

4 Matrices, vectors and scalars

One defining feature of Matlab is that numerical variables are typically matrices, not scalars. A scalar is a single number such as 5, 7.89, or 10243. All the numerical examples so far have used scalars. A matrix, on the other hand, is a rectangular set of numbers, arranged in rows and columns. In text, a matrix is usually enclosed within brackets. As an example, consider the following matrix that has three rows and four columns:

$$\begin{bmatrix} 1 & 32 & 7 & 8 \\ 2 & 4 & 1 & 9 \\ 3 & 19 & 3 & 9 \end{bmatrix}$$

The size of the matrix is measured by the number of rows and columns. This matrix is a 3x4 matrix (pronounced three-by-four). As a special case, you may consider a scalar to be a 1x1 matrix. Matrices with only one column or one row often have special meaning, and are usually referred to as column vectors and row vectors. Furthermore, a matrix with equally many rows and columns is often called a square matrix.

Oftentimes, the columns and rows of a matrix represent some distinguishing feature of the data. In the example matrix, the first column might denote the day the observations were made, numbered from one to three. The second to fourth columns might be certain outcomes that occurred during the corresponding days. For example, it could be the amount of rain in three different locations measured in, say, millimeters. The amount of rain in the second location on the third day would then be 3 millimeters. The important feature is that the order of the numbers represents relations between them. If you are familiar with spreadsheet programs, such as Excel, you can think of a matrix as a rectangular set of cells in a spreadsheet. Note, however, that it is not possible to have “empty” entries in a matrix. If no value is appropriate for a certain entry, you use NaN (not-a-number).

Similar to scalars, matrices can be added, subtracted, multiplied, etc. This is called matrix algebra. Matlab does matrix algebra really well. While you do not have to know any matrix algebra to work with Matlab, some of its power is lost if you do not. And you still have to know how to create and change matrices, as they are used for several different purposes in Matlab. For instance, in Section 6, you will see that when data is imported into Matlab, it is typically imported as a matrix.

In this section, we will describe how to create and change matrices, as well as how to address parts of them. Section 5 describes mathematical operations with matrices, including some matrix algebra.

4.1 Creating matrices

The most straightforward way to create a matrix in Matlab is to enter it element-by-element. Say we want to define a variable, `testMatrix`, which is equal to the matrix in the example above. We then enter the variable name followed by the assignment operator and a left square bracket, often produced by typing `<Ctrl>-<Alt>-` (. We then enter the first row of numbers, separated by one or several blank spaces or, alternatively, separated by commas. To indicate a new row we enter a semicolon, and then continue with the numbers on the second row, etc. Lastly, we enter a closing right square bracket. What we see in the Command Window is

```
>> testMatrix = [1 32 7 8 ; 2 4 1 9 ; 3 19 3 9]
testMatrix =
     1     32      7      8
     2      4      1      9
     3     19      3      9
```

When Matlab prints the matrix in the Command Window, it does not print any brackets. Now, `testMatrix` is defined as a variable and it is listed in the Workspace. Its class is "double", similarly to other numerical variables. However, in the value-column you do not see its value. Instead it states `<3x4 double>`, where the 3x4 refers to the size of the matrix.

Entering matrices manually is only practical when they are sufficiently small. Surprisingly often, however, useful matrices have systematic features that make them easy to create in other ways. They might contain only zeroes or only ones, or they might contain equally spaced ascending numbers. For these purposes, it is very useful to know the commands listed below.

4.1.1 Commands for creating matrices

<code>[... ; ...]</code>	Typing all values manually within square brackets and indicating a new row with a semicolon. For example, <code>[1 2 3 ; 4 5 6]</code> .
<code>1:3:20</code>	<p>The colon operator</p> <p>This can only be used for creating a matrix with one row; a row vector.</p> <p>The first element will be equal to the first number (1 in the example). Subsequent elements will increase by the number in the middle (the increment; 3 in the example) and the last element will be no higher than 20.</p> <p>The resulting matrix in the example is <code>[1 4 7 10 13 16 19]</code>. 20 is not included as the next element would have been $19+3=22$, which is greater than 20.</p> <p>If the middle number is omitted, Matlab uses an increment of 1. Consequently, <code>4:8</code> is the matrix <code>[4 5 6 7 8]</code>.</p>
<code>linspace(2,9,5)</code>	<p>Linear spacing</p> <p>This creates a matrix with only one row; a row vector. The first element is 2, the last element is 9, and there are 5 linearly spaced elements (i.e., the distance between each is the same).</p> <p>In the example, the resulting matrix is <code>[2 3.75 5.5 7.25 9]</code>. Note that the difference between each consecutive number is 1.75 and that there are 5 elements.</p>
<code>zeros(2,3)</code>	Creates a 2x3 matrix of zeros.
<code>ones(2,3)</code>	Creates a 2x3 matrix of ones.
<code>eye(4)</code>	Creates a 4x4 square matrix with a diagonal of ones and all other elements equal to zero.
<code>rand(5,4)</code>	Creates a 5x4 matrix of random numbers between 0 and 1 (drawn from a uniform distribution).
<code>randn(5,4)</code>	Creates a 5x4 matrix of random numbers between minus infinity and plus infinity (drawn from a standard normal distribution, meaning the numbers will usually be between -3 and 3).
<code>repmat(A,2,3)</code>	Repeats the matrix A twice vertically and three times horizontally.

