Introduction to MATLAB and its operations

Includes non textbook material and also Chapter 1 (1.1-1.5) and

Chapter 2 (2.1-2.2,2.5-2.15) and Chapter 3 (3.1-3.3) and Chapter 6 (6.2.1,6.2.7-6.2.9,6.4)

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2014 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS 101 : Fall 2014 . All rights reserved.

As we mentioned in the previous chapter, programs and data are stored in the main memory of a computer. Both program and data are stored as sequences of a collections of binary digits i.e. as a sequence of bytes. The fundamental aggregate is thus the **byte**, a collection of 8 **bits**.

2.1. Machine code. If one wants to program a computer in the absence of helpful tools, he needs to manipulate bits and bytes at a very low level according to a specific computer-dependent format that is known as the **machine code** or **machine language**. This is tedious, complicated, error-prone, confusing and buggy.

For the sake of an example, think of the pretty tedious task of writing a simple integer, say `1` into memory. An integer nowadays is represented by 32 bits (i.e. a word) in two's-complement representation, and thus following the notation and discussion of the previous chapter its binary representation might look like the bit sequence `00000000 00000000 00000000 00000001` with spaces added every eights bits for readability and to highlight the bytes in this 32-bit word. Suppose we intend to store this integer in main memory and let memory locations 10,11,12,13 be used for this purpose. How do we store those 4 bytes in those 4 memory locations, one byte per location? Depending on the CPU or microprocessor, one or the other byte-order shown below might be used. A microprocessor using the former byte-order is called a big-endian processor, as opposed to the alternative; Intel processors are little-endian.

```
Big-Endian                              Little-endian
10       11       12       13          10       11       12       13
00000000 00000000 00000000 00000001    00000001 00000000 00000000 00000000

Byte order for storing 32-bit integer 1 = 00000000 00000000 00000000 00000001
```

2.2. Assembler (Assembly language). If machine code is too complicated and too bit- or byte-oriented, one can avoid to use such low-level manipulation/programming by programming at a higher level of abstraction by using an **assembly language**. An assembly language is a user-friendlier (but not very user-friendly) version of machine code/language in which the user (the programmer) uses mnemonic codes to execute machine code instructions. In other words, the input to an assembly language is a higher-level language (eg. mnemonics) converted into the output language which is machine code. For example instead of using machine code `10101011`, one could use a mnemonic code `STORE A` that writes back into memory the contents of processor register `A` (a register is a small piece of memory located on the CPU that can be accessed very quickly and efficiently).

(c) Copyright A. Gerbessiotis. All rights reserved.

2.3. Compiler. Even assembly language is too low-level for most programmers and end-users. It is also machine-dependent. A user can use instead a **compiler** that translates code written in a higher-level programming language that is mostly machine independent (for example one can write a piece of code in `C, C++, Java` that is identical on a Windows PC, a Sun Workstation, a Linux PC, an Apple OS-X laptop) into a lower-evel language that can then be executed by the target platform (Sun workstation or Windows PC). The lower-level language can be assembly (and then needs to be converted by an assembler into machine code) or even plain machine code that can just be executed directly.

Thus when using a compiler a user performs two independent tasks:

(a) the user first writes a program in a higher-level programming language that is saved into a file (or more than one files), and then compiles the file(s) using a compiler, with the compiler converting these file(s) into assembly or machine code,

(b) the object code (generic for machine code or assembly that is almost ready for execution) thus generated by the compiler can then be separately executed by the CPU.

Thus in executing a program with the use of a compiler there are at least two distinct steps: compilation (step (a)) and execution (step (b)). In several instances, compilation might require more steps because for example the compiler sometimes does not translate higher-level language code into assembly code directly but to some intermediate code, that is then translated into some machine-independent assembly and so on. And if someone made a mistake in the higher-level language program, both steps need to be repeated.

For example, a C program `testprog.c` can be compiled by a command `gcc testprog.c -o testprog` that compiles the source code `testprog.c` that contains the source program into an executable file named `testprog`, and then the executable file is run by issuing a `./testprog`.

2.4. Interpreter. If the two steps of a compiler-based program execution seem complicated, an interpreter can be used that can execute directly (or indirectly) higher-level programming language statements with out the requirement for separate compilation and execution. **MATLAB** (initials for **MAT**rix **LAB**oratory), Python, Perl, Ruby, csh/tcsh/bash, Lisp, are interpreted programming languages that is, programs written in them can be executed directly by an interpreter of the language. The interpreter for **MATLAB** is slightly more complex and intelligent. It executes several precompiled pieces of code that are available to it, and compiled by a compiler that is part of **MATLAB** itself.

(c) Copyright A. Gerbessiotis. All rights reserved.

In this course we will learn how to write small programs that implement efficiently computer-based solutions of computational problems.

2.5. Algorithm. An **algorithm** is a well-defined (i.e. correct) sequence of computational steps that takes some value or set of values as input and generates some value or set of values as an output.

An algorithm is thus a method/tool for solving well-specified computational problems. A (computational) *problem* defines an input-output relationship. An **algorithm** describes a specific procedure for achieving this relationship, i.e. for each problem we may have more than one algorithms that solve it.

Before one embarks on solving a computational problem with a computer (and MATLAB) one needs to understand the problem itself i.e the input and output relationship of the problem. Computers are inherently precise and error-prone (buggy) and human beings are inherently imprecise. Combining (or coordinating) the two can have very disastrous consequences.

For example some time ago a piece of code contained the code `DO 10 I=1.10` written in programming language FORTRAN. The intent was to write `DO 10 I=1,10`. This piece of code was an orbit computation program that was used in NASA's Mercury flights in the early 60s. However when a user started running this program with known input and output data, the output the program was generating was close to but not identical to the known output data. Only because of this discrepancy was the missing comma (or incorrectly appearing period) discovered, and disaster was averted.

Consider for example the very simple problem of addition of numbers. One could describe the input output relationship for this problem as follows.

```
Input:  Two numbers a and b
Output:  a+b
```

Every NJIT student can understand what the problem is. Finding an algorithm that realizes a solution to that problem is also easy (it probably requires elementary school background). However the description of the problem is imprecise and writing code that solves it might be a complicated task even for a Computer Science student.

(c) Copyright A. Gerbessiotis. All rights reserved.

Question 2.5.1. Why is it so?

Answer. The user never specified explicitly how long the numbers can be.

- The user never specified whether a, b are natural (positive integer) numbers or
- arbitrarily sized integer numbers (say, `13213239219321` or `-23123210321020210210`),
- in decimal or binary or hexadecimal notation or some other notation,
- or fractionals (say, `1232132/1323223`),
- or real numbers (say, `232321.213213223`),
- or a mix of all of these,
- or even a complex number (i.e. $21323.23 + i2132$),
- or something else.

The same applies to the output.

Question 2.5.2. Why specifying the input form is important.

Answer. Computers are not as flexible as human beings. For example, computers are susceptible to attacks (by criminals) known as **buffer overflow attacks**. Because the input (and output) specification is imprecise, a reckless (or not very careful) programmer might make some generic assumptions on how large the input can be. A careless or intentionally devious user might supply a non-conforming input that would crash the program or force it to behave unpredictably and harmfully.

2.6.a What is a function (mathematics) and what is a function (computer science)? In mathematics a function is a relation between a set of elements (the domain) and another set of elements (the codomain) defined so that each element of the domain is associated with exactly one element of the codomain. In computer science (or programming), a function is a piece of code that performs a specific task. A function in computer science is sometimes called a subroutine, subprogram, method, procedure, routine, or just program (if it performs only a single task).

2.6.b Function names in mathematics and computer science and MATLAB. In mathematics when we write `incr`(x), we so denote the INCREMENT functions that returns $x+1$ for input x. If we decide to implement this function, we use the same name `incr` or something different that is more informative (e.g. `incrementbyone`). Of course using such a long name raises readability issues: we might write instead `increment_by_one` by using underscores to separate words. For other cases, a more drastic action is need; for example most of us will be hard-pressed to type in a keyboard $\sqrt{x}$. Instead, we usually write something like `sqrt`(x) to denote the $\sqrt{x}$ function. Thus for some very familiar mathematical functions (e.g. those appearing in a scientific calculator), MATLAB might be using different names than the ones with which we are familiar.

2.6.c How is a function defined ? In mathematics we could define `incr`(x) very simply by writing $f(x) = x + 1$. Math function names are very short and most of the times f, g, h are being used as names. When we write computer programs we need to be more imaginative: we can't name all of our functions f. In addition when we write x the implicit assumption is that x is a number. In MATLAB for example inputs to a functions can be a number, a vector (a linear collection of numbers), or a generic object called matrix (and do not forget that MATLAB stands for MATrix LABoratory). Given that computers are dummy, they do not know that f denotes a mathematical function and not say the sixth letter of the alphabet in lower-case. Thus when we write programs we need to provide lots of details about what we attempt to do (program) and what we expect as input and as an outcome (output). Is the output a number similarly to the input, or is it just a number despite the input being a vector?

(c) Copyright A. Gerbessiotis. All rights reserved.

