Chapter 1. MATLAB

1. Basics of MATLAB
2. MATLAB variables and build-in functions
3. MATLAB script files
4. MATLAB arrays
5. MATLAB two-dimensional and three-dimensional plots
6. MATLAB used-defined functions I
7. MATLAB relational operators, conditional statements, and selection structures I
8. MATLAB relational operators, conditional statements, and selection structures II
9. MATLAB loops
10. Summary

Text : A. Gilat, *MATLAB: An Introduction with Applications*, 4th ed., Wiley

Additional text: H. Moore, *MATLAB for Engineers*, 4th ed., Pearson

1.1. Basics of MATLAB

- MATLAB capabilities
- MATLAB command window and workspace
- MATLAB commands
- MATLAB arithmetic expressions

Reading assignment

Gilat, 1.1 – 1.4

1.1. Basics of MATLAB

MATLAB (**mat**rix **lab**oratory) is a numerical computing environment and fourth-generation programming language.

Developed by *MathWorks*, MATLAB allows for

- Simple computations as a large and "clever" calculator with user-defined variables.
- Vector and matrix manipulations, solving linear algebra problems.
- Numerical solution of many problems of mathematical analysis including interpolation, curves fitting, integration, solution of differential equations, etc.
- Plotting of functions and data.
- Import/export of data from/to other computational tools.
- Symbolic computing.
- Implementation of user-defined functions and algorithms.
- Interfacing with programs written in other languages, including C, C++, Java, and Fortran.

MATLAB also includes many applications (**toolboxes**) for specific problems of data analysis, e.g.

- Signal analysis.
- Image processing.
- Curve fitting etc.

How can we solve engineering problems with MATLAB?

Arithmetic and symbolic calculations	Use of Toolboxes	Programming (development of algorithms)
Arithmetic expressions	Pre-defined algorithms for solving specific engineering problems: Powerful, but limited by available toolboxes Difficult to tune for new problems Example: Simulink	Structuring algorithms : Used-defined functions
Variables and arrays		Making decisions: Selection structures
Build-in functions		Repeating calculations: Loops
Scripts		Data import/export to/from MATLAB
Symbolic mathematics		Versatile: Any computational algorithm written in C/C++ or Fortran can be implemented in MATLAB
Data visualization		

What shall we consider in the classroom?

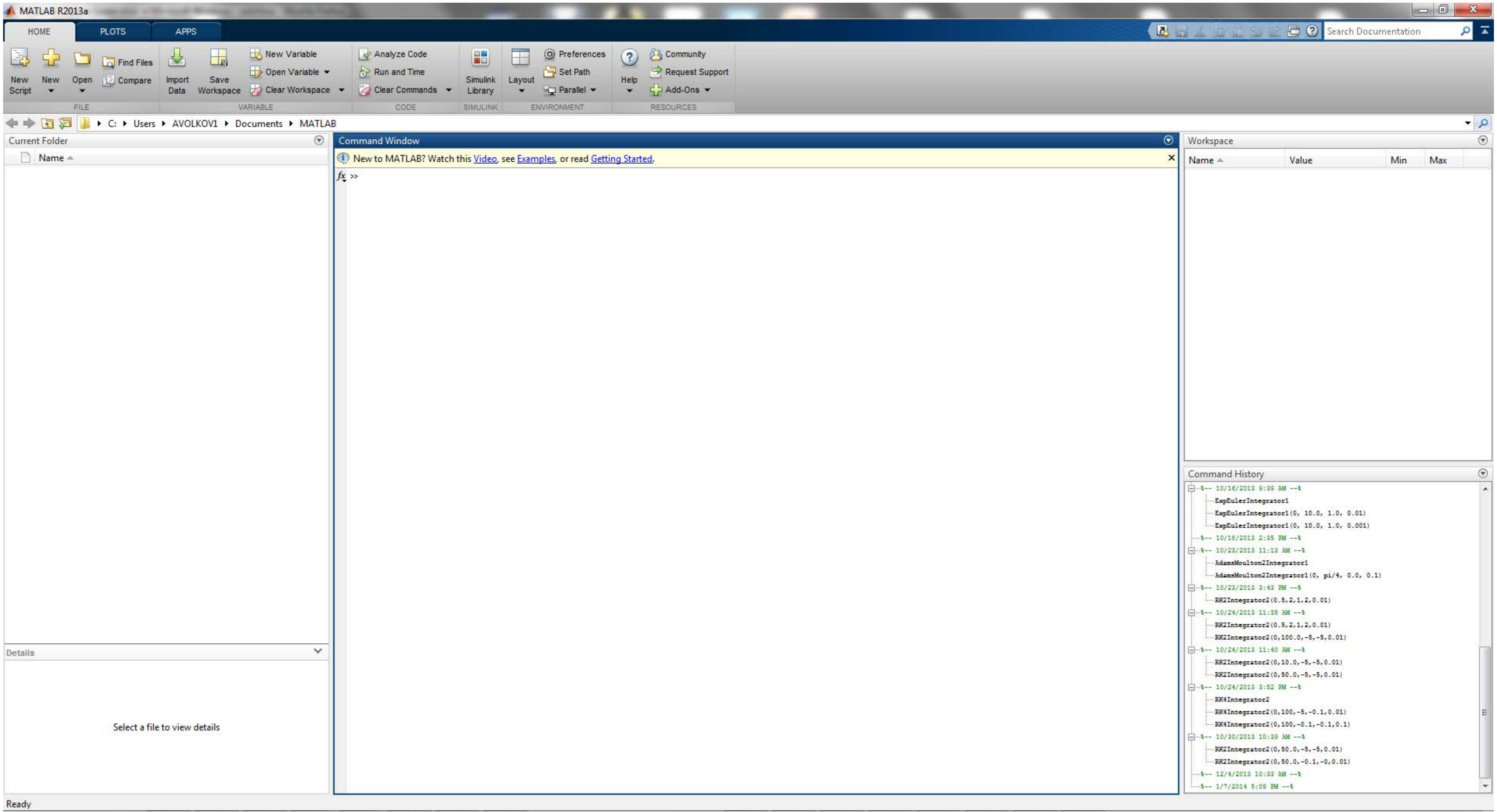
Arithmetic and symbolic calculations
- Arithmetic expressions
- Variables and arrays
- Build-in functions
- Scripts
- Data visualization

Use of Toolboxes
- Curve fitting toolbox

Programming (development of algorithms)
- Structuring algorithms : Used-defined functions
- Making decisions: Selection structures
- Repeating calculations: Loops

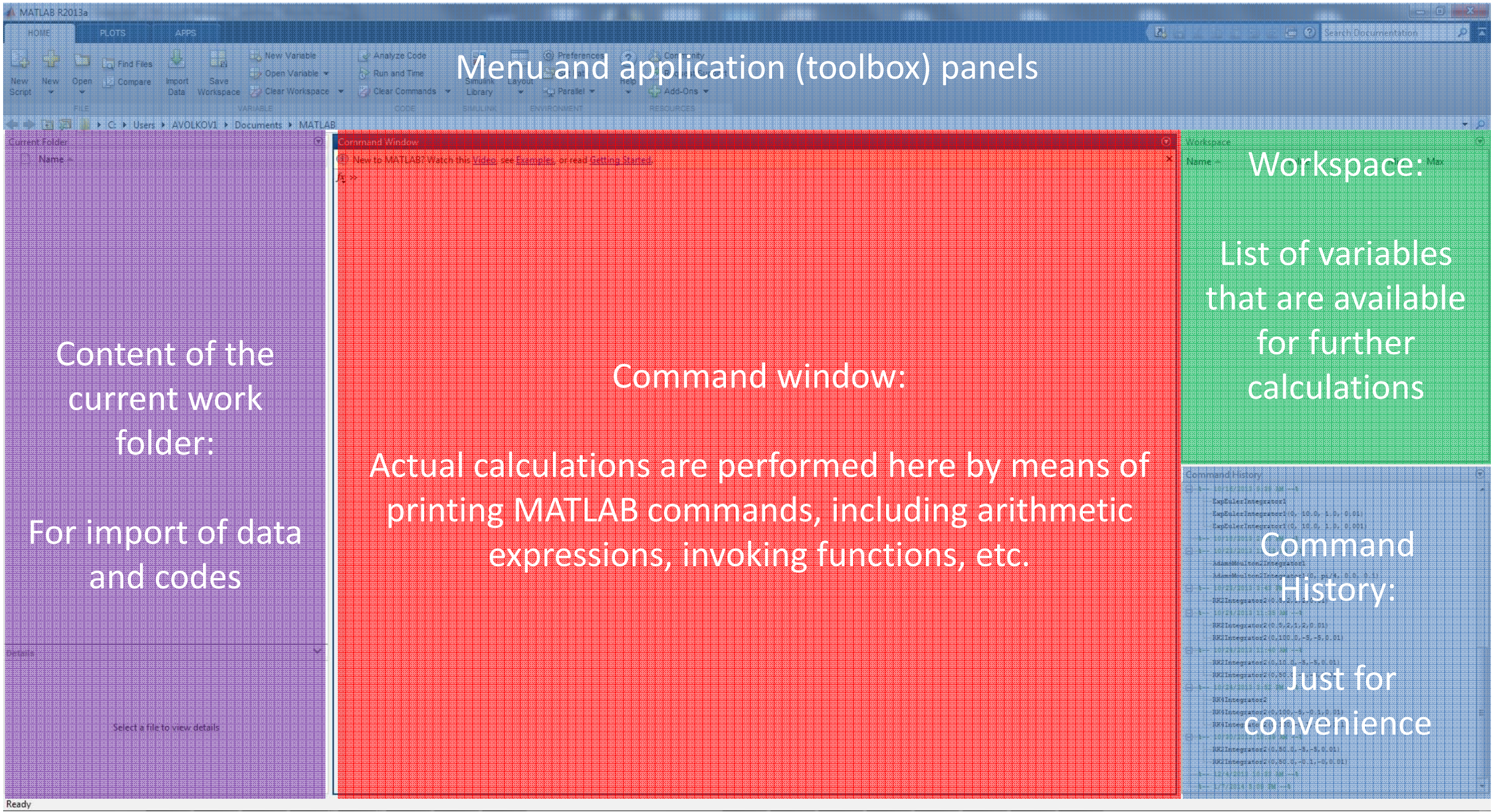
1.1. Basics of MATLAB

MATLAB program window



1.1. Basics of MATLAB

MATLAB program window



- ➢ When executing commands, MATLAB looks for the files only in the current work folder

1.1. Basics of MATLAB

MATLAB Command window is the primary place to perform calculations by defining variables, printing arithmetic expressions, and invoking commands and functions.

MATLAB Workspace is the list of **variables** we create and store in memory during a MATLAB session. We can

- Add variables to the workspace by invoking MATLAB instructions, using functions, and running MATLAB code.
- Save workspace to a disk file for use during the next MATLAB session.
- Load previously saved workspaces.

MATLAB Command history keeps the list of commands that was executed in the command window

When we execute any command in the command window

- It is saved in the Command history.
- If new variables are defined in the command, these variables are added to the Workspace.

Example: Commands executed in the command window

```
a = 5
b = 2
( a + b ) / 2.0
```

MATLAB Commands

- Any operation can be performed by executing a MATLAB **command** in the MATLAB command window
- MATLAB has two types of commands:
  - ✓ **Arithmetic expressions**, including definitions of new variables and invoking functions, which serve for real calculations or other purposes.
  - ✓ **Build-in (predefined) commands** that usually do not perform real calculations, but serve to change the default settings of the workspace and command window and perform other auxiliary operations.
- MATLAB build-in command can have a list of **arguments**: command [arg1] [arg2] …
- We can terminate execution of any command by typing "**Ctrl-C**" in Command window.

Example 1: Command **format** changes the default representation of real numbers in the command window

Example 2: Command **clc** clears the current contents of the command window

MATLAB Command	average_cost	Comments
format long	35.83333333333334	16 digits
format short e	3.5833e+01	5 digits plus exponent
format long e	3.583333333333334e+01	16 digits plus exponent
format hex	4041eaaaaaaaaaab	hexadecimal
format bank	35.83	2 decimal digits
format +	+	positive, negative, or zero
format rat	215/6	rational approximation
format short	35.8333	default display

MATLAB Arithmetic Expressions

- MATLAB arithmetic expressions can include
  - Numerical constants: 1, 2.3, -12.123e-4
  - Variable names: a, b, and, Results
  - Basic arithmetic operations: +, -, *, /, \, ^
  - Build-in and used-defined functions.

- To evaluate an expression, print it as a command in the command window

  Example: 1.0+sqrt(pi)/2.0.

- Basic arithmetic operations include:

Operation	Symbol	Example
addition, $a + b$	+	5+3
subtraction, $a - b$	–	23–12
multiplication, $a \times b$	*	3.14*0.85
division, $a \div b$	/ or \	56/8 = 8\56
power, a^b	^	5^2

1.1. Basics of MATLAB

Problem 1.1.1: Convert temperature 1000°F from Fahrenheit (°F) to Celsius (°C)

°C = (°F − 32.0) ÷ 1.8

Solution:

TF = 1000.0;

(TF − 32) /1.8

Problem 1.1.2: Calculate distance between points with Cartesian coordinates (1,3,5) and (7,8,-1)

Solution:

sqrt ((1 - 7)^2 + (3 - 8)^2 + (5 -(-1))^2)

Problem 1.1.3: Calculate $f = x^5$ at $x = -3$ using only multiplication

Solution:

F = 1;

X = -3;

F = F * X % Repeat this command 5 times

1.1. Basics of MATLAB

- We can use "↑" and "↓" keys in order to edit and repeat previous commands.
- MATLAB stores the result of the evaluation of an arithmetic expression in the pre-defined variable **ans**.
- MATLAB evaluates expressions from left to right with the following priority:
  - Function calls.
  - Powers.
  - Multiplication and division.
  - Addition and subtraction.

