SECTION 1: INTRODUCTION

ENGR 112 – Introduction to Engineering Computing

Course Overview

What is Programming?

- **Programming**
  - The implementation of ***algorithms*** in a particular computer ***programming language*** for execution on a ***computer***

- **Algorithm**
  - A step-by-step procedure for performing a computation, solving a problem, performing some action, etc. – a recipe
  - ***Algorithm design*** is the meat of programming – the rest is just translation into a particular language
- **Programming language**
  - We'll use MATLAB. Others include C, C++, Python, Fortran, etc.
- **Computer**
  - May be a PC, or may be a microcontroller, FPGA, etc.

Why Programming?

- I don't want to be a *software* engineer. Why do I need to learn to program?
  - ***All*** engineers will need to write computer code throughout their careers
    - Design and simulation
    - Numerical solution of mathematical problems
    - Data analysis – from measurements or simulation
    - Firmware for the control of mechatronic systems
  - More importantly: ***development of algorithmic thinking ability***
    - Learn to think like an engineer – single most important takeaway from your engineering education

Course Overview

- Section 1: Introduction
- Section 2: Vectors and Matrices
- Section 3: Two-Dimensional Plotting
- Section 4: Algorithmic Thinking & Flow Charts
- Section 5: Structured Programming in MATLAB
- Section 6: User-Defined Functions
- Section 7: Three-Dimensional Plotting
- Section 8: File I/O
- Section 9: Engineering Applications

Introductory material:
- Course overview
- Introduction to required tools
- Linear algebra basics

Platform- (MATLAB) specific material:
- A valuable engineering tool – learn to use it effectively

Algorithm fundamentals:
- Generic; Platform-independent
- Engineering thinking – transcends programming

Application of the fundamentals:
- MATLAB-specific, but
- Similar to other languages

MATLAB

- This a course in ***programming fundamentals*** and ***algorithmic thinking***

- The tool we'll use to develop these concepts is ***MATLAB***
  - Could just as well use another language, e.g., Python, C, C++, Java, Fortran, …
  - The important concepts are not language-specific

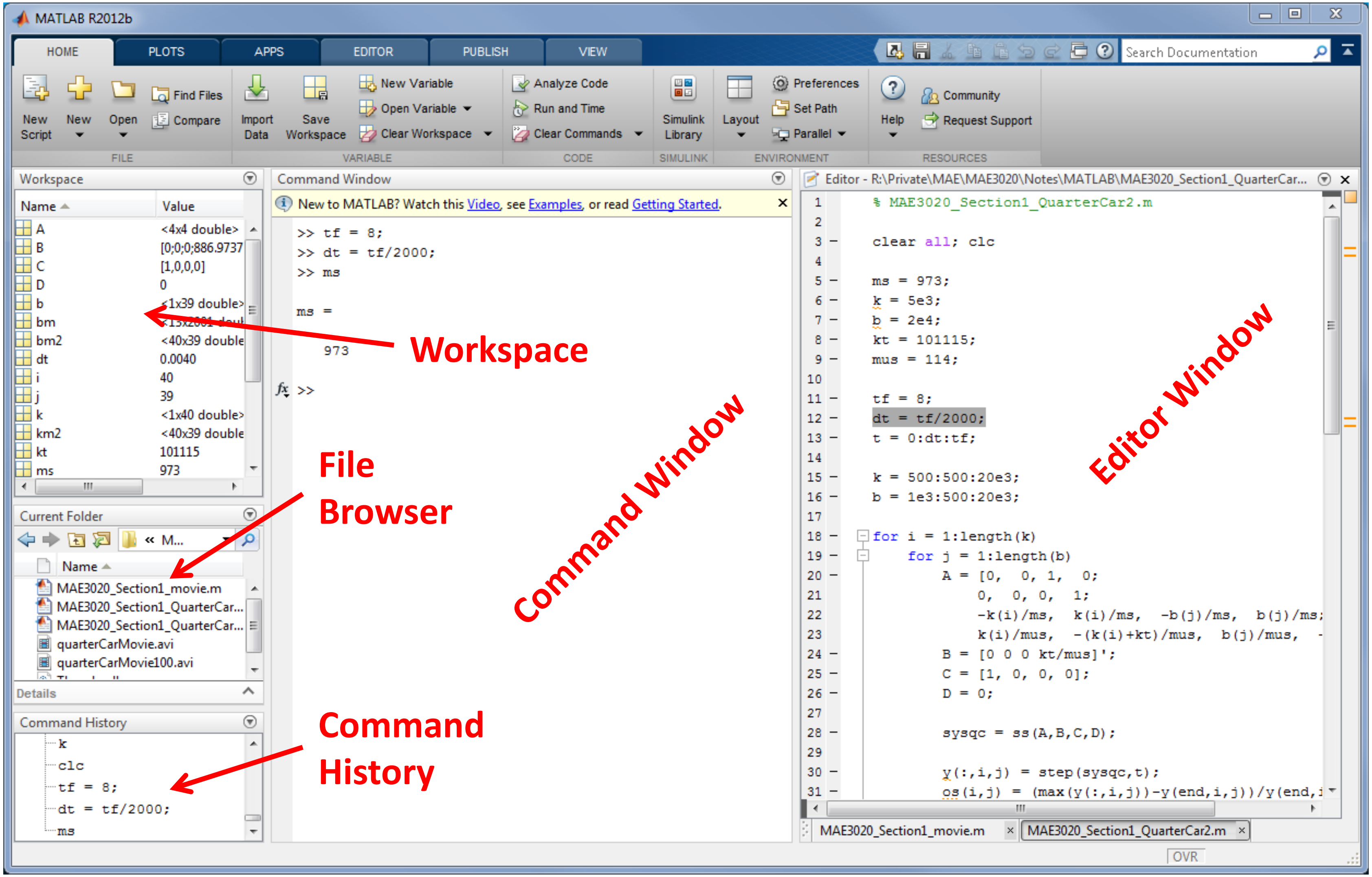
- ***Two goals*** of this course:
  - Learn to develop basic algorithms and to write structured computer code
  - Learn to use MATLAB

Introduction to MATLAB

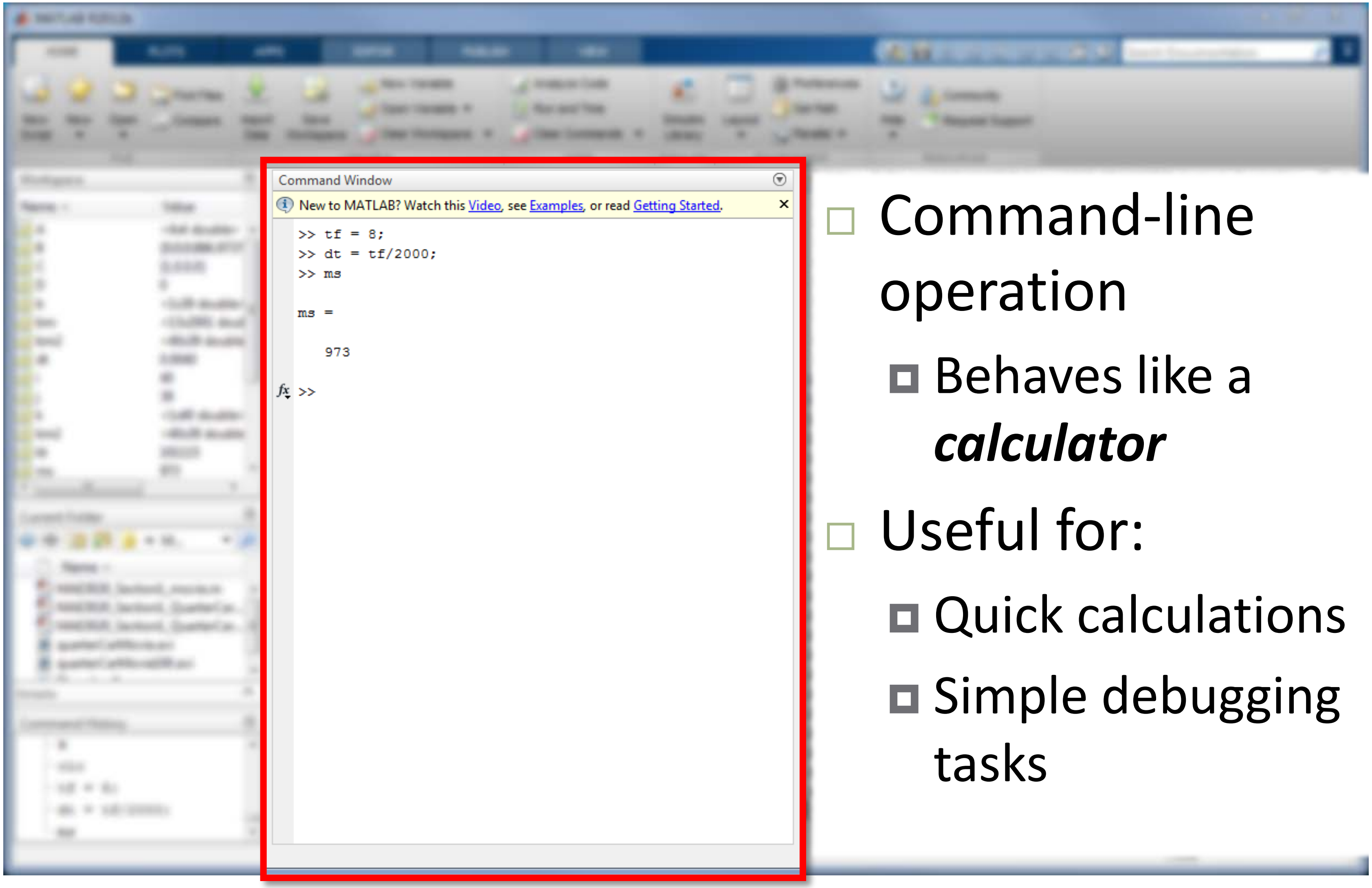
The remainder of this section of notes is intended to provide a brief introduction to MATLAB.

This is not intended to be a thorough tutorial on the use of the tool, but the beginning of a process that will continue throughout the course.

