CEE221 – Matrix Structural Analysis

Lecture-3: Matlab

Petros Komodromos, *komodromos@ucy.ac.cy*

Department of Civil & Environmental Engineering

University of Cyprus

Topics

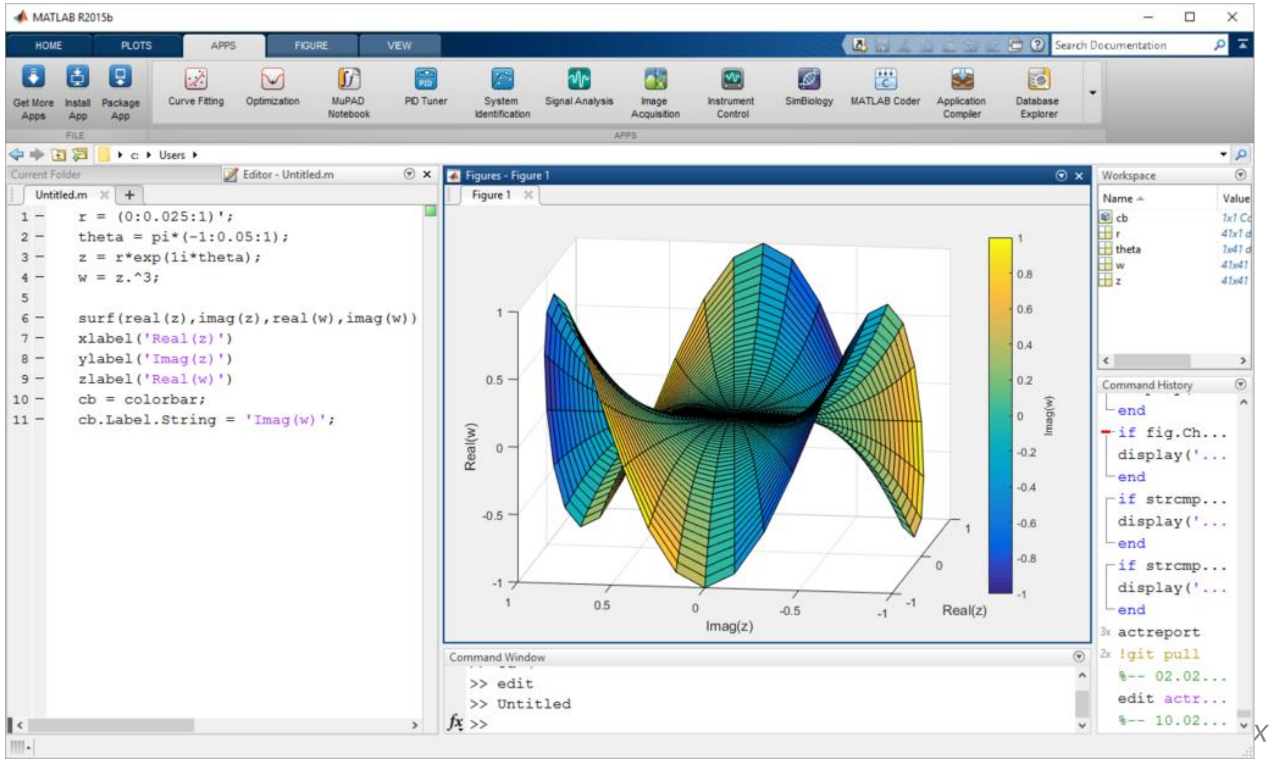
- Introduction to Matlab
- Installing and using Matlab at the UCY
- Getting started with Matlab and its environment
- Defining, manipulating and using simple variables
- Operations and mixed expressions
- Mathematical functions
- Input/Output
- Defining, manipulating and using vectors and arrays
- Functions defining/computing special arrays
- Operations on matrices
- Solving systems of algebraic equations

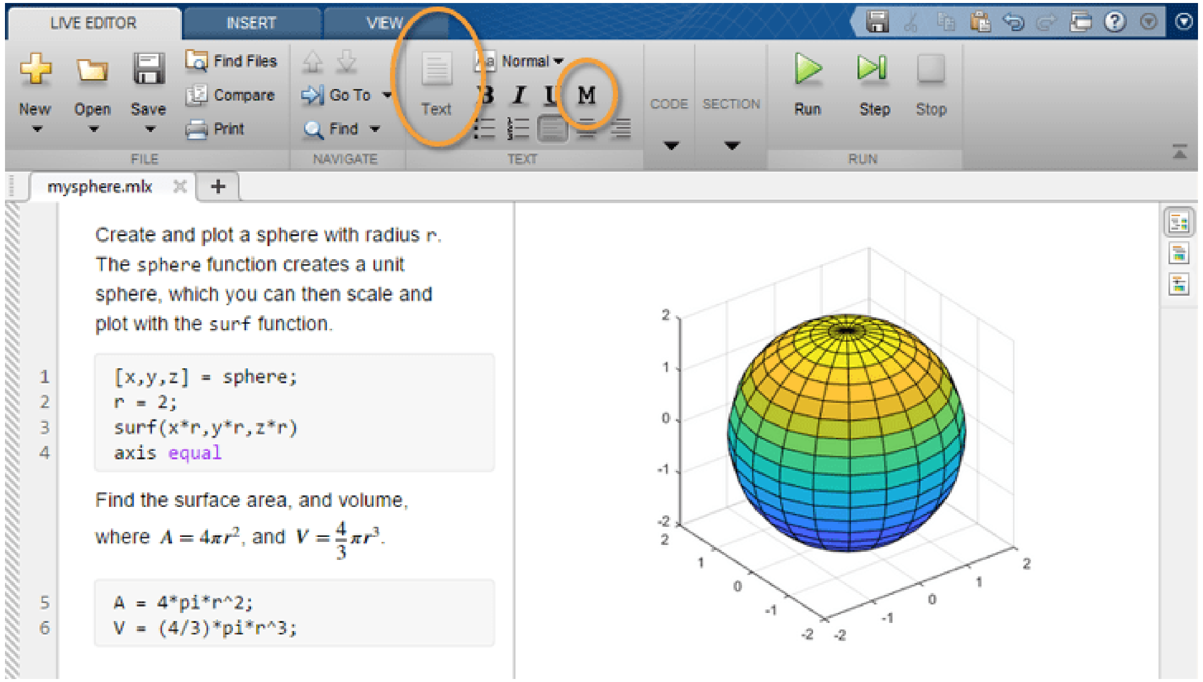
- Basic plotting using the *plot()* function
- Multiple plots on the same graph
- Multiple graphs on the same figure
- Using multiple figures
- Logarithmic plotting functions
- Manipulating figures
- Saving and utilizing figures
- Other specialized two-dimensional (2D) graphing functions
- Three-dimensional (3D) graphing functions

- Matlab scripts
- Matlab functions
- Using data files with Matlab
- Loading/saving Matlab files
- Relational operators
- Logical operations and expressions
- ***if*/*else*** and ***switch*** selection control structures
- ***for*/*while*** iterative control structures
- More commands and functionalities
- Comparison of Matlab with programming languages

Introduction to Matlab

- MATLAB (*MATrix LABoratory*), is a major computing environment developed by *MathWorks*, that is used by most engineering students, engineers and scientists for performing numerical calculations, developing algorithms and analyzing/visualizing data/results.
- In addition to computing, processing, plotting data, Matlab offers some basic programming capabilities to implement numerical methods and algorithms.



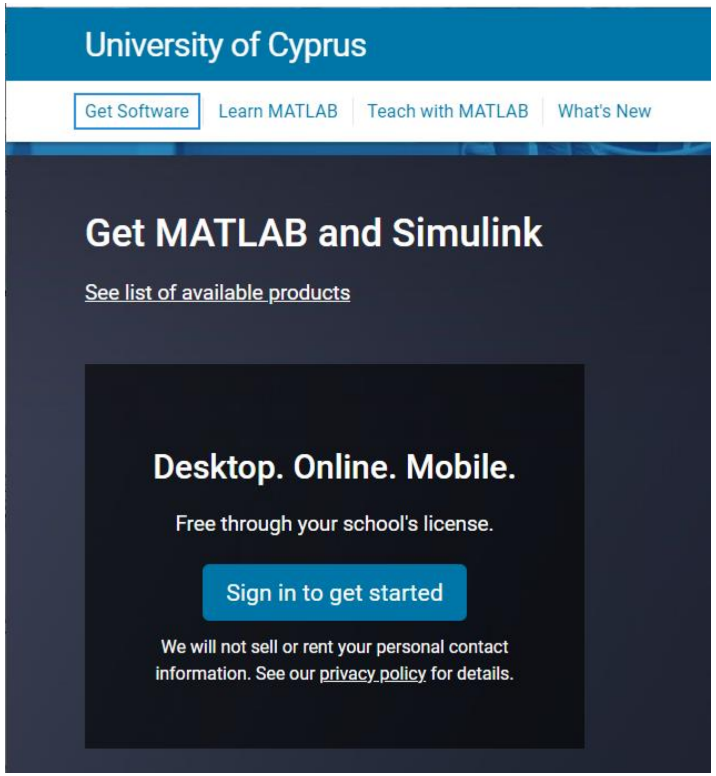


Matlab Capabilities

- Powerful calculator with sophisticated capabilities and options
- Advanced computing: Complex calculations, with emphasis on matrices
- Data analysis: Organize, process and analyze complex data sets
- Graphics/Visualization: Plot and visualize data
- Programming: Create scripts, functions and classes
- Building applications: Create desktop and web applications
- External Language Interfaces: Use MATLAB with other programming languages (Python, C/C++, Fortran, Java, etc)
- Hardware connectivity: Connect to hardware through Matlab (e.g. small shake table)
- etc.

Installing and using Matlab at the UCY

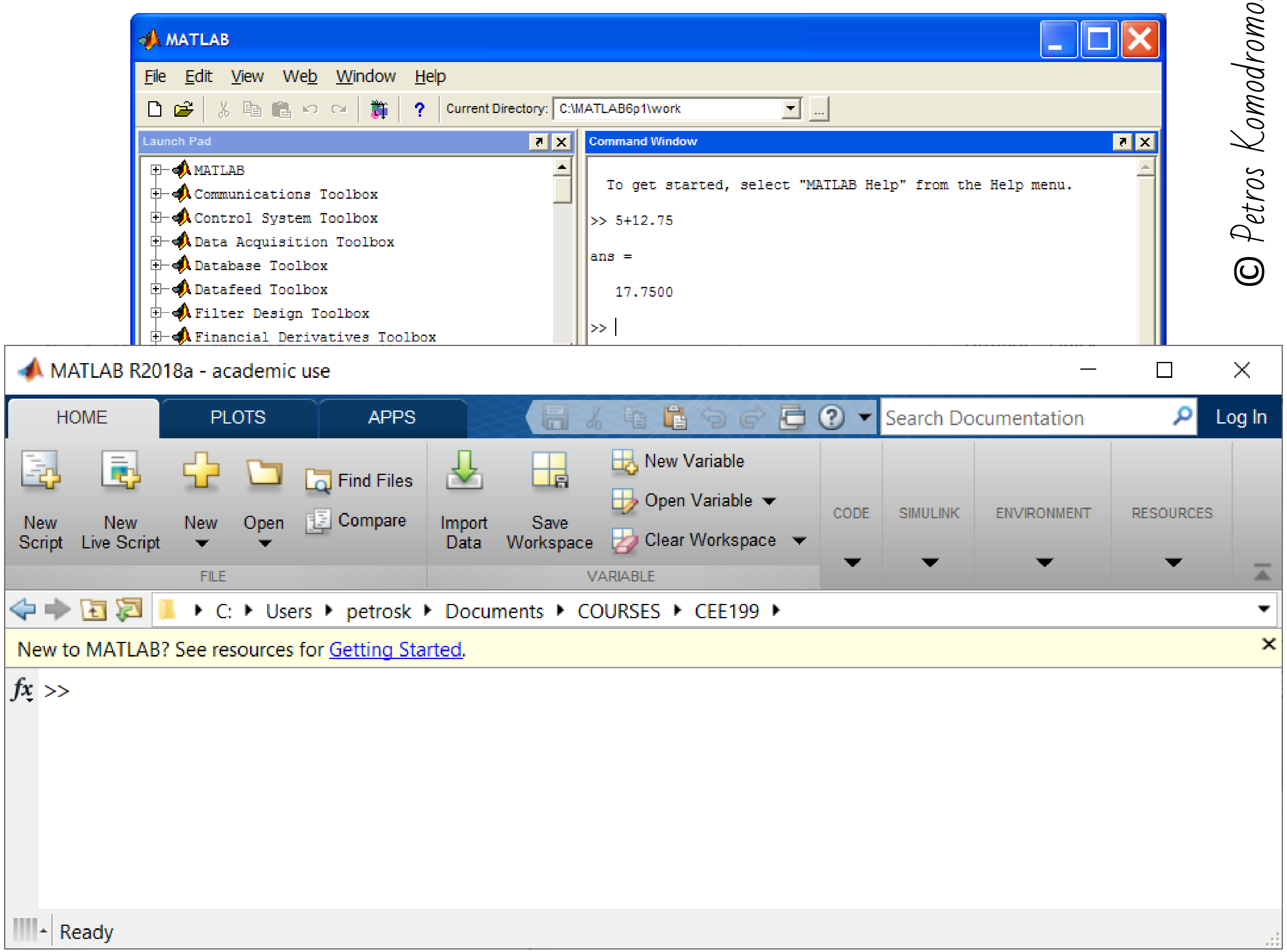
- MATLAB Access and Support for Everyone at the University of Cyprus



- Use the MATLAB Portal:
  https://www.mathworks.com/academia/tah-portal/university-of-cyprus-40702022.html
- Click here for a detailed installation video guide
  - Select Get Software
  - Sign in to get started with Matlab
  - Create a Mathworks account, with your UCY credentials, if you do not already have one
  - Install the software after downloading the proper version, corresponding to your operating system
  - Login with your Mathworks account and complete the installation, as shown on the aforementioned video guide

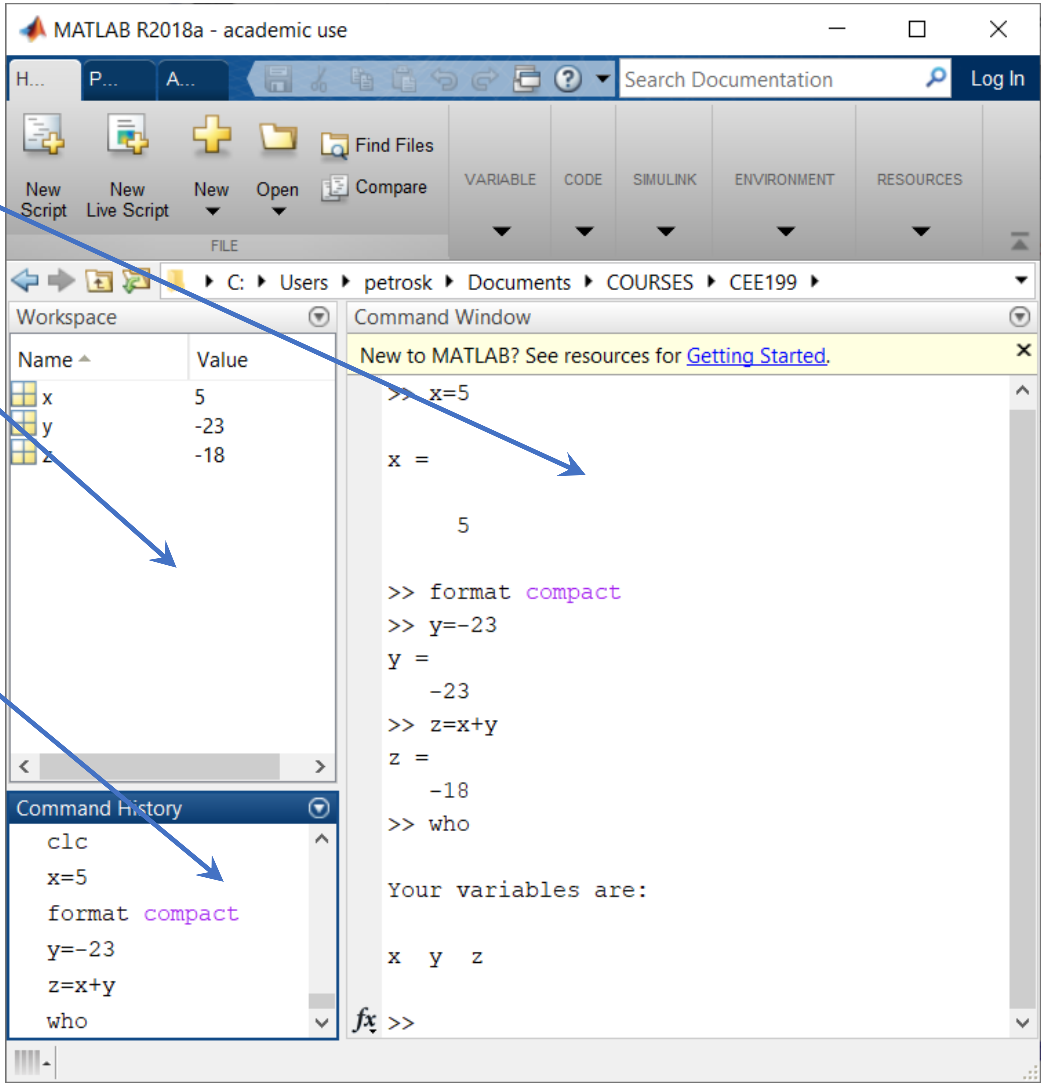
Getting started with Matlab and its environment

- The MATLAB's Integrated Development Environment (IDE) helps us to execute commands, develop code and functions, manage data and files, and view and plot results.
- Depending on the edition of Matlab, its desktop environment may look different.
- The desktop layout and the various preferences (which components to display, fonts style and size, etc.) can be easily modified to facilitate our current needs



Major windows of the MATLAB Environment

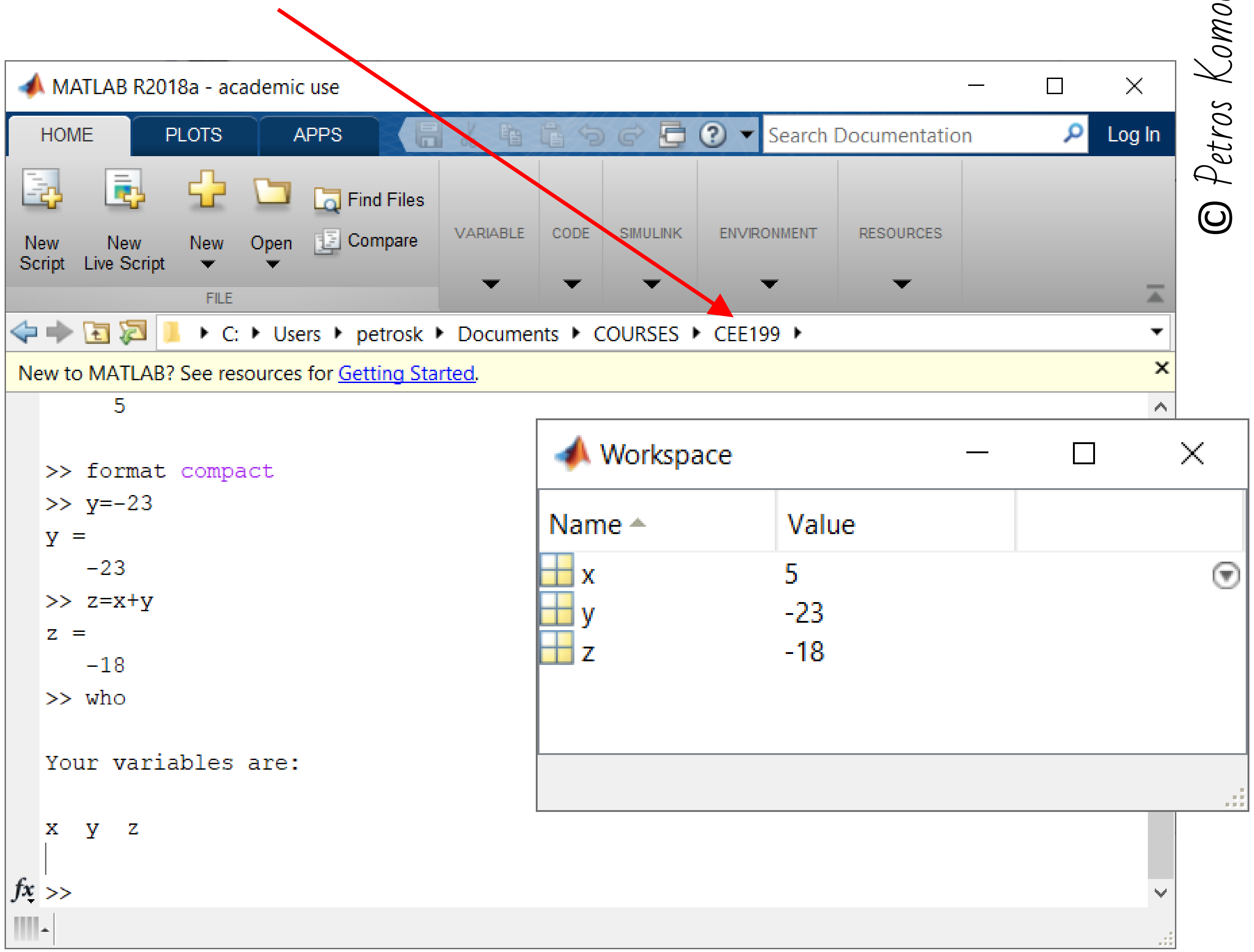
- ***Command Window***: where commands are entered and information and results are provided.
- ***Workspace Window***: provides information and (view, manipulate, save and clear) access to the variables of the current workspace.
- ***Command History Window***: lists previously executed commands in the Command Window.
- ***Current Directory Window:*** displays the contents (files and other directories/folders) of the active current directory.
- ***Figure Window:*** where graphical output is displayed.



MATLAB Environment

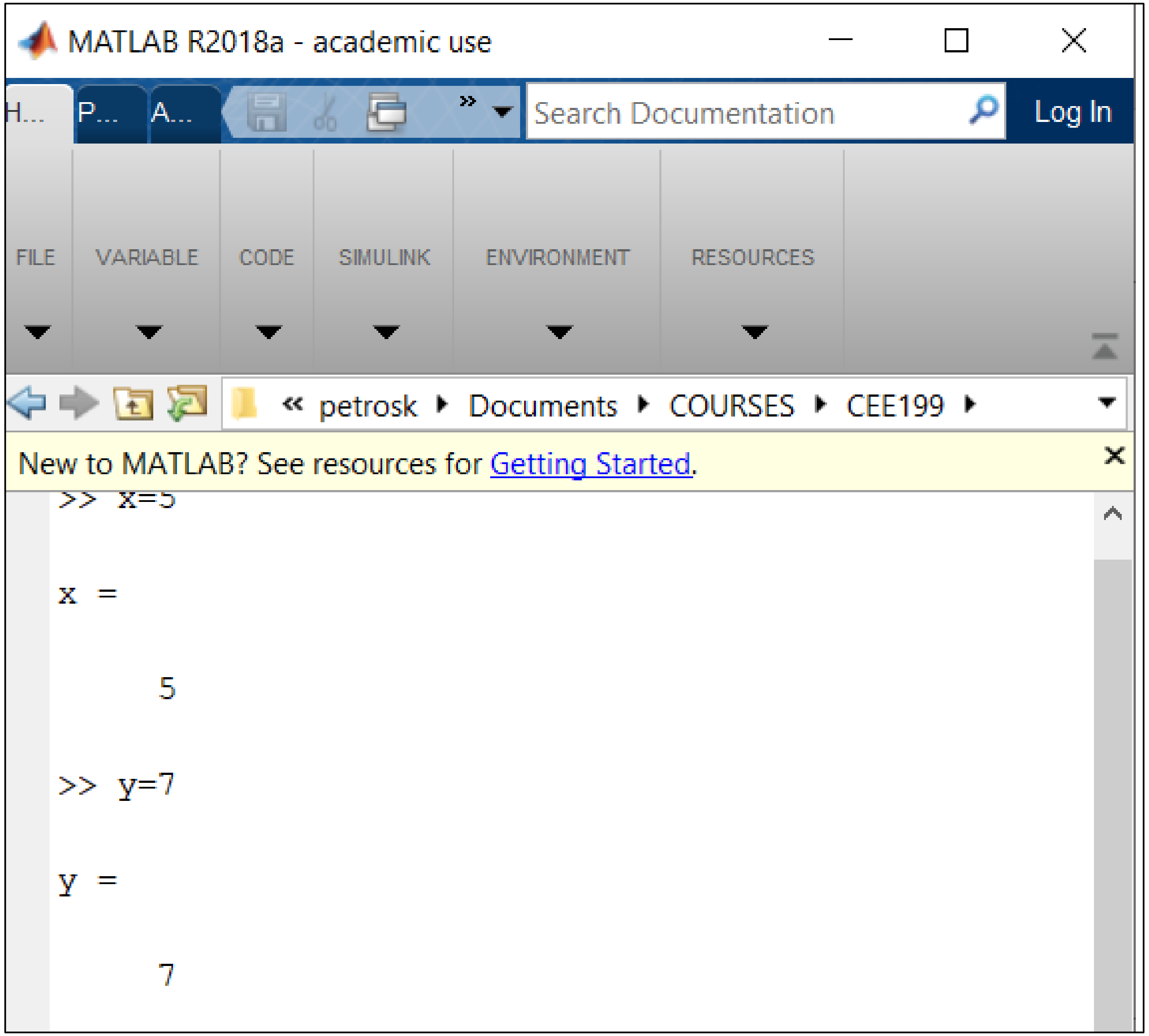
- The various windows can be closed, undocked or reappear (from the Layout options of the Environment or by executing the corresponding commands, such as *workspace*, *commandhistory*, etc.)
- The most essential window is the ***Command Window***, which has the command prompt, where commands can be entered and executed.
- It might be preferable to close all other windows in order to have more space to display files with Matlab commands and plots that Matlab creates.

- ***Current Directory:*** the active current directory/folder



Command window

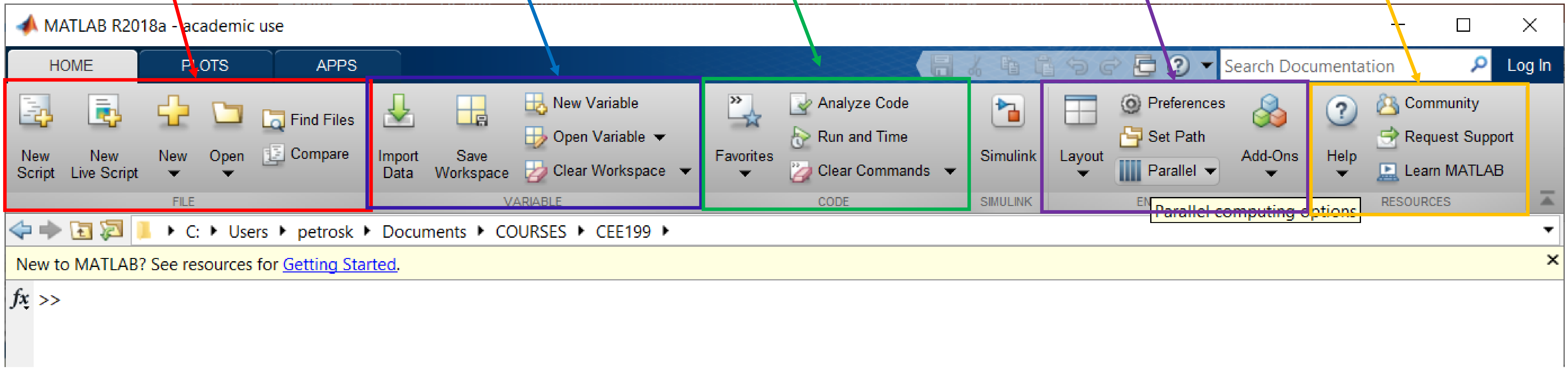
- The most important window, where:
  - Commands are given
  - Data are provided
  - Results are outputted
  - Software is controlled



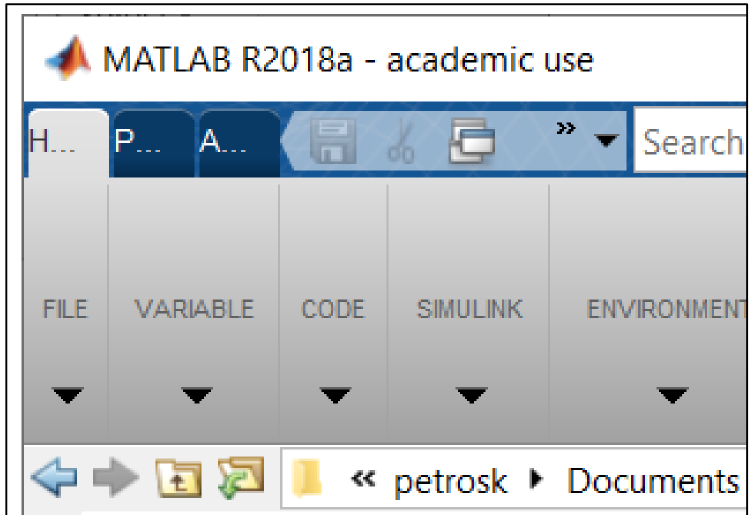
MATLAB Graphical User Interface (GUI)

- 3 tabs: **HOME** – PLOTS – APPS

- FILE section
- VARIABLE section
- CODE section
- ENVIRONMENT section
- RESOURCES section



Defining, manipulating and using simple variables



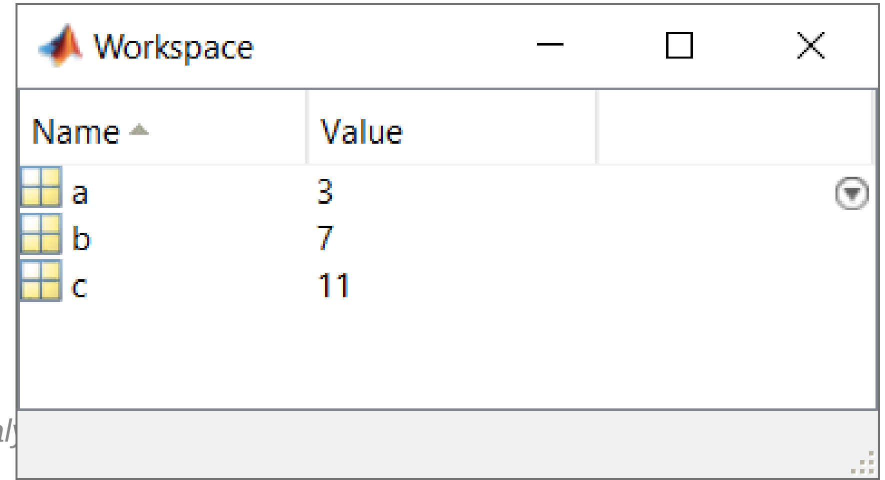

```
>> 3+5
ans =
     8
>> x=7
x =
     7
>> y=x+3;
>> y
y =
    10
>> z=y/x
z =
    1.4286
```

```
>> ans
ans =
     8
>> who

Your variables are:

ans  x    y    z

>> whos
  Name      Size            Bytes  Class

  ans       1x1                 8  double
  x         1x1                 8  double
  y         1x1                 8  double
  z         1x1                 8  double

>>
```

- All numerical values in Matlab are stored as doubles, actually as arrays (or matrices) of doubles.
- *ans* is a variable that stores the last computed result that is not assigned to another variable.
- *who*/*whos* are commands that show the names and details of the variables that are currently used.

```
>> clear
>> a=4;
>> b=7;
>> c=a+b;
>> a=3;
>> a
a =
     3
>> b
b =
     7
>> c
c =
    11
>> whos
  Name      Size            Bytes  Class

  a         1x1                 8  double
  b         1x1                 8  double
  c         1x1                 8  double
```

- In this case variables *a*, *b* and *c* are 1x1 (1 row by 1 column) arrays (i.e. matrices) of doubles, each with 1 element.
- *clear*: removes all variables from the current workspace, releasing the corresponding memory.
- The *Workspace window* provides information about all variables in the current workspace, as well as access to manipulate them.