- **Brackets ()** can be used in order to change the order of evaluation.

  Example: (1.0 + sqrt (pi)) / 2.0 ≠ 1.0 + sqrt (pi) / 2.0

- If an expression is too long, print **three periods ...** + Enter to continue the expression on the next line.

  Example:

  (1.0 + sqrt (pi)) ...

  / 2.0

- Use **semicolon ;** in the end of expression in order to suppress printing the result.

1.2. MATLAB variables and build-in functions

- MATLAB scalar variables
- MATLAB build-in functions
- MATLAB help

Reading assignment

Gilat, 1.5 – 1.7

1.2. MATLAB variables and build-in functions

MATLAB variables

MATLAB **variable** stores a (numerical) value in the computer memory, which can be used for further calculations.

Scalar variable = individual numerical (or text) value.

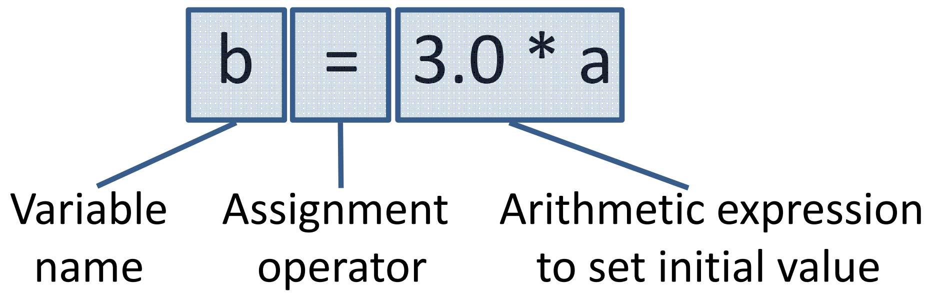
Array = set of indexed scalar variables.

Scalar variables

- **In order to define a new variable it is sufficient to assign to it some value.**

Example:

```
a = 2;
b = 3.0 * a
c = 1.0 + a^b
```



- Variable **name** is a sequence of characters, digits, and "_" starting with a character.
- **Assignment operator =** serves to change values of variables: a = 22.733, c='Jan. 10, 2014'
- A variable has "general type," but can store *integer* (0, $\pm$1, $\pm$2, ...), *real* (1.234), and *complex* (1.7+2.28i) values and *strings* ('This is a string')
- Once initialized, a variable can be used in arithmetic expressions in places of any constant.

1.2. MATLAB variables and build-in functions

Basic rules about variable names

We should not use variables which names which coincide with names of MATLAB build-in commands, variables or functions, and keywords

Rule	Comments
Variables are case sensitive.	`fruit`, `Fruit`, `FrUiT`, and `FRUIT` are all different MATLAB variables.
Variables can contain up to 63 characters.	
Variables must start with a letter, followed by any number of letters, digits, or underscores.	Punctuation characters are not allowed since many have special meaning to MATLAB.

Pre-defined variables (they always exist in the workspace)

Variable	Value
`ans`	Default variable name used for results
`pi`	Ratio of the circumference of a circle to its diameter
`eps`	Smallest number such that when added to 1 creates a floating-point number greater than 1 on the computer
`inf`	Infinity, e.g., 1/0
`NaN`	Not-a-Number, e.g., 0/0
`i` and `j`	`i = j =` $\sqrt{-1}$
`realmin`	The smallest usable positive real number
`realmax`	The largest usable positive real number

Command clear

- Command **clear** **[name]** deletes variable **[name]** from the workspace.
- **clear** deletes all variables from the work space.

MATLAB build-in functions

MATLAB **function** is a stand-alone part of the code, which performs some specific operation, e.g. calculation of an elementary mathematical function like calculation of $\sqrt{x}$, $\sin x$, etc.

Two types of functions:

- **Build-in functions** are part of MATLAB and can be used at any time.
- **User-defined functions** are written (coded) by user in the form of stand-alone files can be repeatedly executed (will be considered later).

Simple syntax of the function call:

Result = FunctionName (arg1, arg2, arg3)

Function = function name.

Result = variable which will contain the value calculated by the function (**output argument**).

arg1, **arg2**, **arg3** = list of the function parameters (**input arguments)**.

Typical purpose of the function:

To perform some calculations using arguments (arg1, arg2, arg3) as input parameters and assign the result of calculations to the return variable (Result).

- A function can have arbitrary number of arguments and returned values.
- Arithmetic expression can be used in the place of an individual function argument.

Elementary math build-in functions

`abs(x)`	Absolute value
`acos(x)`	Inverse cosine
`acosh(x)`	Inverse hyperbolic cosine
`angle(x)`	Angle of complex
`asin(x)`	Inverse sine
`asinh(x)`	Inverse hyperbolic sine
`atan(x)`	Inverse tangent
`atan2(x,y)`	Four quadrant inverse tangent
`atanh(x)`	Inverse hyperbolic tangent
`ceil(x)`	Round towards plus infinity
`conj(x)`	Complex conjugate
`cos(x)`	Cosine

`cosh(x)`	Hyperbolic cosine
`exp(x)`	Exponential: e^x
`fix(x)`	Round towards zero
`floor(x)`	Round towards minus infinity
`imag(x)`	Complex imaginary part
`log(x)`	Natural logarithm
`log10(x)`	Common logarithm
`real(x)`	Complex real part
`rem(x,y)`	Remainder after division: `rem(x,y)` gives the remainder of `x/y`
`round(x)`	Round toward nearest integer
`sign(x)`	Signum function: return sign of argument, e.g., `sign(1.2)=1`, `sign(-23.4)=-1`, `sign(0)=0`
`sin(x)`	Sine
`sinh(x)`	Hyperbolic sine
`sqrt(x)`	Square root
`tan(x)`	Tangent

Example: Calculation of the square root $x = \sqrt{238/\pi}$: x = sqrt (238.0 / pi).

1.2. MATLAB variables and build-in functions

Problem 1.2.1: Calculate

$$F = \log\left|\frac{e^{cx} - 1}{\sin(ax)}\right|$$

at $a = -2, c = -\pi/2, x = 1/3$

Solution:

```
a = - 2.0 ; c = - pi / 2.0 ; x = 1.0 / 3.0 ;
F = log (  abs ( ( exp ( c * x ) – 1.0 ) / sin ( a * x ) ) )
```

Problem 1.2.2: Roots of a quadratic equation

$$ax^2 + bx + c = 0$$

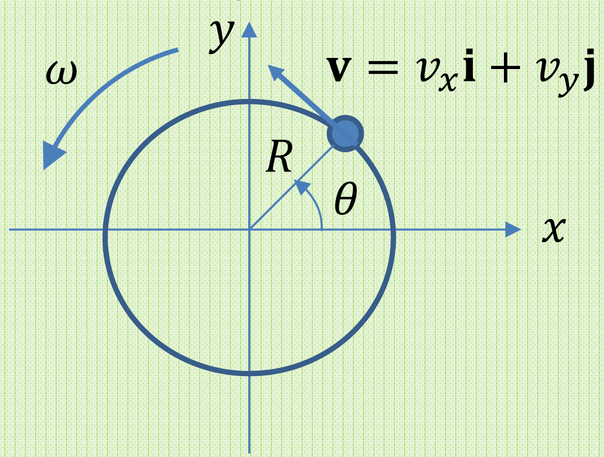
$$x_1 = \frac{-b + \sqrt{D}}{2a}, \qquad x_2 = \frac{-b - \sqrt{D}}{2a}, \qquad D = b^2 - 4ac$$

Solution:

1. Introduce variables for coefficients, e.g., a = 2, b = 2, c = -4.
2. Introduce variable $D = b^2 - 4ac$.
3. Introduce variables x1 and x2 for roots. Answer: x1 = 1, x2 = -2.
4. Repeat calculations for a = 1, b = 0, c = 4. Answer: x1 = 2i, x2 = -2i.

1.2. MATLAB variables and build-in functions

Problem 1.2.3: Calculate components of the velocity vector of a point rotating around axis Oz with frequency $f = 5$ Hz and at distance $R = 1$ cm at time t = 10 s



Solution:

$\omega = 2\pi f$

$\theta = \omega t$

$v = R\omega$

$v_x = -v \sin\theta$

$v_y = v \cos\theta$

```
f = 5.0;
R = 0.01;
t = 10.0;
Omega = 2.0 * pi * f;
Theta = Omega * t;
V = R * Omega;
Vx = - V * sin ( Theta )
Vy = V * cos ( Theta )
```

MATLAB Help

- Great source of help is the online MATLAB manual available at http://www.mathworks.com/help/matlab/
- Help is available through the MATLAB panel of instruments/menu or by pressing F1 key
- Information about specific MATLAB command/function is available in the command window by typing commands **help** and **lookfor**:
  - ✓ **help sqrt** retrieves information about topic "sqrt"
  - ✓ **help** shows all topics available
  - ✓ **lookfor sqrt** shows all topics related to word "sqrt"

1.3. MATLAB script files

- MATLAB script files
- Use of the MATLAB editor to create scripts
- Comments in script files

Reading assignment

Gilat, 1.8

MATLAB script files

MATLAB **script file** is a regular text file that contains a sequence of MATLAB commands. Default extension for the script files is "m", e.g. script.m. We can

- Create/edit a script file in the **MATLAB editor** or any external text editor.
- Run the script typing its name (script) in the command window. The name of the script should not be the current variable/build-in command, otherwise the current variable/build-in command will be executed instead of the script.

The results of the script execution is equivalent to typing all commands from the script in the command window. All variables defined in the script will be added to the workspace.

Four reasons to use scripts:

- To perform calculations repeatedly with different sets of data.
- To save our work for future MATLAB sessions (saving workspace, we save only variables).
- To debug/look for errors that usually requires multiple running of the same code.
- Script can contain a definition (initial values) of large arrays generated by stand-alone software or from laboratory measurements.

MATLAB editor has a lot a features that help to write script files.

1.3. MATLAB script files

Problem 1.3.1: Create, save to the disk, and run the script QuadEq.m for finding roots of the quadratic equation with arbitrary coefficients a, b, and c.

$$ax^2 + bx + c = 0$$

$$x_1 = \frac{-b + \sqrt{D}}{2a}, \qquad x_2 = \frac{-b - \sqrt{D}}{2a}, \qquad D = b^2 - 4ac$$

Solution:

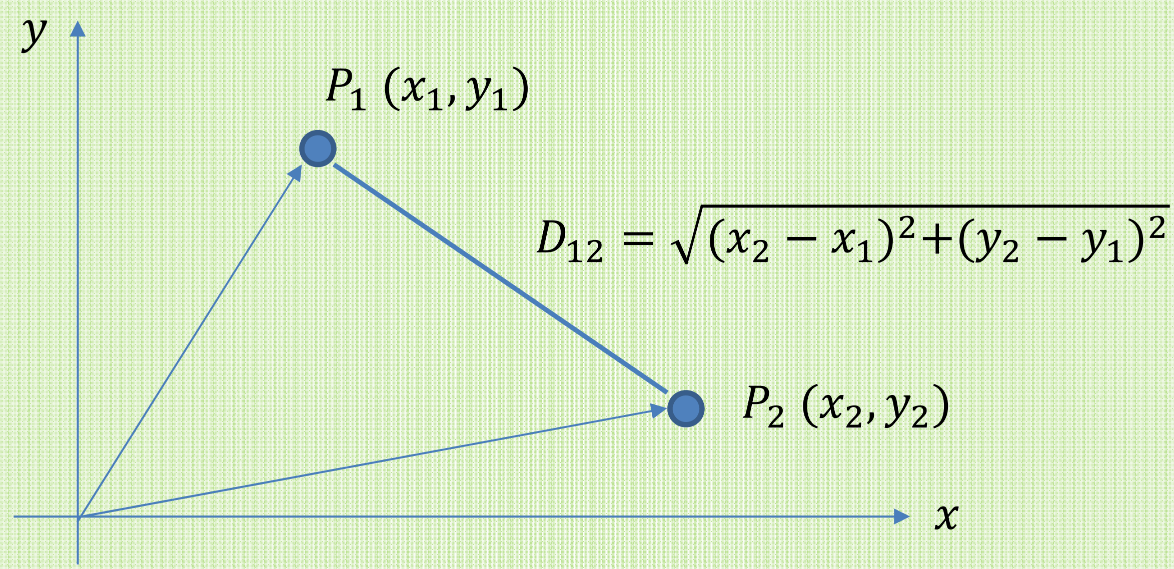
Script QuadEq.m:

```
D = b * b - 4.0 * a * c ;
x1 = ( - b + sqrt ( D ) ) / ( 2.0 * a )
x2 = ( - b - sqrt ( D ) ) / ( 2.0 * a )
```

1. a = 2, b = 2, c = -4: Roots are x1 = 1, x2 = -2.
2. a = 1, b = 0, c = 4: Roots are x1 = 2i, x2 = -2i.

1.3. MATLAB script files

Problem 1.3.2: Create, save to the disk, and run the script Distance2D.m for calculation of a distance between two arbitrary points on the plane Oxy.



Solution:

Script Distance2D.m:

```
DX = X2 – X1;
DY = Y2 – Y1;
D12 = sqrt ( DX^2 + DY^2 )
```

X1 = 2.0, Y1 = -4, X2 = 3, Y2 = 5: D12 = 9.0554.

Comments in MATLAB script files

MATLAB **comment** = any text starting from **%** until the end of the line. Comments are ignored during the execution of script/function files.

