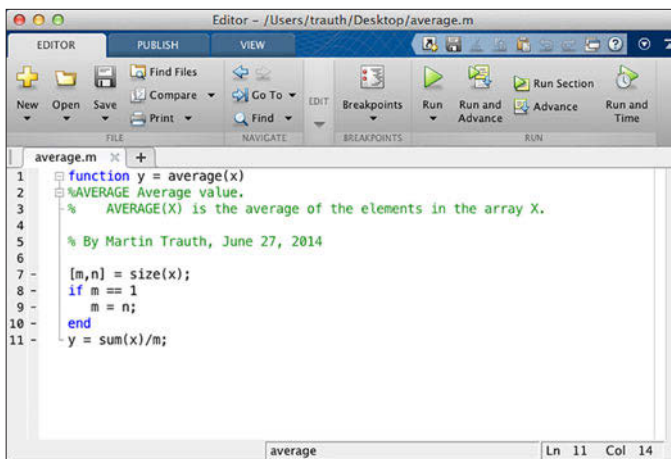
commands that are applied to variables in the workspace. MATLAB functions, however, allow inputs and outputs to be defined. They do not automatically import variables from the workspace. To convert the above script into a function we need to introduce the following modifications (Fig. 2.3):

```
function y = average(x)
% AVERAGE Average value.
% AVERAGE(X) is the average of the elements in the array X.

% By Martin Trauth, June 27, 2014

[m,n] = size(x);
if m == 1
    m = n;
end
y = sum(x)/m;
```

The first line now contains the keyword `function`, the function name `average`, the input `x` and the output `y`. The next two lines contain comments, as indicated by the percent sign, separated by an empty line. The second comment line contains the author's name and the version of the M-file. The rest of the file contains the actual operations. The last line now defines the value of the output variable `y`, and this line is terminated by a semicolon to suppress the display of the result in the Command Window. Next we type



**Fig. 2.3** Screenshot of the MATLAB Editor showing the function `average`. The function starts with a line containing the keyword `function`, the name of the function `average`, the input variable `x`, and the output variable `y`. The subsequent lines contain the output for `help average`, the copyright and version information, and also the actual MATLAB code for computing the average using this function.

```
help average
```

which displays the first block of contiguous comment lines. The first executable statement (or blank line in our example) effectively ends the help section and therefore the output of *help*. Now we are independent of the variable names used in our function. The workspace can now be cleared and a new data vector defined.

```
clear
```

```
data = [3 6 2 -3 8];
```



Movie  
2.4

Our function can then be run by the statement

```
result = average(data);
```

This clearly illustrates the advantages of functions compared to scripts. Typing

```
whos
```

results in

Name	Size	Bytes	Class	Attributes
data	1x5	40	double	
result	1x1	8	double	

revealing that all variables used in the function do not appear in the workspace. Only the input and output as defined by the user are stored in the workspace. The M-files can therefore be applied to data as if they were real functions, whereas scripts contain sequences of commands that are applied to the variables in the workspace. If we want variables such as *m* and *n* to also appear in the memory they must be defined as *global* variables in both the function and the workspace, otherwise they are considered to be *local* variables. We therefore add one line to the function *average* with the command *global*:

```
function y = average(x)
% AVERAGE    Average value.
%    AVERAGE(X) is the average of the elements in the array X.

% By Martin Trauth, June 27, 2014

global m n
[m,n] = size(x);
if m == 1
    m = n;
end
y = sum(x)/m;
```

We now type

```
global m n
```

in the Command Window. After running the function as described in the previous example we find the two variables `m` and `n` in the workspace. We have therefore transferred the variables `m` and `n` between the function `average` and the workspace.

