

Chapter 2

Data Handling

MATLAB stores numbers, strings, and logical values into variables. Variables can be either simple, i.e., referring to one data type only, or complex, i.e., referring to different data types at the same time. Furthermore, variables can be imported, exported, and manipulated at will.

Types of Variables (Logical Values, Strings, NaN, Structures, Cells)

In the previous chapter we saw how to store and operate with numbers; however, sometimes we need to deal with other types of data such as strings, logical values, and so on. MATLAB can be used to deal with these types of variables as well.

Logical Variables

Logical data (or simply *logical*) represent the logical TRUE state and the logical FALSE state. Logical variables are the variables in which logical data are stored. Logical variables can assume only two states:

- False, always represented by 0;
- True, always represented by a nonzero object. Usually, the digit 1 is used for TRUE. However, MATLAB treats any nonzero value as TRUE.

There are two ways to create a logical variable. The first is to explicitly declare it using the `logical(X)` function; this function converts the elements of the array `X` into logical data types.

```
>> clear all;1
>> Test_Logic_Var = logical(1);
>> x = [0,1,2,3];
>> x_becomelogic = logical(x)
x_becomelogic =
    0     1     1     1
>> whos
  Name                Size      Bytes      Class      Attributes
  Test_Logic_Var       1x1         1      logical
  ans                  1x4         4      logical
  x                    1x4        32      double
  x_becomelogic        1x4         4      logical
```

By typing `whos` at that MATLAB prompt, you can see that `Test_Logic_Var` is a 1×1 matrix (a scalar value) of logical values. Vector `x` contains numbers different from 0 and 1. When the `logical` function is applied to the vector `x`, the nonzero values are marked as 1, that is, TRUE values.

We can also create logical variables indirectly, through logical operations, such as the result of a comparison between two numbers. These operations return logical values. For example, type the following statement at the MATLAB prompt:

```
>> 5 > 3
ans =
    1
>> 10 > 45
ans =
    0
```

Since 5 is indeed greater than 3, the result of the comparison is true; however, 10 is not greater than 45, and hence the comparison is false. The operator `>` is a *relational* operator, returning logical data types as a result.

When a vector or a matrix is involved in a logical or relational expression, the comparison is carried out *element by element*. Therefore, if we are checking whether the content of an array is greater than 5, the comparison is made for each element of the array, and the resulting logical array has the same length as that of the array we are checking. The following example shows how this works:

```
>> clear all;
>> a = [1,3,5,9,11];
>> b = [3,4,5,8,10];
>> c = a > 3;
>> D = a > b;
>> whos a b c d
```

¹Command examples are intended followed by the <ENTER> key, i.e.:

```
>> clear all
```

means: Type `clear all` after the `>>` prompt, and press the Enter key.

Name	Size	Bytes	Class	Attributes
a	1x5	40	double	
b	1x5	40	double	
c	1x5	5	logical	
d	1x5	5	logical	

Using the command `whos` you can see that the vectors `c` and `d` are logical vectors. They are not considered by MATLAB as numeric vectors but as vectors of logical values.

The following table lists the relational operators used by MATLAB.

MATLAB operator	Description	Example	Meaning
<code>></code>	Greater than	<pre>>> c = a > 3 c = 0 0 1 1 1 >> d = a > b d = 0 0 0 1 1</pre>	<p>c shows which values of a are greater than 3</p> <p>d shows which values of a are greater than the values in the same position of b</p>
<code>>=</code>	Greater than or equal to	<pre>>> e = a >= 3 e = 0 1 1 1 1 >> f = a >= b f = 0 0 1 1 1</pre>	<p>e shows which values of a are greater than or equal to 3</p> <p>f shows which values of a are greater than or equal to the values in the same position of b</p>
<code><</code>	Less than	<pre>>> g = b < 3 g = 0 0 0 0 0 >> h = a < b h = 1 1 0 0 0</pre>	<p>g shows which values of b are less than 3.</p> <p>h shows which values of a are less than the values in the same position of b</p>
<code><=</code>	Less than or equal to	<pre>>> i = b <= 3 i = 1 0 0 0 0 >> j = a <= b j = 1 1 1 0 0</pre>	<p>i shows which values of a are less than or equal to 3</p> <p>j shows which values of a are less than or equal to the values in the same position of b</p>
<code>==</code>	Equal to	<pre>>> k = a == 9 k = 0 0 0 1 0 >> l = a == b l = 0 0 1 0 0</pre>	<p>k shows which values of a are equal to 9</p> <p>l shows which values of a are equal to the values in the same position of b</p> <p>Note: the logical operator <code>==</code> is different from the assign operator <code>=</code>! Pay attention when you use it: a=b is different from a == b</p>
<code>~=</code>	Not equal to	<pre>>> k = a ~= 9 k = 1 1 1 0 1 >> l = a ~= b l = 1 1 0 1 1</pre>	<p>k shows which values of a are equal to 9</p> <p>l shows which values of a are equal to the values in the same position of b</p>

Logical operators are useful in different occasions, such as in preliminary data analysis. Let's say you need to calculate the average of a data set that includes some outliers, and you want to have the average calculated without them. In this case, you can use the logical operators to average only the data you want to include and according to a cutoff value of your choice (later in the text you will see more examples).

