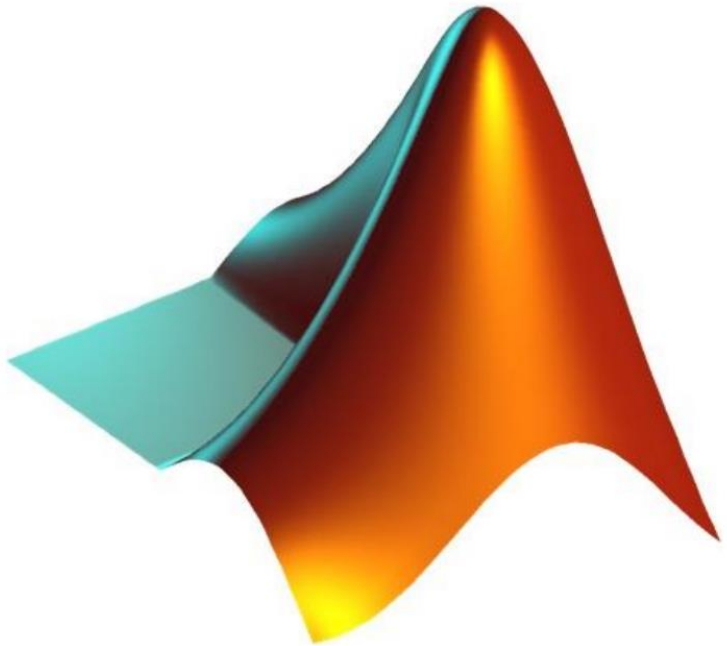
Practical Course MATLAB/SIMULINK

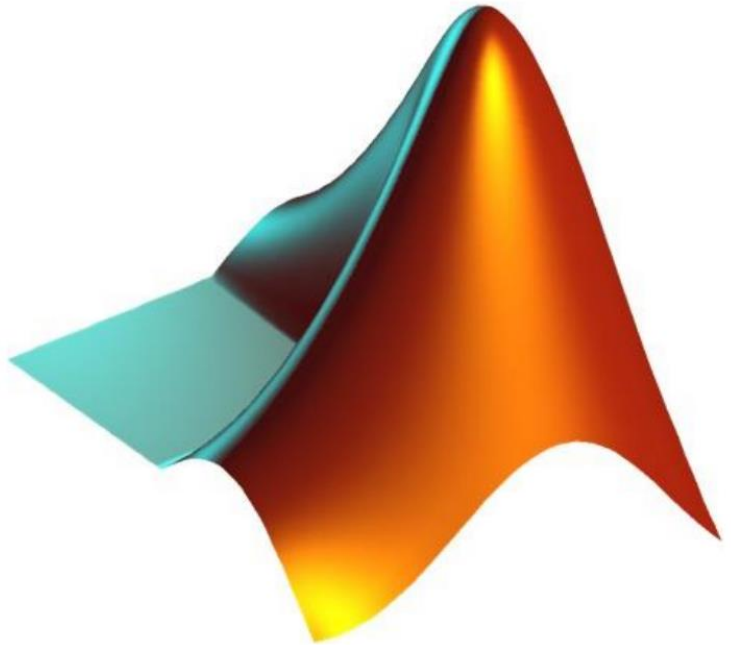
Session 1: MATLAB Fundamentals



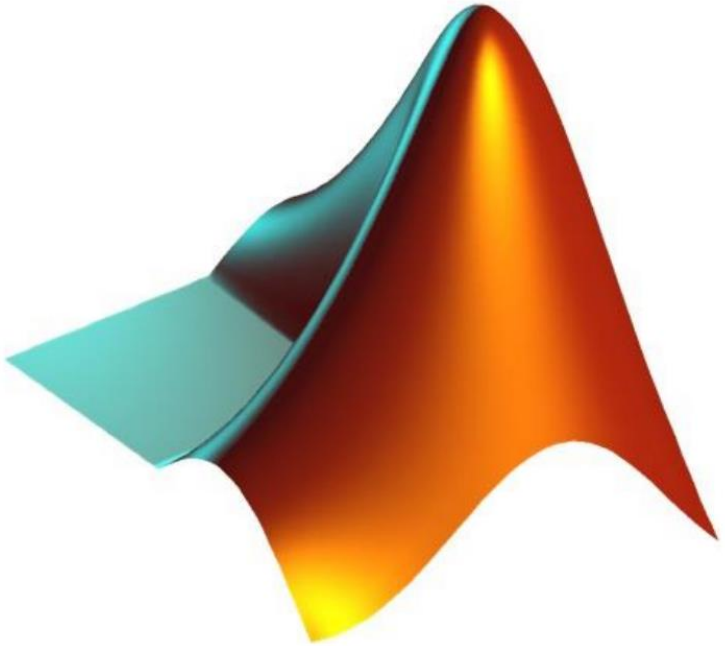
Lecture Objectives & Preparation

- Which MathWorks products are covered?
  - MATLAB
- What skills are learnt?
  - MATLAB interfaces – how to get around
  - Help & documentation
  - Basic coding skills (variables, expressions, code structures…)
  - Debugging
- How to prepare for the session?
  - MathWorks Tutorials:
    - https://matlabacademy.mathworks.com/details/matlab-fundamentals/mlbe
    - https://matlabacademy.mathworks.com/details/matlab-fundamentals/gettingstarted

Lecture Outline

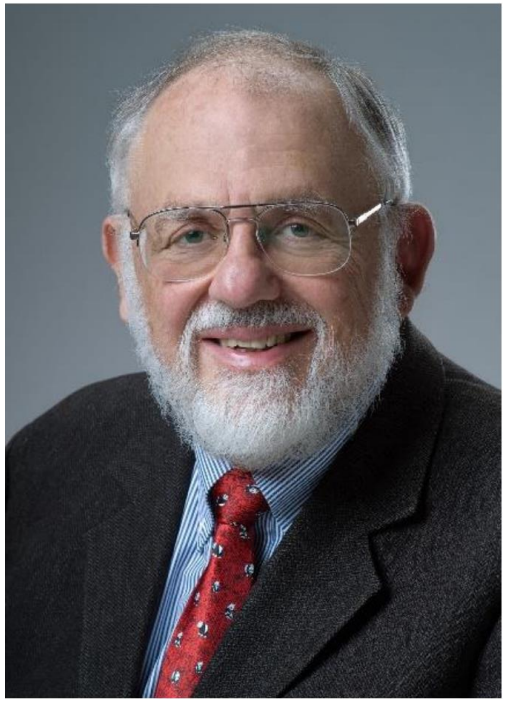


1. Introduction

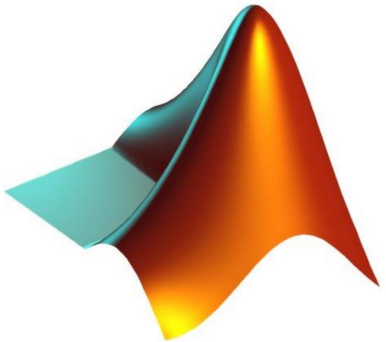


Introduction

- **MAT**rix **LAB**oratory is a numerical computing environment and fourth generation programming language
- Developed by **Cleve Moler**, chairman of the computer science department at the University of New Mexico, in the late 1970s
- Initially designed to give students **easy access** to the software libraries **LINPACK** (numerical linear algebra) and **EISPACK** (numerical computation of eigenvalues and eigenvectors)
- Recognizing the commercial potential, the engineer **Jack Little** joined Moler along with **Steve Bangert** and founded **The MathWorks**
- Today, MathWorks has over **3500 employees** and a yearly **revenue of approximately $1.05 billion**
- MATLAB logo displays **L-shaped membrane** from Moler's PhD thesis

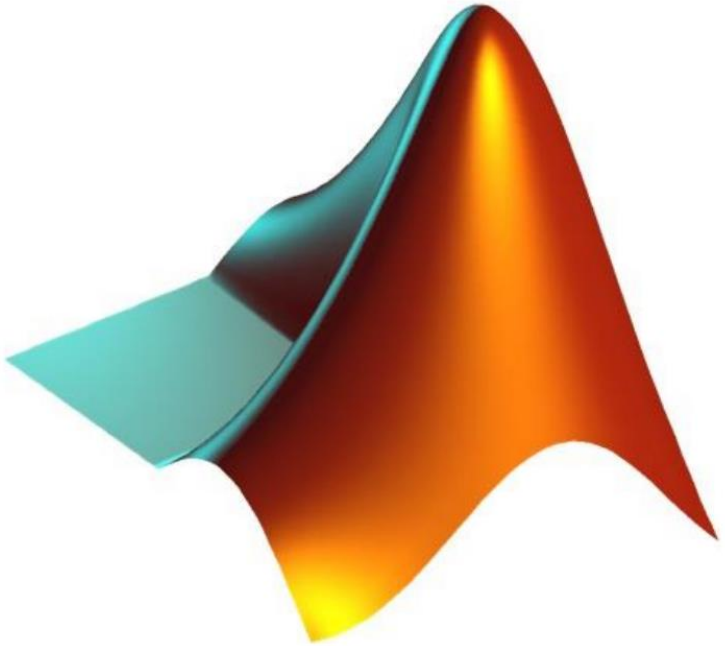


Cleve Moler (mathworks.de)