The MATLAB Desktop

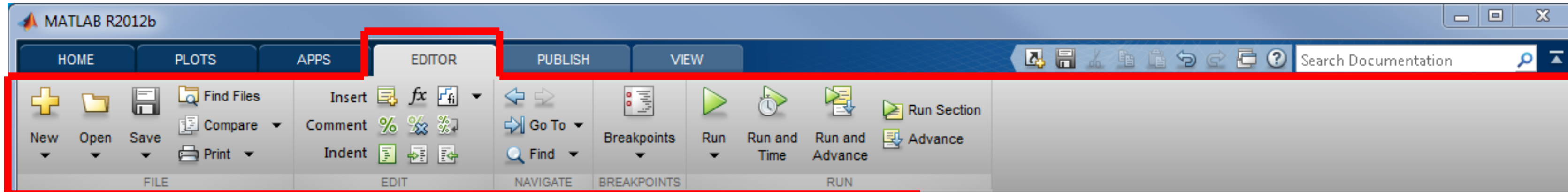


The MATLAB Desktop – Command Window

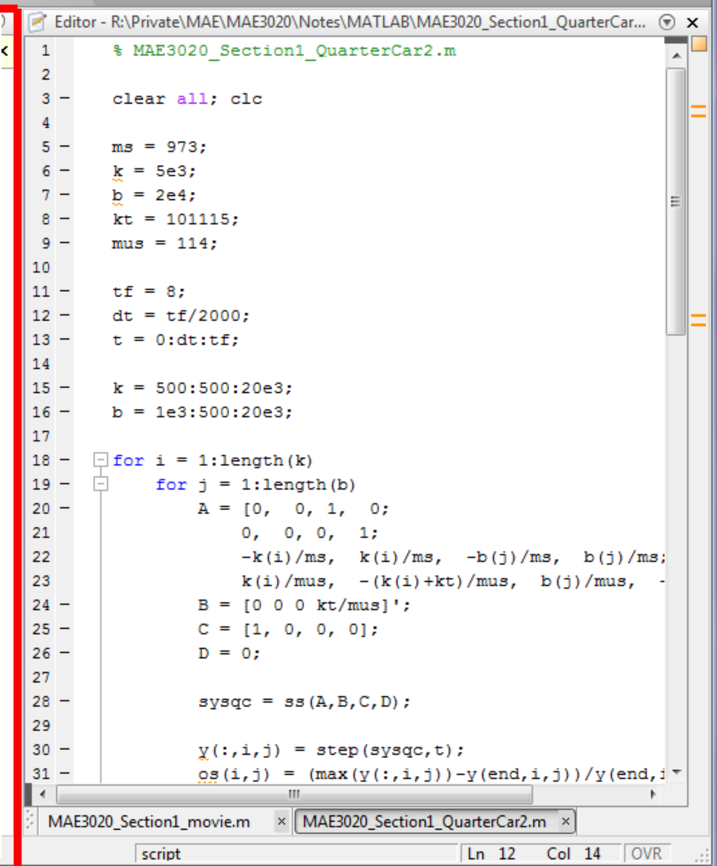


- Command-line operation
  - Behaves like a ***calculator***
- Useful for:
  - Quick calculations
  - Simple debugging tasks

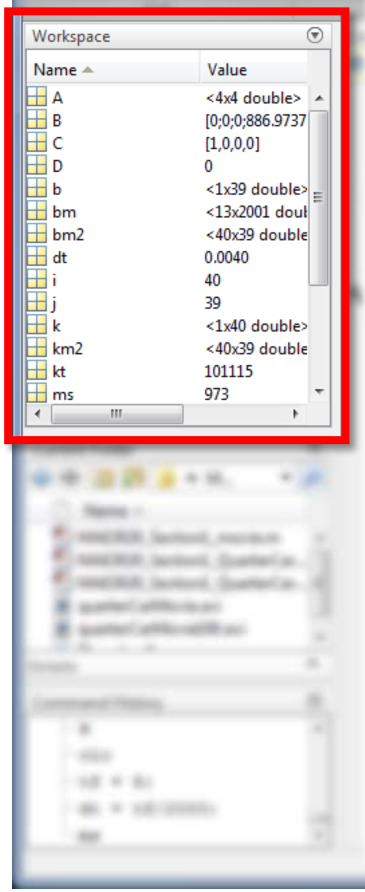
The MATLAB Desktop – Editor Window



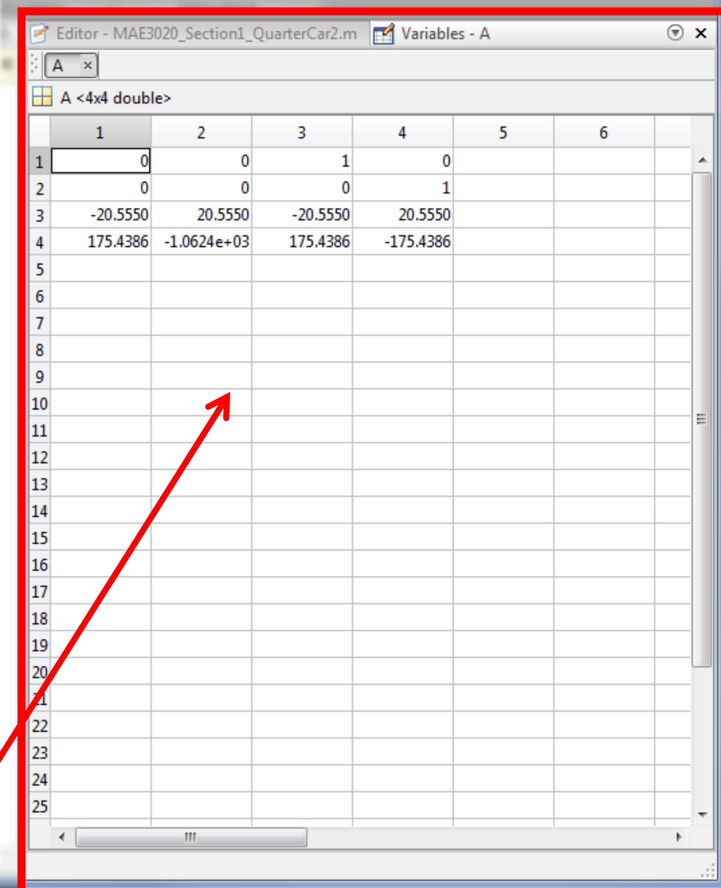
- Editor for m-files
  - Scripts
    - Collections of commands executed sequentially
  - Functions
- Built-in debugger
  - Set breakpoints
  - Step through code line-by-line or by section



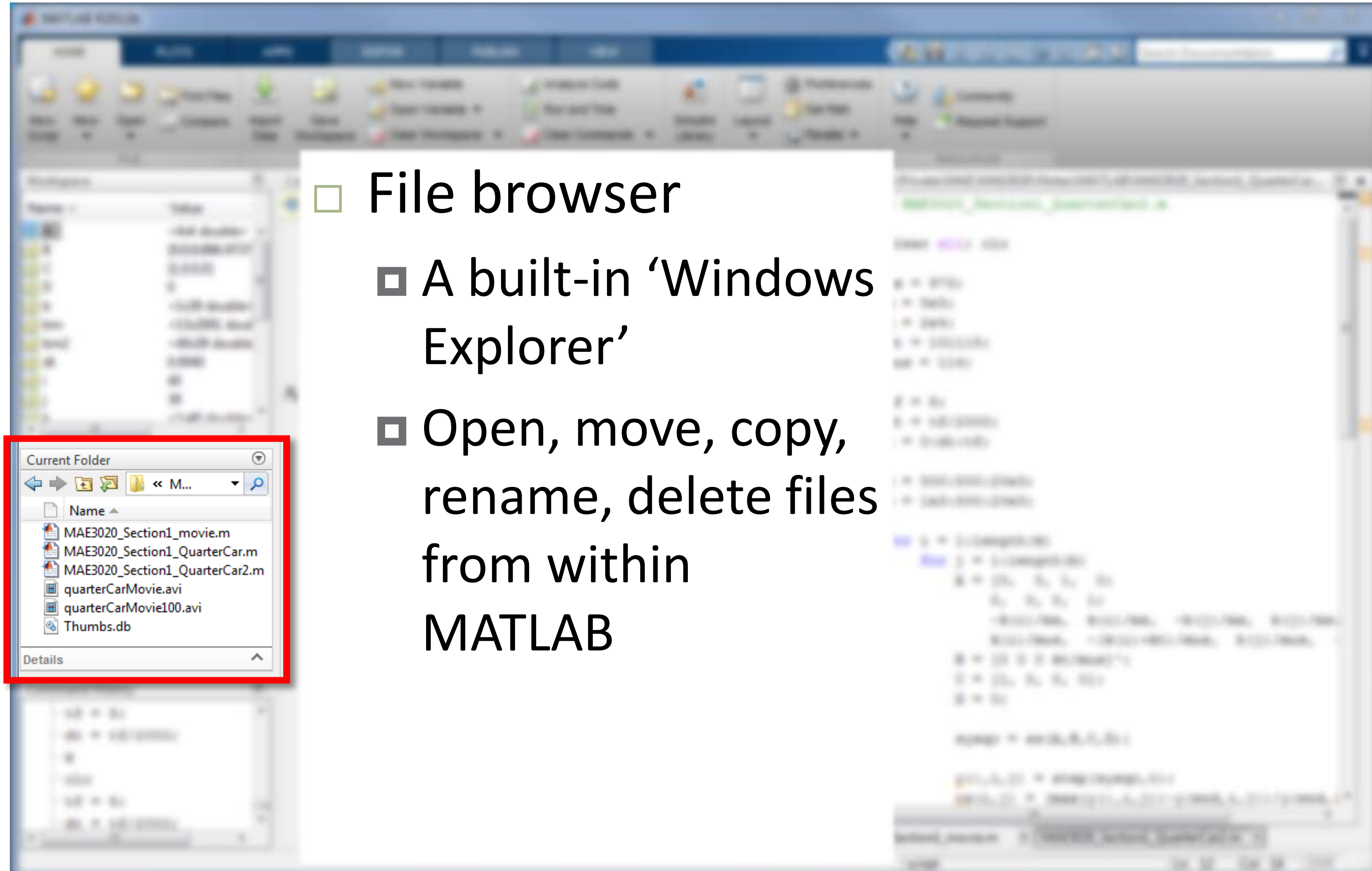
The MATLAB Desktop – Workspace



- Lists all variables currently stored in memory
  - Values for scalars and small arrays
  - Size and data types for larger arrays
- Double-click a variable to open in a spreadsheet

