Functions; IF and FOR instructions

Pedro Barahona
DI/FCT/UNL
Métodos Computacionais
1st Semestre 2017/2018

Scripts

- In very abstract terms, we have defined a program as a sequence of instructions (implementing some algorithm) that take some data as input as produce an output.
- Moreover, we have seen that the angle between two vectors (of the same kind, both row or columns vectors) can be computed by the following sequence of instructions.

```
>> V1 = ...;
>> V2 = ...;
>> m1 = sqrt(V1*V1');
>> m2 = sqrt(V2*V2');
>> m12 = V1*V2';
>> angR = acos(m12 / (m1*m2));
>> andG = 180* angR / pi;
```

- Assuming that the assignment instructions V1 = …; and V2 = ...; correspond to the input of data the above set of instructions can be made as program, producing the angle angR as the output.

Scripts

- This program could be implemented as a script in MATLAB.

angle.m

```
% input data: V1 and V2
m1 = sqrt(V1*V1');
m2 = sqrt(V2*V2');
m12 = V1*V2';
angR = acos(m12 / (m1*m2));
andG = 180* angR / pi
```

```
>> V1 = ...;
>> V2 = ...;
>> m1 = sqrt(V1*V1');
>> m2 = sqrt(V2*V2');
>> m12 = V1*V2';
>> angR = acos(m12 / (m1*m2));
>> andG = 180* angR / pi;
```

- In this case, all the intermediate computation steps can be abstracted away from the user. To obtain the angle between two vectors, she only needs to enter the vectors, and invoke the script to obtain the result. For example:

```
>> V1 = [1, 1];
>> V2 = [-1 1];
>> angle
angG = 90
```

Scripts vs. Functions

- However, this approach to implement programs through scripts suffers from two major problems:
  - If the vectors have been assigned to other variables, say P and Q, the script requires that they are first assigned to variables V1 and V2.
    - This may of course be a problem, as the variables V1 and V2 might have been already assigned with other relevant information:
  - The variables used in the script (m1, m2 ,…) may have also been used to denote other relevant information to the program.
    - Running the script will replace such relevant information by the intermediate values computed during the script execution.
- In conclusion, implementing programs as scripts does not **separate** the computations inside the script from those outside the script, sharing the same variable space and so allowing undesirable interference.
- What is needed is a mechanism that clearly separates the different computations and this is achieved by using (user-defined) **functions**.

Functions

- Before any formalisation, we illustrate the use of a function in this example.

```
function angG = angle(V1, V2)
% this function computes the angle between
% vectors V1 and V2, both of the same kind
   m1  = sqrt(V1*V1');
   m2  = sqrt(V2*V2');
   m12 = V1*V2';
   angR = acos(m12 / (m1*m2));
   angG = 180* angR / pi;
end
```

- If during the computation the vectors of interest have been specified in variables **U1** and **U2**, all that is needed to compute their angle is to call

```
>> ang = angle(U1,U2)
ang = 90
```

with the guarantee that no variables in the rest of our programs would be affected, except of course, variable **ang** that is assigned the value of the angle.

Functions

- Thus, user-defined functions very much like pre-defined functions.
- They are called by invoking their name, together with their arguments (in terms of the variables used in the context of the call), and assigning the result to some other variable of our choice.
- Like with scripts, user-defined functions must be defined and made accessible.
  - They must be defined in a text file with the same name and extension ".m".
  - They are accessible, if the file is stored in the working directory.

angle.m

```
function angG = angle(V1, V2)
% this function computes the angle between
% vectors V1 and V2, both of the same kind
   m1  = sqrt(V1*V1');
   m2  = sqrt(V2*V2');
   m12 = V1*V2';
   angR = acos(m12 / (m1*m2));
   angG = 180* angR / pi;
end
```

Functions

- A function is composed of the following elements
  - The signature
  - The documentation
  - The body
  - The end statement

```
function angG = angle(V1, V2)
% this function computes the angle between
% vectors V1 and V2, both of the same kind
   m1  = sqrt(V1*V1');
   m2  = sqrt(V2*V2');
   m12 = V1*V2';
   angR = acos(m12 / (m1*m2));
   angG = 180* angR / pi;
end
```

Functions

- The signature is composed of
  - The key word **function**
  - The result (followed by a = sign)
  - The name of the function (the same as the name of the file containing it)
  - The parameters, within (round) brackets and separated by commas

```
function angG = angle(V1, V2);
```

- A function may return no result.
  - For example, a function may be used to write some text in the terminal and that is all it must do – no value is returned as a result.
- A function may have no parameters (is a constant), in which case, the parentheses may be ommitted.
  - For example we may want to use the number chi, i.e. the golden ratio number, **chi** = (1+sqrt(5))/2 and define it as a function making it available, as **pi** and **e** are available.