MATLAB uses several logical operators such as `&`, `|`, and `~`. The following table shows their use by considering the vectors `a`, `b`, `c`, and `d` implemented above.

MATLAB operator	Description	Truth table	Example	Meaning															
&	Logical AND	<table><tr><th>A</th><th>B</th><th>A&B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	A&B	0	0	0	0	1	0	1	0	0	1	1	1	<pre>>> m = a & b m = 1 1 1 1 1 >> n = c & d n = 0 0 0 1 1</pre>	<p>m contains the element-by-element AND of the vectors a and b. The values of a and b are all different from zero, that is, they are TRUE values. The result is a vector of 1s</p> <p>n contains the element-by-element AND of the logical vectors c and d</p>
A	B	A&B																	
0	0	0																	
0	1	0																	
1	0	0																	
1	1	1																	
	Logical OR	<table><tr><th>A</th><th>B</th><th>A B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	A B	0	0	0	0	1	1	1	0	1	1	1	1	<pre>>> o = a b o = 1 1 1 1 1 >> p = c d ans = 0 0 1 1 1</pre>	<p>o contains the element-by-element OR of vectors a and b. The values of a and b are all different from zero; they are TRUE values</p> <p>p contains the element-by-element OR operation of the logical vector c.</p>
A	B	A B																	
0	0	0																	
0	1	1																	
1	0	1																	
1	1	1																	
~	Logical NOT	<table><tr><th>A</th><th>~A</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	~A	0	1	1	0	<pre>>> q = ~a q = 0 0 0 0 0 >> r = ~c r = 1 1 0 0 0</pre>	<p>q contains the element-by-element NOT. Values in a are all different from zero; they are TRUE values. The result is a vector of 1s</p> <p>r contains the element-by-element NOT logical operation of the logical vector c</p>									
A	~A																		
0	1																		
1	0																		

In many cases we need to perform multiple comparisons at once. This is, of course, possible in MATLAB, but we need to follow the MATLAB rules if we do not want to get the wrong result. To show this, let's test whether a variable `x` falls within the range from 0 to 2. We might be tempted to prompt: `0 < x < 2`. However,

using this syntax leads to an incorrect result. Indeed, if, let's say, x is equal to 3, hence outside our range, MATLAB returns 1, which is TRUE.

```
>> x = 3;
>> 0 < x < 2
ans =
     1
```

Why does MATLAB return an incorrect result? Because MATLAB makes the comparisons in succession. It first compares x with 0, and because 3 greater than 0, the result of the comparison is true, i.e., 1. Then MATLAB compares the result, i.e., 1, with 2. Because 1 is less than 2, the result of the operation is true. So we need to use a different syntax to obtain the correct result. Multiple comparisons like the previous one have to be written in the following way:

```
>> (0 < x) & (x < 2)
ans =
     0
```

Let us see how to use logical values to target different positions in a vector. We have already seen in Chap. 1 that the elements of a vector can be referenced by means of another numeric vector; we can access some of the vector's elements using another vector to point to the positions we want. For example, if we want the elements in the third and fifth positions of a vector **a** of size 5, we can use another vector **b** as an index vector pointing to the positions we need:

```
>> clear all
>> a=[3, 4, 7, 9, 11];
>> b=[3,5];
>> a(b)
a=
     7     11
```

When programming, we use logical index vectors in several contexts. Let's say that we have a numeric vector and we want to store in a second vector only the values outside the range 6–2. We can do so in this way:

```
>> clear all;
>> a = [1, -2, 5, 7, 3, 26];
>> c = (a>=7) | (a<2)
c =
     1     1     0     1     0     1
>> d=a(c)
d =
     1    -2     7    26
```

d contains those values stored in **a** that are TRUE according to **c**. Note, however, that logical vectors, such as **c**, can be used to index another vector only when their sizes are identical. Indexing through logical vectors is a very practical way to remove elements from a vector. So we can create the vector **d** with the following simple command line:

```
>> d = a( (a>=7) | (a<2) )
d =
      1      -2      7      26
```

Logical indexing is very useful when one variable is used to categorize a second variable. Let's suppose we have collected some data in the following experiment, which is based on the Posner cueing paradigm (Posner 1980). Subjects are asked to react as fast as possible to the appearance of a target that can appear to either the left or the right of a central fixation point. Before the target appears, a cue indicates its location, but this cue has a limited validity (e.g. 70% of the trials).