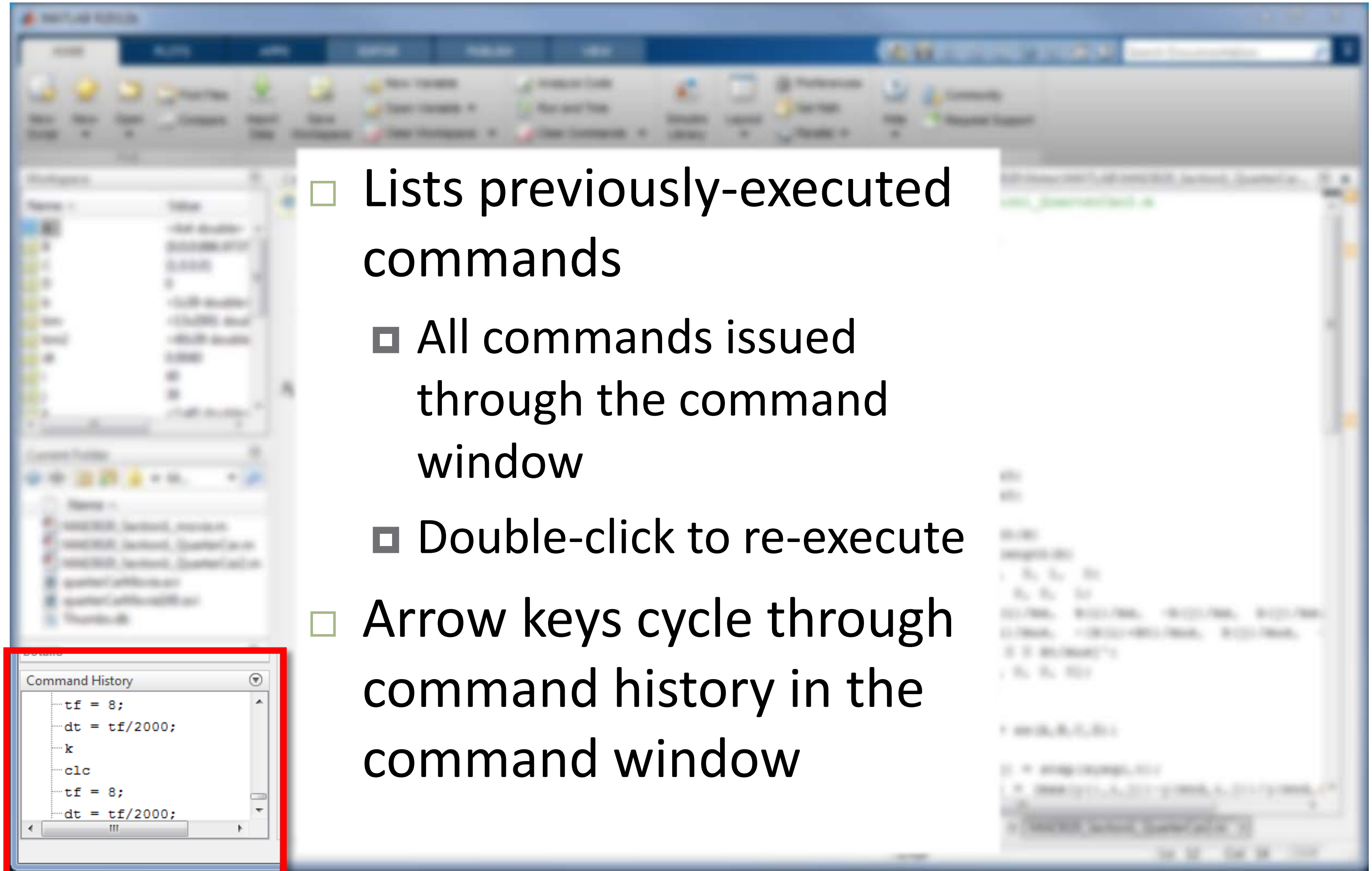


The MATLAB Desktop – Current Folder



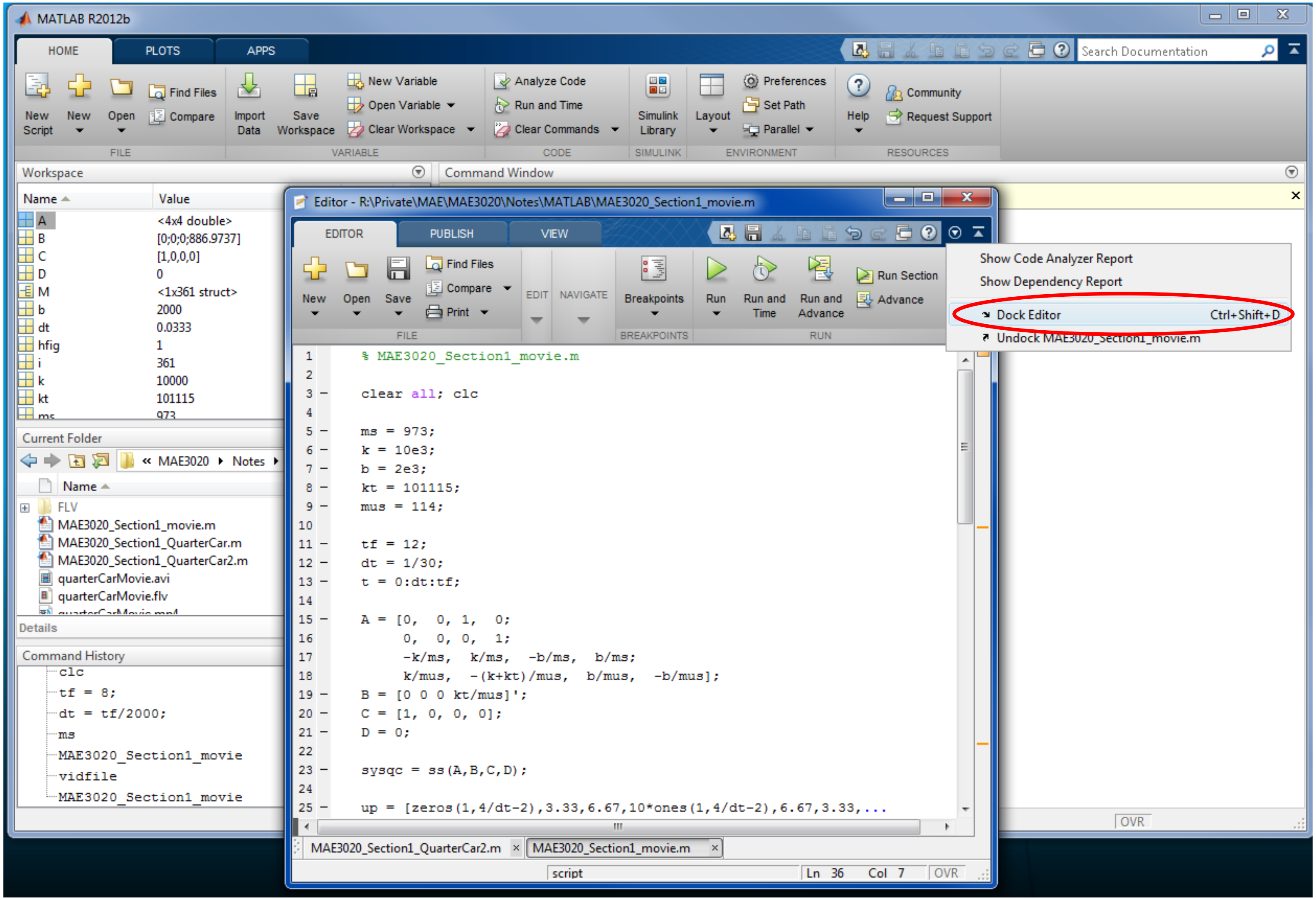
- File browser
  - A built-in ‘Windows Explorer’
  - Open, move, copy, rename, delete files from within MATLAB

The MATLAB Desktop – Command History

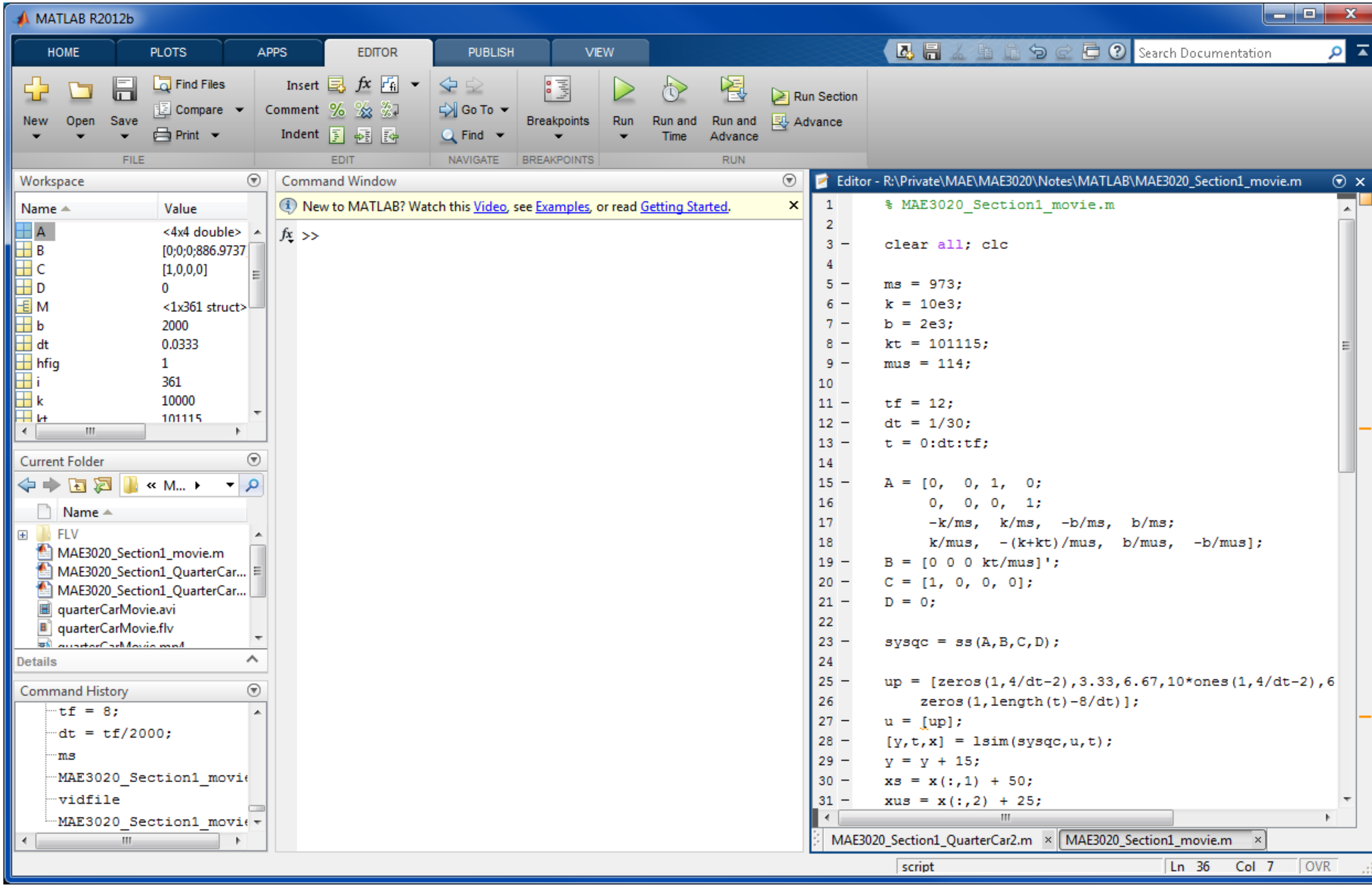


- Lists previously-executed commands
  - All commands issued through the command window
  - Double-click to re-execute
- Arrow keys cycle through command history in the command window