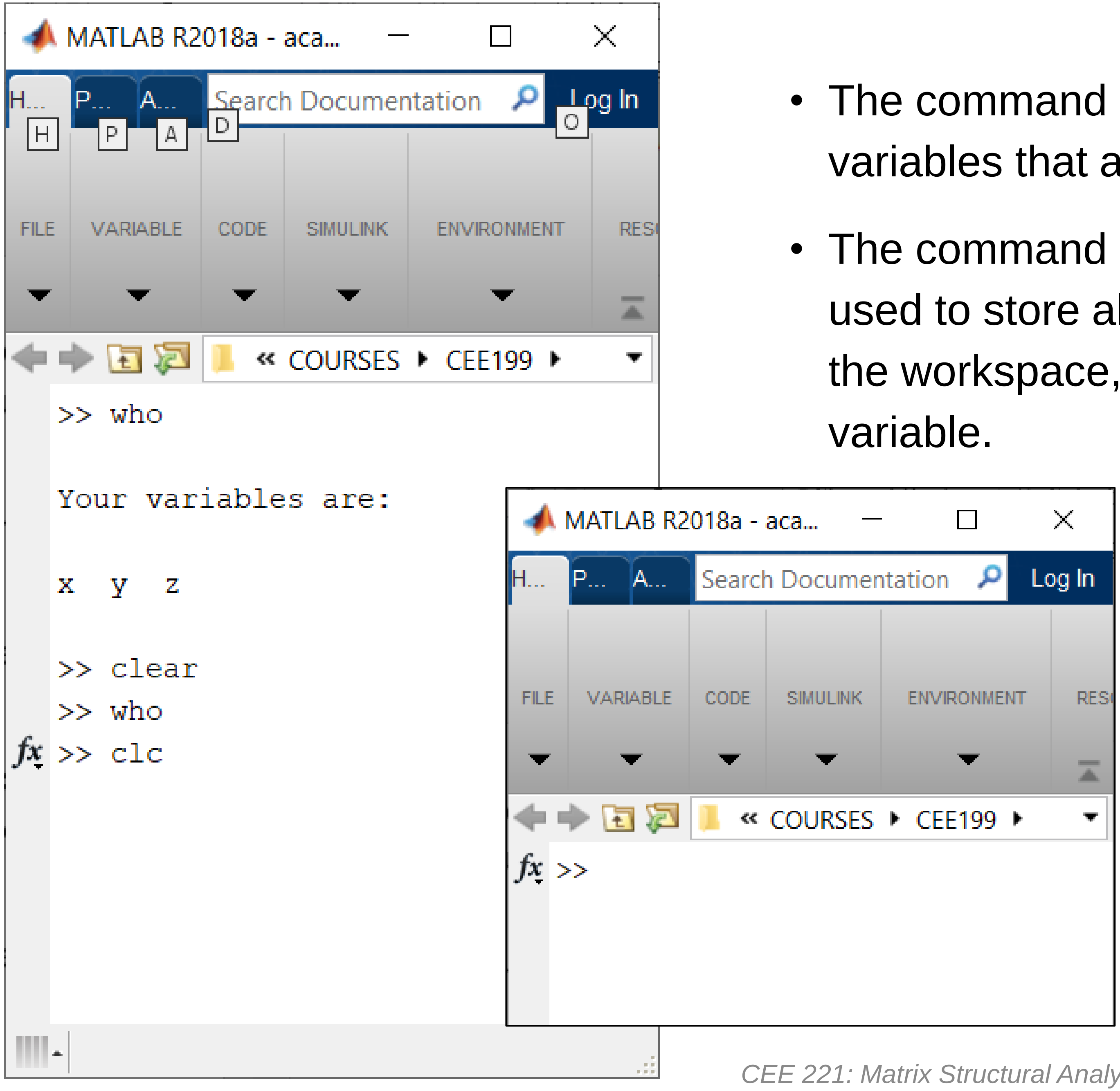
```
>> x=5;
>> y=7
y =
      7
>> z=x+y
z =
     12
>> y=3
y =
      3
>> z=x+y;
>> z
z =
      8
```

- The value 5 is assigned to the variable named *x*. The semicolon simply suppresses the output, while the command is executed.
- The value 7 is assigned to the variable named *y*, i.e. to the memory space in the workspace that is assigned to store the value of variable *y*.
- The values that are currently stored in variables *x* and *y* are retrieved, added and assigned to a variable named *z.*
- The value 3 is assigned to the variable named *y*, i.e. replacing the value that was previously stored there, while the semicolon simply suppresses the output.
- The values that are currently stored in variables *x* and *y* are retrieved, added and assigned to a variable named *z*.
- The value that is stored in the variable *z* is outputted.

Naming variables

- **Matlab is case sensitive**. Therefore, X and x are different variables.
- A valid variable name starts with a letter, followed by letters, digits, or underscores.
- The name of the variable should be meaningful. In case of a composite name, it is wise to capitalize the first letter of the second word, i.e. *xCoord*, *storyHeight*, *elasticityModulus*, etc.
- The name of a variable cannot have the same name as a Matlab keyword (e.g. *if*, *else*, *while*, etc.). For a complete list, run the *iskeyword* command.
- Avoid naming a variable with the same name of a function (e.g *sin()*, *sqrt()*, etc.) or a constant (*pi*).
- Check whether a name that you are considering to use has already been used with the *exist*. (which returns 0 if the name has not been used) or *which* (which locate functions, variables and files with the specific name) commands.

```
>> which sqrt
built-in (C:\Program Files\MATLAB\R2018a\toolbox\matlab\elfun\@double\sqrt)  % double method
>> which x
x is a variable.
```



- The command *who* provides the names of all variables that are currently stored in the workspace.
- The command *clear* releases the memory that is used to store all variables that are currently stored in the workspace, i.e. clearing the workspace from any variable.
- The command *clc* clears the command window and takes the cursor at the top left corner.

```
>> clear
>> x=1/2+2/7

x =

    0.7857

>> format compact
>> x
x =
    0.7857
>> format long
>> x
x =
   0.785714285714286
>> y=4+3*x-1.4*x+2/x;
>> y
y =
   7.802597402597403
>> format short
>> y
y =
    7.8026
>> 
```

- The expression 1/2+2/7 is executed (with double precision, i.e. accuracy of 15 significant digits) and its result is assigned to the variable named *x*. First, 1 is divided by 2, then 2 is divided by 7, and then their respective results are added.
- The command *format compact* can be used to set a more compact output in the command window (less white space), while the opposite can be set with the command *format loose*.
- After using the command *format long* all values and results are outputted with 15 significant digits, while with the command *format short* all values and results are outputted with 5 significant digits.
- In any case, the calculations are always performed with double precision, i.e. accuracy of 15 significant digits.
- The command *format* acts as a switch, the default settings are *loose* and *short*, which can be changed at any time, accordingly, to *compact* and *long*.

Operations and mixed expressions

```
()

^ '

* /

+ -

:

< > <= >= ~= ==

&&

||
```

```
>> 7-3*2
ans =
     1
>> (7-3)*2
ans =
     8
```

- In mixed expressions the order is precedence (shown for the most common operators on the left) is followed, executing first the operations with higher precedence and then the lower precedence operations, while when the operators are of the equal precedence the associativity (for most operators) is from left to right (the order of execution).

```
>> 5+3*2-4/2-9/3
ans =
     6
>> 5+(3*2)-(4/2)-(9/3)
ans =
     6
>> (((5+(3*2))-(4/2))-(9/3))
ans =
     6
```

➢ Firstly, the multiplications and divisions are performed, from left to right and then the additions and the divisions from left to right.

Order of operations in mixed expressions - Precedence

```
>> help precedence
 precedence Operator Precedence in MATLAB.

    MATLAB has the following precedence for the built-in operators when
    evaluating expressions (from highest to lowest):

       1. parentheses ()

       2. transpose (.'), power (.^), complex conjugate
          transpose ('), matrix power (^)

       3. power with unary minus (.^-), unary plus (.^+), or logical
          negation (.^~) as well as matrix power with unary minus (^-),
          unary plus (^+), or logical negation (^~)

       4. unary plus (+), unary minus (-), logical negation (~)

       5. multiplication (.*), right division (./), left
          division (.\), matrix multiplication (*), matrix right
          division (/), matrix left division (\)

       6. addition (+), subtraction (-)
```

- The command *help* displays help information in the command window for any topic, e.g. precedence

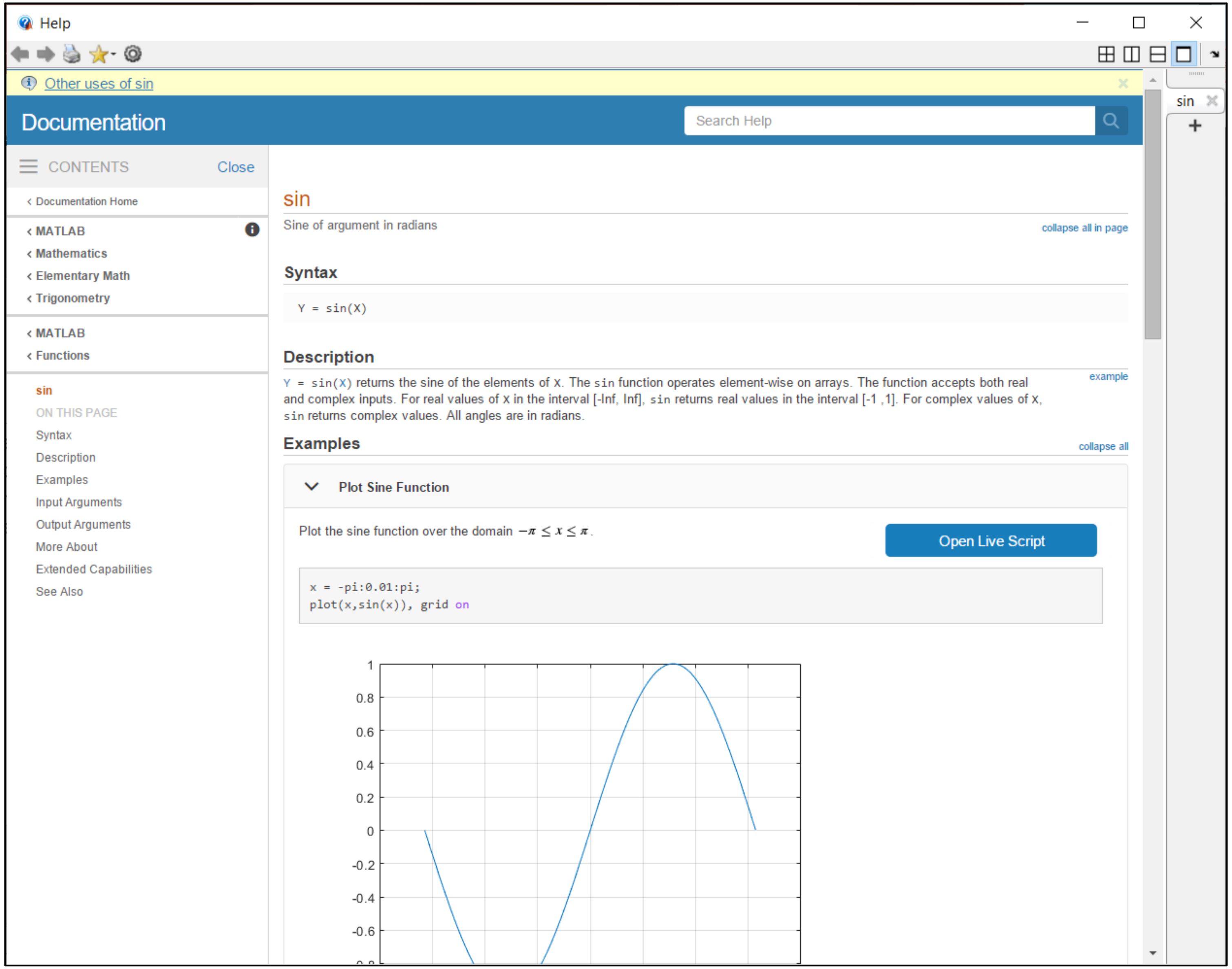
```
       7. colon operator (:)

       8. less than (<), less than or equal to (<=), greater than
          (>), greater than or equal to (>=), equal to (==), not
          equal to (~=)

       9. element-wise logical AND (&)

      10. element-wise logical OR (|)

      11. short-circuit logical AND (&&)

      12. short-circuit logical OR (||)

    See also syntax, arith.
```

```
>> help help
  help Display help text in Command Window.
     help, by itself, lists all primary help topics. Each primary topic
     corresponds to a folder name on the MATLAB search path.

     help NAME displays the help for the functionality specified by NAME,
     such as a function, operator symbol, method, class, or toolbox.
     NAME can include a partial path.

     Some classes require that you specify the package
     properties, and some methods require that you spec
     name. Separate the components of the name with per
     of the following forms:

         help CLASSNAME.NAME
         help PACKAGENAME.CLASSNAME
         help PACKAGENAME.CLASSNAME.NAME

     If NAME is the name of both a folder and a functio
     help for both the folder and the function. The hel
     is usually a list of the program files in that fol

     If NAME appears in multiple folders on the MATLAB
     information about the first instance of NAME found
```

```
>> help clc
 clc    Clear command window.
    clc clears the command window and homes the cursor.

    See also home.

    Reference page for clc

>> help clear
 clear  Clear variables and functions from memory.
    clear removes all variables from the workspace.
    clear VARIABLES does the same thing.
    clear GLOBAL removes all global variables.
    clear FUNCTIONS removes all compiled MATLAB and MEX-functions.
    Calling clear FUNCTIONS decreases code performance and is usually unnec
    For more information, see the clear Reference page.

    clear ALL removes all variables, globals, functions and MEX links.
    clear ALL at the command prompt also clears the base import list.
    Calling clear ALL decreases code performance and is usually unnecessary
    For more information, see the clear Reference page.
```

The command ***help*** can be used to get brief information about a command or function in Matlab, while the command ***doc*** can be used to retrieve more detail information in a separate window

```
>> help sin
 sin    Sine of argument in radians.
    sin(X) is the sine of the elements of X.

    See also asin, sind.

    Reference page for sin
    Other functions named sin

>> doc sin
>>
```



The command ***lookfor*** can also be used when you do not know the exact name of a function, as it provides all functions that might be related to the name you enter with the *lookfor* command.

The command ***which*** displays the full path (location) of a function or a file to be used.

C: ▸ Users ▸ petrosk ▸ Documents ▸ COURSES ▸ CEE199 ▸ Matlab

```
>> lookfor sqrt
realsqrt                       - Real square root.
sqrt                           - Square root.
sqrtm                          - Matrix square root.
sqrtm_tbt                      - Square root of 2x2 matrix from block diagonal of Schur form.
sqrtm_tri                      - Square root of quasi-upper triangular matrix.
cordicsqrt                     - CORDIC-based square root.
eml_fisqrt_helper              - Helper function for fixed-point square root
sqrt                           - Square root of fi object, computed using a bisection algorithm
>> which sqrt
built-in (C:\Program Files\MATLAB\R2021b\toolbox\matlab\elfun\@double\sqrt)  % double method
>> which Example1
C:\Users\petrosk\Documents\COURSES\CEE199\Matlab\Example1.m
```

```
>> x = 3^2
x =
     9
>> y = 4*3 ^2;
>> y
y =
    36
>> (4*3)^2
ans =
   144
>> 4*(3^2)
ans =
    36
```

^ is the power operator for scalars and has higher precedence than other operators for multiplication/division and addition/subtraction

It is always a good practice to write meaningful comments in your code to refresh your memory about your own commands and code or/and help others understand your code.

In Matlab anything after a single % character is considered a comment and it is not taken into account.

During code development and testing, the IDE allow you to instantly the commenting out of any code that does not need to run.

```
>> x=6+7*2;
>> x
x =
    20
>> % x=3
>> y=6 %sddsadd
y =
     6
>> x
x =
    20
```

The **colon operator** (*:*) is a very useful operator in Matlab, which can be used to create vectors, subscript arrays and specify for iterations.

```
>> help :
 :  Colon.
    J:K  is the same as [J, J+1, ..., J+m], where m = fix(K-J). In the
    case where both J and K are integers, this is simply [J, J+1, ..., K].
    This syntax returns an empty matrix if J > K.

    J:I:K  is the same as [J, J+I, ..., J+m*I], where m = fix((K-J)/I).
    This syntax returns an empty matrix when I == 0, I > 0 and J > K, or
    I < 0 and J < K.
```

```
>>  x = 1:6
x =
     1     2     3     4     5     6
>> y = 2:2:20
y =
     2     4     6     8    10    12    14    16    18    20
>> z = 5:0.5:8
z =
    5.0000    5.5000    6.0000    6.5000    7.0000    7.5000    8.0000
>> w = 10:-0.75:8
w =
   10.0000    9.2500    8.5000
```

Constants in Matlab

```
>> help pi
 pi      3.1415926535897....
    pi = 4*atan(1) = imag(log(-1)) = 3.1415926535897....

    Reference page for pi

>> pi
ans =
   3.141592653589793
>> help exp
 exp     Exponential.
    exp(X) is the exponential of the elements of X, e to the X.
    For complex Z=X+i*Y, exp(Z) = exp(X)*(COS(Y)+i*SIN(Y)).

    See also expm1, log, log10, expm, expint.

    Reference page for exp
    Other functions named exp

>> exp(1)
ans =
   2.718281828459046
```

- The *pi* is a constant with the value of π

- The function *exp()* computes and returns the exponential of the value that is sent as a parameter (argument) while calling the function.

Mathematical functions

```
>> sqrt(16)
ans =
     4
>> sin(0)
ans =
     0
>> sin(pi/2)
ans =
     1
>> sind(90)
ans =
     1
>> cosd(180)
ans =
    -1
>> cos(pi)
ans =
    -1
```

```
>> help sqrt
 sqrt   Square root.
    sqrt(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.
```

```
>> help sin
 sin    Sine of argument in radians.
    sin(X) is the sine of the elements of X.
```

```
>> help sind
 sind   Sine of argument in degrees.
    sind(X) is the sine of the elements of X, expressed in degrees.
    For integers n, sind(n*180) is exactly zero, whereas sin(n*pi)
    reflects the accuracy of the floating point value of pi.
```

```
>> help cosd
 cosd   Cosine of argument in degrees.
    cosd(X) is the cosine of the elements of X, expressed in degrees.
    For odd integers n, cosd(n*90) is exactly zero, whereas cos(n*pi/2)
    reflects the accuracy of the floating point value for pi.
```

Trigonometric functions

- The trigonometric functions ending with the letter ***d***, assume that the parameter is given in degrees or that the result should be given in degrees, otherwise radians are used.

sin	- Sine.
sind	- Sine of argument in degrees.
sinh	- Hyperbolic sine.
asin	- Inverse sine.
asind	- Inverse sine, result in degrees.
asinh	- Inverse hyperbolic sine.
cos	- Cosine.
cosd	- Cosine of argument in degrees.
cosh	- Hyperbolic cosine.
acos	- Inverse cosine.
acosd	- Inverse cosine, result in degrees.
acosh	- Inverse hyperbolic cosine.

tan	- Tangent.
tand	- Tangent of argument in degrees.
tanh	- Hyperbolic tangent.
atan	- Inverse tangent.
atand	- Inverse tangent, result in degrees.
atan2	- Four quadrant inverse tangent.
atanh	- Inverse hyperbolic tangent.
sec	- Secant.
secd	- Secant of argument in degrees.
sech	- Hyperbolic secant.
asec	- Inverse secant.
asecd	- Inverse secant, result in degrees.
asech	- Inverse hyperbolic secant.

Logarithmic functions

```
>> exp(1)
ans =
   2.718281828459046
>> log(ans)
ans =
     1
>> log(10)
ans =
   2.302585092994046
>> log10(10)
ans =
     1
>> log2(8)
ans =
     3
```

```
>> help exp
 exp    Exponential.
    exp(X) is the exponential of the elements of X, e to the X.
    For complex Z=X+i*Y, exp(Z) = exp(X)*(COS(Y)+i*SIN(Y)).
```

```
>> help log
 log    Natural logarithm.
    log(X) is the natural logarithm of the elements of X.
    Complex results are produced if X is not positive.
```

```
>> help log10
 log10  Common (base 10) logarithm.
    log10(X) is the base 10 logarithm of the elements of X.
    Complex results are produced if X is not positive.
```

```
>> help log2
 log2   Base 2 logarithm and dissect floating point number.
    Y = log2(X) is the base 2 logarithm of the elements of X.
```

Rounding functions

fix	- Round towards zero.
floor	- Round towards minus infinity.
ceil	- Round towards plus infinity.
round	- Round towards nearest integer.
mod	- Modulus (signed remainder after division).
rem	- Remainder after division.
sign	- Signum.

```
>> x = 1.333
x =
   1.3330

>> floor(x)
ans =
     1

>> ceil(x)
ans =
     2

>> round(x)
ans =
     1
```

```
x =
  -0.6667

>> floor(x)
ans =
    -1

>> ceil(x)
ans =
     0

>> round(x)
ans =
    -1
>>
```

Input/Output

```
>> input(' Enter a number: ')
 Enter a number: 4.7
ans =
   4.70000000000000
>> x=input(' x = ');
 x = 3.2
>> y=input(' y = ');
 y = 4.4
>> z=x+y
z =
   7.60000000000001
>> x
x =
   3.20000000000000
>> y
y =
   4.40000000000000
```

```
>> help input
 input  Prompt for user input.
    RESULT = input(PROMPT) displays the PROMPT string on the screen, waits
    for input from the keyboard, evaluates any expressions in the input,
    and returns the value in RESULT. To evaluate expressions, input accesses
    variables in the current workspace. If you press the return key without
    entering anything, input returns an empty matrix.

    STR = input(PROMPT,'s') returns the entered text as a MATLAB string,
    without evaluating expressions.
```

```
>> firstName = input('First name: ', 's')
First name: Petros
firstName =
    'Petros'
>> lastName = input('Last name: ', 's')
Last name: Komodromos
lastName =
    'Komodromos'
```

Input/**Output**

1. Type the name of a variable without a semicolon at the end of the command.
2. Call the *disp()* function, which displays the value of the given parameter
3. Use the *fprintf()* function, which enables the formatting of the output

```
>> help disp
 disp Display array.
    disp(X) displays array X without printing the array name or
    additional description information such as the size and class name.
    In all other ways it's the same as leaving the semicolon off an
    expression except that nothing is shown for empty arrays.

    If X is a string or character array, the text is displayed.
```

```
>> help fprintf
fprintf Write formatted data to text file.
   fprintf(FID, FORMAT, A, ...) applies the FORMAT to all elements of array A and
   any additional array arguments in column order, and writes the data to a text
   file.  FID is an integer file identifier.  Obtain FID from FOPEN, or set it to 1
   (for standard output, the screen) or 2 (standard error). fprintf uses the
   encoding scheme specified in the call to FOPEN.

   fprintf(FORMAT, A, ...) formats data and displays the results on the screen.

   COUNT = fprintf(...) returns the number of bytes that fprintf writes.

   FORMAT is a character vector that describes the format of the output fields, and
   can include combinations of the following:
```

Input/**Output**

1. Type the name of a variable without a semicolon at the end of the command.

```
>> x=5
x =
     5
>> y=7;
>> z=x+y
z =
    12
>> x
x =
     5
>> y
y =
     7
>> 77
ans =
    77
>> z
z =
    12
```

Input/**Output**

2. Call the *disp()* function, which displays the value of the given parameter or the value to be outputted.

```
>> a = 13
a =
    13
>> b = 4;
>> disp(a)
    13
>> disp(b)
     4
>> disp(a+b)
    17
>> disp(17)
    17
>> s='Testing'
s =
    'Testing'
>> disp(s)
Testing
```

```
>> help disp
 disp Display array.
    disp(X) displays array X without printing the array name or
    additional description information such as the size and class name.
    In all other ways it's the same as leaving the semicolon off an
    expression except that nothing is shown for empty arrays.

    If X is a string or character array, the text is displayed.
```

Input/**Output**

3. Use the *fprintf()* function, which enables the formatting of the output, according to the formatting (or control) string.

```
>> help fprintf
fprintf Write formatted data to text file.
   fprintf(FID, FORMAT, A, ...) applies the FORMAT to all elements of array A and
   any additional array arguments in column order, and writes the data to a text
   file.  FID is an integer file identifier.  Obtain FID from FOPEN, or set it to 1
   (for standard output, the screen) or 2 (standard error). fprintf uses the
   encoding scheme specified in the call to FOPEN.

   fprintf(FORMAT, A, ...) formats data and displays the results on the screen.

   COUNT = fprintf(...) returns the number of bytes that fprintf writes.

   FORMAT is a character vector that describes the format of the output fields, and
   can include combinations of the following:
```

```
FORMAT is a character vector that describes the format of the output fields, and
can include combinations of the following:

   * Conversion specifications, which include a % character, a
     conversion character (such as d, i, o, u, x, f, e, g, c, or s), and optional
     flags, width, and precision fields.  For more details, type "doc fprintf" at
     the command prompt.

   * Literal text to print.

   * Escape characters, including:
         \b     Backspace              ''   Single quotation mark
         \f     Form feed              %%   Percent character
         \n     New line               \\   Backslash
         \r     Carriage return        \xN  Hexadecimal number N
         \t     Horizontal tab         \N   Octal number N
     For most cases, \n is sufficient for a single line break. However, if you
     are creating a file for use with Microsoft Notepad, specify a combination of
     \r\n to move to a new line.
```

The values of the parameters that are given after the formatting (or control) string, which describes the desired formatting of the output fields, take the place of the placeholders (or conversion specifications, which begin with % and have a conversion character, a letter indicating the way in which they should be handled and outputted, and optional width, and precision fields.

A %f indicate floating point number, %5.3f indicates that a floating point number should be printed using totally up to 5 significant digits of which 3 should be after the decimal point, while a %d indicates that an integer formatting should be used.

A \n is an escape character, which indicates that a new line should be provided at that point.

```
>> x=5/3;
>> y=3.5*4;
>> z=x+y;
>> fprintf('Adding %f and %f equals %f', x, y, z)
Adding 1.666667 and 14.000000 equals 15.666667>>
>> fprintf('Adding %f and %f equals %f \n', x, y, z)
Adding 1.666667 and 14.000000 equals 15.666667
>> fprintf('Adding %5.3f and %6.4f equals %4.2f \n', x, y, z)
Adding 1.667 and 14.0000 equals 15.67
```

Defining, manipulating and using vectors and arrays

- Even the scalars that we have used so far, are essentially considered as 1x1 arrays in Matlab, i.e. arrays with 1 row and 1 element.
- An actual array is defined using the square brackets [ ].
  - Whitespace or comma separate elements in the same row, indicating another column.
  - A newline or a semicolon separates elements in different rows, indicating another row.

```
>> x = [ 4 -5 6 12 ]
x =
     4    -5     6    12
>> y = [ 44,5,7]
y =
    44     5     7
>> v = [ 3 ; 8 ; 11]
v =
     3
     8
    11
```

- ➢ *x* is a (1x4) row vector
- ➢ y is a (1x3) row vector
- ➢ *v* is a (3x1) column vector
- ➢ *w* is a (4x1) column vector
- ➢ *a* is a 2x3 matrix

```
>> w = [ 8
-4
5
2]
w =
     8
    -4
     5
     2
>> a = [ 4 2 6 ; 5 4 3]
a =
     4     2     6
     5     4     3
```

- A vector is a matrix with one row (row vector) or one column (column vector)
- The elements of vectors and matrices are indexed, with the index starting with 1 at the first element and ending with the number of rows or columns as the index of the last element of the corresponding row or column, respectively, within parentheses ().
- In order to access (or retrieve) an element of a vector one index is sufficient, while to access an element of a matrix two indices are required, the first refers to the number of the row and the second to the number of the column of the specific element.

```
>> a = [ 5 6 2 4]
a =
     5     6     2     4
>> a
a =
     5     6     2     4
>> a(2)
ans =
     6
>> a(1)
ans =
     5
>> a(4)
ans =
     4
```

```
>> b = [ 7 ; 4 ; 2]
b =
     7
     4
     2
>> b(1)
ans =
     7
>> b(3)
ans =
     2
>> b(2,1)
ans =
     4
```

```
>>  m = [ 3 17 23; 4 5 -3]
m =
     3    17    23
     4     5    -3
>> m(1,1)
ans =
     3
>> m(2,1)
ans =
     4
>> m(2,3)
ans =
    -3
>> m(3,2)
Index in position 1 exceeds array bounds (must not exceed 2).
```

- Function *size()* provides the size of an array, returning 2 numbers, the number of rows and the number of columns of the array that is used as a parameter.

```
>> a = [ 3 1 8 9 ; 4 2 -8 7
5 6 3 12 ]
a =
     3     1     8     9
     4     2    -8     7
     5     6     3    12
>> size(a)
ans =
     3     4
>> sz = size(a)
sz =
     3     4
>> nRows = sz(1)
nRows =
     3
>> nColumnss = sz(2)
nColumnss =
     4
>> workspace
```

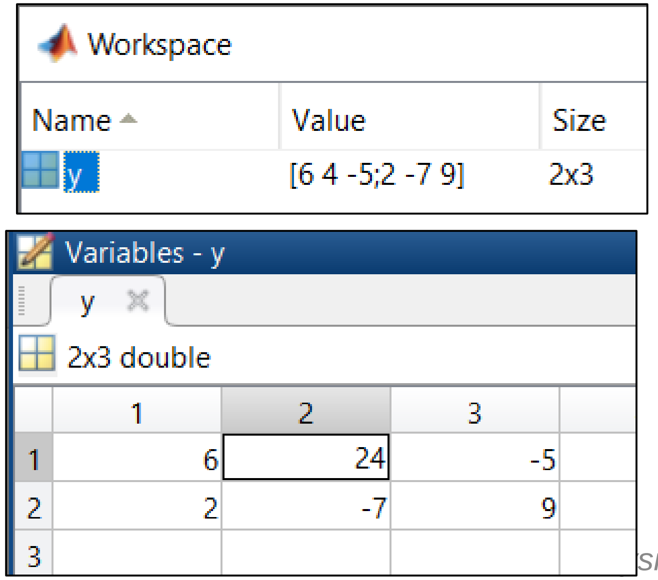
- The Workspace Window provides information for the arrays stored in the current workspace, such as the values and size of the array. In addition, by right-clicking on empty columns of the Workspace Window more properties and statistical information can be provided (such as the mean value of all elements, the minimum and maximum elements, etc.)