The cue's validity (or invalidity) can be represented with 1s (or 0s) in the CueValidity vector. Subjects' response times, in seconds are stored in the vector RT. And these are the data we have collected so far:

```
>> RT = [0.90, 0.55, 1.01, 0.33, 0.442, 0.51, 0.85, 0.44];
>> CueValidity = logical([0 , 1, 0, 1, 1, 1, 0, 1]);
```

Response time can be categorized according to cue validity in the following way:

```
>> RTvalid = RT(CueValidity)
RTvalid =
      0.5500      0.3300      0.4420      0.5100      0.4400

>> RTinvalid = RT(~CueValidity)
RTinvalid =
      0.9000      1.0100      0.8500
```

MATLAB has a number of *logical functions* operating on scalars, vectors, and matrices. Some examples are given in the following table. The examples use the following vectors:

```
>> clear all
>> a=[1 3 6 3 1 7];
>> b=a>5;
>> d=[ ];
```

MATLAB function	Description	Example 1	Example 2
any(x)	Return the logical 1 (true) if any element of the vector is a nonzero number. For matrices, any(x) operates on the columns of x, returning a row vector of logical 1s and 0s	>> any(a) ans = 1	>> any(b) ans = 1

(continued)

(continued)

MATLAB function	Description	Example 1	Example 2
<code>all(x)</code>	Return the logical 1 (true) if all the elements of the vector are nonzero. For matrices, <code>all(x)</code> operates on the columns of <code>x</code> , returning a row vector of logical 1s and 0s	<pre>>> all(a) ans = 1</pre>	<pre>>> all(b) ans = 0</pre>
<code>exist('A')</code>	Check whether variables or functions are defined. 0 if <code>A</code> does not exist, 1 if <code>A</code> is a variable in the workspace	<pre>>> exist('c') ans = 0</pre>	<pre>>> exist('a') ans = 1</pre>
<code>isempty(x)</code>	Return the logical 1 (true) for the empty array	<pre>>> isempty(a) ans = 0</pre>	<pre>>> isempty(d) ans = 1</pre>

Strings

A string is a variable containing characters instead of numbers. Strings can be used to record subjects' names or any other type of textual information. A string is assigned to a variable by enclosing it within apostrophes as in the following example:

```
>> nameStr = 'Anne';
>> whos name
      Name      Size  Bytes    Class  Attributes
      nameStr   1x4      8      char
```

The string 'Anne' is composed of four characters, and the variable `nameStr` is a 1×4 row vector of characters. The second letter of the name can be accessed in the following way:

```
>> name(2)
ans =
     n
```

If you want to include a string containing an apostrophe, the apostrophe must be repeated:

```
>> sentence='Anne''s dog is Buddy'
sentence =
Anne's dog is Buddy
```

Because strings are vectors, they may be linked with square brackets, e.g.,

```
>> name = 'Andrea';
>> surname = 'Palladio';
>> fullname=[name, ' ', surname]
```

```

fullname =
Andrea Palladio
>> whos name surname fullname
    Name      Size  Bytes  Class  Attributes
    fullname  1x15      30   char
    name      1x6       12   char
    surname   1x8       16   char

```

Note that we have put a space character between the name and the surname to separate them. The result is a vector of 15 characters: 6 characters belong to the name, 8 to the surname, and 1 for the space.

Now let's suppose you want to create an array of names, each row for one name. Because names can differ in length, we need to use the `char` function. Indeed, if we implement the variable `NameList` in the following way, we get an error.

```

>> NameList=['John', 'Milly', 'Giovanni'];
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.

```

Function `char()` overcomes the problem:

```

>> NameList2=char('John', 'Milly', 'Palladio')
NameList2 =
John
Milly
Palladio

```

MATLAB has many functions for working with strings, which are listed in the following table. Use the string `MyString='Vision Search'`, `str1='hello'` and `str2='help'` to get some practice with them:

Function	Description	Example
<code>int2str(n)</code>	Convert numeric arguments into a string	<code>>> IntStrIng=int2str(25);</code>
<code>num2str(n)</code>	Convert numeric arguments into a string	<code>>> numString=num2str(23.4);</code>
<code>lower(S)</code>	Convert a string into a lowercase string	<code>>> lower(MyString)</code> <code>ans =</code> <code>vision search</code>
<code>upper(S)</code>	Convert a string into an uppercase string	<code>>> upper(MyString)</code> <code>ans =</code> <code>VISION SEARCH</code>
<code>strcmp(S1,S2)</code>	Compare the strings <code>S1</code> and <code>S2</code> and return true (1) if strings are identical, and false (0) otherwise	<code>>> strcmp(str1,str2)</code> <code>ans =</code> <code>0</code>

(continued)

(continued)