2.6.d Parameter vs Argument. For example is $f(1/3)$ ok , i.e. is $x = 1/3$ a legitimate input? Are rational numbers allowed as an input? And if you say yes, how about complex numbers? If you don't know what a complex number is that is fine: our intend is to show you to expect the unexpected! Now that we are still talking about f, let us remind ourselves that x is the **parameter** of function f. In MATH sometimes we call x the **indeterminate**. When we evaluate f at a point and we ask 'What is the value of $f(1/3)$' we substitute $1/3$ for x. This value becomes the **argument** to the function. Thus **parameter** is something generic that is used to help us define a function, and argument is something specific that help us with the evaluation of a function. Several MATLAB functions are associated with filenames of the same name as the function itself. A simpler form is what is known an anonymous function that is not associated with a computer file (and name); an example appears on page 20XXXXXX. MATLAB functions might have different names than their MATH counterparts and might also behave differently. Be prepared!

When we write in math $f(x) = x^2 - 2x + 1$, we define a function on an unknown variable x (sometimes we call x the indeterminate). This specific function is called a polynomial. The degree of the polynomial is two, and sometimes we call $f(x)$ a quadratic function. Human beings are quite intelligent and can easily fill-in details. Computers are dummy unless designed by extremely intelligent humans; the presence of many bugs (i.e. irregular instructions that cause problems in a computer code/program) indicates that we still have to accept that computers are not very intelligent or human beings do not yet know how to program them intelligently.

2.6.e Human intelligence and inherent lack of precision

For a human $f(x) = x^2 - 2x + 1$ means $f(x) = x^2 - 2 \cdot x + 1$. (A $\cdot$ multiplication operator is implied between the 2 and the x.) When a human is asked `How much is` $f(0.5)$?, that human understands that the polynomial must be evaluated at $x = 0.5$, i.e. the human evaluates $f(0.5) = 0.5^2 - 2 \cdot 0.5 + 1$ and responds with the answer of `0.25`.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.6.f Computer precision and lack of intelligence
A computer on the other-hand needs guidance. It needs to be told what is expected to do.

- It needs to be told that it ought to evaluate a function.
- It needs to be told what the name of the function is going to be i.e. `f`.
- It needs to be told how the input is going to be made available i.e. `through a parameter to the function`.
- It needs to be **precisely** told what the range of values for the input parameter will be.
- It needs to be told what to do with the computed value i.e. `return it to the user who invoked the function` and what the range of values for the computed function value is going to be.
- It needs to be given precisely the input function (i.e. $\cdot$) or be told otherwise that $2x$ means $2 \cdot x$.
- It needs to be allocated in advance the scratch space where extra calculations can be performed.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.7. Constant. If the value of an object can never get modified, then it's called a constant. 5 is a constant, its value never changes (ie. a 5 will never have a value of 6).

2.8. Variable. In computer programs every parameter or argument whose value changes is a **variable**. Every variable has a **data-type**. What is the data-type of a variable?

2.9. Data-type. In a programming language, every variable has a data-type, which is the set of values the variable takes and the operations defined (or are allowable) on the data-type.

2.9.1 Built-in or primitive data-types. Composite data-types. In mathematics, an integer or a natural number is implicitly defined to be of arbitrary precision. Computers and computer languages have **built-in** (also called **primitive**) data-types for integers of finite precision. These primitive integer data-types can represent integers with 8-, 16-, 32- or (in some cases) 64-bits. An integer data-type of higher precision is only available not as a primitive data-type but as a **composite** data-type through **aggregation** and **composition** and built on top of primitive data-types. One way to build a composite data-type is through an aggregation called an array: an array is a sequence of objects of (usually) the same data-type. We use the word **sequence** over the word **set** because the former implies order whereas the latter does not. In a sequence we can specify its first, second, third, etc element: in a set elements have no order at all.

2.10 Data model. A Data model is an abstraction that describes how data are represented and used.

2.11.1 Strongly-typed and Weakly-typed computer programming languages. In languages that are **strongly-typed** a variable needs to be fully defined (i.e. its name is presented and its data-type specified) before it is used. In **weakly-typed** languages the first time a variable is used becomes the declaration point of the variable and implicitly serves as its definition. Its data-type is implied by the value assigned to the variable. This is convenient but sometimes confusing.

2.11.2 Declaration and definition. In computing we use the term **definition** of a variable to signify where space is allocated for it and its data-type explicitly defined for the first time, and **declaration** of a variable to signify our intend to use it. A declaration assumes that there is also a definition somewhere else, does not allocate space and serves as a reminder. For a variable there can be only ONE definition but MULTIPLE declarations. This discussion makes sense for compiled languages, for interpreted languages, definitions are usually not available and declarations coincide with the use of a variable.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.12 MATLAB. MATLAB is a weakly-typed language and thus we do not need to bother with a variable's definition or declarations. A variable is defined (and also declared) the first time it is being used.

2.12.1 MATLAB constants and special names. A number is a constant in MATLAB; its value can not change. So a 5 is a five, no matter what. MATLAB has a number of built-in constants that are assigned default values. Because these built-in costants are referenced as variable we prefer to call them `special names`. These are `pi` for `3.14...`, `i` and `j` for the square root of -1, and `true` and `false`. MATLAB variable `pi` has a fixed value that of π i.e. $3.14\ldots$. MATLAB variables i, j are in fact complex numbers and are defined to be the square roots of -1, i.e. $i = j = \sqrt{-1}$. If you want to convince yourselves about that, type in MATLAB $i * i$ or $j * j$. It is imperative that you not use these names as variable names. MATLAB also has special names for `Inf` and `Nan` for infinity and not-a-number (eg the division of infinity by infinity).

2.12.2 MATLAB variables : Naming

A user can name a variable in MATLAB in a very specific MATLAB-dictated way. The name of a MATLAB variable is a sequence of characters that is unique to its first 63 characters. That is, a MATLAB variable's name can be longer than 63 characters but only the first 63 count and identify uniquely the variable. The first character of a variable's name must be an `alphabetic` one, i.e. one of the 52 English lower-case or upper-case characters `A-Za-z`. The remaining ones can be `alphabetic`, `numeric` (i.e. the ten digits `0-9`) or the underscore _. (Note that there are only 63 choices for the 52 alphabetic characters, 10 digits and the underscore.) No other character is allowed to appear in a variable's name; thus among other things space and punctuation marks can not be part of a variable's name. Note that variables are **case sensitive**.

If you define a function in MATLAB the same convention applies to naming a function. Remember that MATLAB is a weakly-typed language. The definition or declaration of a variable is when it is being used.

The number 63 mentioned is not fixed in MATLAB and might change in future versions of it. In fact the number of characters that are important in identifying a variable is the value of a MATLBA built-in constant `namelengthmax` which gives the number of significant characters recognized by the current MATLAB implementation (currently 63).

Example 2.12.3 Thus `alex1` and `alex_1` are ok, but `alex 1` and `1alex` are not valid MATLAB variables.

Example 2.12.4. The variables `alex`, `Alex` and `ALEX` are all different.

Example 2.12.5 : Assigning a value If you want to assigne a value to a variable you write something like `x = 10`. This sets the value of constant `10` to `x`. If you write `x= y`, then MATLAB first evaluates y and then the value of y is assigned to x. If y does not exist, MATLAB will produce an error. Errors in MATLAB are in red color.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.12.6 MATLAB variable : ans. MATLAB variable `ans` is short for `answer`. If you compute an experssion in MATLAB whose value is not assigned to a variable. MATLAB uses variable `ans` to assign the value of the expression to that default variable. Although you may read the value of `ans` and you may change it, be very very careful as the value of `ans` changes often and autmatically by MATLAB.

2.12.7 MATLAB reserved words: Stay clear of them!

MATLAB is a weakly-typed language and thus it allows use to mess with its variables. So one can write in MATLAB a $pi = 4$. This is one good reason to avoid using such names. Other variables you should not be redefining are `ans, true, false`. All of these are also called `reserved words.` More **keywords** that should also not be used as variables are. (For now we use the terms reserved words and keywords interchangeably. The former might be used without MATLAB generating an error or a warning but if you use the latter an error will be flagged.) Variable `ans` is the variable that MATLAB is using to store the value of expressions written in the command window that are not assigned to a variable.

```
A sample of MATLAB's keywords...

 break case catch continue end else elseif for function global if otherwise persistent
 return switch try while
```

2.12.8 MATLAB caution: playing with fire (aka built-in constants/special names). MATLAB is a weakly-typed language and thus it allows us to mess with its variables in sometime unpredictable ways. One can write $pi = 4$. What one does with this statement is to create its own local variable in the user's WORKSPACE that overrides the default value for variable pi. The system pi still exists but from that point on the user will have access only to his own same named variable whose value is 4. This is one good reason to avoid using such MATLAB special names that we called them `reserved words.` or **keywords** before. To restore order, do a `clear pi` and this will wipe out YOUR pi.

2.12.9 Finding a user's MATLAB variables: The who and whos commands

All variables introduced by a user can be retrieved by a `who` or `whos` in MATLAB's command prompt, that shows all the variable introduced in the user's **WORKSPACE**. In MATLAB we can view the internal representation of numbers or strings by using the function `whos` that shows the variables defined, their internal data-type plus size information.