Workspace

Name	Value	Size	Mean	Min	Max
a	*3x4 double*	3x4	4.3333	-8	12
ans	[3 4]	1x2	3.5000	3	4
nColumnss	4	1x1	4	4	4
nRows	3	1x1	3	3	3
sz	[3 4]	1x2	3.5000	3	4

- Function *length()* provides the size (number of elements) of a vector.
- When the function *length()* is used with an array as a parameter, it returns the largest dimension of the array, i.e. it is equivalent to using the functions *max(size())* combined with an array as a parameter.
- The Workspace Window, besides providing information, enables the editing of the values of variables (scalars, vectors, arrays, etc.), by double-clicking on the variable.

```
>> y = [ 6 4 -5 ; 2 -7 9];
>> workspace
>> y

y =

     6    24    -5
     2    -7     9
```




```
>> x = [ 4 2 7 -3 5]
x =
     4     2     7    -3     5
>> n = length(x);
>> n
n =
     5
>> a = [ 7 3 -2 ; 5 -4 9];
>> a
a =
     7     3    -2
     5    -4     9
>> size(a)
ans =
     2     3
>> length(a)
ans =
     3
>> max(size(a))
ans =
     3
```

Functions defining/computing special arrays

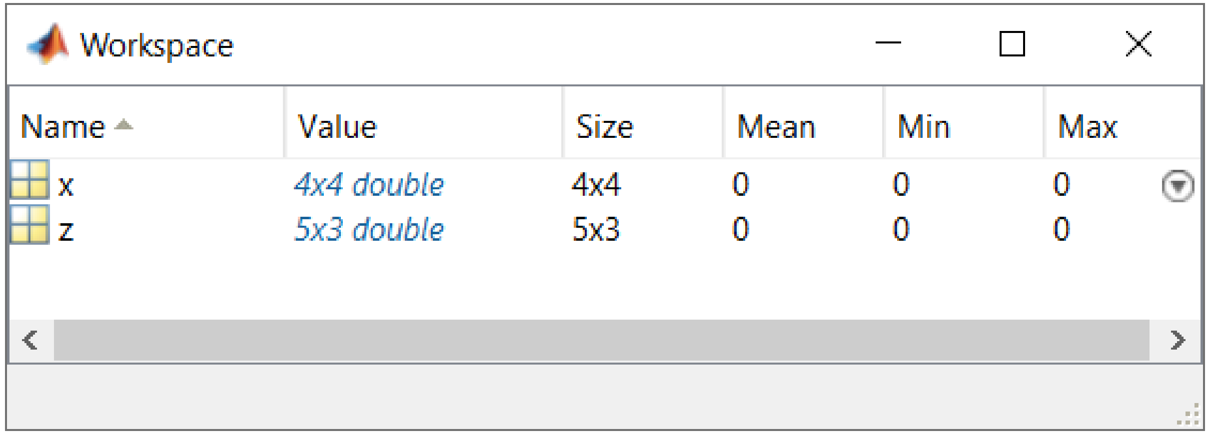
- Function *ones()* returns an array of ones.
  - *ones(n)* returns an n by n (size *nxn*) square array of ones
  - *ones(m,n)* returns an m by n (*mxn*) array of ones

Workspace

Name	Value	Size	Mean	Min	Max
a	[1 1 1;1 1 1;1 1 1]	3x3	1	1	1
b	[1 1 1 1;1 1 1 1]	2x4	1	1	1
c	3x5 double	3x5	7.5000	7.5000	7.5000

```
>> a = ones(3)

a =

     1     1     1
     1     1     1
     1     1     1

>> b=ones(2,4)

b =

     1     1     1     1
     1     1     1     1

>> c = 7.5 * ones(3,5)

c =

    7.5000    7.5000    7.5000    7.5000    7.5000
    7.5000    7.5000    7.5000    7.5000    7.5000
    7.5000    7.5000    7.5000    7.5000    7.5000
```

- Function *zeros()* returns an array of zeros.
  - ➢ *zeros(n)* returns an n by n (size *nxn*) square array of zeros
  - ➢ *zeros(m,n)* returns an m by n (*mxn*) array of zeros



Name	Value	Size	Mean	Min	Max
x	4x4 double	4x4	0	0	0
z	5x3 double	5x3	0	0	0

```
>> clear
>> x = zeros(4)

x =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0

>> z = zeros(5,3)

z =

     0     0     0
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

- Function *eye(n)* returns an identity array of size *n*.

Name	Value	Size	Mean	Min	Max
ans	[1 0;0 1]	2x2	0.5000	0	1
i3	[1 0 0;0 1 0;0 0 1]	3x3	0.3333	0	1

```
>> clear
>> i3 = eye(3)

i3 =

     1     0     0
     0     1     0
     0     0     1

>> eye(2)

ans =

     1     0
     0     1

>> ans

ans =

     1     0
     0     1
```

- Function *diag(V)* returns a diagonal matrix of size equal to the size of the vector V, putting the elements of V in the diagonal.

```
>> x = [ 4 6 2 -5 6]
x =
     4     6     2    -5     6
>> diag(x)
ans =
     4     0     0     0     0
     0     6     0     0     0
     0     0     2     0     0
     0     0     0    -5     0
     0     0     0     0     6
>>
```

- Function *rand()* returns an array of uniformly distributed random numbers from the standard uniform distribution on the open interval (0,1).
  - *rand(n)* returns an n by n (size *nxn*) square array of uniformly distributed random numbers
  - *rand(m,n)* returns an m by n (*mxn*) array of uniformly distributed random numbers

Workspace

Name	Value	Size	Mean	Min	Max
ans	[0.3571 9.3399 7....	2x4	6.3077	0.3571	9.3399
r1	[0.8147 0.9134 0....	3x3	0.5860	0.0975	0.9575
r2	*4x3 double*	4x3	9.0557	7.4257	9.9118

```
>> clear
>> r1 = rand(3)

r1 =

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575

>> r2 = 7 + 3 * rand(4,3)

r2 =

    9.8947    8.4561    9.7472
    7.4728    9.4008    9.3766
    9.9118    7.4257    9.8785
    9.8715    8.2653    8.9672

>> 10*rand(2,4)

ans =

    0.3571    9.3399    7.5774    3.9223
    8.4913    6.7874    7.4313    6.5548
```

- The **transpose of a matrix**, A^T, is derived by interchanging the columns of the array with the rows of the array and vice versa.

$$\underline{A} = \begin{bmatrix} 2 & 11 & 6 \\ 3 & 4 & 9 \end{bmatrix} \Rightarrow \underline{A}^T = \begin{bmatrix} 2 & 3 \\ 11 & 4 \\ 6 & 9 \end{bmatrix}$$

- In Matlab, the **transpose of a matrix** A is computed by using a single quote (') just after the name of the array (i.e. A'), or using the *transpose()* function.

```
>> A = [ 2 11 6 ; 3 4 9 ]
A =
     2    11     6
     3     4     9
>> A'
ans =
     2     3
    11     4
     6     9
>> transpose(A)
ans =
     2     3
    11     4
     6     9
```

```
>> x = rand(4,3)

x =

    0.1712    0.0462    0.3171
    0.7060    0.0971    0.9502
    0.0318    0.8235    0.0344
    0.2769    0.6948    0.4387

>> y = x';
>> y

y =

    0.1712    0.7060    0.0318    0.2769
    0.0462    0.0971    0.8235    0.6948
    0.3171    0.9502    0.0344    0.4387
```

- The **determinant of a matrix** A, |A|, is computed in Matlab using the function *det(A)*.
- The **inverse of a matrix** A, A^{-1}, is computed in Matlab using the function *inv(A)*.

$$\underline{A}^{-1} = \frac{1}{\det(\underline{A})} \cdot \mathrm{adj}(\underline{A}) \qquad \underline{A} \cdot \underline{A}^{-1} = \underline{A}^{-1} \cdot \underline{A} = \underline{I}$$

```
>> B= [ 2 -2 -3 ; 6 7 1 ; 8,1,5]
B =
     2    -2    -3
     6     7     1
     8     1     5
>> B*inv(B)
ans =
    1.0000   -0.0000    0.0000
    0.0000    1.0000   -0.0000
         0    0.0000    1.0000
```

```
>> A = [ 2,5 ; -6,10]
A =
     2     5
    -6    10
>> det(A)
ans =
    50
>> inv(A)
ans =
    0.2000   -0.1000
    0.1200    0.0400
>> B= [ 2 -2 -3 ; 6 7 1 ; 8,1,5]
B =
     2    -2    -3
     6     7     1
     8     1     5
>> det(B)
ans =
   262
>> inv(B)
ans =
    0.1298    0.0267    0.0725
   -0.0840    0.1298   -0.0763
   -0.1908   -0.0687    0.0992
```

Operations on matrices

- **Matrix (array) addition and subtraction**: two or more arrays can be added or subtracted as long as they have the same dimensions, since each of the corresponding elements of the arrays are added or subtracted, respectively.

```
>> a = [  4 5 3 ; 2 6 7]
a =
     4     5     3
     2     6     7
>> b = [ 1 2 3; 10 200 1000]
b =
           1           2           3
          10         200        1000
>> c=a+b
c =
           5           7           6
          12         206        1007
>>
```

```
>> a = [  4 5 3 ; 2 6 7];
>> b = [ 1 2 3; 10 200 1000];
>> c = [ 1 1 1; 2 2 2];
>> d=a-b
d =
     3     3     0
    -8  -194  -993
>> e=a-b+c
e =
     4     4     1
    -6  -192  -991
```

- **Multiplication or division of a matrix with a scalar**:
  - ➢ Each of the elements of the matrix is multiplied or divided, respectively, by the scalar (number).

$$\underline{A} = \begin{bmatrix} 2 & 5 \\ -6 & 10 \end{bmatrix}, \quad \underline{B} = \begin{bmatrix} 0 & 1 \\ 3 & 8 \\ 7 & 3 \end{bmatrix}$$

$$\Rightarrow \quad \underline{A} \cdot 3 = \begin{bmatrix} 6 & 15 \\ -18 & 30 \end{bmatrix}, \quad \underline{B} \cdot (-4) = \begin{bmatrix} 0 & -4 \\ -12 & -32 \\ -28 & -12 \end{bmatrix}$$

```
>> A = [ 2 5 ; -6 10 ]
A =
     2     5
    -6    10
>> B = [ 0 1 ; 3 8 ; 7 3]
B =
     0     1
     3     8
     7     3
>> A*3
ans =
     6    15
   -18    30
>> B*-4
ans =
     0    -4
   -12   -32
   -28   -12
>> A/2
ans =
    1.0000    2.5000
   -3.0000    5.0000
```

- **Multiplication of matrices**: two matrices, let's say ***A*** and ***B*** can be multiplied (AxB), ***following the rules of linear algebra***, as long as their inner dimensions agree, i.e. the number of the columns of the first matrix (***A***) should be equal to the number of the rows of the second array (***B***).
- The resulting array (***C***) should have rows as many as the number of rows of the first array (***A***) and columns as many as the number of columns of the second matrix (***B***)

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j).$$

$$\underline{A} = \begin{bmatrix} 4 & -8 \\ 6 & 11 \end{bmatrix}, \quad \underline{B} = \begin{bmatrix} 1 & 7 \\ 0 & 3 \end{bmatrix}$$

$$\Rightarrow \underline{A} \cdot \underline{B} = \begin{bmatrix} 4 \cdot 1 + (-8) \cdot 0 & 4 \cdot 7 + (-8) \cdot 3 \\ 6 \cdot 1 + 11 \cdot 0 & 6 \cdot 7 + 11 \cdot 3 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 6 & 75 \end{bmatrix}$$

$$\Rightarrow \underline{B} \cdot \underline{A} = \begin{bmatrix} 1 \cdot 4 + 7 \cdot 6 & 1 \cdot (-8) + 7 \cdot 11 \\ 0 \cdot 4 + 3 \cdot 6 & 0 \cdot (-8) + 3 \cdot 11 \end{bmatrix} = \begin{bmatrix} 46 & 69 \\ 18 & 33 \end{bmatrix}$$

- In Matlab the * operator is used for matrix multiplication

```
>> A = [ 4 -8
         6 11 ]
A =
     4    -8
     6    11
>> B = [ 1 7 ; 0 3 ]
B =
     1     7
     0     3
>> A*B
ans =
     4     4
     6    75
>> B*A
ans =
    46    69
    18    33
```

- In Matlab the * operator is used for matrix multiplication

```
>> a = rand(3,2)
a =
    0.9593    0.1493
    0.5472    0.2575
    0.1386    0.8407
>> b=rand(3,2)
b =
    0.2543    0.9293
    0.8143    0.3500
    0.2435    0.1966
>> a*b
Error using  *
Incorrect dimensions for matrix
multiplication. Check that the
number of columns in the first
matrix matches the number of
rows in the second matrix. To
perform elementwise
multiplication, use '.*'.
```

- **Element-wise multiplication (.*), division (./) and power (.^) of the elements of an array with the elements of another array**
    - In the element-wise multiplication (A.*B), division (A./B) and power (A.^B), each of the elements of matrix A is multiplied, divided or raised to the power, respectively, with the corresponding element of the array B.
    - Obviously, the two arrays should be of exactly the same size.

```
>> a = [ 3 5 2 ; 1 4 2]
a =
     3     5     2
     1     4     2
>> b = [ 2 1 3; 4 2 3]
b =
     2     1     3
     4     2     3
>> c = a .* b
c =
     6     5     6
     4     8     6
>> d = a ./b
d =
    1.5000    5.0000    0.6667
    0.2500    2.0000    0.6667
>> e = a .^ b
e =
     9     5     8
     1    16     8
```

- Arrays can be **concatenated** to create another array, as long as their dimensions are compatible in deriving a rectangular array with equal number of columns at each row.

```
>> a = [ 3 5 2]
a =
     3     5     2
>> b = [ 4 7 1]
b =
     4     7     1
>> c = [ a ; b]
c =
     3     5     2
     4     7     1
```

```
>> x = [ 4 2 8 ; 3 7 4]
x =
     4     2     8
     3     7     4
>> y = [ 6 9 1 ; 2 5 6]
y =
     6     9     1
     2     5     6
>> z = 3*ones(2,3)
z =
     3     3     3
     3     3     3
>> w = [x y z]
w =
     4     2     8     6     9     1     3     3     3
     3     7     4     2     5     6     3     3     3
>> w = [x ; y ; z]
w =
     4     2     8
     3     7     4
     6     9     1
     2     5     6
     3     3     3
     3     3     3
```

- A part of an array (submatrix) can be selected, giving a range of indices for the rows and columns, using the colon operator (**:**).
- A colon operator without a start and end indicates all rows or all columns.

```
>> x
x =
     4     2     8
     3     7     4
     6     9     1
     2     5     6
     3     3     3
     3     3     3
>> x(2,2:3)
ans =
     7     4
>> x(2,:)
ans =
     3     7     4
>> x(4,:)
ans =
     2     5     6
```

```
>> a = [ 5 3 7 3 7 ; 6 8 3 1 12 ; 5 67 3 2 1; 9 0 21 4 23]
a =
     5     3     7     3     7
     6     8     3     1    12
     5    67     3     2     1
     9     0    21     4    23
>> b = a(2:3,2:5)
b =
     8     3     1    12
    67     3     2     1
>> c = a(3,:)
c =
     5    67     3     2     1
>> d = [ a(3:4,1:3), b ]
d =
     5    67     3     8     3     1    12
     9     0    21    67     3     2     1
```

Solving systems of algebraic equations

- A system of *N* linear algebraic equations ***A . X = B*** can be easily solved with Matlab to find the unknown ***X***, assuming that the matrix of coefficients ***A*** is not singular.
- The matrix of coefficients ***A*** is not singular, i.e. it can be inverted if its rank is equal to its size. The rank of a matrix can be computed using the function ***rank()*** of Matlab and compared with the dimension of the matrix, using either the size() or the length() function.

Example: 3x + 2y = 8

2x - 4y = 5

```
>> A = [ 3 2 ; 2 -4]
A =
     3     2
     2    -4
>> rank(A)
ans =
     2
>> length(A)
ans =
     2
```

```
>> A = [ 3 2 ; 2 -4]
A =
     3     2
     2    -4
>> B = [8 5]'
B =
     8
     5
>> x = inv(A)*B
x =
    2.6250
    0.0625
```

```
>> A = [ 3 2 ; 2 -4]
A =
     3     2
     2    -4
>> B = [8 5]'
B =
     8
     5
>> x = A\B
x =
    2.6250
    0.0625
```

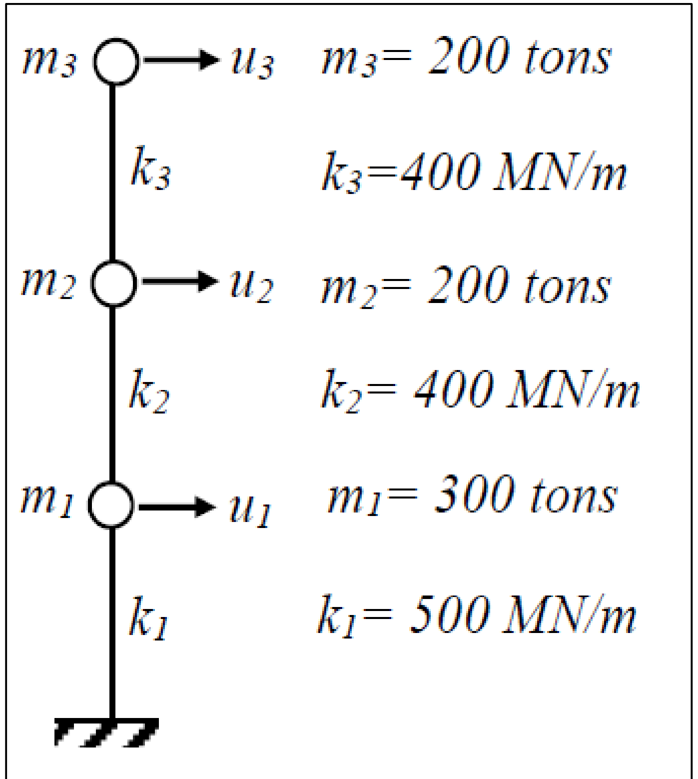
```
>> clear
>> a = rand(5)
a =
    0.2511    0.5853    0.7537    0.5308    0.4694
    0.6160    0.5497    0.3804    0.7792    0.0119
    0.4733    0.9172    0.5678    0.9340    0.3371
    0.3517    0.2858    0.0759    0.1299    0.1622
    0.8308    0.7572    0.0540    0.5688    0.7943
>> b = rand(5,1)
b =
    0.3112
    0.5285
    0.1656
    0.6020
    0.2630
>> x = inv(a)*b
x =
    1.8932
    1.1830
    1.2205
   -2.2298
   -1.2630
>> rank(a)
ans =
     5
```

```
>> a = rand(5)
a =
    0.6541    0.2290    0.9961    0.0046    0.3998
    0.6892    0.9133    0.0782    0.7749    0.2599
    0.7482    0.1524    0.4427    0.8173    0.8001
    0.4505    0.8258    0.1067    0.8687    0.4314
    0.0838    0.5383    0.9619    0.0844    0.9106
>> b = rand(5,1)
b =
    0.1818
    0.2638
    0.1455
    0.1361
    0.8693
>> rank(a)
ans =
     5
>> x = a \ b
x =
    0.7056
    0.8335
   -1.1960
   -1.7604
    1.8234
>> x = inv(a)*b
x =
    0.7056
    0.8335
   -1.1960
   -1.7604
    1.8234
```

- Solving the system of linear algebraic equation using Gauss elimination.
- Solving the system of linear algebraic equation using the inverse of the matrix of coefficients.

Computing eigenevalues & eigenvectors

- During structural dynamics courses, you will need to be able to compute the eigenmodes (eigenvectors) $\boldsymbol{\varphi_i}$ and eigenfrequencies (eigenvalues) $\boldsymbol{\omega_i}$ of a structural system. The eigenmodes and eigenfrequencies of structure can be computed from its stiffness matrix, ***K***, and its mass matrix, ***M***, using the function ***eig(K,M)*** of Matlab, which takes as arguments the stiffness and mass matrices of the structure.



$$\underline{K} = \begin{bmatrix} k_1 + k_2 & -k_2 & 0 \\ -k_2 & k_2 + k_3 & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix} = \begin{bmatrix} 900 & -400 & 0 \\ -400 & 800 & -400 \\ 0 & -400 & 400 \end{bmatrix} \text{MN/m}$$

$$\underline{M} = \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix} = \begin{bmatrix} 300 & 0 & 0 \\ 0 & 200 & 0 \\ 0 & 0 & 200 \end{bmatrix} \text{tons}$$

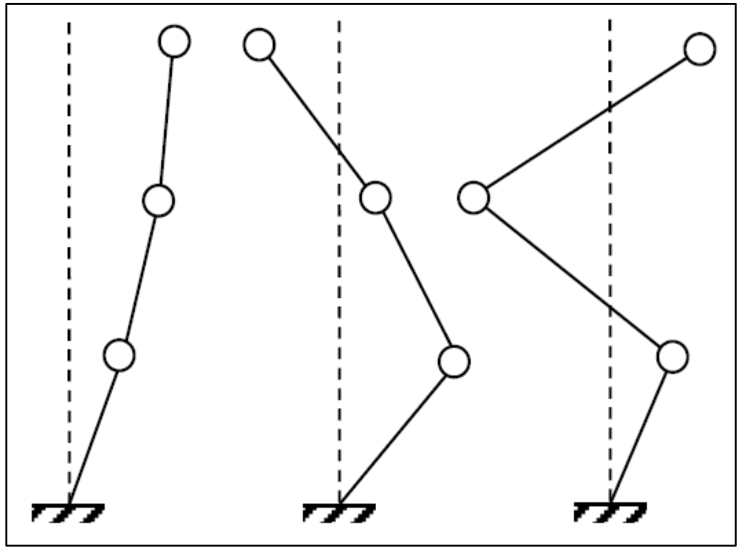
$$\underline{K} = \begin{bmatrix} k_1 + k_2 & -k_2 & 0 \\ -k_2 & k_2 + k_3 & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix} = \begin{bmatrix} 900 & -400 & 0 \\ -400 & 800 & -400 \\ 0 & -400 & 400 \end{bmatrix} \text{MN/m}$$

$$\underline{M} = \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix} = \begin{bmatrix} 300 & 0 & 0 \\ 0 & 200 & 0 \\ 0 & 0 & 200 \end{bmatrix} \text{tons}$$

$$\Rightarrow \omega_1 = 20.63\,\text{rad/sec}, \quad \omega_2 = 51.44\,\text{rad/sec}, \quad \omega_3 = 77.0\,\text{rad/sec}$$

$$\Rightarrow T_1 = 0.3048\,\text{sec}, \quad T_2 = 0.1221\,\text{sec}, \quad T_3 = 0.0816\,\text{sec}$$

$$\Rightarrow \underline{\varphi}_1 = \begin{bmatrix} 0.00067 \\ 0.00129 \\ 0.00164 \end{bmatrix}, \quad \underline{\varphi}_2 = \begin{bmatrix} 0.00149 \\ 0.00040 \\ -0.00122 \end{bmatrix}, \quad \underline{\varphi}_3 = \begin{bmatrix} 0.00081 \\ -0.00178 \\ 0.00091 \end{bmatrix}$$



- Solving the eigenproblem using the function ***eig(K,M)*** of Matlab:

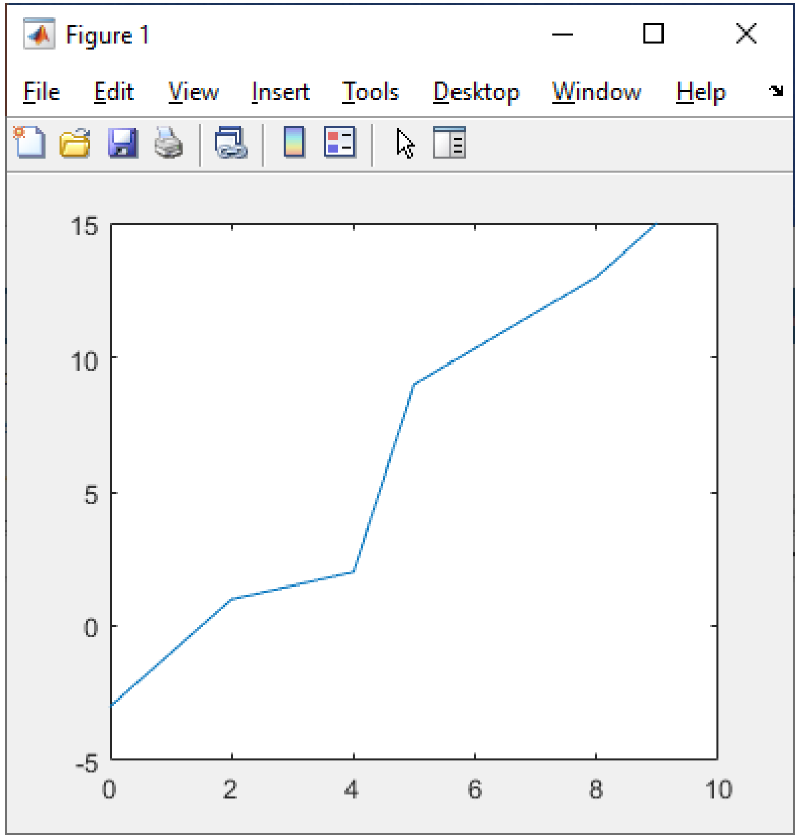
```
>> K = [ 900 -400 0 ; -400 800 -400 ; 0 -400 400] * 1e6
K =
   900000000  -400000000           0
  -400000000   800000000  -400000000
           0  -400000000   400000000
>> M = 1000 * diag([300 200 200])
M =
      300000           0           0
           0      200000           0
           0           0      200000
>> [V,D] = eig(K,M)
V =
   -0.0007    0.0015   -0.0008
   -0.0013    0.0004    0.0018
   -0.0016   -0.0012   -0.0009
D =
   1.0e+03 *
    0.4249         0         0
         0    2.6464         0
         0         0    5.9287
>> format long
>> f1=-V(:,1)
f1 =
   0.000666847703695
   0.001287896288212
   0.001635326988992
```

```
>> f2=-V(:,2)
f2 =
  -0.001492844308436
  -0.000395893863414
   0.001224904835107
>> f3=-V(:,3)
f3 =
   0.000812442825177
  -0.001784542294181
   0.000908467822218
>> w1=sqrt(D(1,1))
w1 =
  20.613265253995944
>> w2=sqrt(D(2,2))
w2 =
  51.443245859863360
>> w3=sqrt(D(3,3))
w3 =
  76.997959394844059
>> T1=2*pi/w1
T1 =
   0.304812713064058
>> T2=2*pi/w2
T2 =
   0.122138197194936
>> T3=2*pi/w3
T3 =
   0.081601971747842
```

Basic plotting using the *plot()* function

- The most important and useful function for plotting is the ***plot()*** function, which, in its most common use, accepts 2 parameters, e.g. *x* and *y* vectors, as in the function call *plot(x,y)* and plots vector *y* (vertical coordinates) *vs.* vector *x (horizontal coordinates)*, using the defaults line (which is a continuous, or solid, line) with the default color (which is blue) in the default figure (which is *Figure 1*).

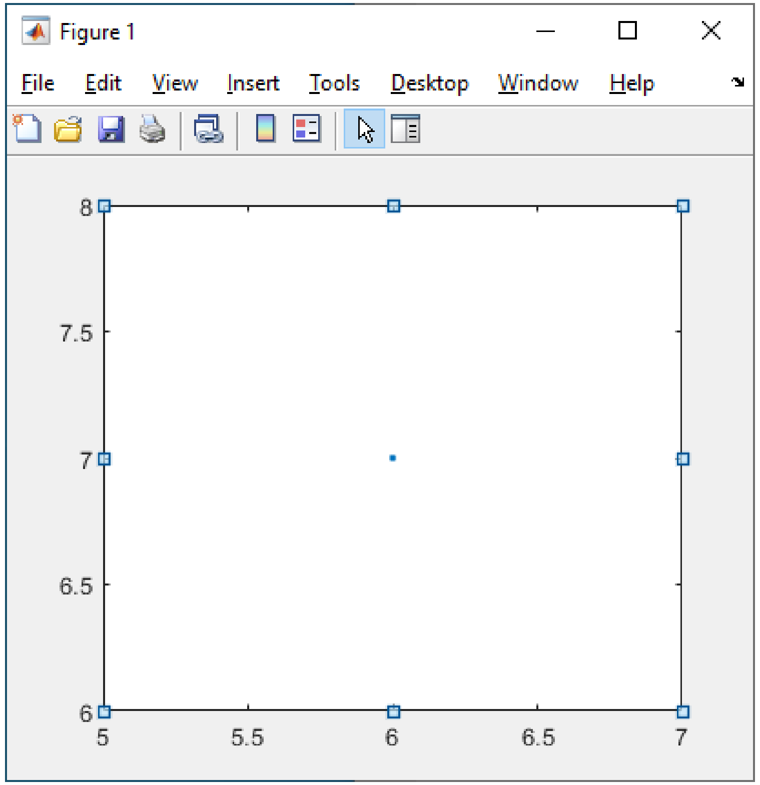
```
>> a = [ 0 2 4 5 8 9]
a =
     0     2     4     5     8     9
>> b = [ -3 1 2 9 13 15]
b =
    -3     1     2     9    13    15
>> plot(a,b)
```



➢ The function call *plot(a,b)* plots, in Figure 1, blue solid (continuous) line segments between the successive pairs of a and b coordinates, i.e. from point (a(1),b(1)) to the point (a(2),b(2)), from point (a(2),b(2)) to the point (a(3),b(3)), etc.

- If the provided parameters are scalars (i.e. 1x1 vectors) then a point will be plotted, using, by default (unless otherwise specified) a blue point (or dot) symbol in the default figure (which is *Figure 1*).