Function	Description	Example
<code>strrep(S1,S2,S3)</code>	Replace all the occurrences of the string S2 in the string S1 with the string S3	<pre>>> strrep(str1,'llo','avy') ans = heavy</pre>
<code>findstr(S1,S2)</code>	Return the starting indices of any occurrences of the shorter of the two strings in the longer	<pre>>> findstr(str1,'l') ans = 3 4</pre>
<code>strmatch(S1,CAR)</code>	Look through the character array CAR to find strings beginning with the string contained in S1, returns the matching row indices	<pre>>> strmatch('he',str3) ans = 1 2</pre>
<code>disp(S1)</code>	Displays the array S1, without printing the array name	<pre>>> disp(str3); hello help >> disp([1 2; 3 4]); 1 2 3 4</pre>

To create formatted strings there is another useful function: `sprintf(format, variables)`. Let us see an example. Type the following commands:

```
>> RTmean=[0.431,0.321];
>> Pos=char('left', 'right');
>> sprintf('The RT for objects in %s position is %1.1f sec.',...
Pos(1,:),RTmean(1))
ans =
The RT for objects in left position is 0.4 sec.
```

```
>> sprintf('The RT for objects in %s position is %1.3f sec.',...
Pos(2,:),RTmean(2))
ans =
The RT for objects in right position is 0.321 sec.
```

Note that the three ellipsis points ... allow you to continue the command in the next line.

The function `sprintf()` contains the format argument and some variables. The format argument is a string containing ordinary characters and conversion specifications. A conversion specification controls the notation, the alignment, the significant digits, the field width, and other aspects of the output format. Conversion specifications begin with the `%` character. There are also special characters beginning with the `/` character. Some of these are presented in the following table; however, for a complete list of them, refer to the MATLAB help.

Conversion specification and special characters	Description	Example
%c	Single character	>> sprintf('character: %c','c') ans = character: c
%d	Decimal notation (signed). There is an implicit conversion from number to string without use the int2str function	>> sprintf('integer: %d',12) ans = integer: 12
%f	Fixed-point notation. A decimal number can be inserted between the % and f symbols to specify the size of the integer part and the fractional part, respectively	>> c=5.12345; >> sprintf('float: %3.1f',c) ans = float: 5.1 >> sprintf('float: %3.3f',c) ans = float: 5.123
%s	String of characters	>> s='test'; >> sprintf('string: %s',s) ans = string: test
\n	New line	>> sprintf('go to \n new line') ans = go to new line
\t	Horizontal tab	>> sprintf('Test the \t tab \t tab') ans = Test the tab tab

Another useful function is `input()`. It waits for input from the keyboard, ending with the ENTER key. Type the following:

```
>> input('How old are you? ')
How old are you? 35
ans =
    35
```

By default, `input()` takes numbers as its argument. If we need strings instead, we need to add the optional argument 's' to the `input` call:

```
>> input('How old are you? ','s')
How old are you? thirtyfive
ans =
thirtyfive
```

NaN

NaN means *Not a Number*. This variable type is used for missing data. For example, let's suppose you need to calculate the mean of the elements of a vector but there are some missing values:

```
>> Meanex=[2 NaN 12 4 NaN 3 NaN]

Meanex =
      2   NaN   12    4   NaN    3   NaN
>> nanmean(Meanex)
ans =
      5.25
```

MATLAB has few built-in functions for working with NaNs. However, not all standard MATLAB functions can deal with NaNs. To overcome this limitation we need to use logical operators. The function `isnan(X)` finds where the NaNs are. It returns a logical array containing 1s wherever the elements of `X` are NaNs. To calculate the mean of a vector using the standard `mean()` function, we need to use the following syntax:

```
>> S=mean(Meanex (~ (isnan(Meanex))))
S =
      5.25
```

Note that if you don't use logical operators, the result you get is a NaN:

```
>> S=sum(Meanex)
S =
      NaN
```

Structures

MATLAB can manage complex variables, that is, variables that are of different types, at the same time. These variables are called *structures*. A structure collects different types of elements under the same name. The elements are called *fields*. For example, to store the information about the participant of an experiment, we can create a structure variable called `SubjectTest` and assign different values to the various fields as follows:

```
>> SubjectTest.name='Nelson';
>> SubjectTest.surname='Cowan';
>> SubjectTest.age=24;
>> SubjectTest.TestDone=[1,2,3,6,12];
>> SubjectTest.Response=[12.3, 11.2, 14.3, 12.2,12.4];
>> SubjectTest.CorrectConduction=logical([1,1,0,1,1]);
```

```
>> SubjectTest
SubjectTest =
    surname: 'Cowan'
      name: 'Nelson'
       age: 24
  TestDone: [1 2 3 6 12]
  Response: [12.3000 11.2000 14.3000 12.2000 12.4000]
CorrectConduction: [1 1 0 1 1]
```

The structure name is `SubjectTest`, while `name`, `surname`, `age`, `TestDone`, `Response`, `CorrectConduction`, are the fields. Fields are addressed using the structure name followed by a dot and then the field name. Hence, the second element of the `Response` field can be addressed as follows:

```
>> SubjectTest.Response(2)
ans =
    11.2000
```