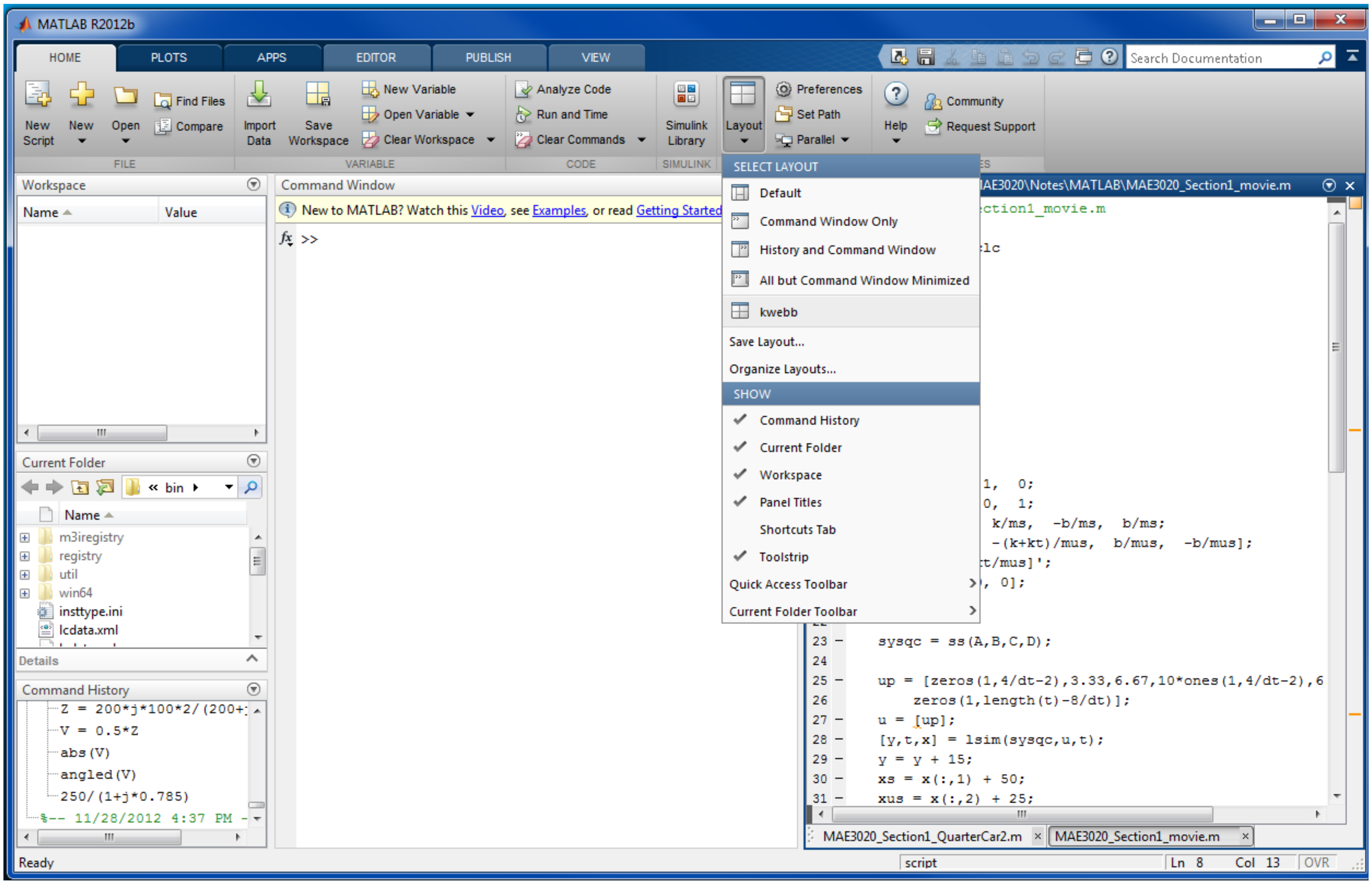
The MATLAB Desktop – Docking Windows



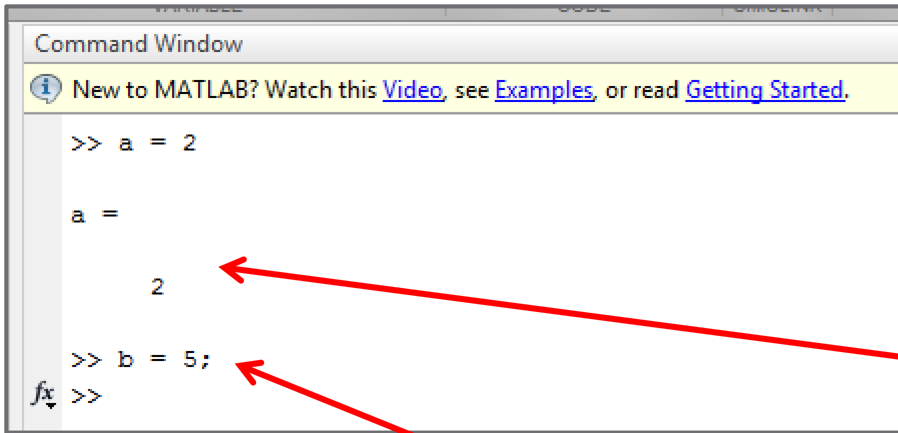
The MATLAB Desktop – Docking Windows

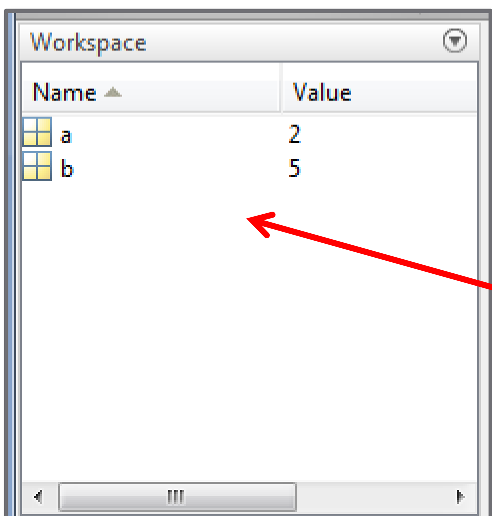


The MATLAB Desktop – Saving Layouts



Assignment of Variables





- Can define variables and assign values
- Variable and value echoed in command window
- Terminating command with semicolon suppresses echo
- Variables then appear in workspace

Data Types

- Variables used in MATLAB can be of many different types, e.g. integers, floating-point numbers, alphanumeric characters, etc.
- The following section introduces each of these data types. You'll gain a better understanding of each as the course progresses.

Variable Declaration

- In MATLAB, it isn't necessary to declare a variable before using it, e.g.:

```
a = 7.4039;
```

- Declaration occurs automatically upon assignment
  - Default type is `double`

- This differs from many other languages, e.g. in C:

```
float a;
a = 7.4039;
```

or

```
float a = 7.4039;
```

Variable Names

- Variable names must ***start with a letter***
- Names may contain ***letters***, ***numbers***, and ***underscore*** characters
  - ***No spaces***
- Names are ***case sensitive***
- Don't name variables with names of ***built-in functions***
  - Can be done, but that function will not be available as long as the variable is defined in the workspace

Fundamental MATLAB Data Types

- MATLAB supports many different numeric and non-numeric ***data types***
- **Numeric types**
  - int8, int16, int32, int64
  - uint8, uint16, uint32, uint64
  - single
  - double
- **Non-numeric types**
  - logical
  - char
  - table
  - cell
  - struct
  - function handle

Data Types – `double`

- When you assign a variable a numeric value, e.g.

```
a = 7.4039;
```

  by default, its type is `double`

- `double`
  - Numeric value stored using double-precision floating-point format
  - 64 bits used to store each variable value
  - Accurate representation of very large and very small values
    - Range: $\pm 2.22507 \times 10^{-308} \ldots \pm 1.79769 \times 10^{308}$
  - Can *usually* ignore numerical errors due to inaccurate numeric representation

Non-Default Data Types

- It is possible to force MATLAB to store numeric variable values as other types
  - We'll rarely, if ever, do this here
  - May become important if writing code for execution on non-PC target hardware,
    - E.g., microcontroller for control system application
  - Default, double, type requires 64 bits
    - May consume excessive memory
    - Mathematical operations may be too slow on some hardware
  - Other data types trade off precision for memory

Data Types – `single`

- `single`
  - Single-precision floating-point format
  - 32 bits
  - Less memory required than for `double`
  - Less precise than `double`
  - Range: $\pm 1.17549 \times 10^{-38} \ldots \pm 3.40282 \times 10^{38}$

Data Types – `int8`, `int16`, `int32`, `int64`

- ***Signed integers***
  - One sign bit – remainder for integer value

- `int8`
  - 8-bit
  - Min: -128
  - Max: 127
- `int16`
  - 16-bit
  - Min: -32768
  - Max: 32767
- `int32`
  - 32-bit
  - Min: -2147483648
  - Max: 2147483647
- `int64`
  - 64-bit
  - Min: -9223372036854775808
  - Max: 9223372036854775807

Data Types – `uint8`, `uint16`, `uint32`, `uint64`

- ***Unsigned integers***
  - All bits used to store integer value

- `uint8`
  - 8-bit
  - Min: 0
  - Max: 255
- `uint16`
  - 16-bit
  - Min: 0
  - Max: 65535
- `uint32`
  - 32-bit
  - Min: 0
  - Max: 4294967295
- `uint64`
  - 64-bit
  - Min: 0
  - Max: 18446744073709551615

Non-numeric Types – `logical`

- Data type that stores one of only two values:
  - True – stored as a 1
  - False – stored as a 0
- For example, relational (comparison) operations return `logical` values:

