Universidade Nova de Lisboa
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT
Faculdade de Ciências e Tecnologia

Introduction; Data Types (in MatLab)

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2018/2019

Introduction

- This course introduces the basic concepts of
  - Programming,
  - Algorithms; and
  - Databases
- The language adopted is MatLab
  - To introduce basic programming concepts and algorithm techniques, with an interface to sqlite (databases)
- The database tool used in sqlite
  - To introduce the topic of databases and the language SQL

- All information about the course (synopsis, lectures, exercises, rules, lecturers, etc.) is maintained in the web page
  - **http://mc.ssdi.di.fct.unl.pt**

Programs and Algorithms

- In very abstract terms, a computer program can be regarded as a set of instructions that applied to input data yields some result as output.

- An algorithm is a sequence of instructions, written in a programming language, understood by some executor.
- Programs and data are stored in computers, that are able to execute the stored programs on the data (input or stored) to produce other data, eventually sent to the output.

Programming Languages

- A program may be specified at different levels of abstraction. The more knowledgeable the executor is, the more high level the program may be specified.
- At the lowest level, such information is coded in binary digits (bits) and stored somewhere in the memory of the computer. Processing it requires the execution of machine instructions (almost directly encoded in assembly) that command the computer to move and transform the bit-encoded data.
- For example, one may instruct to add the data stored in memory positions 1001 and 1002, and store the result in position 1003, by means of the assembly code below

```
LDA 1001
LDB 1002
ADD A,B
STA 1003
```

- Of course, specifying a program at this low level requires the user to keep track of all the details regarding the encoding of data, the positions in memory it is stored in, etc, which is very “unpractical”.

Programming Languages

- High level languages allow the user to specify the programs in a more human readable form, so easing the task of programming.
- For the previous example, where we wanted to sum two numbers, high level languages allow this instruction to be specified as (or similarly to)

```
C = A + B
```

- Of course, in this case, the executor of the program must take care of all the details mentioned before, namely that the memory positions where the data corresponding to the numbers A, B and C is stored.

Programming Languages

- In fact there are several types of high-level programming languages with different features namely:
- **Imperative Languages** : Fortran, Pascal, C, Octave/MATLAB
  - With an explicit control of execution
- **Object Oriented Languages**: Smalltalk, C++, Java, Python
  - Allowing abstraction of data objects, with arbitrarily complex properties.
- **Functional Languages**: LISP, Scheme
  - Programs are specified as functions, rather than “instructions”
- **Logic Languages**: Prolog, ASP
  - Programs are specified as predicates of 1st order logic
- **Database Languages**: SQL
  - Mixing features of the above, to query databases.

Programming Languages

- In practice, programming at high level requires several “programs” namely:
  - An **editor** – to write down the programs and store them in text files.
    - Currently, these editors are often included in IDEs (Integrated Development Environment) that help organizing and inspecting different components of the programs.
  - An **executor** – to interpret and execute the instructions
- There are two main types of executors to deal with programs:
  - **Compilers**: Whole programs (or rather components) are translated into machine language, producing an executable file (e.g. and exe file) that can be executed from the operating system.
  - **Interpreters**: Programs are executed instruction by instruction, from the IDE (that can be as simple as a console) and no executable file is produced.
- In this course we will adopt the imperative language **MatLab**, and use the **Octave** IDE (free download – instructions in the course web page).

Basic Operations

- In the imperative programming paradigm, the programmer specify explicitely the control of execution, i.e. the sequencing of the basic operations.
- Informally, we may consider the following basic instruction in a high-level language like MatLab.
  - **Assignment**: `V = Expression`
    - Variable **V** takes the value of the **Expression.**
  - **Input**: `Input V`
    - Variable takes a value read from some external device (e.g. from a file)
  - **Output**: `Output V`
    - Variable V is written to some output device (e.g. to a file, or a printer).
- Although in most cases values are simple numbers, assignment operations can be used with more complex data structures and with data of different types.
- The exact syntax and the data that can be involved in these operations depend on the programming language adopted. With no loss of generality, we will adopt the MatLab language in this course.

Data Types

- In assignment operations several types of data can be used. The simplest data types, as used in MatLab, are
  - Numeric;
  - Boolean (True / False)
  - Characters
- Notice that these types are relevant from a programming language perspective. At the machine level they are all encoded into sequence of bits (e.g. 16, 32 or 64 bits) stored in memory, depending on the architecture of the computer being used.
- The executor of the language should be able not only to keep track of the memory positions where the data is stored but also the interpretation it does to the bits.
  - For example, the byte **00101000**, can be interpreted as the character ASCII character "**(**", or the integer **40**, or even the boolean **True**;
- In many languages (but not in MatLab) the programmer must explicitly declare the type of the variables to be used, so that not only they may be efficiently encoded but also to support a compiler to debug a program.

Numeric Operations

- Assignments instruction such as

```
V = Expression
```

  assign the value of the right-hand expression to the left-hand variable. Hence the left hand **must always be a variable**.

  - Notice that the = sign does not represent equality, but rather assignment!

