Introduction to MATLAB vector/matrix functions/operations

This is Chapter 2 of the textbook

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 2011 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis.

D1. Matrix and Vector storage.

D1.1. MATLAB is a weakly-typed language. In MATLAB a matrix is an abstraction used to store information. Variables that hold a single value are matrixes (or matrices) themselves. They are also called `scalar`. They are a 1×1 matrix. MATLAB matrixes can have elements of arbitrary data types. We mentioned in Subject 3 some of the available data types such as `logical, char, double, single, int8, int16, int32, int64, uint8, uint16, uint32, uint64`. These are built-in (or primitive) data types supported by MATLAB. A user can define his/her own composite data types as well.

D1.2 A MATLAB array stores a MATLAB matrix object in column-major form. MATLAB stores matrices internally, as we have already mentioned, in a data structure called `array` (Subject 3, C1.6), which holds a multiple number of elements of the same data type. Thus a string becomes an `char array` that holds multiple characters. An array stores a MATLAB matrix in **column-major** form.

D1.3. Matrix I/O in MATLAB is in row-major form. In MATLAB matrices are defined (i.e. typed) in row-major order. However MATLAB prints matrices in column-major order.

D1.6 Real numbers in MATLAB are of `double` data type. We discussed briefly in Subject 3 real numbers represented as floating-point numbers in MATLAB. The most common representation is in the form of a `double` which is 8-bytes long. Thus reals in the range from 10^{-308} to 10^{308} or so can be represented with up to 15 or 16 significant digits in a `double`.

D1.7 Significant number of digits of reals in storage and on display. We have just mentioned that reals numbers stored as `double` can maintain 15-17 significant digits. Yet, the default display of a real number shows only 4 decimal digits. This is because the default display is `format short`. An alternative `format long` print 15 decimal digits for a `double` but only 7 for a `single`.

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

```
>> d= [ 1 2 3 4 ]
d =
     1     2     3     4
>> c= [ 1; 2; 3; 4]

c =

     1
     2
     3
     4
>> f=[1 1 1 1];
>> d+f

ans =

     2     3     4     5
>> g= d'
g =
     1
     2
     3
     4
```

D2.1 Space or comma for column separation but semicolon for row separation. We can initialize scalar variables and matrices the same way. Thus in this example variable d becomes a 4-element `row-vector`. The left bracket [starts a matrix and thus also a row vector and a right bracket] concludes it. The elements are listed one after the other separated by spaces; one could have used commas as well to separate the elements of the single row but of multiple columns (column separator). If a `column-vector` is to be initialized such as vector c, we need to separate its multiple row elements with a semicolon instead (row separator).

Nnother row-vector f is defined and initialized. The two vectors d and f are both of the same structure/geometry i.e. dimension is 1×4. Thus we can perform a matrix/vector addition, where their elements are added element-wise (first element of d with first element of f and so on). The sum is shown above.

D2.2 Suppressing the echoing of input lines. If you type semicolon before hitting ENTER, nothing will be printed afterwards. This was the case with the f variable initialization, as opposed to the d variable initialization.

D2.3 Row vectors and column-vectors; Transpose operators. If you generate a row vector, a column vector can be generated the same way except in between the elements of the row vector you include semicolons to separate rows. Alternately you can define a row vector and then take its **transpose** using the quote ' which is the transpose operator. Thus $g = d'$ created a column vector g equivalently to (and equal to) c.

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

```
>> clear
>> whos
>> a = [ 1 2 3 ; 4 5 6; 7 8 9]
a =
     1     2     3
     4     5     6
     7     8     9
>> b=ones(3,3)
b =
     1     1     1
     1     1     1
     1     1     1
>> a+b
ans =
     2     3     4
     5     6     7
     8     9    10
>> a(2,3)
ans =
     6
>> c=[ 1 , 2, 3 ; 4, 5, 6; 7, 8, 9 ]
c =
     1     2     3
     4     5     6
     7     8     9
>> a-c
ans =
     0     0     0
     0     0     0
     0     0     0
```

In this example, we define a 3×3 matrix and assign it to a variable a. We define the matrix row by row separating row with semicolons. MATLAB however, as we noted before, will store matrices internally in column-major (form). We intend to initialize b to an all-one matrix. We can do it the standard way or we can use a MATLAB function `ones` that does it for us.

