Bil 108

Introduction to the Scientific and Engineering Computing with MATLAB

Lecture 4

F. Aylin Konuklar Lale T. Ergene

Overview

Script m-files

- Creating
- Side effects

Function m-files

- Syntax of I/O parameters
- Text output

Flow control

- Relational operators
- Conditional execution of blocks
- Loops

Preliminaries

- Programs are contained in **m**-files
 Plain text files – not binary files produced by word processors
- File must have "**.m**" extension
- m-file must be in the path
 Matlab maintains its own internal path
- The path is the list of directories that Matlab will search when looking for an m-file to execute.
- Manually modify the path with the **path**, **addpath**, and **rmpath** built-in functions.

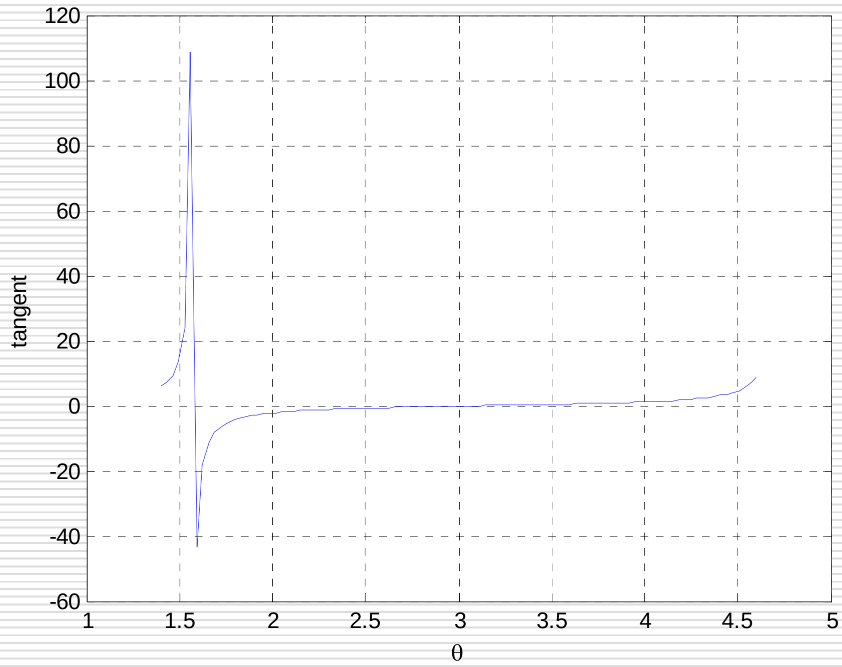
Script Files

- Not really programs
- No **input/output** parameters
- Script variables are part of workspace

- Useful for tasks that never change
- Use a script to run function for specific parameters required by the assignment
- Eg.1 MyCON

Example Tanplot.m

```
theta = linspace (1.4 , 4.6);
tandata = tan(theta);
plot (theta, tandata);
xlabel('\theta');
ylabel('tangent');
grid;
```



Save as tanplot.m

```
>> tanplot
```

Script Side-Effects

- All variables created in a script file are added to the workplace.
- This may have undesirable effects because
  - Variables already existing in the workspace may be overwritten
  - Eg.3 easyplot.m

Function m-files

- Functions are subprograms:
- Functions use input and output parameters to communicate with other functions and the command window
- Functions use *local* variables that exist only while the function is executing. Local variables are distinct from variables of the same name in the workspace or in other functions.
- Input parameters allow the same calculation procedure (same algorithm) to be applied to different data. Thus, function m-files are *reusable*.
- Functions can call other functions.

Function Input and Output

- **twosum.m**

```
function twosum(x,y)
% twosum Add two matrices
% and print the result
x+y
```

threesum.m

```
function s = threesum(x,y,z)
% threesum Add three variables
% and return the result
s = x+y+z;
```

Eg.4 ave_3.m

Eg.5 triangle.m

- Eg.6 Addmult

addmult.m

```
function [s,p] =addmult(x,y)
% addmult Compute sum and product
% of two matrices
s = x+y;
p = x*y;
```

Sample Program

- 1. Write a function called FtoC (ftoc.m) to convert Fahrenheit temperatures into Celsius.

```
function C=FtoC(F)
% Celsius=FtoC(Fahrenheit)
% Converts Fahrenheit temperatures to Celsius
C=5*(F-32)/9;
```

Test

- FtoC(96)

Summary of Input and Output Parameters

- Values are communicated through input arguments and output arguments.

- Variables defined inside a function are local to that function. Local variables are invisible to other functions and to the command environment.

- The number of return variables should match the number of output variables provided by the function.

Text Input and Output

- It is usually desirable to print results to the screen or to a file.
- Inputs to functions:
  - input function can be used .
  - Input parameters to functions are preferred.

- Text output from functions:
  - disp function for simple output
  - fprintf function for formatted output.

The fprintf function

- You can control the exact way in which values are printed to the screen with the ‘fprintf()’ function (fprintf= “file print formatted”).

- This function takes one argument representing the formatting instructions, followed by a list of values to be printed. Embedded within the format string are ‘percent commands’ which control where and how the values are to be written.

- The *outFormat* string specifies how the *outVariables* are converted and displayed. The *outFormat* string can contain any text characters.