Structure fields are case-sensitive; this means that MATLAB creates additional fields if we do not type the field name correctly. It is also possible to combine structures to create a matrix of structures. This is useful, for example, when you need to save the results of different participants. For example, the data of a second participant can be added to the structure in the following way:

```
>> SubjectTest(2).name='Johan';
>> SubjectTest(2).ResponseTime=[11.9, 11.1, 14.1, 11.8,12.0];
>> whos SubjectTest
```

Name	Size	Bytes	Class	Attributes
SubjectTest	1x2	1045	struct	

As you can see from the `whos` command, `SubjectTest` is a 1×2 row vector of struct:

```
>>SubjectTest(2).Name = 'Johan'
SubjectTest =
1x2 struct array with fields:
    surname
      name
       age
  TestDone
  Response
CorrectConduction
ResponseTime
      Name
```

If you want to address the second element of the matrix, you can type:

```
>> SubjectTest(2)
ans =
```

```

        surname: []
        name: 'Johan'
        age: []
    TestDone: []
    Response: []
    CorrectConduction: []
    ResponseTime: [11.9000 11.1000 14.1000 11.8000 12]
    Name: 'Johan'

```

As you can see, some fields are empty; this is because they haven't been filled for the second participant. It is possible to fill them in later. To do this, type:

```

>> SubjectTest(2).surname= 'Baptist'
>> SubjectTest(2).age= 31
>> SubjectTest(2)
ans =

        surname: 'Baptist'
        name: 'Johan'
        age: 31
    TestDone: []
    Response: []
    CorrectConduction: []
    ResponseTime: [11.9000 11.1000 14.1000 11.8000 12]
    Name: 'Johan'

```

Let's see how to access an element of a vector that is a field of a structure while the structure is simultaneously a member of a matrix of structures. To do this, we have to type the structure name, the index of the structure within parentheses, followed by a point and finally the field name within parentheses. For example, let's suppose you want to display the last of John's response times:

```

>> SubjectTest(2).ResponseTime(5)
ans =
    12

```

Dynamic Field Names

Another way to access structure data that are embedded in a vector is to use *dynamic field names*. Dynamic field names express the field as a string variable. For example, to extract all the surnames in the vector, you have to type:

```

>> StrField='surname'
>> SubjectTest.(StrField)
ans =
    Cowan
ans =
    Baptist

```

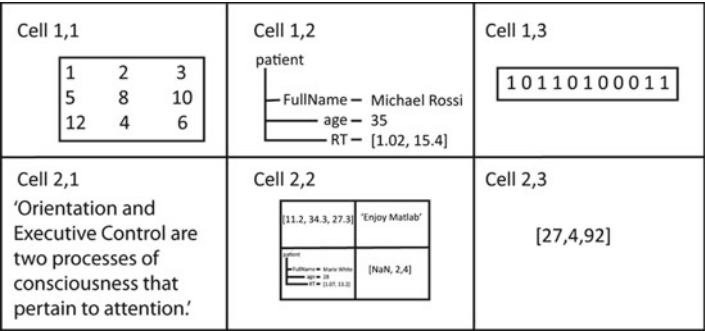


Fig. 2.1 An example of cell array

You can remove a given field from a structure embedded within a structure array using the `rmfield()` function:

```
>> rmfield(SubjectTest, 'TestDone')
ans =
1x2 struct array with fields:
    age
  ResponseTime
CorrectConduction
  surname
    name
```

Cells

A *cell* is a variable containing different types of data; it is therefore similar to a structure, but cells are more general and have notational differences as well. A cell can contain any data type, from a simple number to a structure or another cell. You can have even a *cell array*, i.e., a matrix in which each element is a cell.

In Fig. 2.1 is reported an example schema which should clarify the concept by showing a 2x3 cell array. In the first row and first column, the cell array contains a matrix of numbers, while in the first row and second column, the cell array contains a structure, and in the second row and second column, the cell array contains another cell array, and so on.

Let's see how to create cell arrays and how to insert and retrieve data from them. Be aware of the notational differences between simple arrays, such as numeric and character arrays, and cell arrays, because they can be source of confusion (and errors!).

To create a cell array there are two methods, called “Cell indexing” and “Content indexing”:

- **Cell Indexing:**

```
>> SoundInf={'sine','square','Sting'}
SoundInf =
    'sine' 'square' 'Sting'
>> whos SoundInf
Name      Size      Bytes   Class   Attributes
SoundInf  1x3         210    cell
```

Here the curly braces { } are on the **right-hand side**, and indicate the cell contents. This is a *cell array* of strings, which is different from an *array* of strings; indeed, strings stored in cell arrays can have different numbers of characters. Let's add additional values to the cell:

```
>> SoundInf(2,3) = {[5, 6; 7, 8] }
SoundInf =
    'sine' 'ramp' 'sting'
         []      [] [2x2 double]
```

The parentheses on the left-hand side of the assignment refer, in the normal way, to elements in a cell array, while on the right-hand side the curly braces indicate the content of a cell.