- **Syntactically**, a variable is denoted by a sequence of letters (including _) and digits, started by a letter (note that upper and lower case letters are considered different)
- As to the expression, they are composed of operations opn both variables and constants of the appropriate type.
- For example, **numeric variables** can be assigned with expressions including the usual arithmetic operators (sum, subtraction, multiplication, division and exponentiation) possibly using parenthesis to take into account the usual precedence of operators.

  - Notice that division is always real division, even when applied to integer values.

Numeric Operations

- The following examples can be tested in the Octave console

```
>> 3 + 7;
    ans = 10
>> a = 3 * (4 + 6^2) - 6
    a = 114
>> b = 5^0
    b = 1
>> 5 / 0
    warning: division by zero
    ans = Inf
>> Inf * (-1)
    ans = -Inf
>> c = Inf + Inf
    c = Inf
>> Inf - Inf
    ans = NaN
```

Numeric Operations

- In addition to the standard operators, there are a number of useful **functions** that are available in MatLab (as well as in other languages)
  - **`abs/1`**
    - obtain the absolute value of the argument
  - **`ceil/1, floor/1, round/1`**
    - convert reals into integers
  - **`rem/2, mod/2`**
    - obtain the remainder of the integer division
  - **`factorial/1`**
    - obtain the factorial of an integer
  - **`sqrt/1`**
    - obtain the square root of an number
  - **`log/1, log10/1, exp/1`**
    - Implement the logarithm and exponential functions (base e and base 10)
  - **`sin/1, cos/1, tan/1, asin/1, acos/1, atan/1`**
    - Implement the trigonometric functions on the arguments in radians
    - Note: **pi** and **e** are provided as variables pre-assigned with their values.

Numeric Operations

- Examples:

```
>> abs(-3)
    ans = 3
>> a = 5.13; rem(floor(a),3)
    ans = 2
>> b = 5.13; c = round(b), factorial(c)
    c = 5
    ans = 120
>> a = sqrt(16), b = sqrt(2), c = sqrt(-4)
    a =  4
    b =  1.4142
    c =  0 + 2i
>> log(10), log10(1000), e^1
    ans =  2.3026
    ans =  3
    ans =  2.7183
>> cos(pi/2), sin(90)
    ans =    6.1232e-17
    ans =  0.89400
```

Boolean Operations

- As to **Boolean variables**, the expressions involve the Boolean operators for
  - && conjunction,
  - || disjunction; and
  - ! (or ~) negation
- as well as the relational operations for
  - == equality,
  - != (or ~=) disequality; and
  - >=, =< inequality
  - > , < strict inequality
- In MatLab, there is a bridging between numbers and Booleans.
  - In fact, number 0 is considered as **False** in a Boolean expression. All the other numeric values are considered **True**.

Boolean Operations

- Examples:

```
>> a = true && ! False
    a = 1
>> b = 56 && sqrt(0), c = 56 && cos(pi/2),
    b = 0
    c = 1   % beware of effect of precision
>> d = exp(0) > 9
    d = 0
>> e = sqrt(0) == 0, f = sin(2*pi) == sqrt(0)
    e = 1
    f = 0 % beware of effect of precision
>> 8 || 5 == 2 && 3
    ans = 1
>> s = (a == 1) + (b == 1) + (c != 0)
    s = 2
```

- **Note**: Boolean values (variables and expressions) are mostly used as to express conditions in conditional and iterating instructions.

Arrays: Vectors and Matrices

- In most applications information is naturally structured in arrays, and programming languages have support for these data structures.
- In general, arrays may be unidimensional (vectors), bidimensional (matrices) or of higher dimensionality.
- In fact, MatLab was designed so as to consider matrices as the primitive data type, hence its name: Matrix Laboratory.
- Vectors and matrices are specified in MatLab, delimited by square brackets.
  - Row Vectors have their elements separated by commas or spaces;
  - Column Vectors have their elements separated by semicolons.
  - Matrices have rows separated by semicolons, with elements separated by colons or spaces.
- Elements of an array (vector or matrix) are identified by their indices, specified in round brackets.
- The indices in all dimensions of any array are counted starting at 1.

Arrays: Vectors and Matrices

- Examples:

```
>> A = [ 4 7 5]
    A = 4 7 5
>> x = A(2)
    x = 7
>> B = [4;5;6]
    B = 4
        5
        6
>> y = B(3)
    y = 6
>> M = [ 3*3 2*4 7*1; 6,5,4; 3,2,1]
    M = 9 8 7
        6 5 4
        3 2 1
>> M(3,1)
    ans = 3
>> A(4)
    error: A(I): index out of bounds; value 4 out of bound 3
```

Arrays: Vectors and Matrices

- The size of an array can be obtained with the predefined functions rows, columns and size.

```
>> M = [ 1 4 7 5 6; 3,6,8 9 0]
    M = 1 4 7 5 6
        3 6 8 9 0
>> m = rows(M)
    m = 2
>> n = columns(M)
    n = 5
>> d = size(M)
    d = [ 2 5]
```