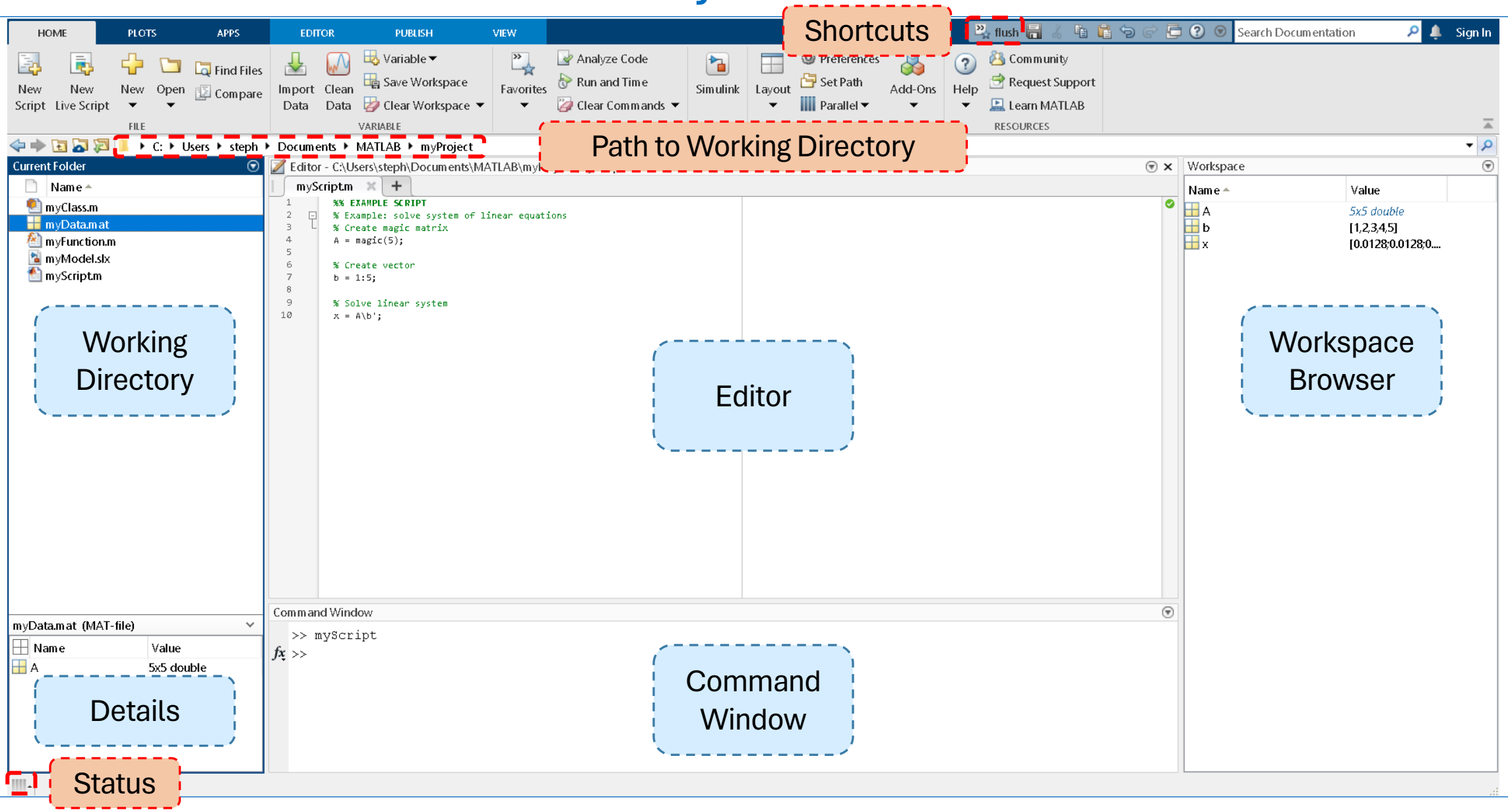


L-shaped membrane

2. Graphical User Interface



Overview of the Default GUI Layout



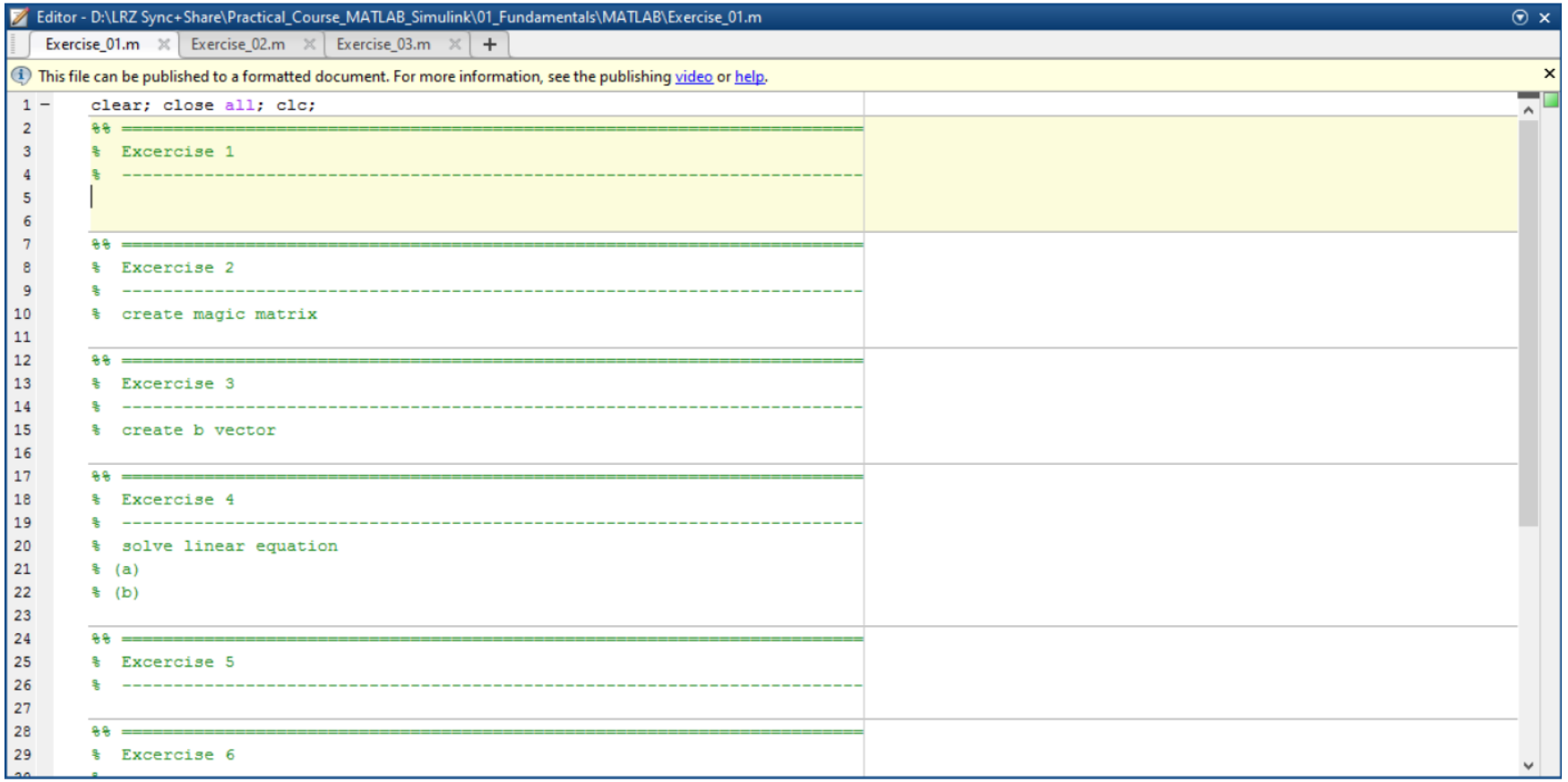
Command Window

- The Command Window has **two basic functions:**
  - Directly **type** and **execute** commands
  - **Display** function return

Editor

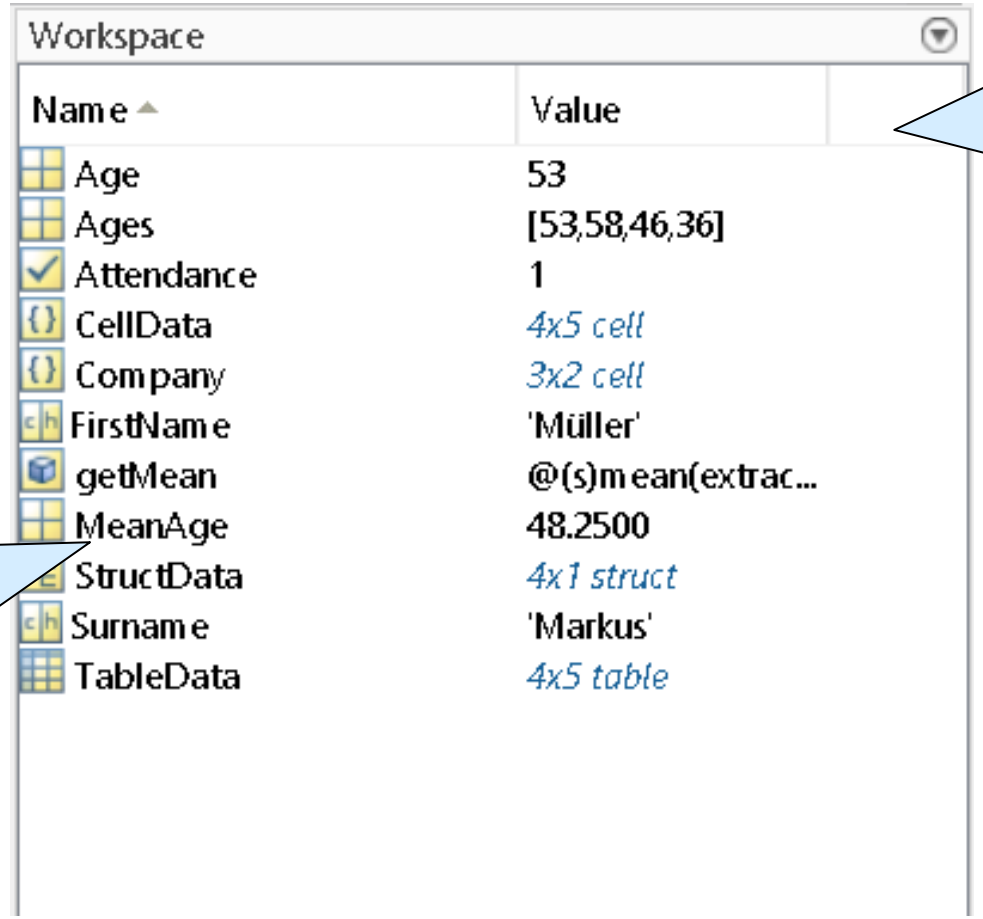
- The Editor is used to **open**, **edit** and **save** programs (e.g. scripts and functions).

```
>> edit Excercise_01.m

>>
```



Workspace Browser

- The Workspace Browser is used to **view** and **edit** variables in the **current workspace.**

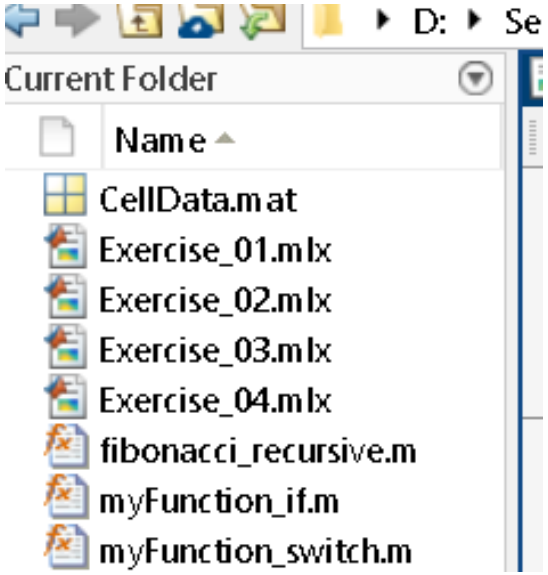


Different properties of each variable can be displayed by right clicking the header row (class, size, etc…)

