

2 Introduction to MATLAB

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, to the setup on the computer (Section 2.2), the syntax (Section 2.3), data

input and output (Sections 2.4 and 2.5), programming (Section 2.6), and visualization (Section 2.7). Advanced sections are also included on generating M-files to regenerate graphs (Section 2.8) and on publishing M-files (Section 2.9). It is recommended to go through the entire chapters in order to obtain a good knowledge of the software before proceeding to the following chapter. A more detailed introduction can be found in the *MATLAB 7 Getting Started Guide* (The MathWorks 2010) which is available in print form, online and as PDF file.

In this book we use MATLAB Version 7.10 (Release 2010a), the Image Processing Toolbox Version 7.0, the Mapping Toolbox Version 3.1, the Signal Processing Toolbox Version 6.13, the Statistics Toolbox Version 7.3 and the Wavelet Toolbox Version 4.5.

2.2 Getting Started

The software package comes with extensive documentation, tutorials and examples. The first three chapters of the book *MATLAB 7 Getting Started Guide* (The MathWorks 2010) are directed at beginners. The chapters on programming, creating graphical user interfaces (GUI) and development environments are aimed at more advanced users. Since *MATLAB 7 Getting Started Guide* 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. The *Command History* panel records all operations previously typed into the Command Window and enables them to be recalled by the user. In this book we mainly use the Command Window and the built-in *Editor*, which can be launched by typing

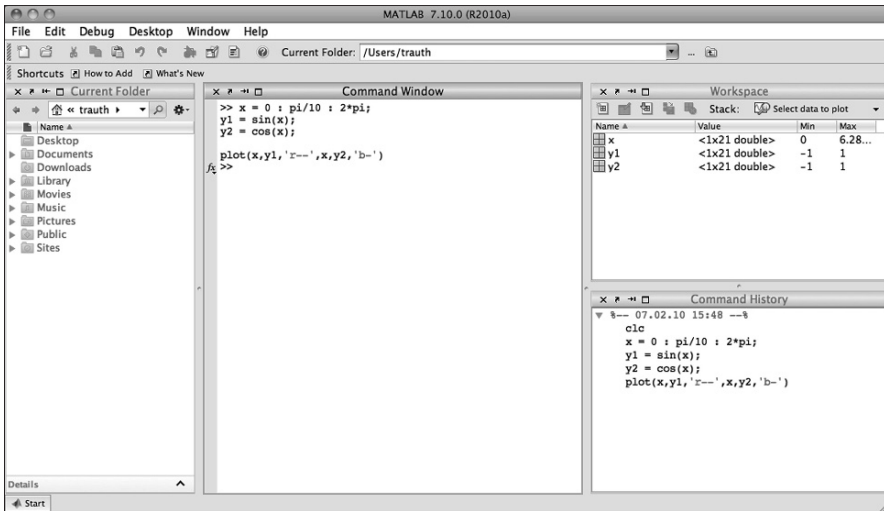


Fig. 2.1 Screenshot of the MATLAB default desktop layout including the *Current Folder* (left in the figure), the *Command Window* (center), the *Workspace* (upper right) and *Command History* (lower right) panels. This book uses only the *Command Window* and the built-in *Editor*, which can be called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

`edit`

or by selecting the *Editor* from the *Desktop* menu. 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 organized in directories on the hard disk. The default search path includes only the MATLAB directory that has been created by the installer in the applications folder and the default working directory. To see which directories are in the search path or to add new directories, select *Set Path* from the *File* menu, 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 matrix or array is a scalar. Matrices with one column or row are vectors, time series or other one-dimensional data fields. An m -by- n matrix or 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. 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 matrix 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 matrix

```
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 matrices, such as digital elevation models consisting of thousands or millions of elements. To suppress the display of a matrix 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 matrix 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 matrix 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 matrices. 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 matrix 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
<code>A</code>	4x4	128	double	
<code>ans</code>	1x4	32	double	
<code>b</code>	1x1	8	double	

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 matrices and lower-case letters for vectors and scalars. We could now delete the contents of the variable `ans` by typing