D3.1 `ones(n,m)`, `ones(n)` **function**. Generates a MATLAB matrix that has all of its elements equal to 1. What the dimension of the matrix is going to be depends on its parameters. Thus `ones(n,m)` will generate an $n \times m$ matrix of ones, but `ones(n)` an $n \times n$ matrix.

D3.2 How to retrieve an element of a matrix. For a matrix a we can retrieve the element of its second row and third column by writing $a(2,3)$. Thus for the matrix a so defined writing $a(2,3)$ returns 6 since the element in the second row and ultimate column is a `6`. (Note that we are discussing a not $a + b$!)

D3.3 Commas, spaces, and semicolons. As we mentioned earlier, semicolons splits rows of a matrix. Commas/spaces can be used to separate columns within a row. Thus c is initialized identically to a except that commas are being used to separate visually its column elements. The end-result remains the same; if we evaluate $a - c$ we get an all-zero matrix.

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

```
>> c(1,3)=100

c =

     1     2   100
     4     5     6
     7     8     9

>> c(1,4)=101

c =

     1     2   100   101
     4     5     6     0
     7     8     9     0
>> k=pi
k =
    3.1416
>> k(4)= -4
k =
    3.1416         0         0   -4.0000
```

D4.1 Modifying a variable that stores a non-trivial matrix. What happens if we write $c(1,3) = 100$? This will modify the element of the first row and third column from its previous value (it was a 3) into its new value 100. But what if we do a $c(1,4) = 101$, by accident or by a design. MATLAB is forgiving but behaves unexpectedly. The side-effect of this is that MATLAB will extend our 3×3 matrix c into a 4×4 matrix and then set the element of the first row and fourth column into a 101. **WATCH-OUT for side-effects!** The creation of the new matrix and its initialization will cost $4 \times 4 = 16$ operations, the copying of the old will cost an additional $3 \times 3 = 9$ and finally $c(1,4) = 101$ will cost one more for a total of 26.

D4.2 Modifying a variable that stores a scalar value. What happens if we introduce a scalar variable k, set it to π, but then we accidentally set $k(4) = -4$. MATLAB will extend the 1×1 vector (or matrix) into a full-blown 1×4 row-vector!

D4.3 Special symbols, side-effects, and overriding default behavior. Variable i, j and pi are special symbols. Both i, j are the square roots of -1 i.e. they are complex numbers. Setting $i = 1$ changes the default behavior: a command window variable of the same name is defined that takes the value of 1. The same applies to pi and other special symbols.

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

```
>> a=[1:3]'
a =
     1
     2
     3
>> b= [ a' a']
b =
     1     2     3     1     2     3
>> c=[a a]
c =
     1     1
     2     2
     3     3
```

D5.1. Shortcut expressions. Certain arrays in MATLAB can be initialized using shortcut expressions; this is faster than writing the elements one by one. Several elements of a sequence of elements are generated using a short shortcut expression. A shortcut (colon) expression has the form

`first:increment:last`

where `first` is the value of the first element in the sequence that will be generated in the form of a row vector,
`increment` is the value that will be added to the previously generated value (or `first`) to generate the next value of the sequence, and
`last` is an upper bound of the last value that will be generated, in the sense that any value generated is NO MORE than `last` (i.e. it can be equal or less than `last`). Thus the sequence
`first, first+increment, first+2*increment` will be generated. The last value that will be generated is
`first + k*increment` where k is the largest integer such that $\texttt{first} + \texttt{k} * \texttt{increment} \leq \texttt{last}$.
Instead of writing

`first:1:last`

one can just write

`first:last`

implying that the increment is one.

In general

a:b:c

If $b > 0$ and $a > c$ returns an empty vector, and if $b < 0$ and $a < c$ it also returns an empty `1-by-0` vector.