```
>> c = 6;
>> d = 7;
>> plot(c,d)
```

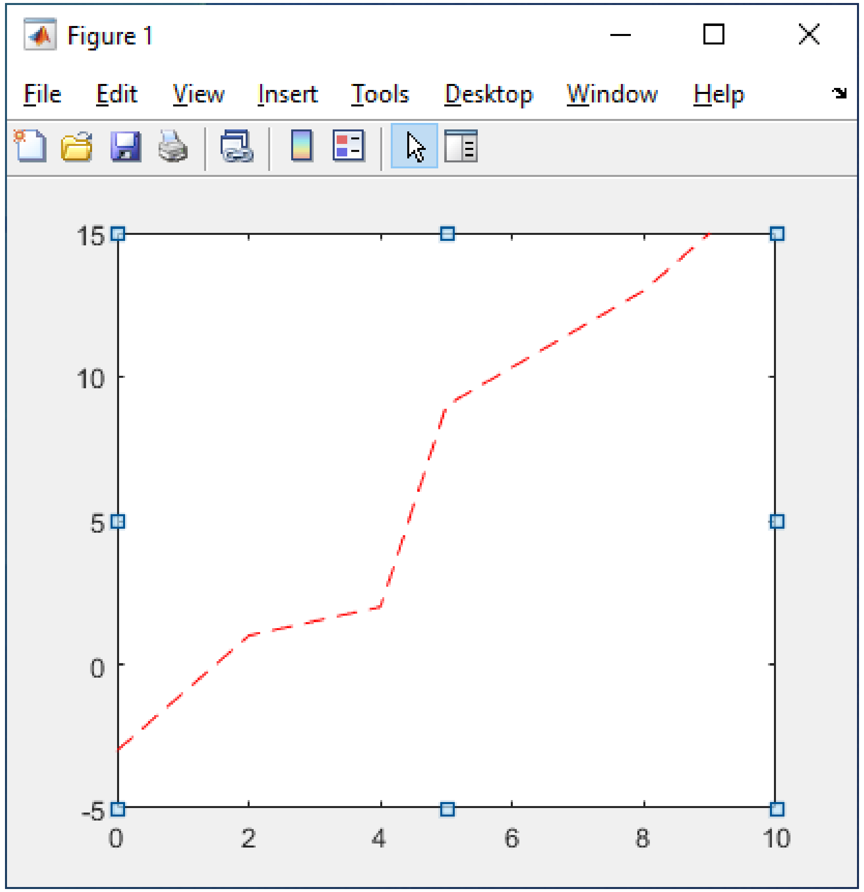


➢ The function call *plot(c,d)* plots, in Figure 1, a blue point at the *c* (horizontal) and *d* (vertical) coordinates, i.e. at point (6,7).

```
>> help plot
 plot   Linear plot.
    plot(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
    then the vector is plotted versus the rows or columns of the matrix,
    whichever line up.  If X is a scalar and Y is a vector, disconnected
    line objects are created and plotted as discrete points vertically at
    X.
```

- The function ***plot()*** takes a 3rd parameter, which indicates what color and/or what symbol or line type should be used during the plotting, e.g. the function call *plot(x,y,'r--')* plots vector *y* (vertical coordinates) *vs.* vector *x (horizontal coordinates)*, using the dashed line (--) of red ('r') color in the default figure (which is *Figure 1*).

```
>> a = [ 0 2 4 5 8 9];
>> b = [ -3 1 2 9 13 15];
>> plot(a,b,'r--')
```



Various line types, plot symbols and colors may be obtained with **plot**(X,Y,S) where S is a character string made from one element from any or all the following 3 columns:

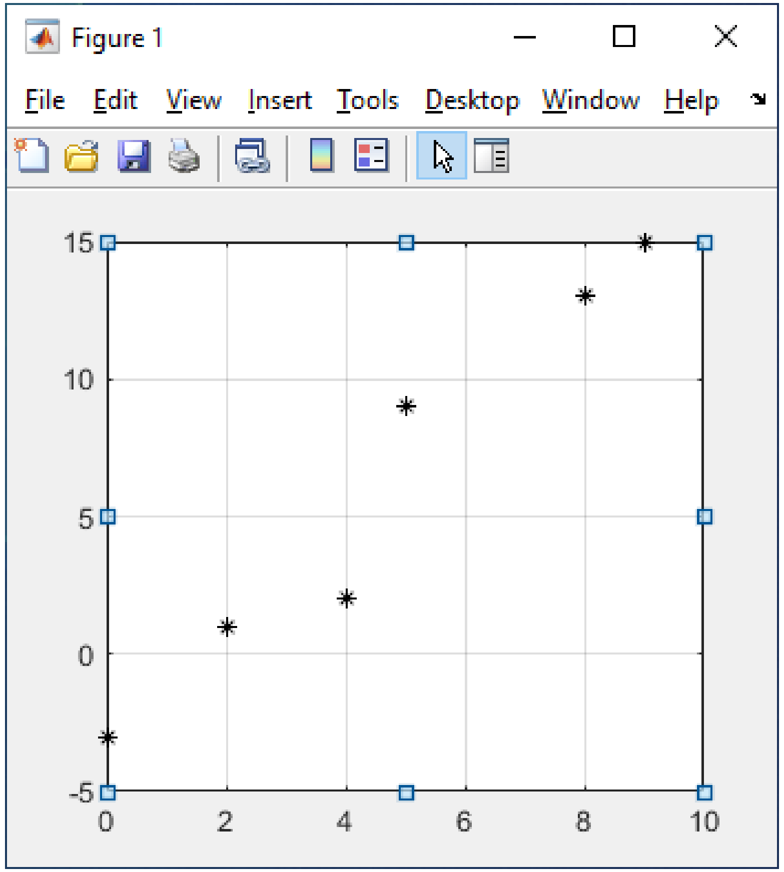
b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		

```
>> help plot
plot   Linear plot.
   plot(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
   then the vector is plotted versus the rows or columns of the matrix,
   whichever line up.  If X is a scalar and Y is a vector, disconnected
   line objects are created and plotted as discrete points vertically at
   X.

   plot(Y) plots the columns of Y ve
   If Y is complex, plot(Y) is equiv
   In all other uses of plot, the im

   Various line types, plot symbols
   plot(X,Y,S) where S is a characte
   from any or all the following 3 c

          b     blue          .
          g     green         o
```

```
Various line types, plot symbols and colors may be obtained with
plot(X,Y,S) where S is a character string made from one element
from any or all the following 3 columns:

       b     blue          .     point              -     solid
       g     green         o     circle             :     dotted
       r     red           x     x-mark             -.    dashdot
       c     cyan          +     plus               --    dashed
       m     magenta       *     star             (none)  no line
       y     yellow        s     square
       k     black         d     diamond
       w     white         v     triangle (down)
                           ^     triangle (up)
                           <     triangle (left)
                           >     triangle (right)
                           p     pentagram
                           h     hexagram
```

- Using a 3rd parameter of the point symbol type in the function ***plot()*** will result in plotting points with that symbol instead of line segments, e.g. the function call *plot(x,y,'k*')* plots vector points at the corresponding pairs of coordinates of the *y* (vertical coordinates) *vs.* vector *x (horizontal coordinates)*, using black stars ('*') in the default figure (which is *Figure 1*).

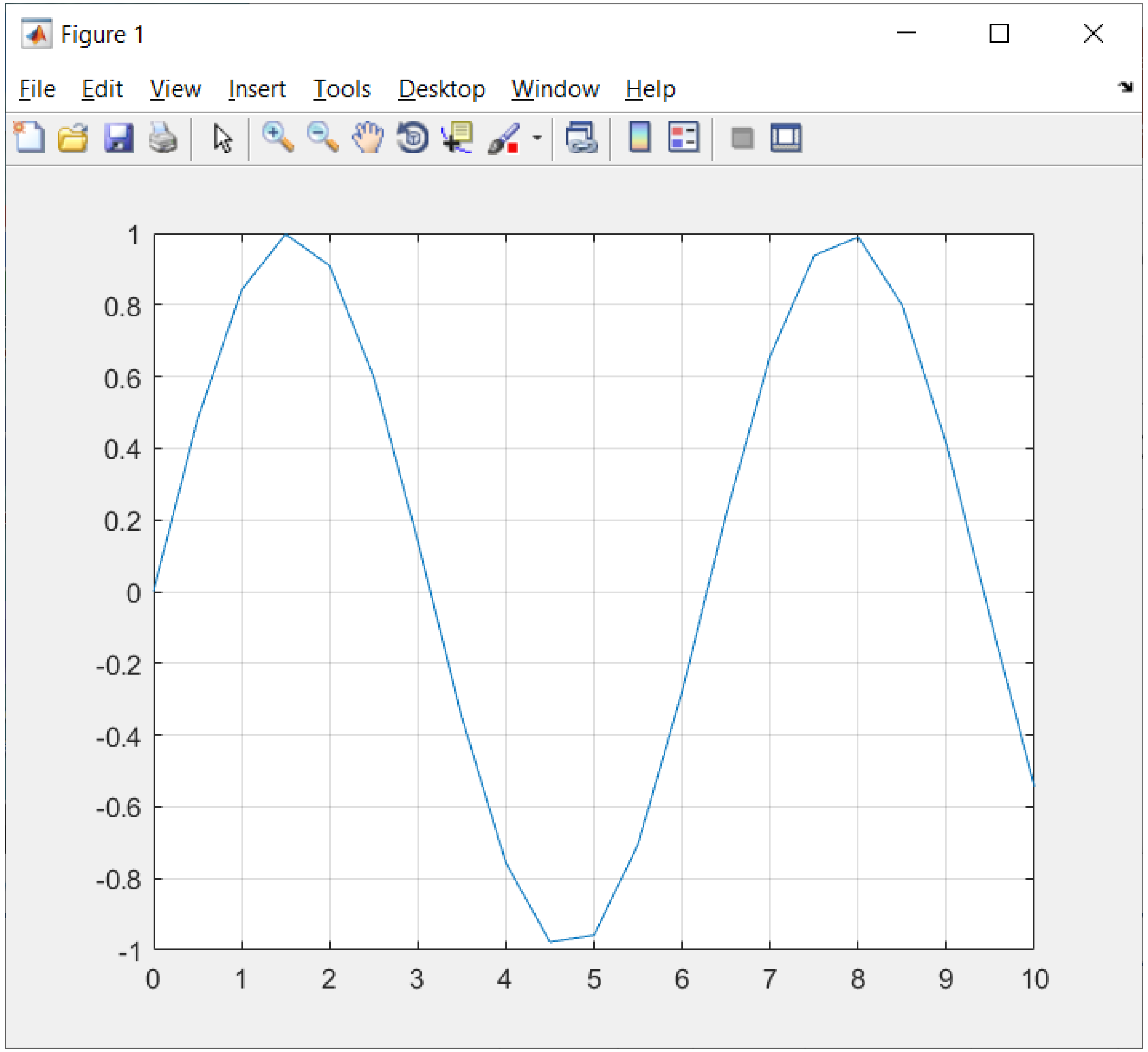
```
>> a = [ 0 2 4 5 8 9];
>> b = [ -3 1 2 9 13 15];
>> plot(a,b,'k*')
>> grid on
```



- The command ***grid on*** adds grid lines to the current axes of the current plot, while the command ***grid off*** removes grid lines from the current axes.

- The colon operator **:** can be used to create a series of numbers, i.e. forming a vector

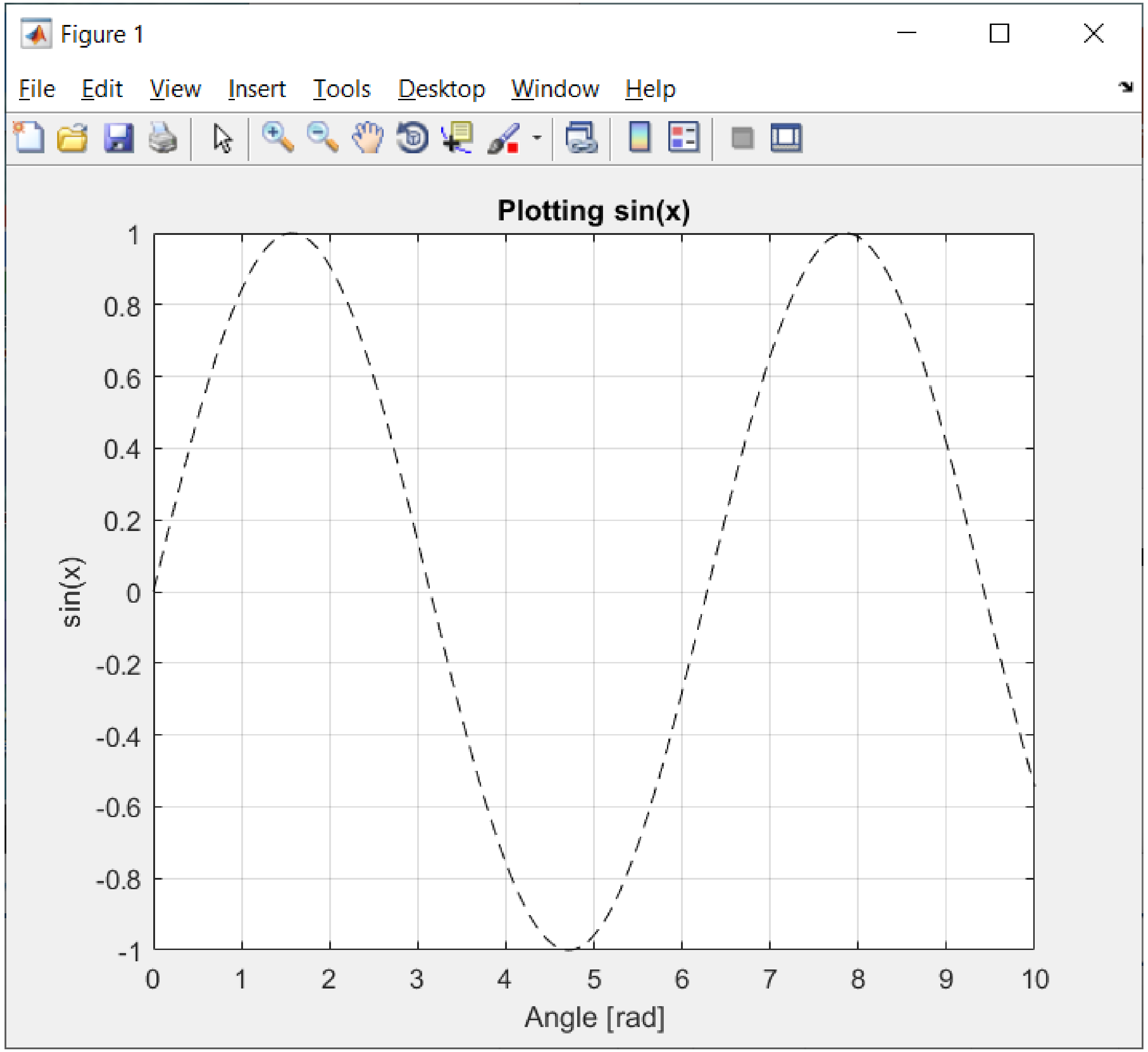
```
>> clear
>> x=0:0.5:10;
>> y=sin(x);
>> plot(x,y)
>> grid on
```



- A vector with values 0, 0.5, 1, 1.5….10 is created and stored in vector x.
- Vector y is computed as *sin(x)*
- Line segments are plotted in the current figure, which is the default figure (*Figure 1*) in the entire graph using the default color (solid) and the default color (blue).
- The command *grid on* adds a grid on the plot.

- In order to have a smoother plot, more points can be defined, decreasing the interval that is used.

```
>> clear
>> clf
>> xx=0:0.05:10;
>> yy=sin(xx);
>> plot(xx,yy,'k--')
>> grid on
>> title('Plotting sin(x)')
>> xlabel('Angle [rad]')
>> ylabel('sin(x)')
```



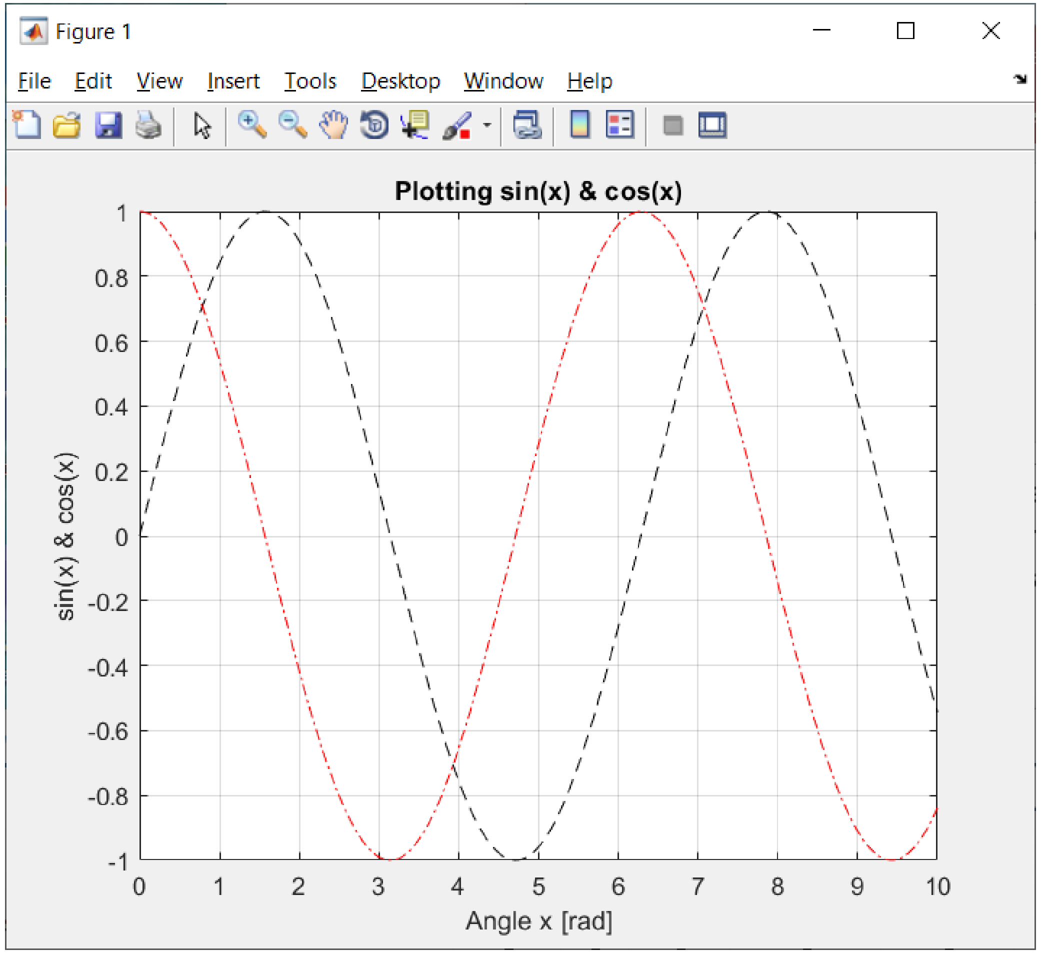
- The ***title()*** function can be used to add a title (provided as parameter) to the current plot.
- The ***xlabel()*** and ***ylabel()*** functions can be used to add a label (provided as parameter) to the horizontal and vertical axes, respectively, of the current plot.

Multiple plots on the same graph

- By default, when the ***plot()*** function is called, the current (active) figure is cleared (deleting any existing plot) and then a new plot is created, according to the provided parameters. Therefore, anything previously plotted is by default cleared upon calling the ***plot()*** function.

  - In order to keep a plot, preventing any erasing before a new plotting, the ***hold on*** command should be used

```
>> clf
>> xx=0:0.05:10;
>> yy=sin(xx);
>> plot(xx,yy,'k--')
>> title('Plotting sin(x) & cos(x)')
>> xlabel('Angle x [rad]')
>> ylabel('sin(x) & cos(x)')
>> hold on
>> zz=cos(xx);
>> plot(xx,zz,'r-.')
>> grid on
```



- The command ***hold on*** retains plots in the current axes of the current graph (which by default, unless differently specified with the function *subplot()*, takes the entire figure), in the current figure (which by default, unless differently specified with the function *figure()*, is Figure 1), so that new plots are added without deleting existing plots.
- The command ***hold off***, which is the default behavior, sets the hold state to off so that, before new plots added to the current graph of the current figure, existing plots are cleared and all axes properties are reset.
- The command ***clf*** clears the current figure, while calling ***clf()*** with an argument clears the corresponding figure.

  e.g. *clf*: clears the current figure

  clf(5): clears figure 5

Multiple graphs on the same figure

- In order to create multiple graphs on the same figure, the current (active) figure can be subdivided into a number of rows and columns, using the function ***subplot().***
- The function ***subplot()*** takes 3 arguments: the number of rows and the number of columns, in which the current figure should be subdivided, and an index that indicates the active graph (i.e. subplot), in which the next call to the function *plot()* should be used to plot accordingly.
- The numbering (indexing) of the subplots starts with 1 on the top left and increases from left to right and then from top to bottom.

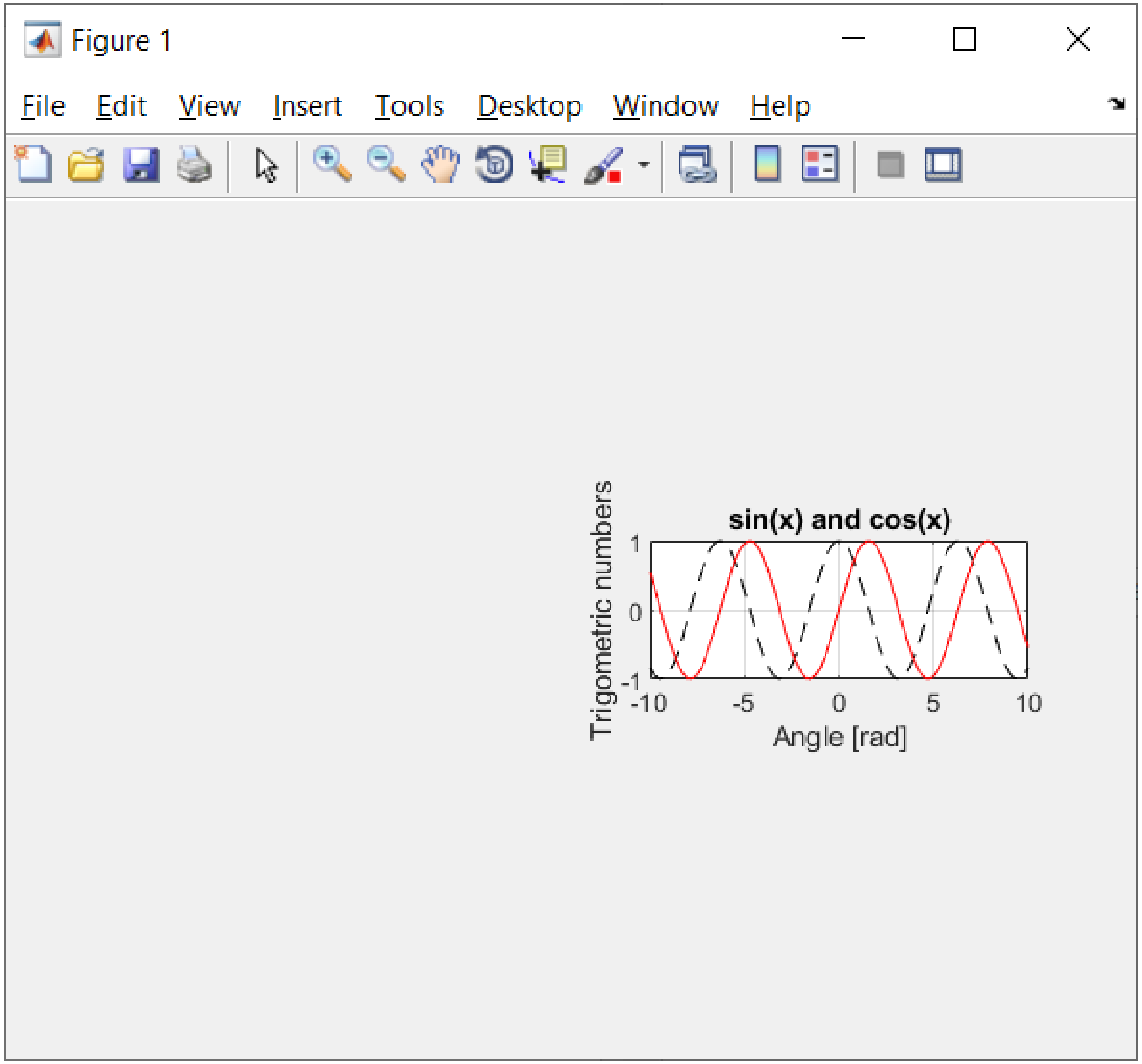
➢ e.g. subplot(2,3,5) splits the active figure in a 2x3 grid of graphs (2 rows & 3 columns) and makes active the subplot 5.

1	**2**	**3**
4	**5**	6

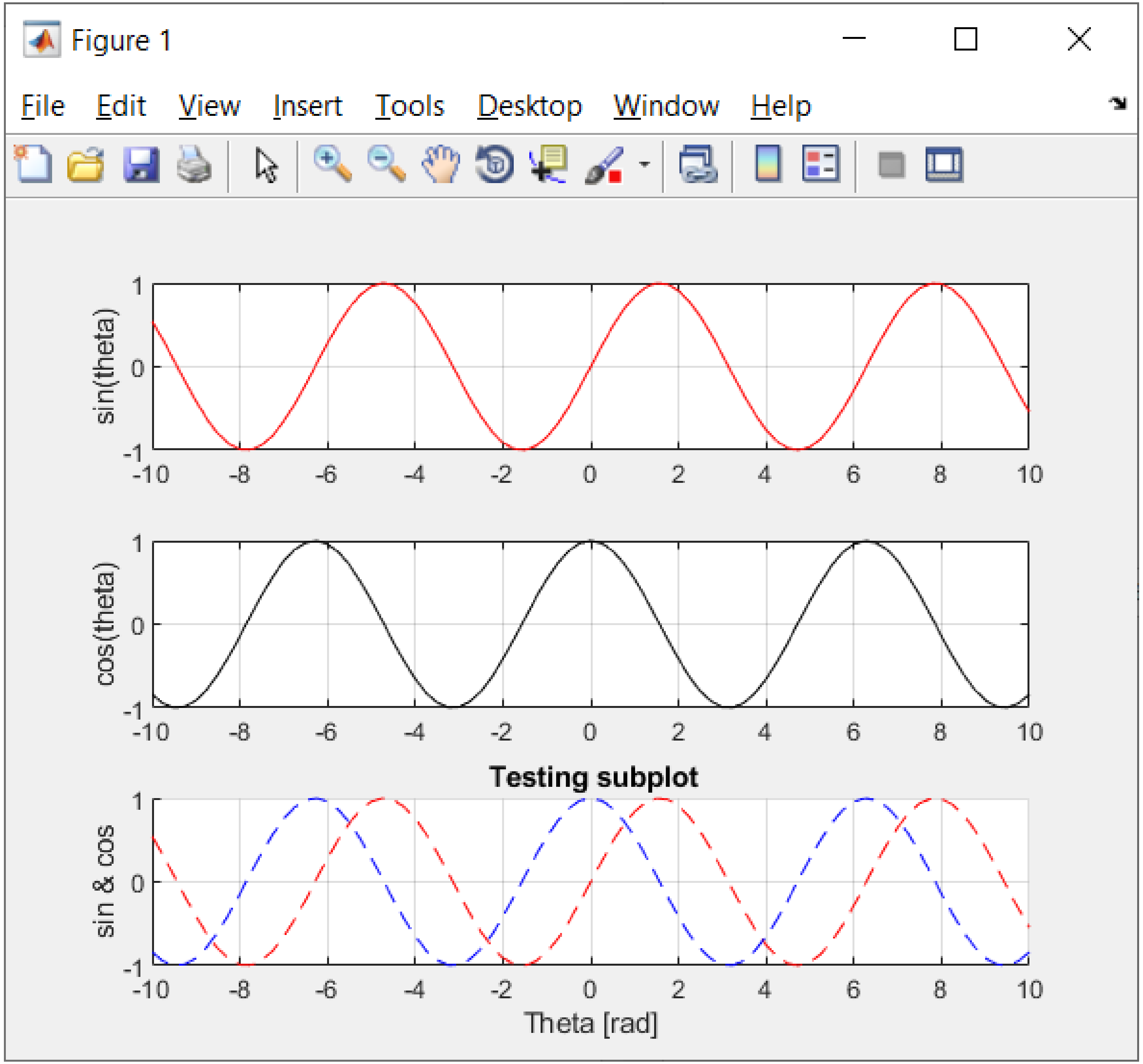
➢ e.g. subplot(2,2,4) splits the active figure in a 2x2 grid of graphs and makes active the subplot 4.

1	**2**
3	**4**

```
>> x=-10:0.25:10;
>> s1 = sin(x);
>> c1 = cos(x);
>> clf
>> subplot(3,2,4)
>> plot(x,c1,'k--')
>> hold on
>> plot(x,s1,'r-')
>> grid on
>> title('sin(x) and cos(x)')
>> xlabel('Angle [rad]')
>> ylabel('Trigometric numbers')
```



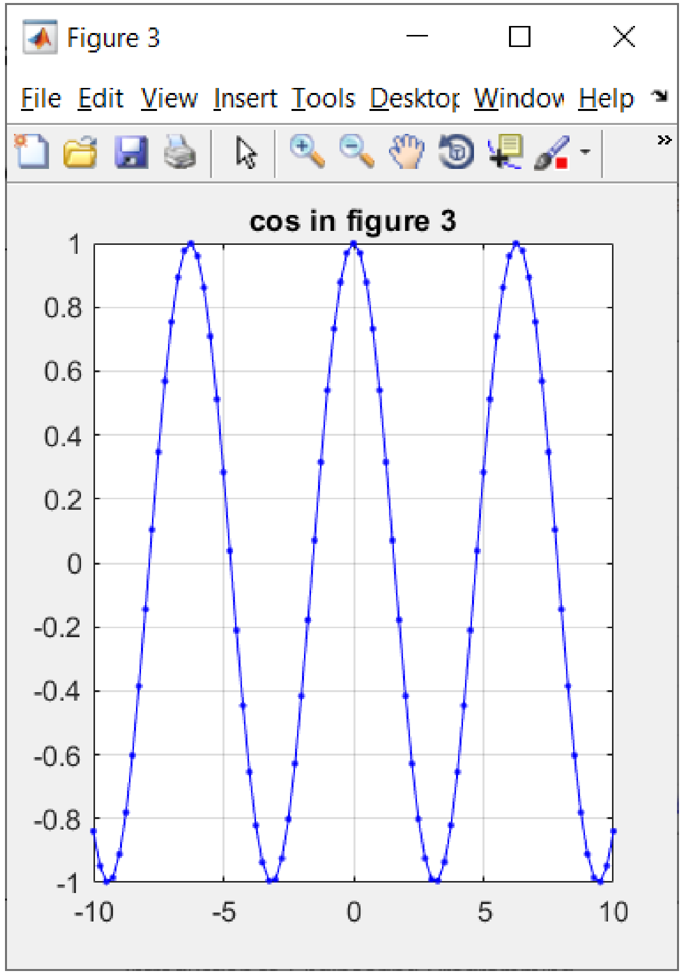
```
>> clf
>> clear
>> x=-10:0.25:10;
>> s1 = sin(x);
>> c1 = cos(x);
>> subplot(3,1,2)
>> plot(x,c1,'k-')
>> grid on
>> ylabel('cos(theta)')
>> subplot(3,1,1)
>> plot(x,s1,'r')
>> grid on
>> ylabel('sin(theta)')
>> subplot(3,1,3)
>> hold on
>> plot(x,s1,'r--')
>> plot(x,c1,'b--')
>> grid on
>> ylabel('sin & cos')
>> xlabel('Theta [rad]')
>> title(' Testing subplot')
```

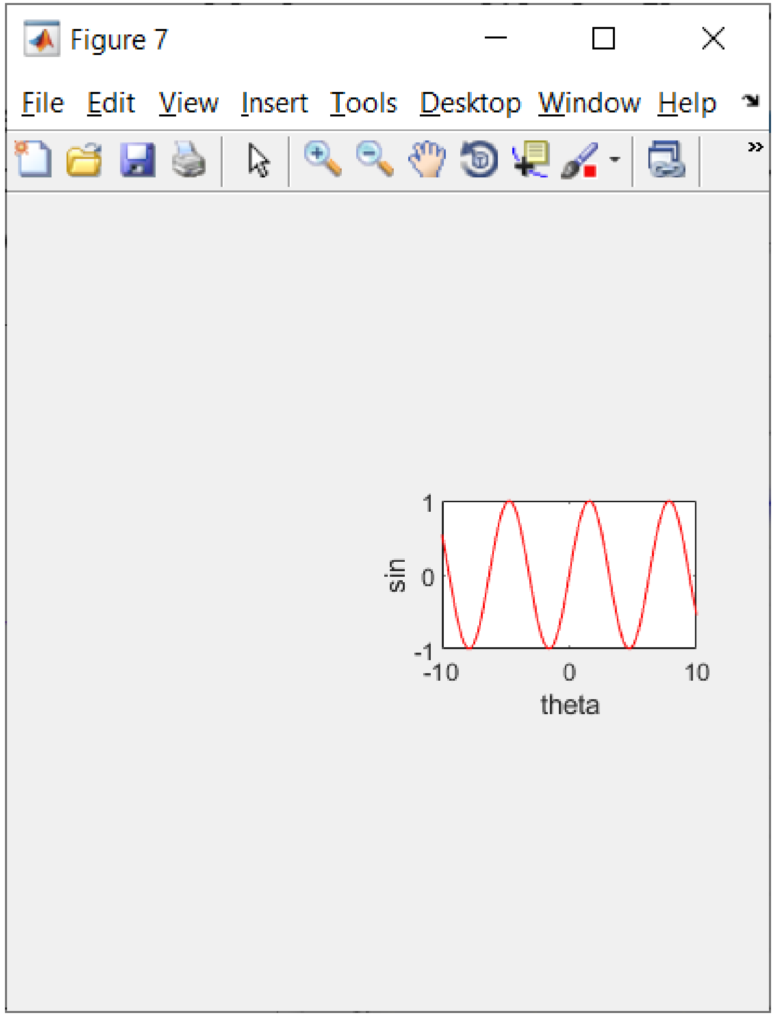


Using multiple figures

- Matlab enables the use of several figures. Function ***figure(n)*** can be used to create the figure *n* if its not already available and forces it to become visible, making it the current (active) figure, raised above all other figures on the screen.