2.12.10. MATLAB help.

If something is not clear in MATLAB type `help` followed by a words for which assistance is needed. Thus `help whos` (not the space between the two words) will provide assistance for the MATLAB command `whos`.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.12.11 MATLAB primitive data-types for numbers. MATLAB supports a variety of integer data-types: 8-,16-,32-, and 64-bit signed (two's complement) and unsigned integers. Also single (32-bit) and double (64-bit) precision real numbers in floating-point format. These data-types are called in MATLAB `int8, int16, int32, int64, uint8, uint16, uint32, uint64, single, double` respectively.

2.12.12 Default data-type for a MATLAB number: double The default data-type for a number in MATLAB is `double`. Thus when one writes 10, or 10.0 then MATLAB interprets the 10 as a real number and between the two data-types for real numbers it uses `double` to represent the ten. If you want to use one of the alternative representations you need to explicitly ask MATLAB to do the conversion; see below.

2.12.13 Default printing for a MATLAB number: few decimal digits By default MATLAB prints a number with few decimal digits. This does not mean that a number internally is stored with few decimal digits. The default printing of numbers in MATLAB is `format short`. You may affect the output (BUT NOT THE INTERNAL REPRESENTATION OF A NUMBER) by using `format long` instead.

2.12.14 MATLAB logical data-type. The result of a relational (or logical operators) that evaluate to a true (in the form of 1) or false (in the form of 0) are stored not in a `double` but in a `logical` data-type that uses only 1 byte. The value of `true` or `false` built-in constant in MATLAB is a 1 or 0 whose data-type is `logical`.

2.12.15 Characters and strings in MATLAB. Characters are represented in MATLAB as UNICODE characters (in 16 bits i.e. 2B). A **string** is a sequence of characters. A MATLAB string of n characters uses $2n$ bytes. A string in MATLAB may contain 0 characters i.e. it could be empty. The data-type **char** for character comes from the programming language C. MATLAB is implemented in the programming language called `C`. Thus it uses data types available in C. A **string** is an array of 0 or more characters in MATLAB. In order to signify the beginning and end of a string when we write it down we use one single quotation mark such as `'`. Thus `'abc'` defines a string of three characters. The length of a string can be computed using MATLAB function `length` thus `length('abc')` would return 3, but `length('')` would return 0.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.12.16 Conversion. One can ask MATLAB to store a number in a specific data-type by enclosing it into the name of the data-type. Thus `int32(10)` stores 10 as a 32-bit signed (two's complement) integer, and `int8(10)` as an 8-bit two's complement signed integer. Note that an `int8` data-type can only represent in two's complement the integers from -128 to 127 (inclusive). Thus `int8(128)` and `int8(-129)` will return 127 and -128 respectively, the largest and smallest (most negative) numbers that can be represented in 8-bit two's complement. There is no reason typing `double(10)` over `10` as the latter writing is equivalent to doing the former. If you want to convert a character or a string to its UNICODE number do an arithmetic calculation on it such as `'abc'+0` and it will show you the UNICODE code for characters `a, b, c`.

2.12.17 Bonus `format hex`: Use `format hex` to view integers in hexadecimal. Note that 10 is not a integer but `int8(10)` is one!

2.12.18 MATLAB variable : Assigning a value If you want to assigne a value to a variable you write something like `x = 10`. This sets the value of constant `10` to `x`. (This is a repetition of 2.12.5.) The data-type of variable `x` will then be the data-type of 10, i.e. a double!

(c) Copyright A. Gerbessiotis. All rights reserved.

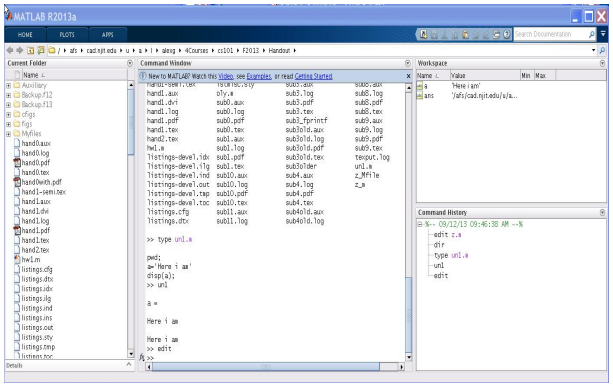


Figure 1: MATLAB Window view

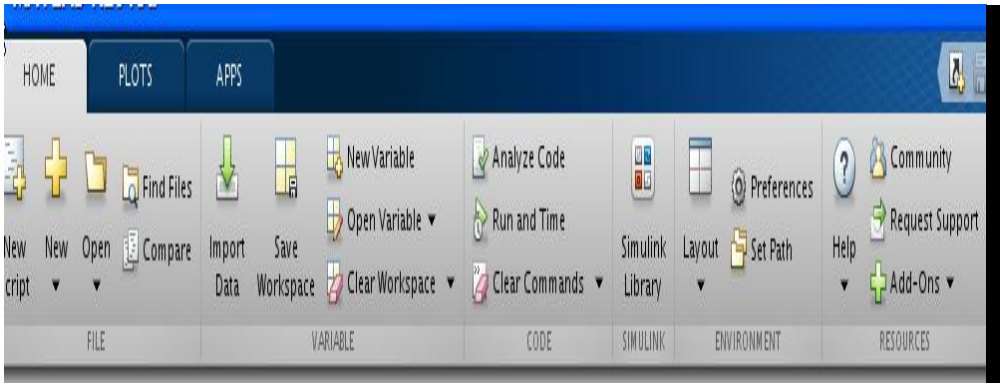


Figure 2: Toolstrip

2.13 Window view of MATLAB. A view of a MATLAB window for Matlab R2013a is shown in Figure 1. Behavior (default or otherwise) might be different in other recent versions of MATLAB and is definitely different from older versions of it. The Top pane of MATLAB contains three tabs, `HOME`, `PLOTS` and `APPS`. If you click once or twice on HOME the toolstrip (Figure 2 will disappear or re-emerge. Sometimes the `Toolstrip` appears attached to the `HOME` tab. If it is not shown click on `HOME` once to make it appear or once more to make it disapper. Somewhere in the middle to the right of the toolstrip you may find a button called `Layout`: this determines among others what windows are open (eg Command History, Workspace, etc) or permanently/by-default displayed (eg toolstrip). Note that such behavior or layout might be MATLAB version-dependent. Next to Layout you can find `Preferences`.

The MATLAB `Command Window` is centrally located and it is the widest one. On the left of the command window there usually is a vertical window called `Current Folder` that lists the contents of the current working directory/folder. On the right of the command window there are two vertical windows usually names `Workspace` and `Command history`. The `Workspace` is the user's workspace and can also be retrieved by typing `who` or `whos` in the command window. The `Command History` includes all the commands the user has entered in MATLAB's command window, not only in the current session but since the last time the user cleared it. If you want to clear the command history on the right of the title area there is a circular button with an arrow pointing downwards. Click on it.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.13.1 ans, semicolon(;), percent(%). If you type in MATLAB `1+2` the expression is evaluated, assigned to default variable `ans` and then shown onscreen in the command window.

This display can be suppressed by typing a semicolon at the end of the expression as in `1+2;`. The semicolon can be used to separate multiple expressions such as `1+2;20+30;` as well. The evaluation of the last expression (righmost) determines the value that will be assigned to `ans`.

If you want the contents of a line or fraction of to be ignored type the comment symbol of matlab which is the percent sign. Everything on that line starting the percent symbol will be ignored by MATLAB's evaluator (interpreter).

2.13.2 dieresis(...), clc, MATLAB cursor ≫, clear, errors and warnings.

If an expression is long and needs to be split over two (or more lines) one must type a dieresis i.e. three periods one after the other to continue the writing of the expression on the following line. Otherwise an error will be reported.

MATLAB errors are in red while warnings (softer errors) are not in red.

If the screen contains too many expressions, you can clear the screen of the command window by typing `clc`. This also moves the MATLAB cursor ≫ to the top line.

2.13.3 who, whos, clear.

If you want to find out all the variables that you have defined in the WORKSPACE, use `who` or `whos`: the latter provides also size information (the s of `whos` implies it). If you want to wipe out all the variables from the WORKSPACE, use a `clear` or `clear all`; a `clc` will clear the screen not wipe out variables from the WORKSPACE. If you want to wipe out only one variable type something like `clear x` and it will wipe out variable x only.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.13.4 MATLAB anonymous functions. In MATLAB a very easy way to define a very simple function is by creating on-the-fly an "anonymous" function whose presence does not require an M-file (most MATLAB functions are required to reside in an M-file whose name should coincide with the name of the function to be defined). Thus an anonymous function $f(x) = x^2$ is defined by typing `f=@(x) (x*x)`. We note that x, the indeterminate, is the parameter of f in this context whereas $2, 5$ are the arguments for which the function f evaluates to 4 and 25 respectively.

```
>> f= @(x) (x^2)          % x described the function: it is the parameter aka indeterminate
  f= @ (x)(x^2)           % the value of f. In MATLAB terminology f is a function_handle
>> whos
  f         1x1               32  function_handle
>> f(5)
 ans = 25
>> x =10                   % this x is a variable; it has nothing to do with the x of the function_handle
>> whos
  ans       1x1                8  double
  f         1x1               32  function_handle
  x         1x1                8  double
>> f(x)                    % x has a value (100)
ans = 100                 % MATLAB first evaluates x to 10
                          % MATLAB then  uses the 10 to substitute for the parameter x in the
                          % functions handle  f= @ (x)(x^2)  i.e
                          %  f= @(10  (10^2)
                          % The value computed for (10^2) ie. 100 is returned as the value for f(x)
```

