

Programming

Until now we worked with Matlab interactively, executing simple statements line by line, often reentering the same sequences of commands. Alternatively, we can store the Matlab input commands in a file called an *M-file*, an ordinary ASCII text file with the extension “.m”, (e.g., *myfile.m*). M-files can be created using your favorite text editor (*emacs*, *vi*) or a built-in Matlab editor. There are two types of M-files: *script* files and *function* files.

1. Script files.

A script file is an M-file which contains a set of Matlab commands. A script file is executed by typing the name of the file without the “.m” extension from the Matlab command line. It should be “visible” to the Matlab environment - by changing the *Current directory* to the place where the script was saved using the Matlab GUI; or from the command line, check **help cd**.

Example: script which solves quadratic equations.

```
%----- solve quadratic equation -----  
%      a*x^2+b*x+c=0  
%      x1,x2=(-b+-sqrt(b^2-4ac))/(2*a)  
clc;    % clear command window  
a=2;  
b=-1;  
c=5;  
sqd=sqrt(b^2-4*a*c);  
x1=(-b-sqd)/2/a      % no semicolon  
x2=(-b+sqd)/2/a      % to see results
```

Notice, that all variables in the script files are *global* and accessible from the workspace after the script execution. Thus, never name a script file the same name as a variable it computes!

In some cases, it is convenient to enter input data interactively using the `input` command. Instead of a line `a=2;` we put `a=input(' enter a: ');` and Matlab will ask to input the number (or expression, which will be evaluated).

2. Function files.

A function file is similar to the subroutine in Fortran or function in C. It is also an M-file, but all variable in a function file are local.

Consider a function which solves quadratic equations, similar to the script above. The first line in a function file is a function definition which begins with the word *function*. The name of the function (in this case `quad_eq_sol`) must be the same as the filename in which the function is defined, without the “.m” extension. Also the set of comment lines after the function definition is special: they are displayed by the command `help quad_eq_sol`. The first comment line, called the “H1” line, is the line which is scanned by the lookfor command:

```
>> lookfor quad_eq_sol
quad_eq_sol Solves quadratic equation
```

The rest of the M-file is the body of the function.

H1 line

output list

function name

input list

```

function [x1,x2]=quad_eq_sol(a,b,c)
% quad_eq_sol Solves quadratic equation
% May, 2001
% simple function example
sqd=sqrt(b^2-4*a*c);
x1=(-b-sqd)/2/a;
x2=(-b+sqd)/2/a;

```

body of the function

Now our function can be executed, with the output explicitly specified:

```

>> clear
>> a1=1; b1=5; c1=-3;
>> [root1,root2]=quad_eq_sol(a1,b1,c1);
>> root1,root2
root1 =
    -5.5414
root2 =
     0.5414

```

Note, the variables that are passed to the function do not need to have the same name as in the function definition.

If the output is omitted and there is no semicolon at the end of the call statement,

```

>> quad_eq_sol(a1,b1,c1)

```

```
ans =  
-5.5414
```

the first variable in the output list is displayed.

Any function, built-in or user-written, can be executed inside of another function - in the same way we used `sqrt` to calculate the square root. Matlab allows several functions to be written in a single file. The first function in a such file, the main function, is like a regular function. But all functions, written below the first function, so called *subfunctions*, are accessible only to the main function, and cannot be called from the outside.

3. Language elements.

Matlab, as a programming language, has several standard control-flow constructions to specify the order in which computations are performed.

if-elseif-else

example:

```
if x<= -3  
    f=-1;  
elseif (x>-3) & (x<3)  
    f=0;  
else  
    f=1;  
end
```

Six relational operations support if-elseif-else construction:

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

and for logical:

- &** logical AND
- |** logical OR
- ~** logical complement (NOT)
- xor** exclusive OR.

These operations work not only with scalars, but also with vectors and matrices of the same size. For instance,

```
>> u=[1 3 7 9]; v=[0 5 7 8];  
>> a1= u > v, a2 = u==v, a3= (u>v) & (u<5)  
a1 =  
     1     0     0     1  
a2 =  
     0     0     1     0  
a3 =  
     1     0     0     0
```

The result is 1 if the relation is true, or 0 where it is false.

for loops

example:

```
for i=0:5:100  
    y=x+5*i  
end
```

The counter of the loop is *initial:step:last*. Nested loops are allowed.

while loops

the pattern:

```
while condition  
    --statements--  
end
```

The loop will execute a group of statements while the condition is satisfied (true).

example: find the sum of all odd numbers less than 100.

```
s=0; i=1;
while i<100
    s=s+i;
    i=i+1;
end
```

break

This command inside *for* or *while* loop terminates the execution of the loop without checking any other conditions. In the case of a nested loops, *break* terminates only the innermost loop.

error

The command `error('some error message')` inside a script or a function aborts the execution and displays the error message.

