Introduction to MATLAB operations

This is Chapter 1 of the textbook plus Chapter 3 (page 99)

plus chapter 6.1, 6.4, 8.6.1-8.6.3

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 2013 ; distribution outside this group of students is NOT allowed.*

```
>> 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    |
```

4.1 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.

4.2 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.

4.3 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.

4.4 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, $(1-2)-3 = -4 \neq 1-(2-3) = 2$.

4.5 Commutativity. Operators $*$ and $+$ are commutative, that is, $a+b = b+a$ and $a*b = b*a$, for all a, b.

4.6 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 a 1×1 matrix. Thus $1+2$ for MATLAB denotes the addition of two 1×1 matrices. Therefore operators such as $+, -$ can be overloaded explicitly or implicitly (we think they are used for example to add numbers but they add matrices). Similarly $*, /, \backslash$ can be applied to general matrices (more than one rows/columns) to denote matrix multiplication and division as explained. However if an operation similar to $+, -$ is required for two matrices (we call such operations array operation) the corresponding array operator must be used and is made available separately in MATLAB. Thus $.*, ./, .\backslash$ are the array operators for array multiplication, division and backdivision (See Subject 2).

4.7 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.

4.8 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.

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

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

(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!!!
```

4.11 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`.

4.12 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!

4.13 `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`.

4.14 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`.

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

4.15 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
```

(c) Copyright A. Gerbessiotis.

```
>> 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
```

4.16 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 !

4.17 `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!

4.18 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!

4.19 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.

4.20 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 $\sim b$.

4.21 Logical operators $\&, |,$ `xor`$, \sim$**: 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.

4.22 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.

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

4.23 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 $a \cdot b$.)

4.24 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 $a + b$.)

4.25 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.

4.26 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.

4.27 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.

4.28 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).

4.29 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.

4.30 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.

4.31. 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 $\sim$

4.32 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
```

4.33 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.

In MATLAB we are going to use quite frequently strings.

4.34 String. A **string** is a concatenation of zero or more characters. The string with no characters will be called the NULL string or the empty string and for this discussion we shall use the symbol ϵ to represent it.

4.35 Length of a string. The length of a string is the number of characters in it or zero if it is ϵ.

4.36 MATLAB strings. If we want to define a string in MATLAB we use the single-quote character to surround it. Thus `'abcdefgh'` is a string of eight characters, the first eight letters of the alphabet in lower-case. And `''` is the empty string; note that the second quote is immediately after the first with no other space in between. We usually assign a string to a variable to hold it in the memory of a computer and to retrieve it easily by the name of the variable rather than the (quite possibly lengthy) value of it.

Thus `a= 'abcdefgh'` assigns string `'abcdefgh'` to variable `a` in MATLAB. It is easier to say or write variable **a** rather than `'abcdefgh'`. MATLAB has a `function` that gives us the length of a string; this is helpful if we forgot its length, or we have difficulty counting characters in very long strings. The function `length` returns the length of a string. Thus `length(a)` will return 8 and so will `length('abcdefgh')`. Naturally `length('')`, i.e. the length of the empty string is `0`. Given that in MATLAB a string is a row vector of MATLAB `char` data type, it is clear why `length` works in this context. (As a reminder `length` gives the length of a vector or the largest dimension of a matrix in MATLAB.)

4.37 String storage. Note that MATLAB is using UNICODE characters that require two bytes per character. Thus a string of length 10 characters will require a total of 20 bytes when stored in MATLAB. A `char` data type will be used to store a string in a `char` array of geometry/shape 1×10 for this example.

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