The next few pages show examples of using certain features of MATLAB. Some of the reserved words already introduced or to be introduced are included in those examples. The list below shows what page's examples use what keywords.

```
Page 18: help, clc, clear, who, whos, pi, ans,  i, j,

Page 19: int8, ..., uint8, ..., single, double (def) , char , format hex, format long, format short (def)

Page 20: length

Page 21: ls , dir, pwd, type, cd

Page 22: bin2dec, dec2bin, hex2dec, base2dec, intmax, intmin, realmax, realmin
         cos,sin,log,exp,tan, acos,asin,atan, eps, sqrt

Page 23: Operators +,-,*,/,\, @, ^  and an anonymous function
```

(c) Copyright A. Gerbessiotis. All rights reserved.

```
>>  % the percent sign denotes a comment; everything past the percent on that line would be ignored
>> help clc
>> clc           % clear screen
>> clear         % clear MATLAB workspace for the user
>> who           % find variables defined in user's workspace
>> whos          % same as who but with size information (s of whos denotes size)
>> pi            % pi is a MATLAB constant for 3.1416....
ans = 3.1416     % ans is a variable just defined in the workspace
                 % The previous statement asks MATLAB to evaluate expression    pi
                 % Its value is the value of constant pi i.e. 3.1416
                 % That value is returned to a MATLAB local variable named ans for answer
>> clear         % clear workspace
>> whos
>> pi;           % suppress output but ans is still created and value is still available
>> whos          % ans was define with previous statement
>> ans           % and its value can be retrieved
ans = 3.1416
>> i             % i,j are MATLAB constants
>> j
>> i*i           % i and j are the square roots of -1 i.e
 ans = -1        % i * i = j *j = -1
>> j*j           % they are complex numbers
 ans = -1
>> whos          % i,j,pi are constant
>> pi     % vs
>> pi ;   %
           %semicolon ; suppresses output
>> who    %
>> clear
>> pi =10 % OUCH !!!
>> whos   % MATLAB creates a local variable named pi; the constant pi is not available any more
          % local pi hides constant pi. To unhide the latter delete (aka clear) the former
>> clear
>> pi     % Our 3.1416 is back...
>> 5 + 6
ans =  11
>>  5 +
      |
    Error: Expression or statement is incomplete or incorrect
>>  5 + ...  % ... allows you to continue to next line
    6
ans = 11
```

(c) Copyright A. Gerbessiotis. All rights reserved.

```
%  Characters vs Strings: A String is a sequence of 0 or more characters
%  Sequence vs Set  : Order matters (sequence) vs Order does not matter (set)
%  A Set is a collection of elements  : order DOES NOT MATTER
%  Set {1,2,3} is equal to {2,3,1}
%
%  But a sequence is an ordered set of elements: order MATTERS
%  Sequence <1,2,3> is different from <3,1,2}
%  the 1st element of <1,2,3>   is 1
%  but 1st element of <3,1,2>   is 3
%int8, int16, int32, int64     : MATLAB primitive datatypes for 8-,16-,32-, 64-bit integers
%                                (two's complement representation
%                              : int8 represents  -128 ... 0 ... +127
%
%uint8, uint16, uint32, uint64 : MATLAB primitive datatypes for 8-,16-,32-, 64-bit unsigned
%                                 integers
%                              : uint8 represents  0 ... +255
%
% single   : 4B representation of a real number in MATLAB
% double   : 8B representation of a real number in MATLAB  (DEFAULT)
%
% char     : 2B Unicode representation of a character in MATLAB
%
>> a= 1   ;  % 1 is assigned to variable a; the default data-type of numerical variable a
             % is a double i.e. the value of a is 1.0 using 8 bytes
>> whos      % here is a confirmation
  a 1x1 8 double
>> format hex
>> a         % a in hexadecimal is 3ff0000000000000
             % a in binary is  00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000
             % (space between the bytes for readability)
>> format short
>> pi
>> format long
>> pi
>> format
>> pi
>>
>>  a=int8(1)   % a is an 8-bit integer (two's complement) holding integer value 1
                % it is 0x01 in hexadecimal or 00000001 in binary
```

```
>>  a=int32(1)  % a is 32-bit integer
>>  a=uint64(1);   % a is now a 64-bit unsigned integer holding integer value 1
>>  format short   % or just type format
>>  a= '10';       % a is a string of two characters one is character 1 and the other is character 0.
                   % it is 4B long since a character in MATLAB is 2B in UNICODE
                   % The UNICODE value for 0-9 a-z A-z is the same with the ASCII value
>>  a+ 0           % is a trick to retrieve the  UNICODE values of character 1 and 0
    ans  = 49 48   %  Character 1 is 49  and Character 0 is 48 i.e. 0x31 and 0x30 respectively
>>  format hex
>>  a+0            % 49.0 and 48.0 in hexadecimal (real numbers not integers)!
ans = 4048800000000000   4048000000000000
>>  whos  a        % provides information about a : it is a 2-character string/array 1x2
 a  1x2   4 char   % of 4B long

>> a=10;         % semicolon suppresses output
>> a=10          % absence of semicolon forces output;
  a= 10          % confirms 10 is assigned to variable a
>> int8(100)
>> int8(127)
>> int8(128)    % Observe the limits of 8-bit integers
>> intmax('int8');
>> int8(-128)
>> int8(-129)
% Value is the same int8(1) int16(1) uint32(1) single(1) double(1)
% but stored representation is different!
>> format short
>> a='123';
>> length(a) % 3 characters in a; length of string
 ans = 3
>> a=10;
>> length(a)
 ans = 1;
>> a='';    % Empty string of length 0
>> length(a)
>> a= ''''  % String containing ' (1 character)
>> length(a)
```

(c) Copyright A. Gerbessiotis. All rights reserved.

```
>> pi
>> pi ;   % semicolon ; suppresses output
>> who
>> whos   % whos is who but includes size information !
>> format short  %default prints only  4 decimal digits for a real
>> pi
>> format long   % prints 15  decimal digits
>> pi
>> format hex    % hexadecimal
>> pi            % pi (a double ) in hexadecimal notation
>> format short  ;
>> pi, single(pi), double(pi)   % multiple commands separated by ,
>> format long   ;
>> pi, single(pi), double(pi)   % Observe differences!
>> pi
   ans = 3.1416
>> ans
  ans= 3.1416
% ans for answer, stores all evaluated expressions!
>> pi   % evaluates pi to 3.141...etc and displays it in the standard format (default short)
>> whos
>> pi = 10;
>> whos
% You redefined pi to be a local variable ... It's not out pi (3.1415...)
>> help who   % offers assistance to commands
>> ls
>> dir
>> pwd
>> type filename  % print contents of file with  name filename (must exist)
>> cd direct   % go into directory direct (must exist or error)
```

(c) Copyright A. Gerbessiotis. All rights reserved.

```
 >> clc           % clear screen
 >> clear         % clear MATLAB workspace
 >> whos          % no variables listed whos provides size information, who does not
 >> bin2dec('111');  % convert binary number given as string into decimal
    ans = 7         % variable ans (for answer) holds values when MATLAB is being used as
                    % a calculator
 >> whos            %  ans was defined in the workspace. Default numeric data-type is double: 8B
    ans  1x1  8  double

 >> dec2bin(7)      % decimal to bin results is a string i.e. 111 (look how it gets printed)
ans=
111
 >> whos
    ans  1x3  6  char
>> hex2dec('7')    % hexadecimal to decimal conversion
  ans =
      7

>> base2dec('100',8)  % arbitrary base (octal in this example) to decimal conversion
ans =
      64
>> % ISSUES WITH REALS NUMBERS
>> intmax , intmin     % separate multiple statements with comma
>> intmax ; intmin     % separate multiple statements with semicolon   , Compare behavior
>> realmax, realmin
>> cos(pi/2)     % it is supposed to be zero but it is not!
ans = 6.123e-17  % MATLAB
>>  1 == sqrt(1e-17+1)  % is 1 and sqrt(1.00000000000000001) the same ?
>>  1 == sqrt(1e-16+1)  % is 1 and sqrt(1.0000000000000001)  the same ?
>>  1 == sqrt(1e-15+1)  % is 1 and sqrt(1.000000000000001)   the same ?
>>  eps  % for epsilon aka accuracy
  ans = 2.20 ....  e-16
>> eps(0), eps(double(0)), eps(single(0))
>>  (5 + eps(5)) == 5
>>  (5 + eps(sqrt(5))) == 5
```

```
operator :  *  is a symbol denoting an operator for operation multiplication
operation:  multiplication is the operation denoted by *
            multiplication is a binary operation
operator :  *  is a binary operator ; it requires two operands

    a * b
 a: left operand
 *: binary operator for multiplication
 b: right operand

    +: binary operator for addition       Example: 5+6  gives 11
    -: binary operator for subtraction    Example: 5-6  gives -1
    *: binary operator for multiplication Example: 5*6  gives 30
    /: binary operator for division       Example: 5*4  gives 1.2500 (format short, default one)
    new operator
    \: binary operator for back division  Example: 5*4  gives 0.8000  (i.e. 4/5)
    ^: binary operator for exponentiation Example: 2^4  gives 16

    but
    +: unary  operator for positive sign  Example: +5
    -: unary  operator for negative sign  Example: -5
                                          Example: -(-(-5))
                              But also in  MATLAB : ---5
  Functions abs, floor, ceil, round,  sin, cos, exp, log, tan, asin, acos, atan etc

>> f = @(x) (x*x +5.0)   % Define an 'anonymous' function  f(x)= x^2 + 5
>> f(5)                  % f is the function and x is the parameter (aka variable, indeterminate)
  ans = 30               % 5 is the argument that substitutes for x in f(x)
>> z=5                   % z is a variable whose value is 5
>> f(z)                  % z is the argument you first evaluate z to 5 and this value then
                         % substitutes for x in f(x)
  ans = 30
```

(c) Copyright A. Gerbessiotis. All rights reserved.