Note, that *for* or *while* loops are much slower than operations on vectors and matrices in Matlab. As an example, consider problem 4b from the last Exercises set. We can use the command `cputime` to get the CPU time in seconds for each method. First, using *for* loop:

```
t=cputime; n=50000; s=0;
for i=1:n
    s=s+(-1)^(i+1)/i;
end;
s, cputime-t
```

after running this script we get:

```
s =
    0.6931
```

```
ans =  
    2.6040
```

Using vectorized operations:

```
t=cputime; n=50000;  
repmat([1 -1],1,n/2)*(1./(1:n))'  
cputime-t
```

gives

```
ans =  
    0.6931
```

```
ans =  
    0.1400
```

which is ~18 times faster.

EXERCISES

1. The sequence of *Fibonacci* numbers begins with the integers 1, 1, 2, 3, 5, 8, 13, 21, ...

where each number after the first two is the sum of the two preceding numbers. Write a script/function that displays the vector with first n Fibonacci numbers.

2. Write a script/function to evaluate

a) $e^x = \sum_{i=0}^N \frac{x^i}{i!}$ for $x = -25, -45$, and $N = 25, 50, 75, \dots, 150$

b) $e^{-x} = \frac{1}{\sum_{i=0}^N \frac{x^i}{i!}}$, $x = 25, 45$, and the same values of N as above.

3. Suppose $\mathbf{x}=[-2\ 0\ 0\ 3\ 6\ 9]$ and $\mathbf{y}=[-6\ -1\ 0\ 4\ 5\ 12]$
Check the result of the following operations:

- a) $\mathbf{u1} = \mathbf{x}\&\mathbf{y}$
- b) $\mathbf{u2} = \mathbf{x}|\mathbf{y}$
- c) $\mathbf{u3} = \mathbf{x}\geq\mathbf{y}$
- d) $\mathbf{u4} = \mathbf{x}\sim\mathbf{y}$

4. Input/Output

First, we discuss the export/import of the numeric data files. The simplest way to write some data to the file is the command **save**. For example, to export a small table of the sine function

```
>> x=linspace(0,2*pi,6); y=sin(x); Sine=[x' y']  
Sine =  
      0      0  
  1.2566  0.9511  
  2.5133  0.5878  
  3.7699 -0.5878  
  5.0265 -0.9511  
  6.2832 -0.0000
```

we issue the following command:

```
>> save SineFile.dat Sine -ASCII
```

which creates the file **SineFile.dat** in the current working directory. To check what is in the file, type **more SineFile.dat** from the Unix prompt:

```
0.0000000e+000  0.0000000e+000  
1.2566371e+000  9.5105652e-001  
2.5132741e+000  5.8778525e-001
```

```
3.7699112e+000 -5.8778525e-001
5.0265482e+000 -9.5105652e-001
6.2831853e+000 -2.4492936e-016
```

The **-ASCII** option specifies 8-digits plain text output instead of a binary (the default). Check **help save** for more options.

If we need to import the data from the file **SineFile.dat**, we use the command **load**:

```
>> A=load('SineFile.dat'); A
A =
           0           0
    1.2566    0.9511
    2.5133    0.5878
    3.7699   -0.5878
    5.0265   -0.9511
    6.2832   -0.0000
```

The table of the sine function is represented now by the matrix **A**.

Another useful command which writes *formatted* data to a file or to the screen is **fprintf**. Let's write the matrix **A** above to the file **sine2.txt**:

```
>> fid1=fopen('sine2.txt','w');
>> fprintf(fid1,'%12.5e %12.5e\n',A);
>> fclose(fid1);
```

These lines need some explanations, as an example of a *low-level* I/O facilities in Matlab. First, **fopen** opens a file **sine2.txt** in the *write* mode **'w'**, and assigns the *file identifier* (similar to the file unit number in Fortran, i.e., `write(unit=fid,*)`) to **fid1**. The **fprintf** command writes the data to the file identified by **fid1**. The formatting of the data

is based on the rules of the C-language I/O syntax, with some Matlab extensions, which allow, in particular, an explicit output of matrices/vectors. The string `'%12.5e %12.5e\n'` reads: two numbers on a line (`\n` stands for new line), using for each number the floating-point conversion. The *minimum field width* is **12**, and the *precision*, or number of digits after the decimal point, is **5**. Since the format string is recycled through the elements of **A** columnwise by default, **A** was transposed. Finally, the last statement closes the file.

