Scientific Computing Lecture Series
Introduction to MATLAB Programming

Eda Oktay*

*Scientific Computing, Institute of Applied Mathematics

Lecture II
Scripts and Functions, Control Loops and Advanced Data Structures

Lecture II–Outline

M-files

- Text files containing MATLAB programs can be called from
  - the command line
  - the M-files
- Two kind of M-files:
  - Scripts
  - Functions

A Precaution

- Be careful naming files!

  It's easy to get unexpected results:

  - if you give the same name to different functions
  - if you give a name that is already used by MATLAB

- Check new names with the command which.

- It is also useful to include some error checking in your functions.

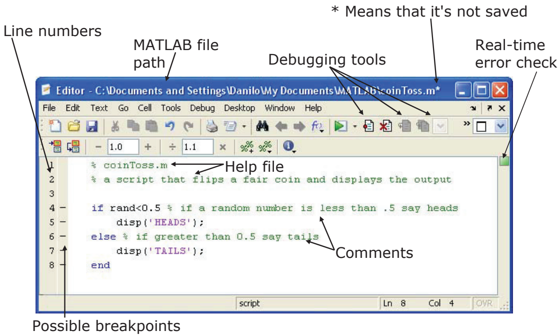
M-Files: Scripts

- Scripts are
  - collection of commands executed in sequence
  - written in the MATLAB editor
  - saved as MATLAB files (.m extension)
- To create an MATLAB file from command-line

```
>> edit helloWorld.m
```

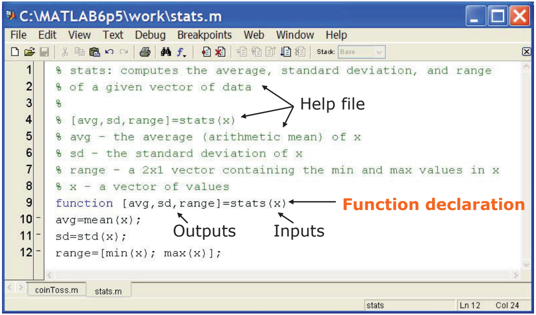
- To open scripts from command window

```
>> open helloWorld.m
```



- COMMENT!
  - Anything following a % is seen as a comment.
  - The first contiguous comment becomes the script's help file.
  - Comment thoroughly to avoid wasting time later.
- Note : Scripts are somewhat static, since there is no input and no explicit output.
- All variables created and modified in a script exist in the workspace even after it has stopped running.

M-Files: Functions



- Functions look exactly like scripts, but for ONE difference: Functions must have a function declaration:

function outArguments = NameOfFunAsYouLike(inArguments)

- Variable scope: Any variables created within the function but not returned disappear after the function stops running.

Input

- input prompt the user to input a number or string

```
>> input('Enter a number:', 's')
Enter a number: 5
ans = 5
```

- If a character or string input is desired, 's' must be added after the prompt.

```
>> name = input('Enter a name:  ')
Enter your name:  Mehmet
Error using input
Undefined function or variable 'Mehmet'.

>> name = input('Enter a name:  ','s')
Enter your name:  Mehmet
name = Mehmet
```

Number of Inputs/Outputs

- Query number of inputs passed to a function
  - nargin
  - Do not try to pass more than in function declaration
- Determine number of outputs requested from function
  - nargout
  - Do not request more than in function declaration

```
function [o1,o2,o3] = narginout ex(i1,i2,i3)
    fprintf('Number inputs = %i;\t',nargin);
    fprintf('Number outputs = %i;\n',nargout);
    o1 = i1; o2=i2; o3=i3;
end
```

```
>> narginout ex(1,2,3);
Number inputs = 3; Number outputs = 0;
>> [a,b]=narginout ex(1,2,3);
Number inputs = 3; Number outputs = 2;
```