Let's see how content indexing differs from cell indexing.

- **Content Indexing:**

```
>> SoundInf{2,3}=5
SoundInf =
    'sine'    'ramp'    'sting'
         []         []    [      5]
>> SoundInf{3,3}=[2 3; 7 8]
SoundInf =
    'sine'    'ramp'    'sting'
         []         []    [      5]
         []         [] [2x2 double]
```

Here, the **index of the cells within curly braces { }** and the content are specified in the standard way after the assignment sign.

You can access the content of a cell by indexing:

```
>> PreferArtist = SoundInf{1,3}
PreferArtist =
Sting
>> whos PreferArtist
Name      Size      Bytes   Class   Attributes
PreferArtist  1x5         10    char
```

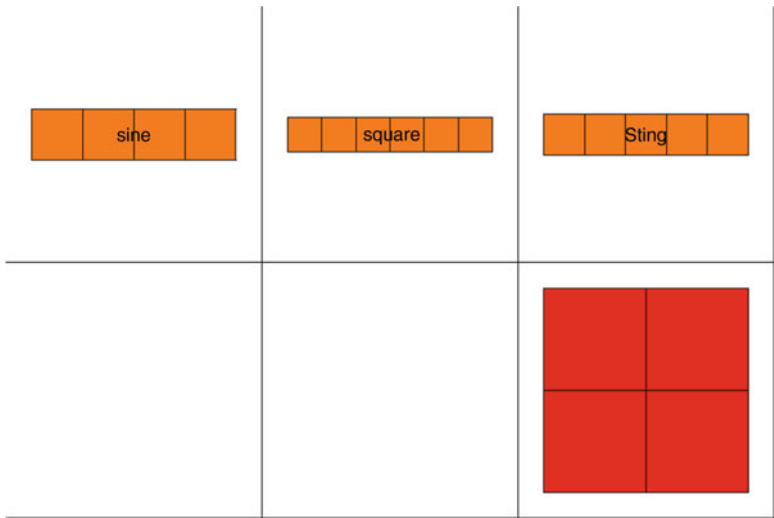


Fig. 2.2 Output of the cellplot command

To access an element of an array stored in a cell you have to concatenate curly braces and parentheses as follows:

```
>> SoundInf{2,3}(2,2)
ans =
      8
```

The `celldisp()` function recursively displays the contents of a cell array. The function `cellplot` draws a visualization of a cell array, as shown in Fig. 2.2. Nonempty array elements are shaded.

```
>> cellplot SoundInf
```

We have seen that both *structures* and *cell arrays* can contain different element types. The main difference between them is reported in the following table:

Structures	Cell arrays
Structures are different data type collections, organized in fields having their own names	Cell matrices can contain different data types. Each cell is indexed by numbers

Import/Export

In this section we see how to save and load the variables that have been created and used in a working session. If we do not save the variables, MATLAB automatically clears them from the working space when you quit the program. The `save` command followed by a filename saves all the variables that are in the workspace.

In the following example we create a file named `test1` containing the vector `a`, the number `b`, and the string `f`:

```
>> clear all;
>> a=[4,2,1,43; 2,5,1,6];
>> b= sin(a)+5;
>> f='test';
>> save test1
```

If we clear all the variables by means of `clear all`, we will see that these variables no longer exist in the workspace:

```
>> clear all;
>> whos
>>
```

However, we can retrieve them by means of the `load` command, followed by the filename:

```
>> load test1
>> whos
```

Name	Size	Bytes	Class	Attributes
a	2x4	64	double	
b	2x4	64	double	
f	1x4	8	char	

If you want to save only a subset of the variables you have created, you have to type the list of variables, separated by blanks (not by commas), that you want to save after the filename:

```
>> save prova2 b f
>> clear all
>> load prova2
>> whos
```

Name	Size	Bytes	Class	Attributes
b	2x4	64	double	
f	1x4	8	char	

By default, MATLAB saves the variables in a `.mat` file. However, other extensions can be given such as ASCII (save the information in ASCII format so that the file can be editable from a standard text editor) or TAB (delimited with tabs).

The commands `save` and `load` can be seen as functions. The argument of these functions is a string. The first argument is the filename, the others are the variable names. Let's suppose you have the matrices `RT1=[0.34,0.45]` and `RT2=[0.23,0.39]`:

```
>> save('test1','RT1');
```

MATLAB provides some built-in functions that can import/export numbers and characters from and to different file formats. In the following table the most common of these formats are presented.