If is file identifier is omitted (or equal to 1), the output of `fprintf` goes to the screen. In the next example `fprintf` displays the table of the sine function on the screen in a Fortran/C style. The following script

```

clc
x=linspace(0,2*pi,6);
y=sin(x);
fprintf('\n');
fprintf('      x                sin(x)\n');
fprintf('-----\n');
for i=1:6
    fprintf(' %10.5f  %10.5f\n',x(i),y(i));
end
fprintf('-----\n');

```

prints

x	sin(x)
0.00000	0.00000
1.25664	0.95106
2.51327	0.58779
3.76991	-0.58779
5.02655	-0.95106

```
6.28319    -0.00000
```

With `fprintf` it is also easy to mix numbers with text:

```
>> fprintf('The sine of Pi/4 is:\n...  
sin(%6.4f)=%6.4f\n', pi/4, sin(pi/4))
```

```
The sine of Pi/4 is:  
sin(0.7854)=0.7071
```

Matlab also provides the `disp` command, a higher-level command than `fprintf`. `disp` is easy to use, but it has limited control over the appearance of the output. In the simplest case, `disp` displays the contents, but not the name of the array:

```
>> A=reshape(1:6,2,3); disp(A)  
1     3     5  
2     4     6
```

or

```
>> x=(1:3), y=(2:4)  
x =  
1     2     3  
y =  
2     3     4  
>> disp([x' y'])  
1     2  
2     3  
3     4
```

The mixed text-numbers output is possible with the **disp** command, but first we need some information about *character strings*. In Matlab character strings are entered within two single quotes, and treated as a row vector with one element per character, including spaces.

```
>> line='I"m a string'
line =
I"m a string
>> size(line)
ans =
     1     12
```

The row vector with strings as its components concatenates all strings:

```
>> line1=['one' 'two' 'three']
line1 =
onetwothree
```

Thus, to mix text with numbers, we should first transform numbers to strings, which is accomplished by the function **num2str**:

```
>> disp(['Pi = ' num2str(pi)])
Pi = 3.1416
>> num2str(pi,12) % to get 12 digits
ans =
3.14159265359
>> x=1:3; disp(['x-vector: ' num2str(x)])
x-vector: 1 2 3
```

but to display a column vector:

```
>> y=(2:4)'; disp('y='); disp(y);    % !with ;
y=
    2
    3
    4
```

The disp-version of the script which prints the sine function table:

```
clc
x=linspace(0,2*pi,6);
y=sin(x);
disp(' ');
disp('      x          sin(x) ');
disp('-----');
for i=1:6
    disp(['      ' num2str(x(i)) '      ' ...
          num2str(y(i)) '  ']);
end
disp('-----');
```

which produces

x	sin(x)
0	0
1.2566	0.95106
2.5133	0.58779
3.7699	-0.58779
5.0265	-0.95106
6.2832	-2.4493e-016

It is not as good as with **fprint**.

5. 2D Plots.

Matlab has powerful tools for visualization which include high-level 2D and 3D color graphics and animation functions.

plot is the most useful command for producing simple 2D plots.

The general pattern of the command is

plot(xvalues,yvalues,'style-options'),

where **style-options** is a character string made from one element from any or all the following 3 columns:

y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dashdot
r	red	+	plus	--	dashed
g	green	*	star		
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

The following script illustrates several elements of the Matlab plotting environment:

```

% script plots sums of Taylor series terms of
%   sin(x)
%
%   sin(x)=x-x^3/3!+x^5/5!-...
%
clf
x=linspace(0,2*pi,100);
ysin=sin(x);
y1=x;
y3=x-x.^3/6+x.^5/120;
plot(x,ysin,x,y1,'--',x,y3,'o')
axis([0 5 -1 5])
xlabel('x')
ylabel('y=sin(x)')
title('Series approximations of sin(x)')
text(3.5,0,'sin(x)')
text(2.5,3,'x')
text(2.8,1.5,'x-x^3/6+x^5/120')

```