```
>> x=-10:0.25:10;
>> s1 = sin(x);
>> c1 = cos(x);
>> figure(3)
>> plot(x,c1,'b.-')
>> title('cos in figure 3')
>> figure(7)
>> subplot(3,2,4)
>> plot(x,s1,'r-')
>> xlabel('theta')
>> ylabel('sin')
>> figure(3)
>> grid on
```



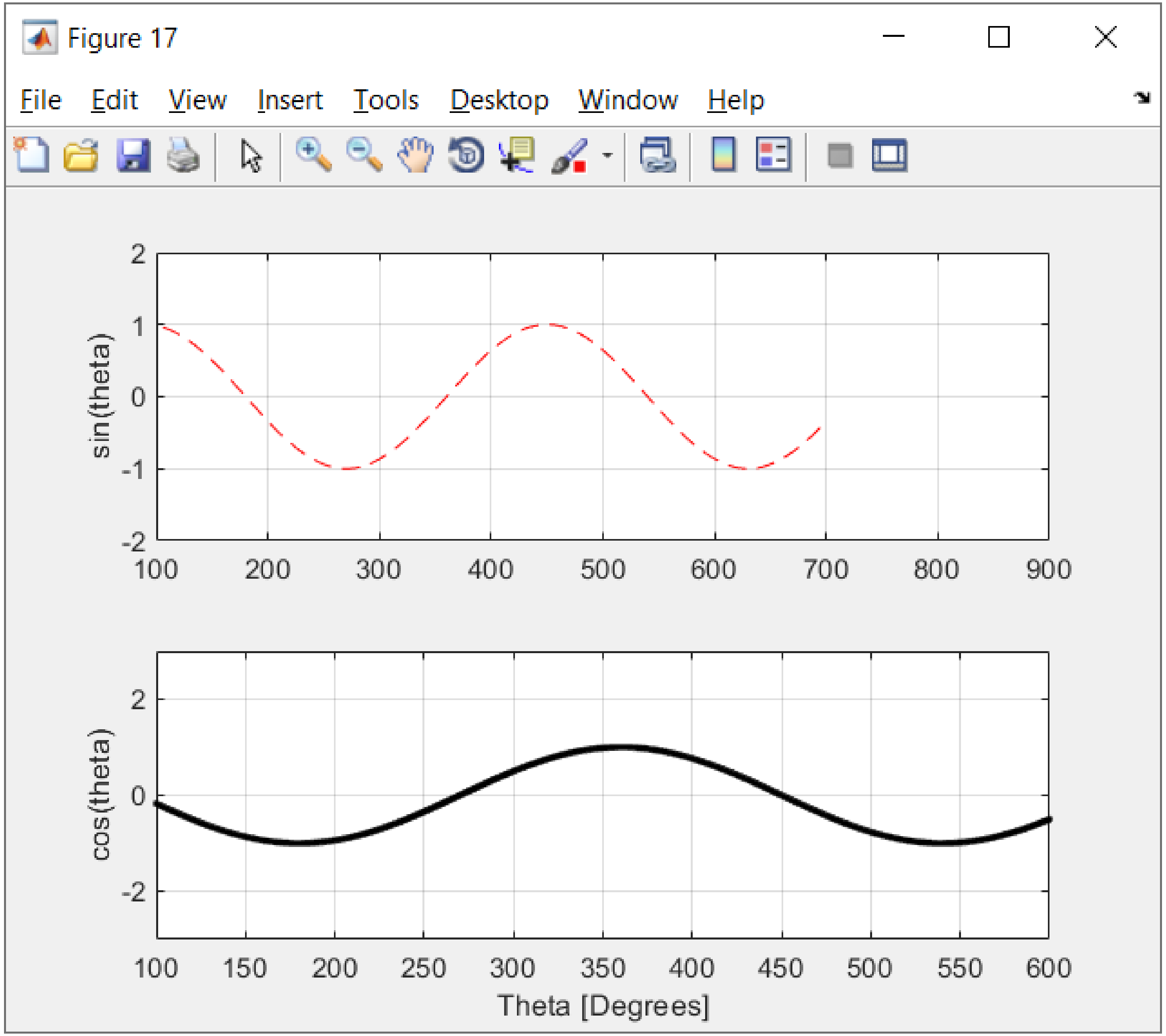


- Function ***axis()*** can be used to control axis scaling and appearance, by defining the minimum and maximum values in the horizontal (x) and vertical (y) directions on the current (active) graph.
  - e.g. *axis([xmin xmax ymin ymax])* sets scaling for the x- and y-axes, of the current graph (on the current figure), so that the x axis ranges from *xmin* to *xmax* and the x axis ranges from *ymin* to *ymax.*

- Another way to control the axis is through the usage of the functions ***xlim()*** and ***ylim()***, which define the ranges (limits) of the axes in the horizontal (x) and vertical (y) axes, respectively.

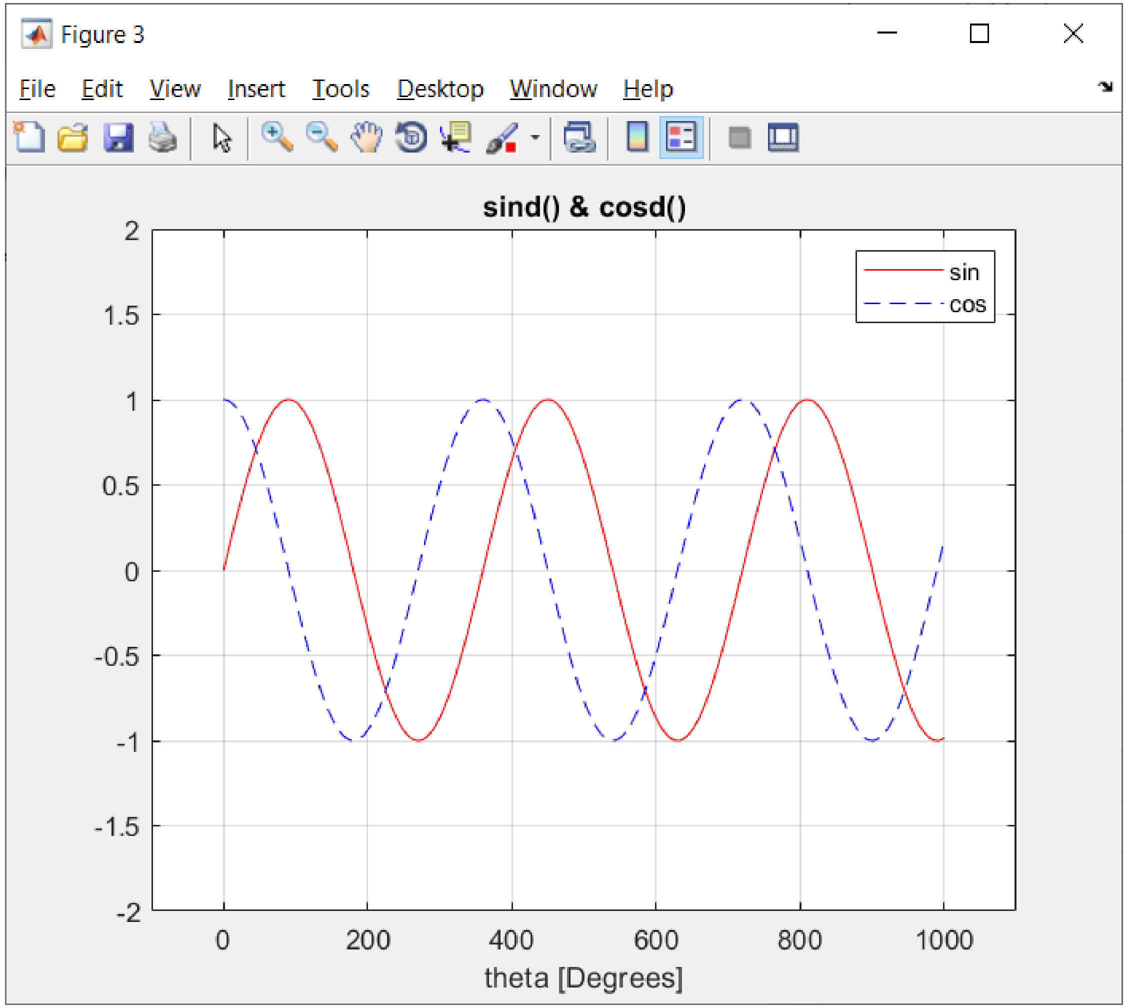
- A figure can be closed, in addition to clicking on the close icon of its window, by calling the function ***close()*** and giving its number as an argument.
- The command ***close***, closes the current figure, while the command ***close all*** closes all figures.

```
>> clear
>> theta=0:0.5:700;
>> s = sind(theta);
>> c = cosd(theta);
>> figure(17)
>> subplot(2,1,1)
>> plot(theta,s,'r--')
>> grid on
>> subplot(2,1,2)
>> plot(theta,c,'k.-')
>> grid on
>> subplot(2,1,1)
>> axis([ 100 900 -2 2])
>> subplot(2,1,2)
>> xlim([100 600])
>> ylim([-3 3])
>> ylabel('cos(theta)')
>> xlabel('Theta [Degrees]')
>> subplot(2,1,1)
>> ylabel('sin(theta)')
```



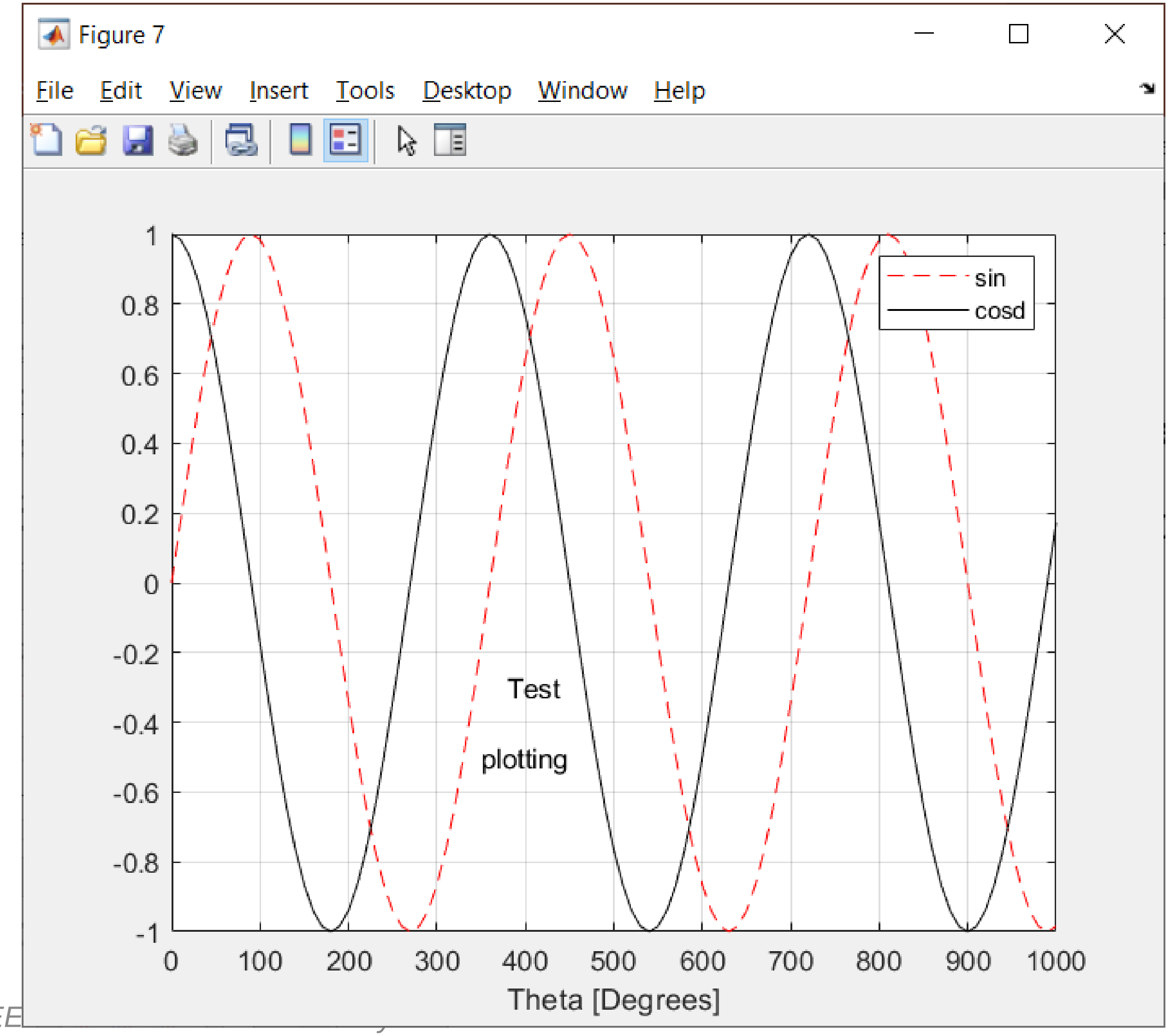
- Function ***legend()*** adds a legend to the current (active) graph, based on the provided parameters.
- The command ***box on*** adds a box to the current graph, while ***axis off*** takes it off.

```
>> clear ; close all;
>> theta=0:0.5:1000;
>> s = sind(theta);
>> c = cosd(theta);
>> figure(3)
>> hold on
>> plot(theta,s,'r-')
>> plot(theta,c,'b--')
>> grid on
>> legend('sin','cos')
>> box on
>> title('sind() & cosd()')
>> myLimits = [ -100 1100 -2 2];
>> axis(myLimits)
>> xlabel('theta [Degrees]')
```



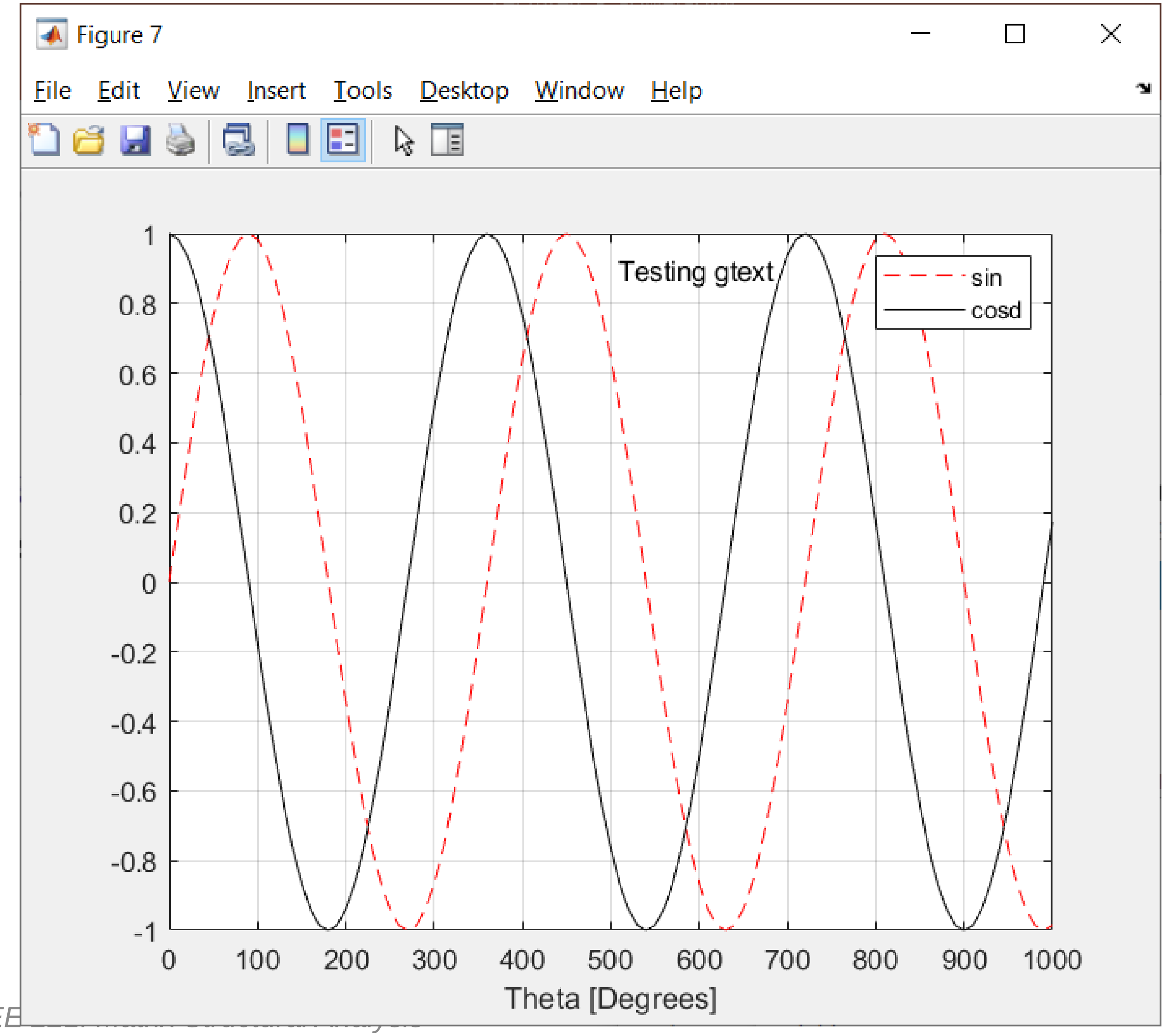
- Function ***text(x,y,str)*** adds the text specified by the 3rd parameter (*str*), at the x and y coordinates of the active graph.

```
clc
clear
theta=0:10:1000;
s=sind(theta);
c=cosd(theta);
figure(7)
clf
plot(theta,s,'r--')
hold on
plot(theta,c,'k')
legend('sin', 'cosd')
grid on
xlabel('Theta [Degrees]')
text(380,-0.3, 'Test')
text(350,-0.5, 'plotting')
```



- Function ***gtext(str)*** adds the text specified, by the parameter *str*, at the point specified with the mouse by the user, who is provided by a cross-hair to put on the active plot.

```
clc
clear
theta=0:10:1000;
s=sind(theta);
c=cosd(theta);
figure(7)
clf
plot(theta,s,'r--')
hold on
plot(theta,c,'k')
legend('sin', 'cosd')
grid on
xlabel('Theta [Degrees]')
gtext('Testing gtext')
```



Logarithmic plotting functions

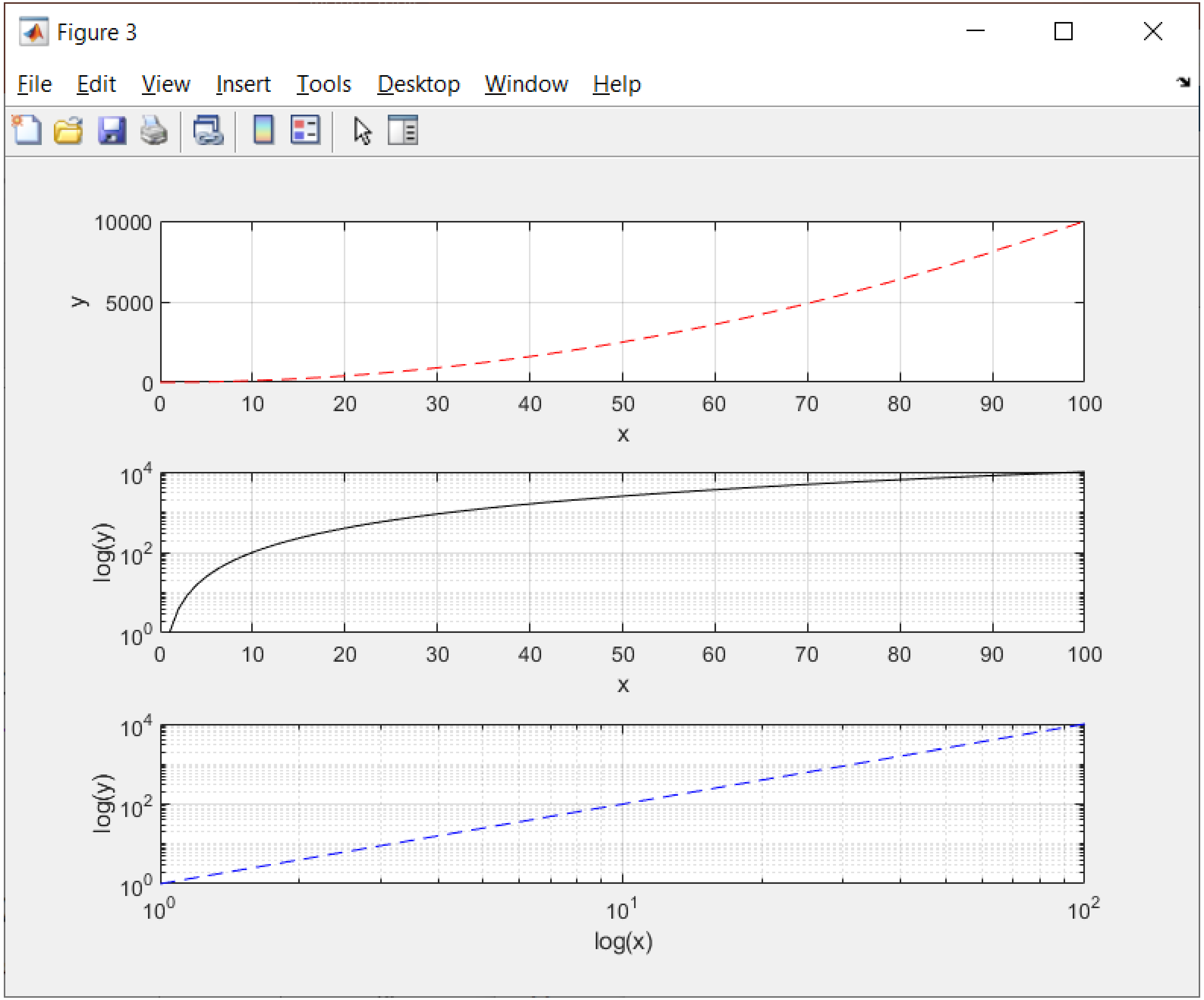
plot() : Linear plot.

loglog() : Log-log scale plot.

semilogx() : Semi-log scale plot.

semilogy() : Semi-log scale plot.

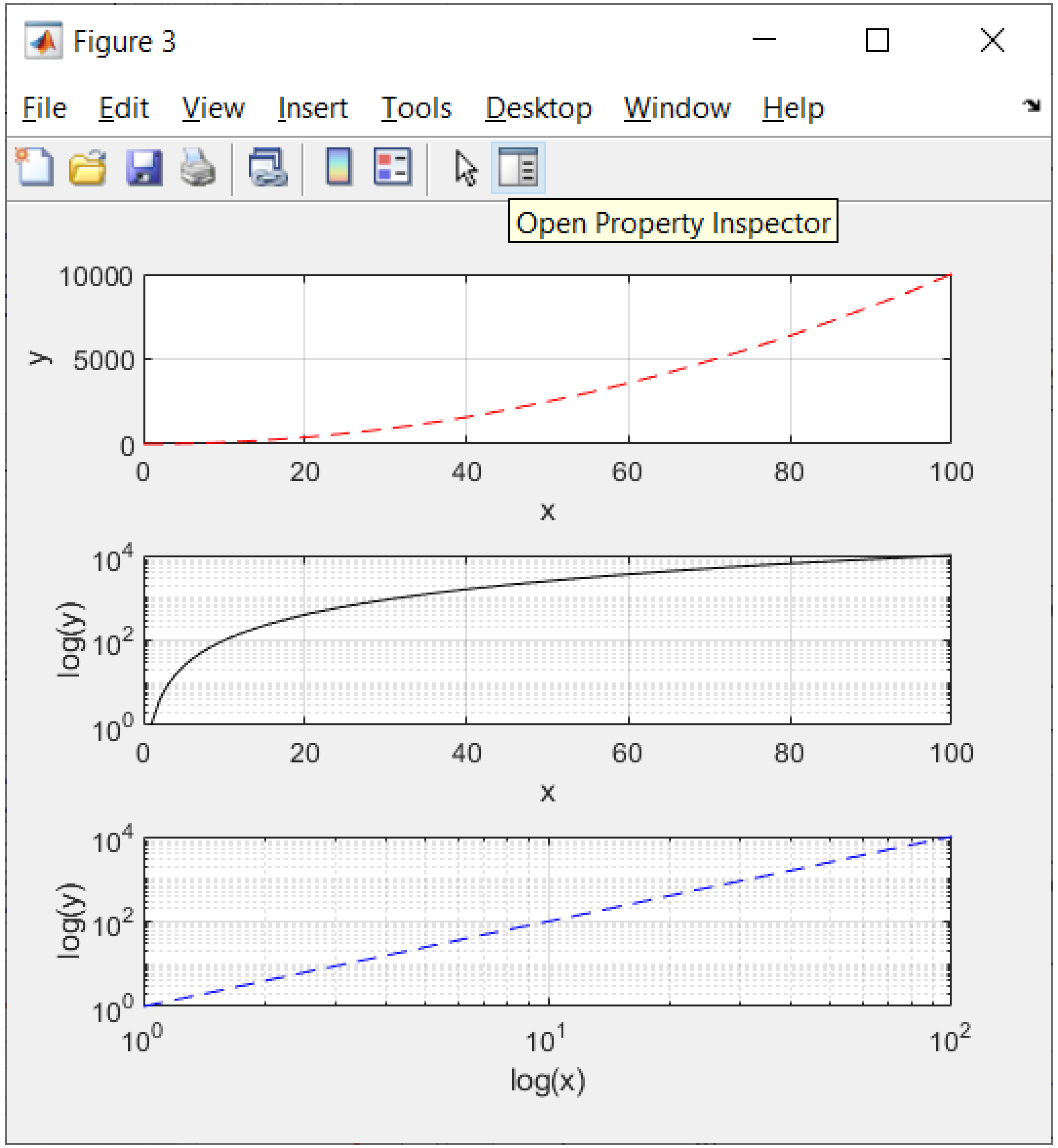
```
clc
clear
x=0:1:100;
y = x.^2;
figure(3)
clf
subplot(3,1,1)
plot(x,y,'r--')
grid on
xlabel('x')
ylabel('y')
subplot(3,1,2)
semilogy(x,y,'k-')
grid on
xlabel('x')
ylabel('log(y)')
subplot(3,1,3)
loglog(x,y,'b--')
grid on
xlabel('log(x)')
ylabel('log(y)')
```



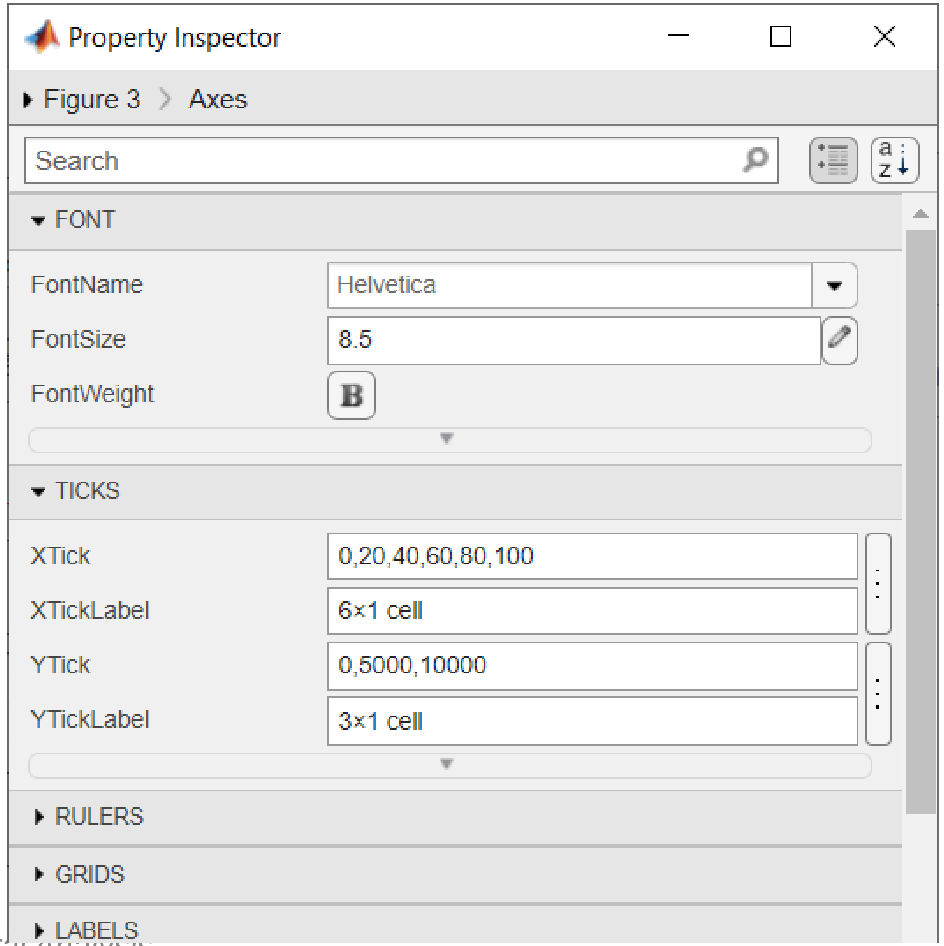
Manipulating figures

- ***plotedit*** toggles the state of plot edit mode for the current figure.
- ***plotedit on*** starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily.
- In plot edit mode, you can label axes, change line styles, and adding text, line, and arrow annotations.
- ***plotedit off*** ends plot mode for the current figure.