Workspace

Name	Class	Value	Min
a	double	6	6
b	logical	1	

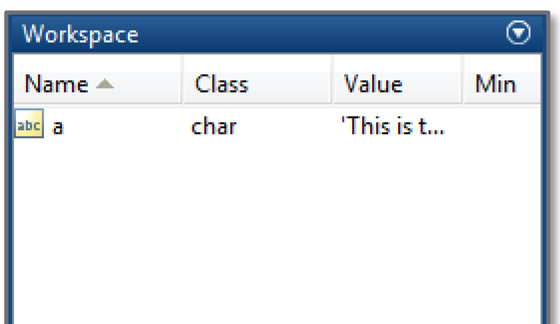
Command Window

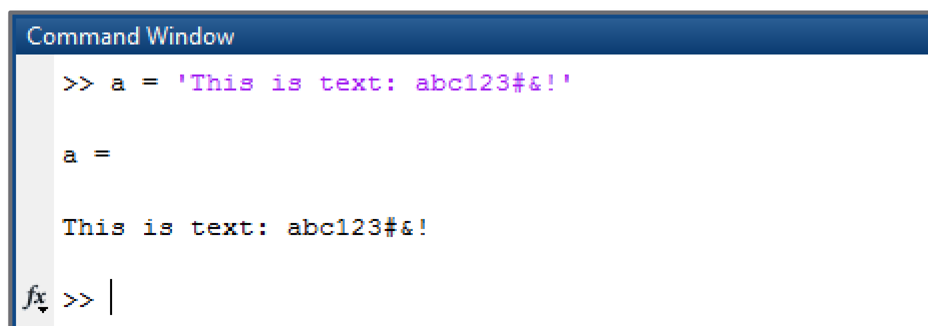
```
>> a = 6;
>> b = a < 10

b =

     1
```

- Relational operation evaluates as true
  - b stored as a `logical` with value 1

Non-numeric Types – char

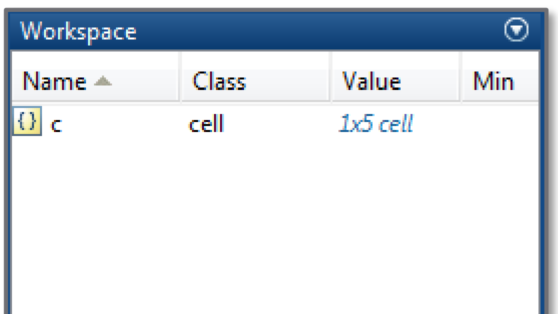
- Data type used for ***text***
  - Alphanumeric characters
- Useful for:
  - Reading from and writing to files
  - Annotating plots, etc.
- Variables are of the `char` type when their assigned value is enclosed in ***single quotes***:

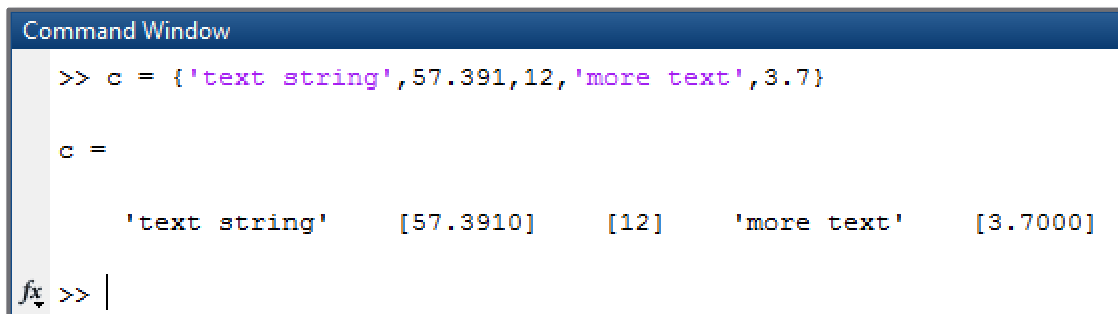




Non-numeric Types – `cell`

- A ***cell array*** is a collection of individual data containers called ***cells***
- Each cell in an array may be of a different size or different type
- For example, `c` is an array of `char` and `double` cells:





- `c` is of type `cell`

Non-numeric Types – `table`

- Container to hold column-oriented data of different types and sizes
  - Useful for spreadsheet-like data
  - Each column contains a different variable
- May also include metadata properties, e.g.:
  - Variable units
  - Row names, etc.

Command Window

```
>> day = {'Monday';'Tuesday';'Wednesday';'Thursday';'Friday'};
>> Thigh = [63;71;68;59;74];
>> Tlow = [48;51;50;47;49];
>> sky = {'Partly Cloudy';'Clear';'Partly Cloudy';'Cloudy';'Clear'};
>> precip = [0.01;0;0.04;0.89;0];
>> T = table(day,Thigh,Tlow,sky,precip)

T = 

        day          Thigh    Tlow          sky            precip
    ___________      _____    ____    _______________     ______

    'Monday'         63       48      'Partly Cloudy'      0.01
    'Tuesday'        71       51      'Clear'                 0
    'Wednesday'      68       50      'Partly Cloudy'      0.04
    'Thursday'       59       47      'Cloudy'             0.89
    'Friday'         74       49      'Clear'                 0

fx >>
```

Workspace

Name	Class	Value	Min
T	table	5x5 table	
Thigh	double	[63;71;68...	59
Tlow	double	[48;51;50...	47
day	cell	5x1 cell	
precip	double	[0.0100;0...	0
sky	cell	5x1 cell	

Non-numeric Types – `struct`

- ***Structures*** are arrays with named ***fields***, each containing data of varied type and size
- Field data can be accessed individually

Command Window

```
>> Weather.day = {'Monday';'Tuesday';'Wednesday';'Thursday';'Friday'};
>> Weather.Thigh = [63;71;68;59;74];
>> Weather.Tlow = [48;51;50;47;49];
>> Weather.sky = {'Partly Cloudy';'Clear';'Partly Cloudy';'Cloudy';'Clear'};
>> Weather.precip = [0.01;0;0.04;0.89;0];
>> Weather

Weather = 

       day: {5x1 cell}
     Thigh: [5x1 double]
      Tlow: [5x1 double]
       sky: {5x1 cell}
    precip: [5x1 double]

>> Weather.sky

ans = 

    'Partly Cloudy'
    'Clear'
    'Partly Cloudy'
    'Cloudy'
    'Clear'

fx >> 
```

Workspace

Name	Class	Value	Min
Weather	struct	*1x1 struct*	
ans	cell	*5x1 cell*	

Mathematical Operations and Built-In Functions

MATLAB includes an extensive library of general-purpose, as well as many application-specific, built-in functions.

Basic Mathematical Operations

- MATLAB includes all of the basic mathematical functions you would expect in a scientific calculator
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Exponentiation

Command Window

```
>> x = 2;
>> y = 3;
>> z = x + y

z =

     5

>> a = x - y

a =

    -1

>> b = x*y

b =

     6

>> c = x/y

c =

    0.6667

>> d = x^y

d =

     8
```

Order of Operations

- MATLAB order of operations:
  1) ( ) parentheses
  2) ^ exponentiation
  3) - negation
  4) *, /, \ multiplication, division
  5) +, - addition, subtraction
- Expressions are evaluated left to right within each level of the precedence hierarchy

Command Window

```
>> 2 + 9 - 3^2/10*4

ans =

    7.4000

>> 2 + 9 - 3^(2/10*4)

ans =

    8.5918

>> 2 + (9 - 3)^2/10*4

ans =

   16.4000

>>
```

Built-In Functions

- MATLAB includes many built-in mathematical functions, including:
  - Square root: $\sqrt{x}$
  - Exponential: e^x
  - Factorial: $x!$
  - Absolute value: $|x|$
  - And many, many more…

Command Window

```
>> x = 2;
>> y = sqrt(x)

y =

    1.4142

>> z = exp(x)

z =

    7.3891

>> q = factorial(x^3)

q =

       40320

>> p = abs(y-x)

p =

    0.5858

>>
```

Trigonometric Functions

- ***Trigonometric functions*** expect input arguments expressed in ***radians***
- Inverse trig functions return values in radians
- To operate in ***degrees***
  - Convert
  - Use degree-specific functions: `sind`, `cosd`, `atand`, etc.

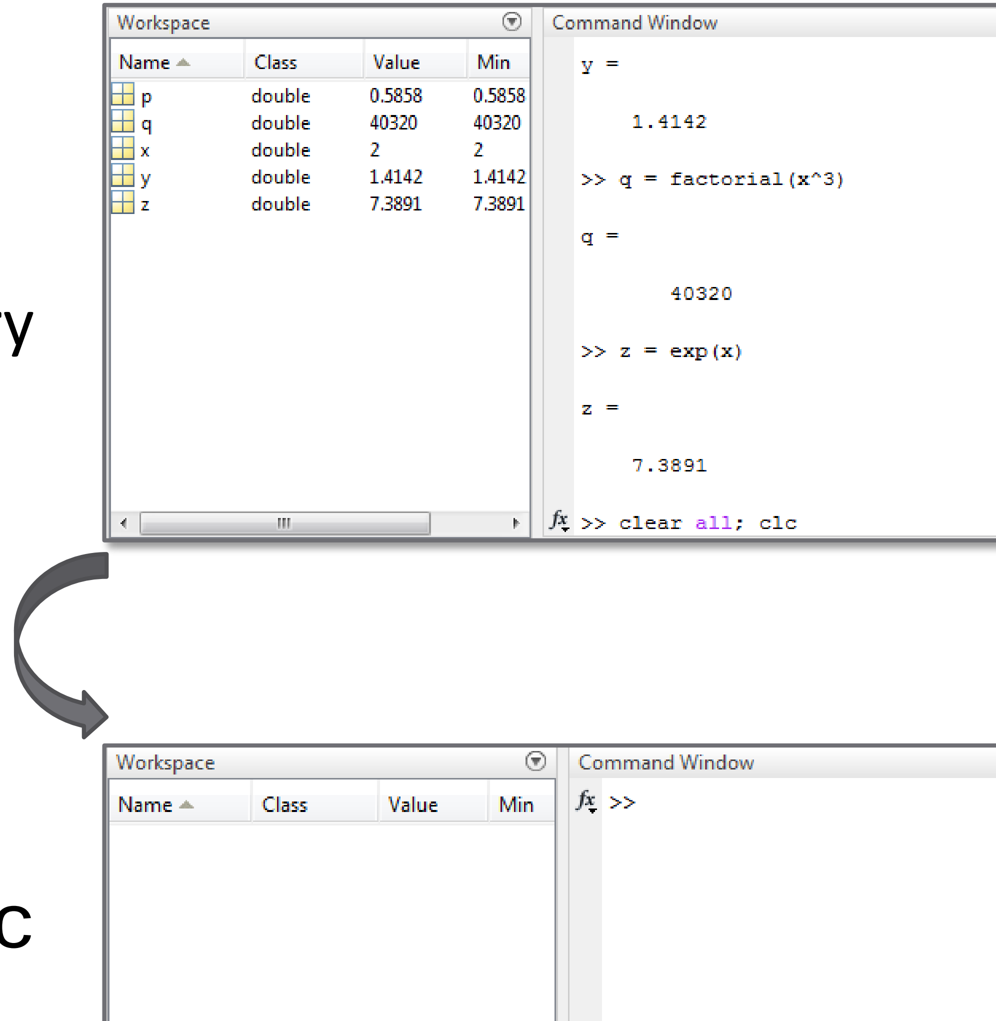
Command Window

```
>> sin(90)

ans =

    0.8940

>> sin(90*pi/180)

ans =

     1

>> sind(90)

ans =

     1

>> atan(1)

ans =

    0.7854

>> atand(1)

ans =

    45
```