The **Variable Editor** can be opened by double clicking a variable

Current Working Directory

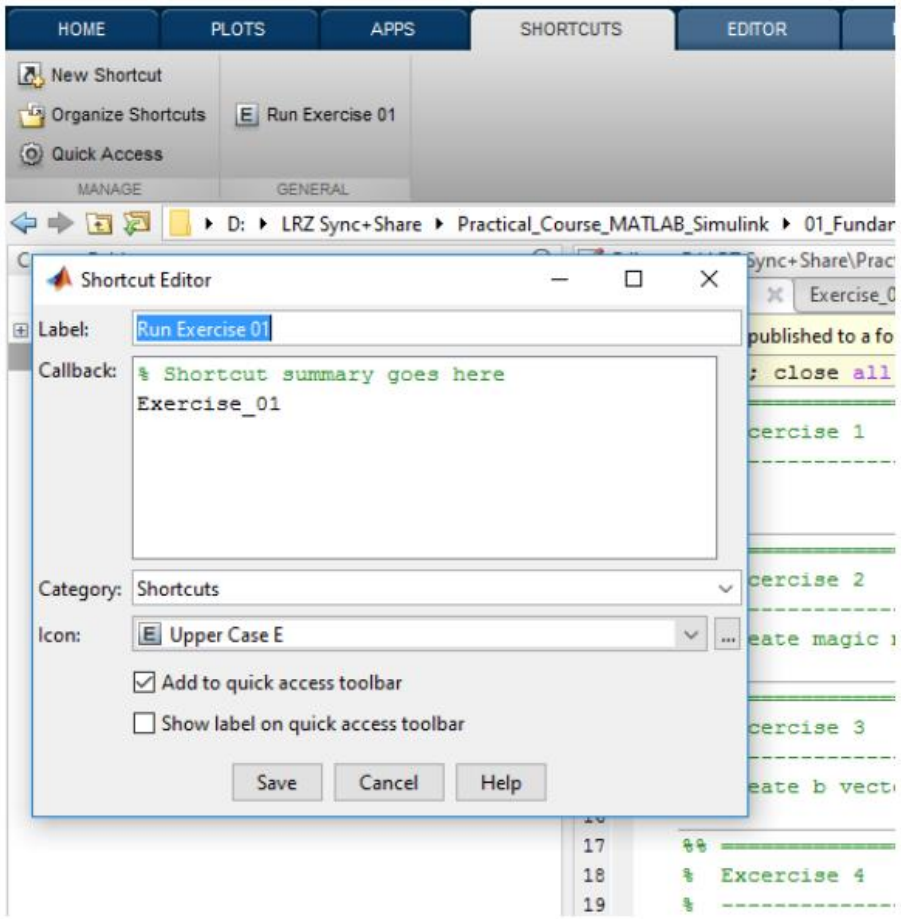
- The Current Working Directory:
  - Contains **active files**, that can be **called from a program**
  - Gives an **overview** of the current working directory
  - **Change** current working directory:
    - Interactively
    - By using the **cd** command
  - Additional folders can be **added** to the MATLAB **search path**:
    - Interactively by right clicking the folder
    - By using the **addpath** command




```
>> cd ..
>> addpath Matlab
```

Shortcuts

- You can create **shortcuts** to rerun commands that are used often.
- On more recent versions, they are known as Favorite Commands.
- Some examples may be:
  - format compact
  - clear
  - workspace
  - filebrowser
  - clc
- **Create** shortcuts by selecting “New Shortcut” from the **SHORTCUTS ribbon** or the **Quick Access toolbar**.



Clear Variables & Commands

- To **clear variables** from the Workspace Browser, use the clear function. You can use it:
  - To remove **all the contents** from the workspace:

```
>> clear all
```

  - To remove a **specific item type** or **variable**, type its name:

```
>> clear variables
```

```
>> clear MeanAge
```

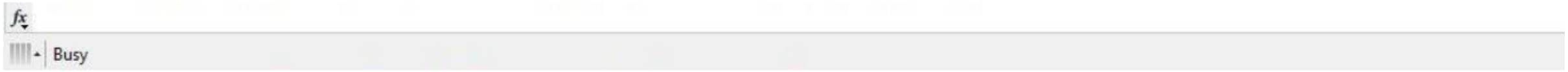
- Good variable hygiene is important, especially when working on Live Scripts.

- To **clear commands** from the Command Windows, use the clc function.

```
>> clc
```

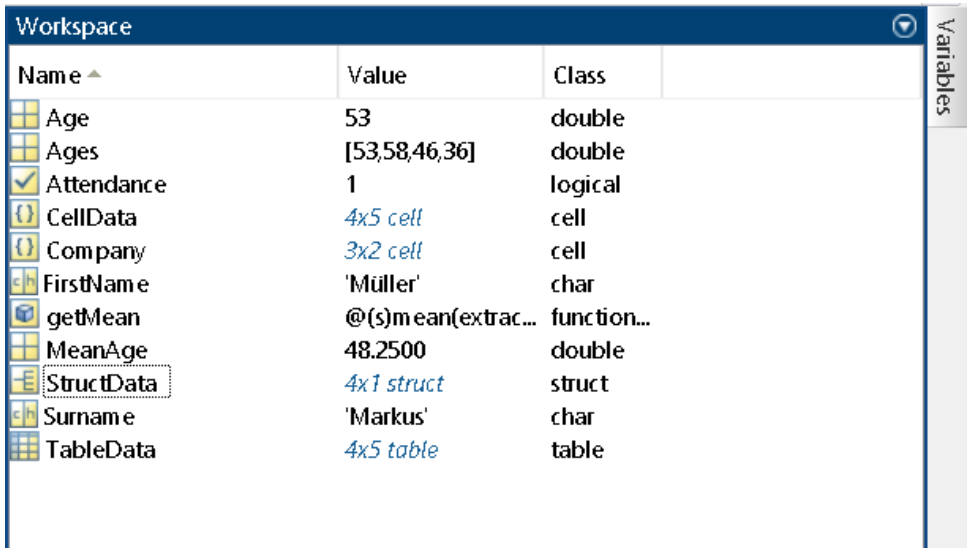
Status

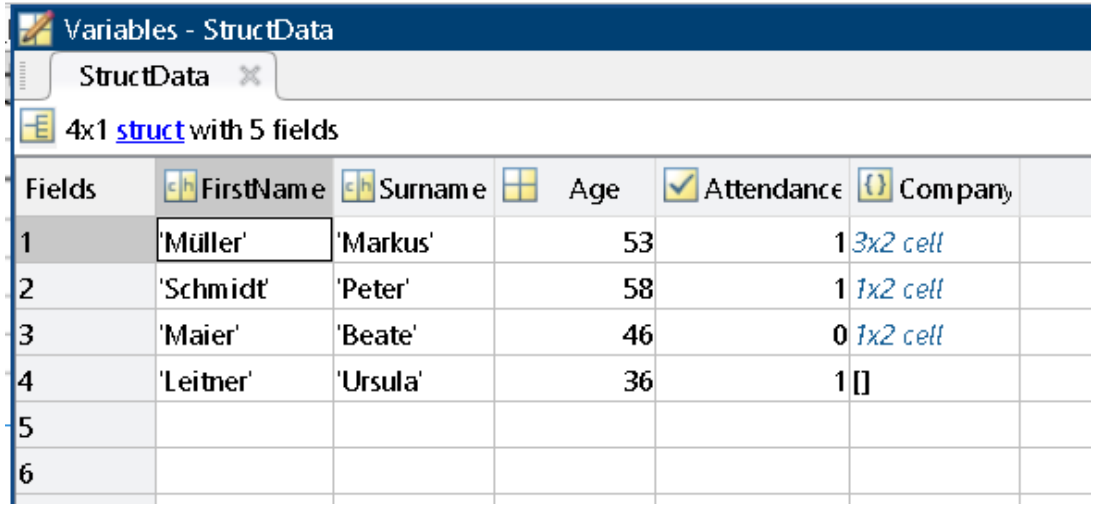
- The status is displayed in the MATLAB status bar.



Variable Editor

- The Variable Editor allows you to **inspect** and **edit** variable in the MATLAB workspace.



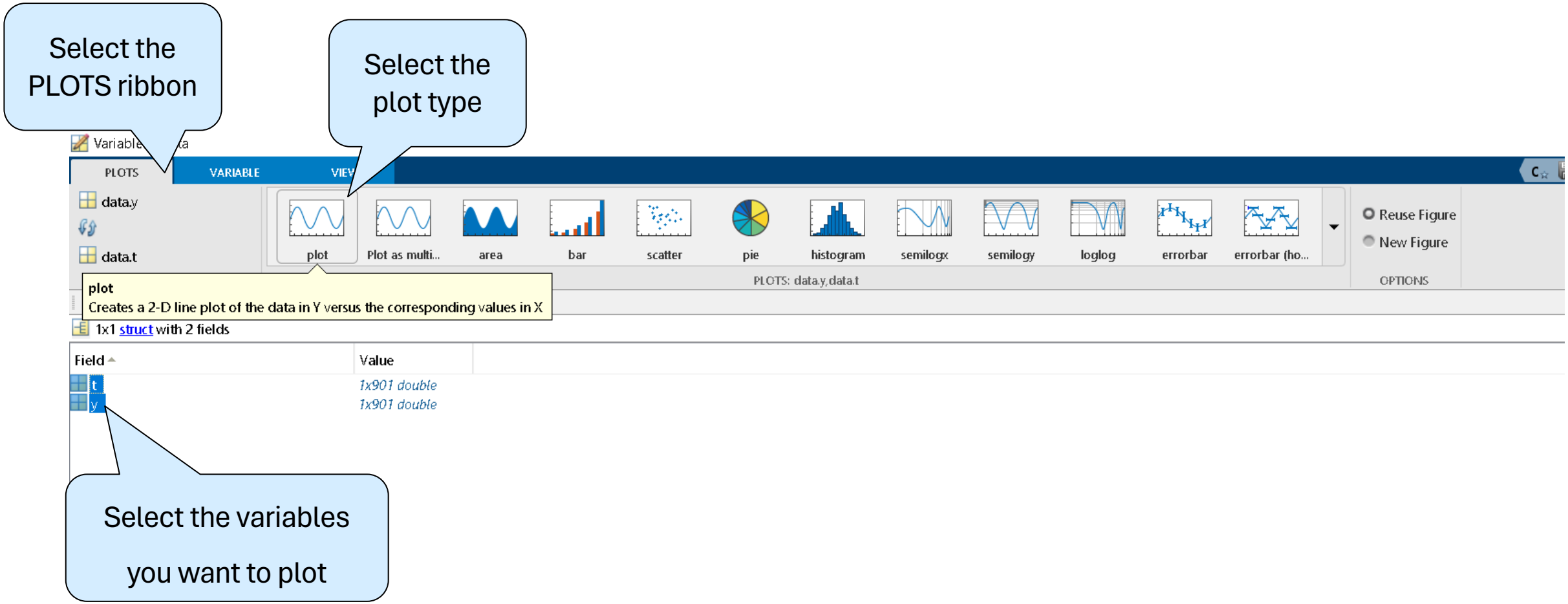


- Open it:
  - Interactively by **double clicking** the variable in the Workspace Browser
  - By using the **openvar** command

```
>> openvar Attendant
```