Functions

- The documentation of a function is a set of comments lines written either before or immediately after the signature.

```
function angG = angle(V1, V2)
% this function computes the angle between
% vectors V1 and V2, both of the same kind
   ...
end
```

- It should provide an explanation of what the function is supposed to do, as clearly and completely as possible.
- It is optional, but it is a good (mandatory…) practice to declare it
  - In particular, the help command returns the function documentation.

```
>> help angle
% this function computes the angle between
% vectors V1 and V2, both of the same kind
>>
```

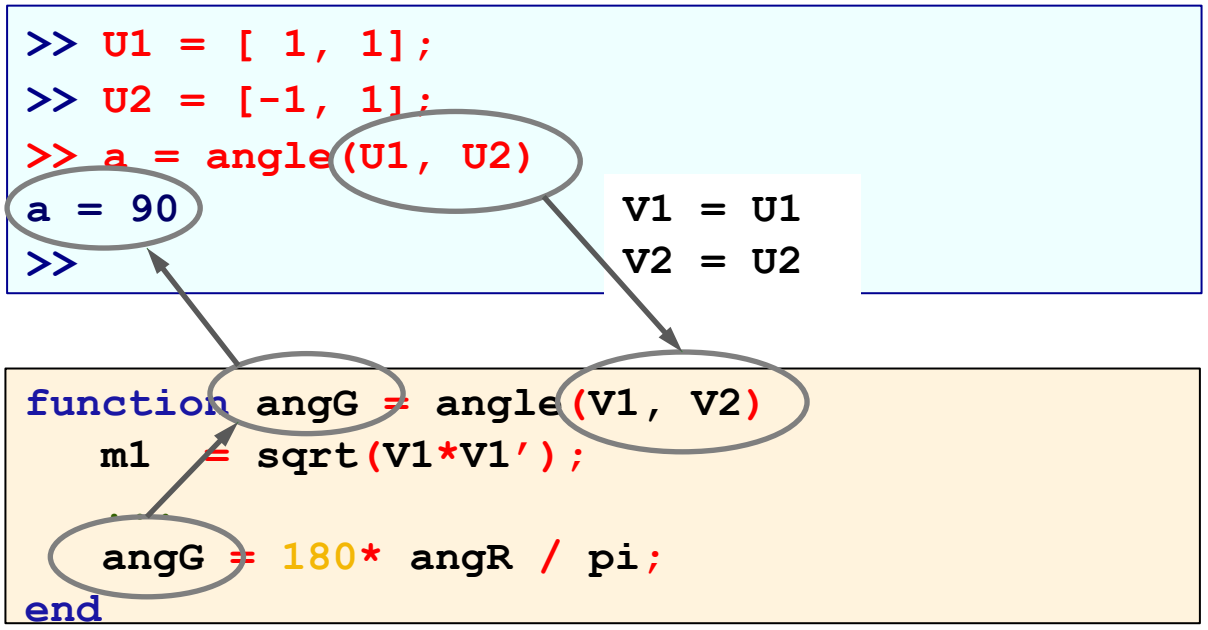
Functions

- The body of the function is the set of instructions that compute the result.
- It is thus necessary that the result declared in the signature is assigned in the body of the function (typically it is the last instruction).

```
function angG = angle(V1, V2)
% this function computes the angle between
% vectors V1 and V2, both of the same kind
   m1  = sqrt(V1*V1');
   m2  = sqrt(V2*V2');
   m12 = V1*V2';
   angR = acos(m12 / (m1*m2));
   angG = 180* angR / pi;
end
```

Functions

- The execution of a function can be explained as follows:
  1. The arguments of the function call are assigned to the parameters of the function signature;
  2. The body of the function is executed
  3. The result identified in the function signature, and assigned during the function execution is passed to the invoking program.