Example : y = (exp (x) + exp (- x)) / 2 % Variable y is equal to the hyperbolic cosine of x

The good programming practice is to add comments to the code in order to

- Explain the conditions/restrictions applied to the code.
- Explain non-obvious logics/order of calculations.
- Provide references to literature/other sources, containing coherent description of algorithms or warranting the choice of the simulations parameters.

Problem 1.3.3: Add comments to **script QuadEq.m**:

```
% This script file solves problem 1.3.2 from the lecture notes
% This script finds roots x1 and x2 of the quadratic equation
% a * x^2 + b * x + c = 0.
% Coefficients a, b, and c are defined in the script.
echo on % Here we switch on printing the commands
a = 2.0
b = 2.0
c = -4.0
D = b * b - 4.0 * a * c ; % Semicolon suppresses printing the result for D
echo off % Here we switch off printing the commands
x1 = ( - b + sqrt ( D ) ) / ( 2.0 * a )
x2 = ( - b - sqrt ( D ) ) / ( 2.0 * a )
```

1.4. MATLAB arrays

- One-dimensional arrays
- Creation of one-dimensional arrays
- Vectorized mathematics for arrays
- Use of arrays to manipulate the physical vectors

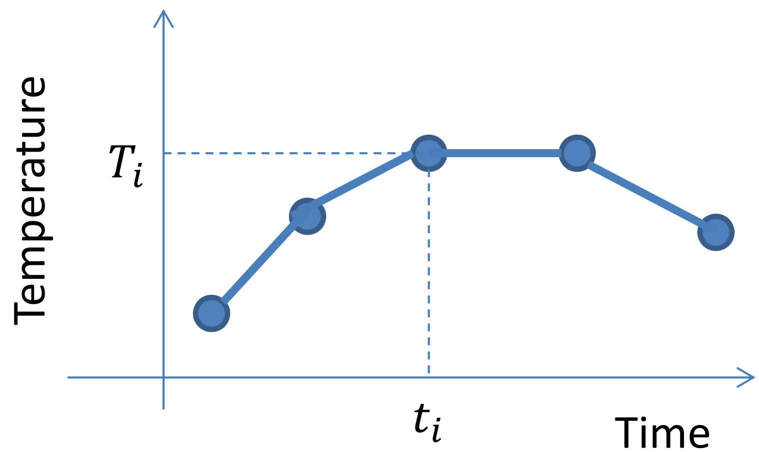
Reading assignment

Gilat, 2.1-2.6, 3.1, 3.4, 3.5, 3.6

Why do we need arrays?

Example: Assume that using a thermocouple we register the body temperature every second during 100 seconds. Then we will have 100 values of measured temperature. How can we keep in the computer memory all these values and plot temperature versus time?

- **In mathematics**, we usually use **indexed variables**:
  In order to distinguish these values we can introduce index i and assume that T_i is the temperature measured at time t_i .



- **In programming**, we use **arrays** in order to keep in the computer memory all values of indexed variables

  Array for time **t = [t(1) t(2) ... t(i-1) t(i) t(i+1) ... t(99) t(100)]**

  Array for temperature **T = [T(1) T(2) ... T(i-1) T(i) T(i+1) ... T(99) T(100)]**

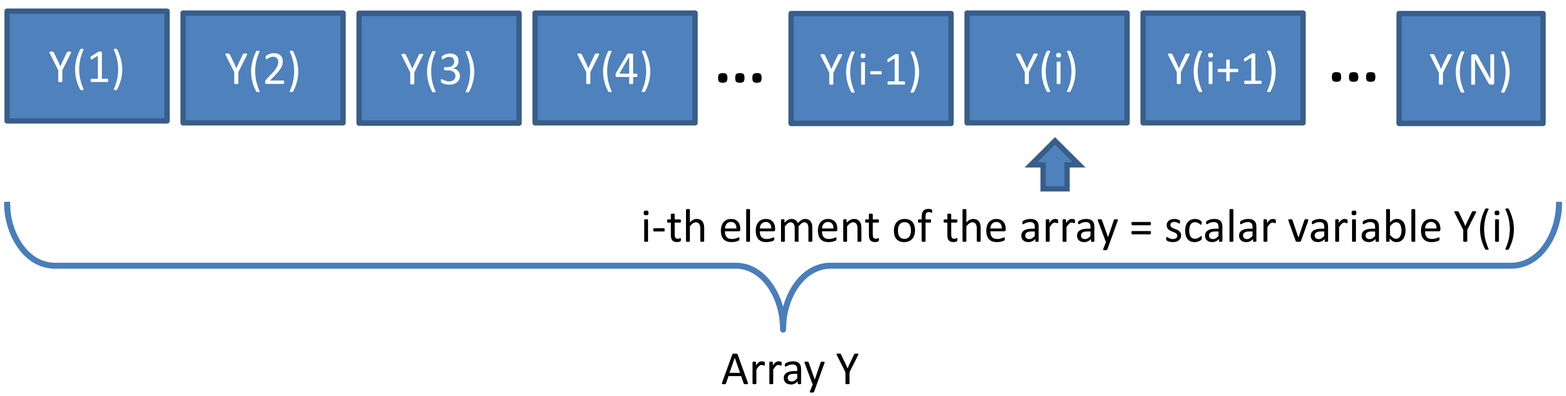
  Then to plot temperature versus time we can say: **plot (t, T)**

- Arrays are useful when
  - We need to analyze a large set of uniform/similar data, e.g., **tabulated data**.
  - We perform similar operations on every individual variable/value from this set.

MATLAB one-dimensional arrays

- MATLAB **array** is the list (ordered set, collection) of scalar variables of the same type
- Array serves to keep tabulated data in computer memory
- **Scalar variable** X = individual value

One-dimensional array Y = an ordered set of N scalar variables Y(i) of the same type, where every element has **one** index



Y = **Name** of the array

Y(i) = i-th **element** of the array Y

i = Integer **index** of elements of array Y varying from 1 to N

N = **Size** of array Y (the number of its elements)

1.4. MATLAB arrays

Two basic operations with arrays:

- **Creation**:
  - To create an array we must specify its name, size, and individual value of every element.

    ArrayName = [Value1 Value2 … ValueN]

  - **Brackets []** can be used in order to specify initial values of the array elements.
    Example: Array of three elements, Y = [1 -1 (cos (pi / 4))].

- **Accessing**, i.e. getting an element or group of elements of an array
  - **Parenthesis ()** can be used to access individual element of an array.

    Example: Y(2), individual scalar variable = element of array Y with index 2.

  - An individual element of an array can replace a scalar variable in any arithmetic expression.

Problem 1.4.1: Calculate distance between points with Cartesian coordinates (1,3,5) and (7,8,-1)
Script Distance3DVec.m

```
X = [ 1 3 5 ];
Y = [ 7 8 -1 ];
L = sqrt ( ( X(1) - Y(1) )^2 + ( X(2) - Y(2) )^2 + ( X(3) - Y(3) )^2 )
```

Creation of one-dimensional arrays

Four ways to create an array in MATLAB:

- **Explicit definition of every element of the array with square brackets []:**

  Example: Array x with three elements of given values x(1)=0.1, x(2)=2 * pi, x(3)=8.

  x = [0.1 2 * pi 2^3] or **x = [0.1, 2 * pi, 2^3]**

- **Create an array with equal spacing between neighbor points using square brackets []**

  Array x with first element x(1)=m, last element n, and spacing q. The number of elements is equal to (n - m) / q + 1.

  x = [m:q:n] or **x = m:q:n** (x = [m:n] means q = 1)

- **Create an array with equal spacing between neighbor points using function linspace**

  Array of n elements, where the first element is equal to x0, last element is equal to x1, and spacing q = (x1 - x0) / (n - 1).

  x = linspace (x0, x1, n) :

- **Create an array based on another array: Number of elements and their values will be inherited from the source array**

  Example: Array y such that y(i) = sin (x(i)).

  y = sin (x)

1.4. MATLAB arrays

Problem 1.4.2: Create an array x containing 11 numbers from 0 to 1 with equal spacing and calculate array y such that $y_i = \exp(x_i)$

Script Create1DArrays.m

- Explicit definition of every element of the array with square brackets **[]**:

  x = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0]

- Create an array with equal spacing between neighbor points using square brackets **[]**

  x = [0 : 0.1 : 1]

- Create an array with equal spacing between neighbor points using function **linspace**

  x = linspace (0.0, 1.0, 11)

- Create an array based on another array

  y = exp (x)

Useful functions for arrays

- **sum** (x) calculates the sum of elements of array x
- **max** (x) returns the maximum value in array x
- **min** (x) returns the minimum value in array x

$$\mathrm{sum}(\mathbf{x}) = x_1 + x_2 + \cdots + x_N = \sum_{i=1}^{N} x_i$$

1.4. MATLAB arrays

- **Vectorised mathematics** allows us to perform arithmetic operations on every element of an array with a single instruction.

Use of arrays with elementary build-in math functions

- Majority of build-in elementary functions can be applied to whole arrays

  Example: x = [0 pi/2 pi] ; y = cos (x) ;

Scalar-array mathematics

- Arithmetic operations a + b, a - b, a * b, and a / b can be used if a is an **array** and b is a **scalar variable**.
  - ✓ c = a + b means c(i) = a(i) + b for all i
  - ✓ c = a - b means c(i) = a(i) - b for all i
  - ✓ c = a * b means c(i) = a(i) * b for all i
  - ✓ c = a / b means c(i) = a(i) + b for all i

  b + a, b - a, b * a are calculated similarly

Array-array mathematics

- Arithmetic operations a + b and a - b can be used if a and b are **arrays of the same size**
  - ✓ c = a + b means c(i) = a(i) + b(i) for all i
  - ✓ c = a - b means c(i) = a(i) - b(i) for all i

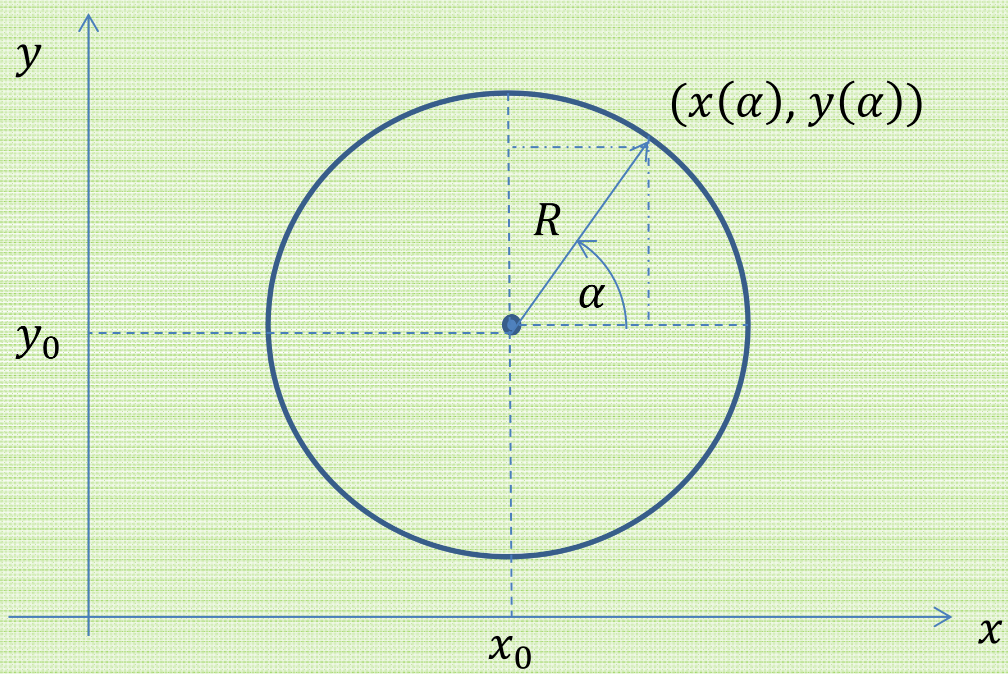
Element-by-element operations

- For two arrays of the same structure, x and y, one can use element by element operations .*, ./, .\, .^
  - ✓ c = a .* b means c(i) = a(i) * b(i) for all i
  - ✓ c = a ./ b means c(i) = a(i) / b(i) for all i
  - ✓ c = a .^ b means c(i) = a(i)^b(i) for all i

Problem 1.4.3: Calculate coordinates of points on circle of radius R=2 with center in point (1, 2)

Parametric representation of a circle:

$$\mathbf{r}(t) = x(\alpha)\mathbf{i} + y(\alpha)\mathbf{j}$$
$$x(\alpha) = x_0 + R\cos\alpha$$
$$y(\alpha) = y_0 + R\sin\alpha$$



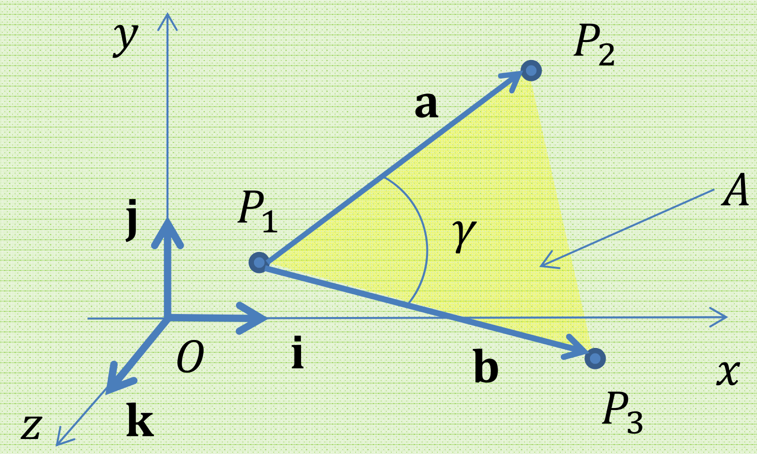
Script Circle.m:

```
R = 2.0;
X = [ 1 2 ];
angle = [0:0.1:1] ;
angle = 2.0 * pi * angle;
x = X(1) + R * cos ( angle )
y = X(2) + R * sin ( angle )
```