Arrays: Vectors and Matrices

- If the size of an array is known before its elements it is good practice to initialise it with 0's or 1's, with the predefined function zeros/1 and ones/1.

```
>> Z = zeros(3,4)
    M = 0 0 0 0
        0 0 0 0
        0 0 0 0
>> O = ones(2,3)
    O = 1 1 1
        1 1 1
>> Q = zeros(3)
    Q = 0 0 0
        0 0 0
        0 0 0
```

Arrays: Vectors and Matrices

- Arrays can also be specified by composition of simpler arrays, or by extending existing arrays with further elements.

```
>> A = [ 4 7 5];
    A = 1 4 7
>> B = [ 1, A, 6]
    B = 1 4 7 5 6
>> C = [ 3,6 8,9 0];
>> M = [ B; C; C]
    M = 1 4 7 5 6
        3 6 8 9 0
        3 6 8 9 0
>> D = [ 2; 3]
>> E = [ D; 4]
>> F = [ 9; 8; 7]
>> N = [E,F]
    M = 2 9
        3 8
        4 7
```

Arrays: Vectors and Matrices

- Arrays can also be obtained from other arrays, by projection of the relevant rows and columns.
- Contiguous indices may be specified by ranges `i:j`. Non-contiguous indices may be specified by means of vectors indicating the relevant indices.

```
>> M
 M = 1 2 3 4
     3 5 7 9
     2 4 6 8
>> N = M(1:2,2:3). % rows 1 to 2, columns 2 to 3
 N = 2 3
     5 7
>> P = M([1,3],[2,4]) % rows 1 and 3, columns 2 and 4
 P = 2 4
     4 8
>> Q = M(2:end, :) % rows from 2 to the END, ALL columns
 Q = 2 3 4
     5 7 9
```

Operations on Arrays

- Since matrices are the primitive datatype in MatLab, all operations with vectors and matrices are specified with the usual operators. Hence, when having the adequate dimensions and sizes, vectors and matrices can be
  - Summed, subtracted or multiplied
  - Inverted
  - Divided (multiplied by the inverted)
  - Compared (and other Boolean operations)
- Moreover, from two arrays with the same size, another array of the same size can be obtained by pointwise operations (operating the corresponding elements) of sum, subtraction, multiplication and division.
- Finally, usual arithmetic operations can be applied between a scalar and an array, thus distributing the operation between the scalar and each of the elements of the array.

Arrays: Vectors and Matrices

```
>> M = [ 1 2 3 ; 4 2 0];
>> D = M + M
 D = 2 4 6
     8 4 0
>> E = D + 4
 E = 5 6 7
     8 6 2
>> N = [ 1 2; 3 4]
>> P = N .* N
 P = 1 4
     9 16
>> Q = N * N
 Q =  7 10
     15 22
>> R = Q > 9
 R = 0 1
     1 1
>> S = rem(Q,5) == 0
 R = 2 0
     0 2
```

Predefined Array Functions

- Some aggregation functions are predefined on vectors that compute the sum, product, maximum and minimum of their elements.
- When applied to matrices, these functions return the results from applying the corresponding operations to each of the columns of the matrices.
- Hence to obtain the maximum element of a matrix, the maximum function must be applied twice!
- By combining these functions with Boolean functions the number of elements with certain properties can be counted.

Arrays: Vectors and Matrices

```
>> A = [ 7, 10]
>> B = [15, 22]
>> P = [A ; B]
>> I = P^-1
 I =  5.50 -2.50
     -3.75  1.75
>> P * I
 ans =  1.0000e+00   3.1086e-15
       -7.9936e-15   1.0000e+00
>> m1 = max(A)
 m1 = 10
>> m2 = min(P(2,:)
 m2 = 15
>> s1 = sum(P)
 s1 = [22 32]
>> s2 = sum(sum(P))
 s2 = 54
>> sum( P > 9)
 ans = 1 2
>> sum(sum(P>9)
 ans = 3
```

Scripts

- All instructions have been exemplified directly in the Octave console, one by one.
- One may be interested in repeating the very same operations time and again possibly with different values for the variables.
- For doing so, one may use special programs named **scripts**.
- Scripts are (usually small) text files containing all the instructions that can be specified in the console, one by one.
- Hence, rather then repeating all the instruction in the console, the user may only call the script, so that Octave runs it.
- Scripts are files with extension ".m". To be identified by Octave, they must be in the working directory.
- To call a script, i.e. to execute all the instructions written in it, all that is needed is to type their name in the console.

Arrays: Vectors and Matrices

test.m

```
A = [ 7, 10]
B = [15, 22]
P = [A ; B]
I = P^-1
U = P * I
m1 = max(A);
m2 = min(P(2,:)
p1 = prod(P)
p2 = prod(prod(P))
s1 = sum( P > 9)
S2 = sum(sum(P>9)
```

```
>> test
A =   7 10
B =  15 22
P =   7 10
     15 22
I =  5.50 -2.50
    -3.75  1.75
U = 1.0000e+00   3.1086e-15
   -7.9936e-15   1.0000e+00
m2 = 15
s1 = 105 220
s2 = 23100
s1 = 1 2
s2 = 3
>>
```