File format	File content	Extension	Functions
MATLAB formatted	Saved MATLAB workspace	.mat	load, save
Text	Text	.txt	textread
	Comma-separated numbers	.csv	Csvread csvwrite
Extended markup language	XML-formatted text	.xml	Xmlread xmlwrite
Spreadsheet	Excel worksheet	.xls	Xlsread xlswrite

There are other built-in functions that enable importing images or sound files, and they will be described in the following chapters. If you don't want to type the function in the command window, you can simply use the MATLAB Import Wizard by selecting **Import Data** in the MATLAB **File** menu, or equivalently, by typing `uiimport` at the MATLAB command line prompt.

Summary

- Logical variables can assume only two values: true (not equal to 0) and false (equal to 0).
- Logical and relational operators are used to assess whether a statement is true or false.
- Logical variables can be defined using the function `logical(variablename)`.
- Logical variables are the results of element comparison.
- Logical variables can be used to select or remove elements from a matrix. Logical variables are also useful in categorizing elements.
- Strings are arrays in which each element represents one character. Because strings are arrays, all the rows must have the same number of columns.
- Strings are defined using apostrophes'.
- There are many functions operating on strings to compare them, display them, etc.
- `sprintf` is a useful function to write formatted text.
- NaN means *Not a Number*. It is useful for representing missing data.
- *Structures* are collections of different data types, organized in fields.
- A *cell matrix* contains different data types, from simple numbers to structures or even other cell arrays. Each cell is indexed by numbers.
- There are two ways to insert data into a cell: *Cell indexing* (Index the cell array with the content in curly braces { }) and *content indexing* (the indices of the cells are in curly braces { }; the content is specified in standard way after the = sign).
- To store data in files, use the `save` command; to retrieve data from files, use the `load` command. There are other commands for importing or exporting data in specific formats including xml and xls.

Exercises

1. Categorize the elements of the vector `x = [-2 3 0 2 -6 1 -2 0 0 -13 12]` as positive, negative, and zero and store them in three separate vectors. Count the number of elements in each vector.

Solution:

```
>> pos=x(x>0); neg=x(x<0); zer=x(x==0);
>> npos=length(pos); nneg=length(neg); nzeros=length(zer);
```

or equivalently:

```
>> npos=sum(x>0); nneg=sum(x<0); nzeros=sum(x==0);
```

where the `sum(Z)` function calculates the sum of the elements in the vector `Z`.

2. Define the following cell matrix:

```
NAMES=char('John','Robert','James','Michael'...
'Mary','Tatiana','Jenny','William');
and evaluate the following questions:
```

Question	Solution	Result
How many names start with the letter 'J'?	<pre>>> sum(NAMES(:,1)=='J')</pre>	<pre>ans = 3</pre>
Substitute the string 'es' with the string 'iroquai' in the third name	<pre>>> strep(NAMES(3,:), ... 'es','iroquai')</pre>	<pre>ans = Jamiroquai</pre>
Is the first name equal to the third?	<pre>>>strcmp(NAMES(1,:), NAMES(2,:))</pre>	<pre>ans = 0</pre>
Convert the names in the odd rows to uppercase	<pre>>> odds=[2:2:length(NAMES)]; >> NAMES(odds,:)=... upper (NAMES(odds,:))</pre>	<pre>NAMES = Joh ROBERT James MICHAEL David MARY Tatiana JENNY William</pre>
Display the formatted string 'The best is' followed by the name of the seventh name	<pre>>> sprintf('The best is %s,' NAMES(7,:));</pre>	<pre>ans = The best is Tatiana</pre>
Delete the names starting with J	<pre>>> NAMES(NAMES(:,1)=='J',:)=[]</pre>	<pre>NAMES = Robert Michael David Mary Tatiana William</pre>

A Brick for an Experiment

Read the Results

We are still in the introductory chapters, and during this brick we will realize a program to run a quite complex experiment. In order to use the concepts we have learned so far, for the brick we need to hypothesize that the experiment is over and that the data file has been safely stored onto the hard drive. In the current brick we will see how to analyze the data.

Before showing how to use the concepts learned so far for the current brick, we need to give you some detail about the data file we will write in the successive chapters. Moreover, we have to inform you about how the data will be stored into this data file. At the end of the experiment we will save a tab-delimited text file. This file will be organized in rows and columns. The first row of the file contains the header, that is, a set for strings identifying the content of the column. The data file will contain on the left text data and on the right numeric data. From left to right, the columns of our data file will contain the subject name, the subject sex, a text note about the subject, the subject number, the subject age, the block number, the trial number, the motion condition (coded as 1 for continuous motion, 2 for the stop at overlap), the sound condition (coded as 1 for the no sound condition and 2 for the sound condition), and the subject's response, which will be coded as 0 (streaming response) or 1 (bouncing response).