2.14 MATLAB file types: An M-file. MATLAB recognizes and treats special filenames in a specific way. One such case is a filename ending with the suffix `.m` (`mm` is also a suffix). This is known in MATLAB as an M-file. For example `file.m` is such a file. For this case `file` is the prefix, `m` or `.m` is a suffix. Sometimes `m` is called the file extension and thus a period separates the prefix from the extension. In MATLAB an M-file can contain a sequence or collection of MATLAB functions/commands. One can call such an M-file from within MATLAB by typing in the name of the prefix only, that is by typing `file` without the `.m`. Then the sequence of MATLAB functions in the M-file will be executed in the command window and when this completes control will pass back to the MATLAB command window and the prompt will appear.

Reminder. When calling an M-file such as `myfile.m` , do not type `.m`, just `myfile`

2.15 MATLAB special M-files: startup.m and finish.m. There are two special M-files in MATLAB. The M-file `startup.m` is executed as soon as MATLAB loads up (and the command window appears), and `finish.m` that is invoked when the user is about to exit the command window by say, typing the function `quit`. Note that those two files will be executed as long as they reside in the directory from which MATLAB will be invoked or from which MATLAB will be terminated. One way to see whether the execution of `startup.m` or `finish.m` has occurred or is to occur is to type in the command window the operating system-related command `dir` to get the directory listing of the current directory within the command window and confirm the existence of one or the other files there. The typing of the comand `type startup.m` or `type finish.m` will show the contents (i.e. MATLAB commands) of those two files.

2.16 dir,ls, type, pwd, cd

The function `dir` described in 2.15 is a MSDOS/Windows command prompt relic equivalent to the UNIX command `ls`; both can be invoked from within the command window and both list the contents of the current working directory of MATLAB (the word folder is equivalent and a synonym to the word directory and will be used interchangeably). What this current directory is can be revealed by typing `pwd` (which is the initials of `print working directory`). You can view the contents of a file say `finish.m` by typing the MSDOS/Windows-style `type finish.m` command with a space separating the two. If you want to change directory, the `cd` command can be used appropriately (UNIX or Windows syntax as required). If you want to navigate to the top of your directory space type `cd` $\sim$.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.17 Editing a file from within the Command Window or through Home/New. Note that in Matlab R2013a or in other variants the current working directory appears on a long line over the Command Window window (and its title/label). In fact on the left of the command window there usually is a vertical window calls `Current Folder` that lists the contents of the current working directory/folder. A user can edit (i.e. create, save) a MATLAB M-file by typing the word `edit` in the Command Window. If no argument follows the user will be given the opportunity to save the work in a file (M-file) later on through the opened window related to the Editor that has just been invoked. Of course one can type `edit myfile.m` and if `myfile.m` exists it will be opened, or a new file will be opened for writing. One can edit a file from outside the command window. MATLAB's toolstrip attached to the `HOME` tab (top of the window) has a button (second from left, with a huge plus sign) that is called `New`. One can create a New M-file by clicking on button `New` attached to tab `HOME` of the MATLAB window and then choose the first option or `Script` if it happens not to be the first option. An M-file is a script file that contains MATLAB commands. One can edit a file by clicking on `New` (it is the button with the huge plus sign second from left in the `Toolstrip`).

2.18 Diary keeping with `diary, diary on, diary off`. If you want to keep track of the activity in the command prompt (both the MATLAB commands you type, but also MATLAB's output), you can use the command `diary`. For example `diary('myd.txt')` will use file `myd.txt` to store information. When you type in MATLAB this command it also initiates a `diary on`. All the commands you type, and MATLAB's output are to be saved into the designated file. However you need to type a `diary off` for this to be committed/save into the file. You can then view the contents of it with `type myd.txt`. You can then issue `diary on` to go on saving additional information, as needed (also requiring a `diary off` eventually). The new material is added to (appended) the existing contents.

(c) Copyright A. Gerbessiotis. All rights reserved.

MATLAB M-file and diary example

```
%   let a file is name  myfile.m
%  Filename is        myfile.m
%  Prefix of it is    myfile
%  Suffix of it is    .m  or sometimes m (aka extension)
%  Extension of it is m

%  M-file is a file with suffix .m i.e. extension m
%  Special M-files are  startup.m and finish.p

>> ls ~/startup.m ~/finish.m
% If they exist they serve a special purpose
% All the MATLAB command in startup.m are executed before the command windows returns to the user the
% promptt >>
%
% When the user exits the command window, all the MATLAB commands in finish.m are executed
% before the windows closes (with a slight delay)
% The instructor startup.m file contains
   disp('HELLO. This is startup.m');
   diary(['CS101_alexg_' , datestr(now,'yyyymmdd_HHMMSS') ]);
   pause(2);
% The middle line is not obvious: it creates a diary file that keeps track of MATLAB commands
% typed in the command window
% If you don't have one type in a MATLAB command window
% diary('fname.txt');   create a diary with name  fname,txt (also does diary on)
% diary on              start it or restart it
%  diary off            stop and commit changes

>> edit ~/startup.m  % to create or edit one
>> edit ~/finish.m
% You can execute M-file myfile.m from within a command window by typing its prefix
>> myfile
```

(c) Copyright A. Gerbessiotis. All rights reserved.

2.19 MATLAB MAT (.mat) data files.

Whereas an M-file has a suffix `.m`, a MATLAB MAT file has a suffix of `.mat` in its filename. Thus `file.m` denotes an M-file but `file.mat` would denote a MATLAB MAT file. As we have already mentioned an M-file contains a sequence of MATLAB commands that operate on MATLAB variables and more. In MAT files MATLAB stores NOT COMMANDS but plain data such as variables (their names and their values) in binary format.

Think about storing in an MAT files the contents of the Workspace only but in an M-file the contents of the command history of a MATLAB environment. The data stored into an M-file are in binary format: it is very difficult for a human to read them. Whereas one can edit an M-file with an editor, this cannot happen with a MAT file.

Thus if one uses MATLAB as a calculator and has some intermediate results but wants to take a break and continue the calculations and did not maintain an M-file, the that person can save the values of all data generated into an MAT file, exit MATLAB, and then start MATLAB again and recover the variables and their values but not the commands that generated the variables and their values. The latter could be retrieved either through the use of a diary or an M-file. Because of this it is always a good idea to use an M-file for your calculations rather than working directly in a command window.

2.20 MATLAB load, save.

All variables defined in a MATLAB command window can be saved into a MATLAB file say `mydata.mat` by typing `save mydata` (the suffix will be created automatically). One can then issue a `clear` command in the command window to clear the variables (along with their values). The data however are still available in `mydata.mat`. Subsequently one can load them by using `load mydata` (the suffix is added as needed). A user can selectively load a specific variable (and its value) by typing instead `load mydata y`, or `load mydata x y`, where the list of variables to be loaded follows the name of the MAT file. This applies also to the `save` command. A `save mydata x y` could have also been used for the same effect as `save mydata`. The `append` option of save can be used to add one more variable into `mydata` without modifying `x,y`. This can be effected by typing `save -append mydata z`.

(c) Copyright A. Gerbessiotis. All rights reserved.