Use of arrays in order to manipulate physical vectors

Problem 1.4.4: Assume we introduce Cartesian coordinates and fix three points

$P_1 = (x_1, y_1, z_1),\ P_2 = (x_2, y_2, z_2),\ P_3 = (x_3, y_3, z_3).$



Let's introduce vectors

$\mathbf{a} = (x_2 - x_1, y_2 - y_1, z_2 - z_1) = (a_x, a_y, a_z)$

$\mathbf{b} = (x_3 - x_1, y_3 - y_1, z_3 - z_1) = (b_x, b_y, b_z).$

We need to calculate:

Dot product $\mathbf{a} \cdot \mathbf{b}$

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

Vector product $\mathbf{a} \times \mathbf{b}$

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = (a_y b_z - a_z b_y)\mathbf{i} - (a_x b_z - a_z b_x)\mathbf{j} + (a_x b_y - a_y b_x)\mathbf{k}$$

Area A of triangle with vertexes $P_1, P_2,\ P_3$

$$A = \frac{1}{2} |\ \mathbf{a} \times \mathbf{b}\ |$$

The angle γ between vectors **a** and **b**:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\gamma, \qquad |\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2} = \sqrt{\mathbf{a} \cdot \mathbf{a}}$$

1.4. MATLAB arrays

Script Vectors.m:

```
P1 = [ -1 2 15 ];
P2 = [ 0.75 -32.0 1.5e+1 ];
P3 = [ 0 3 -1 ];
a = P2 - P1;
b = P3 - P1;
aabs = sqrt ( sum ( a.^2 ) );
babs = sqrt ( sum ( b.^2 ) );
ab = dot ( a, b ); % = sum ( a .* b );
axb = cross ( a, b ); % =  [ a(2) * b(3) - a(3) * b(2), a(3) * b(1) - a(1) * b(3), a(1) * b(2) - a(2) * b(1) ]
gamma = acosd ( ab / ( aabs * babs ) )
A = 0.5 * sqrt ( dot ( axb, axb ) ) % = 0.5 * sqrt ( sum ( axb.^2 ) )
```

$$\gamma = \text{acos}\,\frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|} = 93.381^{\circ}, \qquad A = 272.946$$

- Function **dot** (a, b) calculates the dot product of vectors a and b
- Function **cross** (a, b) calculates the cross product of vectors a and b

1.5. MATLAB two-dimensional and three-dimensional plots

- Two-dimensional line plots
- Plotting multiple graphs in the same plot
- Formatting the plot
- Export of plots to graphic (image) files

Reading assignment

Gilat, 5.1, 5.3– 5.5, 5.10, and 511

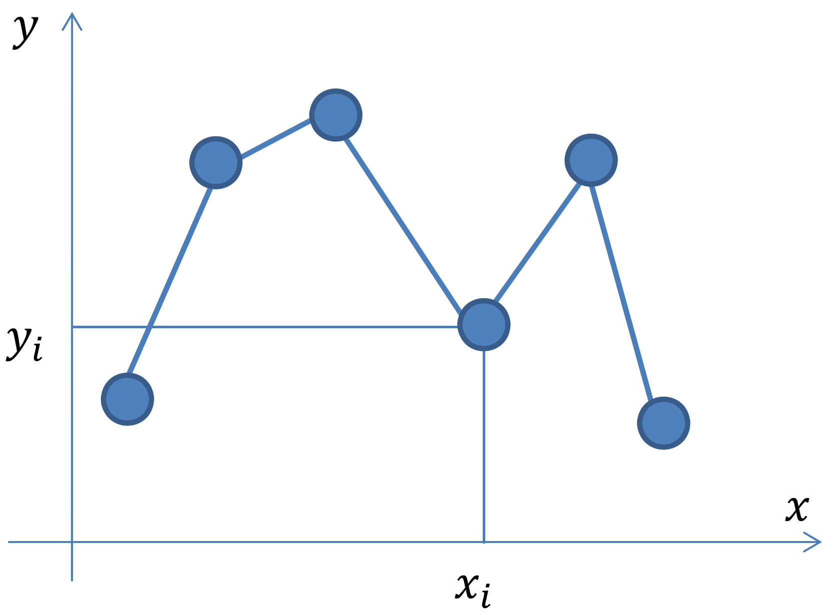
Export of plots to graphic (image) files:

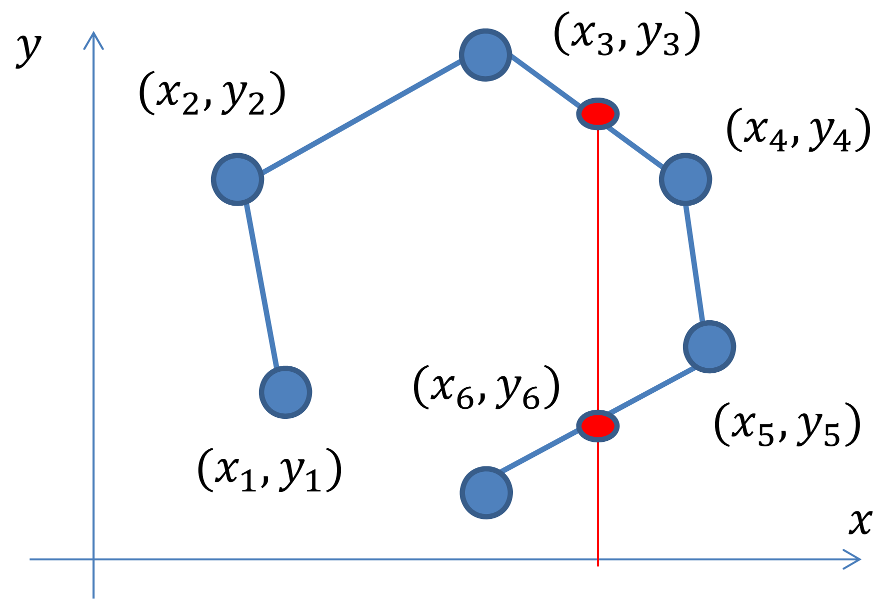
http://www.mathworks.com/help/matlab/printing-and-exporting.html

Plotting two-dimensional (2D) data with the plot function

Function $y = y(x)$ (unique y for every x)

2D Curve (non-unique y for every x)





- The **plot** function plots 2D data (function and curves) in a special **figure window**.
- The **plot** function plots values of one 1D array x versus values of another 1D array y.

x = [x1 x2 x3 ... xN]
y = [y1 y2 y3 ... yN]

Coordinates of point 3 (x_3, y_3)

plot (x, y [, optional parameters])

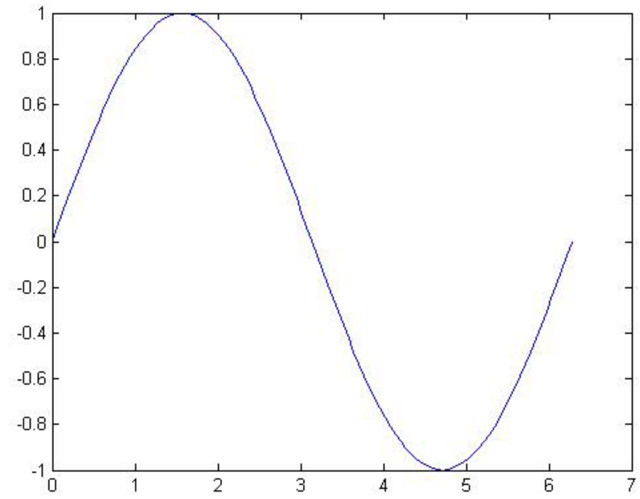
- Plot is composed of a polyline connecting points (x1,y1), (x2,y2), etc.

1.5. MATLAB two-dimensional and three-dimensional plots

Problem 1.5.1 : Plot of one period of $\sin x$.

Script SinPlot.m

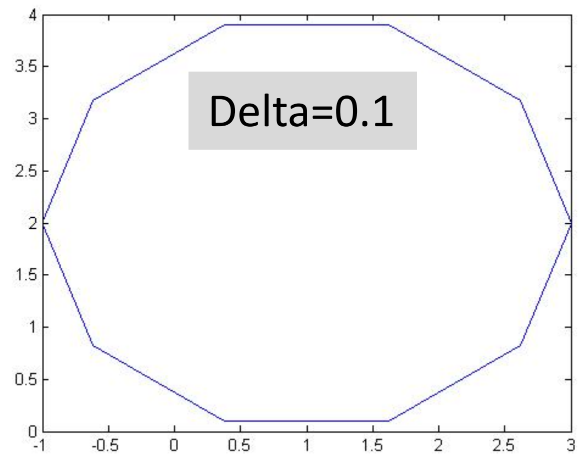
```
x = [0:0.01:1] ;
x = x * 2 * pi ;
y = sin ( x ) ;
plot ( x, y )
```

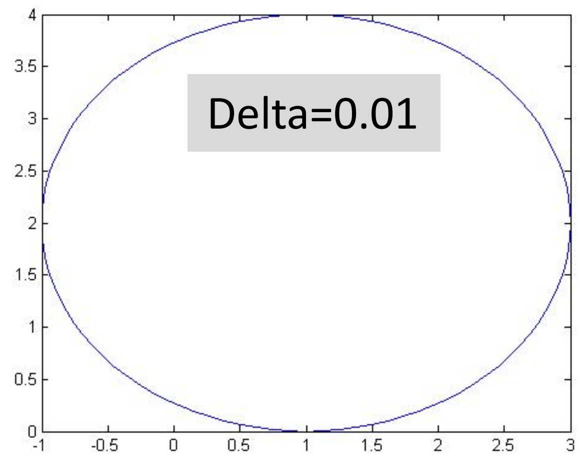


Problem 1.5.2: Print a circle of radius R=2 with the center in the point $\mathbf{X} = (\,1, 1\,)$

Script CirclePlot.m

```
R = 2.0;
X = [ 1 1 ];
Delta=0.1;
angle = [ 0 : Delta : 1 ] ;
angle = 2.0 * pi * angle;
x = X(1) + R * cos ( angle );
y = X(2) + R * sin ( angle );
plot ( x, y )
```





- By default, plot function updates the current figure window. How can we plot multiple graphs in the same figure window?

Plotting multiple graphs in the same plot field

There are three ways to plot a few graphs in the same plot (in the same figure window):

- The **plot** function can plot curves for many pairs of vectors.
  Example: **plot** (x1,y1,x2,y2) plots y1 vs. x1 and y2 vs. x2 in the same plot.

- The **line** function can add an additional curve (graph) to the plot that already exists.
  Example: **plot** (x1, y1) ; **line** (x2, y2)

- The **hold on** and **hold off** commands can be used in order to keep the figure window open for adding additional curves with successive plot functions.
  Example : **plot** (x1, y1) ; **hold on** ; **plot** (x2, y2) ; **hold off**

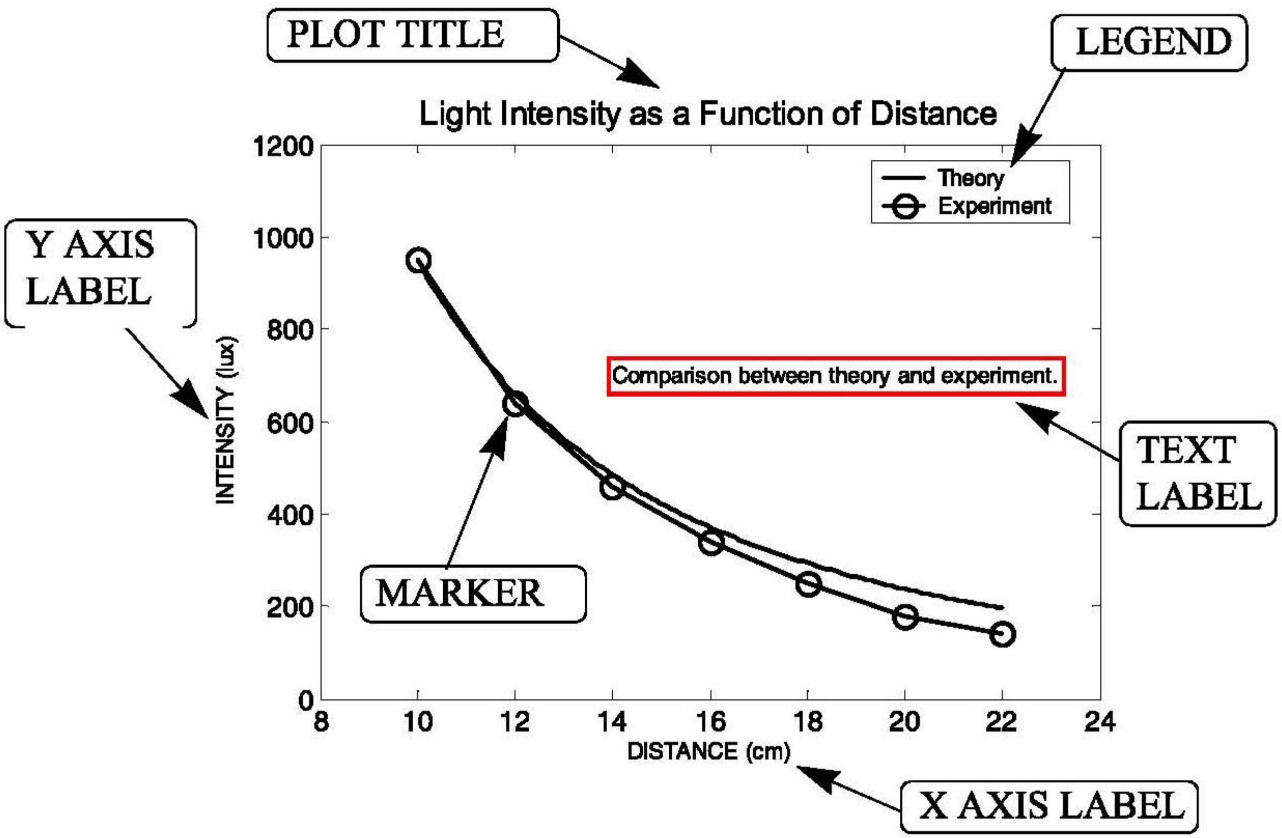
Problem 1.5.3: Plot sine and cosine in the same figure window

Script SinCosPlot.m

```
angle = 2.0 * pi * [ 0: 0.01 : 1 ] ;
y = sin ( angle ) ;
plot ( angle, y, 'r' ) ;
y = cos ( angle ) ;
line ( angle, y, ‘Color‘, ‘Green’ ) ;
```

Formatting plots

- Basic components of the two-dimensional line plot are show in the figure.
- We can change visual appearance of all these components either by specifying addition arguments to the **plot/line** functions or by using additional functions/commands after invoking the **plot** function.



