PHYSICS 210

OVERVIEW OF MATLAB PROGRAMMING

PRELIMINARIES

- Principal unit of Matlab usage: statement

```
>> a = 2

>> vr = [1 2 3 sqrt(17)]

>> vc = [5; 6; cos(pi/12); exp(2.3)]

>> M = [cos(pi) sin(pi); -sin(pi) cos(pi)]

>> linspace(0.0, 100.0, 101)

>> diag(M)
```

PRELIMINARIES

- Principal modes of Matlab programming
  - Matlab scripts (programs)
    - Arbitrary sequence of Matlab statements, including assignments, control structures, input/output statements, etc.
  - Matlab functions
    - Completely analogous to Maple procedures
- Programming in Matlab $\leftrightarrow$ Writing Matlab scripts and functions
- Whereas in Maple we focused on procedures (functions), in Matlab we will also use scripts extensively, especially for term projects
- As we saw in the lab, Matlab source code (scripts/functions) must *always* be prepared in a text files with a **`.m`** extension

DEFINING MATLAB FUNCTIONS

- Recall meta-syntax
  - Meta-values: to be replaced by specific instance of <thing>, e.g.
    - `<Bexpr>` Boolean expression `a > b`
    - `<ss>` statement sequence `x = 3,` `y = exp(2.3)`
  - Reserved words & operators: parts of language syntax, must be typed verbatim, e.g.
    - `function`
    - `if`
    - `then`
    - `else`
    - `for`
    - `end`
    - `[`
    - `:`

FUNCTION DEFINITION: SYNTAX & GENERAL FORMS

- Note: A Matlab function can return 0, 1, 2, … values (as many as you wish), and each value can be a scalar, vector, array …

- Meta notation:

  - `<ss>` arbitrary sequence of Matlab statements (commands)

    - In function definitions (as well as in scripts) will generally want to end each statement with a semi-colon to suppress output, but can omit semi-colons for an easy and useful way to “trace” execution of statements when developing/debugging

  - `<fcnname>` valid Matlab name
  - `<inarg>` input argument (formal argument)
  - `<outarg>` output argument” (a.k.a. “return value”)

FUNCTION RETURNING 0 VALUES

- General:

```
function <fcnname>(<inarg1>, <inarg2>, … <inargm>)
     <ss>
end
```

  - **end** is optional, but I will always use it, recommend that you do as well
  - Will refer to **function** line as “header”, **<ss>** as “body”

- Example: 1 inargs, 0 outarg

```
function zero_outarg(x)
     fprintf('The input argument is %g', x);
end

>> zero_outarg(2013)

The input argument is 2013
```

NOTE: Mapping of formal input arg → actual arg: **x** → **2013**

FUNCTION RETURNING 0 VALUES

```
function zero_outarg(x)
    fprintf('The input argument is %g', x);
end

>> zero_outarg(2013)

The input argument is 2013
```

- Definition of function *must* be made in a file with name

  `<fcnname>.m`

- For specific case considered above, this is (literally)

  `zero_outarg.m`

- Define only one function per text file, and name that text file `<fcnname>.m`

FUNCTION RETURNING 1 VALUE

- General:

```
function <outarg> = <fcnname>(<inarg1>, <inarg2>, … <inargm>)
     <ss>
end
```

- Example: 2 inargs, 1 outarg (defined in text file `one_outarg.m`)

```
function out1 = one_outarg(in1, in2)
     % CRUCIAL! A value MUST be assigned to 'out1' within the
     % body of the function

     out1 = in1 + in2;
end

>> val = one_outarg(3, 4)

val = 7
```

- NOTE: Mapping between formal and actual args: in1→3, in2 → 4

FUNCTION RETURNING 2 VALUES

- General: Output is a length-2 vector whose elements are the 2 outargs

```
function [<outarg1> <outarg2>] = <fcnname>(<inarg1>, <inarg2> … )
    <ss>
end
```

- Note the syntax: square brackets enclose the `<outargs>`

- Example: 4 inargs, 2 outargs (defined in text file `two_outarg.m`)

```
function [out1 out2] = two_outarg(in1, in2, in3, in4)
    % CRUCIAL! A value MUST be assigned to BOTH 'out1' and 'out2'
    % within the body of the function.

    out1 = in1 + in2;
    out2 = in3 - in4;
end
```

- More syntax: Commas between the `<outargs>` not needed (optional, won't hurt if you include them) but are *absolutely required* between the `<inargs>`