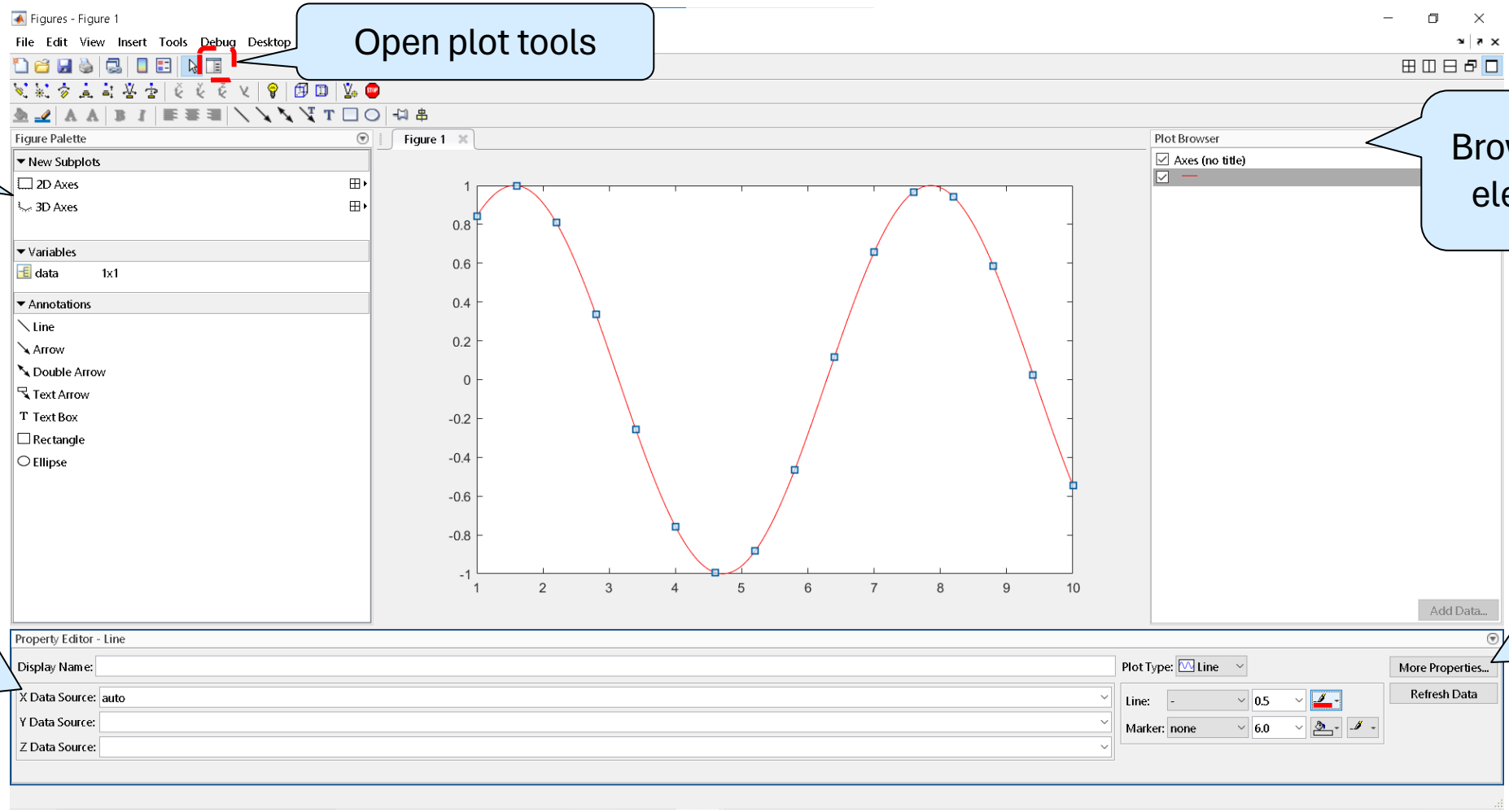
Plotting in the Variable Editor: Plot Generation

- Data can be plotted **interactively** from the Variable Editor.

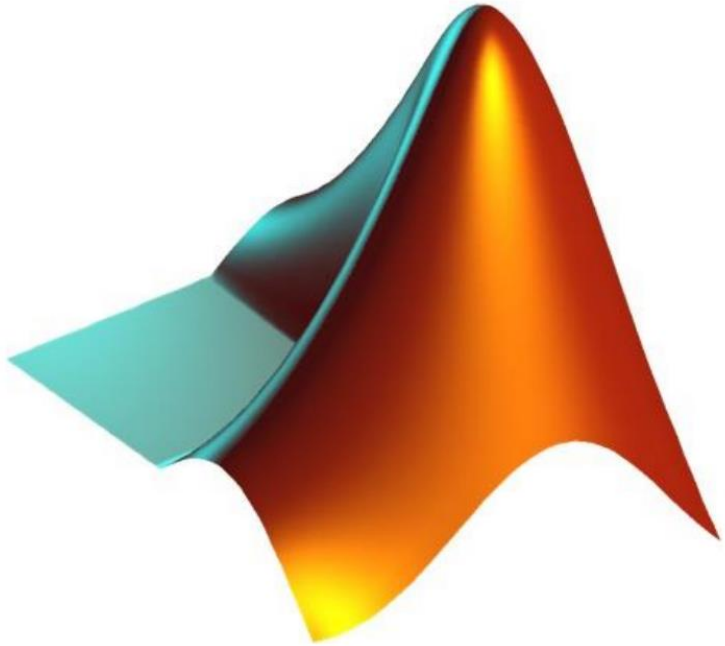


Plotting in the Variable Editor: Plot Tuning

- Plots can be edited **interactively** using the **plot tools**. These tools can be accessed via the **plot's sub-window**.



3. MATLAB help and doc

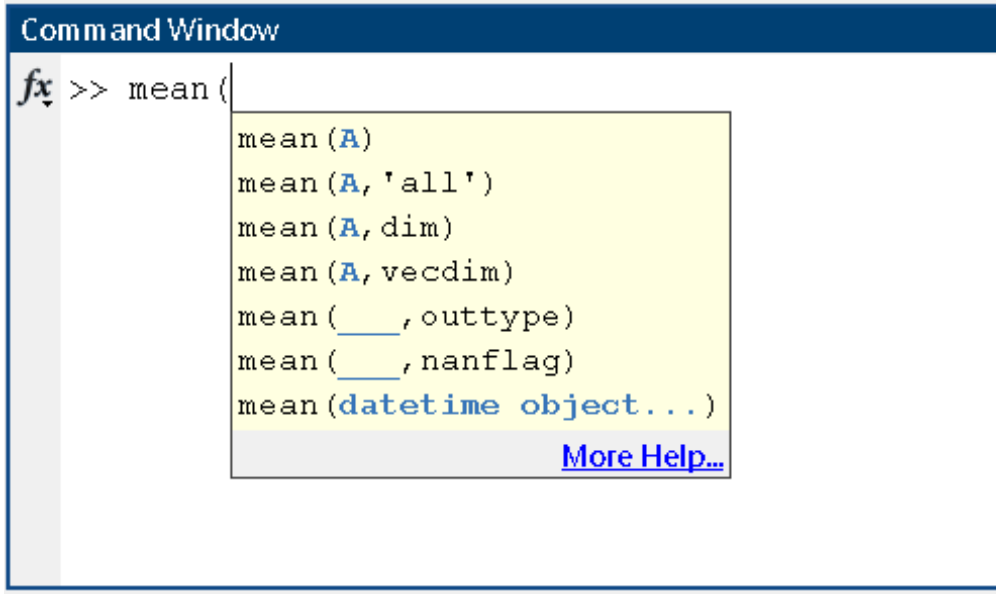


MATLAB help

- **Help** is one of the most **important features** in MATLAB.
- There are several ways to access help:
  - Use the **help** command

```
>> help mean
```

  - Start **typing** the command

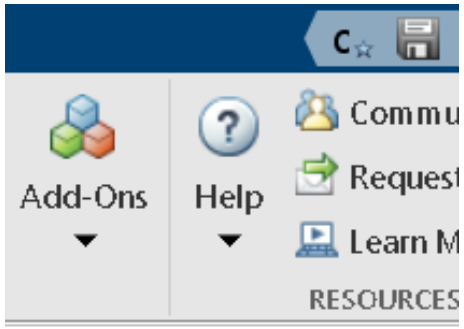


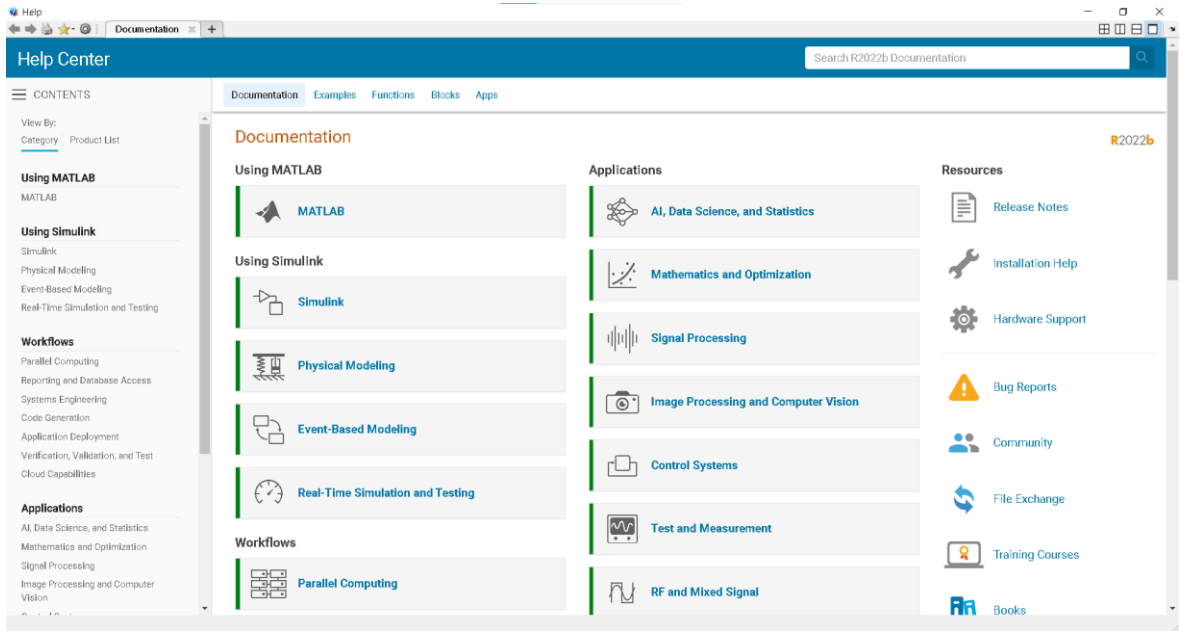
MATLAB doc

- Use the **documentation browser** to view help for all toolboxes in MATLAB. Many of them contain quick start **guides** and easy **examples** to demonstrate the functionality.
- To access documentation:
  - Use the doc command

```
>> doc mean
```