```
clear ans
```

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

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

simply yields the matrix element located in the third row and second column, which is 9. The matrix indexing therefore follows the rule (*row, col-*

umn). We can use this to replace single or multiple matrix elements. As an example, we type

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

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

```
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 matrix A . The colon operator also has several other uses in MATLAB, for instance as a shortcut for entering matrix elements such as

```
c = 0 : 10
```

which creates a row vector 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 matrices 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 vector 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 axes 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
    75    136   -76    39
    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, right, /, and left, \, representing different transformations. 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 an array 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 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 data for 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 file and assigns the matrix 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 com-

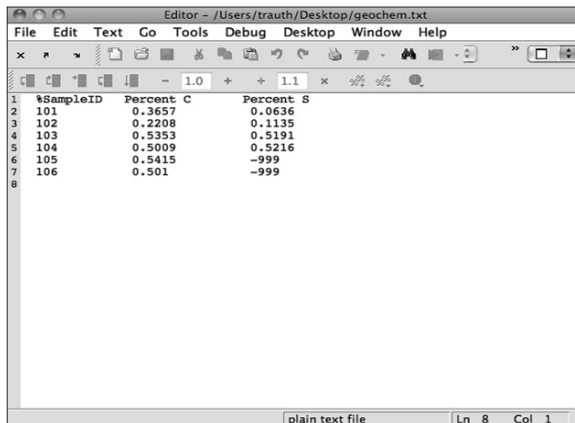


Fig. 2.2 Screenshot of MATLAB Editor showing the content of the file *geochem.txt*. The first line of the text is commented by a percent sign at the beginning of the line, followed by the actual data matrix.

puter platforms running different floating-point formats. The command

```
save geochem_new.mat geochem
```

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 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.

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. This double precision array allows storage of the sign of a number (first bit), the exponent (bits 2 to 12) and roughly 16 significant decimal digits between approximately 10^{-308} and 10^{+308} (bits 13 to 64). As an example, typing

```
clear

rand('seed',0)
A = rand(3,4)
```

creates a 3-by-4 array of random numbers with double precision. We use the function `rand` that generates uniformly distributed pseudorandom numbers within the open interval (0,1). To obtain identical data values, we 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 =
    0.2190    0.6793    0.5194    0.0535
    0.0470    0.9347    0.8310    0.5297
    0.6789    0.3835    0.0346    0.6711
```

By default, the output is in a scaled fixed point format with 5 digits, e.g., 0.2190 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 3
    0.218959186328090    0.679296405836612    0.519416372067955
    0.047044616214486    0.934692895940828    0.830965346112365
    0.678864716868319    0.383502077489859    0.034572110527461
Column 4
    0.053461635044525
    0.529700193335163
    0.671149384077242
```

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 is not changed. The precision is, however, affected by converting the data type from *double* to 32-bit *single precision*. Typing

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

yields

```
B =
    0.2189592    0.6792964    0.5194164    0.0534616
    0.0470446    0.9346929    0.8309653    0.5297002
    0.6788647    0.3835021    0.0345721    0.6711494
```

Although we have switched to `format long`, only 8 digits are displayed. The command `who` 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 applications where 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 3×4 elements, requires $3 \times 4 \times 64 = 768$ 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 =
    0.2190    0.6793    3.0000 + 4.0000i    0.0535
    0.0470    0.9347    0.8310    0.5297
    0.6789    0.3835    0.0346    0.6711
```

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 24-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 of 0 to 1. We then multiply the random numbers by 256 and round the results towards plus infinity using the function `ceil` to get values between 1 and 256.

```
clear

I1 = 256 * rand(100,100); I1 = ceil(I1);
I2 = 256 * rand(100,100); I2 = ceil(I2);
I3 = 256 * rand(100,100); I3 = ceil(I3);
```

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

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

Since RGB images are represented by integer values between 1 and 256 for each color, we convert the 64-bit double precision values to unsigned 8-bit integers using `uint8`.