```
>> a = 'abcdefgh'   % a string of 8 characters
a=                  % note the indentation; strings is getting printed from the far left corner
abcdefgh
>> b = ''
b=   ''             % there is one exception: the empty string
>> whos
  a         1x8                16  char
  b         0x0                 0  char
>> length(a)
ans = 8
>> length(b)
ans = 0
```

4.38 Examples with strings. We define two strings `a, b`. String `a` is as defined above, i.e. `a= 'abcdefgh'`, whereas `b` is the empty string. The two variables used to represent the two strings are stored internally by MATLAB. If we forget their names, we can retrieve all variables defined by us, the user, by typing in MATLAB `whos`. Command (or function) `whos` provides us with information about the variables defined, such as their name, what array dimension they are, their length in characters if they are strings (this is our case), the amount of space used internally by MATLAB for them and some other classification of the variables. For our case `a` is an 1×8 array (i.e. a row vector). We can hypothesize that a (UNICODE) character in MATLAB is stored in two bytes since the total number of bytes used is 16. If the variables defined are no longer needed, we can clean them up by using the function `clear`. After we use the `clear`, the variables previously defined such as `a,b` are no longer available to us.

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

4.39 `int2str` **and** `num2str` The former converts an integer into a string of the appropriate length and the latter an arbitrary number (integer or not) into a string. The length of the string will depend on the digits of the number plus a possible sign (for negative numbers) plus a decimal point for real numbers.

Thus `num2str(-1.234)` will be converted into the MATLAB string '-1.234' of 6 characters stored into a `char array` of 12 bytes. Note that `-1.234` can be stored into a `double` in arithmetic form that requires only 8 bytes!

4.40 `double(x)` **and** `single(x)` `char(x)`. We can explicitly convert a matrix object from one data type to another. One needs to be careful in interpreting the result. Yet for the example below the weakly-typed nature of MATLAB is exemplified. In the first case a is a logical variable, in the second case it becomes a double, and then a character occupying 1, 8 and 2 bytes respectively! Function `single(x)` converts x into a 4-byte floating-point number as opposed to `double(x)` that converts x into an 8-byte floating-point number.

```
>> a=int8(123)     % A one-bytes integer 123 (int8 data type)
>> whos a
a  1x1   1  int8
>> b=int2str(a)    % Convert 123 into a string and store the string into b
b =
123
>> whos b          % b is '123' i.e. 6B (2 per character 1 2 3)
b  1x3   6  char
>> a= true;
>> whos
  a         1x1                         1  logical array
>> a=double(a);
>> whos
  a         1x1                         8  double array
>> a=char(a);
>> whos
  a         1x1                         2  char array
>>c=str2num('123')          % 123 will be converted into a number (i.e. double)!!
>> whos c
c   1x1   8  double
```

4.41 Default format. The default value of the `format` command is `short`. By default numbers in MATLAB are printed to 4 decimal digits (on the right of the decimal point) rounded up (thus a 3.14157 will be printed as 3.1416). If the number is too small or too large, the number will be converted into scientific notation and will be displayed with the use of `e` to denote 10^. The first three examples below show the possible output for a real number.

```
>> a=123.45                    |>> format bank
a = 123.4500                   |>> b
>> b=1234.5                    |b = 1234.50
b = 1.2345e+03                 |>> format rat
>> c=0.00012345                |>> a
c = 1.2345e-04                 |a = 2469/20
>> format short e              |>> b
>> a                           |b = 2469/2
a = 1.2345e+02                 |>> c
>> format short g              |c = 9/72904
>> a                           |
a = 123.45                     |
```

4.42 User specified format. If a default format is not desired, a user can change the default behavior.

```
1.   format short     :  default to 4 digits             ||    int8()  : convert to int 8-bit
2.   format long      :  14 digits after decimal         ||    int16() :               16-bit
3.   format short e   :  x.abcd [1+4digits] plus exponent||    int32() :
4.   format short g   :  best of 1 and 3                 ||    int64() :
5.   format long  e   :         [15digits]  plus exponent||   uint8()  : convert to unsigned 8-bit int
6.   format long  g   :  best of 2 and 5                 ||   uint16() :
7.   format bank      :  "dollar and cents" notation     ||   uint32() :
8.   format hex       :  hexadecimal                     ||   uint64() :
9.   format rat       :  rational approximation          ||   get(0,'Format');
10.  format compact   :  suppress line feeds             ||   format('short','g')
11.  format loose     :  restore  line feeds             ||
12.  format +         :  only signs are printed          ||   intmax,intmin,realmax,realmin
13.  format short eng :  exponent is multiple of 3
14.  format long  eng :  exponent is multiple of 3
```

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