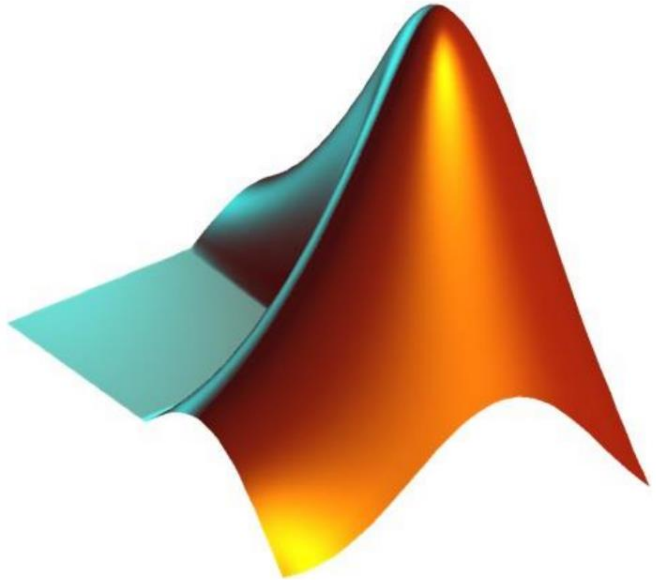
  - Search directly the documentation





4. Variables and Expressions

4.1. Commands and Assignments



Variable Assignment and ans Variable

- The ans variable automatically stores **most recent answer** when **no output argument is specified**.

```
>> 1 + 1
ans =
     2
```

- Command return is **assigned to a variable, when specified**. The colon (;) **suppresses** output in the command window.

```
>> b = ans * 2
b =
     4
>> c = b^2;
>> c
c =
    16
```

Variable Naming

- Variable names must:
  - **Start with a letter**, followed by letters, numbers and underscores
  - **Not** be MATLAB **keywords**
- Invalid variable names:

```
>> while = 1;
>> 6x = 1;
>> n! = 1;
>> my home = 1;
```

- They are:
  - **Case sensitive**
  - **Limited in length**: shorter than the return value of the namelengthmax command

Basic Math Functions

- In MATLAB, a large variety of **built-in math functions** is available. You can find an overview by searching “Mathematics” in the documentation.

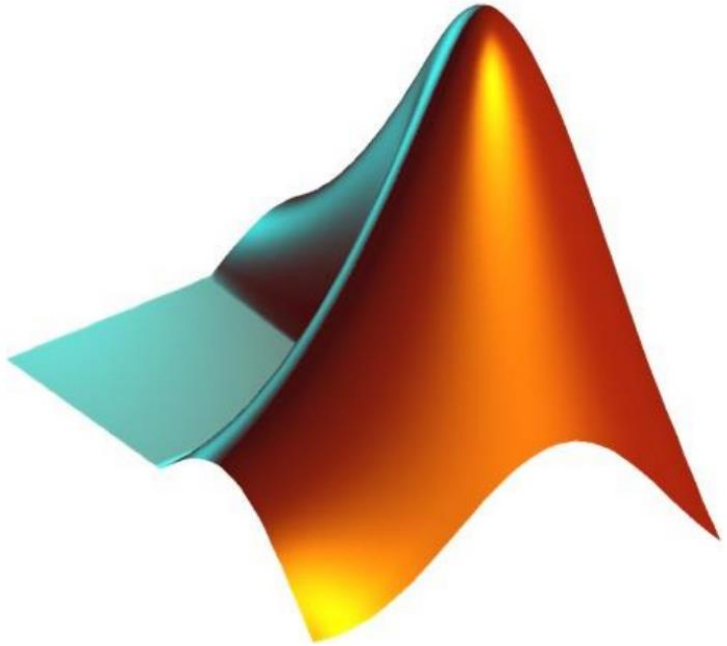
```
>> sin(pi/2)
ans =

     1
```

```
>> exp(i*pi)+1
ans =

   0.0000e+00 + 1.2246e-16i
```

Notice that, by default, calculations are performed numerically!

```
>> eps % machine epsilon
ans =

   2.2204e-16
```

4.2. Arrays, Vectors and Matrices



Creation

- Use the [] operator to **create arrays**:
  - Columns are separated by a comma (,)
  - Rows are separated by a semicolon (;)

```
>> Matrix = [1,2,3;4,5,6]
Matrix =
     1     2     3
     4     5     6
```

- The colon operator (:) can be used to create number series'

```
>> Matrix2 = [[1;5],[2:4;6:2:10]]
Matrix2 =
     1     2     3     4
     5     6     8    10
```

- Use the size command to determine the matrix's dimension

Special Matrices

- **Special matrices** can be created by using various commands including:

diag(), eye(), true(), false(), linspace(), logspace(), meshgrid(), ngrid(), ones(), zeros(), rand(), randn(), nan()…

```
>> diag([1,2,3])
ans =
     1     0     0
     0     2     0
     0     0     3
>> eye(2,3)
ans =
     1     0     0
     0     1     0
>> linspace(0,10,6)
ans =
     0     2     4     6     8    10
```

Concatenation

- Several matrixes can be to a **combined matrix** using the cat, vertcat, horzcat or [] commands.

```
>> horzcat(Matrix1,Matrix2)
ans =
     1     2     5     6
     3     4     7     8
>> cat(3,Matrix1,Matrix2)
ans(:,:,1) =
     1     2
     3     4
ans(:,:,2) =
     5     6
     7     8
>> size([Matrix1; Matrix2])
ans =
     4     2
```

Operations

- **Common** matrix operations can be performed in MATLAB.
  - Plus/minus

```
>> Vector1 = [1, 2, 3]; Vector2 = [1, 1, 1]; Vector1 + Vector2;
```

```
ans =

     2     3     4
```

  - Transpose and multiply

```
>> Vector1*Vector2'
```

```
ans =

     6
```

  - Inverse, determinate and eigenvalues

```
>> Matrix = magic(2); disp(inv(Matrix)); det(Matrix)
```

```
   -0.2000    0.3000
    0.4000   -0.1000

ans =

   -10
```

Element-wise Operator

- Using the **element-wise operator** (.), **scalar operations** can be performed on each element of two arrays with **equal dimensions**:

```
>> Vector1 = [1, 2, 3]; Vector2 = 3:-1:1; Vector1.*Vector2
ans =
     3     4     3
>> Matrix.^2
ans =
     1     9
    16     4
>> Matrix^2
ans =
    13     9
    12    16
```

Sorting and Reshaping

- There are various possibilities to **sort** and **reshape** arrays:

```
>> reshape(Matrix,[1,4])
ans =
     1     4     3     2
>> ans(:)
ans =
     1
     4
     3
     2
>> repmat(sort(ans),[1,4])
ans =
     1     1     1     1
     2     2     2     2
     3     3     3     3
     4     4     4     4
```

Reshaping to a column vector equal to:

```
>> reshape(ans,[],1)
```

Indexing

- In MATLAB, there are three ways to select a subset of an array or matrix:
  - **Subscript indexing:** use ( ) operator to access **subscript range** of the matrix

```
>>  A = magic(3); [A(2,[1,3]) A(1,:)]
ans =
     3     7     8     1     6
```

Using a single colon operator (`:`) is equivalent to typing `1:end` and can be used to select entire rows/columns

  - **Linear indexing:** in MATLAB, elements can be accessed using a **linear index** which acts if the matrix has been reshaped to a column vector

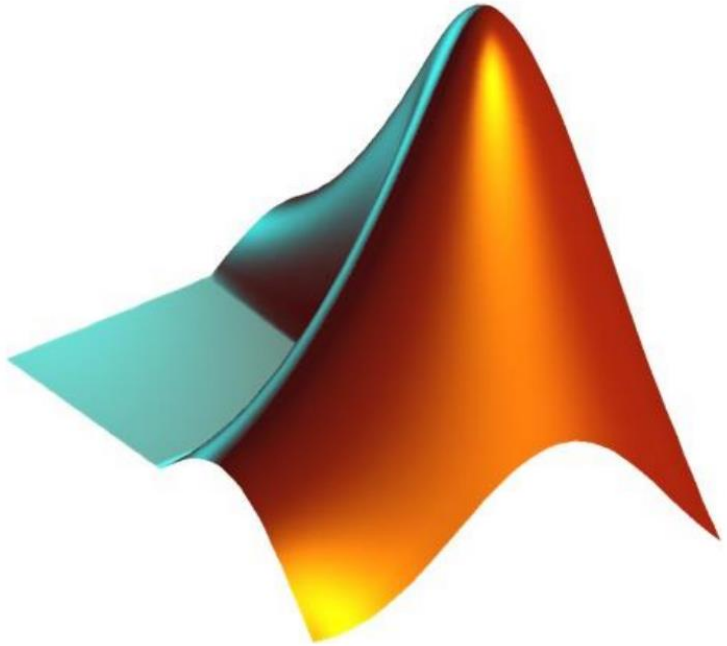
```
>>  A(:) = 1:numel(A); A(2:4)
ans =
     2     3     4
```

  - **Logical indexing:** access all **non-zero entries** of a logical matrix of the size of A

```
>>  A(mod(A,2)==0)'
ans =
     2     4     6     8
```