MATLAB MAT `.mat` file example

```
>> x=1;
>> y=2;
  x         1x1                        8  double array
  y         1x1                        8  double array
>> save mydata
>> dir
 mydata.mat
>> clear
>> whos
>>
>> load mydata
>> whos
  x         1x1                        8  double array
  y         1x1                        8  double array
>> z=4;
>> save -append mydata z
>> clear
>> whos
>>load mydata
>>whos
  x         1x1                        8  double array
  y         1x1                        8  double array
  z         1x1                        8  double array
```

```
 %  A .mat file  (mat from MATRIX) contains variables and their respective values copied from workspace
 %  save fname.mat   % saves all existing variables from the workspace into file fname.mat
 %                       % variables still exist in workspace and if modified, changes
 %                       % WILL NOT affect values at fname.mat
 %                       % if fname.mat exists all prior contents WILL BE ERASED first.
 %      save fname.mat a b c  % stores only a,b,c but not other variables
 %                       % all contents of fname.mat will be erased before the save
 %      load fname.mat   % loads all variables from fname.mat
 %      load fname.mat a % loads  only a from fname.mat
 %
 %       save fname.mat  d  % Erases all previous contents of fname.mat and stores d only
 %
 %       To maintain old variables + values!
 %   To maintain old variables + values!
 %
 %      save fname.mat -append d  % Existing variables are maintained
```

(c) Copyright A. Gerbessiotis. All rights reserved.

```
>> 1+2     %operator is +          |   >> +5;        % + is unary operator for sign
ans =  3   %operation is addition |   >> ++5;       % silly but correct
>> 3-4     %operator is -          |   >> -5 ;       % - is unary operator for sign/negate
ans = -1                           |   >> --5
>> 4*5     %operator is *          |    ans = 5
ans = 20                           |   >> -------5   % sillier
>> a=10;b=20;                      |    ans = -5
>> a/b     % division              |   >> 5 ----- 5  % playing with FIVE or with FIRE
ans = 0.5000                       |   ans = 0       % the first - is binary
>> a\b     % back division         |                 % the remaining 4 are unary
ans = 2                            |                 % this is 5 - (----5) = 5-5 =0
>> 1-2-3   % left to right         |   >> 2^4
ans = -4                           |   ans = 16
>> (1-2)-3 % i.e. this way         |   >> 2^2^2^2    % left to right is ((2^2)^2)^2
ans = -4                           |   ans= 256      % not .... 2^(2^(2^2)) = 65536
>> 1-(2-3) % parentheses can      |
ans = 2    % change eval order     |
```

2.21 Binary Operators $+, -, *, /, \backslash, \hat{}$**.** An operator such as $+, -, *, /, \backslash$ is a **binary operator** that accepts two arguments (sometimes called operands), and combines them according to the rules implied by the indicated operator by performing the corresponding operation. Thus operator + (i.e. plus) adds its two arguments by realizing the operation known as addition. The operator $-$ indicates subtraction and the operator $\backslash$ backdivision, that is $x\backslash y$ is y/x. Operator $\hat{}$ used as $a\hat{}b$ raises a to the b-th power.

2.22 Unary operators $+, -,$. The unary operator + indicates or assigns a postive sign to a scalar(number) and the $-$ a negative sign, i.e. turns a number into negative. Both of them have higher precedence than the corresponding binary operators $+, -$. So these operators are context specific and can behave as binary and unary operators.

2.23 Precedence of $+, -$ **and** $/, *, \hat{}$**, Parentheses and Left-to-right order of evaluation.** Parentheses can be used to **force** a given order in evaluating an arithmetic expression. In the absence of parentheses, if multiple but same type operators are defined as in `4 * 3 *2 *1` the expression is evaluated left-to-right i.e. first `4*3` is computed, then the result is multiplied by 2 and so on. Thus for `1-2-3`, we first perform `1-2` that gives `-1` as the answer and then `-1-3` is performed to give `-4` as the final answer. Operators + and $-$ have the same precedence and so do $*$ and $/, \backslash$. Unary $+, -$ have higher precedence than the 5 binary operators, and $\hat{}$ exponentiation tops them all. Same precedence operations are evaluated left to right. Thus $2 * 10/5$ will first evaluate $2 * 10$ and then the result will be divided by 5. **Precedence of operators including scalar operators is further discussed at the end of this Subject.**

(c) Copyright A. Gerbessiotis. All rights reserved.

2.24.1 Associativity. Operations such as addition and multiplication are called associative in the sense that the corresponding operators satisfy $a*(b*c) = (a*b)*c$ and $a+(b+c) = (a+b)+c$ for all a, b, c. Subtraction and division are not associative. For example, $(1-2)-3 = -4 \neq 1-(2-3) = 2$.

2.24.2 Commutativity. Operators $*$ and $+$ are commutative, that is, $a+b = b+a$ and $a*b = b*a$, for all a, b.
2.24.3 Operator overloading. The operators $+, -$ can be used as binary or unary operators to signify addition/subtraction or the sign of a number. However they can also indicate matrix/array operations as well. In fact numbers in MATLAB are abstracted as matrices. Therefore operators such as $+, -$ can be overloaded explicitly or implicitly (we think they are used for example to add numbers but they also add matrices).

2.24.4 Example. On the previous page we use MATLAB to perform some very simple scalar operations involving + and − (i.e. addition and subtraction) and $*, /$ (i.e. multiplication and division). When we type in `1+2` in the MATLAB prompt and press ENTER, the result is stored in a MATLAB scalar called `ans` (for answer). The value calculated is then printed. Because the answer in all cases involves an integer the answer is printed as an integer. We also use MATLAB to further study its division capabilities. The symbol / when used in the context of `a/b` calculates a non-integer result. The answer is then printed using the default number of decimal digits, in this case 4 (format short) and the result is rounded appropriately. When the alternative division operator is used for back-division $a\backslash b$, the result 2 is printed as an integer. We are reminded that $a\backslash b$ is equal to b/a. Then the expression `1-2-3` is evaluated. This is an ambiguous expression because between the two subtractions one needs to figure out which one to perform first. If there is an ambiguity MATLAB decides based on it own rules (left-to-right) to resolve them. If you want another order of evaluation, you must tell MATLAB so by using parentheses! **WE STRONGLY RECOMMEND** that you use parentheses in your MATLAB expressions.

2.25 Be careful with multiple operators in an expression. Thus `5/5/5` and **`2*3/4`** might pose a danger if you forget the rules of left-to-right evaluation.

2.26 Exponentiation. The operation `2^2^2^2` gives 256.

2.27 Playing with fiVe. Expressions $--5$ and $---5----5$ are correct MATLAB expressions and evaluated to 5 and 0 respectively.

2.28.1 When is a zero a zero? Note also that when we apply a variety of functions the result is not the real number we have in mind but an approximation of it as well. For example sin(pi) is the mathematical function of applying the `sine` function to MATLAB's representation (or approximation of) π. Though normally $\sin(\pi)$ is 0, the result given by MATLAB is not zero although it might be very close to it. In my case sin(pi) returns $1.2246e-16$ which is how MATLAB writes 1.2246×10^{-16}. It is a very small, close-to-zero number, but not zero. So beware of real numbers in MATLAB, and how they compare (or not) to zero.

2.28.2 Exponential notation. Note that MATLAB has problems raising 10 to powers that need to appear as a superscript to 10. Thus writing 10^{-16} in MATLAB poses a variety of output problems. An easy way is to write it as $1.e-16$ instead, or $1.0e-16$, or $1.E-16$, or $1.0E-16$, where a lower-case `e` or an upper-case `E` might be used.

2.28.3 `true` and `false`. The result of a logical (sometimes called boolean operator) is a `true` or `false`. MATLAB stores the result in a `logical` data-type that only uses one byte. Thus `1` the number may be stored in a `double` or 8-bytes but `1` for `true` only uses one-byte. Thus `1==1` will give `ans=1` and this would use one byte (do a `whos`).

(c) Copyright A. Gerbessiotis. All rights reserved.

```
>> clear                                  | >> 1 ~= 2
>> true                                   | ans = 1
ans = 1                                   | >> 1 < 2 < 3  % left-to-right (1<2)<3 is
>> false                                  | ans = 1                          1  <3 is 1
ans = 0                                   | >> 5 == 5 == 5  % playing with FIVE
>> whos                                   | ans = 0
 ans  1x1  1  logical                     | >> 5 ~= 5 ~= 5  % more FIVE or FIrE!
>> 1 < 2                                  | ans = 1
ans = 1                                   | >> (1<2>3)      % parentheses?
>> 1 <= 2                                 | ans = 0
ans = 1                                   | >> (1==1) + (1==1) ;  % semicolon at the end
                                          | ans = 2
>> 1 == 2                                 | >>  1==1 + 1== 1  % mix+match arithmetic and
ans =0                                    | ans = 0           % relational operators! BE ALERT!!!
```

2.29 MATLAB relational operators: $<, >, <=, >=, \sim=, ==$. These include the familiar `less-than`, `greater-than`, `less-than-or-equal-to` and `greater-than-or-equal-to` relational operators. They also include the unique to MATLAB form of the `not-equal-to` operator, as well as the more prevalent form of the `equal-to` operator. In most cases the operator is denoted by two consecutive symbols for example the tilde followed by the equal sign for $\sim=$ `not-equal-to`, or two consecutive equal signs for `equal-to`.

2.30 Binary relational operators: left-to-order default order of evaluation. All operators are **binary** i.e. they accept two arguments (one the left and right of the operator). All operators have the same precedence and if several of them appear in an expression they are evaluated left to right absence of properly placed parentheses. Thus you should use parentheses to describe the intended order of evaluation!

2.31 `true, false,` and `logical` data-type. The result of the comparison between the arguments is true or false. For MATLAB, true is the numeric value `1` and false is `0`. In MATLAB there are variables `true` and `false` whose values are respectively `1` for true and `0` for false. Note that even if the value of such a variable is a single bit, MATLAB uses a full byte to represent it. The data-type of `true, false` and the results of a relational operator is a `logical` data-type. Thus relational operator results are stored into a `logical` data-type not `double`.

2.32 Example. Therefore, the result $1 < 2$ is true, i.e. `1` whereas $1 == 2$ is false, i.e. `0`. If we want to test for equality we need to write two $=$ one immediately following the other, or just test for inequality and flip the answer. $1 \sim= 1$ is naturally false, i.e. `0`.