## 2.9 Basic Visualization Tools

MATLAB provides numerous routines for displaying data as graphics. This section introduces the most important graphics functions. The graphics can be modified, printed and exported to be edited with graphics software other than MATLAB. The simplest function producing a graph of a variable `y` versus another variable `x` is `plot`. First, we define two one-dimensional arrays `x` and `y`, where `y` is the sine of `x`. The array `x` contains values between 0 and  $2\pi$  with  $\pi/10$  increments, whereas `y` is the element-by-element sine of `x`.

```
clear

x = 0 : pi/10 : 2*pi;
y = sin(x);
```

These two commands result in two one-dimensional arrays with 21 elements each, i.e., two 1-by-21 arrays. Since the two arrays `x` and `y` have the same length, we can use `plot` to produce a linear 2D graph of `y` against `x`.

```
plot(x,y)
```

This command opens a *Figure Window* named *Figure 1* with a gray background, an *x*-axis ranging from 0 to 7, a *y*-axis ranging from -1 to +1 and a blue line. We may wish to plot two different curves in a single plot, for example the sine and the cosine of `x` in different colors. The command

```
x = 0 : pi/10 : 2*pi;
y1 = sin(x);
y2 = cos(x);

plot(x,y1,'--',x,y2,'-')
```

creates a dashed blue line displaying the sine of `x` and a solid red line representing the cosine of this array (Fig. 2.4). If we create another plot, the window *Figure 1* will be cleared and a new graph displayed. The command `figure`, however, can be used to create a new figure object in a new window.

```

plot(x,y1,'--')
figure
plot(x,y2,'-')

```

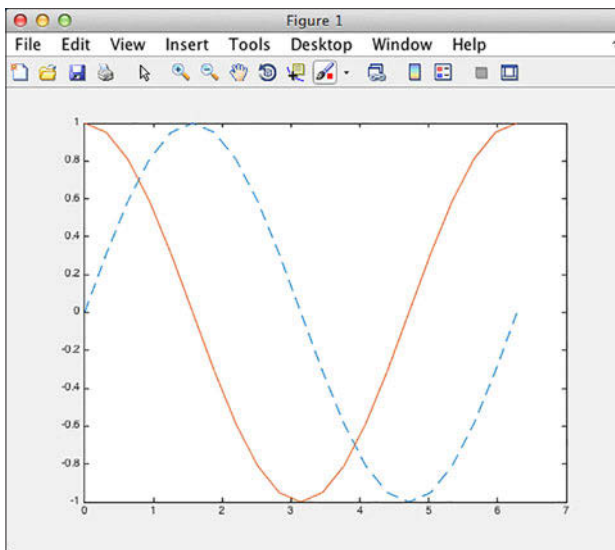
Instead of plotting both lines in one graph simultaneously, we can also plot the sine wave, hold the graph and then plot the second curve. The command `hold` is particularly important for displaying data while using different plot functions, for example if we wish to display the sine of  $x$  as a line plot and the cosine of  $x$  as a bar plot.

```

plot(x,y1,'r--')
hold on
bar(x,y2)
hold off

```

This command plots  $y_1$  versus  $x$  as a dashed red line using `'r--'`, whereas  $y_2$  versus  $x$  is shown as a group of blue vertical bars. Alternatively, we can plot both graphics in the same Figure Window but in different plots using `subplot`. The syntax `subplot(m,n,p)` divides the Figure Window into an  $m$ -by- $n$  array of display regions and makes the  $p$ th display region active.



**Fig. 2.4** Screenshot of the MATLAB Figure Window showing two curves in different colors and line types. The Figure Window allows editing of all elements of the graph after selecting *Edit Plot* from the *Tools* menu. Double clicking on the graphics elements opens an options window for modifying the appearance of the graphics. The graphics can be exported using *Save as* from the *File* menu. The command *Generate Code* from the *File* menu creates MATLAB code from an edited graph.

```
subplot(2,1,1), plot(x,y1,'r--')
subplot(2,1,2), bar(x,y2)
```

For example the Figure Window is divided into two rows and one column. The 2D linear plot is displayed in the upper half of the Figure Window and the bar plot appears in the lower half. It is recommended that all Figure Windows be closed before proceeding to the next example. Subsequent plots would replace the graph in the lower display region only, or in other words, the last generated graph in a Figure Window. Alternatively, the command

```
clf
```

clears the current figure. This command can be used in larger MATLAB scripts after using the function `subplot` for multiple plots in a Figure Window.