```
>> format hex
>> 14
ans= 402c000000000000             % 14 is not integer 14 but double(14)
>> whos
  ans   1x1   8  double array
>>int8(14)
ans = 0e

>>whos
  ans   1x1   1  int8 array

>>int16(14)
ans = 000e
>>whos
  ans   1x1   1  int16 array
>>uint8(256)
ans= ff
>>uint8(1000)
ans= ff
```

4.43 Default representation of a number is `double` even if it looks to be an integer. Note that `format hex` will print in hexadecimal form the values of variables usually stored in double. If you want to print the corresponding integer you need to explicitly convert the double say 14 in the example above to an integer 14 using 8 or 16 bits. Then that integer will be printed as a hexadecimal 8-bit or 16-bit integer i.e. using 2 or 4 hexadecimal digits. Note that integer conversions truncate large numbers. Thus 256, and 1000 are represented in 8-bit as `ff` which is 255 in unsigned 8-bit format. If `int8(256)` were to be attempted the largest 8-bit integer is 127 and thus `7f` would be printed instead!

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

```
>> a=num2str(12345)
a =
12345
>> disp(a)  % a is a string containing characters 1 2 3 4 5
12345
>> a-0      % 1 2 3 4 5 in Unicode is  49, 50, 51, 52, 53 (and so are in ASCII btw)
ans =
      49             50             51             52             53
>> whos a
  Name      Size                     Bytes  Class
  a         1x5                         10  char array
>> disp(a,a,a)                    % Only one argument to disp!
Error using disp
Too many input arguments
>> disp([a a a ])
123 123 123
>> a= 123.45                      % a now is a number
  123.4500
>> disp([ a a a ])
 123.45000 123.4500 123.4500
>> fprintf('a=  %e a= %f a=%g',a,a,a)      % fprintf allows for more options!
a=  1.234500e+02 a= 123.450000 a=123.45>>
```

4.44 `disp`. Function `num2str` converts a number into a string of characters, as noted earlier.Then that array can be displayed with the use of function `disp`. Function `disp` displays strings or numbers in a more compact form that printing the variable's name and expecting its value to be displayed over possibly multiple lines. Command `disp` takes only one argument. If you need to print more than one variable or values you need to pack all othem in an vector/array/matrix.

4.45 `date, now, datestr(now), calendar, weekday(now)`. The `date` function returns date information. And `calendar` returns a current month calndar. Variable `now` contains time/date information. Thus `datestr(now)` returns date and time information. A `weekday(now)` would return day of the week information in the form of a integer (1 being a Sunday).

4.46 C language `fprintf`. We mentioned earlier that MATLAB has been implemented in *C*. Thus certain language features of the C programming language are available to MATLAB. Function `fprintf` is one of them. The format of `fprintf` is `fprintf(format,data)`. The `format` is a format string that contains placeholders (i.e. **conversion characters**) of where variables are to be displayed and in what format. The variable names is the list `data`. The variable names in data are separated by commas, and how many variables there are in the `data` are determined by the number of placeholders in `format`. This is a variable-number-of-argument function. For those of you proficient in C, note that the double-quotes `"` of C have become single-quotes `'` in MATLAB's `format` string.