Length of Input/Output Argument List

- Input-output argument list length unknown or conditional
  - Think of plot, get, set and the various Name-Property pairs that can be specified in a given function call
- varargin, varargout allow number of inputs and outputs to be determined by the function call

```
funtion [varargout] = circ(varargin)
    r = zeros(nargin,1);
    for in = 1:nargin
        r(in) = varargin{in};
    end
    diam = r*2;
    area = pi*(r.^2);
    varargout = {diam,area};
end
```

- Suppose we want to write a function that returns the color specification for blue, in either the RGB color model (by default) or the HSV model:

```
function b = blue(varargin)
if nargin < 1
    varargin = {'rgb'};
end
switch(varargin{1})
    case 'rgb'
           b = [0 0 1];
    case 'hsv'
         b = [2/3 1 1];
otherwise
     error('Unrecognized color model.')
end
```

Anonymous Functions

- Functions without a file
  - Stored directly in function handle
  - Store expression and required variables
  - Zero or more arguments allowed
  - Nested anonymous functions permitted
- Array of functions handle not allowed; function handle may return array

```
>> f = @(x,y) x^2 + y^2;
>> f(1,2)
ans = 5

>> ezplot(@(x,y) x.^4 + y.^4 -1,[-1,1])
>> ezsurf(@(x,y) exp(-x.^2 -2*y.^2))
```

Local Functions

- A given MATLAB file can contain multiple functions:
- The first function is the `main` function
  - Callable from anywhere, provided it is in the search path
- Other functions in file are local functions
  - Only callable from main function or other local functions in same file
  - Enables modularity (large number of small functions) without creating a large number of files
  - Unfavorable from code reusability standpoint

Local Function Example

- Contents of loc_func_ex.m

```
function main out = loc_func_ex()
    main out = ['I can call the ',loc func()];
end

function loc_out = loc_func()
    loc_out = 'local function';
end
```

- Command-line

```
>> loc_func_ex()
ans =
I can call the local function

>> ['I can''t call the ',loc_func()]
??? Undefined function or variable 'loc_fun
```

1 Scripts and Functions

2 Control Loops

3 Advanced Data Structures

Rational and Logical Operators

- Boolean values: zero is false, nonzero is true
- Some of the logical operators:

Operator	Meaning
$<, <=, >, >=$	less than, less than or equal to, etc.
$==$, $\sim=$	equal to, not equal to
&	logical AND
$\vert$	logical OR
$\sim$	logical NOT
all	all true
any	any true
xor	Xor

Logical Indexing

- Construct a matrix R

```
>> R = rand(5)
R =
0.8147    0.0975    0.1576    0.1419    0.6557
0.9058    0.2785    0.9706    0.4218    0.0357
0.1270    0.5469    0.9572    0.9157    0.8491
0.9134    0.9575    0.4854    0.7922    0.9340
0.6324    0.9649    0.8003    0.9595    0.6787
```

- Test for some logical cases

```
>> R(R<0.15)'
ans =
0.1270    0.0975    0.1419    0.0357
>> isequal(R(R<0.15), R(find(R<0.15)))
ans =
1
```

If/Else/Elseif

- The general form of the if statement is

```
if     expression1
           statements1
elseif     expression2
           statements2
           ⋮
else
           statements
end
```

- No need for parentheses:command blocks are between reserved words

Switch

- The general form of the switch statement is

```
switch    variable
  case    variable value1
        statements1
  case    variable value2
        statements2
        ⋮
  otherwise    (for all other variable values)
        statements
end
```

Try–Catch

- The general form:

```
try
    statements1
catch
    statements2
end
```

- A simple example:

```
a = rand(3,1);
try
    x = a(10);
catch
    disp('error')
end
```

For

- for loops: use for a known number of iterations
- The basic syntax is

```
for  variable = expr
     statements;
end
```

- A simple example:

```
M = rand(4,4); suma = 0;
for i = 1:4
    for j = 1:4
        suma = suma + M(i,j);
    end
end
fprintf('sum = %d\n',suma);
```

While

- Don't need to know number of iterations
- The basic syntax is

```
while   a logical test
        commands to be executed
        when the condition is true
end
```

- A simple example:

```
S=1; n=1;
while S+(n+1)^2 < 100
    n=n+1; S=S+n^2;
end
>> [n,S]
ans = 6    91
```

- Beware of infinite loops!