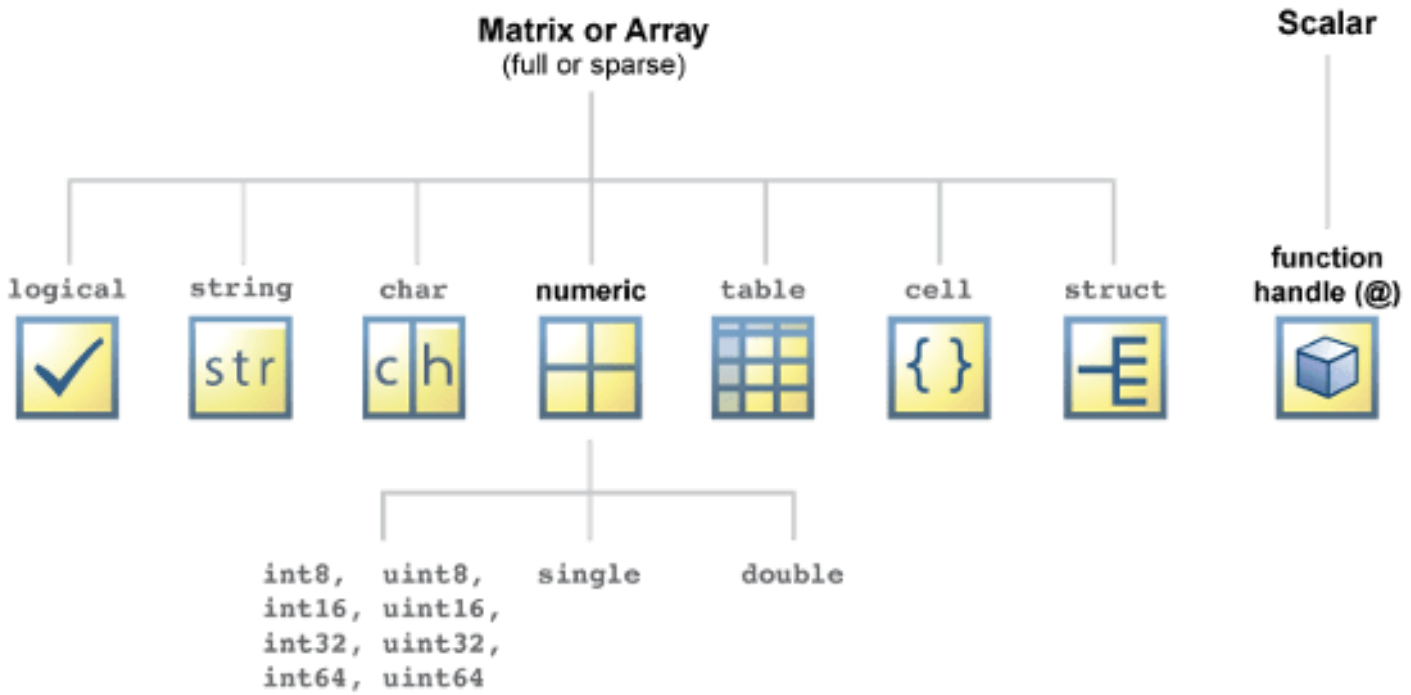
```
>> mod(A,2)==0
ans =
     0     1     0
     1     0     1
     0     1     0
```

4.3. Data Types



Data Types

- Several **data types**, or classes, can be used in MATLAB to work with different data. The data type is **automatically set** by MATLAB when assigning a variable.
- Common data types include:



MATLAB Data Types (mathworks.de)

Logical Data Type

- Boolean data can be stored using MATLAB‘s **logical data type** (**zeros** are treated as false, **all other numeric values** as true).

```
>> A = true; B = 0; C = logical(B); whos A B C
  Name        Size                Bytes  Class     Attributes
  A           1x1                     1  logical
  B           1x1                     8  double
  C           1x1                     1  logical
```

- Using these variables, **Boolean operations** can be performed:

```
>> [A&C, A|C, xor(A,B), ~A]
ans =
     0     1     1     0
```

Logical Operations

- Using these variables, **Boolean operations** can be performed:

```
>> [A&C, A|C, xor(A,B), ~A]
ans =
     0     1     1     0
```

- Short Circuit Logical operations (&& and ||) can be **more efficient**:
  - If the **first operand determines the solution**, the second is **not evaluated**, i.e., since A is true, A||B always returns true and B does not have to be evaluated
  - Similarly, since B is false, A&&B will always return false and A does not have to be evaluated

Character Arrays

- Character arrays, i.e. arrays of **numerical values** that represent Unicode characters, can be used to represent text in MATLAB:

```
>> s = [72   101   108   108   111    32    87   111   114   108   100    33];
>> s = char(s)
s =
Hello World!
```

- Besides regular array operations, **special operations** can be performed such as:
  - parsing: strfind, sscanf, strsplit…
  - comparing: strcmp, strcmpi, strncmp…
  - modification: upper, lower, deblank, strjust…
- Data can be **formatted** into a string using the sprintf command:

```
>> sprintf('The number pi is %2$8.5g and e is %1$4.5g',exp(1),pi)
ans =
The number pi is   3.1416 and e is 2.7183
```

Formatting Operator

```
>> sprintf('The number pi is %2$8.5g and e is %1$4.5g',exp(1),pi)
ans =
The number pi is   3.1416 and e is 2.7183
```

A **formatting operator** is used to **specify the format** of a numeric value in a character array. It takes the form:

mathworks.de

The formatting operator must **start with a %** and **end with a conversion character**. For example, to represent an unsigned integer, we use %d.

Strings

- Strings are created by **enclosing a piece of text in double quotes**. In contrary to character arrays, it is **possible to concatenate** pieces of text into an array:

```
>> str = ["Flight", "System", "Dynamics"]
str = 1x3 string array
    "Flight"    "System"    "Dynamics"
```

- There are many **built-in functions** to manipulate strings known from other programming languages, e.g. the plus operator:

```
>> str(1) + " " + str(2) + " " + str(3) + "!"
str =

"Flight System Dynamics!"
```

Strings v.s. Character Cell Arrays

- Besides **easier handling and manipulation**, string arrays are more **space-efficient** than corresponding cell arrays:

```
>> str = ["Flight", "System", "Dynamics"]; cll = {'Flight','System','Dynamics'};
>> whos str cll
```

Name	Size	Bytes	Class	Attributes
cll	1x3	376	cell	
str	1x3	298	string	

Numeric Data Type

- By default, numeric data is stored as **double-precision floating point** (double):

```
>> a = 25; whos a
```

```
  Name        Size              Bytes  Class     Attributes
  a           1x1                   8  double
```

- The data type can be **converted** to a different class using the **corresponding command** (e.g. single):

```
>> b = single(a); whos b
```

```
  Name        Size              Bytes  Class     Attributes
  b           1x1                   4  single
```

- Similarly, other classes (such as strings) can be **converted to numeric values**:

```
>> s = 'Hello World';
>> int8(s)
```

```
ans =
   72   101   108   108   111    32    87   111   114   108   100
```

Full and Sparse Data

- Numeric values can be stored as sparse data to both:
  - **Reduce** the **memory demand**:

```
>> A = zeros(1000); whos A
  Name          Size                Bytes  Class     Attributes
  A          1000x1000            8000000  double
```

```
>> B = sparse(A); whos B
  Name          Size                Bytes  Class     Attributes
  B          1000x1000               8024  double    sparse
```

  - **Reduce** the number of **arithmetic operations** (and thus computation time):

```
>> tic; A*rand(size(A));toc
Elapsed time is 0.114932 seconds.
>> tic; B*rand(size(A));toc
Elapsed time is 0.025196 seconds.
```

Cell Arrays

- A **cell array** is a data type with **indexed data containers** called cells, where each cell can contain **any type of data**.
  - Use the ( ) operator to refer to the cell:

```
>> PatientData = {'Smith',38,71;'Johnson',43,69;'Williams',38,64;'Jones',40,67}
>> PatientData(:,2)'
ans =
    [38]    [43]    [38]    [40]
```

  - and the { } operator to refer to its content:

```
>> [PatientData{:,2}]
ans =
    38    43    38    40
```

Applying Functions on Arrays and Cell Arrays

- In some cases, it can be necessary to **apply a function to each element** of an Array or Cell Array.
  - For an **Array**, use the arrayfun function:

```
>> arrayfun(@sqrt,[-1 0 4])
ans =
   0.0000 + 1.0000i   0.0000 + 0.0000i   2.0000 + 0.0000i
```

  - For a **Cell Array**, use the cellfun function:

```
>> C = {1:10, [2; 4; 6], []}; cellfun(@mean,C)
ans =
   5.5000   4.0000   NaN
```

Structures and Structure Arrays

- Structure arrays contain data in **fields** that can be **accessed by name**. You can create a Structure either by:
  - **Assigning a value to a field**. Data can be assigned or accessed using the '.' operator:

```
>> PatientStruct(2).Name = 'Johnson';
```

  - Using the struct command:

```
>> PatientStruct =
struct('Name',PatientData(:,1),'Age',PatientData(:,2),'Height',PatientData(:,3))
```

```
PatientStruct =

5x1 struct array with fields:

    Name

    Age

    Height
```

Tables

- A table is a data type for collecting **heterogeneous data** and metadata properties, such as variable names, row names, descriptions, and variable units, in a **single container**.
  - Data can be **slightly differently than in Structures**:

```
>> PatientTable.Name(1)
ans =
    'Smith'
```

In a Structure, the syntax would have been: PatientStruct(1).Name

  - The **Properties** field of the table contains **information about the table**:

```
>> PatientTable.Properties.DimensionNames
ans = 1x2 cell
    {'Row','Variables'}
```

  - The fields from Properties can be changed to alter the table's appearance:

```
>> PatientTable.Properties.Description = "Cabinet patient data";
```

Creating Tables

- There are two main ways to create a Table:
  - Using the table command:

```
>> table([1:3]',{'one';'two';'three'},categorical({'A';'B';'C'}));
```

  - You can convert a Structure to a Table by using the struct2table command:

```
>> PatientTable = struct2table(PatientStruct(1:3))
```

```
PatientTable =

      Name         Age    Height
    ___________    ___    ______

    'Smith'        39     71
    'Johnson'      44     69
    'Williams'     39     64
```