Functions

- The separation between the variables spaces of the function and the program that calls it is guaranteed,
  - The variables of the function body are different from the variables in the calling program, even if they have the same name!
  - In particular function internal variables are not seen from the calling program

```
function t = triple(a)
% this function computes the triple of the argument
   y = a + a
   t = y + a
end
```

```
>> y = 7, z = triplo(5), y, a
  y = 7
  y = 10
  t = 15
  z = 15
  y = 7
  error: 'a' undefined near line 1 column 3
```

Unitary Tests

- Before start discussing the instructions that can be used (namely in the body of a function) it is important to stress some important points.
- Functions should be coded in a clear way, documenting the instructions with comments (after the % sign), highlighting their purpose.
  - It is a known fact, that very obvious programs tend to become misterious, even for their authors, after some time (e.g. a month) without checking them.
- To enhance the understand ability of the programs, all functions should be documented with the documentation section in the beginning of their code.
- In the code of any non-trivial function there are calls to other functions. If something goes wrong it is very important to understand where the bugs occur.
- Hence the need for **Unitary Tests**: it is very important that every function is thoroughly tested in "isolation" (the unitary tests), before it is used in other functions.

Conditional Execution - IF

- The function **angle/2** that was used before as an example executes a sequence of assignement instructions, some of them calling pre-defined functions, like **sqrt/1** and **acos/1**). This is a very rare situation. In most programs/functions **the sequence of instructions depends on conditions of the data being used**.
- For specifying this conditional execution, all languages include an instruction: **IF**. Syntax may vary for different languages so here we will use the MATLAB syntax.
- In its simplest form this instruction conditions the execution of a THEN-BLOCK.

```
if <CONDITION>
   THEN BLOCK
end;
```

- Very often the instructions selects one of two sequence of instructions: either the THEN-BLOCK or the ELSE-BLOCK is executed

```
if <CONDITION>
   THEN-BLOCK
else
   ELSE-BLOCK
end;
```

Conditional Execution - IF

- We may illustrate the first case with a function to compute the absolute value of a number (in fact this function already exists pre-defined, as **abs/1**).

```
function a = absolute(x)
% this function computes the absolute
% value of its argument
   a = x;
   if x < 0
     a = -a;  % change the sign of a
   end
end
```

- A more natural specification of this function would use the else statement

```
function a = absolute(x)
% this function computes the absolute
% value of its argument
   if x < 0
     a = -x;
   else
     a = +x;
   end
end
```

Conditional Execution - IF

- A more complex example: Find the roots of a 2nd degree

```
function roots = equation_2(a, b, c)
% roots = equation_2(a, b, c)
% roots return the solutions of equation
% ax^2 + bx + c = 0 (assuming a != 0)
   d = b^2 - 4*a*c;
   if d < 0       % no solutions
      roots = [];  % an empty vector is returned
   else
      if d == 0     % one single solution
         roots = [-b/(2*a)];
      else          % two distinct solutions
         roots = [-b + sqrt(d) / /(2*a),
                  [-b - sqrt(d) / /(2*a)];
      end
   end
end
```

- IMPORTANT: Notice the indentation – it makes the code much more readable!

Conditional Execution - IF

- The previous example illustrates the “nesting” of if statements (if inside the if blocks).
- The code becomes more readable if one uses not a single ELSE-BLOCK but several ELSEIF-BLOCKS.

```
function roots = equation_2(a, b, c)
% roots = equation_2(a, b, c)
% roots return the solutions of equation
% ax^2 + bx + c = 0 (assuming a != 0)
   d = b^2 - 4*a*c;
   if d < 0       % no solutions
      roots = []; % an empty vector is returned
   elseif d == 0   % one single solution
      roots = [-b/(2*a)];
   else
      roots = [-b + sqrt(d) / /(2*a),
               [-b - sqrt(d) / /(2*a)];
   end
end
```

Iterative Execution - FOR

- In many cases it is necessary to repeat a block of instructions. There are several variants to specify such repetition, and the simplest one is with a FOR statement.
- In MATLAB syntax

```
for ITERATION-VAR = ITERATION-VECTOR
    FOR-BLOCK
end;
```

- This instructions specifies that the FOR-BLOCK
  - is executed as many times as there are elements in the ITERATION-VECTOR;
  - In each execution the ITERATION-VAR takes the value of the corresponding element of the vector
  - Note: The ITERATION-VAR is usually used in the FOR-BLOCK, although this is not necessary
- The next examples illustrate the use of the FOR instruction.