2.33 Example. The sample code on the previous page depicts some irregular examples that take advantage of the fact that the relational operators return a `0,1` as an answer. Given that relational operators are binary operators, $1 < 2 < 3$ is correctly computed as `1` using the left-to-right evaluation in the absence of parenthesization. If there is one good reason to avoid practices such as those depicted in that sample code, the next examples will convince you.

Note that we have used invariably three different ways to denote the same thing.

- We write true or false to say that the result of a comparison is true of false. This is written in plain-text without any font insertion and it's a statement.
- We write `true` and `false` in a different font to indicate the MATLAB variables of the same name that holds a true or false value respectively. These variables, by default take the values of `1` and `0` respectively, corresponding to a true and false value.
- We write `1` and `0` to indicate the values of MATLAB variables `true` and `false` respectively. These variables represent in MATLAB true and false.

One way to disambiguate this usage , is to consider the well known π symbol (by the way MATLAB prefers to using `pi` instead). This variable takes, for most human beings the value of $\pi = 3.14$. In fact we call it a **constant** because its value is fixed through its definition that states that π is the ratio of any circle's circumference to its diameter. The value of 3.14 set to π is not the true value of the ratio but an approximation of it to two decimal digits. So we humans picked the symbol π to denote that ratio, and also picked the value of 3.14 (to two decimal digits) as the value of it. Similarly, MATLAB picked the variable names `true` and `false` (rather than TRUE or True etc) to denote the true and false values and to ease evaluation with them assigned to them the numeric values of 1 and 0 respectively. However the MATLAB `true, false, pi` can still be hidden behind variables of the same name defined by a careless user/progammer. **Therefore, be very very careful in dealing with them and if doubt type `whos`.**

```
>> 1 == 2 ;    % A very dangerous sequence of evaluations  1==2 return false and result stored in ans
>> ans == 0    % thus ans == 0 is equivalent to asking 0 == 0 which is true(1) and result stored in ans
  ans =1
>> ans         % Thus when ans is printed value is 1
  ans = 1
>> whos
  Name      Size                    Bytes  Class
  ans       1x1                         1  logical array
Grand total is 1 element using 1 bytes
```

```
>> 1 == 1
  ans = 1
>> whos
  ans       1x1                     1  logical
>> 1*1
 ans =1
>> whos
  ans       1x1                     8  double
>> true
ans = 1
>> whos
  ans       1x1                     1  logical
>> true  + false
ans = 1
>> whos
  ans       1x1                     8  double
```

2.34 MATLAB is a weakly-typed language. Notwithstanding the previous discussion if a relational operator is succeeded by an arithmetic evaluation (such as $1 * 1$) this will be detected by MATLAB and will result to storing the new result into a arithmetic data-type; the default such data-type is a `double`. Note that `ans`'s data-type changes all the time in the example above. In fact `true+false` is an arithmetic expression !

2.35 `true, false` overload! A note of danger. One can easily mistype a `true == 0` (i.e. an application of a relational operator) to type instead a `true = 0` (i.e. an assignment of a zero value to special variable `true`). MATLAB will create a local Workspace variable `true` with that value and from that point on this `true` will be used for `true`. **BE VERY CAREFUL** with your equals!

2.36 Playing with FIvE and FIre. What is the result of the evaluation **5 == 5 == 5**? Or 5 <= 5 <= 5 == 5? This serves as a reminder to the very clear message: PARENTHESIZE!

2.37 Things to be careful with. Note what happens with $0 == 0/1$. When one mixes arithmetic expressions and relational operators, the arithmetic operators apply first. And a $0/0 == 1$ or $0/0 == 0$ will both give a false. Note that `Inf == Inf` will give true (`Inf` is infinity) but `NaN == NaN` will give false. Same for the variables `nan, inf` whose values are `NaN, Inf` respectively.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.38 Logical Operators, Logical Operations and their Symbols (regular and alternative).
Operation Names: negation (NOT), disjunction (OR), conjunction (AND), exclusive-or (XOR)
Operator Names: LOGICAL-NOT, LOGICAL-OR or LOGICAL-INCLUSIVE-OR, LOGICAL-AND, LOGICAL-XOR or LOGICAL-EXCLUSIVE-OR.
Operator Symbols: ∼, |, &, `xor`.
Operator Symbols (alternative): `not, or, and, xor`.
Usage Example: $or(a, b)$, $not(b)$ for the equivalent $a \mid b$ and $\sim b$.

2.39 Logical operators &, |, `xor`, ∼: conjunction, disjunction, exclusive-or and negation. The definitions for the
(a) LOGICAL-AND (&),
(b) LOGICAL-OR or LOGICAL-INCLUSIVE-OR (|),
(c) LOGICAL-EXCLUSIVE-OR or LOGICAL-XOR (`xor`), and
(d) the LOGICAL-NOT (∼)
operators are given in Table 1, Table 2, Table 3, Table 4 respectively.

2.40 Operands for logical operators. The arguments of the logical operators can be

- logical variables such as `true` or `false`,
- variables that evaluate to `true` or `false`, or
- arbitrary arithmetic expressions. If the expression evaluates to `0` it is `false`. If it evaluates to any non-zero value it is `true`.

Thus any non-zero value is `true` and a zero value is always `false`. Note that `false` is lower-case false; `FALSE` is something totally different.

2.41 LOGICAL-AND. A LOGICAL-AND on two arguments evaluates true if both its arguments are `true`; otherwise it is false. We write $a\&b$ to denote the LOGICAL-AND of a and b. (Think of $a \cdot b$.)

2.42 LOGICAL-OR. A LOGICAL-OR on two arguments evaluates false if both its arguments are `false`; otherwise it is true. We write $a \mid b$ to denote the LOGICAL-OR of a and b. (Think of $a + b$.)

2.43 LOGICAL-XOR. A LOGICAL-XOR on two arguments evaluates false if both its arguments are `false` or both are `true`; otherwise it is true (i.e. when one argument is `true` and one is `false`). It is for this reason that we call it an exclusive OR; it does not allow both arguments to be true/false at the same time. We write `xor`(a, b) to denote the LOGICAL-XOR of a and b.

2.44 LOGICAL-NOT. A LOGICAL-NOT on an argument is `true` if the argument is `false`, and `false` if the argument is `true`. A LOGICAL-NOT is a unary operator. We write $\sim a$ to denote the LOGICAL-NOT of a. (Think of $1 - a$.)

Caution 1. If a careless user defines a variable `FALSE` and sets its value to 1 then it is conceivable that `true` & `FALSE` gives as an answer `1`, i.e. `true`. Note that `false` and `FALSE` are different.

Caution 2. Note that `true`&`true` has result which is 1 (true) and the result is stored in `ans` whose data-type is `logical`. Yet `true` + `false` gives the same result 1, which is a scalar value not a logical value and thus it is stored into a double.

Caution 3: Evaluation sequence. For the binary operators (LOGICAL-AND, LOGICAL-OR, LOGICAL-XOR) and the unary operator LOGICAL-NOT, first all of the arguments are evaluated and then the corresponding operator is applied. Because of this the following **shortcut** in the evaluation of both arguments can be taken instead.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.45 Logical shortcut operators && **and** ||**.** The definitions for the
(e) LOGICAL-AND-WITH-SHORTCUT-EVALUATION (&&), and
(f) LOGICAL-OR-WITH-SHORTCUT-EVALUATION (||)
are given by tables identical to & and | respectively. The only difference is that the SHORTCUT operators &&, || may not evaluate one of their arguments.

This means that if we are to evaluate a && b the evaluation of b might not take place if as soon as a is evaluated it evaluates to `false` and thus there is no reason to evaluate b, since the expression must be `false`.

Similarly for a || b if a evaluates to `true` the result is true and no evaluation of b is necessary.

If you expect both arguments to be evaluated use the standard &, | operators; if you want to lazy-evaluate the expression, use (WITH CAUTION) the SHORTCUT versions of the operators.

2.46 LOGICAL-AND-WITH-SHORTCUT. A LOGICAL-AND-WITH-SHORTCUT on two arguments evaluates immediately false if the left argument is false; otherwise the second argument (and then and only then) is evaluated and its logical value determines the result (true if it is `true` and `false` otherwise).

2.47 LOGICAL-OR-WITH-SHORTCUT. A LOGICAL-OR-WITH-SHORTCUT on two arguments evaluates immediately true if the left argument is true; otherwise the second argument is (then and only then) evaluated and its value determines the result.

2.48 Caution. For LOGICAL-AND-WITH-SHORTCUT, LOGICAL-OR-WITH-SHORTCUT the left argument is evaluated first; whether the right argument will be evaluated or not will depend on the first evaluation and whether the first evaluation suffices to obtain the answer! Thus 0&& `toast` will not complain about the undefined variable `toast` whereas 0& `toast` will loudly complain about it.

(c) Copyright A. Gerbessiotis. All rights reserved.

2.49. Tables for MATLAB logical operators. In the tables that follow, an `1` is taken to mean any non-zero value; and `0` is anything that evaluates to zero. The answer `true` in MATLAB is given by an output that is `1` and `false` an output that is `0`.