- **Additional arguments** of the **plot** and **line** functions changes visual appearance of an **individual curve** (pattern, thickness, and color of the line segments and markers).
- **Additional functions** change visual appearance of the **common plot elements** (title, axis labels, legend, etc.).

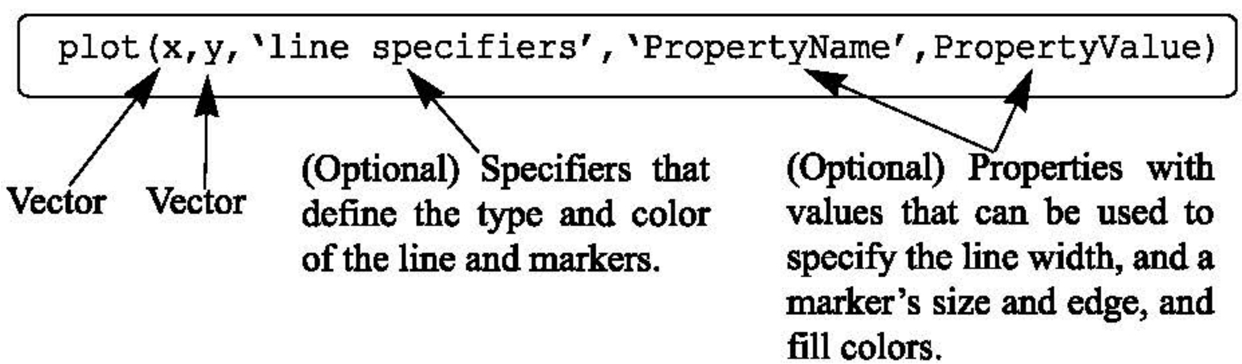
Formatting individual curves in the plot

- In the plot function, every pair of arrays (x and y) can be followed by a series of expressions of two types:
  - ✓ Line specifiers
  - ✓ Line properties

  that allows us to change visual appearance of the corresponding curve.

- The **line specifier** is a string that symbolically defines the line color, pattern, and maker type.
  Example: **plot** (x, y, 'r--x')

- The **line property** is pair of a string, containing the property name, and a value of this property
  Example: **plot** (x, y, 'LineWidth', 3)



- Only line properties can be used in the **line** function.
- See details on line specifiers and properties in Gilat, pages 135-137 (Required for the exam!).

Formatting common elements of a plot

Element	Function/command
Title	**title** ('Title text')
Axis labels	**xlabel** ('X label text') ; **ylabel** ('Y label text')
Text labels	**text** ('Text label')
Legend	**legend** ('String1', 'String2', …, pos) Optional integer pos argument (-1,..4) specifies position of the legend with respect to the plot edges .
Axis	**axis** ([xmin, xmax, ymin, ymax]) specifies limits for x- and y-axes **axis equal** : Sets the same scale for both axes (circle will be shown as a circle). **axis tight**: Sets the axis limits to the range of the data.
Grid	**grid on**: Adds grid lines to the plot. **grid off** : Removes grid lines from the plot.
Scale type	**semilogy** (x, y) : plots y in the **logarithmic scale** (use instead of **plot**) **semilogx** (x, y) : plots x in the logarithmic scale (use instead of **plot**) **loglog** (x, y) : plots both x and y in the logarithmic scale (use instead of **plot**)

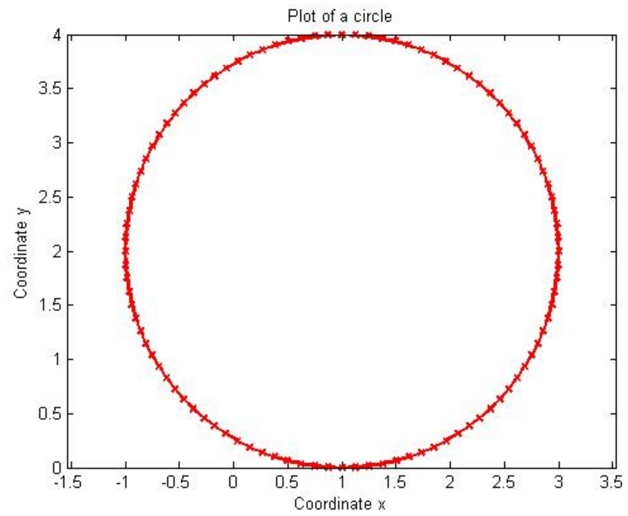
- Text strings allow for complex formatting (addition of Greek characters, etc.; Gilat, 146-147)
- Common plot elements can be changed through the menu of the figure window (Gilat, 5.4.2)

1.5. MATLAB two-dimensional and three-dimensional plots

Problem 1.5.4: Set equal scale for axes and add title and axis labels for the plot in problem 1.5.2

Script CirclePlotFormatted.m

```
R = 2.0; X = [ 1 2 ]; Delta = 0.01;
angle = [0:Delta:1]; angle = 2 * pi * angle;
x = X(1) + R * cos ( angle );
y = X(2) + R * sin ( angle );
plot ( x, y, 'r-x', 'LineWidth', 2 )
axis equal
title ( 'Plot of a circle' )
xlabel ( 'Coordinate x' )
ylabel ( 'Coordinate y' )
```



Exporting plots to graphic (image) files

- The **print** command sends the content of the current figure window to a printer or the graphic file of the specified format:

  print

  print argument1 argument2 ... argumentn

- In order to print to file, the filename and format should be specified:

  print –dformat 'Filename'

  format = bmp for the 24-bit bmp file.

  format = jpeg for 24-bit jpeg file, etc.

- See details on http://www.mathworks.com/help/matlab/printing-and-exporting.html.
- Graphic files can be printed with "File/Save As" command of the menu in the figure window.

1.6. MATLAB used-defined functions

- Why do we need functions?
- Major components of a user-defined function
- How to write the functions:

  Syntax of user-defined functions
- How to use functions:

  Calls of user-defined functions
- Workspace, local, and global variables in MATLAB
- Passing input and output parameters

Reading assignment

Gilat, 7.1, 7.2, 7.4-7.7

Why do we need functions ?

User-defined MATLAB **function** is the stand-alone MATLAB code (sequence of MATLAB commands) written in the MATLAB language, saved into a regular **text file with a special syntax**, and **used like build-in MATLAB functions**.

Example: Calculation of the angle between two vectors (see problem 1.4.4)

Solution without functions	Solution with a function
a = [1 3 5];	function uv = ab (u, v) % Dot product of v and u
b = [7 8 -1];	uv = u(1) * v(1) + u(2) * v(2) + u(3) * v(3);
aabs= sqrt (a(1) * a(1) + a(2) * a(2) + a(3) * a(3));	end
babs= sqrt (b(1) * b(1) + b(2) * b(2) + b(3) * b(3));	a = [1 3 5];
ab = a(1) * b(1) + a(2) * b(2) + a(3) * b(3)	b = [7 8 -1];
gamma = acos (ab / (aabs * babs))	gamma = acos (ab (a, b) / sqrt (ab (a, a) * ab (b, b)))

- **Functions is the major tool to logically divide a complex problem into simple sub-problems.**
- A function implements solution of a logically simple problem, which later can be used as a part of the solution of **multiple** more complicated problems: **We save time when we solve different problems**.
- A function allows one to reduce the size of the code if it implements an algorithm that is used a few times for different data sets: **We save time when we write the code.**
- A function can be written and debug independently of the rest of the code. We can easily isolate and correct errors in functions. **We save time when we debug the code**.

Major components required to define and use a function

Function **output arguments** (results) uv

Function **name** (ab)

Function **input arguments**(data) u and v

Function **body** (algorithm)

```
function uv = ab ( u, v )
      uv = u(1) * v(1) + u(2) * v(2) + u(3) * v(3);
end

a = [ 1 3 5 ];
b = [ 7 8 -1 ];
gamma = acos ( ab ( a, b ) / sqrt ( ab ( a, a ) * ab ( b, b ) ) )
```

Definition of the function

Use of the function in the external code

Function **calls**

Functions has input and output arguments: This is the major difference compared to scripts.

Two steps to use a function:

- **Create/define** function in the form of an individual text file with extension ".m" in the MATLAB editor or any external text editor.
- **Call (Run)** the function in the command window like any build-in function.

Syntax of MATLAB used-defined functions

- MATLAB functions are distinct from MATLAB scripts: Functions have a special syntax that defines the list of input arguments, output parameters, name, and body of the function.
- Syntax of the user-defined MATLAB function file:

```
function [ oarg1 oarg2 ... ] = UserFun ( iarg1, iarg2, ... )
%UserFun This is an example of the user-defiled function
% No real calculations are performed
.....
end
```

Function definition line (FDL)
H1 line (optional)
Help lines (optional)
Body (algorithm)
End line (optional)

- FDL includes keyword **function**, **name** of the function (UserFun) and lists of **output** [oarg1, oarg2, ...] and **input** (iarg1, iagr2, ...) **arguments**.
- Content of H1 is used in search of **lookfor** command.
- Content of Help lines is used in the **help** command.
- Body contains a list of commands that transform input arguments into output ones.
- End line consists of keyword **end**, but can be omitted.

Creation of a function

- Name of the function should coincide with the function file name.
- The MATLAB editor allows one not only to create/edit functions, but also to run functions separately from the command window, mostly for debug purposes.
- **In order for the function to work, the output arguments must be assigned values of the function body.**
- Simplified FDLs are available:
  - **function** oarg1 = UserFun (...) ≡ **function** [oarg1] = UserFun (...)
  - **function** = UserFun (...) ≡ **function** [] = UserFun (...)

Problem 1.6.1: Write a function solving the quadratic equation.

File QuadEqFun.m file:

```
function [ x1 x2 ] = QuadEqFun ( a, b, c )
%QuadEqFun Calculates roots of the quadratic equation
% This functions finds roots x1 and x2 of the quadratic equation a * x^2 + b * x + c = 0.
    D = b * b - 4.0 * a * c ;
    x1 = ( - b + sqrt ( D ) ) / ( 2.0 * a ) ;
    x2 = ( - b -  sqrt ( D ) ) / ( 2.0 * a ) ;
end
```

Call of a MATLAB user-defined function

The syntax of the function call is

[oarg1, oarg2, ...] = UserFun (iarg1, iarg2, ...)

When MATLAB calls a function, It:

I. **Evaluates every expression in the place of input parameter.**
II. **Executes the sequence of instruction in the function body .**
III. **Updates values of actual variables used in place of output arguments.**

- It is the responsibility of programmer to ensure that the type of actual input parameters corresponds to the type of input arguments assumed in the definition of the function.
- Output parameters can be the names of existing or no-existing workspace variables. If such variables do not exits, they will be created as a result of the function call.

Problem 1.6.2: Find the roots of the quadratic equation for a = 1 + $\sqrt{\pi}$, b = 2, c = 2.37 / 2.0

Solution: [root1 root2] = QuadEqFun (1.0 + **sqrt** (**pi**), 2.0, 2.37 / 2.0)

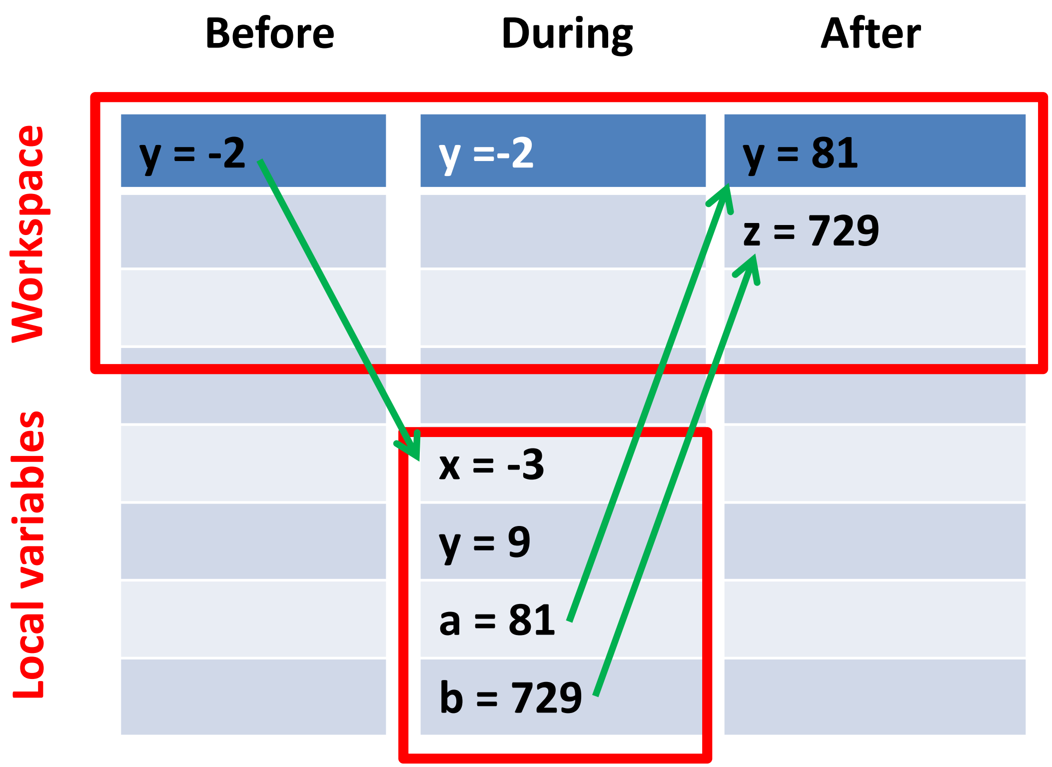
Workspace, local, and global variables in MATLAB

Three classes of MATLAB variables can be used inside functions:

- **Workspace variables**
  - Are defined in the workspace (command window) and exist until explicitly deleted by the **clear** command.
  - Can be passed to/received from a function through its arguments.
- **Local function variables**
  - Are function arguments and any variables defined in the body function.
  - Exist only during the execution of the function body and are not available for calling code.
  - Names of local variable can coincide with names of workspace variables, because these variables represent different cells of the computer memory.
- **Global variables**
  - Defined inside the function using keyword **global.**
  - Available inside any function where they are defined as global.
  - Can be also available in the workspace if defined as global in the workspace (type "**global** var" in order to define var as a global variable in the workspace).
  - We will not use global variables. See Gilat's book, 7.3 (p. 225).