```
fprintf('%d %f %s %8d %5.2f \n',1,2.3,'abc',4,pi);
```

In the `format` of the example above,

- `format` is `'%d %f %s %8d %5.3f \n'`
- `data` is `1,2.3,'abc',4,pi`.
- a `%d` indicates that a variable will be displayed as integer base-10,
- a `%f` indicates that a variable will be displayed as a real in floating-point format,
- a `%s` indicates that a string variable will be displayed.
- a `%8d` indicates that a variable will be displayed as an 8-digit integer base-10.
- a `%5.2f` indicates that a variable will be didsplayed as a real in floating-point format: exactly two decimal digits will be printed, the decimal point, and at least one integer digit (or as many as needed no matter what). The total length will be 5 including the 2 decimal digits and the period. If the number only requires 4 digits (and period) then a space will precede it to make 5.
- A one-to-one correspondence between placeholders and data will print `1` as `%d` , `2.3` as `%f`, `'abc'` as `%s`, 4 as `%8d`, and `pi` as `%5.2f`.
- `\n` is one character. It denote a newline; the current line will be completed and the printing cursor will move to the first character position of the next line.

Thus a **conversion character** such as the `d` in `%d` indicates an integer. An integer number can be inserted between the percent and the conversion character to indicate **length**. Another character can be inserted between the two such as `h, l, b` to indicate `subtype`. Instead of a single integer one can have `x.y` two integers separated by a period and the integer to the right of the period indicates the precision. By default justification is right; a $-$ after the percent but before x would denote left justification. Padded zeroes can be introduced with a 0 after percent, and a sign will be printed with a + (for negative numbers a sign is always printed anyway).

```
Conversion Characters (following the %)
***************************************
  %d   %i      :  integer base-10                 Example:  %d %i %10d  %8i
  %ld  %li     :  64-bit integer base-10          Example:  %10ld %8li
  %hd  %hi     :  16-bit integer base-10          Example:  %10hd %10hi
  %hu  %u  %lu : 16-bit 32-bit 64-bit unsigned integer base-10
  %ho  %o  %lo : 16-bit 32-bit 64-bit unsigned integer base-8  (octal)
  %hx  %x  %lx : 16-bit 32-bit 64-bit unsigned integer base-16 (hexadecimal a-f)
  %hX  %X  %lX : 16-bit 32-bit 64-bit unsigned integer base-16 (hexadecimal A-F)
  %f           : fixed point real number
  %e           : exponential notation           i.e. 3.1415e+00
  %E           : same as %e but with capital E  i.e. 3.1315E+00
  %g %G        : the more compact of %e and %f and %E and %f respectively
  %bx %bX      : double-precision hexadecimal
  %bo %bu      : double-precision octal and unsigned
  %c           : single character
  %s           : string of characters
Subtypes
********
  l,h,b   in %hd etc , %lu etc, %bo etc.
Width/Justification
*******************
 none : right justify   - : left justify   0 : padded zeroes  + : print sign
Examples: %10.7f  ,  %-10.7f ,  %010.7f  , %+10.7f
```

```
Special Characters (partial list)
*********************************
‘‘  : single quote           \a  : alarm (bell)   \t  : tab
%%  : single percent         \b  : backspace      \xN : hexadecimal ASCII/UNICODE char
\\  : single backslash       \n  : newline        \N  : octal       ASCII/UNICODE char

Examples. fprintf('\40') prints an octal, fprintf('\x20') prints hexadecimal
```

4.47 Special characters. Special characters can also be included in the format string: `\n` prints a new line; a `\t` prints a tab character; \\ prints a single backslash, `%%` prints a single percent (rather than being interpreted as a placeholder). A `\xN` prints the ASCII/UNICODE character with hexadecimal value N, and `\N` prints the char for that octal number.

4.48 Width. and justification. If a number appears between the percent and the placeholder character, it indicates minimum number of digits printed. For example `%8d` means at least 8 digits will be printed The more complex `%5.3f` means that at least 5 digits (including the decimal point) will be printed of which exactly 3 to the right of the decimal; this leaves room for a single integer digit. If the number to be displayed is 10.123 definitely we need a width of 6 characters not 5! And of course -10.123 would require 7 characters total. Thus `5.3` provides a suggestion rather than a hard limit. A `%-5.3f` left justifies (default: right), a `%+5.3f` prints a sign, and a `%05.3f` pads with zeroes.

(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]

String functions
  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)

Vector functions
[value,index ] = min(x)         % <- The percent indicates a MATLAB comment; the rest of the line is ignored
[value,index ] = max(x)         % min(x),max(x) return two value, the maximum/minimum and the index of x
                                % in which it occurs
```

(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 1: MATLAB Operators or Functions

Note. Scalar? means that they can be overloaded to become vector operators etc.

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 2: 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.