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	0

Table 1: Operator &

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Table 2: Operator |

a	b	`xor`(a,b)
0	0	0
0	1	1
1	0	1
1	1	0

Table 3: Operator `xor`

a	$\sim$ a
0	1
1	0

Table 4: Operator $\sim$

2.50 Examples.

Thus if one gives in MALTAB `pi&pi`, then the output will be `1` for true. This is because both `pi` evaluate to a non-zero i.e. `true` value. And of course if one gives `true | false`, the output will also be `1` since `true` evaluates to `1` and `false` to `0`.

```
>> pi & pi
ans = 1
>> whos
 ans  1x1 1 logical
>> pi + pi & pi
 ans = 1
>> whos
 ans  1x1 1 logical
>> pi + (pi &pi)
 ans =  4.1416
>> whos
 ans  1x1 8 double
```

```
>> true & false
ans = 0
>> true | false
ans = 1
>> ~ true
ans = 0
>> xor(true,true)
ans = false
>> 0 & 0
ans = 0
>> 1 | 1
ans = 1
>> 1 & ~0
ans = 1
>> xor(0,~0)
ans = 1
```

```
>> 5 & 6  % 5 is true, and so is 6
ans = 1
>> 0 & toast
Error using toast
Too many output arguments.
>> cute=0;
>> program = 10;
>> cute & program
ans = 0
>> 1 | (1/0)
ans = 1
>> 1 & (1/0)  % infinity is not zero
ans = 1
>> NaN & Nan   % We call NaN  Not A Number
Error using &  % for a reason. Here it is!
NaN's cannot be converted to logical
```

2.51 More Examples. First we use the `true` and `false` variables with the logical operators. Then we use instead the numerical values of `0,1`. Note that expressions can be complicated. For example $1\& \sim 0)$ evaluates to 1. This is because 0 evaluates first to 1. Variable `toast` is undefined and has no value. Yet its presence as the second argument of a properly formed shortcup operator cause no hiccups! Try also 0&&`toast` and 0&`toast`. Finally, relational operators have higher precedence than logical operators: if you don't parenthesize you play with fire (or five below).

```
>> 1 | toast
Error using toast
Too many output arguments.
>> 1 || toast
ans = 1
>> 1 || NaN
ans = 1
>> 1 |  NaN
Error using  |
NaN's cannot be converted to logical
```

```
>> 25 - (25 * ~0)
ans = 1
>> 100 & 100 > 10 % first the relational
ans = 1
>> 5 & 5 >= 5   % playing with FIVE
ans = 1         % is confusing
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Other functions available in MATLAB

A set of functions that are available in MATLAB for the benefit of the user includes the following. A partial list is given that includes some common mathematical functions, rounding functions and string related functions beyond those discussed previously

```
Mathematical functions
  abs(x)                                : absolute value of x ,  [abs(-5)=5, abs(5)=5]
  sin(x),  cos(x),  tan(x)              : Trigonometric functions
 asin(x), acos(x), atan(x), atan2(x)    : Inverse trigonometric functions
  exp(x), log(x)                        : Exponential and Logarithmic functions
  sqrt(x)                               : Square root of x
  mod(x,y)                              : Integer remainder of division of x by y  [mod(5,3)=2]
  floor(x)                              : largest integer <= x  [floor(3.1)=3, floor(-3.1)=-4]
  ceil(x)                               : smallest integer >=x  [ceil(3.1)=4,ceil(-3.1)=-3]
  fix(x)                                : Closest (towards zero) integer to x.
                                          [fix(-3.5)=-3, fix(-3.1)=-3, fix(4.9)=4]
  round(x)                              : rounds x to nearest integer [round(-3.4)=-3,round(-3.5)=-4,
                                          round(3.5)=4,round(3.4)=3]


  bin2dec(x),dec2bin(x)                 : Converts binary into decimal and other way around
  hex2dec(x),base2dec(x,base)           : Converts hexadecimal into decimal
  base2dec(x,base)                      : Converts arbitrary base <= 32 into decimal
  int2str(x)                            : Converts x into an integer character string
  num2str(x)                            : Converts x into a character string
  str2num(x)                            : Converts character string s into a numeric array
  char(x)                               : Creates char array (string) of array of (say ASCII) codes.
  double(x)                             : Converts x into double (if already double, no effect)
                                          Likewise for other data type int8(x) .... uint8(x)
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Name	Type	Description
+	Scalar?	Addition
−	Scalar?	Subtraction
*	Scalar?	Multiplication
/	Scalar?	Division (Forward)
\	Scalar?	Division (Backward)
^	Scalar?	Exponentiation
<	Relational	Less-than
>	Relational	Greater-than
<=	Relational	Less-than-or-equal-to
>=	Relational	Greater-than-or-equal-to
∼=	Relational	Not-equal-to
==	Relational	Equal-to
&	Logical	Logical-AND
|	Logical	Logical-OR(inclusive)
xor	Logical	Logical-OR(exclusive)
∼	Logical	Logical-NOT
&&	Logical	Logical-AND-shortcut
||	Logical	Logical-OR-shortcut
'	String	String-begin
'	String	String-end
length	String	String-length
whos	Command	List of user variables
clear	Command	Clear user variables
char	Internal	Internal data type (character)
int	Internal	Internal data type (integer two's complement) as in int8,int16,int32,int64
uint	Internal	Internal data type (unsigned integer) as in uint8,uint16,uint32,uint64
logical	Internal	Internal data type (8-bit logical true/false)
single	Internal	Internal data type (32-bit real)
double	Internal	Internal data type (64-bit real)

Figure 3: MATLAB Operators or Functions

Note. Scalar? means the result of the application generates a (or more) numbers

Name	Type	Description
`abs(x)`	Math.Funct.	absolute value
`acos(x)`	Math.Funct.	Inverse of cosine
`asin(x)`	Math.Funct.	Inverse of sine
`atan(x)`	Math.Funct.	Inverse of tangent
`atan2(x)`	Math.Funct.	Inverse of tangent(angle restriction)
`ceil(x)`	Math.Funct.	ceiling
`cos(x)`	Math.Funct.	Cosine
`exp(x)`	Math.Funct.	Exponential e**x
`fix(x)`	Math.Funct.	closest to x (towards zero)
`floor(x)`	Math.Funct.	floor
`log(x)`	Math.Funct.	Logarithmic base e
`mod(x,y)`	Math.Funct.	modulo(x,y)=r, x=qy+r 0¡=r¡y
`round(x)`	Math.Funct.	rounds x
`sin(x)`	Math.Funct.	Sine
`sqrtx)`	Math.Funct.	Square root
`tan(x)`	Math.Funct.	Tangent
`int2str(x)`	String	Converts x into an integer character string
`num2str(x)`	String	Converts x into a character string
`str2num(x)`	String	Converts string x into a numeric array
[value,index] = max(x)	Vector	Find value and location of maximum in vector x
[value,index] = min(x)	Vector	Find value and location of minimum in vector x
`sort(x)`	Vector	Sort vector x (smallest to largest)
`sort(x)`	Matrix	Sort matrix x (row major smallest to largest)
`sort(x(:))`	Matrix	Sort matrix x (column major smallest to largest)

Figure 4: MATLAB Operators or Functions (continued)

(c) Copyright A. Gerbessiotis. All rights reserved.

Precedence of operations

An expression in MATLAB is evaluated by following the rules below in the order specified. The rules are listed in order of precedence, higher-order rules are listed first and thus resolved first; if multiple instances of the same or equivalent operator appear, the resolution occurs in a left-to-right fashion with a left-most operator resolved (i.e. evaluated) first. **CAUTION. In (high) school you learned that divisions were to be performed before multiplications! This does not apply to MATLAB. They are at the same level, i.e. they have the same precedence.**

1. In an expression all parenthesized subexpressions are resolved (i.e. evaluated) first starting with the innermost and proceeding to the outermost of nested parenthesized expressions.
2. After parenthesized expressions are resolved, then expressions involving exponentiation (^) or transposition ($'$) are resolved left-to-right.
3. Then expressions with unary plus(+), minus(-) or LOGICAL-NOT $\sim$ are resolved left-to-right.
4. Then expressions with $*, /, \backslash, .\backslash, ./, .*$ are resolved left-to-right.
5. Then expressions with $+, -$ are resolved left-to-right.
6. Then the colon operator (:) is resolved.
7. Then expressions involving relational operators such as $<, >, <=, >=, \sim=, ==$ are resolved left-to-right.
8. Then expressions involving the LOGICAL-AND & are resolved left-to-right.
9. Then expressions involving the LOGICAL-OR |, LOGICAL-XOR `xor`
10. Then expressions involving the short-circuit LOGICAL-AND && are resolved left-to-right.
11. Then expressions involving the short-circuit LOGICAL-OR || are resolved left-to-right.

CAUTION. We called several of the operators of this page scalar operators. The truth that we have hidden (intentionally) is that these operators can also apply to vectors and other matrices. This is what we call in computer-science `operator overloading`. We will discuss more about using these operators on non-scalar arguments later in this course.

Examples.

The expression 2^3^4 is 4096 since (2^3)^4 = 8^4 = 4096. The expression 1&1 > 10 is 0 since 1&1 > 10 = 1&(1 > 10) = 1&0 = 0, with 0 signifying the value for a `false`. The result 2^(3^4) is a very long number. The only way for MATLAB to represent that result is by default as a real number and a result `2.4179e+24` might be output. Without the explicit parenthesization, a 4096 would have resulted as an answer.