Passing input and output parameters

Let's consider a **sketch of the computer memory** at three stages of execution of Powers: Before, during, and after calling Powers function in the script



MATLAB function file Powers.m:

```
function [ a b ] = Powers ( x )
    y = x * x;
    a = y * y; % = x^4
    b = a * y; % = x^6
end

y = - 2;
[ y z ] = Powers  ( y - 1 )
```

- During a function call, MATLAB creates additional **local variables.**
- **Local variable y and workspace variable y are different variables.** There is no any relationship between them.
- Local variables exist only during execution of the function body.

- Why do we need branching? What does 'to make a decision' mean?
- Logical variables
- Logical operators
- Relational operators

Reading assignment

Gilat, 6.1 and 6.2

Why do we need branching?

- Now we know how to perform **computations** with various data in the MATLAB.
- We also need to know how to **make decisions** when analyzing data.

  "To make a decision" means branching, i.e. "to select an alternative from a few options."

Simple examples:

- Find maximum c of two values a and b.
- Set value x to zero if it is negative.
- Calculate sign of value of x.

$$\max(a,b) = \begin{cases} a & a \geq b \\ b & b > a \end{cases}$$

$$\operatorname{sign}\ x = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

More complex examples involving selection:

- Finding of maximum element of an array.
- Sorting of elements of an array in the ascending order.

Making decisions implies that we

- Have two (a few in the general case) alternative **paths** of computations.
- Have a **condition**, which can be either valid (**true**) or invalid (**false**). This condition allows us to chose one of the alternative computation paths.

To make a decision = to check the condition and then, based on the result of this check, chose one of the alternative paths of computations.

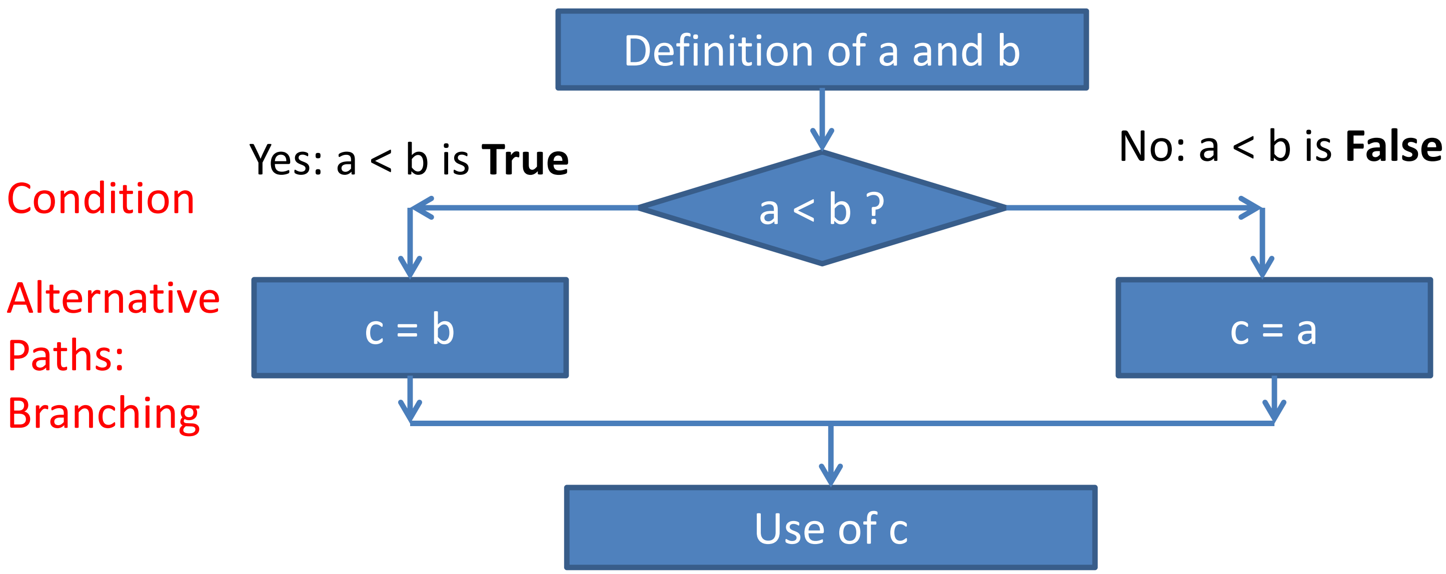
1.7. MATLAB relational operators, conditional statements, and selection structures I

To make a decision = to check the condition and then, based on the result of this check, chose one of the alternative paths of computations.

Example: Find maximum c of two variables a and b.

Rectangle = regular command
Rhombus = Binary branching of the algorithm based on the result of the condition check

Flowchart of the algorithm:



Code:

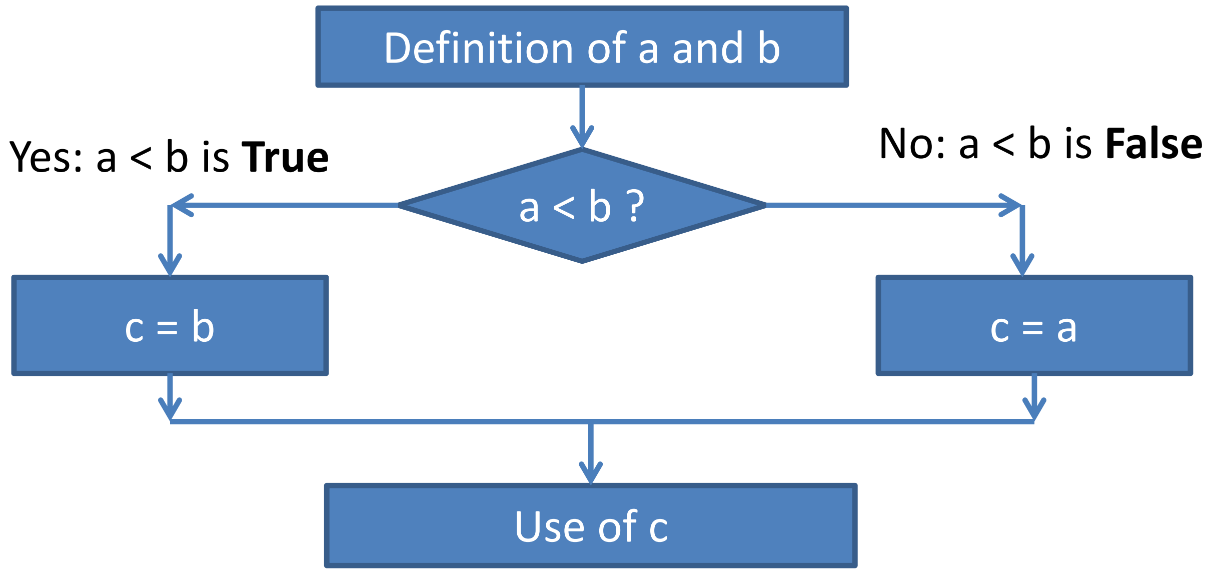
```
if a < b
  c = b
else
  c = a
end
```

Example: Find maximum c of two variables a and b.

Flowchart of the algorithm:

Condition

Alternative paths



Code:

```
if a < b
  c = b
else
  c = a
end
```

In order to make decisions/do branching in the code we need

- **Relational operators** (<, >) that allow us to formulate conditions, e.g. to compare values.
- **Logical variables** that can keep the results of checking the conditions.
- **Logical operators** for manipulating logical values and composing complex conditions.
- **Selection structures** in the programming language (if-else-end in the example above), which allow us to choose one of the alternative paths of computations.

Logical variables and logical operators

- **Logical variable** can take only two logical values: **True** and **False**.
- In MATLAB, logical variable takes numerical values: *Non-zero means True, 0 means False*.
- Any numerical variable in MATLAB can be treated as a logical one.
- **Logical operators** are operations with logical values, which return logical values and implement the Boolean logics (algebra) that is the low-level basis of all computations in digital computers.
- Rules about four logical operators, **and**, **or**, **xor**, and **not** can be summarized in a **truth table**

INPUT		OUTPUT				
A	B	AND A&B	OR A|B	XOR (A,B)	NOT ~A	NOT ~B
false	false	false	false	false	true	true
false	true	false	true	true	true	false
true	false	false	true	true	false	true
true	true	true	true	false	false	false

Problem 1.8.1: a = -1 ; b = 0 ; c = - 2 * xor (a, b) + ((a | b) & ~a). Result ? Why ?

Relational operators

- **Relational operators** make a comparison of two arithmetic expressions and calculate the result of the comparison in the form of a logical value.

Operator	Description	Math. Notation
<	Less than	$<$
<=	Less than or equal to	$\leq$
>	Greater than	$>$
>=	Greater than or equal to	$\geq$
==	Equal to	$=$
~=	Not equal to	$\neq$

<= : Valid

=< : Invalid

Examples:

a = 12.0

b = -12.0009

c = a < b

d = a < abs (b)

e = a < abs (b) & a <= 0

- There are strict rules that define the priority (order of evaluation) of all operations and operators in the MATLAB, see Gilat, page 178. If we are not sure about the default order of evaluation of expressions, we must use parenthesis '()' in order to set the order manually.

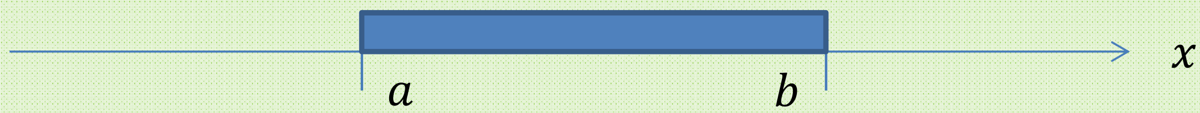
- **Logical/Conditional expression** is an expression with arithmetic operations and logical and conditional operators.

  Example: a = 1, b = 3 ; c = (b < a) & a.

1.7. MATLAB relational operators, conditional statements, and selection structures I

- Combination of logical and relational operators allows one to combine simple conditions into complex ones.

Problem 1.7.2: Introduce logical variable Flag which is true if and only if $a < x \leq b$ (x is within the interval (a, b) or $x = b$)

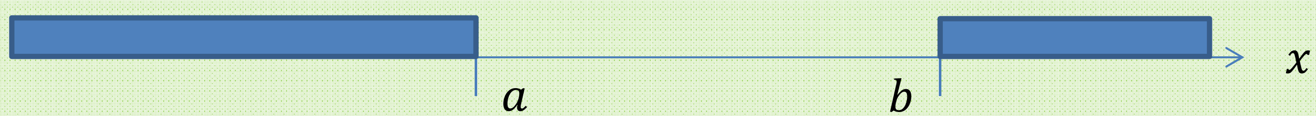


Solution 1: Flag = (a < x) & (x <= b)

Solution 2: Flag = a < x & x <= b % Relational operators have higher priority than logical ones

Solution 3: Flag = **and** (a < x, x <= b)

Problem 1.7.3: Introduce logical variable Flag which is true if and only if $x < a$ or $x > b$ (x is outside the interval $[a, b]$)



Solution 1: Flag = (x < a) | (x > b)

Solution 2: Flag = x < a | x > b

Solution 3: Flag = **or** (x < a, x > b)