Built-In Functions – `clear`, `clc`

- `clear x y`… or `clear all`
  - Deletes some or all variables from memory
- `clc`
  - Clears the command window
- Good practice to start all scripts with:

  `clear all; clc`



Logarithms

- Natural logarithm

  `y = log(x)`

- Log base 10

  `y = log10(x)`

- Log base 2

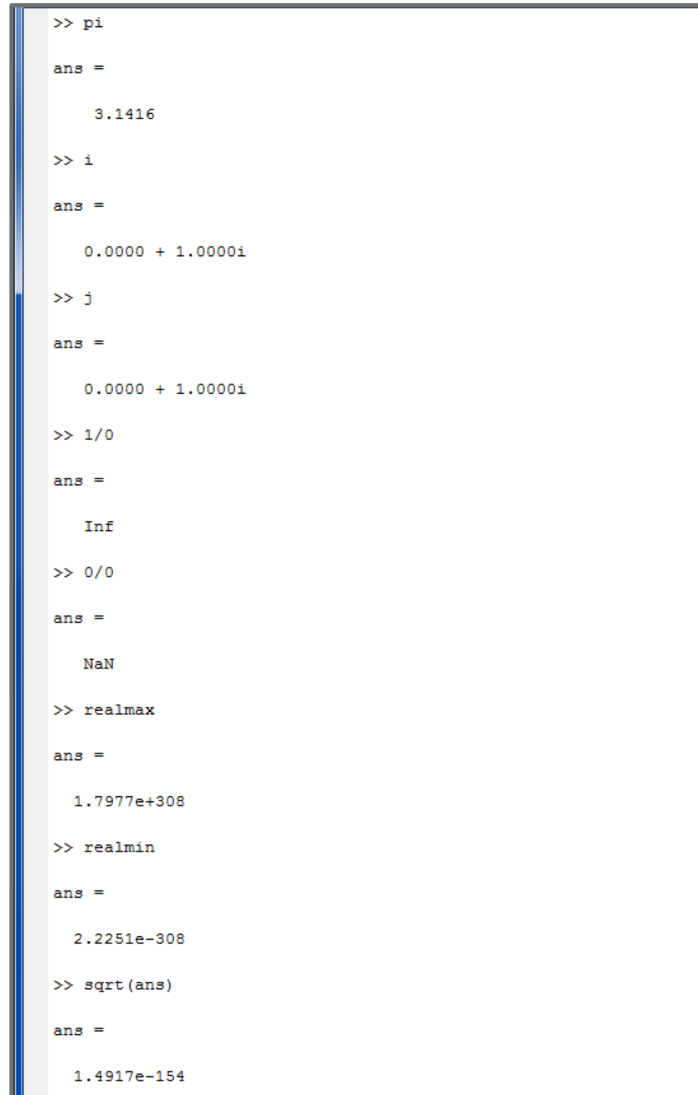
  `y = log2(x)`

Command Window

```
>> y = log(0.3679)

y =

   -0.9999

>> y = log10(10e6)

y =

     7

>> y = log2(4096)

y =

    12

>>
```

Built-In Constants

- Some built-in MATLAB constants:
  - π: `pi`
  - Imaginary unit ($\sqrt{-1}$): `i` or `j`
  - Infinity (∞): `inf`
  - Not-a-number: `NaN`
  - Result of most recently executed command: `ans`
  - Largest positive floating-point number: `realmax`
  - Smallest positive floating-point number: `realmin`



Scientific Notation

- Use ***scientific notation*** to represent very large or small numbers, e.g.:

$$1.58 \times 10^{-9}$$

- Very bad practice to type a lot of zeros – ***never do this***:

$$0.00000000158$$

  - Difficult to read, and much too easy to miscount zeros

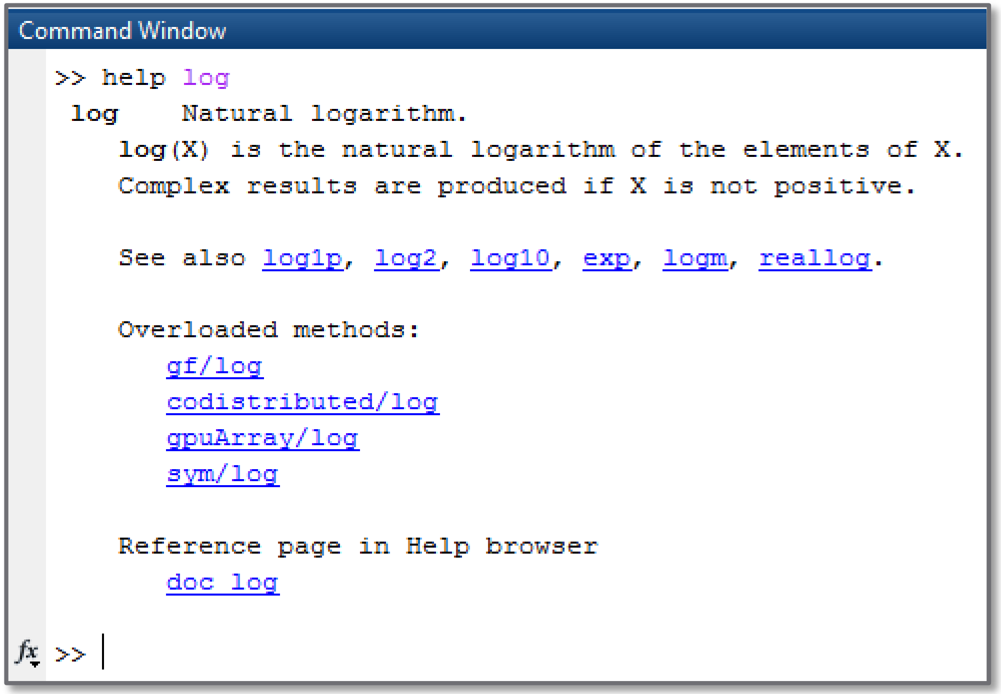
- In MATLAB use `e` for $\times\, 10^{x}$, e.g.:

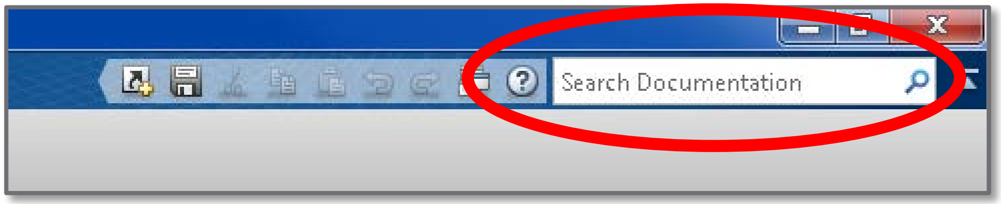
```
x = 1.58e-9;
```

- Don't confuse with e^{x} (i.e. 2.718^{x}) represented by `exp(x)`

MATLAB Help Documentation

- Two ways to access MATLAB help files:
  - Type:
    help <function> at the command line
  - Use the help documentation browser
- Not sure if a function exists to do something you want?
  - It probably does – search for it in the documentation





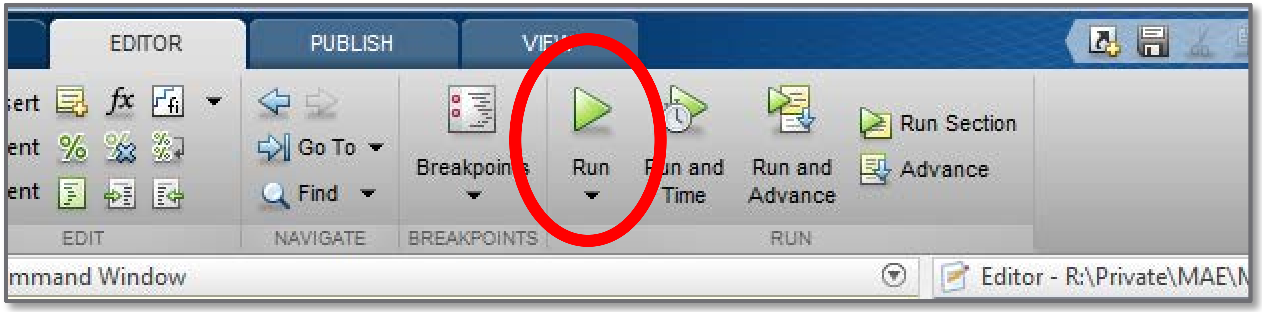
MATLAB Scripts – M-Files

MATLAB Command Window

- As we've seen, we can enter commands into MATLAB through the ***command window***
  - Useful for quick calculations, debugging, etc.
  - Enter one expression at a time
  - To execute a sequence of commands repeatedly, must re-enter all commands each time
  - Command history is only record of executed commands
- Better practice is to write all commands to be executed in a single file or ***script***, called an ***m-file***

M-Files

- ***M-files*** are files containing a series of MATLAB commands
  - These are ***scripts*** or ***programs***
  - So called due to the `.m` filename extension
  - Quickly and easily re-run at any time – no need to re-type all commands in the command window
  - Executed by entering the m-file name on the command line or by clicking the ***Run*** button