```
>> help plotedit
 plotedit  Tools for editing and annotating plots
    plotedit ON   starts plot edit mode for the current figure.
    plotedit OFF  ends plot edit mode for the current figure.
    plotedit  with no arguments toggles the plot edit mode for
       the current figure.

    plotedit(FIG)  toggles the plot edit mode for figure FIG.
    plotedit(FIG,'STATE')  specifies the plotedit STATE for
       the figure FIG.
    plotedit('STATE')  specifies the plotedit STATE for
       the current figure.

       STATE can be one of the strings:
           ON - starts plot edit mode
           OFF - ends plot edit mode
           SHOWTOOLSMENU - displays the Tools menu (the default)
           HIDETOOLSMENU - removes the Tools menu from the menubar

    When plotedit is ON, use the Tools menu to add and
    modify objects, or select the annotation toolbar buttons
    to add annotations such as text, line and arrows.
    Click and drag objects to move or resize them.

    To edit object properties, right click or double click on
    the object.

    Shift-click to select multiple objects.
```



- Interactively, with the mouse, opening of the *Property Inspector* to annotate and edit (label axes, change line styles, and adding text, line, and arrow annotations) plots easily.



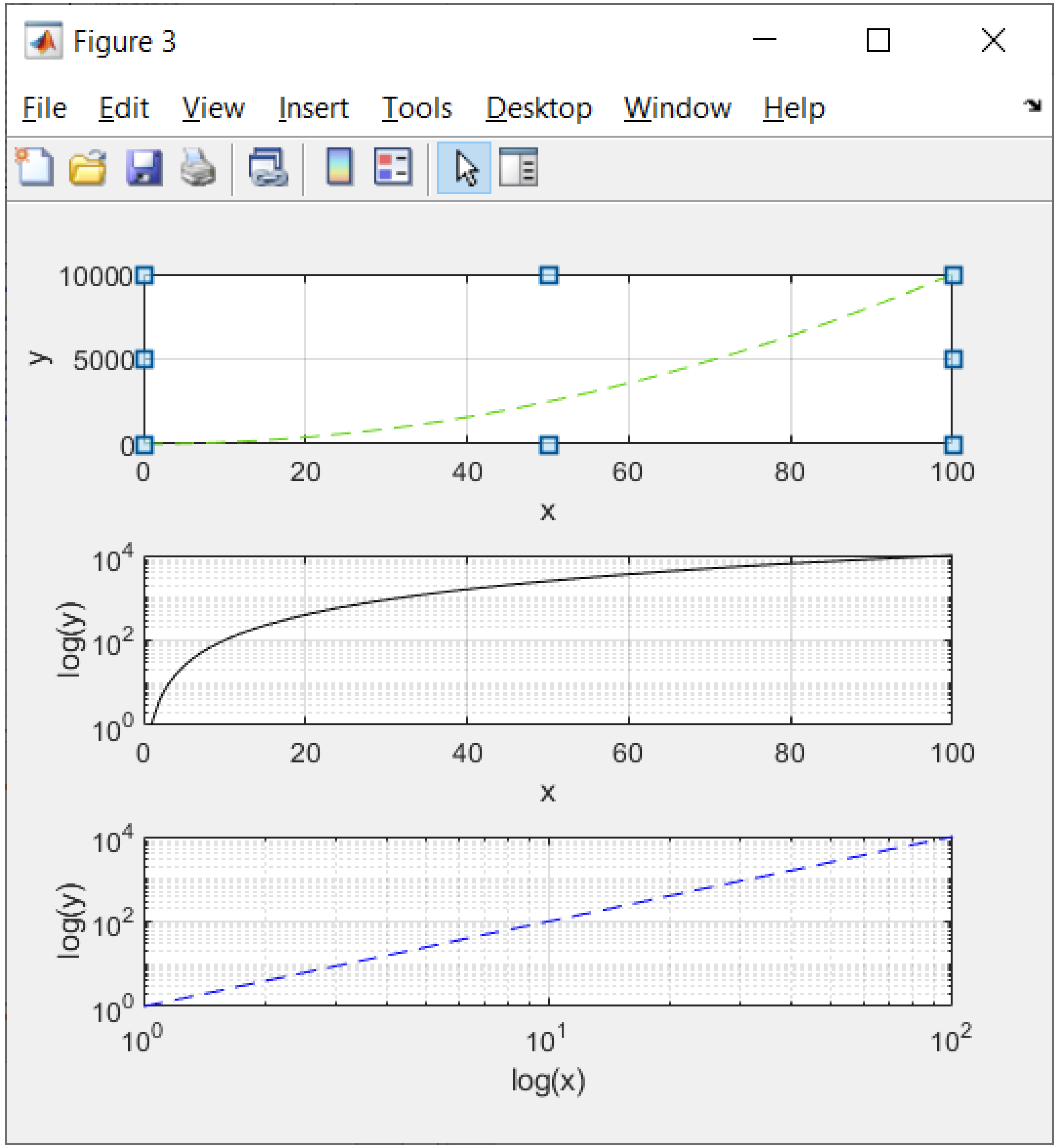
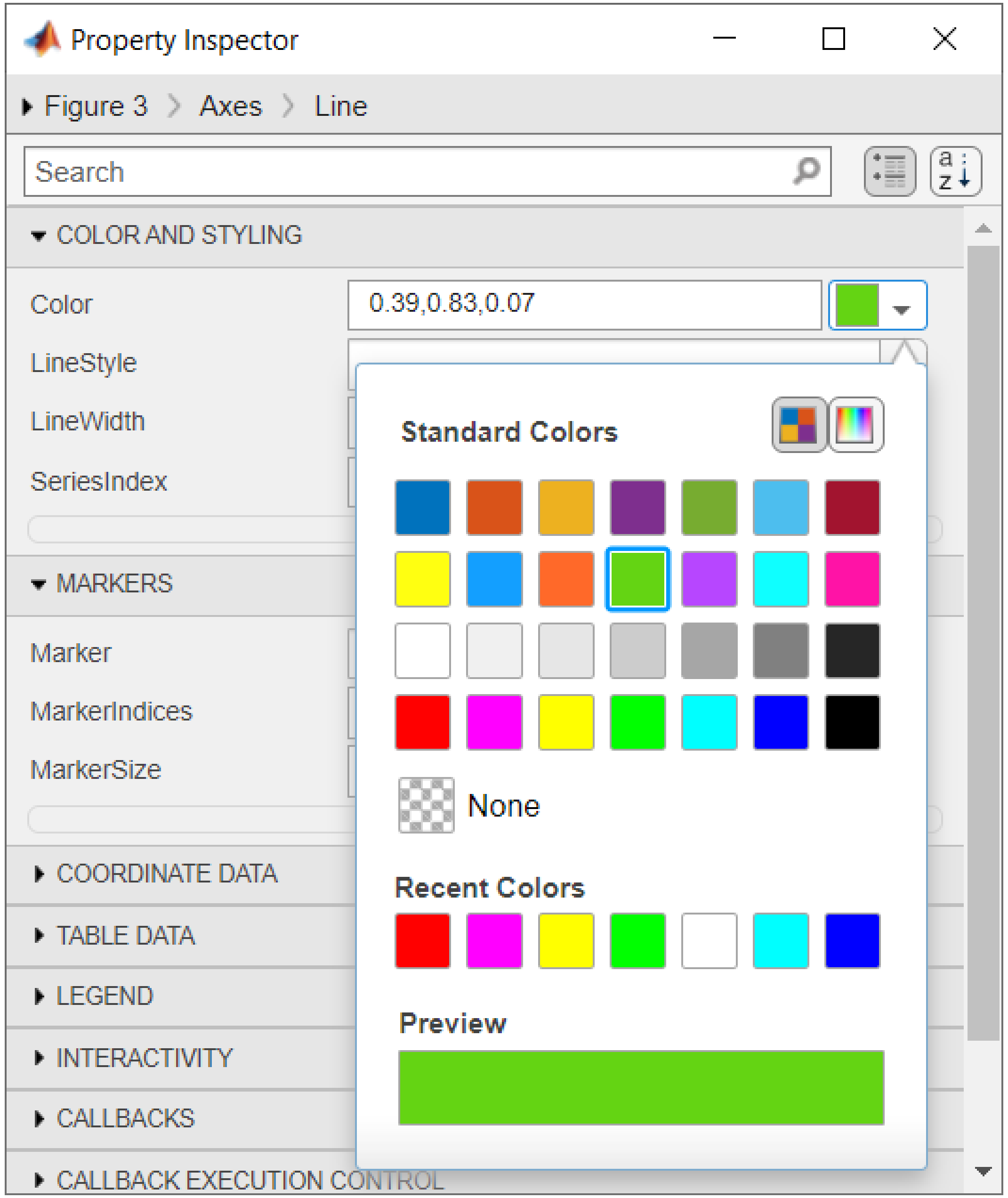
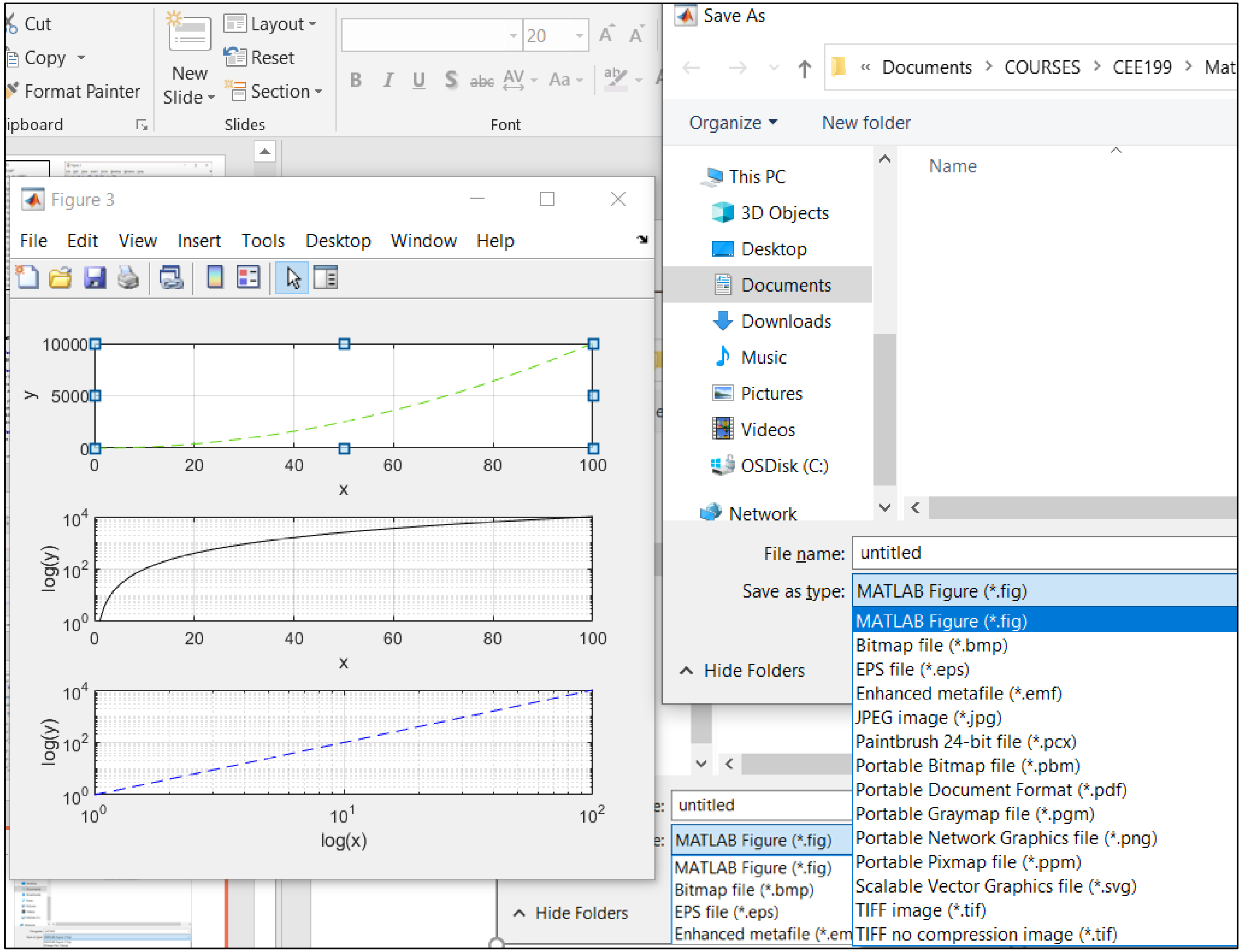


Figure 3
File
Edit
View
Insert
Tools
Desktop
Window
Help
10000
5000
0
y
0
20
40
60
80
100
x
10^4
10^2
10^0
log(y)
0
20
40
60
80
100
x
10^4
10^2
10^0
log(y)
10^0
10^1
10^2
log(x)


Property Inspector
Figure 3
Axes
Line
Search
COLOR AND STYLING
Color
0.39,0.83,0.07
LineStyle
LineWidth
SeriesIndex
MARKERS
Marker
MarkerIndices
MarkerSize
COORDINATE DATA
TABLE DATA
LEGEND
INTERACTIVITY
CALLBACKS
CALLBACK EXECUTION CONTROL
Standard Colors
None
Recent Colors
Preview

Saving and utilizing figures

- Save the code by clicking ***File* > *Save As***.
- Always save important figures in both the Matlab format (as a Matlab figure) and an image format (.jpg, .png, etc.)
- Matlab figures (vector graphics) can be easily opened and edited with Matlab whenever some changes may be needed, while images can be easily added/inserted/uploaded to documents and other media.



- Alternatively, using the function ***saveas(fN,'filename')*** the *fN* figure can be saved in the file named *filename*.

  >> saveas(3,'TestFig3')

  >> dir

  .   ..   TestFig3.fig

```
>> help saveas
 saveas Save Figure or Simulink block diagram in desired output format
    saveas(H,'FILENAME')
    Will save the Figure or Simulink block diagram with handle H to file
    called FILENAME.
    The format of the file is determined from the extension of FILENAME.

    saveas(H,'FILENAME','FORMAT')
    Will save the Figure or Simulink block diagram  with handle H to file
    called FILENAME in the format specified by FORMAT. FORMAT can be the
    same values as extensions of FILENAME.
    The FILENAME extension does not have to be the same as FORMAT.
    The specified FORMAT overrides FILENAME extension.

    Valid options for FORMAT are:

    'fig'  - save figure to a single binary FIG-file.  Reload using OPEN.
    'm'    - save figure to binary FIG-file, and produce callable
             MATLAB code file for reload.
    'mfig' - same as M.
    'mmat' - save figure to callable MATLAB code file as series of creation
             commands with param-value pair arguments.  Large data is saved
             to MAT-file.
             Note: MMAT Does not support some newer graphics features. Use
                   this format only when code inspection is the primary goal.
                   FIG-files support all features, and load more quickly.
```

- In order to save a figure in a image format the function ***print(fN,'filename')*** can be used to save the fN figure in the file named filename as a jpg image.

  >> print('savedFig3.jpg','-djpeg')

  >> dir

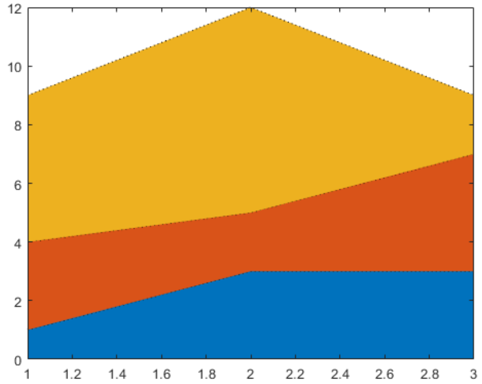
  .   savedFig3.jpg

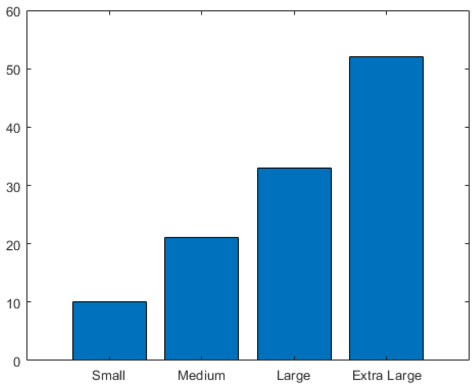
  ..   TestFig3.fig

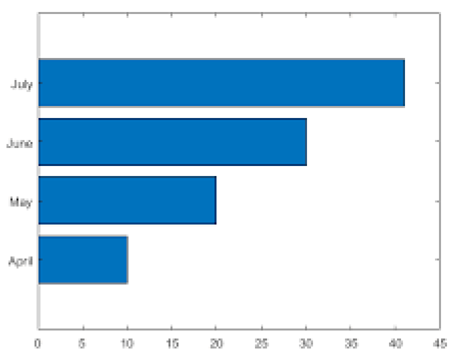
```
print(filename, formattype) saves the current figure to a file in the
specified format. Vector graphics, such as PDF ('-dpdf'), and encapsulated
PostScript ('-depsc'), as well as images such as JPEG ('-djpeg') and PNG ('-dpng')
can be created. Use '-d' to specify the formattype option
  print(fig, '-dpdf', 'myfigure.pdf'); % save to the 'myfigure.pdf' file
The full list of formats is documented here.
```

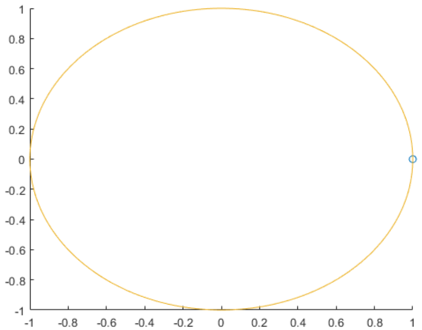
Other specialized two-dimensional (2D) graphing functions

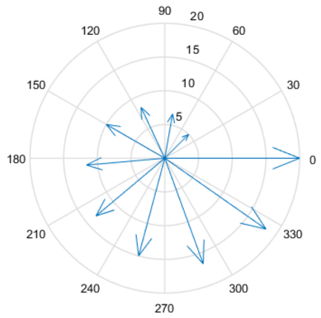
- ***area()***: Filled area 2D plot
- ***bar()***: Bar graph
- ***barh()***: Horizontal bar graph
- ***comet()***: Comet-like trajectory
- ***compass()***: Compass plot
- ***feather()***: Feather plot
- ***fill()***: Filled 2-D polygons
- ***hist()***: Histogram
- ***pie()***: Pie chart

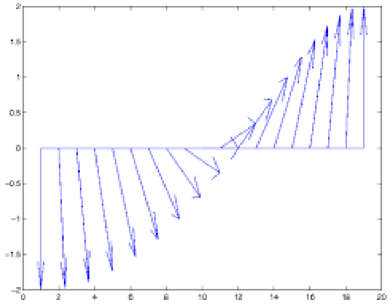


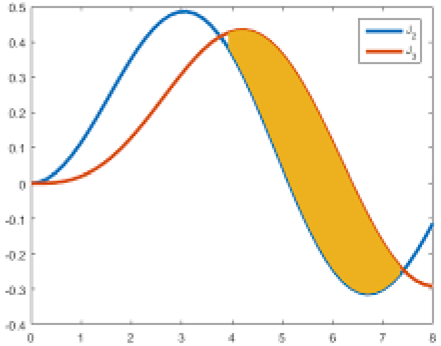




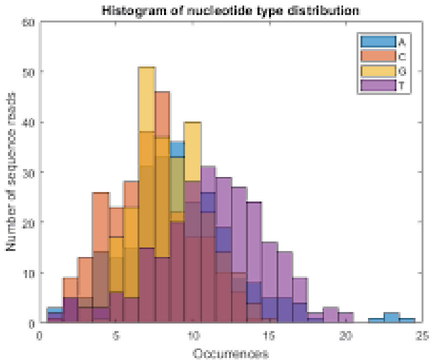


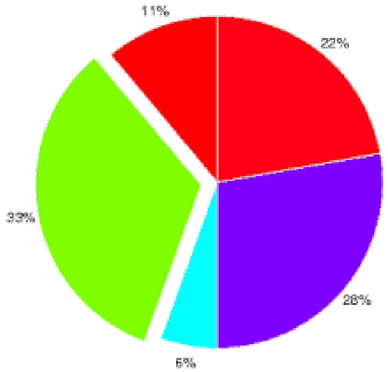




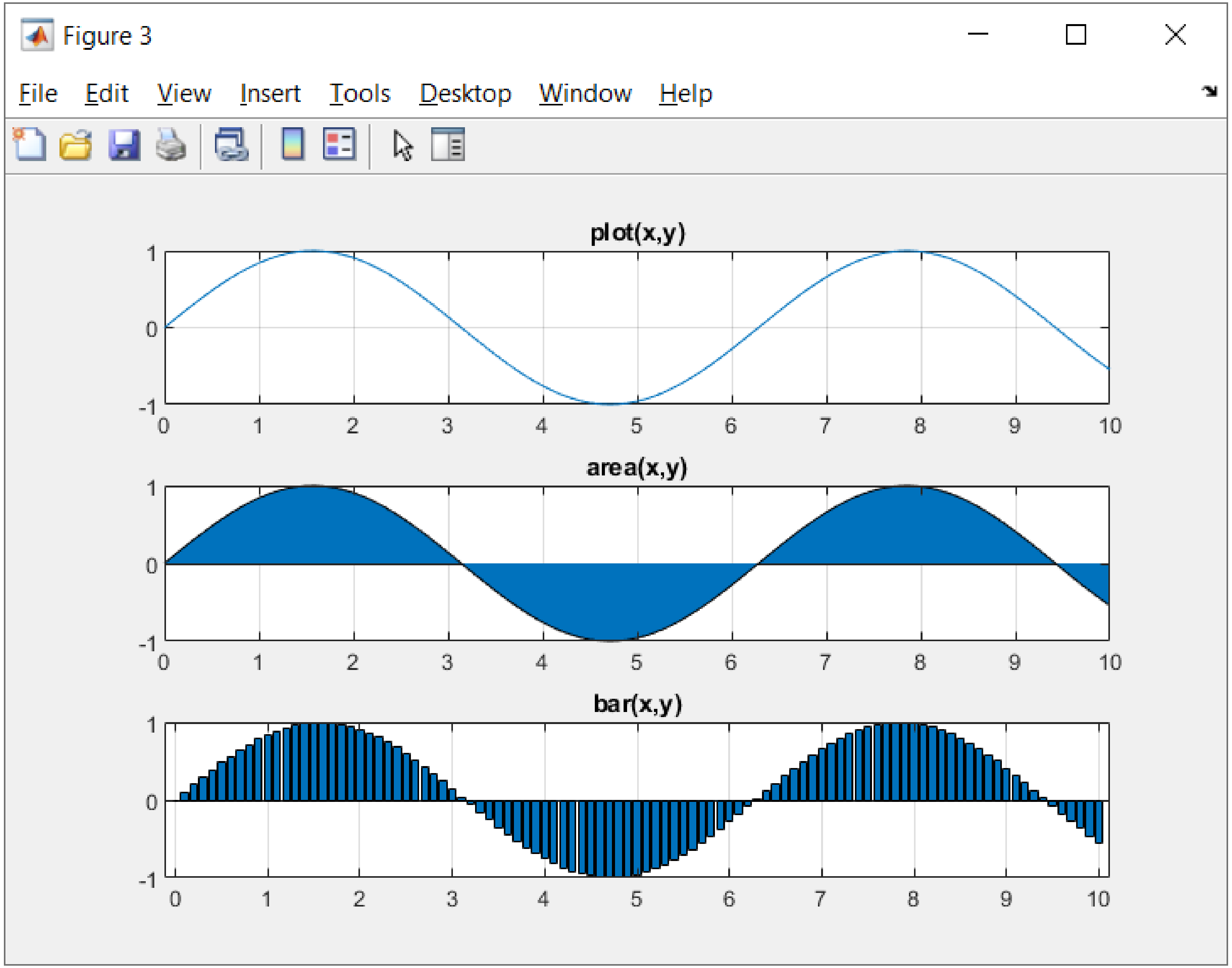


ural





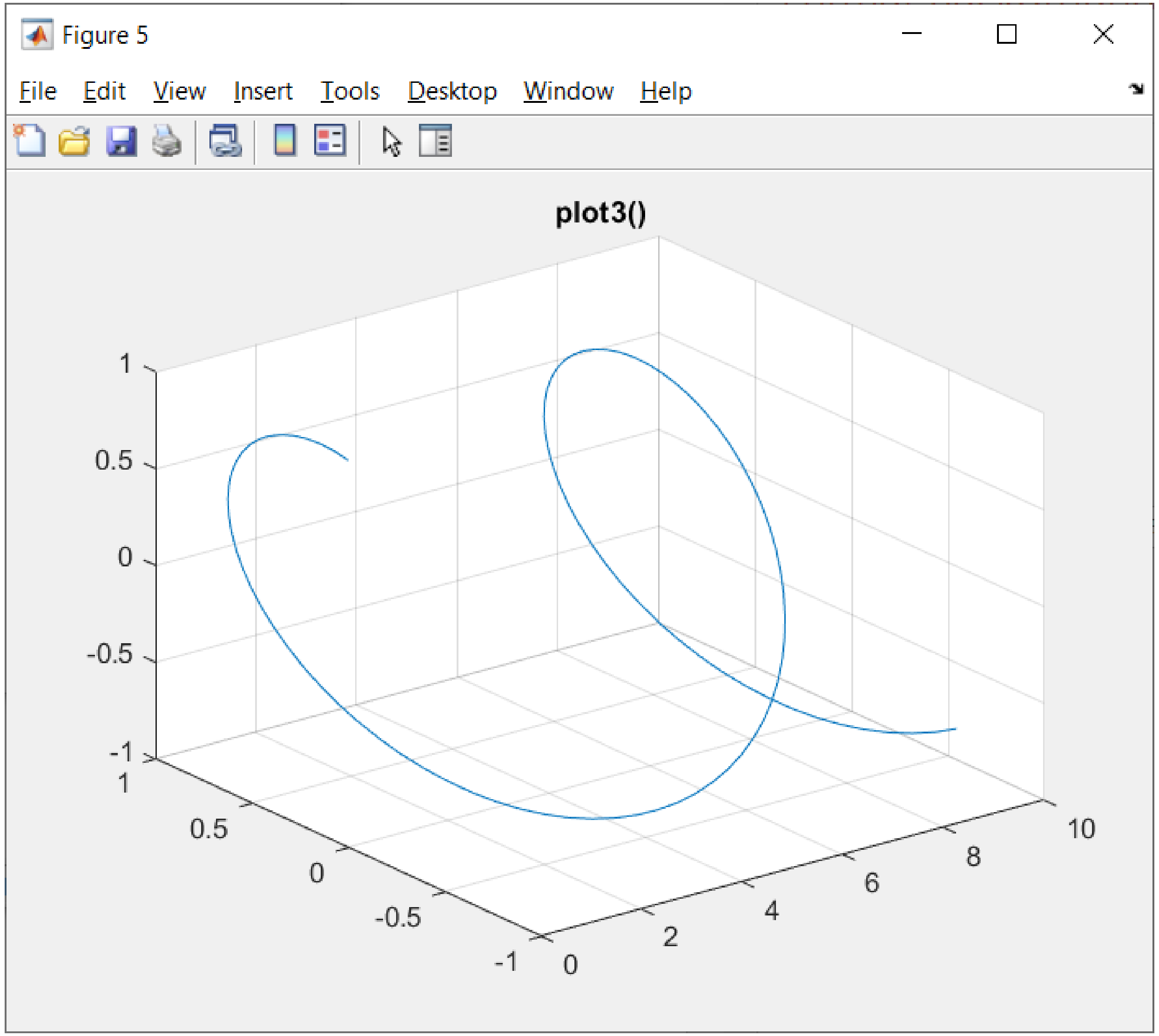
```
x=0:0.1:10;
y=sin(x);
z=cos(x);
figure(3)

subplot(3,1,1)
plot(x,y)
grid on
title('plot(x,y)')

subplot(3,1,2)
area(x,y)
grid on
title('area(x,y)')