Function Handles

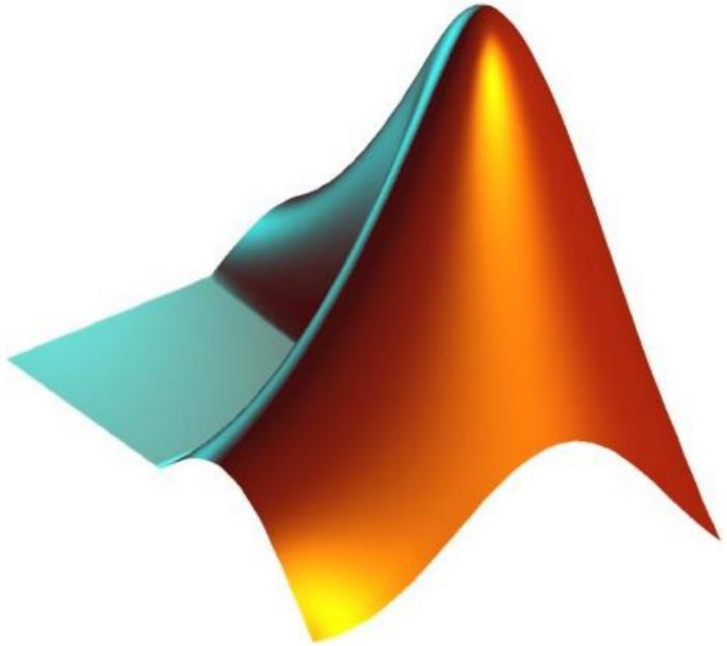
- A function handle stores an **association to a function**. Indirectly calling a function enables you to **invoke the function** regardless of where you call it from.
  - A function handle an be created using the @ command:

```
>> f = @ones
f =
    @ones
>> f(1,2)
ans =
     1     1
```

  - A function handle can store anonymous functions, which is a one-line expression without program file (see Chapter 5: Scripts and Functions of this lecture):

```
>> f = @(x)x^2;f(2)
ans =
     4
```

5. Scripts and Functions



Scripts vs Functions

- Scripts and functions contain programs that consist of a **series of MATLAB statements**, which can be **edited using the MATLAB editor** and stored in a **.m-file**.
  - Scripts are the **simplest** types of programs used to **automate** commands that must be performed repeatedly from the command line
  - Functions offer additional **programming flexibility**:
    - Input and outputs
    - Individual workspace (separate from the base workspace)

```
function [out1, out2] = FuncName(in1, in2)
% calculate area
out1 = in1 * in2;

% calculate circumference
out2 = 2*(in1 + in2);
end
```

```
>> [area, circumference] = FuncName(3,4)
area =
    12
circumference =
    14
```

- Live Scripts (.mlx-file) are an **interactive** form of scripts, allowing the user to run selected sections of code.

Function Syntax

- Functions must be saved in a m-file with the **same name as the function it contains.**
- The syntax elements of a function are:

Syntax Element	Description
Function keyword (required)	MATLAB Keyword function and end
Function name (required)	Valid function names follow the same rules as variable names
Input arguments (optional)	Names of input variables that are used within the function
Output arguments (optional)	Names of output variables that are set within the function

```
1
2  function c = myFunction(a,b)
3
4  c = a*b;
5
6  end
7
8
```

Function Types

- One program file (.m) can contain **several functions** – the main function and a **combination of local and nested functions:**

Type	Description	Location
Local functions	subroutines that are available to any other function within the same file	Same file
Nested functions	Completely contained in another function, can use variables defined in parent function	Same file
Private functions	Like local functions, but can be used by any function within a folder immediately above the private folder	Subfolder called “private”
Anonymous functions	Function that consists of one single expression with no file but completely stored within a function handle	No file

```
1   % myFunction.m
2   function b = myFunction(a)
3   b = squareMe(a) + doubleMe(a);
4
5       function y = doubleMe
6       y = 2.*a;
7       end
8
9   end
10
11  function y = squareMe(x)
12  y = x.^2;
13  end
```

Variable-length Input and Output

- MATLAB supports functions with a **variable number of input and output arguments:**

Keyword	Description
nargin	Holds the number of input arguments passed to the function
varargin	Completely contained in another function, can use variables defined in parent function
nargout	Like local functions, but can be used by any function within a folder immediately above the private folder
varargout	Function that consists of one single expression with no file but completely stored within a function handle

```
1  function [ varargout ] = VarArgsFun( varargin )
2  %print the number of inputs and outputs
3  fprintf('Number of Input Arguments: %i\n', nargin);
4  fprintf('Number of Output Arguments: %i\n', nargout);
5
6  % if there are any inputs --> display in the command
7  window
8  if nargin > 0
9      fprintf('The Inputs are:\n');
10     for i = 1:nargin
11         display(varargin{i});
12     end
13 end
14
15 % if there are any outputs --> create a number
16 sequence
17 if nargout > 0
18     fprintf('Creating Outputs:\n')
19     varargout = cell(1,nargout);
20     for i = 1:nargout
21         varargout{i} = i;
22     end
23 end
24 end
```

Conditional Statements: if

- Conditional statements enable **selecting which code block** to execute at run time.
  - if statement **applies conditions** using the keywords if, elseif and else

```
function Compare(a, b)
if a < b
    disp('smaller')
elseif a > b
    disp('larger')
else
    disp('equal')
end
end
```

```
>> Compare(1,10)
smaller
>> Compare(11,10)
larger
>> Compare(10,10)
equal
```

Conditional Statements: switch

- switch statement tests equalities against a **set of known values** using keywords switch, case and otherwise

```
function WeekDay(dayString)
switch dayString
   case 'Monday'
      disp('Start of the work week')
   case 'Tuesday'
      disp('Day 2')
   case 'Wednesday'
      disp('Day 3')
   case 'Thursday'
      disp('Day 4')
   case 'Friday'
      disp('Last day of the work week')
   otherwise
      disp('Weekend!')
end
end
```

```
>> WeekDay('Tuesday')
Day 2
```

```
>> WeekDay('Saturday')
Weekend!
```

Loop Control Statements: for

- Loop control statements allow for **repeated execution of code blocks**.
  - for loops through a code block for a specific number of times using prespecified values for a loop iterator (similar to the foreach loop in C++)

```
% myScript.m
a = zeros(1,10);
for iter = [3, 5:2:10]
   a(iter) = iter/2*(iter-1);
end
a(5:end)
```

```
>> myScript
ans =
    10     0    21     0    36     0
```

Loop Control Statements: while

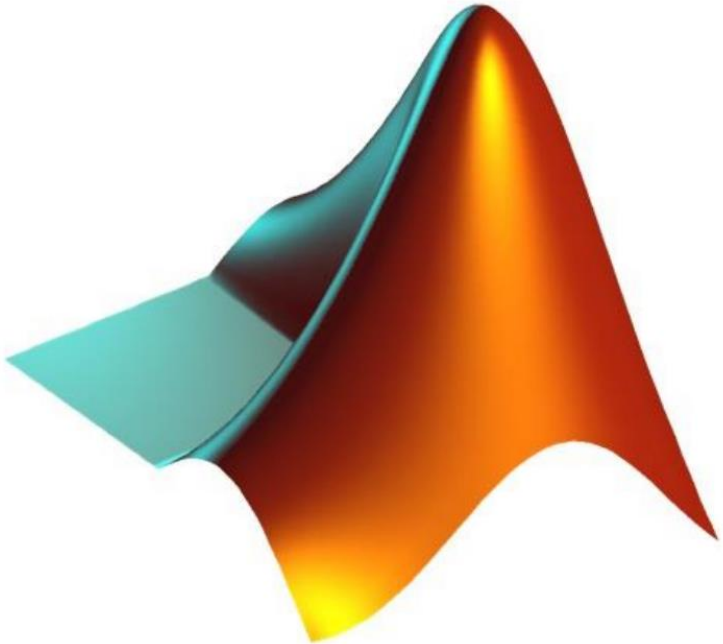
- while loops through a code block as long as a condition remains true (like the while loop in C++)

```
% myScript.m
a = zeros(1,10);
iter = 1;
while iter <= 10
    a(iter) = iter/2*(iter-1);
    iter = iter + 1;
end
a(5:end)
```

```
>> myScript
ans =
    10    15    21    28    36    45
```

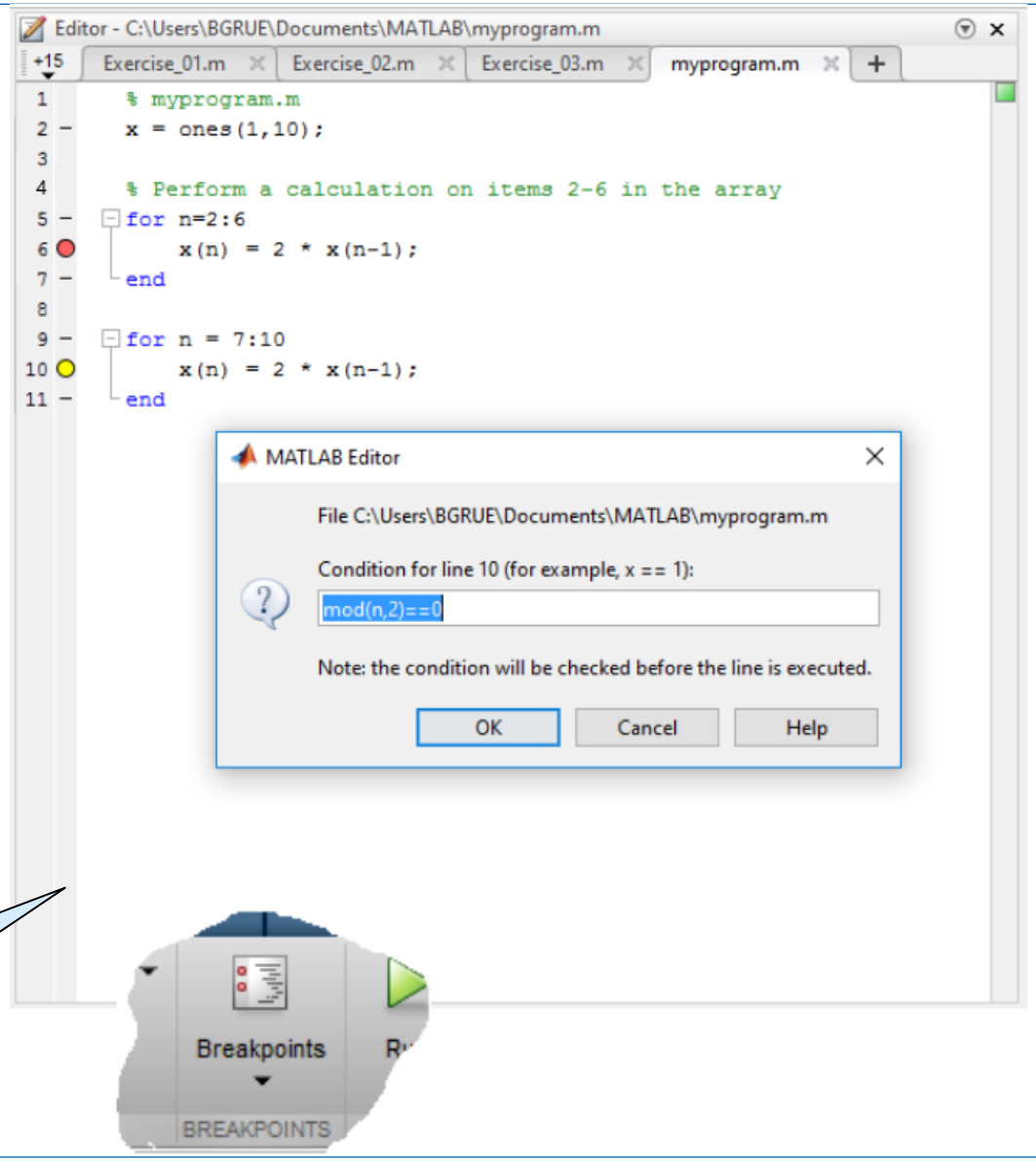
Use the break statement to exit the loop, or skip to the next iteration using the continue statement