- Our primary mode of interaction with MATLAB

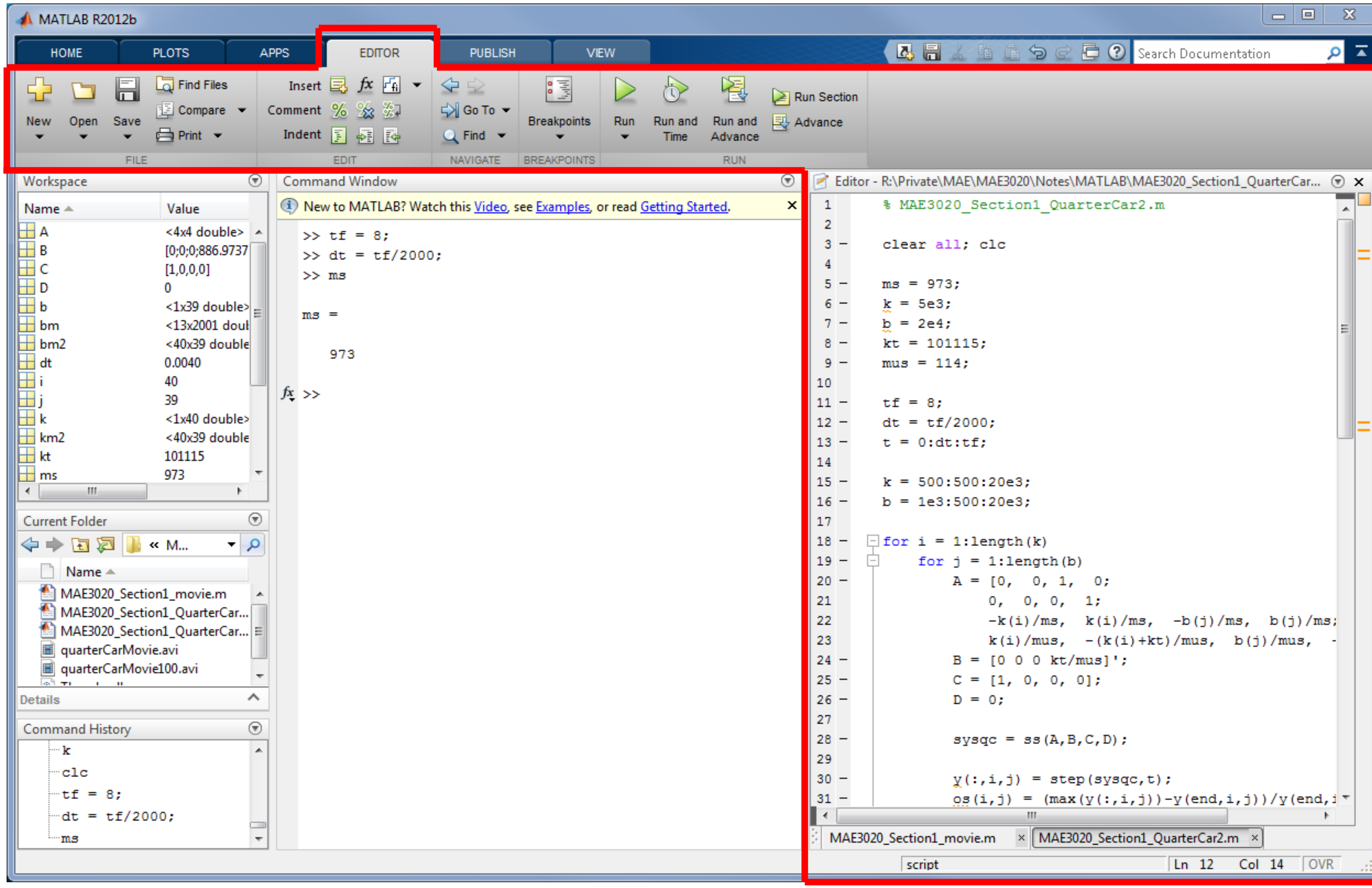
Scripts vs. Programs

- We'll use the terms ***scripts*** or ***programs*** interchangeably when referring to MATLAB m-files
- Technically, m-files are scripts, but this distinction is not important for our purposes.

- **Programs**
  - Written (possibly) in a high-level language – ***source code***
  - ***Compiled*** (once) by a ***compiler*** into a ***machine language*** executable file – ***object code***
  - Fast, because compilation performed once, ahead of runtime

- **Scripts**
  - High-level source code is ***interpreted*** and executed line-by-line by an ***interpreter*** at runtime
  - Slower than compiled programs

MATLAB Editor



M-File Naming Requirements

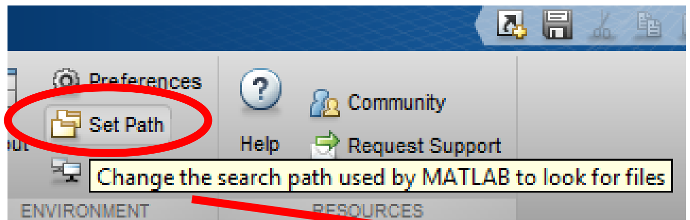
- M-file names must ***start with a letter***
- Names may contain ***letters***, ***numbers***, and ***underscore*** characters
  - ***No spaces***
- Names are ***case sensitive***
- Don't name m-files with names of ***built-in functions***
  - Can be done, but may lead to confusion
  - Local m-file will take precedence over the built-in function – determined by the MATLAB path

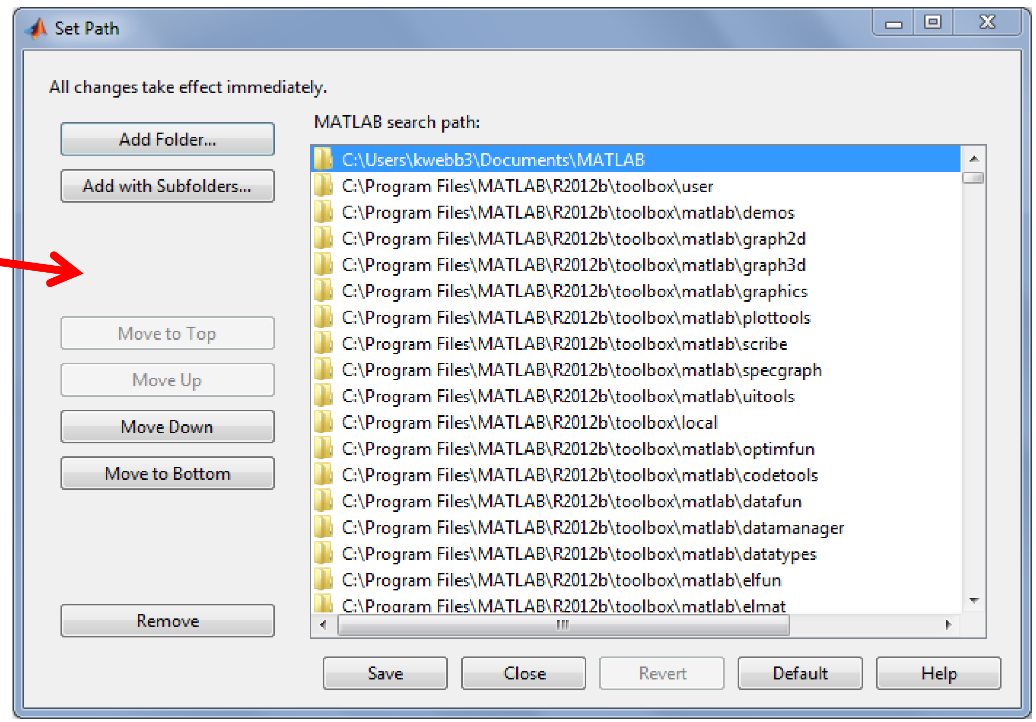
The MATLAB Path

- M-files can be executed three ways
  - Click the “Run” button (see p. 4)
  - Enter the m-file name at the command line
  - Call the m-file by name from within another m-file
- But, the m-file must be in the ***MATLAB path***
- The path is an ordered list of directories where MATLAB looks to find m-files when called
- The ***MATLAB path includes***:
  - The present working directory
  - All MATLAB libraries of built-in functions
  - Any directory that you explicitly add to the path

The MATLAB Path

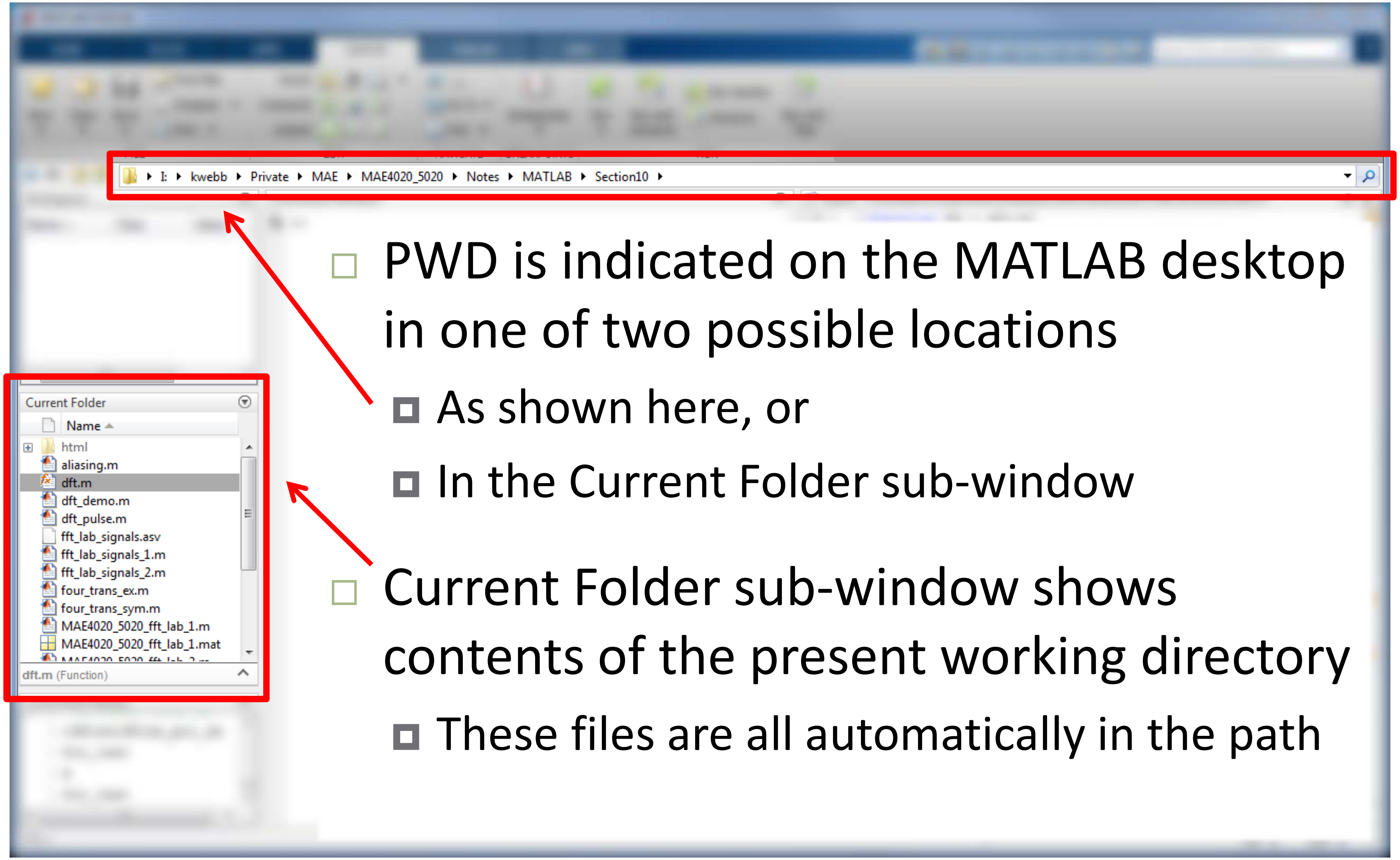
- All m-files outside of the PWD – user-defined or built-in – must be in the ***path*** to be accessed





- For now, this means you must set the PWD to be the location of the m-file you're working with

Present Working Directory – Current Folder



- PWD is indicated on the MATLAB desktop in one of two possible locations
  - As shown here, or
  - In the Current Folder sub-window
- Current Folder sub-window shows contents of the present working directory
  - These files are all automatically in the path

M-Files – Best Practices

Start m-files with a comment listing the file name. ***File names cannot contain spaces or special characters***.