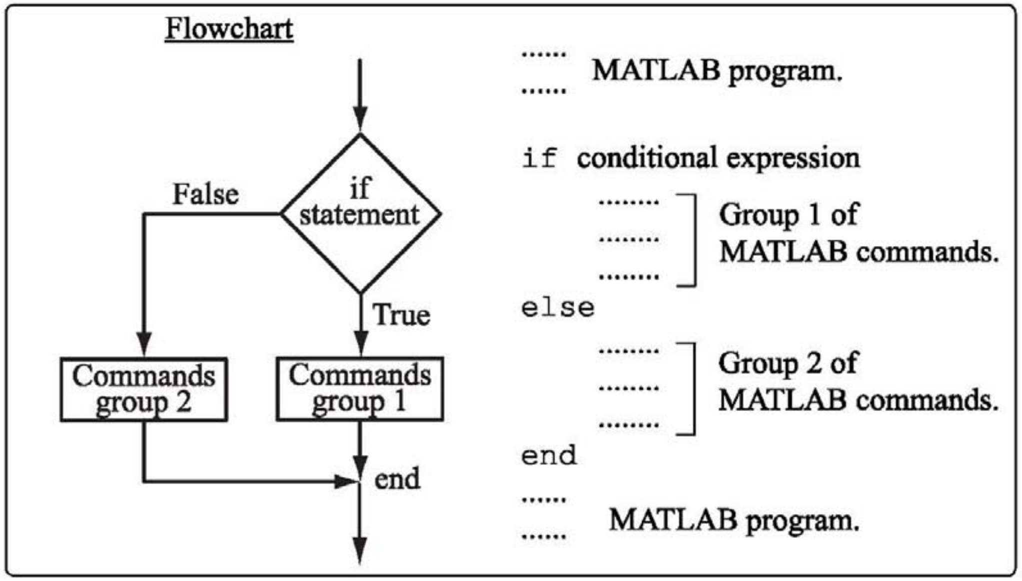
Solution 4: Flag = ~ (a <= x & x <= b)

- Conditional structure if-else-end (two alternative paths)
- Conditional structure if-else-end (many alternative paths)
- Nesting of selection structures
- return command

Reading assignment

Gilat, 6.2, 6.3, and 6.5

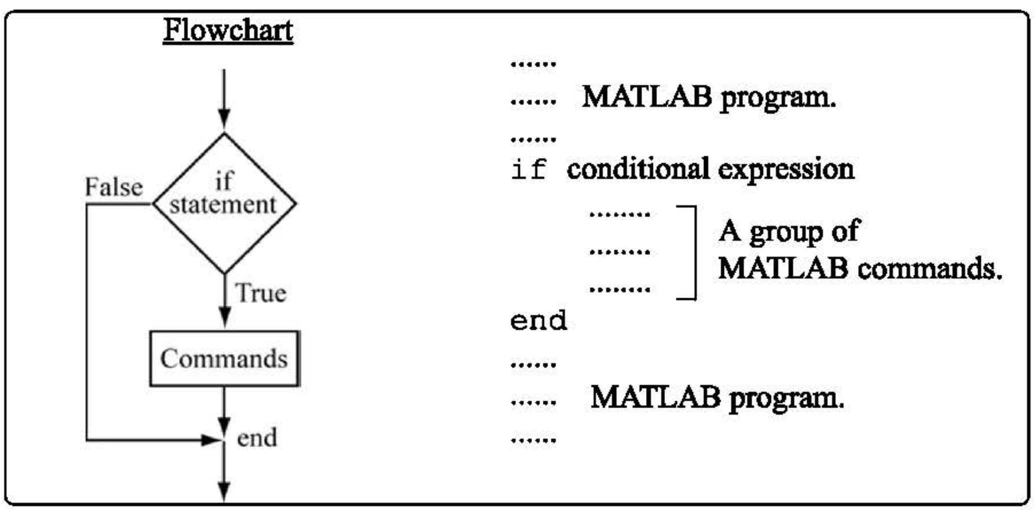
if-else-end structure (two alternative paths, arbitrary condition)



Example: Calculation of minimum of two real numbers

```
function [ res ] = min ( a, b )
   if a < b                 Condition
        res = a ;           Group 1
   else
        res = b ;           Group 2
   end
end
```

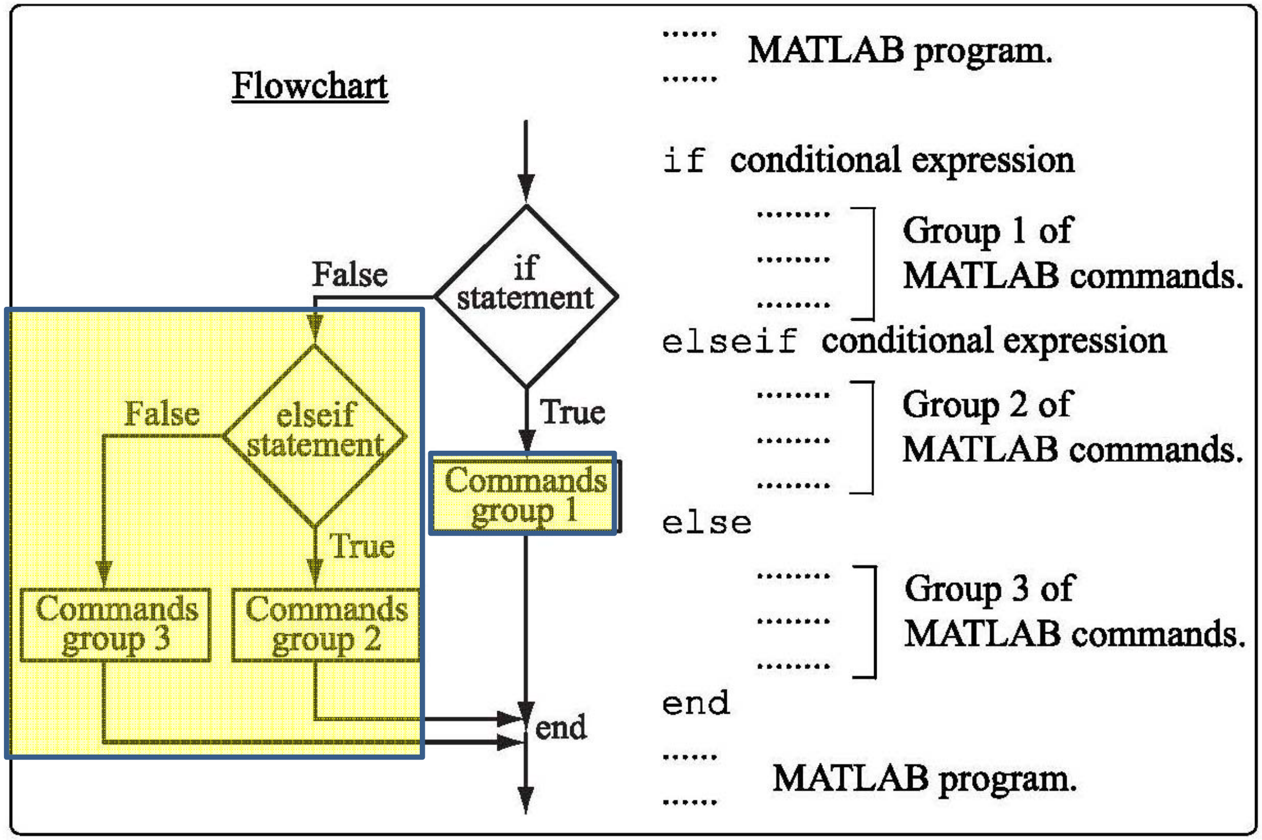
If Group2 is empty, keyword **else** is omitted




```
function [ res ] = min ( a, b )
   res = b ;
   if a < b                 Condition
        res = a ;           Group 1
   end
end
```

- Condition can be an arbitrary expression

If-elseif-else-end structure
(many alternative paths, arbitrary condition)



Example: Calculation of the sign of a real number.

```
function [ res ] = sign ( x )
   if x < 0
        res = -1 ;
   elseif x > 0
        res = 1 ;
   else
        res = 0 ;
   end
end
```

Total number of branches (groups) = Total number of conditions + 1

1.8. MATLAB relational operators, conditional statements, and selection structures II

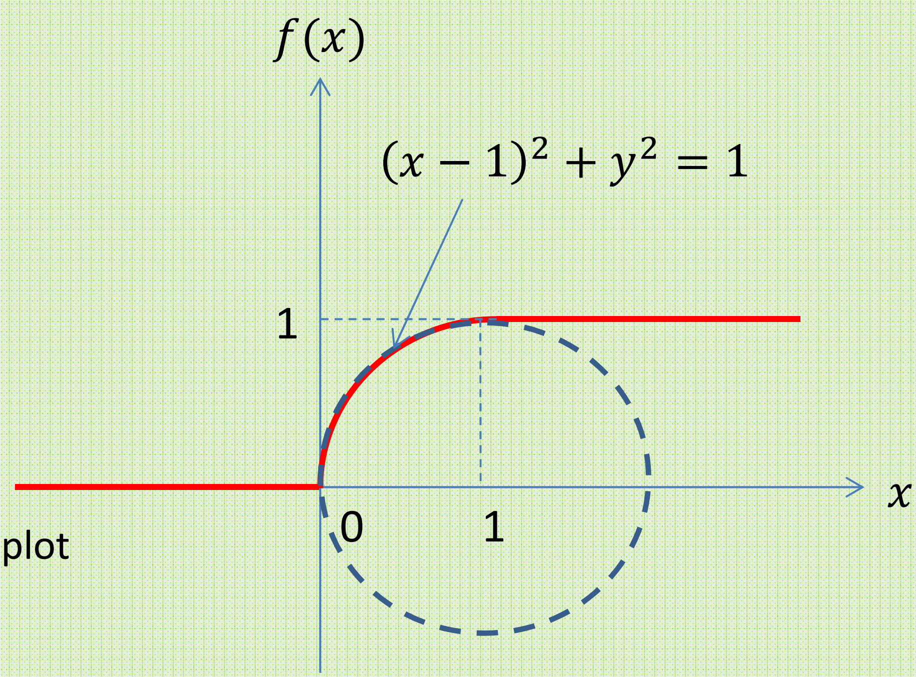
Problem 1.8.1: Conversion energy into Joules

1 cal = 0.239 J

1 eV = 6.24e+18 J

```
function [ res ] = GetE_J ( E, Unit )
%GetE_J Converts energy to SI units (Joules)
    if strcmp ( Unit, 'J' )
            res = E ;
    elseif strcmp ( Unit, 'cal' )
            res = E / 0.239 ;
    elseif strcmp ( Unit, 'eV' )
            res = E / 6.24e+18 ;
    else
            res = NaN ;
    end
end
```

Problem 1.8.2: Program function $f(x)$ given by the plot




```
function [ F ] = CFun ( x )
    if x < 0.0
            F = 0.0;
    elseif x < 1.0
            F = sqrt ( 1.0 - ( x - 1.0 )^2 );
    else
            F = 1.0;
    end
end
```

In order to execute such user-defined function to an array one can use **arrayfun** function:

```
x = [ - 3 : 0.01 : 3 ];
y = arrayfun ( @CFun, x );
plot ( x, y )
axis equal
```

1.8. MATLAB relational operators, conditional statements, and selection structures II

- if-else-end structures can be **nested** in arbitrary combinations.
- **Nesting** means placement of one structure inside another.

Problem 1.8.3: Solve problem 1.8.3 using nested if-else-end

File FFunNestedIf.m

```
function [ F ] = CFunNestedIf ( x )
    if x < 0.0
            F = 0.0;
    else
            if x < 1.0
                    F = sqrt ( 1.0 - ( x - 1.0 )^2 );
            else
                    F = 1.0;
            end
    end
end
```

⇔

```
function [ F ] = CFun ( x )
    if x < 0.0
            F = 0.0;
    elseif x < 1.0
            F = sqrt ( 1.0 - ( x - 1.0 )^2 );
    else
            F = 1.0;
    end
end
```

Alternative solution

```
function [ F ] = CFun ( x )
    if x < 0.0
            F = 0.0; return;
    elseif x > 1.0
            F = 1.0; return;
    end
    F = sqrt ( 1.0 - ( x - 1.0 )^2 );
end
```

- Here we use command **return** in order to immediately terminate the execution of the function (not necessary for the exam)

1.9. MATLAB loops

- Pre- and post-condition loops
- while-end loop
- for-end loop
- Calculation of mean and standard deviation of tabulated data
- Sorting
- Calculation of a polynomial function

Reading assignment

Gilat, 6.4-6.6

Why do we need loops?

In many problems, we need to repeat some commands.
Two common situations:

- Analyzing large arrays of data, we often perform **similar operation on every element** of arrays.
  Example: Calculations of the average value of N elements of array X
  1. Xa = 0.0
  2. **Repeat** Xa = Xa + X[i] for i = 1, 2, 3, ..., N
  3. Calculate average Xa = Xa / N

$$X_a = \frac{X_1 + X_2 + \cdots + X_N}{N} = \frac{\sum_{i=1}^{N} X_i}{N}$$

- Some calculations involving only scalar variables require **iterations**.
  Example: Calculation of the factorial $n! = n(n-1)(n-2) \ldots 2\ 1$ can be performed as:
  Fact = 1 ; Fact = Fact * 2 ; Fact = Fact * 3; ...
  or
  1. Fact = 1
  2. **Repeat** Fact = Fact * i for i = 2, 3, ..., n

Algorithmic structures called **loops** provide us with the possibility to repeat some portions of codes or to perform multiple **passes** of the same code.