`[A B], [A;B]`

Concatenation

Assuming that A and B are already defined matrices, this produces a matrix with elements equal to those of A and B. In the first case, the matrices are lined up beside each other, and in the second, they are stacked on top of each other. Note that, A and B must have the same number of rows in the first case, and the same number of columns in the second.

`diag(X)`

Diagonal

This command has different meanings depending on the input matrix, X.

If X is a matrix and not a vector (i.e., if it has at least two rows and two columns), the command creates a row vector containing the values of the diagonal of X, beginning with the top-left element.

if X is a vector, the command creates a square matrix with the values of X on the diagonal and zeros elsewhere.

To see an example of some of these commands, consider the following. (Note that the semicolons suppress all output except the last one.)

```
>> A = 2:2:6; B = linspace(10,16,3); C = zeros(1,3);
>> D = [A ; B ; C]; E = [D eye(3)]
E =
     2     4     6     1     0     0
    10    13    16     0     1     0
     0     0     0     0     0     1
```

A is defined using the colon operator, and is a row vector with the first element equal to 2, the last element no higher than 6, and with an increment of 2 for each element: `[2 4 6]`. B is defined as a row vector with 3 linearly spaced elements, starting with 10 and ending with 16: `[10 13 16]`. C is a 1x3 matrix of only zeros: `[0 0 0]`.

D is defined by concatenating the first three variables on top of each other (since they are separated by semicolons). Finally, E is defined by concatenating D and a 3x3 matrix with ones on the diagonal and zeros elsewhere. The concatenation is horizontal since there are no semicolons between the input matrices. The resulting matrix, E, is displayed in the Command Window.

Lastly, you may note that semicolons have a different meaning in matrices (i.e., when between square brackets) from the one we first learnt. At the end of commands, they suppress output in the Command Window; within a matrix, they indicate new rows.

4.2 Addressing parts of matrices

Sometimes we want to pick out, or change, a subset of the elements of a matrix. There are three different ways of addressing a subset. The first is to use the row and column numbers, the second is to use one single index, and the third is to use conditional statements.

4.2.1 Addressing using row and column numbers

Say we have the matrix `testMatrix` defined earlier. We noted that, for the second location in the third day it rained 3 millimeters (i.e., the corresponding element is 3). To pick this number out from the matrix, note that it is located on the third row and in the third column. Addressing this particular location within the matrix is done by entering the row and column numbers within parentheses after the variable name:

```
>> testMatrix(3,3)
ans =
     3
```

You can also pick out a larger subset of a matrix. Suppose, for instance, that we only want the day numbers and the observations from the third location (i.e., columns one and four). This can be addressed as

```
>> testMatrix([1 2 3],[1 4])
ans =

     1     8
     2     9
     3     9
```

Here, the addressing of the rows and columns of the matrix is done using vectors. The first vector picks out rows 1, 2, and 3, and the second picks out columns 1 and 4. Oftentimes, it is convenient to use the colon operator to create the addressing vectors. In the example, we wanted each row from row 1 to row 3. This can be written as `1:3`, since this creates a row vector starting with 1, ending with 3, and increasing with 1 in each step. (As we have not entered any increment, Matlab assumes an increment of 1.)