6. Debugging



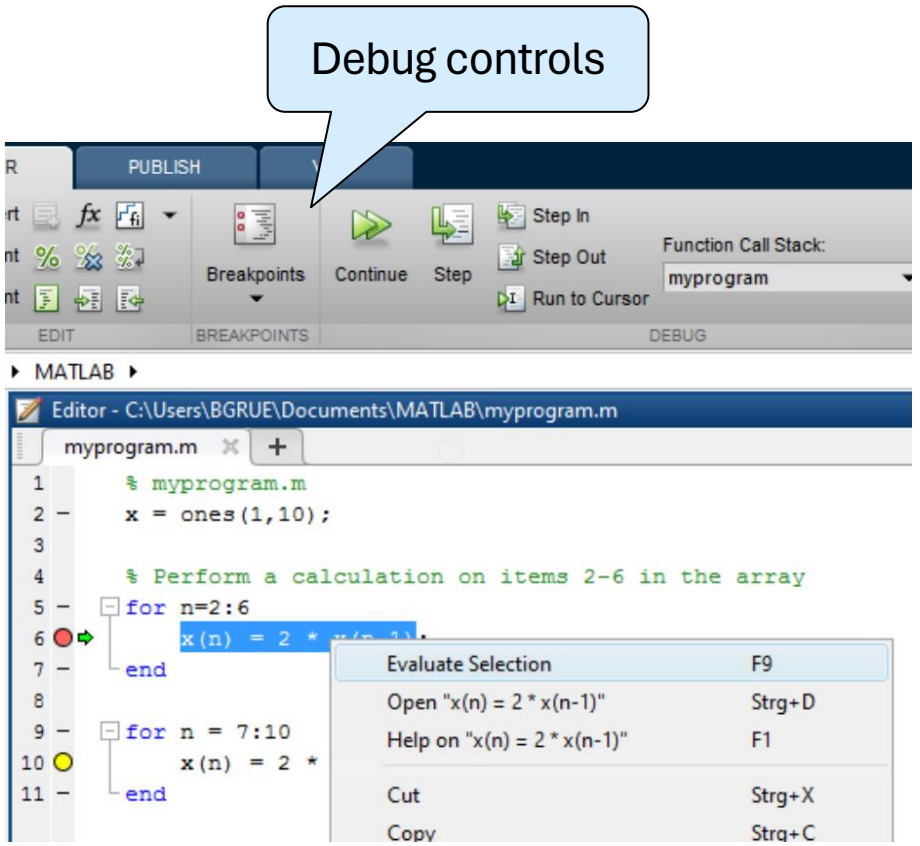
Breakpoints

- Diagnosing Problems in code is a key task in programming. MATLAB provides several features to facilitate this.
- Breakpoints are added by **clicking an executable line** in the Breakpoint Alley, marked by "–". There are **several types of breakpoints**:
  - **Standard**: program stops once it reaches this line
  - **Conditional**: program stops when a specified **conditional statement is fulfilled**
  - **Error**: program stops on **errors or warnings**



Debugging

- To diagnose a program the following steps can be taken:
  - Click “Run” to run the script of function to investigate
  - The code will be stopped at the first active breakpoint
    - **Evaluate** parts of the code by right clicking a selection and selecting “Evaluate Selection” or pressing the F9 key
    - **Step through the program** using the controls in the DEBUG panel of the Editor ribbon
  - **Finish debugging** by
    - Clicking the “Quit Debugging” Button
    - Using the “Continue Button” to run the code until the end of the script or function



Try/Catch

- Errors can be **mitigated** within the code by using a try/catch statement. Using a try/catch statement, **error information can be retrieved** from an MException object created by MATLAB.

```
% myprogram.m
x = ones(1,10);

% Perform a calculation on items in the array
for n=2:6
   x(n) = 2 * x(n-1);
end

try
   for n = 7:11
      x(n) = 2 * x(n);
   end
catch ME
   error(ME.message);
end
```

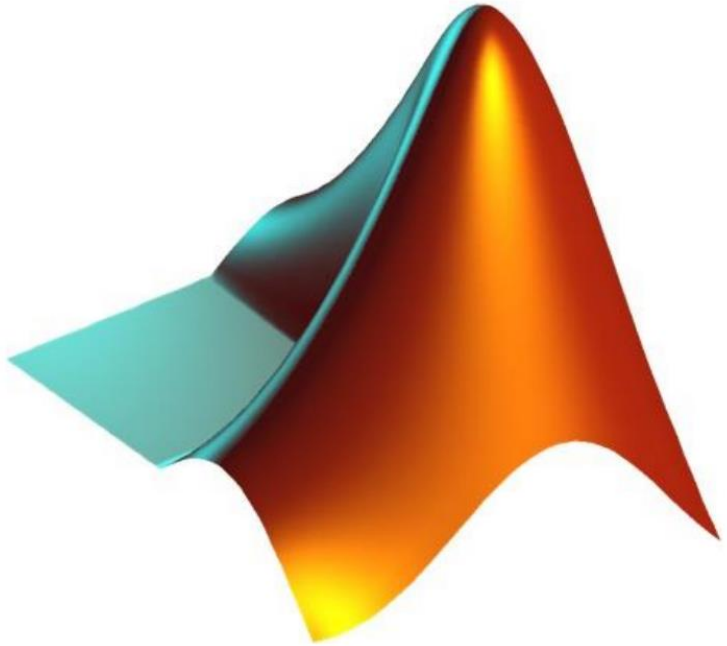
Within the for-loop, n becomes bigger than the number of elements in x

The error command throws another Exception and outputs a message to the Command Window

```
>> myprogram
Error using myprogram (line 14)
Index exceeds matrix dimensions.
```

Within the for-loop n becomes bigger than the number of elements in x

7. Useful Commands



List of Commands (1)

Command	Explanation	Slide #
`edit`	Edit or create file	8
`format`	Set Command Window output display format	11
`clear`	Remove items from workspace, freeing up system memory	11
`workspace`	Open Workspace browser to manage workspace	11
`filebrowser`	Open Current Folder browser, or select it if already open	11
`clc`	Clear Command Window	11
`cd`	Change current folder	12
`addpath`	Add folders to search path	12
`openvar`	Open workspace variable in Variables editor or other graphical editing tool	13
`doc`	Reference page in Help browser	17
`help`	Help for functions in Command Window	17
`ans`	Most recent answer	20
`namelengthmax`	Maximum identifier length	20

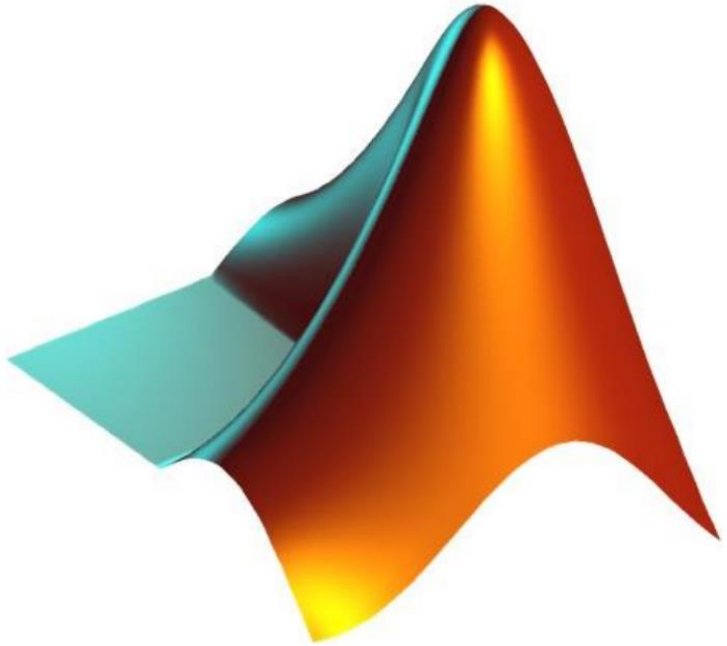
Command	Explanation	Slide #
`clc`	Clear Command Window	21
`sin`	Sine of argument in radians	22
`exp`	Exponential	22
`eps`	Floating-point relative accuracy	22
`diag`	Create diagonal matrix or get diagonal elements of matrix	23
`eye`	Identity matrix	23
`linspace`	Generate linearly spaced vector	23
`cat, vertcat, horzcat`	Concatenate arrays along specified dimension	24
`size`	Array Dimensions	24
`magic`	Magic square	25
`disp`	Display value of variable	25
`inv`	Matrix inverse	25
`det`	Matrix determinant	25
`reshape`	Reshape array	27

List of Commands (2)

Command	Explanation	Slide #
`repmat`	Repeat copies of array	27
`sort`	Sort array elements	27
`numel`	Number of array elements	28
`mod`	Remainder after division (modulo operation)	28
`logical`	Convert numeric values to logicals	30
`whos`	List variables in workspace, with sizes and types	30
`char`	Convert to character array (string)	31
`sprintf`	Format data into string	31
`single`	Convert to single precision	32
`int8, int16, int32`	Convert to 8/16/32-bit signed integer	32
`zeros`	Create array of all zeros	33
`sparse`	Create sparse matrix	33
`rand`	Uniformly distributed random numbers	33

Command	Explanation	Slide #
`tic`	Start stopwatch timer	33
`toc`	Read elapsed time from stopwatch	33
`cellfun`	Apply function to each cell in cell array	34
`struct`	Create structure array	35
`struct2table`	Convert structure array to table	36
`try, catch`	Execute statements and catch resulting errors	51

8. Self-assessment



Self-assessment

- Create your own Shortcut (aka Favorite Command).
- Plot Sine and Cosine using the Variable Editor's Figures tool.
- What are the matrix transpose and element-wise operators in MATLAB?
- What are the three types of Array indexing?
- Create your own Cell Array. What can a Cell Array do, that an Array can't?
- Find the documentation of arrayfun and cellfun and use them on an example of your own.
- How do you index Structures and Tables?
- By hand, write a simple function that displays in the Command Window if its first input is a factor of the second input. Code it and check its syntax.
- List all types of functions you know.
- What loop statements do you know?
- What conditional statements do you know?