An important modification to graphics is the scaling of the axis. By default, MATLAB uses axis limits close to the minima and maxima of the data. Using the command `axis`, however, allows the scale settings to be changed. The syntax for this command is simply `axis([xmin xmax ymin ymax])`. The command

```
plot(x,y1,'r--')
axis([0 pi -1 1])
```

sets the limits of the  $x$ -axis to 0 and  $\pi$ , whereas the limits of the  $y$ -axis are set to the default values  $-1$  and  $+1$ . Important options of `axis` are

```
plot(x,y1,'r--')
axis square
```

which makes the  $x$ -axis and  $y$ -axis the same length, and

```
plot(x,y1,'r--')
axis equal
```

which makes the individual tick mark increments on the  $x$ -axis and  $y$ -axis the same length. The function `grid` adds a grid to the current plot, whereas the functions `title`, `xlabel` and `ylabel` allow a title to be defined and labels to be applied to the  $x$ - and  $y$ -axes.

```
plot(x,y1,'r--')
title('My first plot')
xlabel('x-axis')
ylabel('y-axis')
grid
```

These are a few examples how MATLAB functions can be used to edit the plot in the Command Window. More graphics functions will be introduced

in the following chapters of this book.

## 2.10 Generating Code to Recreate Graphics

MATLAB supports various ways of editing all objects in a graph interactively using a computer mouse. First, the *Edit Plot* mode of the Figure Window needs to be activated by clicking on the arrow icon or by selecting *Edit Plot* from the *Tools* menu. The Figure Window also contains some other options, such as *Rotate 3D*, *Zoom* or *Insert Legend*. The various objects in a graph, however, are selected by double-clicking on the specific component, which opens the *Property Editor*. The Property Editor allows changes to be made to many features (or properties) of the graph such as axes, lines, patches and text objects.

The *Generate Code* option enables us to automatically generate the MATLAB code of a figure to recreate a similar graph with different data. We use a simple plot to illustrate the use of the Property Editor and the Generate Code option to recreate a graph.

```
clear

x = 0 : pi/10 : 2*pi;
y1 = sin(x);
plot(x,y1)
```

The default layout of the graph is that of Figure 2.4. Clicking on the arrow icon in the *Figure Toolbar* enables the Edit Plot mode. The selection handles of the graph appear, identifying the objects that are activated. Double-clicking an object in a graph opens the *Property Editor*.

As an example we can use the Property Editor to change various properties of the graph. Double-clicking the gray background of the Figure Window gives access to properties such as Figure Name, the Colormap used in the figure, and the Figure Color. We can change this color to light blue represented by the light blue square in the 4th row and 8rd column of the color chart. Moving the mouse over this square displays the RGB color code [0.68 0.92 1] (see Chapter 8 for more details on RGB colors). Activating the blue line in the graph allows us to change the line thickness to 2.0 and select a 15-point square marker. We can deactivate the Edit Plot mode of the Figure Window by clicking on the arrow icon in the Figure Toolbar.

After having made all necessary changes to the graph, the corresponding commands can even be exported by selecting Generate Code from the File menu of the Figure Window. The generated code displays in the MATLAB Editor.

```

function createfigure(X1, Y1)
%CREATEFIGURE(X1, Y1)
% X1: vector of x data
% Y1: vector of y data

% Auto-generated by MATLAB on 27-Jun-2014 13:28:13

% Create figure
figure1 = figure('Color',[0.68 0.92 1]);

% Create axes
axes1 = axes('Parent',figure1,'ColorOrderIndex',2);
box(axes1,'on');
hold(axes1,'on');

% Create plot
plot(X1,Y1,'MarkerSize',15,'Marker','square','LineWidth',2);

```

We can then rename the function `createfigure` to `mygraph` and save the file as *mygraph.m*.

```

function mygraph(X1, Y1)
%MYGRAPH(X1,Y1)
% X1: vector of x data
% Y1: vector of y data
(cont'd)

```

The automatically-generated graphics function illustrates how graphics are organized in MATLAB. The function `figure` first opens a Figure Window. Using `axes` then establishes a coordinate system, and using `plot` draws the actual line object. The Figure section in the function reminds us that the light-blue background color of the Figure Window is represented by the RGB color coding `[0.68 0.92 1]`. The Plot section reveals the square marker symbol used and the line width of 2 points.

The newly-created function `mygraph` can now be used to plot a different data set. We use the above example and

```

clear

x = 0 : pi/10 : 2*pi;
y2 = cos(x);
mygraph(x,y2)

```

The figure shows a new plot with the same layout as the previous plot. The *Generate Code* function of MATLAB can therefore be used to create templates for graphics that can be used to generate plots of multiple data sets using the same layout.