D5. Shortcut-expression initialization continued

Examples.

(a). `1:2:10` generates the odd (positive) numbers less than 10, i.e. $1, 3, 5, 7, 9$, since `1:2:10` generates `1, 1+2, 1+2+2, 1+2+2+2, 1+2+2+2+2`.

(b). `1:3` is `1 2 3` and `a= [1:3]'` generates a vector a with `1 2 3` as its elements of rows 1,2,3. Then `b=[a' a']` goes one step further and generates one longer (twice as long) row i.e. a 1×6 array since a is a column-vector, a' becomes a row-vector and thus b become a row-vector of twice as many columns as a'. Finally $c = [a \quad a]$ generates a 3×2 matrix with two identical columns each one being equal to a.

(c). `1:1:3` is another way to write `1:3`.

(d). `1:-1:0` generates `[ 1 0 ]`.

(e). `1:1:0` generates an empty 1-by-0 matrix.

(f). `1:end` might work for row/column vector and matrices as well to fill rows/columns.

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

```
>> a=ones(3)
a =
     1     1     1
     1     1     1
     1     1     1
>> b=ones(3,3)
b =
     1     1     1
     1     1     1
     1     1     1
>> c=zeros(size(a)+1)
c =
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
>> eye(2,2)
ans =
     1     0
     0     1
>> eye(2,3)
ans =
     1     0     0
     0     1     0
>> eye(3,2)
ans =
     1     0
     0     1
     0     0
```

One can initialize matrices using the built-in functions.

D6.1 zeros(), zeros(n), zeros(n,m). If one invokes `zeros(n)` then an $n \times n$ matrix of zeroes is created; one can equivalently issue `zeros(n,n)`. If `zeros (n,m)` is issued then an $n \times m$ matrix of zeros is created. `zeros()` will print one zero.

D6.2 ones(),ones(n),ones(n,m). The discussion on `zeros` applies to the `ones`.

D6.3 eye(n) . Function `eye(n)` generates an $n \times n$ identity matrix whose diagonal is all-one and non-diagonal entries are zeroes. `eye(n,m)` with $n \neq m$ is idiosyncratic; depending on whether $n < m$ or $n > m$ it creates an $n \times n$ or $m \times m$ identity matrix with the remaining columns (respectively) rows equal to zero.

D6.4 `diag(a)` and `diag(diag(a))` , `triu(a),tril(a).` Function `diag` isolates the diagonal of a matrix i.e. the elements for which row and column indexes have the same value. However `diag(diag(a))` will create a matrix whose diagonal is the diagonal of a and everything else is 0. The matrix is square with number of rows/columns the elements of `diag(a)`. `triu(a),tril(a)` will extract the upper or lower triangular matrix of a and fill with zero the remaining elements.

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

D6.4 length(a) For a vector, it returns its length (number of elements), for a matrix it returns the longest/largest of its dimensions.

D6.5 size(a), size(a,1),size(a,2) It returns a row vector with two elements, the number of rows and columns of matrix a. One can retrieve them separately, by typing `size(a,1), size(a,2)` respectively.

In the example above `size(a)` returns two elements (i.e. a row vector of size 2). What is the effect of `size(a)+1`? Food for thought...

D6.6 end It is a variable that relates to the size of a matrix a. If it is used in a row index it holds the number of rows, if it is used in a column index, it holds the number of its columns. For a matrix a observe the output of $a(1:end, 1:end)$ for example.

D6.7 sum, sort For a vector x you can sum its elements by operation $sum(x)$. If x contains numbers then $sum(x)/length(x)$ gives the average value of the elements in x Thus for $x =$ [30 10 20], we get 20. And $sort(x)$ will return the elements of x sorted in increasing (non-decreasing order) i.e. [10 20 30].

D6.8 min,max For a vector x, `min(x), max(x)` return the smallest element/largest element of x. If one wants to know the index one case retrieve two value from `min,max`. Thus $y = minx(x)$ for the example of D6.7 returns `ans=10` but a `[a b] = min(x)` returns $a = 10, b = 2$. Note that `[a b]` indicates that we want `min` to return a row vector of two elements not just one; in that case the first is the minimum value and the latter the index of it in x.