- It also must contain a *conversion code* for each of the *outVariables*. The following list shows the basic conversion codes.

- Code Conversion instruction
- %s format as a string
- %d format with no fractional part (integer format)
- %f format as a floating-point value
- %e format as a floating-point value in scientific notation
- %g format in the most compact form of either %f or %e
- \n insert newline in output string
- \t insert tab in output string

- Eg.7 fprint

Examples

The command %g represents a general real number, %f means a fixed point number, %d a decimal integer, and %s a string. You can put numeric values between the ‘%’ and the letter to control the field width and the number of digits after the decimal point. For example (□=space):

- fprintf('%5g',10)□□□10
- fprintf('%10.4f',123.456)□□123.456□
- fprintf('%10s', 'fred')□□□□□□fred

- You can input a value or a string from the command line with the ‘input()’ function:
- yval=input('Enter a number: ');
- name=input('Enter your name: ', 's');

The format function

- The **format** function controls the precision of **disp** output.
- >> format short
- >> disp(pi)
- 3.1416
- >> format long
- >> disp(pi)
- 3.14159265358979

Flow Control

- To enable the implementation of computer algorithms, a computer language needs control structures for

1. Repetition: looping or iteration
2. Conditional execution: branching
3. Comparison

Comparison

- Comparison is achieved with *relational operators*. Relational operators are used to test whether two values are equal, or whether one value is greater than or less than another. The result of a comparison may also be modified by *logical operators*.

Relational Operators

- Relational operators are used in comparing two values.

Operator Meaning

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

- The result of applying a relational operator is a logical value, i.e. result is either true or false.

- In Matlab any nonzero value, including a non-empty string, is equivalent to true. Only zero is equivalent to false.

- Note: The <=, >=, and ~= operators have "=" as the second character. =<, => and =~ are not valid operators.

Logical and Relational Operators

- **Summary:**
- Relational operators involve comparison of two values.
- The result of a relational operation is a logical (True/False) value.
- Logical operators combine (or negate) logical values to produce another logical value.
- There is always more than one way to express the same comparison

Conditional Execution

Conditional Execution or Branching:

- As the result of a comparison, or another logical (true/false)test, selected blocks of program code are executed or skipped.
- Conditional execution is implemented with if, if...else, and if...elseif constructs, or with a switch construct.
- There are three types of if constructs

1. Plain **if**
2. **if...else**
3. **if...elseif**

The switch Construct

A switch construct is useful when a test value can take on discrete values that are either integers or strings.

Syntax:

- switch *expression*
- case *value1*,
- *block of statements*
- case *value2*,
- *block of statements*
- …
- otherwise,
- *block of statements*
- end

Example:

```
color = '...'; % color is a string
switch color
case 'red'
disp('Color is red');
case 'blue'
disp('Color is blue');
case 'green'
disp('Color is green');
otherwise
disp('Color is not red, blue, or green');
end
```

Flow Control
Repetition or Looping

- A sequence of calculations is repeated until *either*

All elements in a vector or matrix have been processed

OR

The calculations have produced a result that meets a predetermined termination criterion

- Looping is achieved with for loops and while loops.

for loops

- for loops are most often used when each element in a vector or matrix is to be processed.

Syntax:

for index = *expression*

block of statements

end

Example: *Sum of elements in a vector*

```
x = 1:5; % create a row vector
sumx = 0; % initialize the sum
for k = 1:length(x)
   sumx = sumx + x(k);
end
```

for loop variations

Example: *A loop with an index incremented by two*

```
for k = 1:2:n
    ...
end
```

Example: *A loop with an index that counts down*

```
for k = n:-1:1
   ...
 end
```

Example:

```
% My first loop
for i=1:10 % start at 1 and go to 10
disp([i,1/i]) % print the integers and their reciprocals
end % the end of every for loop must be "closed" using the end
    statement
```

Nested Loops

- Loop and branch statements can also be *nested*. Try this,

```
% Nested loops
 for i=1:4
     for j=1:3
        disp([i j i*j])
      end
 end
```

A final word of caution, *never* assign a value to the loop variable inside of the loop.

while loops

- while loops are most often used when an iteration is repeated until some termination criterion is met.

Syntax:

while *expression*

block of statements

end

- The *block of statements* is executed as long as *expression* is true.

Example

we'll add all the reciprocals of the positive integers until the current reciprocal is smaller than .001 . When the current reciprocal is smaller than .001 then the condition of the while statement will no longer be true and the while loop will terminate.

```
i = 1; % start the counter at 1

total = 0; % initialize the sum

while(1/i >= 1e-3) % continue as long as 1/i is at least .001

   total = total + 1/i; % add the next reciprocal to the sum

   disp([i 1/i total]) % display the number of loops, 1/i, and the current
sum

   i = i + 1; % add 1 to the counter

end
```

Example

The function sin(x) can be written as a Taylor series by

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$$

Write a user-defined function file that calculates sin(x) by using Taylor's series. For the function name and arguments use y=Tsin(x,n). The input arguments are the angle x in degrees, and n the number of terms in the series. Use the function to calculate sin(150°) using 3 and 7 terms.

example

- Write an m file to determine the sum of the series given below converges to the natural logarithm of 2. Determine the number of terms to achieve the relative error of 5 %. Plot the change in error with increasing number of terms.

$$\sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k}$$