MATLAB language

Responsible teacher: Anatoliy Malyarenko

November 5, 2003

Abstract

Contents of the lecture:

- MATLAB programming: a quick start.
- Syntax and data.
- Entering matrices.
- More about matrices.
- Arithmetical operations.
- Logical operations.
- Flow control.
- Functions.

MATLAB Programming: A Quick Start

MATLAB provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file, giving the file a name of `filename.m`. The term you use for filename becomes the new command that MATLAB associates with the program. The file extension of `.m` makes this a MATLAB M-file.

M-files can be scripts that simply execute a series of MATLAB statements, or they can be functions that also accept arguments and produce output. You create M-files using a text editor, then use them as you would any other MATLAB function or command.

The process looks like this

① Create an M-file using a text editor.

② Call the M-file from the command line, or from within another M-file.

Kinds of M-Files

There are two kinds of M-files.

Script M-Files	Function M-Files
Do not accept input arguments or return output arguments	Can accept input arguments and return output arguments
Operate on data in the workspace	Internal variables are local to the function by default
Useful for automating a series of steps you need to perform many times	Useful for extending the MATLAB language for your application

Naming variables

MATLAB variable name must begin with a letter, which may be followed by any combination of letters, digits and underscores. MATLAB distinguishes between uppercase and lowercase characters, so `a` and `A` are not the same variable.

Although variable names can be of any length, MATLAB uses only the first `N` characters of the name and ignores the rest. Here `N` is the number returned by the function `namelengthmax`:

```
>> N=namelengthmax
N =
    63
```

Special values

Several functions return important special values that you can use in your M-files.

Function	Return value
`ans`	Most recent answer
`eps`	Floating-point relative accuracy
`realmax`	Largest floating-point number
`realmin`	Smallest floating-point number
`pi`	3.1415926536897...
`i`,`j`	Imaginary unit
`inf`	Infinity
`NaN`	Not a Number
`computer`	Computer type
`version`	MATLAB version string

Entering matrices

You can enter matrices into MATLAB in several different ways.

- ☞ Enter an explicit list of elements.
- ☞ Load matrices from external data files.
- ☞ Generate matrices using built-in functions.
- ☞ Create matrices with your own functions in M-files.

Example: Dürer's matrix

To enter Dürer's matrix, simply type in the Command Window

```
A=[16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
```

MATLAB displays the matrix you just entered.

```
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
```

Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as `A`.

`sum`, `transpose`, and `diag`

The first statement to try is

```
>> sum(A)
```

MATLAB responses with

```
ans =
    34    34    34    34
```

You have computed a row vector containing the sums of the columns of `A`.

How about the row sums? The easiest way is to transpose the matrix, compute the column sums of the transpose, and then transpose the result. The transpose operation is denoted by the apostrophe `'`, so

```
>> A'
```

produces

```
ans =
    16     5     9     4
     3    10     6    15
     2    11     7    14
    13     8    12     1
```

and

```
>> sum(A')'
```

produces

```
ans =
    34
    34
    34
    34
```

The sum of the elements of the main diagonal is obtained with the `sum` and the `diag` functions.

```
>> diag(A)
```

produces

```
ans =
    16
    10
     7
     1
```

and

```
>> sum(diag(A))
```

produces

```
ans =
    34
```

Subscripts

The element in row `k` and column `l` of `A` is denoted by `A(k,l)`. It is also possible to refer to the elements of row and column vectors with a single subscript, `B(k)`.

If you try to use the value of an element outside of the matrix, it is an error:

```
>> A(4,5)
??? Index exceeds matrix dimensions.
```

The colon operator

The colon operator, `:`, occurs in several different forms. The expression

```
1:10
```

is a row vector containing the integers from 1 to 10

```
1 2 3 4 5 6 7 8 9 10
```

To obtain nonunit spacing, specify the increment. For example

```
100:-7:50
```

is

```
100 93 86 79 72 65 58 51
```

and

```
0:pi/4:pi
```

is

```
0 0.7854 1.5708 2.3562 3.1416
```

Subscript expressions involving colons refer to portions of a matrix.

```
A(1:k,l)
```

is the first k elements of the lth column of A.

The colon by itself refers to *all* the elements in a row or column of a matrix and the keyword end refers to the *last* row or column. So

```
sum(A(:,end))
```

computes the sum of the elements in the last column of A.

Generating matrices

MATLAB provides four functions that generate basic matrices.

zeros	All zeros
ones	All ones
rand	Uniformly distributed random elements
randn	Normally distributed random elements

For example,

```
>> Z=zeros(2,4)
```

produces

```
Z =
     0     0     0     0
     0     0     0     0
```

Concatenation

Concatenation is the process of joining small matrices to make bigger ones. In fact, you made matrix A by concatenating its individual elements. The pair of square brackets, [], is the concatenation operator. For example,

```
B=[A A+32; A+48 A+16]
```

produces

```
    16     3     2    13    48    35    34    45
     5    10    11     8    37    42    43    40
     9     6     7    12    41    38    39    44
     4    15    14     1    36    47    46    33
    64    51    50    61    32    19    18    29
    53    58    59    56    21    26    27    24
    57    54    55    60    25    22    23    28
    52    63    62    49    20    31    30    17
```

Deleting rows and columns

You can delete rows and columns from a matrix using just a pair of square brackets. For example, start with

```
>> X=A;
```

(semicolon `;` suppresses the output). Then, to delete the second column of `X`, use

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

This changes `X` to

```
X =
    16     2    13
     5    11     8
     9     7    12
     4    14     1
```

Operators

Expressions use familiar arithmetic operators and precedence rules.