If you want to pick out *all* the elements in one dimension (i.e., all rows or all columns), there is an even simpler way to do that. Note that, in the example we pick out all rows from `testMatrix`. In such cases, you may enter a colon, without any start or end indicators. Both of these two methods produce the same result as the previous example.

```
>> testMatrix(1:3,[1 4]), testMatrix(:,[1 4])
ans =

     1     8
     2     9
     3     9

ans =

     1     8
     2     9
     3     9
```

Oftentimes, we want to pick out all elements from some starting row, or column, until the last one. Then there is another simplifying trick where you can refer to the last row or column as `end`. For example, if we want all rows except the last one, this can be addressed as

```
>> testMatrix(1:end-1,[1 4])
ans =

     1     8
     2     9
```

This is, for instance, useful if we want to lag a matrix one or several periods and we do not know the number of rows or columns, as could be the case in a program. (Although, there are other ways to deal with that as well. See, for example, Section 4.4 and the function `size()`.)

4.2.2 Addressing using a single index

Row or column vectors can be addressed with just one index that indicates where the subset is located. For example, if `testVector` is the row vector `[1 3 5 7 9 11]`, then the first four elements can be picked out as

```
>> testVector(1:4)
ans =
     1     3     5     7
```

Consequently, you do not have to specify any row number. This works the same way with column vectors. However, matrices can also be addressed using a single index. In that case, the elements are numbered columnwise, starting with the upper-left element. The first seven elements of `testMatrix` are

```
>> testMatrix(1:7)
ans =
     1     2     3    32     4    19     7
```

As the addressing vector, `1:7`, is a row vector, the answer is also a row vector.

Similarly to addressing using row and column numbers, you can pick out all elements using the colon operator alone. Since there is only one dimension when using a single index, all elements are picked out as a vector.

```
>> testMatrix(:)
ans =
     1
     2
     3
    32
     4
    19
     7
     1
     3
     8
     9
     9
```

In this case, the result is a column vector.

4.2.3 Addressing using conditional statements

Matrix subsets can also be picked out based on criteria regarding the elements themselves. Suppose, for instance, that we want to select all elements of `testMatrix` that are greater than 5. Issuing the statement

```
>> index_gt5 = find(testMatrix > 5)
index_gt5 =
     4
     6
     7
    10
    11
    12
```

produces a column vector, `index_gt5`, with a list of which elements in `testMatrix` that are greater than 5. Note that the numbers in this list correspond to addressing the matrix in the single index style (i.e., numbering them columnwise). We can then use the index to pick out the elements in `testMatrix` that are greater than 5.


```
>> testMatrix(index_gt5)
ans =

    32
    19
     7
     8
     9
     9
```

Of course, you do not have to take the extra step of defining the index variable. You can do the same thing nesting the commands as `testMatrix(find(testMatrix > 5))`.

There is also an alternative strategy for conditional addressing, using logical variables. If we issue the command

```
>> index_gt5_logic = testMatrix > 5
index_gt5_logic =

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

we get a matrix, `index_gt5_logic`, with only zeros and ones. This matrix is a logical variable (see Section 3.5.2). Each element is an individual answer to the question whether the corresponding element in `testMatrix` is greater than 5 or not, where 1 indicates true (yes, it is greater than 5) and 0 indicates false (no, it is not greater than 5). This type of index can also be used to pick out the values in `testMatrix` that are greater than 5.

```
>> testMatrix(index_gt5_logic)
ans =

    32
    19
     7
     8
     9
     9
```

Alternatively, you can nest the commands as `testMatrix(testMatrix > 5)`, which is somewhat shorter than the example using `find()`.

Here you may note a difference between logical variables and ordinary numerical ones. If we change the variable `index_gt5_logic` into a numerical variable by multiplying it with 1, it is no longer possible to use it for addressing.⁵

```
>> index_gt5_log = index_gt5_logic*1;
>> testMatrix(index_gt5_logic)
??? Subscript indices must either be real positive integers
or logicals.
```

4.3 Changing parts of a matrix

The method of addressing a subset of a matrix can also be used to change it. The way to do that is to address the subset that you want to change, followed by the assignment operator (=) and, lastly, what you want to change the subset to. Suppose we want to change the last two observations for the third location in `testMatrix` from 9 and 9 to 7 and 6. The address of those two elements, using row and column numbers, is rows 2 and 3 and column 4. To change those elements, we assign this part the value of a 2x1 matrix with elements 7 and 6:

```
>> testMatrix(2:3,4) = [7 ; 6]
testMatrix =

     1     32     7     8
     2      4     1     7
     3     19     3     6
```

Naturally, the dimensions of the part you want to change (here, 2x1) must be the same as the dimensions of the new data. Note also that you can do the same thing using the single index addressing method:

```
>> testMatrix(11:12) = [7 ; 6]
testMatrix =

     1     32     7     8
     2      4     1     7
     3     19     3     6
```

You can also use conditional addressing to change data. Suppose you are only interested in observations that are less than 10. Then you can set the rest to NaN (not-a-number) by issuing

```
>> testMatrix(testMatrix >= 10) = NaN
testMatrix =

     1    NaN     7     8
     2      4     1     7
     3    NaN     3     6
```

Conditional addressing is also useful when we want to select a certain period from the data. Suppose we only want observations from day 2 and on. Then we first pick out the correct row numbers from the first column, and then use those to select all columns from the selected rows.