```
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 1 and 256, whereas using `int8` to create signed 8-bit integers generates values between -128 and +127. The value of zero requires one bit and therefore there is no space 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 $10 \times 16 = 160$ bits or $160/8=20$ bytes of memory. Also 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 =  
06-Oct-2009
```

The third layer of `sample_1` contains the image created in the previous example, whereas 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*. Structures are often used in applications where organization of the data is of high importance. Cell arrays are often used when working with data that is intended for processing by index in a programming control loop.

2.6 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-files 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 vector *x*.

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

The first line of the *if loop* yields the dimensions of the variable *x* using the command *size*. In our example, *x* should be either a column vector with dimensions $(m, 1)$ or a row vector with 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* loop 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 a matrix of the same size, made up of elements set to logical 1 where the relation is true and elements set to logical 0 where it is not. 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` loop 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 vector `x`. We then type

```
average
```

without the extension *.m* to run our script and obtain the average of the elements of the vector `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 vector `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.
```

Obviously, we cannot 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 com-

mands 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 have 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 vector X.

% By Martin Trauth, Oct 6, 2009

[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

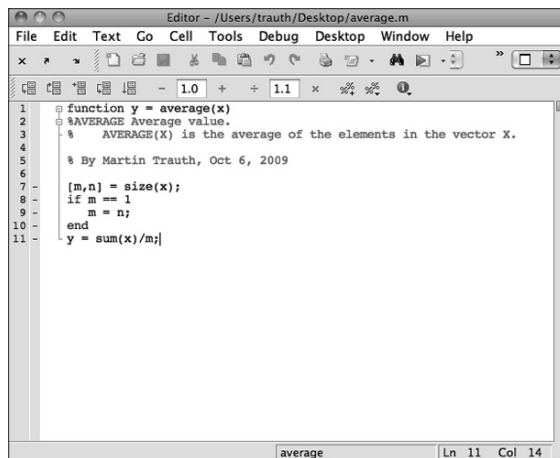


Fig. 2.3 Screenshot of the MATLAB Editor showing the function `average`. The function starts with a line containing the keyword `function`, the name of the function `average`, the input variable `x` and the output variable `y`. The subsequent lines contain the output for `help average`, the copyright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

to suppress the display of the result in the Command Window. Next we type

```
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];
```

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.

2.7 Basic Visualization Tools

MATLAB provides numerous routines for displaying data as graphs. This section introduces the most important graphics functions. The graphs 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 vectors x and y , where y is the sine of x . The vector x contains values between 0 and 2π 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 vectors with 21 elements each, i.e., two 1-by-21 arrays. Since the two vectors x and y have the same length, we can use `plot` to produce a linear 2d graph 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,'r--',x,y2,'b-')
```

creates a dashed red line displaying the sine of x and a solid blue line representing the cosine of this vector (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,'r--')
figure
plot(x,y2,'b-')
```

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 second graph as a bar plot.

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

This command plots $y1$ versus x as dashed line, whereas $y2$ versus x is shown as a group of blue vertical bars. Alternatively, we can plot both graphs in the same Figure Window but in different plots using `subplot`.

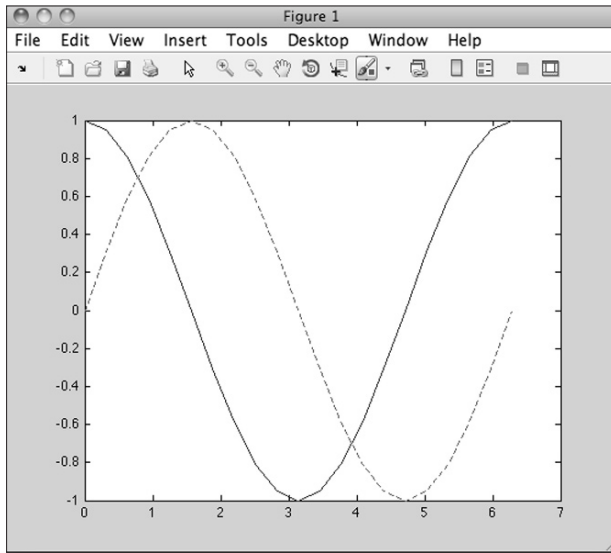


Fig. 2.4 Screenshot of the MATLAB *Figure Window* showing two curves in different colors and line types. The Figure Window allows editing of all elements of the graph after selecting *Edit Plot* from the *Tools* menu. Double clicking on the graphics elements opens an options window for modifying the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