subplot(3,1,3)
bar(x,y)
grid on
title('bar(x,y)')
```



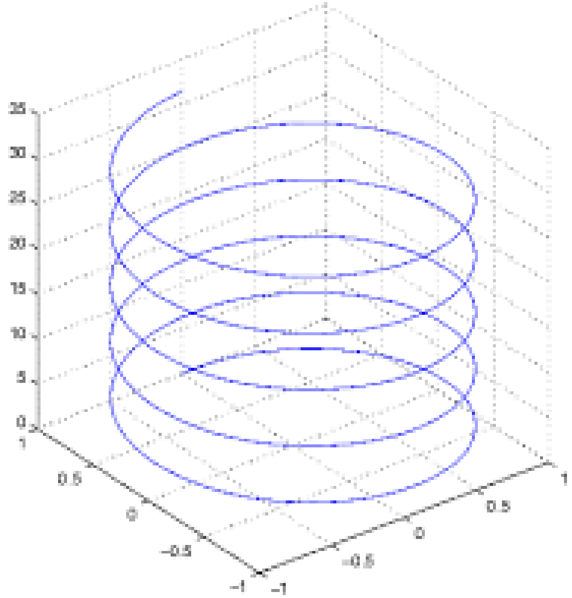
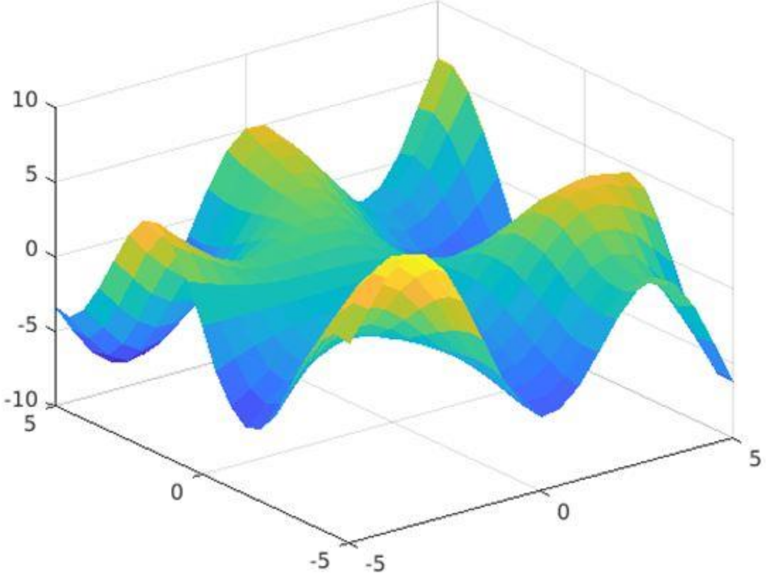
Three-dimensional (3D) graphing functions

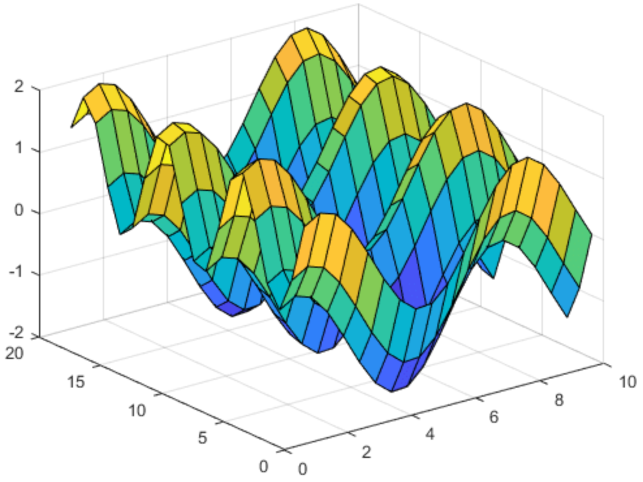
- ***plot3()*** is a three-dimensional analogue of plot() and can be used to plot lines and points in 3-D space.
- ***plot3(x,y,z)***, plots a line in 3-space through the points whose coordinates are the elements of x, y and z, where x, y and z are three vectors of the same length.

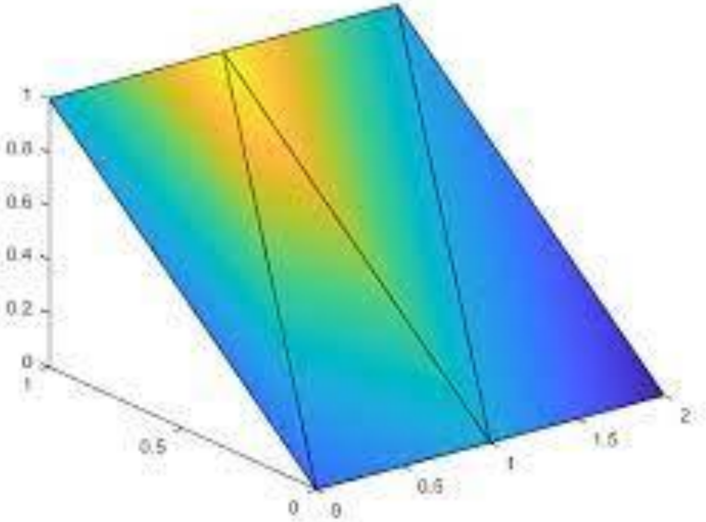
```
x=0:0.1:10;
y=sin(x);
z=cos(x);
figure(3)

figure(5)
plot3(x,y,z)
grid on
title('plot3()')
```



- ***plot3()***: Plots lines and points in 3-D space
- ***mesh()***: creates a 3-D mesh surface
- ***surf()***: 3-D colored surface
- ***fill3()***: Filled 3-D polygons



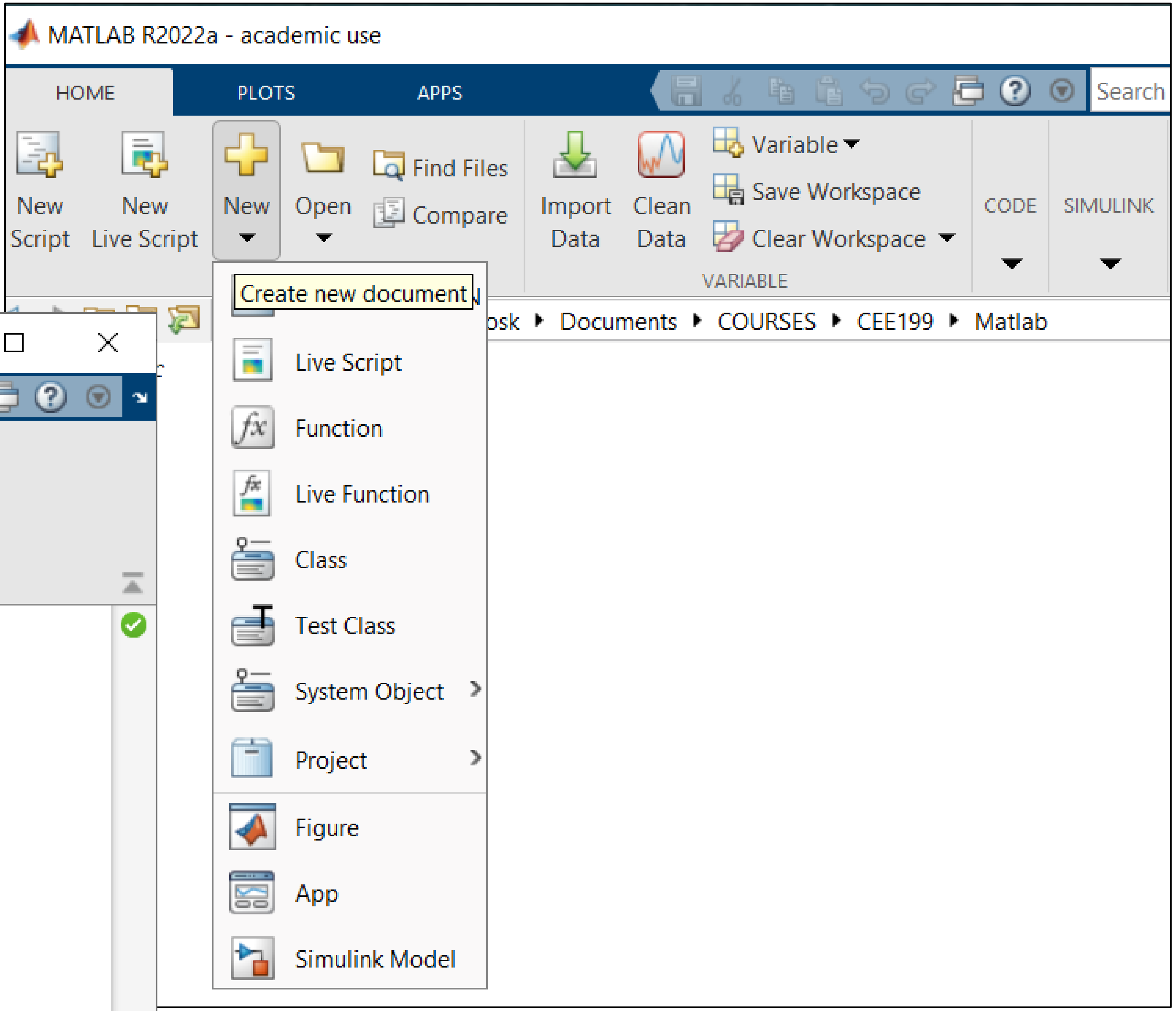


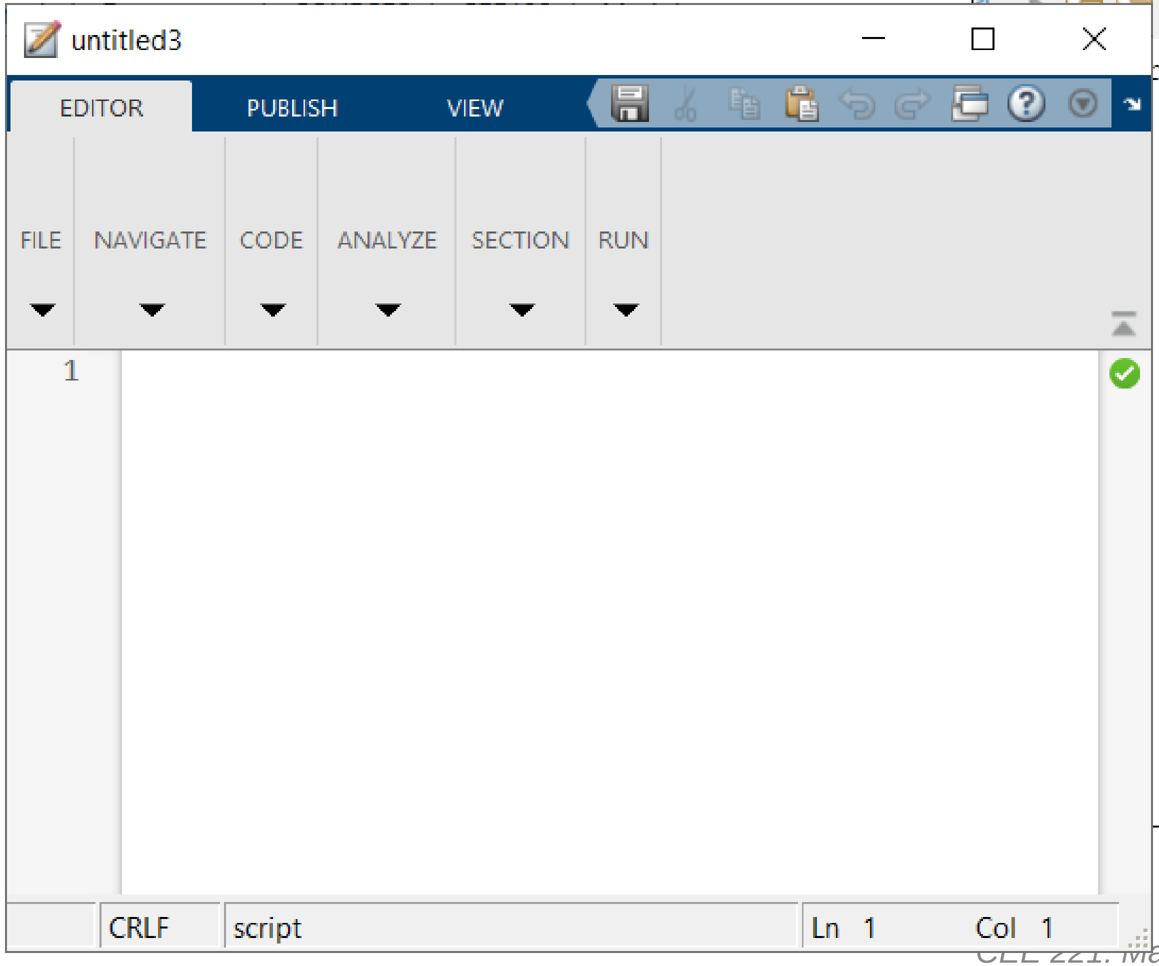
Matlab scripts

- A Matlab script is a sequence of Matlab commands, which are stored in script files, ending in .m extension and executed by typing its name on the command prompt.
- Specifically, by typing the name of a Matlab script at the command prompt, the commands that are contained in the script are sequentially executed, as if they had been directly typed in the command prompt of Matlab.
- Using scripts allows us to be much more productive and efficient, by storing, debugging, reusing and extending series of Matlab commands.
- Instead of typing dozens of commands directly, on Matlab's command prompt, to find out a minor mistake at a command, which would require to start from the beginning, rewriting all previously typed commands, using a Matlab script would require a minor correction and simple reexecution of the script.

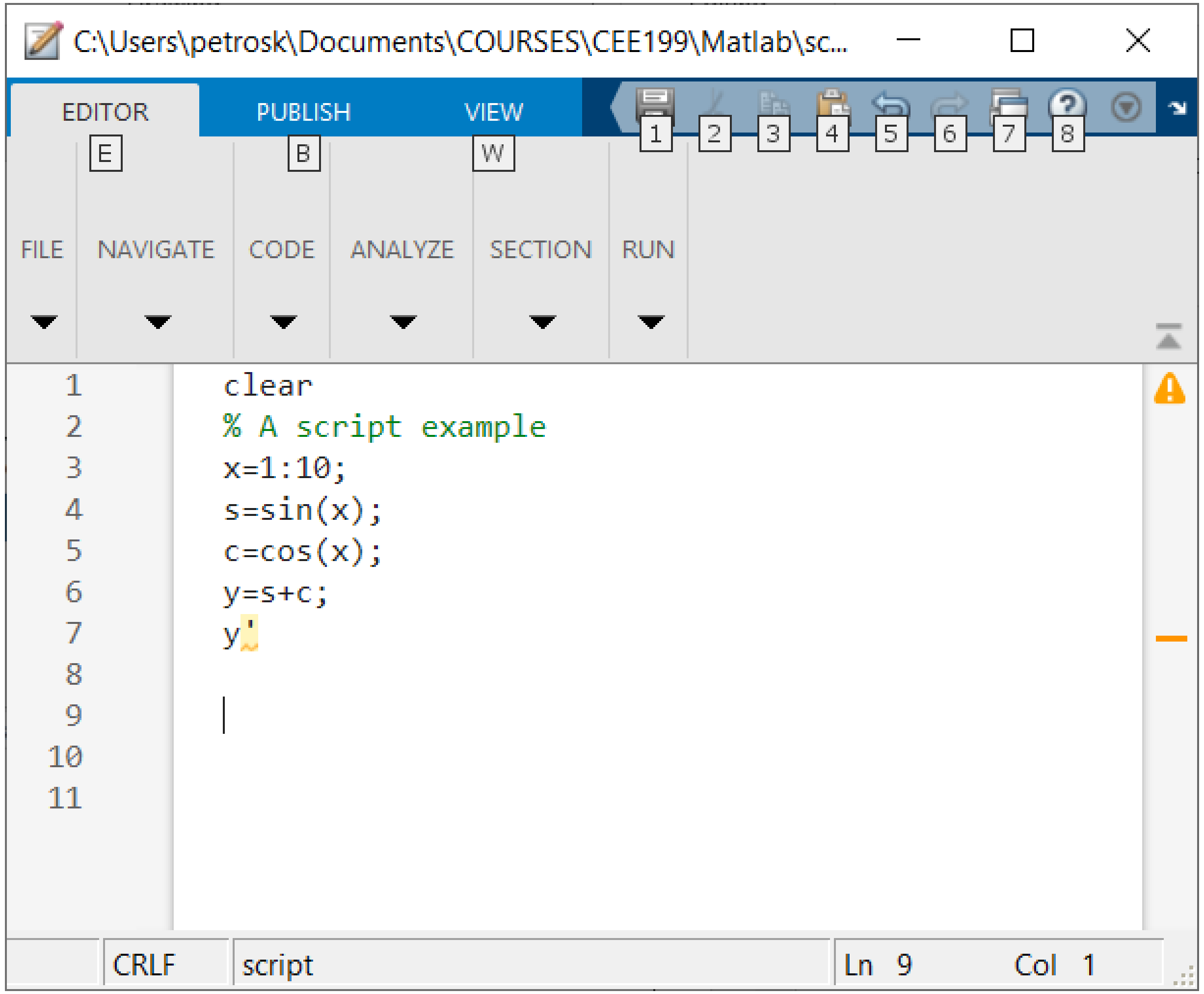
- Script files have a .m filename extension.
- Such files are called M-files and can be either Matlab scripts or Matlab functions.
- A Matlab script is a sequence of Matlab commands that can be executed by typing the name of the script, while a Matlab function is called by its name followed by parentheses, usually containing arguments that can be sent to the function from the point where the function is called (invoked).
- Matlab’s editor can be utilized to more effectively develop, debug, reuse and extend M-files (Matlab scripts and Matlab functions).
- In order to be able to execute a Matlab script, it has to be either located in the currently active directory (folder) or in a file that is contained in the PATH of Matlab.
- The *path* command path prints Matlab's current search path.

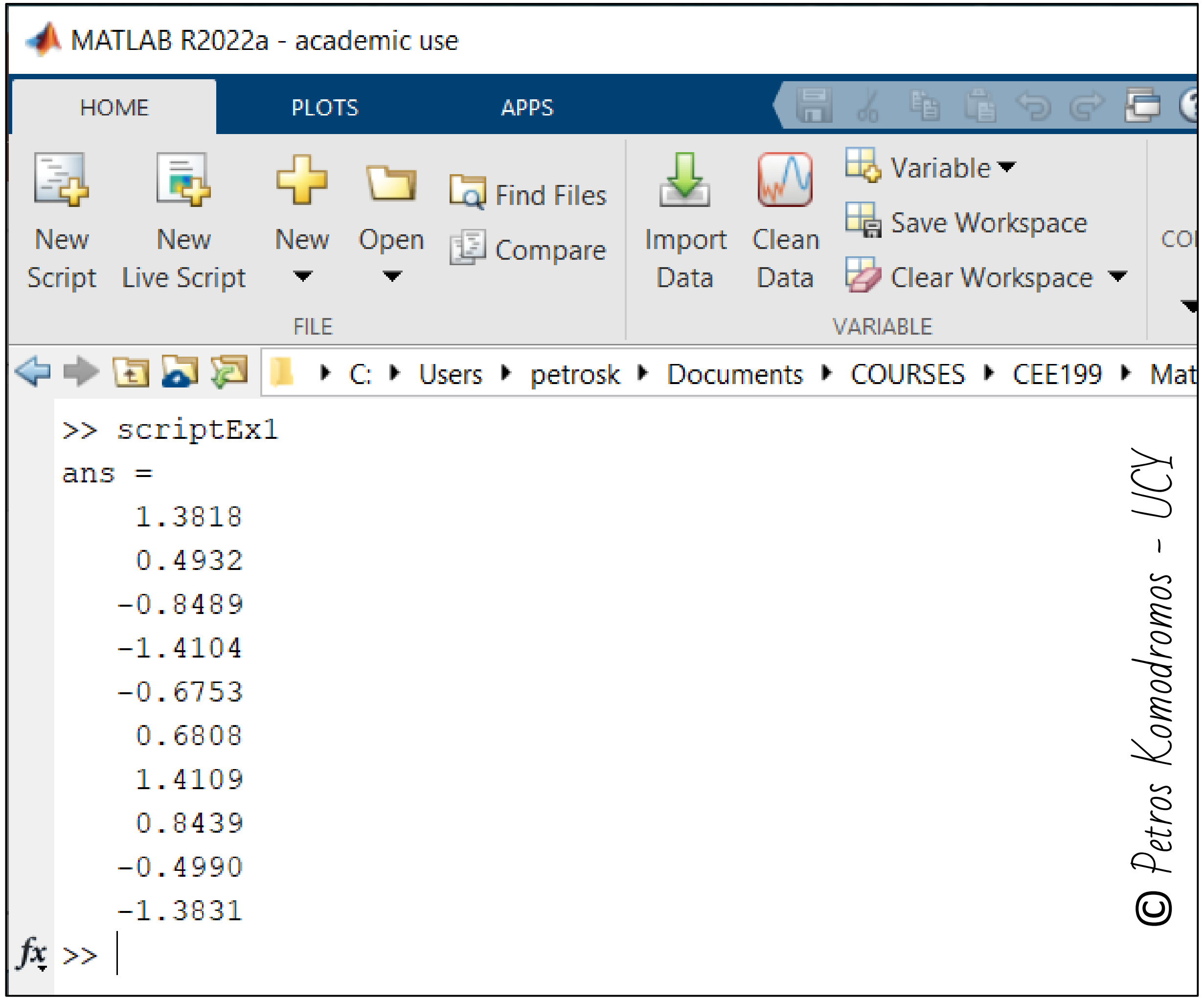
- Use the M-script editor of Matlab by clicking +New and then selecting script:

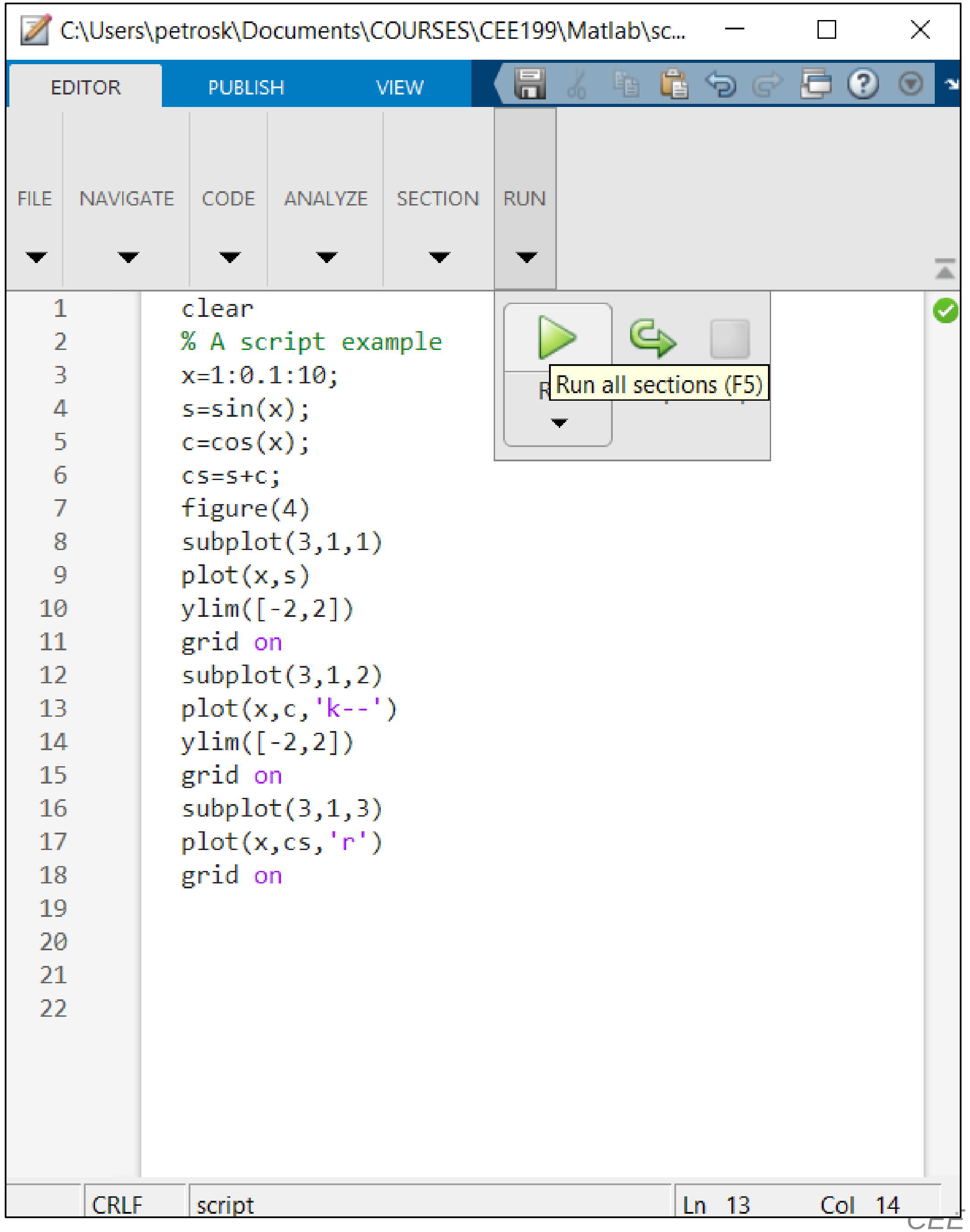




- Write the Matlab commands in the script file, which should be saved either in the current (active) directory (folder) or in a directory (folder) that is included in the PATH of Matlab.






C:\Users\petrosk\Documents\COURSES\CEE199\Matlab\sc...
EDITOR
PUBLISH
VIEW
FILE
NAVIGATE
CODE
ANALYZE
SECTION
RUN
Run all sections (F5)
clear
% A script example
x=1:0.1:10;
s=sin(x);
c=cos(x);
cs=s+c;
figure(4)
subplot(3,1,1)
plot(x,s)
ylim([-2,2])
grid on
subplot(3,1,2)
plot(x,c,'k--')
ylim([-2,2])
grid on
subplot(3,1,3)
plot(x,cs,'r')
grid on
CRLF
script
Ln 13
Col 14

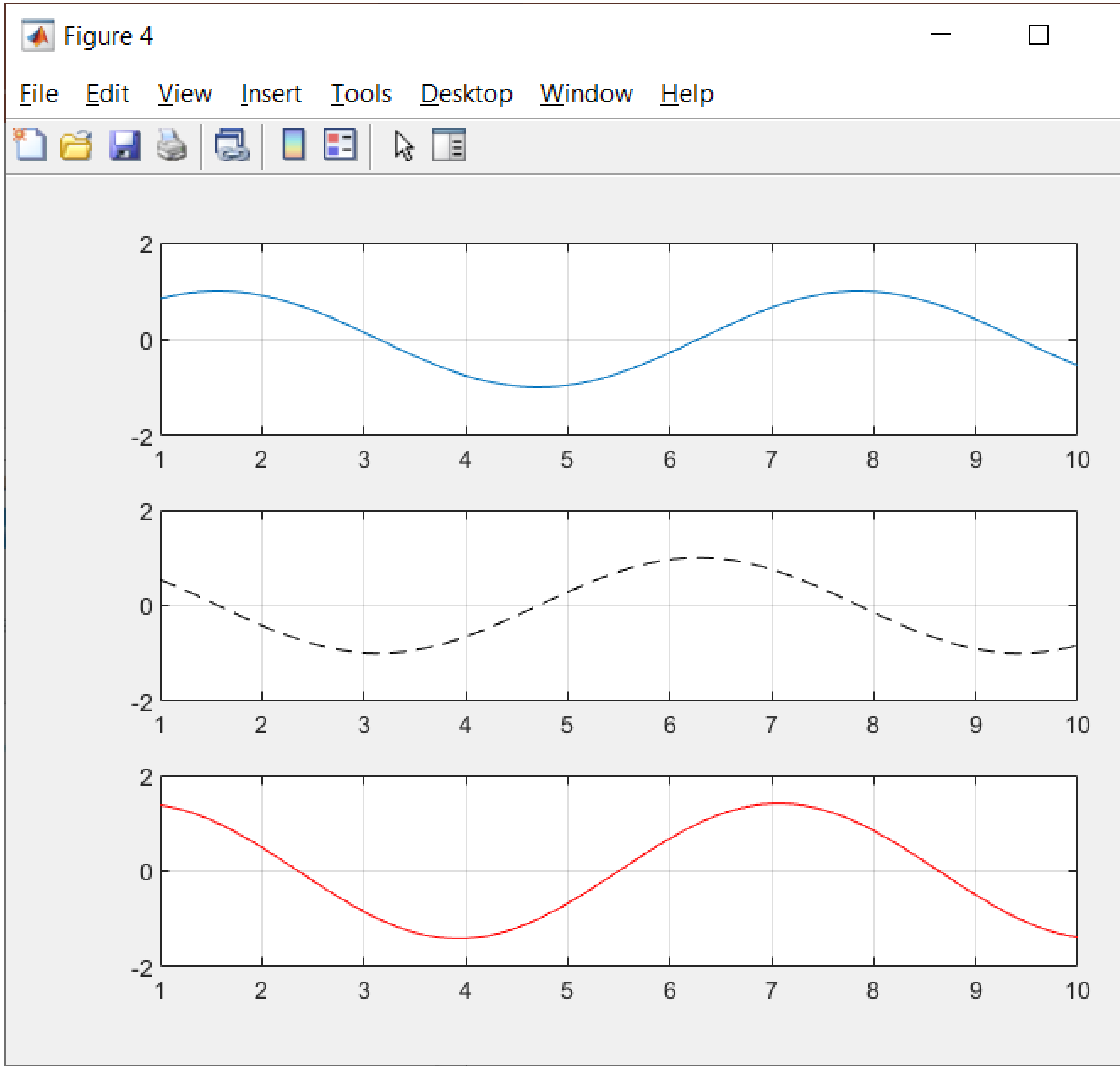


Figure 4
File
Edit
View
Insert
Tools
Desktop
Window
Help
2
0
-2
1 2 3 4 5 6 7 8 9 10
2
0
-2
1 2 3 4 5 6 7 8 9 10
2
0
-2
1 2 3 4 5 6 7 8 9 10

Matlab functions

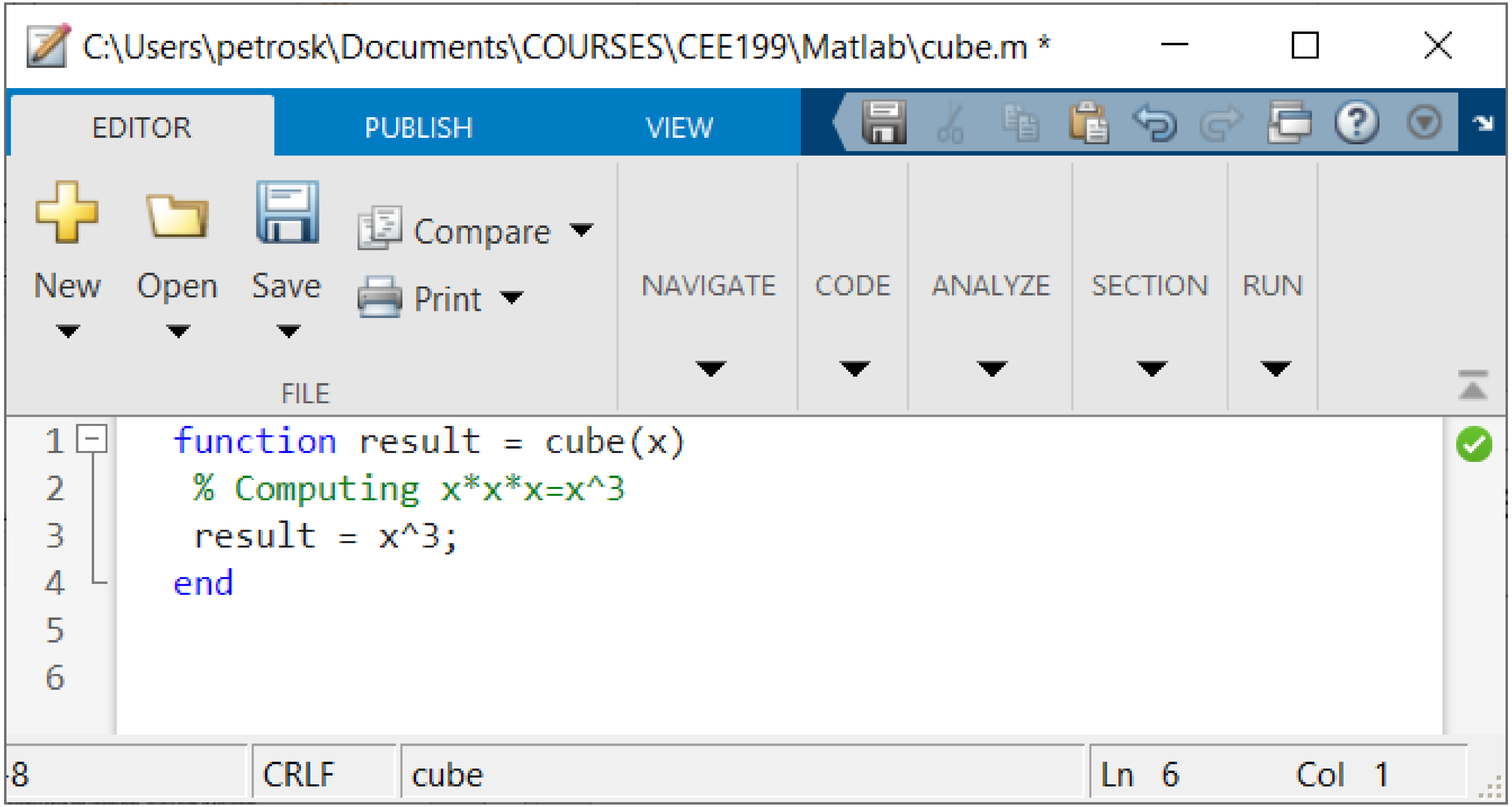
- Matlab has several predefined functions, which can be used such as sin(), cosd(), sqrt(), rand(), etc.
- In addition, Matlab allows the users to define their own functions.
- A function is a sequence of Matlab commands grouped together as a subroutine that can be called (invoked), usually accepting input arguments and returning computed results.
- Functions allow the users to reuse the code frequently.
- Each function has its own area of memory workspace, which is separated from the memory workspace that is used by other functions. Any variables that you create within a function are stored within a workspace specific to that function, which is separate from the base workspace.
- The first line of a function should start with the keyword *function* followed by the name of the variable/variables whose values is/are returned to the point where the function is invoked, the function name and parentheses, usually order of arguments.

- **Syntax**:

```
function returningParameters = functionName(inputParameters)
    % Statements
end
```

- Use lowercase characters for the keyword *function*.
- The name of a function must coincide with the name of the file where the function is stored.
- A Matlab function can return more than one value, enclosed in square brackets.
- Functions provide much more flexibility than scripts, because they accept the values of parameters (arguments), that are used in parentheses while calling functions and return computed values.
- Functions end with either an ***end*** statement, which although it is sometimes optional, its usage increases the code readability.

- Function accepting one parameter and returning one value




```
>> clear
>> x=7;
>> a=3;
>> z=cube(a);
>> z
z =
    27
>> x
x =
     7
>> cube(4)
ans =
    64
>> cube(2)
ans =
     8
```

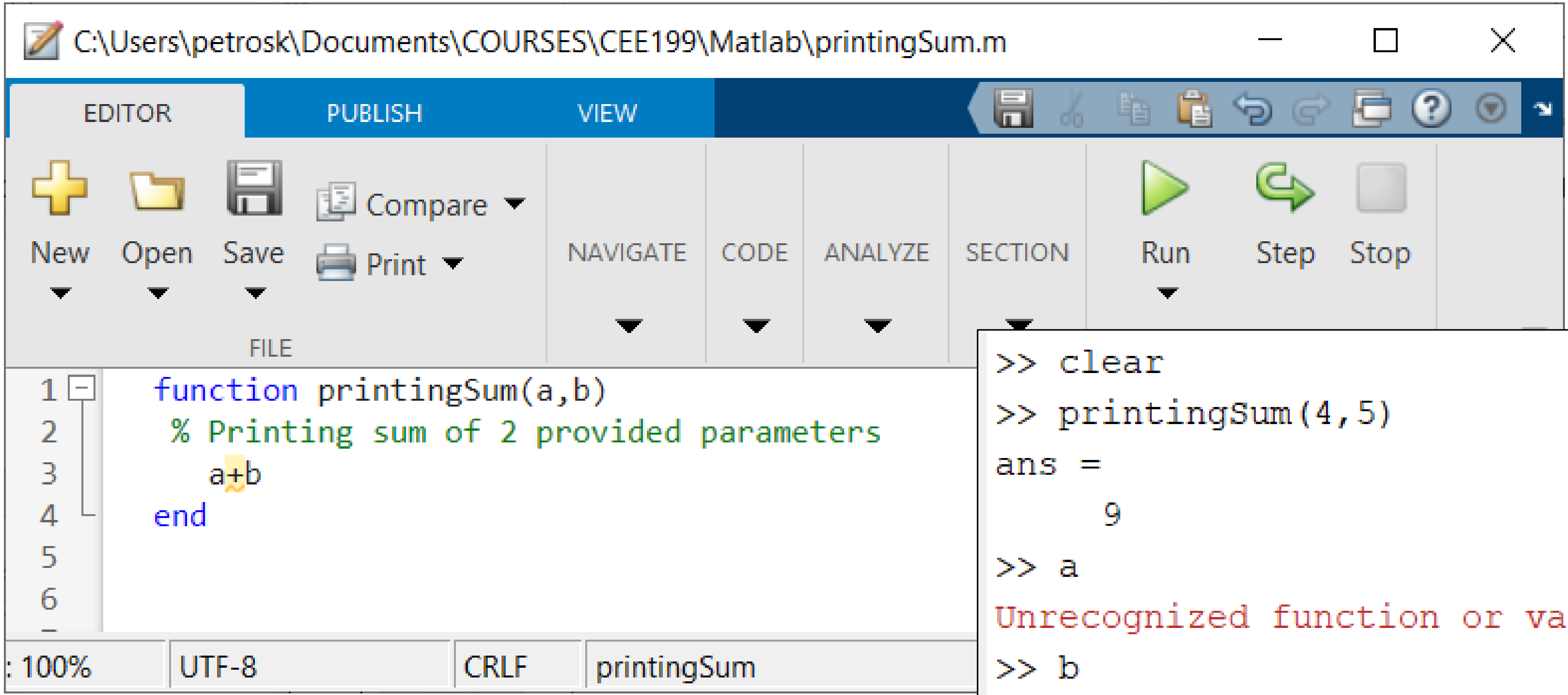
- Function accepting more than one parameters and returning one value

C:\Users\petrosk\Documents\COURSES\CEE199\Matlab\adding3Numbers.m

```matlab
function res = adding3Numbers(x,y,z)
 % Adding x, y and z
 res = x+y+z;
end
```

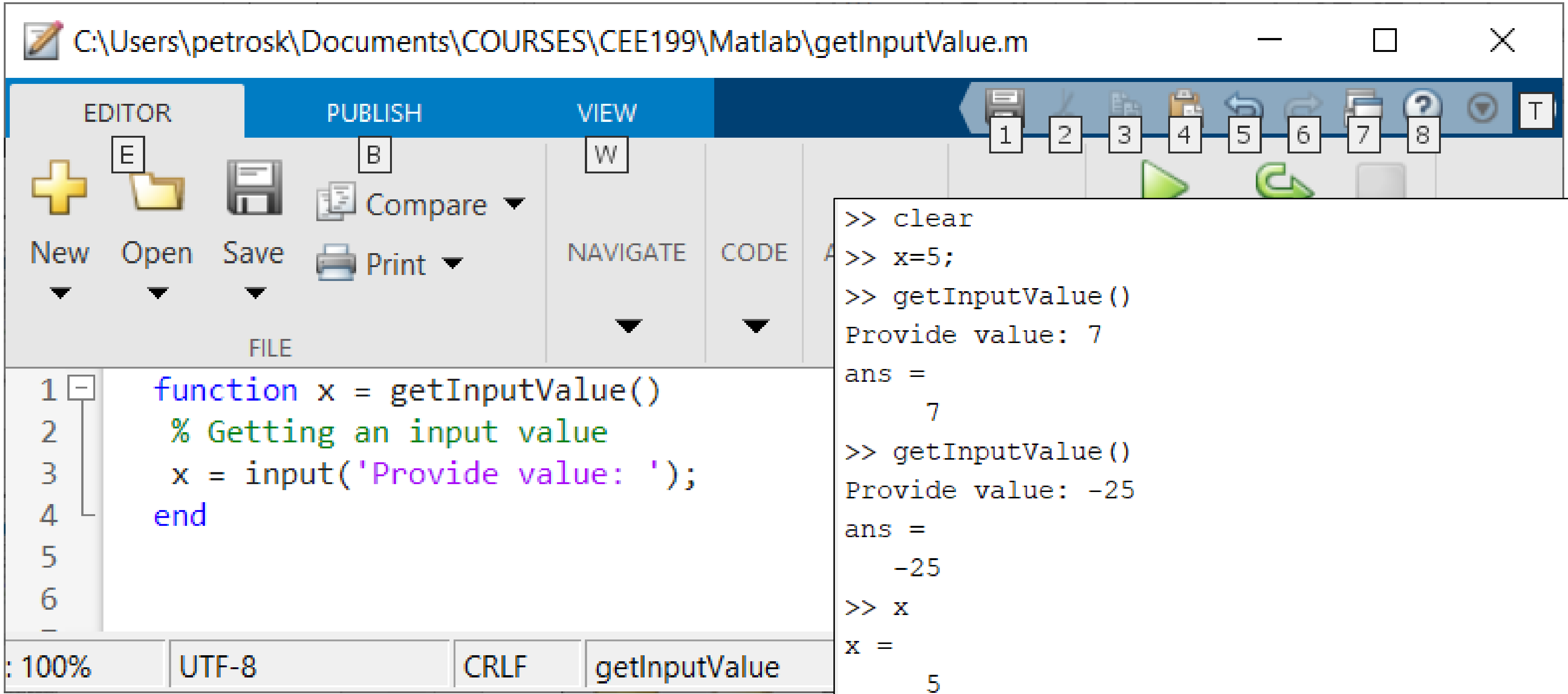
```
>> clear
>> adding3Numbers(3,2,6)
ans =
     11
>> x=5;
>> y=7;
>> z=adding3Numbers(4,y,100)
z =
   111
>> x
x =
     5
>> y
y =
     7
>> z
z =
   111
```

- Function accepting more than one parameters without returning any value




```
>> clear
>> printingSum(4,5)
ans =
     9
>> a
Unrecognized function or variable 'a'.
>> b
Unrecognized function or variable 'b'.
>> printingSum(10,25)
ans =
    35
```

- Function without any parameters, returning one value




```
>> clear
>> x=5;
>> getInputValue()
Provide value: 7
ans =
     7
>> getInputValue()
Provide value: -25
ans =
   -25
>> x
x =
     5
>> whos
  Name      Size            Bytes  Class     Attributes

  ans       1x1                 8  double
  x         1x1                 8  double
```

- Function accepting more than one parameters and returning more than one values

C:\Users\petrosk\Documents\COURSES\CEE199\Matlab\returnCubes.m

EDITOR | PUBLISH | VIEW

New | Open | Save | Compare | Print | FILE | NAVIGATE | CODE | ANALYZE | SECTION

```matlab
function [x3, y3] = returnCubes(x,y)
 % Returing the cubes of the provided parameters
 x3 = x^3;
 y3 = y^3;
end
```

: 100% | UTF-8 | CRLF | returnCubes

```
>> clear
>> x=6;
>> y=17;
>> [a,b] = returnCubes(3,4)
a =
    27
b =
    64
>> x
x =
     6
>> a
a =
    27
>> b
b =
    64
>> y
y =
    17
```

- Using both a script and a function file:

C:\Users\petrosk\Documents\COURSES\CEE199\Matlab\...

```
1   clear
2
3   x3=13;
4   y3=20;
5   x=8;
6
7   [a,b]=fun1(12,33)
8
9   a
10  b
11
12  x3
13  y3
14
15  x
16  y
```

C:\Users\petrosk\Documents\COURSES\CEE199\Matlab\fun1.m

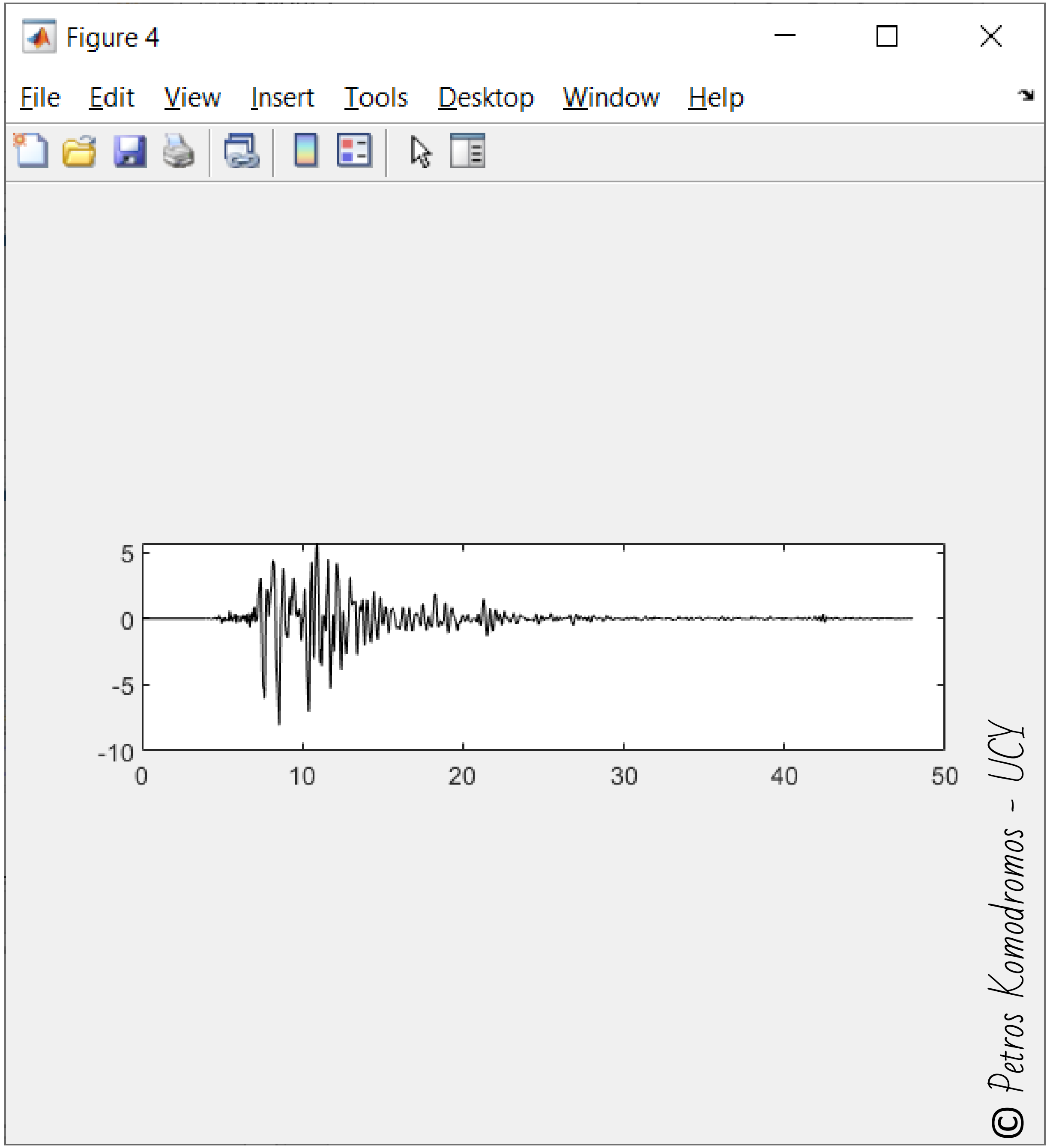
```
function [x3, y3] = fun1(x,y)
 % Returing the cubes of the provided parameters
 x3 = x^3;
 y3 = y^3;
end
```

```
>> clear
>> testFun1
a =
        1728
b =
       35937
a =
        1728
b =
       35937
x3 =
    13
y3 =
    20
x =
     8
Unrecognized function or variable 'y'.
Error in testFun1 (line 16)
y
```

Using data files with Matlab

- ***load('fileName')*** loads data from the file named *fileName*.
  - If *filename* is an ASCII file, it creates a double-precision array, named *filename,* containing data from the file.
  - ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (character between each element in a row) can be a blank, comma, semicolon, or tab, while the file can contain Matlab comments, as well.
  - If the *fileName* is a MAT-file, it loads variables that had been saved in the MAT-file into the Matlab workspace.
- Alternatively, on the *Home tab*, in the *Variable section*, click *Import Data*.
  - Select the file from the recognized data files, e.g. text, spreadsheet file, etc.

```
>> clear
>> load kobeAccel
>> whos
  Name               Size            Bytes  Class

  kobeAccel       2401x2             38416  double

>> t=kobeAccel(:,1);
>> ag=kobeAccel(:,2);
>> whos
  Name               Size            Bytes  Class

  ag              2401x1             19208  double
  kobeAccel       2401x2             38416  double
  t               2401x1             19208  double

>> figure(4)
>> subplot(3,1,2)
>> plot(t,ag,'k')
```



Saving & Loading Matlab files

- ***save 'fileName'*** saves all variables currently in the workspace in a Matlab (.mat) file with the specific *fileName*.
- ***save 'fileName' variables*** saves the specified *variables* in a Matlab (.mat) file with the specific *fileName*.

- ***load 'fileName'*** loads all variables that have been saved in the Matlab (.mat) file named *fileName*.
- ***load 'fileName' variables*** loads the specified saved *variables* from the Matlab (.mat) file named *fileName*.

```
>> clear
>> x = 1:5;
>> y = -45.2;
>> z = 27*y/40;
>> who

Your variables are:

x   y   z

>> save save1 x z
>> clear
>> who
```

```
>> who
>> load save1 x
>> who

Your variables are:

x

>> load save1
>> who

Your variables are:

x   z
```

Relational operators

- Relational operators compare, element by element, the elements in two arrays (which usually are scalars, e.g. of 1x1 size) and return logical true or false values to indicate where the relation holds.
- They return a logical array of the same size, with elements set to true (1) where the relation is true, and elements set to false (0) where it is not.

> < >= <= == ~=

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

```
>> a=3;
>> b=-7;
>> if (a>b)
maxAB=a;
else
maxAB=b;
end
>> maxAB
maxAB =
      3
```

==	Determine equality
>=	Determine greater than or equal to
>	Determine greater than
<=	Determine less than or equal to
<	Determine less than
~=	Determine inequality

```
>> clear
>> x = [ 2 6 3 8 -4];
>> y = [ 2 4 3 8 4];
>> x == y
ans =
  1×5 logical array
   1    0    1    1    0
```

```
>> clear
>> a=4;
>> b=6;
>> c=8/2;
>> a==b
ans =
  logical
   0
>> a==c
ans =
  logical
   1
>> b==c
ans =
  logical
   0
```

```
>> a=3;
>> b=-7;
>> if (a>b)
maxAB=a;
else
maxAB=b;
end
>> maxAB
maxAB =
     3
```

Logical operations and expressions

- Element-wise logical operators operate element-by-element on logical arrays, using the symbols &, |, and ~, which are the logical array operators AND, OR, and NOT.
- Short-circuit logical operators allow short-circuiting on logical operations, using the symbols && and ||, which are the logical short-circuit operators AND and OR.
- When the evaluation of a logical expression terminates early by encountering a decisive value, the expression is said to have short-circuited.
    - For example, in the expression A && B , Matlab does not evaluate condition B at all if condition A is false, since there is no point.

& | ~ && ||

Logical AND	&
Logical OR	|
Logical NOT	~
Short-circuit logical AND	&&
Short-circuit logical OR	||

```
>> clear
>> x=3;
>> y=7;
>> z=4;
>> x<z
ans =
  logical
   1
>> x>z
ans =
  logical
   0
>> x+z==y
ans =
  logical
   1
```

```
>> x+z>=y
ans =
  logical
   1
>> x+z<=y
ans =
  logical
   1
>> x>0 && y>z+2
ans =
  logical
   1
```

```
>> x<2 || y<z
ans =
  logical
   0
>> x<2 || y<z+x
ans =
  logical
   0
>> x<2 || y<=z+x
ans =
  logical
   1
>> ~(x>2)
ans =
  logical
   0
```

```
>> ~(x>y)
ans =
  logical
   1
>> x>2 || y<=z+x
ans =
  logical
   1
>> x>2 | y<=z+x
ans =
  logical
   1
```

if/else selection control structure

```
if expression1
    command/s executed if expression1 is true
end
```

```
if expression1
    command/s executed if expression1 is true
else
    command/s executed if expression1 is not true
end
```

```
if expression1
    command/s executed if expression1 is true
elseif expression2
    command/s executed if expression2 is true
else
    command/s executed if none of the previous expressions is true
end
```

- The *elseif* and *else* blocks are optional.
- The statements execute only if previous expressions in the *if...end* block are false.
- An if block can include multiple *elseif* blocks.

```
r =
    0.7244
>> if r < 0.25
    disp('r < 0.25')
elseif r < 0.5
    disp('0.25 <= r < 0.5')
elseif r < 0.75
    disp('0.5 <= r < 0.75')
else
    disp('0.75 <= r ')
end
0.5 <= r < 0.75
```

```
r =
    0.1199
>> if r < 0.25
    disp('r < 0.25')
elseif r < 0.5
    disp('0.25 <= r < 0.5')
elseif r < 0.75
    disp('0.5 <= r < 0.75')
else
    disp('0.75 <= r ')
end
r < 0.25
```

```
>> r = rand(1)
r =
    0.2969
>> if r < 0.25
    disp('r < 0.25')
elseif r < 0.5
    disp('0.25 <= r < 0.5')
elseif r < 0.75
    disp('0.5 <= r < 0.75')
else
    disp('0.75 <= r ')
end
0.25 <= r < 0.5
```

switch selection control structure

```
switch switchExpression
   case caseExpression
      statements
   case caseExpression
      statements
  ...
   otherwise
      statements
end
```

- The ***switch*** selection control structure evaluates the specified *switchExpression* and chooses to execute the case statement or group of statements with the specified *caseExpression* that has the same value, arithmetic or character-wise.
- When a *caseExpression* is true, Matlab executes the corresponding statements and exits the switch block.
- The statements in the *otherwise* block, which is optional, are executed only when no case has been found to be true.

```
x =
     7
>> switch x
    case 4
        y1 = x
    case 3
        y3 = x+3
    case 7
        y7 = x+777
    otherwise
        zz = x^2
end
y7 =
   784
```

```
>> x=4
x =
     4
>> switch x
    case 4
        y1 = x
    case 3
        y3 = x+3
    case 7
        y7 = x+777
    otherwise
        zz = x^2
end
y1 =
     4
```

```
x =
   100
>> switch x
    case 4
        y1 = x
    case 3
        y3 = x+3
    case 7
        y7 = x+777
    otherwise
        zz = x^2
end
zz =
       10000
```

for loop (iterative control structure)

- The ***for*** loop (iterative control structure) changes (increases or decreases) the *initialValue* by the *optionalStep* (otherwise, by 1) until the value of the *variableIndex* is greater than *finalValue*.

```
for variableIndex = initialValue : optionalStep : finalValue
    statements
end
```

- ***continue*** passes control to the next iteration, of a (for/while), loop, in which it appears, skipping any remaining statements in the body of the loop.
- ***break*** terminates the execution of a (for/while), loop, in which it appears. In nested loops, break exits from the innermost loop only.

```
for i=1:5
    disp(i^2)
end
```

```
     1
     4
     9
    16
    25
```

```
for i=5:-1:1
  if  i == 3
    continue
  end
  disp(i^2)
end
```

```
    25
    16
     4
     1
```

```
clc
Clear

n=5;
sumOfSquares=0

for(i=1:n)
    sumOfSquares = sumOfSquares + i^2;
end

fprintf('Sum of squares of numbers up to %d = %d \n', n, sumOfSquares);
```

```
sumOfSquares = 0
Sum of squares of numbers up to 5 = 55
```

while loop (iterative control structure)

- The ***while*** loop (iterative control structure) iteratively executes the statements as long as the *logicalExpression* is true.

```
while logicalExpression
    statements
end
```

  - ***continue*** passes control to the next iteration, of a (for/while), loop, in which it appears, skipping any remaining statements in the body of the loop.
  - ***break*** terminates the execution of a (for/while), loop, in which it appears. In nested loops, break exits from the innermost loop only.

```
i = 0;
 while i<10
   i = i + 2
 end
 disp(i)
```

```
i = 0;
 while 1
   if i>=10
      break
   end
   i = i + 2
 end
 disp(i)
```

```
2
4
6
8
10
10
```

```
clc
clear
n=5;
sumOfSquares=0
i=1;
while i<=n
    sumOfSquares = sumOfSquares + i^2;
    i=i+1;
end

fprintf('Sum of squares of numbers up to %d = %d \n', ...
    n, sumOfSquares);
```

```
sumOfSquares = 0
Sum of squares of numbers up to 5 = 55
```

Control Structures - Overall

if: Conditionally executes statements.

else: Executes statement if previous *if* condition failed.

elseif: Executes if previous *if* failed and condition is true.

end: Terminates scope of control statements.

switch: Switch among several cases based on expression.

case: *switch* statement case.

otherwise: Default *switch* statement case.

- ***for***: Repeats statements a specific number of times.
- ***while***: Repeats statements an indefinite number of times as long a s the logical test is true.
- ***break***: Terminates execution of a *while* or *for* loop.
- ***continue***: Passes control to the next iteration of a loop.

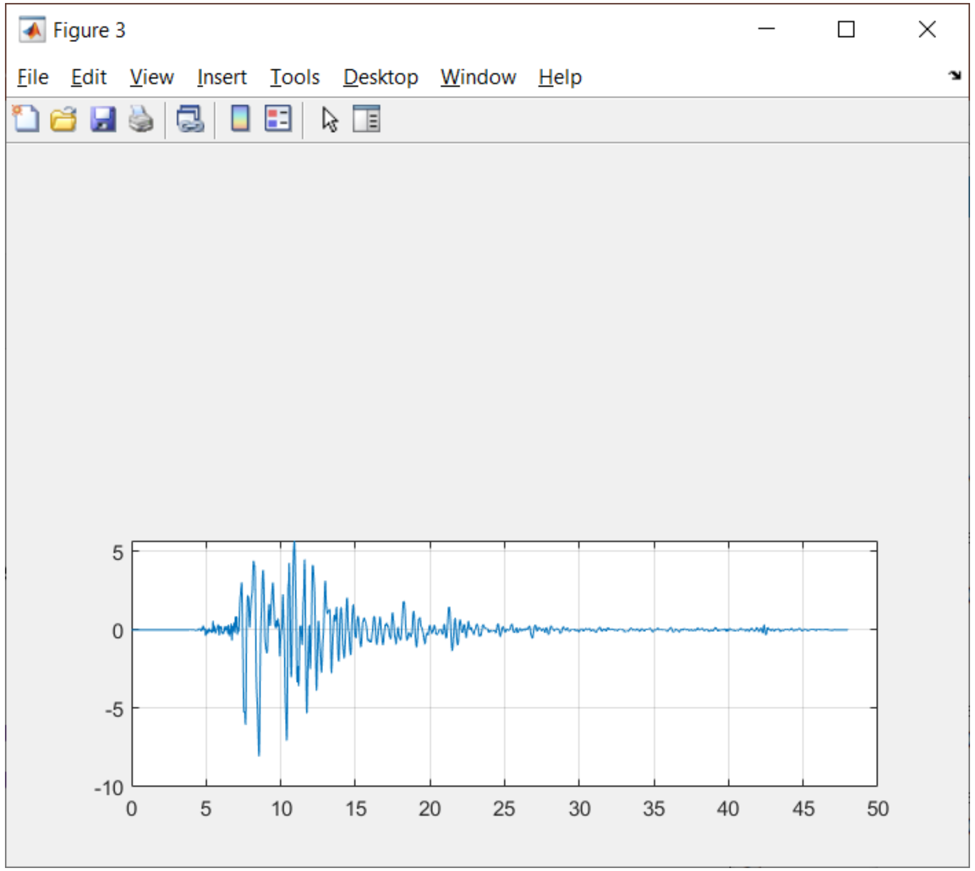
More commands and functionalities

- The function ***eval()*** evaluates (i.e. executes) the Matlab expression provided in text format, as an argument.
  - Security Considerations: When calling *eval* with untrusted user input, validate the input to avoid unexpected code execution.
- The function ***sprintf(')*** writes formatted data to string or character vector, according to the provided format by the control string, similarly to the way *printf()* works.

```
>> x=4/3
x =
    1.3333
>> y=13/2
y =
    6.5000
>> sC = sprintf(' x = %f  y = %f ', x, y)
sC =
    ' x = 1.333333  y = 6.500000 '
```

```
>> clear
>> s='2*3-5+3/2'
s =
    '2*3-5+3/2'
>> eval(s)
ans =
    2.5000
>> s1='x=7/3+4'
s1 =
    'x=7/3+4'
>> s
s =
    '2*3-5+3/2'
>> s1
s1 =
    'x=7/3+4'
>> eval(s1)
x =
    6.3333
```

```
>> clear
>> fileName = input('Accelogram fileName: ','s');
Accelogram fileName: kobeAccel
>> commandToExecute = sprintf('load %s',fileName);
>> eval(commandToExecute)
>> who

Your variables are:

commandToExecute  fileName          kobeAccel

>> commandToExecute = sprintf('t = %s(:,1);',fileName);
>> eval(commandToExecute)
>> commandToExecute = sprintf('ag = %s(:,2);',fileName);
>> eval(commandToExecute)
>> figure(3)
subplot(2,1,2)
plot(t,ag)
grid on
```



who - List current variables.

whos - List current variables, long form.

clear - Clear variables and functions from memory.

load - Load workspace variables from disk.

save - Save workspace variables to disk.

quit - Quit MATLAB session.

exit - Exit from MATLAB.

```
>> realmax
ans =
  1.7977e+308
>> realmin
ans =
  2.2251e-308
```

what - List MATLAB-specific files in directory.

type - List M-file.

open - Open files by extension.

which - Locate functions and files.

Timing commands/functions

now - Current date and time as date number.

date - Current date as date string.

clock - Current date and time as date vector.

datestr - String representation of date.

calendar - Calendar.

cputime - CPU time in seconds.

tic - Start stopwatch timer.

toc - Stop stopwatch timer.

pause - Wait in seconds.

```
>> tic
>> calendar
                  Aug 2022
     S      M     Tu      W     Th      F      S
     0      1      2      3      4      5      6
     7      8      9     10     11     12     13
    14     15     16     17     18     19     20
    21     22     23     24     25     26     27
    28     29     30     31      0      0      0
     0      0      0      0      0      0      0
>> now
ans =
   7.3876e+05
>> date
ans =
    '25-Aug-2022'
>> toc
Elapsed time is 16.636599 seconds.
```

Operating system/s commands

cd - Change current working directory.

copyfile - Copy file or directory.

movefile - Move file or directory.

delete - Delete file or graphics object.

pwd - Show (print) current working directory.

dir - List directory.

ls - List directory.

mkdir - Make new directory.

rmdir - Remove directory.

! - Execute operating system command.

Comparison of Matlab with programming languages

- Matlab compared to:
  - Fortran
  - C/C++
  - Java
  - Visual Basic/VB .net
  - C#
  - Python
  - etc.

Selected Matlab references

- Matlab Tutorial for Beginners – 2021 *https://www.youtube.com/watch?v=1PSFLKiEV7U*
- The Complete MATLAB Course: Beginner to Advanced
- Introduction to Matlab Programming for Engineers and Scientists
- INTRODUCTION TO MATLAB FOR ENGINEERING STUDENTS
- Introduction To MATLAB Programming
- MIT 18.S997 Introduction to MATLAB Programming Fall 2011
- An interactive introduction to MATLAB
- Introduction to MATLAB
- Εισαγωγή στη Matlab, Γ. Γεωργίου & Χ. Ξενοφώντος
- Οδηγός Matlab για Αρχάριους, Χρίστος Ξενοφώντος, ΜΑΣ, ΠΚ