The first thing we have to do is to import the data written in the data file into MATLAB. As we have seen, MATLAB offers several command line options to do this. In order to perform this particular operation, a very convenient (and simple) option is that of using the MATLAB import wizard. If you click on the MATLAB file menu you will see the import data function. An alternative way to call the wizard is to type `uiimport` at the MATLAB prompt. In both cases, MATLAB asks you where the file you want to import can be found. You have now to browse your computer and look for the file. Let's suppose that the file is placed in the same folder where we have run the experiment. Once you have selected the data file,² you should see a graphical interface similar to the following (Fig. 2.3).

MATLAB shows you a preview of the file content. Click next. At the top of the import wizard interface there are several options you can select. For example, in our case, we need to tell MATLAB that the file content is tab delimited (but MATLAB should recognize this). On the right you can tell MATLAB the number of header rows in the file. Our data file has one header row. In importing this particular data file, MATLAB will store two variables. One cell type variable and one double type variable. The reason is simple: our data file contains numbers as well as strings. MATLAB recognizes the first columns as strings (those containing the subject's name, sex, and note). However, the successive columns are recognized as numbers

² You can download this file from the book website (http://www.psy.unipd.it/~grassi/matlab_book.html). The filename is data.txt.

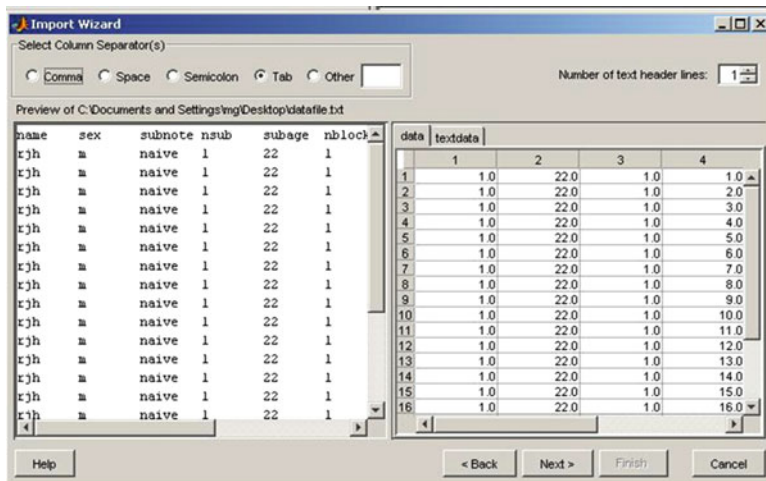


Fig. 2.3 Graphical interface to import data

(the subject's age, the subject number, and so on) and are therefore stored into a matrix of type double.

The first thing you may want to do is to have a preliminary look at the data, such as a set of descriptive statistics. For example, we may want to see whether bounce responses are predominant in the sound condition as well as when the discs stop at the overlap point. The following commands are sufficient to highlight the rows where MATLAB can find the conditions under which we presented the sound and where the discs stopped at the overlap:

```
>> data(:, 5)==1; % continuous motion
>> data(:, 5)==2; % motion with stop
>> data(:, 6)==1; % sound absent
>> data(:, 6)==2; % sound present
```

Note the presence of the % character. It is used for comments in MATLAB when you write an M-script (see Chap. 3). MATLAB takes no action when the % character is encountered, and it ignores everything that follows it.

If we want to compute the mean separately for these four conditions, we need to use the above rows of code when we call for the function `mean`. In detail, we need to tell MATLAB to calculate the mean of the column in which we have stored the dependent variable (i.e., the last column) and to calculate the mean in particular when the conditions outlined by the independent variables are satisfied. To do the calculation we have to write as follows:

```
>> mean(data(data(:, 5)==1, 7)); % continuous motion
>> mean(data(data(:, 5)==2, 7)); % motion with stop
>> mean(data(data(:, 6)==1, 7)); % sound absent
>> mean(data(data(:, 6)==2, 7)); % sound present
```

If we substitute the command `mean` with the command for the standard deviation (i.e., `std`), we can also obtain the standard deviations of the four conditions. The same command lines can be used as well for other descriptive statistics such as `min` and `max`.

Reference

Posner MI (1980) Orienting of attention. *Q J Exp Psychol* 32:3–25

Suggested Readings

Some of the concepts illustrated in this chapter can be found, in an extended way, in the following book:

Hahn BD, Valentine DT (2009) *Essential MATLAB for engineers and scientists*, 4th edn. Elsevier/Academic Press, Amsterdam

Higham DJ, Higham NJ (2005) *MATLAB guide*. Siam, Philadelphia

Kattan PI (2008) *MATLAB for beginners: a gentle approach*, Revised edn. Lulu.com, Raleigh, NC, United States

Rosenbaum DA (2007) *MATLAB for behavioral scientists*. Lawrence Erlbaum Associates, Mahwah, N.J.

MATLAB for Psychologists

Borgo, M.; Soranzo, A.; Grassi, M.

2012, XVI, 284 p., Hardcover

ISBN: 978-1-4614-2196-2