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

```
>> a=input('Enter scalar ')
Enter scalar pi
a =
    3.1416
>> b=input('Enter row vector')
Enter row vector[1 2 3 ]
b =
     1     2     3
>> c=input('Enter string','s')
Enter string12345
c =
12345
>> c*2
ans =
    98   100   102   104   106

>> whos c
  Name      Size                    Bytes  Class
  c         1x5                        10  char array
Grand total is 5 elements using 10 bytes
```

D7.1. `input`**.** We can read an array or matrix through the keyboard. For this we can use the function `input`. Since `input` is a function its first parameter is a string that is printed on the terminal/window. A second parameter may or may not exist. If it is present and is equal `'s'` then the input read is converted into a string and then returned.

Examples. In the example above a requests that a scalar value be read. The user supplies the variable `pi` which is evaluate and its value assigned to a. Note that `'Enter scalar '` has a space before the quote, as opposed to `'Enter row vector'` that requests a vector. Caution should be drawn in the case of variable c. Because of `'s'` we know that the input will be converted into a string. Thus `12345` is converted into a character array that contains 5 characters: 1,2,3,4,5. Thus when one types $c*2$ the result is not a scalar $2*12345 = 24690$ but the doubling of the ASCII values of 1,2,3,4,5 which are respectively 49,50,51,52,53. A `whos c` on c can convince us that this is indeed the case.

```
>> a = [ 1 2 3 ; 4 5 6 ; 7 8 9]
a =
     1     2     3
     4     5     6
     7     8     9
>> a(4)
ans =
     2
>> a(9)
ans =
     9
```

D8.1. Array vs Matrix. What is an array? An array in MATLAB is the implementation of the mathematical construct known as matrix. The textbook interchangeably uses matrix and its representation as an array to refer to variable a. In the next subject, we will define operations on arrays that differ from those of matrices. For example multiplication on arrays is an element-wise operation different from the matrix-multiplication defined previously.

D8.2. Two-dimensional matrix as an one-dimensional matrix A two dimensional $n \times m$ matrix a in MATLAB is stored in column major form. To retrieve its element of row i and column j we just write $a(i, j)$ using parentheses. However we can retrieve that element by viewing a as a one-dimensional entity, a long column or vector or elements. Then what is the m for which $a(m)$ retrieves $a(i, j)$? In column-major form, element $a(i, j)$ is the $m = (j - 1) * n + i$-th element.

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

```
>> a
a =
    1.1000    2.2000    3.3000    4.4000    5.5000
>> a([1 4])
ans =
    1.1000    4.4000
>> a(1:2:5)
ans =
    1.1000    3.3000    5.5000
>> a(3:end)
ans =
    3.3000    4.4000    5.5000
>> a(1:2:5) = [10 20 30]

a =

   10.0000    2.2000   20.0000    4.4000   30.0000
>> a([1 2]) = [100 200]
a =
  100.0000  200.0000   20.0000    4.4000   30.0000
```

D9.1 Subarrays/Subvectors In MATLAB we can retrieve subarrays of an array/vector by specifying what rows or columns we are retrieving. We can use shortcut expressions as well for this selection. `end` denotes the highest index value for a row or column or other dimension subscript. Thus in a two dimensional matrix, a row instance of `end` might take different values than a column instance of `end` if the number of row and columns is different.

Example. Consider `a= [ 1.1 2.2 3.3 4.4 5.5]` . Then $a(3)$ retrieves its third element i.e. 3.3 (note a typo in the textbook on page 35 involving an array `arr1`). If we want to retrieve/read more than one element we can groupd the relevant indexes in a row vector. Then $a([1 \quad 4])$ retrieves only the first and fourth column i.e. returns 1.1 4.4 . Note that the column indexes 1 and 4 are grouped into a row vector. And $a(1:2:5)$ retrieves the odd-indexed columns i.e. 1.1 3.3 5.5 in the form of a row vector. Note again that $1:2:5$ is not surrounded by brackets. The shorcut colon expression returns a row vector instead. If $b = a'$, then b is a column vector. For two-dimensional matrices, such as the 3×3 matrix c, we have that $c(1,:)$ retrieves its first row, and $c(1:2:3,:)$ retrieves its odd-indexed rows. If we write $a(3:\texttt{end})$, we get 3.3 4.4 5.5.