FUNCTION RETURNING 2 VALUES

```
function [out1 out2] = two_outarg(in1, in2, in3, in4)
    % CRUCIAL! A value MUST be assigned to BOTH 'out1' and 'out2'
    % within the body of the function.

    out1 = in1 + in2;
    out2 = in3 - in4;
end

>> [val1 val2] = two_outarg(7, 8, 9, 10)

val1 = 15
val2 = -1
```

- Note the syntax for the *assignment* of the return values, vector of variables must appear on the left hand side to “capture” both values that are returned

FUNCTION RETURNING 3 VALUES

- General: Output is a length-3 vector whose elements are the 3 outargs

```
function [<outarg1> <outarg2> <outarg3>] = <fcnname>(<inarg1> … )
    <ss>
end
```

- Again note the syntax: square brackets enclose the `<outargs>`

- Example: 3 inargs, 3 outargs (defined in text file `three_outarg.m`)

```
function [out1 out2 out3] = three_outarg(in1, in2, in3)
    % Values MUST be assigned to all three of 'out1',
    % 'out2' and 'out3' in the body of the function.
    %
    % Also note that the 2nd and 3rd output arguments are
    % assigned a vector and a matrix respectively.

    out1 = in1;
    out2 = zeros(1, in2);
    out3 = eye(in3);
end
```

FUNCTION RETURNING 3 VALUES

```
function [out1 out2 out3] = three_outarg(in1, in2, in3)
     % Values MUST be assigned to all three of 'out1',
     % 'out2' and 'out3' in the body of the function.
     %
     % Also note that the 2nd and 3rd output arguments are
     % assigned a vector and a matrix respectively.

     out1 = in1;
     out2 = zeros(1, in2);
     out3 = eye(in3);
end

>> [val1 val2 val3] = three_outarg(100, 3, 2)

val1 = 100
val2 = 0 0 0
val3 =
1   0
0   1
```

- Once more, note the vector of variables on the left hand side that is needed to ensure that all three return values are "captured"

BOOLEAN OPERATIONS

- No distinct Boolean type in Matlab (as there was in Maple)
  - Numerical value 1 is defined to be “true”
  - Numerical value 0 is defined to be “false”

  - (In actuality any non-zero value is true)

Relational Operators	
==	Equal
~=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Logical Operators	
&	Logical AND
|	Logical OR
~	Logical NOT

CONTROL STRUCTURES (SELECTION): `if-else-end` STATEMENT

- General: `if-else-end`

```
if <Bexpr>
    <ss 1>
else
    <ss 2>
end
```

- Note: no `then`; use `end` rather than `end if`

- Example

```
if a > b
    c = a + b;
else
    c = a - b;
end
```

CONTROL STRUCTURES: `if-end` STATEMENT

- Special case: no **`else`** clause

```
if <Bexpr>
     <ss>
end
```

- Example:

```
if a > b
     c = a + b;
end
```

CONTROL STRUCTURES: `if-elseif-else-end` STATEMENT

- General: **`if-elseif-else-end`**

```
if <Bexpr 1>
     <ss 1>
elseif <Bexpr 2>
     <ss 2 >
elseif <Bexpr 3>
     <ss 3>
     .
     .
     .
else
     <ss n>
end
```

- Example

```
if x == 0
     y = 1;
elseif x == 1
     y = 2;
elseif x == 2
     y = 4;
else
     y = 0;
end
```

- Note: **`elseif`** not **`elif`** as in Maple

CONTROL STRUCTURES (ITERATION): `for-end` STATEMENT

- General:

```
for <loopvar> = <vector-expression>
     <ss>
end
```

`<vector-expression>` MUST define row vector

- General type 1: `<vector-expression>` created using colon operator

```
for <loopvar> = <first> : <last>
     <ss>
end

for <loopvar> = <first> : <step> : <last>
     <ss>
end
```

- `<first>`, `<last>`, `<step>` don’t need to have integer values, but often will in our work

CONTROL STRUCTURES: **`for-end`** STATEMENT

- Type 1 examples

```
for k = 3 : 6
     k
end

k = 3
k = 4
k = 5
k = 6

for jj = 2 : 3 : 14
     2 * jj
end

jj = 4
jj = 10
jj = 16
jj = 22
jj = 28
```

```
for value = 5 : -6 : -25
       value
end

value = 5
value = -1
value = -7
value = -13
value = -19
value = -25
```

CONTROL STRUCTURES: `for-end` STATEMENT

- General:

```
for <loopvar> = <vector-expression>
     <ss>
end
```

- General type 2: `<vector-expression>` created using any other command or expression that returns/defines a row vector

- Example:

```
for val = [ 1, 3, 9, sqrt(2) ]
     val;
do

val = 1
val = 3
val = 9
val = 1.414
```