+	matrix addition
-	matrix subtraction
*	matrix multiplication
/	matrix division
\	matrix left division
^	matrix power
'	matrix complex conjugate transpose
( )	Specify evaluation order

Arrays

When they are taken away from the world of linear algebra, matrices become two-dimensional numeric arrays. Arithmetic operations on arrays are done element-by-element. This means that addition and subtraction are the same for arrays and matrices, but that multiplicative operations are different. MATLAB uses a dot, or decimal point, as part of the notation for multiplicative array operations.

+	addition
-	subtraction
.*	element-by-element multiplication
./	element-by-element division
.\	element-by-element left division
.^	element-by-element power
.'	unconjugated array transpose

For example, if the Dürer matrix is multiplied by itself with array multiplication

```
A.*A
```

the result is an array containing the squares of the integers from 1 to 16, in an unusual order.

```
ans =
   256     9     4   169
    25   100   121    64
    81    36    49   144
    16   225   196     1
```

Relational Operators

MATLAB provides these relational operators.

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to

The MATLAB relational operators compare corresponding elements of arrays with equal dimensions. Relational operators always operate element-by-element. In this example, the resulting matrix shows where an element of `A` is equal to the corresponding element of `B`.

```
A = [2 7 6;9 0 5;3 0.5 6];
B = [8 7 0;3 2 5;4 -1 7];
A == B
ans =
     0     1     0
     0     0     1
     0     0     0
```

For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand. Locations where the specified relation is true receive the value 1. Locations where the relation is false receive the value 0.

Logical Operators

The following logical operators and functions perform element-wise logical operations on their inputs to produce a like-sized output array. The examples shown in the following table use vector inputs `A` and `B`, where

```
A = [0 1 1 0 1];
B = [1 1 0 0 1];
```

Operator	Description	Example
`&`	Returns 1 for every element location that is true (nonzero) in both arrays, and 0 for all other elements.	`A&B=01001`
`|`	Returns 1 for every element location that is true (nonzero) in either one or the other, or both, arrays and 0 for all other elements.	`A|B=11101`
`~`	Complements each element of input array, `A`.	`~A=10010`
`xor`	Returns 1 for every element location that is `true` (nonzero) in only one array, and 0 for all other elements.	`xor(A,B)=10100`

For operators and functions that take two array operands, (`&`, `|`, and `xor`), both arrays must have equal dimensions, with each dimension being the same size. The one exception to this is where one operand is a scalar and the other is not. In this case, MATLAB tests the scalar against every element of the other operand.

Character Arrays

In MATLAB, the term *string* refers to an array of characters.

Specify character data by placing characters inside a pair of single quotes. For example, this line creates a 1-by-13 character array called `name`.

```
name = 'Thomas R. Lee';
```

In the workspace, the output of `whos` shows

```
Name  Size  Bytes  Class

name  1x13     26  char array
```

Flow Control

There are eight flow control statements in MATLAB:

① **`if`**, together with `else` and `elseif`, executes a group of statements based on some logical condition.

② **`switch`**, together with `case` and `otherwise`, executes different groups of statements depending on the value of some logical condition.

③ **`while`** executes a group of statements an indefinite number of times, based on some logical condition.

④ **`for`** executes a group of statements a fixed number of times.

⑤ **`continue`** passes control to the next iteration of a `for` or `while` loop, skipping any remaining statements in the body of the loop.

⑥ **`break`** terminates execution of `for` or `while` loop.

⑦ **`try...catch`** changes flow control if an error is detected during execution.

⑧ **`return`** causes execution to return to the invoking function.

We consider here only 4 first statements.

`if`

`if` evaluates a logical expression and executes a group of statements based on the value of the expression. In its simplest form, its syntax is

```
if logical_expression
    statements
end
```

If the logical expression is `true` (1), MATLAB executes all the statements between the `if` and `end` lines. It resumes execution at the line following the `end` statement. If the condition is `false` (0), MATLAB skips all the statements between the `if` and `end` lines, and resumes execution at the line following the `end` statement.

For example,

```
if rem(a,2) == 0
    disp('a is even')
    b = a/2;
end
```

else

```
if expression
    statements1
else
    statements2
end
```

`else` is used to delineate an alternate block of statements. If *`expression`* evaluates as `false`, MATLAB executes the one or more commands denoted here as *`statements2`*.