D9.2 Subarrays on the left side of an assignment. If we use a subarray on the left-hand-side of an assignment we update only some of the values (the matrix dimensions remain the same) of the matrix.

Example. Consider `a(1:2:5)` . It modifies the 1st,3rd,5th elements to the values specified by the array on the right-hand side. And `a([1 2])` changes the values of the first and second columns respectively.

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

```
>> a= [ 1 2 3 ; 4 5 6 ; 7 8 9];
>> a(1:2,1:2) = 10

a =

    10    10     3
    10    10     6
     7     8     9
```

D9.3 Assigning scalar values to arrays/subarrays. If a scalar value is applied to a matrix range (submatrix), then all the affected entries are modified to that value.

Example. For the example above, a $a(1:2, 1:2) = 10$ means that the top-left subarray of a defined by the top-two rows and columns is going to have its values modified to 10.

D9.4 Deleting a row or a column. An operation such as **a(:,3)=[]** would delete the third column, whereas **a(3,:)=[]** would delete the third row.

D9.5 Generating vectors for numerical solution of differential equations. Let us say you want to define a time interval between 0 and $t = 1$ and you want to split this interval into subintervals with step $h = 0.2$. One way to effect this is by typing `t= 0:0.2:1` to store the result in array $t = [0 \quad 0.2 \quad 0.4 \quad 0.6 \quad 0.8 \quad 1.0]$. However in some cases one would like to retrieve index information (i.e. the first, second etc interval.) Then an operation, such as $h = t./0.2$ would achieve this: note we use to array division to divide each element of t by the step 0.2. And also note that even if t is a vector, 0.2 is just a number. MATLAB is intelligent enough to realize what one wants to do i.e. divided each element of t with a real number. The only problem is that h is an array of real numbers. For indexing we need integers. One can then do $h = int32(h)$, or $h = floor(h)$. Note something interesting for h. Although MATLAB array indexes start from 1 the elements of h start from 0. Thus $h(1) = 0, h(2) = 1, h(3) = 2, h(4) = 3, h(5) = 4, h(6) = 5$. If you don't know how many elements there are in t a `length(t)` will give the answer (or `length(h)`).

D9.6 Shifting a vector to the left one position. We can cut and paste t at will. Thus $t = t(2 : end)$ will eliminate its first element and shorten t by one. But let's go back to the original t. We can shift its elements by one by performing the operation $t([2 : end \quad , 1]) = t$. This will form a new t that would include $t(2), t(3), t(4), t(5), t(6), t(1)$ in that order. We used subarrays to achieve this

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

```
>> a= [1 2; 3 4]
a =  1  2
     3  4
>> a(5,5)= 10
a = 1  2  0  0  0
    3  4  0  0  0
    0  0  0  0  0
    0  0  0  0  0
    0  0  0  0 10
% A more efficient solution is
>> a= zeros(5,5);
>> a([1 2],[1 2])= [1 2;3 4];
>> a(5,5)=10;
```

D9.6 Dynamic Reallocation of matrices. In the example above, we define a 2×2 matrix a. Yet later we try to access element $a(5,5)$ belonging to its fifth row and column and change its value into 10, even if that element does not exist in the 2×2 matrix. MATLAB non only does not complain about this mistake of ours, but its works nicely and correctly. The reason for this is that MATLAB interprets the `a(5,5)=10` as a request to resize a into a 5×5 matrix and set its last element (bottom rown and column) to 10. For that MATLAB does the following

```
Step 1. Create a new 5x5 matrix called say temp
Step 2. Initialize elements of new matrix temp to zeroes.  25 operations
Step 3. Copy all elements of a into new matrix temp.        4 operations
Step 4. set temp(5,5)=10                                    1 operation
Step 5. rename temp into a, and discard  old a.
```