Additional comments with a brief overall m-file description and other details is useful.

`'clear all'` clears all variables stored in the workspace.

`'clc'` clears the command window – you'll know if error messages are from the current run.

Define constants to be used in equations. Parameters can be changed in a single place.

```
% ode_ex.m

% K. Webb 4/24/14
% Simulation of the motion of a pendulum of specified length
% in response to specified initial conditions

clear all; clc

l = 0.5;                    % length of the pendulum [m]
theta0 = -10*pi/180;        % initial angle [deg]
w0 = 0;                     % initial ang. velocity [rad/sec]
x0 = [theta0, w0];          % vector of initial conditions
ti = 0;                     % initial time for simulation
tf = 10;                    % final time for simulation
tspan = [ti,tf];

% Try different tolerance settings
% RelTol = 1e-3 can be unstable for large theta0
options = odeset('RelTol',1e-6);
[t,x] = ode45(@pendode,tspan,x0,options,l);

```

Always comment your code.
Err on the side of excessive comments.

Comments

- Comments are explanatory or descriptive text added to your code
  - Not executed commands
- In MATLAB, comments are preceded by the percent sign: %
- Comments may occupy an entire line
- Or, may be inserted at the end of a line, after uncommented expressions

```
% K. Webb 4/24/14
% Simulation of the motion of a pendulum of specified length
% in response to specified initial conditions

clear all; clc

l = 0.5;                    % length of the pendulum [m]
theta0 = -10*pi/180;        % initial angle [deg]
w0 = 0;                     % initial ang. velocity [rad/sec]
x0 = [theta0, w0];          % vector of initial conditions
ti = 0;                     % initial time for simulation
tf = 10;                    % final time for simulation
```

- Ctrl+R comments a line of text in the MATLAB editor
- Ctrl+T uncomments a line
- Commenting is useful for temporarily removing instructions from an m-file

Clearing the Workspace – `clear.m`

- Good practice to clear all variables from memory at the start of an m-file
  - Prevents problems due to variables of the same name from previous runs or other m-files
- Use `clear.m` to clear the entire workspace:

```
clear all;
```

- Or, to clear individual variables, e.g.:

```
clear x A N
```

or

```
clear('x','A','N')
```

Clearing the Workspace – Example

- M-file to generate a vector of N = 10 random numbers:

```
% clear_test.m

% demonstration of the importance of clearing the workspace
clc

N = 10;              % length of x

for i = 1:N
    x(i) = i*rand;  % a uniformly-dist. random number scaled by i
end

display(x);
```

- The result:

Command Window

```
x =

  Columns 1 through 7

    0.5472    0.2772    0.4479    1.0300    4.2036    1.5257    5.7000

  Columns 8 through 10

    1.9482    8.3634    3.4998
```

Clearing the Workspace – Example

- Run again for N = 5 without clearing the workspace:

```
% clear_test.m

% demonstration of the importance of clearing the workspace
clc

N = 5;                % length of x

for i = 1:N
    x(i) = i*rand;   % a uniformly-dist. random number scaled by i
end

display(x);
```

- x still contains 10 numbers – last 5 from the previous run:

```
Command Window

x =

  Columns 1 through 7

    0.1966    0.5022    1.8481    1.8932    1.7583    1.5257    5.7000

  Columns 8 through 10

    1.9482    8.3634    3.4998
```

Clearing the Workspace – Example

- Run again for N = 5, but now clear the workspace:

```
% clear_test.m

% demonstration of the importance of clearing the workspace
clear all; clc

N = 5;                  % length of x

for i = 1:N
    x(i) = i*rand;  % a uniformly-dist. random number scaled by i
end

display(x);
```

- Now, the length of x is 5, as expected:

```
Command Window

x =

    0.8308    1.1705    1.6492    3.6688    1.4292

fx >>
```

clc.m

- Issuing the command `clc` clears all text from the command window
- Good practice to always do this at the start of every m-file
- Errors are reported in the command window
  - Want to know if they are from the most recent m-file execution or a previous run
  - If the command window is cleared first, any error messages must be from the most recent run

Pseudocode

- The most important part of the process of writing computer code is ***planning***
  - Determine exactly what the program should do
  - And, how it will do it

- Before writing any code, write a ***step-by-step description*** of the program
  - ***Natural language***
  - ***Graphical*** – flow chart (more later)

- This may be referred to as ***pseudocode***

Programming Process

- **Programming process**:
  - ***Define the problem***
    - Ensure you have a complete understanding of the problem
    - Determine exactly what the program should do
      - Inputs and outputs
      - Relevant equations
  - ***Design the program***
    - ***Pseudocode*** – language-independent
  - ***Write the program***
    - Simple translation from pseudocode
  - ***Validate the program***
    - Do the outputs make sense
    - Test with inputs that yield known outputs
    - Test thoroughly – try to break it

Pseudocode

- Comments can serve as pseudocode
  - Write the comments first
  - Then insert code to do what the comments say
- For example:

```
% This script calculates the theoretical maximum
% power generated by a hydropower facility with
% a given head and flow rate

% clear the workspace/command window

% Define density of water

% Define gravitational acceleration

% prompt user for the amount of head [m]

% prompt user for flow rate [m^3/s]

% calculate the maximum power

% display the power
```

```
% This script calculates the theoretical maximum
% power generated by a hydropower facility with
% a given head and flow rate

% clear the workspace/command window
clear all; clc

% Define density of water [kg/m^3]
rho = 1000;

% Define gravitational acceleration [m/s^2]
g = 9.81;

% prompt user for the amount of head [m]
h = input('Enter head [m]: ');

% prompt user for flow rate [m^3/s]
Q = input('Enter volumetric flow rate [m^3/s]: ')

% calculate the maximum power
P = rho*g*h*Q;

% display the power
fprintf('\nPmax = %1.1f MW\n\n',P/1e6);
```

Sequential Code Execution

- In general code is executed line-by-line ***sequentially*** from the top of an m-file down
- There are, however, very important ***non-sequential code structures***:
  - ***Conditional statements*** – code that is executed only if certain conditions are met
    - `if … else`
    - `switch`
  - ***Loops*** – code that is repeated a specified number of times or while certain conditions are met
    - `for`
    - `while`

Inputs & Outputs

Inputs to Scripts

- Inputs to a script:
  - Assignments of variable values

- Several input methods:
  - At the command line
  - Within the script
  - Specified by user during execution – input.m

User-Specified Input – `input.m`

- Prompt user for value to be assigned to a variable

```
var = input(Prompt)
```

  - *`Prompt`*: a *string* that will be displayed in the command window, prompting the user for an input
  - *`var`*: variable to which the user-specified input is stored

- For example:

```
Ti = input('Enter the initial temperature of the fluid (K):  ');
```

Command Window

```
Enter the initial temperature of the fluid (K):  
```

Outputs from Scripts

- Outputs from scripts:
  - Display of values calculated by the script
- Several output methods
  - Plotting
  - In the command window
    - Omission of trailing semicolon (;) in script
    - `disp.m`
    - `display.m`
    - `fprintf.m`
  - Writing data to files (more later)

`fprintf.m`

- Output formatted data to a string in the command window

```
fprintf(formatSpec,A1,A2,…,An)
```

  - *`formatSpec`*: a *string* – may contain ***formatting sequences*** for insertion of variable values
  - *`A1,A2,…,An`*: variables whose values are to be inserted into the string – one for each formatting sequence in *`formatSpec`*

- For example:

```
fprintf('\nThe value of pi is %0.5f\n\n',pi);
```

Command Window

```
The value of pi is 3.14159

>>
```

Formatting Sequences

Command Window

```
>> s = sprintf('The value of pi is %1.5f.',pi)
```

- String may contain number ***formatting sequences***
  - Percent character (%) followed by conversion sequence

```
%6.4f
```

Indicates the beginning of a formatting sequence

Field width: *minimum* number of character spaces used to display the variable value

Conversion character: here, `f` indicates representation in *fixed-point format*

Precision: # of digits to the right of the decimal point (f, e, or E) or # of significant digits (g or G)

Conversion Characters

- Conversion characters specify how to format variable values within a string

Value Type	Conversion Character
Signed integer	%d
Unsigned integer	%u
Fixed-point notation	%f
Exponential notation (e.g., 1.6e-19)	%e
Exponential notation (e.g., 1.6E-19)	%E
More compact of %e or %f	%g
More compact of %E or%f	%G
Single character	%c
String	%s

Formatting Sequences – Examples

- Fixed-point notation
- Field-width control
- Exponential notation
- Compact format

Command Window

```
>> x = 10e4*pi/2;
>> fprintf('\n\tx = %0.2f\n\n',x);

    x = 157079.63

>> fprintf('\n\tx = %10.2f\n\n',x);

    x =  157079.63

>> fprintf('\n\tx = %0.2e\n\n',x);

    x = 1.57e+05

>> fprintf('\n\tx = %0.2E\n\n',x);

    x = 1.57E+05

>> fprintf('\n\tx = %0.2g\n\n',x);

    x = 1.6e+05

>> fprintf('\n\tx = %0.2G\n\n',x);

    x = 1.6E+05
```