```
>> testMatrix(testMatrix(:,1) >= 2,:)
ans =

     2     4     1     7
     3    NaN     3     6
```

4.3.1 Reducing and increasing the size of a matrix

Sometimes you will want to delete parts of a matrix (i.e., reduce its size). Since matrices cannot have empty entries, you can only delete full rows or full columns. To delete a row or a column, you assign that part an empty value (i.e., square brackets with nothing in between them). For example, to delete column two of the test matrix, we enter

```
>> testMatrix(:,2) = []
testMatrix =

     1     7     8
     2     1     7
     3     3     6
```

The addressing picks out all rows of the second column, and then the columns is assigned an empty value (i.e., it is deleted), leaving `testMatrix` a 3x3 matrix.

Increasing the size of a matrix is done automatically if you add one or several elements outside the existing rows or columns. Then Matlab adds enough rows or columns to fit the new elements, and sets the other elements of the new rows or columns to zero. For example, adding day four in row four and column one, automatically adds a fourth row with the first element equal to 4 and the other ones equal to zero.

```
>> testMatrix(4,1) = 4
testMatrix =

     1     7     8
     2     1     7
     3     3     6
     4     0     0
```

4.4 Some special commands for handling matrices

To handle matrices, it is useful to know the following commands.

<code>find(X>3)</code>	Creates a vector of numbers that indicate which elements in <code>X</code> that are greater than 3. (Note: uses single index addressing.) <code>[I, J] = find(X>3)</code> , where you specify that you want <i>two</i> outputs, creates two vectors. One vector, <code>I</code> , with row numbers and another one, <code>J</code> , with column numbers. This is, consequently, an example of a function that works differently depending on which output you ask for.
<code>size(X)</code>	Creates a 1x2 vector where the first element is the number of rows in <code>X</code> and the second element is the number of columns. This is often used in programs when you do not know how large a certain matrix is.
<code>transpose(X)</code>	This flips the columns and rows of the matrix <code>X</code> so that the first row of <code>X</code> becomes the first column of the new matrix, etc. If <code>X</code> , for example, is a 3x5 matrix, then the new one will be a 5x3 matrix. Note: instead of the command <code>transpose(X)</code> , you can write a single quote after the matrix that is to be transposed: <code>X'</code> . This produces the same result but is much simpler.
<code>sort(X)</code>	<code>X</code> sorted from smallest to largest (columnwise). Each column is sorted separately.
<code>sortrows(X, 3)</code>	<code>X</code> sorted as a group from smallest to largest values in the third column. The third column is sorted and observations in the other columns stick to the corresponding values of that column. Entering a negative sorting column number produces a sorting from largest

Note that, character strings are also matrices; character matrices. Therefore, many matrix commands, including all the ones above, operate on strings as well. Try, for instance, `transpose(sort('Matlab'))`.

Suppose now that we want to sort the data in `testMatrix` with respect to how much it has rained in the first location (i.e., the second column). We then enter

```
>> sortrows(testMatrix,2)
ans =

     4     0     0
     2     1     7
     3     3     6
     1     7     8
```

Note that column 2 is now in ascending order. The values in columns 1 and 3 still correspond to the same values in column 2 as they did before, though. Suppose, instead, we enter