The whole sequence requres at least $5 \times 5 = 25$ steps to create and initialize temp to zeroes, and $2 \times 2 = 4$ steps to copy `a` into `temp` plus other operations. Though for a 2×2 and 5×5 matrix this is not very costly, if we have a $10,000 \times 10,000$ matrix b and then set `b(100000,100000)=10` this would become quite a costly operation. Avoid at all costs such a dynamic reallocation of a matrix. If you know in advance that the matrix will increase to a 5×5 just use the last three lines in the MATLAB code above instead! (More in Subject 7.)

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

```
>> a= zeros(5,5);
>> a([1 2],[1 2])= [1 2;3 4];
>> a(5,5)=10;
>> a+1 ;
>> a-1 ;
>> a>1 ;
```

In the example above and using the matrix of the previous section we perform operations $a+1$, $a-1$ and $a>1$. The first two $+,-$, are interpreted by MATLAB to be array/matrix operations involving addition and subtraction. Even though the second operand is not a matrix of the same dimensions MATLAB turns the scalar value into a matrix of the same dimensions as a whose entries are all equal to the scalar value (i.e. a `ones(5)` matrix) and then addition and subtraction is performed. For the third operation, every element of a is compared to 1 and the results is stored into a 5×5 matrix `ans`. In this latter case the data type of each element of `ans` will be a `logical` one-byte incarnation of true and false (i.e. of zeroes and ones). In the former two case a is 5×5 matrix of `8-byte double` elements.

```
Operation      Array operator     Matrix Operator   Comment

Addition        +  i.e. C=A + B   + i.e. C=A+B      Element-wise addition   ;A,B,C are all nXm
Subtraction     -  i.e. C=A - B   - i.e. C=A-B      Element-wise subtraction;A,B,C are all nXm
Multiplication  .* i.e. C=A .* B  * i.e. D=E*F      .* : element-wise A,B,C are all nXm
                                                     *  : matrix multiply D : nXm E: nXp F: pXm
Division        ./ i.e. C=A ./ B  / i.e. D=E/F      ./, .\ element-wise A,B,C are all nXm
BackDivision    .\ i.e. C=A .\ B  \ i.e. G=E\F      D= E*inv(F) and G=F*inv(E)
                                                    For a square matrix Z nXn, inv(Z) is also nXn
                                                    such that Z * inv(Z) = inv(Z) * Z = I
```

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

```
ones(n)    ones(n,m)       : n x n ,  n x m  all-1 matrix
zeros(n)  zeros(n,m)       : n x n ,  n x m  all-0 matrix
eye(n)     eye (n,m)       : n x n identity matrix, n x m identity matrix (see previous page)
rand(n)   rand(n,m)        : nxn, nxm array of (pseudo)random numbers between 0 and 1.
 ones(size(x)),
zeros(size(x))             : generates all-1 or all-0 of same dimension as matrix x

length(x)                  : return length of vector x or matrix x
size  (x)                  : returns number of rows and columns in x.

input(string,format)       : read from the keyboard the initialization expression
                             if format is present and equal to 's' a string is returned.

whos                       : list all variables and their definitions
whos x                     : list definition of variable x
clear                      : erase all variables
dir, ls, pwd               : directory listing and current working directory
clear                      : clear variables
clc                        : clear screen
```

D10.1. List of builtin matrix initialization functions

```
 pi    : the 3.14 stored internally to 15 digts
 i,j   : the square root of (-1) used in conjunction with complex numbers
 inf   : computer's representation of infinity
 nan   : representation of NaN, i.e. Not-a-Number
 clock : Array containing current date/time
 cputime: cputime in seconds since the start of a session
 date  : String containing current date
 eps   : Epsilon. Two numbers whose absolute value is less than eps are considered
         equal.
 ans   : answer variable for a result not assigned to some other variable.
```

D10.2 List of special symbols/variables

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

D11.1 Default format. By default number in MATLAB are to 4 decimal digits (on the right of the decimal point). 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 above show the possible output for a real number.