Remarks

- break - immediately jumps execution to the first statement after the loop.
- return - immediately end a functions routine.
- Precaution: Avoid i and j if you are using complex values.
- Loops are very inefficient in MATLAB. Only one thing to do: AVOID THEM !!!
- Try using built–in–functions instead
- Allocating memory before loops greatly speeds up computation times !!!

Find

- find returns indices of nonzero values. It can simplify code and help avoid loops
- basic syntax: index = find(condition)

```
>> x = rand(1,10)
x =
Columns 1 through 5
0.4505    0.0838    0.2290    0.9133    0.1524
Columns 6 through 10
0.8258    0.5383    0.9961    0.0782    0.4427

>> inds = find(x>0.4 & x<0.7)
inds =
1     7    10
>> x(inds)
ans =
0.4505    0.5383    0.4427
```

Scoping Exceptions

- A global variable is a factor whose value can be accessed and changed from any other workspaces
- Any variable may be declared global
- The trouble with global variables is that they do not scale well to large or even moderately sized projects

- A persistent variable is a factor whose value is preserved between invocations to that particular function.
- Any variable may be declared global
- It is less general than a global variable and requires a little care to ensure correct use

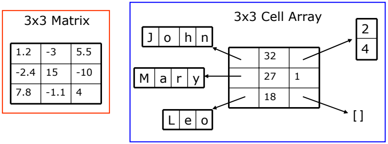
- Persistent variables can be used to record information about a function's internal state, or to preserve costly preliminary results that can be reused later.
- Compute the Fibonacci numbers:

```
function y = fib(n)
persistent f
if length(f) < 2,
    f = [1 1];
end
for k = length(f)+1:n
    f(k) = f(k-2) + f(k-1);
end
y = f(1:n);
```

- In future calls to fib, any previously computed members of the sequence are simply accessed rather than recomputed.

Cell Arrays

- Cell arrays are a mechanism for gathering dissimilar objects into one variable.
- Indexed like regular numeric arrays, but their elements can be anything, including other cell arrays.
- Cell arrays can have any size and dimension, and their elements do not need to be of the same size or type.
- Because of their generality, cell arrays are mostly just containers
- Created or referenced using curly braces {} rather than parentheses.



- Cell initialization:

```
>> a = cell(3,2);
>> a = {'hello world', [1,5,7], rand(2,4)}
```

- To access a cell element, use curly braces {}

```
>> a = {'hello world', [1,5,7], rand(2,4)}
a = 'hello world'    [1x3 double]    [2x4 double]
>> a{1,1}
ans = hello world
>> a{1,3}
ans =
0.9058    0.9134    0.0975    0.5469
0.1270    0.6324    0.2785    0.9575
```

- ```
  T = cell(1,9);
  T(1:2) = { [1], [1 0] };
  for n=2:8
      T{n+1}=[2*T{n} 0] - [0 0 T{n-1}];
  end
  ```

  ```
  >> T
  T =
  Columns 1 through 5
  [1] [1x2 double] ... [1x5 double]
  Columns 6 through 9
  [1x6 double] [1x7 double] ... [1x9 double]
  ```
  ```

Structures

- Structures are essentially cell arrays that are indexed by a name rather than by number.
- The field values can be anything.
- Values are accessed using the dot notation.
- ```
  >> student.name = 'Moe';
  >> student.homework = [10 10 7 9 10];
  >> student.exam = [88 94];
  >> student
  student =
  name: 'Moe'
  homework: [10 10 7 9 10]
  exam: [88 94]
  ```
  ```

- Add another student:

```
>> student(2).name = 'Curly';
>> student(2).homework = [4 6 7 3 0];
>> student(2).exam = [53 66];
>> student
student =
1x2 struct array with fields:
homework
exam
```