In this example, if both of the conditions are not satisfied, then the student fails the course.

```
if ((attendance >= 0.90)
    & (grade_average >= 60))
   pass = 1;
else
   fail = 1;
end;
```

elseif

```
if expression1
    statements1
elseif expression2
    statements2
end
```

If *expression1* evaluates as `false` and *expression2* as `true`, MATLAB executes the one or more commands denoted here as *statements2*.

Here is an example showing `if`, `else`, and `elseif`.

```
for m = 1:k
    for n = 1:k
        if m == n
            a(m,n) = 2;
        elseif abs(m-n) == 2
            a(m,n) = 1;
        else
            a(m,n) = 0;
        end
    end
end
```

For $k = 5$ you get the matrix

```
a =
     2     0     1     0     0
     0     2     0     1     0
     1     0     2     0     1
     0     1     0     2     0
     0     0     1     0     2
```

switch

`switch` executes certain statements based on the value of a variable or expression. Its basic form is

```
switch expression
    case v1
        statements %Executes if expression is v1
    case v2
        statements %Executes if expression is v2
    .
    .
    .
    otherwise
        statements %Executes if expression does
```

```
                %not match any case
end
```

This block consists of:

- The word `switch` followed by an expression to evaluate.
- Any number of case groups. These groups consist of the word `case` followed by a possible value for the expression, all on a single line. Subsequent lines contain the statements to execute for the given value of the expression. These can be any valid MATLAB statement including another `switch` block. Execution of a `case` group ends when MATLAB encounters the next `case` statement or the `otherwise` statement. Only the first matching `case` is executed.
- An optional `otherwise` group. This consists of the word `otherwise`, followed by the statements to execute if the expression's value is not handled by any of the preceding case groups. Execution of the `otherwise` group ends at the `end` statement.
- An `end` statement.

The code below shows a simple example of the `switch` statement. It checks the variable `input_num` for certain values. If `input_num` is -1, 0, or 1, the `case` statements display the value on screen as text. If `input_num` is none of these values, execution drops to the `otherwise` statement and the code displays the text 'other value'.

```
switch input_num
    case -1
        disp('negative one');
    case 0
        disp('zero');
    case 1
        disp('positive one');
    otherwise
        disp('other value');
end
```

while

The `while` loop executes a statement or group of statements repeatedly as long as the controlling expression is `true` (1). Its syntax is

```
while expression
    statements
end
```

For example, this `while` loop finds the first integer n for which $n!$ (n factorial) is a 100-digit number.

```
n = 1;
while prod(1:n) < 1e100
    n = n + 1;
end
```

for

The `for` loop executes a statement or group of statements a predetermined number of times. Its syntax is:

```
for index = start:increment:end
    statements
end
```

The default increment is 1. You can specify any increment, including a negative one. For positive indices, execution terminates when the value of the index exceeds the end value; for negative increments, it terminates when the index is less than the end value.

For example, this loop executes five times.

```
for i = 2:6
    x(i) = 2*x(i-1);
end
```

You can nest multiple `for` loops.

```
for i = 1:m
    for j = 1:n
        A(i,j) = 1/(i + j - 1);
    end
end
```

Functions

Here we show the basic parts of a function M-file, so you can familiarise yourself with MATLAB programming and get started with some examples.

```
function f = fact(n) % Function definition line
% FACT Factorial.    % H1 line
% FACT(N) returns the factorial % Help text
% of N, H!, usually denoted by N!
% Put simply, FACT(N) is PROD(1:N).

f = prod(1:n);       % Function body
```

This function has some elements that are common to all MATLAB functions:

- *A function definition line.* This line defines the function name, and the number and order of input and output arguments.
- *An H1 line.* H1 stands for "help 1" line. MATLAB displays the H1 line for a function when you use `lookfor` or request help on an entire directory.
- *Help text.* MATLAB displays the help text entry together with the H1 line when you request help on a specific function.
- *The function body.* This part of the function contains code that performs the actual computations and assigns values to any output arguments.

Function Workspaces

Each M-file function has an area of memory, separate from the MATLAB base workspace, in which it operates. This area is called the function workspace, with each function having its own workspace context.

While using MATLAB, the only variables you can access are those in the calling context, be it the base workspace or that of another function. The variables that you pass to a function must be in the calling context, and the function returns its output arguments to the calling workspace context.

Problems

1. Write the following script. Execute it. Add your own comments with explanation of results.

```
x=[2 3 4 5]
y=-1:1:2
x.^y
x.*y
x./y
```

2. Calculate square roots of integer numbers from 1 to 1000.
3. (For pass with distinction). Given a vector $\mathbf{x}$ with n elements, write a MATLAB function with $\mathbf{x}$ as input parameter to form the vector $\mathbf{p}$ with elements

$$p_k = x_1 x_2 \ldots x_{k-1} x_{k+1} \ldots x_n,$$

that is, p_k will contain the products of all the vector elements except the kth. Call your function from the Command window using the value $\mathbf{x} = [1\ 2\ 3]$. Call it once more using the value $\mathbf{x} = 1$. Call it once more using the value $\mathbf{x} = [\ ]$.