D11.2 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
```

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)
12345
>> a-0
ans =
      49            50            51            52            53
>> whos a
  Name      Size                    Bytes  Class
  a         1x5                        10  char array
Grand total is 5 elements using 10 bytes
>> a= 123.45
  123.4500
>> fprintf('%e %f %g',a,a,a)
1.234500e+02 123.450000 123.45
```

D11.3 `disp, int2str, num2str, str2num`. Function `int2str` converts an integer to a string of characters and `num2str` a number (real or integer) into a string of characters. Then that array can be turned back into a number with the use of function `disp`. Function `disp` displays strings or numbers; you cannot mix and match; or you need to convert numbers into strings with one of the two functions `int2str,num2str`. Function `str2num` from the previous subject can convert a array of string numbers into an array of numbers. For `disp` its argument must be a string. If we want to display a message such as `a= 123.45` one needs to do something like `disp(['a= ', num2str(123.45)]);`. In this case the argument to `disp` is the concatenation of two strings: one is `a=` and the other is the string of the number `123.45`. In MATLAB an row vector of strings is the string obtained by the concatenation of its elements. The difference between `123.45` and `num2str(123.45)` is that the formed is stored in a `double` that is 8-bytes long. In the latter case the number is converted into a string of 6 characters; since characters in MATLAB are UNICODE, this string will require 12 bytes for storage.

Example.. With reference to the example above, integer `12345` is converted into a string `12345` using `num2str`. This is not obvious from the output `12345` that looks like an integer. However applying `disp` function to a converts it into an integer and prints it as an intger. To convince ourselves that a is not an integer, let's try to subtract 0 from a. It looks silly, but it is not; a is being printed as an array of 5 ASCII characters: 1,2,3,4,5. Of course had we done `whos a`, the structure of a would have been obvious (i.e. it is a character array not scalar value). Also `str2num([ '1 2 ' ; '3 45'])` will create a 2×2 array with first row $1, 2$ and second rows $3, 45$. Note that the strings have the same length 4, i.e. there is a space after the two to compensate for the 5 of 45. The same result can be obtained by `str2num([ '1 2 ' ; '3 4' '5'])` since the strings `'3 4'` and `'5'` will be concatenated before converted.

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

D11.4 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
```

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

```
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
```

D11.5 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.

D11.6 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.