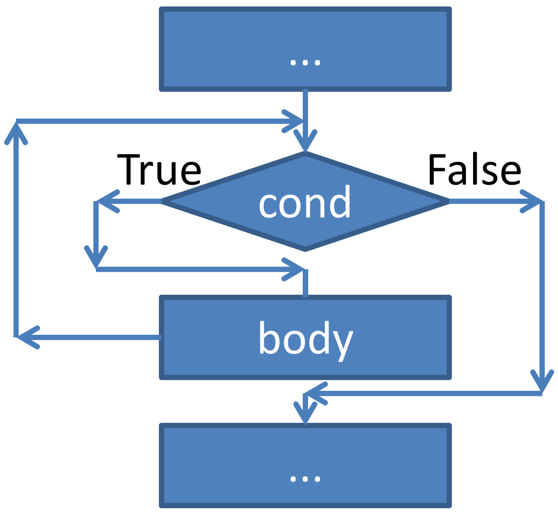
1.9. MATLAB loops

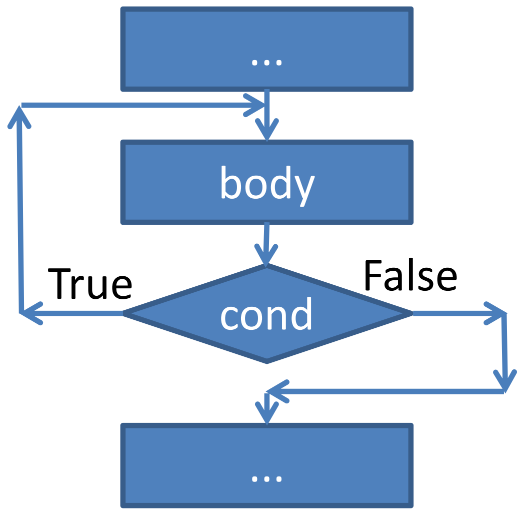
Loops

- The **loop** is a part of the code that is repeatedly performed if some condition is satisfied.
- Any loop includes at least two parts: **Condition** and **Body**.
- Condition is used to determine that the passes of the body of the loop should be ceased after some number of passes. For this purpose, condition should include some variables, which values are modified within the body of the loop.
- Flowcharts of loops:

Pre-condition loop:

Body may be inaccessible





Post-condition loop:

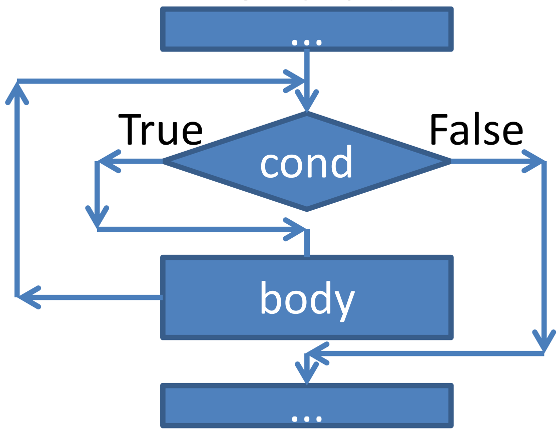
Body will be executed at least once

- Pre-condition and post-condition loops are different by the order of condition and body.
- The MATLAB has only pre-condition loops. Using if-else-end structure along with special **break** and **continue** commands (not necessary for the exam) allow us to turn a pre-condition loop into the post-condition one and vice versa.

while-end loop

- while-end loop is the general-purpose pre-condition loop.
- Condition is an arbitrary logical statement.

Flowchart



Code

```
...
while cond
    body
end
...
```

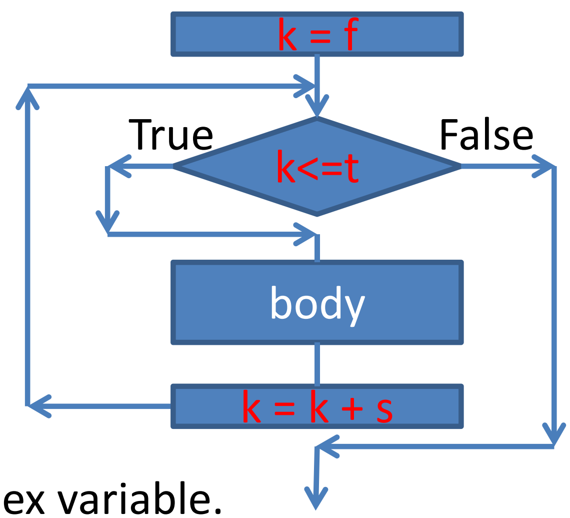
Problem 1.9.1: Write function FactorialW that calculates $n! = n(n-1)(n-2)\ldots 2\ 1$

```
function [ Res ] = FactorialW ( n )
%FactorialW Calculates the factorial of the integer value n
    Res = n ;
    i = n - 1 ;
    while i > 1            Condition
        Res = Res * i ;    Body
        i = i - 1 ;
    end
end
```

for-end loop

- for-end loop is the pre-condition loop that is designed to execute the body of the loop a **predetermined number of times.**
- for-end loop includes definition of the integer **loop index variable** that serves to count the number of times.
- for-end loop is often use to make calculations with elements of arrays.
- A condition of a special form involving the loop index variable is used.

Flowchart



k is the loop index variable.
f is the value of k in the first pass.
t is the value of k in the last pass.
s is the increment of k between passes.
Negative increments are available.

Code

```
...
for k = f:s:t
    body
end
...
```

The loop index variable should not be re-defined in the loop body!

Special cases:

- k = f:t means s = 1.
- if t = f the loop is executed once.
- if f > t and s > 0 or f < t and s < 0, the loop is not executed.
- k = [1 7 12 -5] : Loop will be executed for the specified values of k.

1.9. MATLAB loops

Problem 1.9.2: Solve problem 1.10.1. using for-end loop

File FactorialF.m

```
function [ Res ] = FactorialF ( n )
    Res = 1 ;
    for i = 2 : n
        Res = Res * i ;
    end
end
```

Calculation of mean and standard deviation of tabulated data

Let's assume that we have distribution of some variable given in a tabulated form:

i	1	2	…	i	…	N
X	X_1	X_2	…	X_i	…	X_N

Three numerical characteristics are systematically used for analysis of such distribution.

Mean μ or **expectation** $E(X)$ is the arithmetic mean of values in the table:

$$\mu = E(X) = \frac{X_1 + X_2 + X_3 + \cdots + X_N}{N} = \frac{1}{N}\sum_{i=1}^{N} X_i.$$

Mean gives us value around which the most of points in the table is concentrated.

1.9. MATLAB loops

Variance $\sigma^2 = V(X)$ is the arithmetic mean of squares of deviations of individual values from the mean:

$$\sigma^2 = V(X) = \frac{(X_1-\mu)^2 + (X_2-\mu)^2 + \cdots + (X_N-\mu)^2}{N} = \frac{1}{N}\sum_{i=1}^{N}(X_i-\mu)^2.$$

Standard deviation σ is equal to

$$\sigma = \sqrt{V(X)}.$$

Standard deviation is the measure of deviation of values in the table from the mean. The larger the standard deviation, the broader distribution of values in the table around mean.

It is convenient to calculate variance in the form:

$$\sigma^2 = V(X) = \frac{1}{N}\sum_{i=1}^{N}\left(X_i^2 - 2\mu X_i + \mu^2\right) = \frac{1}{N}\sum_{i=1}^{N}X_i^2 - 2\mu\frac{1}{N}\sum_{i=1}^{N}X_i + \mu^2$$

or

$$\sigma^2 = V(X) = \frac{1}{N}\sum_{i=1}^{N}X_i^2 - \mu^2.$$

1.9. MATLAB loops

Problem 1.9.3: Write a function MeanStd that calculates the mean and standard deviation of values in the table given by array x

File MeanStd.m

```
function [ Mean Std ] = MeanStd  ( X )
    [ m N ] = size ( X ) ; % Here we assume that m = 1, i.e. x is a row vector
    Mean = 0.0;
    Std = 0.0;
    for i = 1 : N
        Mean = Mean + X(i);
        Std = Std + X(i)^2;
    end
    Mean = Mean / m;
    Std = sqrt ( Std / N - Mean^2 );
end
```

$$\mu = \frac{1}{N}\sum_{i=1}^{N} X_i.$$

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} X_i^2 - \mu^2.$$

Note: build-in MATLAB functions **mean** (x) and **std** (x) can be used in order to calculate mean and standard deviation of tabulated data.

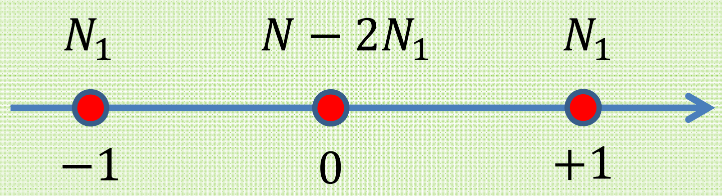
1.9. MATLAB loops

Problem 1.9.4: Let's calculate the mean and standard deviation for the case when among $N = 100$ values we the have only three values 0, +1, and -1 in the table:

N_1 values are equal to -1;

N_1 values are equal to +1;

$N - 2N_1$ values are equal to 0;



Solution of the problem is given in the file Problem_1_9_4.m.

if N1 = 10 then Mean1 = 0 Std1 = 0.4472

if N1 = 30 then Mean2 = 0 Std2 = 0.6325

if N1 = 40 then Mean3 = 0 Std3 = 0.8944

Value of the standard deviation increases with increasing number of points with values ± 1 and approaches the intuitively expected value 1 when N_1 approaches 50.

1.9. MATLAB loops

Sorting

Problem 1.9.5: Write a function Sort that sorts elements of the array in the ascending order.

File Sort.m

```
function [ y ] = Sort ( x )
    y = x ;
    [ n m ] = size ( x ) ; % Here we assume that n = 1, i.e. x is a row vector
    for i = 1 : ( m - 1 )
        for j = ( i + 1 ) : m
            if y(i) > y(j)
                a = y(i);
                y(i) = y(j);
                y(j) = a;
            end
        end
    end
end
```

1	4	2	-3	3	7	4

1. Compare x(1) step-by-step with x(2), x(3), ...
if x(1) > x(i), swap x(1) and x(i)
Now x(1) is the min. element of the array

-3	4	2	1	3	7	4

2. Compare x(2) step-by-step with x(3), x(4), ...
if x(2) > x(i), swap x(2) and x(i)
Now $x(1) \leq x(2) \leq \cdots$.

-3	1	4	2	3	7	4

3. Continue these steps for all elements until x(m-1)

1.9. MATLAB loops

Example: Perform sorting of array x = [1, -2, -3, -4] in the ascending order.

```
function [ y ] = Sort ( x )
    y = x ;
    [ n m ] = size ( x ) ;
    for i = 1 : ( m - 1 )
        for j = ( i + 1 ) : m
            if y(i) > y(j)
                a = y(i);
                y(i) = y(j);
                y(j) = a;
            end
        end
    end
end
```

Individual rows in the table below show contents of array **y** after every step of sorting, i.e., after executing of if-else structure ag given values of **i** and **j**

i	j		1	-2	-3	-4
1	2		-2	1	-3	-4
	3		-3	1	-2	-4
	4		-4	1	-2	-3
2	3		-4	-2	1	-3
	4		-4	-3	1	-2
3	4		-4	-3	-2	1

1.9. MATLAB loops

- If-else-end, while-end, and for-end structures can be nested in arbitrary permutations
- Nested structure must be placed completely inside it's external structure.
- Function [n m] = **size** (x) returns the number of rows, n, and columns, m, in array x.

Polynomial function

The **polynomial function of degree N** is the function

$$f_N(x) = C_N x^N + C_{N-1} x^{N-1} + \cdots + C_2 x^2 + C_1 x + C_0 = \sum_{n=0}^{N} C_n x^n \qquad (1.10.1)$$

where C_i are arbitrary coefficients.

For the computationally efficient calculations, we can re-write equation in the form

$$f_N(x) = ((\ \ldots\ (C_N x + C_{N-1})x + \cdots + C_2)x + C_1)x + C_0$$

e.g.

$$f_3(x) = C_3\ x^3 + C_2 x^2 + C_1 x + C_0 = \ ((C_3 x + C_2)x + C_1)x + C_0$$

- A polynomial function is given by its degree N and the array of coefficients C_i .
- Polynomial of degree N has $N + 1$ coefficients.
- We will use **row** vector C = [**C_N** C_N-1 ... C_2 C_1 **C_0**] in order to store coefficients.

1.9. MATLAB loops

Problem 1.9.6: Develop a function PolyVal (C, x) which calculates a polynomial function.

File PolyVal.m

```
function p = PolyVal ( C, x )
   [ M, N1 ] = size ( C );  % N1 is the degree of the polynomial + 1
   p = C(1);
   for i = 2 : N1
      p = p * x + C(i);
   end
end
```

Calculation of polynomials in the MATLAB

- The build-in **polyval** function can be used in order to calculate the value of the polynomial function in the form given by Eq. (1.10.1):

```
C = [ C_N C_N-1 ... C_2 C_1 C_0 ];     % This is the array of coefficients
f = polyval ( C, x ) ;                 % The degree of the polynomial is determined
                                       % by the number of coefficients
```

Example: Calculation of the polynomial $f(x) = -2 + 3\,x \, - 2.5\,x^2$ at $x \; = \; -7$:

```
C = [ -2.5, 3, -2 ];
f = polyval ( C, -7 ) ;
Myf = PolyVal ( C, -7 ) ;
```

1.11. Summary

For the exam we must know how

- To perform arithmetic calculations in the MATLAB command window.
- To evaluate expressions containing conditional (>, <, etc.) and logical (|,&,etc.) operators.
- To create and manipulate one-dimensional arrays.
- To use vectorised mathematics for calculations with arrays.
- To create, edit, and run scripts and to comment scripts.
- To create, edit and run functions with comments.
- To use if-elseif-else-end, while-end, and for-end algorithmic structures.
- To plot two- and three-dimensional plots, to format plots, and to export plots to graphic files.
- To calculate means and standard deviation of tabulated data.
- To sort elements of an array in ascending/descending order.
- To plot a function given by a sketch using arrayfun function.
- To calculate arbitrary polynomial function and plot its graph.

- Keywords: **function**, **end**, **if**, **elseif**, **else**, **while**, **for, return**.
- Commands: **help**, **lookfor**, **clc**, **clear**, **format, print**.
- Functions: **sqrt**, **sin**, ..., **atan2**, **linspace**, **size, sum, min, max, dot, cross, plot**, **line**, **semilogx, semilogy, loglog**, **xlabel**, **ylabel**, **text**, **axis**, **arrayfun, strcmp, mean, std, polyval.**