Even though MATLAB provides abundant editing facilities and the *Generate Code* function even allows the generation of complex templates



for graphics, a more practical way to modify a graph for presentations or publications is to export the figure and import it into a different software such as CorelDraw or Adobe Illustrator. MATLAB graphics are exported by selecting the command *Save as* from the File menu or by using the command `print`. This function exports the graphics, either as a raster image (e.g., JPEG or GIF) or as a vector file (e.g., EPS or PDF), into the working directory (see Chapter 8 for more details on graphic file formats). In practice, the user should check the various combinations of export file formats and the graphics software used for final editing of the graphics. Readers interested in advanced visualization techniques with MATLAB are directed to the sister book *MATLAB and Design Recipes for Earth Sciences* (Trauth and Sillmann 2012).

## 2.11 Publishing M-Files

Another useful feature of the software is the option to publish reports on MATLAB projects in various file formats such as HTML, XML, LaTeX and many others. This feature enables you to share your results with colleagues who may or may not have the MATLAB software. The published code includes formatted commentary on the code, the actual MATLAB code, and all results of running the code including the output to the Command Window and all graphics created or modified by the code. To illustrate the use of the publishing feature we create a simple example of a commented MATLAB code to compute the sine and cosine of a time vector and display the results as two separate figures.

We start the Editor by typing `edit` in the Command Window, which opens a new window named `untitled`. An M-file to be published starts with a document title at the top of the file, followed by some comments that describe the contents and the version of the script. The subsequent contents of the file include sections of MATLAB code and comments, separated by the double percent signs `%%`. Whereas single percent signs `%` are known (from Section 2.8) to initiate comments in MATLAB, we now use double percent signs `%%` that indicate the start of new code sections in the Editor. The *code sections* feature, previously also known as *code cells* or *cell mode*, is a feature in MATLAB that enables you to evaluate blocks of commands called *sections* by using the buttons *Run*, *Run and Advance*, *Run Section*, *Advance*, and *Run and Time* on the Editor Toolstrip to evaluate either the entire script or parts of the script.

```
%% Example for Publishing M-Files
% This M-file illustrates the use of the publishing
% feature of MATLAB.
```



% By Martin Trauth, June 27, 2014

```
%% Sine Wave
% We define a time vector t and compute the sine y1 of t.
% The results are displayed as linear 2D graph y1 against x.
x = 0 : pi/10 : 2*pi;
y1 = sin(x);
plot(x,y1)
title('My first plot')
xlabel('x-axis')
ylabel('y-axis')

%% Cosine Wave
% Now we compute the cosine y2 of the same time vector and
% display the results.
y2 = sin(x);
plot(x,y2)
title('My first plot')
xlabel('x-axis')
ylabel('y-axis')

%%
% The last comment is separated by the double percent sign
% without text. This creates a comment in a separate cell
% without a subheader.
```

We save the M-file as *myproject.m* and click the *Publish* button in the Publish Toolstrip. The entire script is now evaluated and the Figure Windows pop up while the script is running. Finally, a window opens up that shows the contents of the published M-file. The document title and subheaders are shown in a red font whereas the comments are in black fonts. The file includes a list of contents with jump links to proceed to the chapters of the file. The MATLAB commands are displayed on gray backgrounds but the graphics are embedded in the file without the gray default background of Figure Windows. The resulting HTML file can be easily included on a course or project webpage. Alternatively, the HTML file and included graphics can be saved as a PDF-file and shared with students or colleagues.



Movie  
2.6

## 2.12 Creating Graphical User Interfaces

Almost all the methods of data analysis presented in this book are in the form of MATLAB scripts, i.e., series of commands that operate on data in the workspace (Section 2.8). Only in a few cases are the algorithms implemented in functions such as `canc` for adaptive filtering (Section 6.8) or `minput` for digitizing from the screen (Section 8.7). The MATLAB commands provided by The MathWorks, Inc., however, are mostly functions, i.e., algorithms with input and output variables. The most convenient variants

of these functions are those with a *graphical user interface* (GUI). A GUI in MATLAB is a graphical display in one or more windows containing controls (or components) that enable the user to perform interactive tasks without typing commands in the Command Window or writing a script in the Editor. These components include pull-down menus, push buttons, sliders, text input fields and more. The GUI can read and write data files as well as performing many types of computation and displaying the results in graphics.