- Array and field names alone create comma-separated lists of all the entries in the array.

```
>> roster = {student.name}
roster =
'Moe' 'Curly'
```

cell2mat – cell2struct

- cell2mat Convert cell array to ordinary array of the underlying data type

```
C = {[1],    [2 3 4];
[5; 9], [6 7 8; 10 11 12]}
C =
{[       1]}    {1x3 double}
{2x1 double}    {2x3 double}
A = cell2mat(C)
A =
1     2     3     4
5     6     7     8
9    10    11    12
```

- cell2struct Convert cell array to structure array

```
>> fields={'number','name','value'};
>> c={'one','Hamdullah',3;'two','Hamdi',7};
>> cStruct=cell2struct(c,fields,2)
cStruct =  2x1 struct array with fields:
number
name
value
```

End of Lecture

1. Scripts and Functions
2. Control Loops
3. Advanced Data Structures

Exercises I

- In order to get and save current date and time, write a script by following steps:
  - Create a variable **start** using the function **clock**
  - What is the size of **start**?
  - What does **start** contain? See help **clock**
  - Convert the vector **start** to a string. Use the function **datestr** and name the new variable **startString**
  - Save **start** and **startString** into a mat file named **startTime**

Exercises II

- If A is a square matrix (i.e. of dimension $n \times n$), the matrices $\cos(A)$ and $\sin(A)$ can defined by the formulas

$$\cos(A) = \sum_{k=0}^{\infty} (-1)^k \frac{A^{2k}}{2k!}, \qquad \sin(A) = \sum_{k=0}^{\infty} (-1)^k \frac{A^{2k+1}}{(2k+1)!},$$

respectively. The partial sums

$$C_N(A) = \sum_{k=0}^{N-1} (-1)^k \frac{A^{2k}}{2k!} \qquad S_N(A) = \sum_{k=0}^{N-1} (-1)^k \frac{A^{2k+1}}{(2k+1)!}$$

can thus be used to approximate the matrices $\cos(A)$ and $\sin(A)$.

  - Write a function whose inputs are a square matrix A and a tolerance number (**TOL**), and whose ouputs are the matrices $\cos(A)$ and $\sin(A)$. The outputs should be obtained by using Matlab to compute the sequences $C_N(A)$, and $S_N(A)$, $N = 1, 2, \ldots$ and stopping when the maximum of the absolute values of the entries of the matrix $C_{N+1}(A) - C_N(A)$ and $S_{N+1}(A) - S_N(A)$ is less than **TOL**. (Note that $\cos(A)$ and $\sin(A)$ is NOT the matrix obtained by computing the cosine of the individual entries of the matrix) (Hint: Use the **while** loop as well as the command **max**.)
  - Let

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

where a_{11}, a_{12}, $a_{21} a_{22}$ are the last 4 digits of your student number. Use the above function to compute $\cos(A)$ and $\sin(A)$. Save your answers in the variables Answer1 and Answer2, respectively. Use Matlab to compute the matrix $(\cos(A))^2 + (\sin(A))^2$. Save your answer in the variables Answer3.

Exercises III

- Write a function whose input is a positive integer and whose outputs a matrix and a vector such that $A = (a_{ij})$, where $a_{ij} = i/j$ and $x_j = j$, respectively. Display a warning message if n is nonpositive by using **fprintf** command.

Exercises IV

- Write a function to compute the factorial value of a single scalar argument. This function should have the following components:
  - An **if** statement which returns an error message if the argument is negative by using **disp** command.
  - An **elseif** statement which returns an error message if the argument is not an integer. You should use either the built-in **round**, **floor** or **ceil** functions to test for non-integers.
  - An **else** statement with an embedded **for** loop that does the actual factorial calculation. Make sure that your function is able to handle any non-negative integer, including 0.

For More Information

- http://iam.metu.edu.tr/scientific-computing
- https://iam.metu.edu.tr/scientific-computing-lecture-series
- https://www.facebook.com/SCiamMETU/
- https://www.instagram.com/scmetu/

...thank you for your attention !