```
>> sort(testMatrix)
ans =

     1     0     0
     2     1     6
     3     3     7
     4     7     8
```

Then all columns are sorted independently of each other, and the relations to the observations in the other columns are lost.

4.5 The Workspace Browser and the Variable Editor

The Workspace Browser is an interface to the variables currently defined. Through the interface you can create, view, or change variables, as well as create graphics using these variables.

If you do not have the Workspace Browser open, you can open it by issuing the command `workspace` from the Command Window or by selecting `Desktop > Workspace` from the drop-down menus.

To view or change a variable, double-click its name in the Workspace Browser. This opens the variable in the Variable Editor, as in Figure 4-1 where we have opened `testMatrix`. As you see in the figure, the elements of the four rows and three columns of `testMatrix` are displayed in spreadsheet fashion, similar to, for instance, Excel.


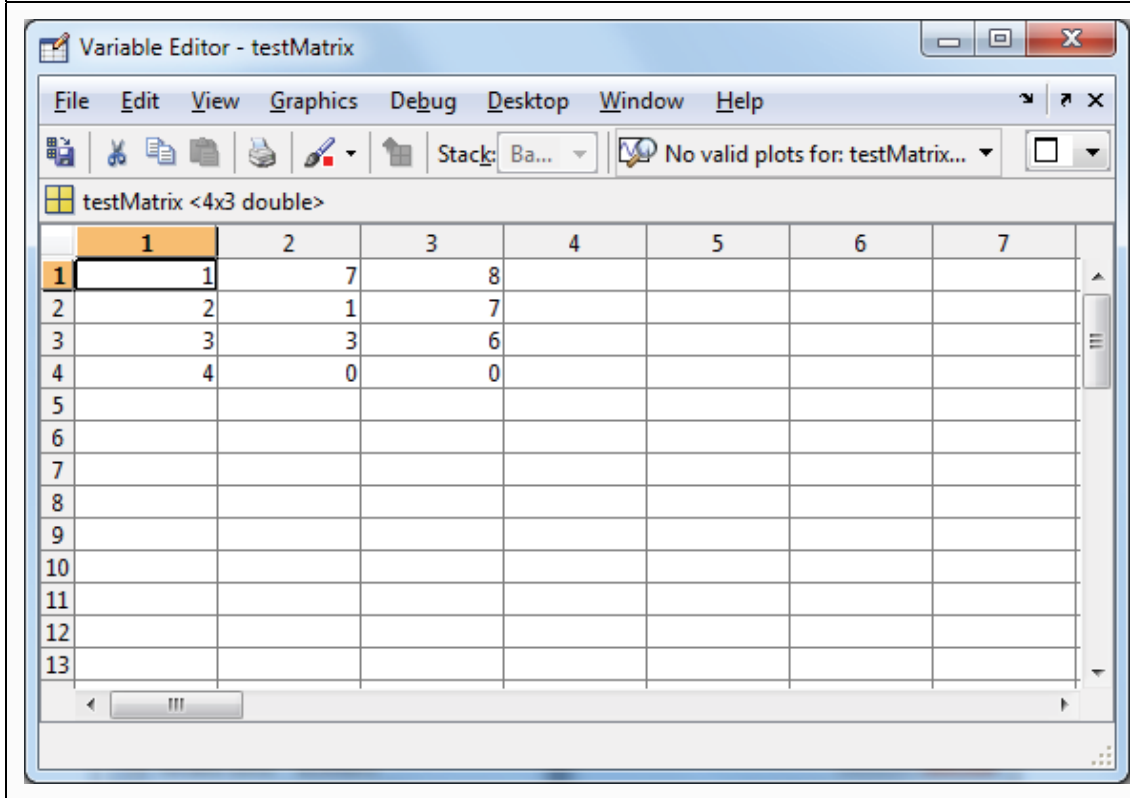
To change an element, you click the cell, enter a new value, and press <Enter>. If you enter values outside the defined range, Matlab will automatically pad the empty rows and columns needed to include the new cell with zeros. To insert rows or columns between existing ones, select the row or column where you want to add one and type <Ctrl>+|. To delete a row or column, select it and type <Ctrl>--. Try, for example, deleting all values in row 4. You close the Variable Editor by clicking  in the upper-right corner. Issue `testMatrix` to see that the matrix has changed.

Figure 4-1: The Variable Editor



The screenshot shows the MATLAB Variable Editor window titled 'Variable Editor - testMatrix'. The window has a menu bar (File, Edit, View, Graphics, Debug, Desktop, Window, Help) and a toolbar with icons for file operations, editing, and plotting. Below the toolbar, it says 'testMatrix <4x3 double>'. The main area is a table with 4 rows and 3 columns. The first row is highlighted in orange. The data in the table is as follows:

	1	2	3	4	5	6	7
1	1	7	8				
2	2	1	7				
3	3	3	6				
4	4	0	0				
5							
6							
7							
8							
9							
10							
11							
12							
13							

4.6 More about matrices

- Matlab also supports a concept called *sparse matrices*. Large matrices can consume a lot of memory and computational power. At the same time, many matrices contain a large number of zeros. Using commands that are tailor made for sparse matrices, is a way to use the fact that many elements are zero to make memory usage and computations more efficient.