The manual entitled *MATLAB Creating Graphical User Interfaces* (MathWorks 2014b) provides a comprehensive guide to the creation of GUIs with MATLAB. Within this manual, however, the section on *Create a Simple GUIDE GUI* demonstrates a rather complex example with many interactive elements instead of providing the simplest possible example of a GUI. The following text therefore provides a very simple example of a GUI that computes and displays a Gaussian function for a mean and a standard deviation that can be defined by the user. Creating such a simple GUI with MATLAB requires two steps: the first step involves designing the layout of the GUI, and the second step involves adding functions to the components of the GUI. The best way to create a graphical user interface with MATLAB is using the *GUI Design Environment* (GUIDE). We start GUIDE by typing

`guide`

in the Command Window. Calling GUIDE opens the *GUIDE Quick Start* dialog where we can choose to open a previously created GUI or create a new one from a template. From the dialog we choose the GUIDE template *Blank GUI (Default)* and click OK, after which the *GUIDE Layout Editor* starts. First, we enable *Show names in component palette* in the *GUIDE Preferences* under the *File* menu and click OK. Second, we select *Grid and Rulers* from the *Tools* menu and enable *Show rulers*. The *GUIDE Layout Editor* displays an empty layout with dimensions of 670-by-388 pixels. We resize the layout to 500-by-300 pixels by clicking and dragging the lower right corner of the GUI.

Next, we place components such as static text, edit text, and axes onto the GUI by choosing the corresponding controls from the component palette. In our example we place two *Edit Text* areas on the left side of the GUI, along with a *Static Text* area containing the title *Mean*, with *Standard Deviation* above it. Double clicking the static text areas, the *Property Inspector* comes up in which we can modify the properties of the components. We change the *String* of the static text areas to *Mean* and *Standard Deviation*. We can also change other properties, such as the *FontName*, *FontSize*, *BackgroundColor*, and *HorizontalAlignment* of the text. Instead of the default *Edit Text* content

of the edit text areas we choose 0 for the mean and 1 for the standard deviation text area. We then place an axis with dimensions of 250-by-200 pixels to the right of the GUI. Next, we save and activate the GUI by selecting *Run* from the *Tools* menu. GUIDE displays a dialog box with the question *Activating will save changes ...?*, where we click *Yes*. In the following *Save As* dialog box, we define a *FIG*-file name such as *gaussiantool.fig*.

GUIDE then saves this figure file together with the corresponding MATLAB code in a second file named *gaussiantool.m*. Furthermore, the MATLAB code is opened in the Editor and the default GUI is opened in a Figure Window with no menu or toolbar (Fig. 2.5). As we can see, GUIDE has automatically programmed the code of our GUI layout, including an initialization code at the beginning of the file that we should not edit. This code is included in the main routine named *gaussiantool*. The file also contains other functions called by *gaussiantool*, for instance the function *gaussiantool\_Opening\_Fcn* (executed before *gaussiantool* is made visible), *gaussiantool\_OutputFcn* (sending output to the command line, not used here), *edit1\_CreateFcn* and *edit2\_CreateFcn* (initializing the edit text areas when they are created), and *edit1\_Callback* and *edit2\_Callback* (accepting text input and returning this input either as text or as a double-precision number).

We now add code to our GUI *gaussiantool*. First, we add initial values for the global variables *mmean* and *mstd* in the opening function *gaussiantool\_Opening\_Fcn* by adding the following lines after the last comment line marked by *%* in the first column:

```
global mmean mstd
mmean = 0;
mstd = 1;
```