The syntax `subplot(m,n,p)` divides the Figure Window into an m -by- n matrix of display regions and makes the p th display region active.

```
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, whereas the bar plot appears in the lower half of the Figure Window. 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 graphs 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 π , 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.

2.8 Generating M-Files to Regenerate Graphs

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 has 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 properties of the graph such as axes, lines, patches and text objects.

The *Generate M-Files* option enables us to automatically generate the MATLAB code of a figure to recreate a graph with different data. We use a simple plot to illustrate the use of the Property Editor and the Generate M-Files 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 3rd column of the color chart. Moving the mouse over this square displays the RGB color code [0.7 0.78 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 6-point square marker. We can close the Property Editor by clicking on the X in the upper right corner of the Property Editor panel below the graph. Finally, 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 M-File* 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 06-Oct-2009 17:37:20

% Create figure
figure1 = figure('XVisual',...
    '0x24 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)',...
    'Color',[0.6784 0.9216 1]);
```

```
% Create axes
axes('Parent',figure1);
box('on');
hold('all');

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

We can then rename the function *createfigure* to *mygraph* and save the file as *mygraph.m* using *Save As* from the *Editor File* menu.

```
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 `[0.702 0.7804 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 M-File function of MATLAB can therefore be used to create templates for graphs that can be used to generate plots of multiple data sets using the same layout.

Even though the MATLAB provides enormous editing facilities and the Generate M-File function even allows the generation of complex templates for graphs, 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 graphs are exported by selecting the command *Save as* from the *File* menu or by using the command `print`. This function exports the graph either as raster image (e.g., JPEG or GIF) or 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 graphs.

2.9 Publishing M-Files

A relatively new 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 graphs 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. Before we start developing the MATLAB code, we activate *Enable Cell Mode* in the *Cell* menu of the Editor. Whereas single percent signs % are known (from Section 2.6) to initiate comments in MATLAB, we now use double percent signs %% that indicate the start of new cells in the Editor. The *Cell Mode* is a feature in MATLAB that enables you to evaluate blocks of commands by using the buttons *Evaluate Cell*, *Evaluate Cell and Advance* and *Evaluate Entire File* on the *Editor Cell Mode* toolbar. The *Save and Publish* button, which was situated next to the Cell Mode buttons in earlier versions of MATLAB, is now included in the Editor Toolbar emphasizing the importance and popularity of this feature.

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 cells of MATLAB code and comments separated by the double percent signs %%.

```
%% Example for Publishing M-Files
% This M-file illustrates the use of the publishing feature
% of MATLAB.
% By Martin Trauth, Feb 8, 2009.

%% 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 myproject.m* button in the Editor Toolbar. 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 sub-headers are shown in a dark 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 graphs 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 graphs can be saved as a PDF-file and shared with students or colleagues.

Recommended Reading

- Davis TA, Sigmon K (2005) *The MATLAB Primer*, Seventh Edition. Chapman & Hall/CRC, London
- Etter DM, Kuncicky DC, Moore H (2004) *Introduction to MATLAB 7*. Prentice Hall, New Jersey
- Gilat A (2007) *MATLAB: An Introduction with Applications*. John Wiley & Sons, New York
- Hanselman DC, Littlefield BL (2004) *Mastering MATLAB 7*. Prentice Hall, New Jersey
- Palm WJ (2004) *Introduction to MATLAB 7 for Engineers*. McGraw-Hill, New York
- The MathWorks (2010) *MATLAB 7 Getting Started Guide*. The MathWorks Inc., Natick, MA



<http://www.springer.com/978-3-642-12761-8>

MATLAB® Recipes for Earth Sciences

Trauth, M.

2010, XI, 336 p. With online files/update., Hardcover

ISBN: 978-3-642-12761-8