```
>> a
a =
  123.4500
>> save filename a
>> clear
>> load filename
>> a
a =
  123.4500
>>
```

D11.7 Programs and Data: `load, save` In MATLAB you can write data files of variables into a file using the `save` and load them all at once using the `load` function/command. Thus `save filename var1 var2 var3` saves into a filename `filename.mat` the variables `var1, var2, var3` (not just the names but also their values). If one does not define the variable names (i.e. `var1 var2, var3` the all of variables currently defined will be saved. (Currently defined variables are those listed by `whos`.) Note that the file name written on disk is going to be `filename.mat`; do not try to read it since it is intended for MATLAB not humans to read it. If you want to write data in ASCII format, you could use `save -ASCII filename var1 var2 var3`. In that case the file will indeed be `filename` and not `filename.mat`. You can reload the saved file and reinstate the variable even after clearing up all variables (using a `clear` function) using a `load filename` or `load -MAT filename` to load `filename.mat` or `load -ASCII filename` to load `filename` saved in ASCII. An ASCII filename can be read by humans by doing `type filename` from the MATLAB command window. In an existing .mat file one can also append information by using `save -append filename` instead.

D11.8 Refresher `M-files startup.m finish.m` A MATLAB M-file is a file whose filename ends with `.m` that contains MATLAB commands/functions. Matlab is also using two special filenames at startup and finish. As soon as MATLAB starts it reads the functions/commands in file `startup.m` and executes them. When MATLAB is about to exit (i.e. after you `quit`), it executes the commands in `finish.m`. Those files are located in the same directory from which MATLAB was invoked. What this directory is can be shown by typing `pwd` immediately after getting into MATLAB (i.e. its command window).

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

D12.1 Array operations $+, -, .*, ./, .\backslash, .\hat{}$**.** For two MATLAB arrays of the same shape a and b, operations $+$, $-$, $.*$, $./$, $.\backslash$, $.\hat{}$ are defined as element-wise operations between the two matrices that is for $a(i,j)$ and $b(i,j)$ the resulting element $c(i,j)$ is one of $a(i,j)+b(i,j)$, $a(i,j)-b(i,j)$, $a(i,j)*b(i,j)$, $a(i,j)/b(i,j)$, $b(i,j)\backslash a(i,j)$, $a(i,j)\hat{}b(i,j)$, respectively.

D12.2 Matrix operations $+, -, *$ Since an array is the representation of a matrix in MATLAB, matrix addition and matrix subtraction are identical to array addition and subtraction. This is 5.16 and 5.17 of Subject 2. For matrix multiplication, the definition of 5.19 of Subject 2 applies i.e. the result $c(i,j)$ is such that $c(i,j)=\sum_k a(i,k)*b(k,j)$. The two matrices/arrays should also have the proper form, i.e. the number of columns of a is equal to the number of rows of b.

D12.3 Matrix operations $/, \backslash$ For two matrices a and b, the quotient a/b could be defined as the matrix product of a and the inverse $1/b$ of b. Thus the inverse of a matrix needs to be defined; for scalars, $1/b$ is the inverse of b if and only if $b(1/b)=(1/b)b=1$. Similarly for a matrix b the inverse $b^{-1}=\texttt{inv}(b)$ is defined as that matrix so that $bb^{-1}=b^{-1}b=I$, where I is the identity matrix. Therefore a/b is defined as $a*\texttt{inv}(b)$ and $a\backslash b$ is defined as $b*\texttt{inv}(a)$ in MATLAB. Thus for / and \ to be define $\texttt{inv}(b)$ and $\texttt{inv}(a)$ must exist and be defined. There is no equivalent to $.\hat{}$ for matrices.

Operation	Matrix	Array
Addition	a + b	a + b
Subtraction	a − b	a − b
Multiplication	a ∗ b	a .∗ b
Division	$a/b = a*$inv(b)	a ./ b
Back-Division	$a\backslash b = b*$inv(a)	a .\ b

Figure 1: Matlab Array and Matrix operators

D13.1 `max, min` **on vectors.** Operator `max` can operate on a vector and finds the maximum of its elements. Similarly `min` finds the minimum of its elements. Both operators can return two values, as opposed to scalar operators that return only one. In that case both return not only the value that is maximum/minimum but also the vector index storing the maximum/minimum.

D13.2 Other builtin functions on vectors/matrices. All the builtin operators defined in Subject 3 such as `sin(x), sqrt(x),` etc work also on inputs that are vectors and matrices. In that case, the operator applies to each element of the vector/matrix.

D13.3 Non-traditional behavior on some inputs. Functions that by default are undefined on some inputs might have MATLAB equivalents that return surprising results. For example consider `sqrt`, the square root scalar function. By D13.2, it can be extended to work on vectors and matrices. What if the input is a negative real number or a vector/matrix has elements that are negative numbers? The MATLAB realization of `sqrt` will work just fine. For negative numbers it returns a complex number. In fact complex number include all real numbers and they include only one extra symbol: an i (sometimes denoted by j) such that $i^2 = j^2 = -1$, i.e. the extra symbol is the $\sqrt{-1}$. Thus for a negative number $-x$, where $x > 0$, the $\sqrt{-x}$ would be simply $\sqrt{-1} * \sqrt{x} = i * \sqrt{x}$, i.e. `sqrt` is well-define for real numbers if i (or j) is available, i.e. for complex numbers.

D13.4 `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).

D13.5 `clock, cputime, tic, toc`. The `clock` function returns clock time in a single variable. Function `cputime` shows the CPU time since the command window was invoked. You can start a specific timing with a `tic` function call and terminate it with a `toc` function calls. The elapsed time between the `tic` and the `toc` will be shown. A `clear` command will not affect the `tic` time.

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