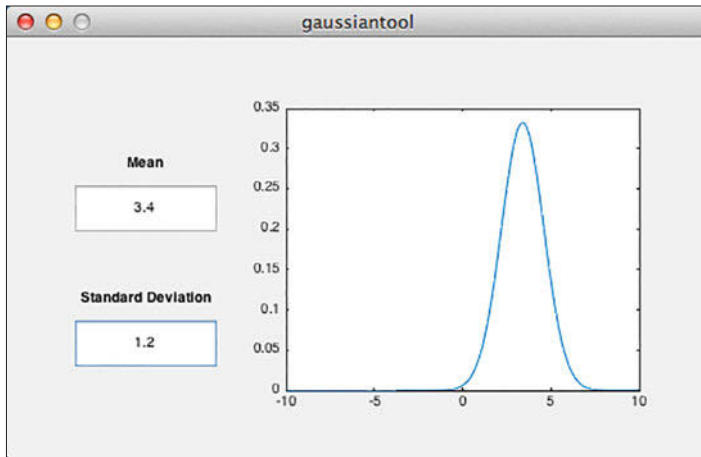
The two variables must be global because they are used in the callbacks that we edit next (as in Section 2.8). The first of these callbacks *edit1\_Callback* gets three more lines of code after the last comment line:

```
global mmean
mmean = str2double(get(hObject,'String'));
calculating_gaussian(hObject, eventdata, handles)
```

The first line defines the global variable *mmean*, which is then obtained by converting the text input into double precision with *str2double* in the second line. The function *edit1\_Callback* then calls the function *calculating\_gaussian*, which is a new function at the end of the file. This function computes and displays the Gaussian function with a mean value of *mmean* and a standard deviation of *mstd*.



Movie  
2.7



**Fig. 2.5** Screenshot of the graphical user interface (GUI) `gaussiantool` for plotting a Gaussian function with a given mean and standard deviation. The GUI allows the values of the mean and standard deviation to be changed in order to update the graphics on the right. The GUI has been created using the MATLAB *GUI Design Environment* (GUIDE).

```
function calculating_gaussian(hObject, eventdata, handles)
% hObject    handle to edit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
global mmean mstd
x = -10 : 0.1 : 10;
y = normpdf(x, mmean, mstd);
plot(x,y)
```

The second callback `edit2_Callback` picks the value of the standard deviation `mstd` from the second Edit Text area, which is then also used by the function `calculating_gaussian`.

```
global mstd
mstd = str2double(get(hObject,'String'));
calculating_gaussian(hObject, eventdata, handles)
```

After saving the file `gaussiantool.m` we can run the new GUI by typing

```
gaussiantool
```

in the Command Window. The GUI starts where we can change the values of the mean and the standard deviation, then press return. The plot on the right is updated with each press of the return key. Using

```
edit gaussiantool
guide gaussiantool
```

we can open the GUI code and Figure Window for further edits. Such GUIs allow a very direct and intuitive handling of functions in MATLAB that can also include animations such as the one used in [canctool](#) (Section 6.8), and the display of an audio-video signal. On the other hand, however, GUIs always require an interaction with the user who needs to click push buttons, move sliders and edit text input fields while the data is being analyzed. The automatic processing of large quantities of data is therefore usually carried out using scripts and functions with no graphical user interface.

## Recommended Reading

- Attaway S (2013) MATLAB: A Practical Introduction to Programming and Problem Solving. Elsevier, New York
- Etter DM, Kuncicky DC, Moore H (2014) Introduction to MATLAB. Prentice Hall, New Jersey
- Gilat A (2010) MATLAB: An Introduction with Applications. John Wiley & Sons, New York
- Hanselman DC, Littlefield BL (2012) Mastering MATLAB 8. Prentice Hall, New Jersey
- MathWorks (2014a) MATLAB Primer. The MathWorks, Inc., Natick, MA
- MathWorks (2014b) MATLAB Creating Graphical User Interfaces. The MathWorks, Inc., Natick, MA
- MathWorks (2014c) MATLAB Programming Fundamentals. The MathWorks, Inc., Natick, MA
- Palm WJ (2010) Introduction to MATLAB 7 for Engineers. McGraw-Hill, New York
- Quarteroni A, Saleri F, Gervasio P (2014) Scientific Computing with MATLAB and Octave – 4th Edition. Springer, Berlin Heidelberg New York
- Trauth MH, Sillmann E (2012) MATLAB and Design Recipes for Earth Sciences. Springer, Berlin Heidelberg New York

MATLAB® Recipes for Earth Sciences

Trauth, M.

2015, XIV, 427 p. 116 illus., 20 illus. in color., Hardcover

ISBN: 978-3-662-46243-0