Ranges and Vectors

- Before that we note that in MATLAB a vector can be specified as a **range**, that are very often used in FOR statements

```
V = first : stop
```

so that the vector starts with element **first**, and subsequent elements differ by **1**, until the last element, that should not exceed the value of **stop**.

- More generally, with specification

```
V = first : step : stop
```

consecutive elements of the vector differ by the value of **step** (that can be any number, different from zero).

- Notice that in any case the vector can be empty.
  - In the first case, this happens when **stop < first**;
  - In the second case, an empty vector is obtained if
    - Step is positive and **stop < first**;
    - Step is negative and **stop > first**;

Ranges and Vectors

- Some examples illustrate the specification of vectors as ranges

```
>> V = 1:5
 V = 1 2 3 4 5
>> U = 1:2:8
 V = 1 3 5 7
>> X = 8:-2:0
 X = 8 6 4 2 0
>> Y = 8:-2.5:-3
 Y = 8.0 5.5 3.0 0.5 -2.0
>> A = 5:1
 A = [] 1x0
>> B = 5:2:1
 B = [] 1x0
>> C = 1:-2:5
 C = [] 1x0
```

Iterative Execution - FOR

- Back to the FOR statement. The following functions compute the same result from a vector passed as an argument.
- What do they compute?

```
function s = name_1 (V)
% ...
    s = 0;
    for v = V
        s = s + v;
    end
end
```

```
function s = name_2 (V)
% ...
    s = 0;
    for i = 1:length(V)
        s = s + V(i);
    end
end
```

Iterative Execution - FOR

- he previous functions use variable s as an **accumulator**. At each iteration the accumulator is updated to take into account the elements of the vector already considered.
- The update of the accumulator variable can be viewed in “debugging” mode, i.e. its update instruction does not end in “;”

```
>> Z =[ 2 6 1 7]
 Z = 2 6 1 7
>> x = name_1(Z)
 s = 0
 s = 2    % 0 + 2
 s = 8    % 2 + 6
 s = 9    % 8 + 1
 s = 17   % 9 + 7
 x = 17
```

```
function s = name_1 (V)
% ...
   s = 0
   for v = V
      s = s + v
   end
end
```

Iterative Execution - FOR

- The following example uses the same technique, but includes an if statement inside the for, so that only some elements produce changes to the accumulator variable.
- What does it compute?

```
function s = name_3 (V)
% ...
   s = -Inf;
   for v = V
      if  v > s
         s = v;
      end
   end
end
```

```
function s = name_4 (V)
% ...
   s = -Inf;
   for i = 1:length(V)
      s = max(V(i),s);
   end
end
```

- Note that this technique can be used
  - with any operation that is commutative and associative, as is the case of operations **sum, product, max** and **min**; and
  - The accumulator is initialized with the neutral element of the operation (**0** for **sum**, **1** for **product**, **-Inf** for **max** and **+Inf** for **min**)

Iterative Execution - FOR

- Again the behaviour of these functions can be “debugged”.

```
function s = name_3 (V)
% ...
   s = -Inf;
   for v = V
      if  v > s
         s = v;
      end
   end
end
```

```
function s = name_4 (V)
% ...
   s = -Inf;
   for i = 1:length(V)
      s = max(V(i),s);
   end
end
```

```
>> Z =[ 2 6 1 7]
 Z = 2 6 1 7
>> x = name_3(Z)
 s = -Inf
 s = 2
 s = 6
 s = 7
 x = 7
```

```
>> Z =[ 2 6 1 7]
 Z = 2 6 1 7
>> x = name_4(Z)
 s = -Inf
 s = 2
 s = 6
 s = 6
 s = 7
 x = 7
```

Iterative Execution - FOR

- The following example uses the same technique, but includes an if statement inside the for, so that only some elements produce changes to the accumulator variable.
- What do they compute?

```
function s = name_5 (V)
% ...
   s = +Inf;
   for v = V
      if  v < s
         s = v;
      end
   end
end
```

```
function s = name_6 (V)
% ...
   s = +Inf;
   for i = 1:length(V)
      if V(i) < s
         s = V(i);
      end
   end
end
```

Nested FORs

- When dealing with matrices it is usual to adopt two iterative variables to represent the indices of the rows and columns of the matrix.
- This is illustrated in the following example, taking a matrix as an argument.
- What does it compute?

```
function s = name_7 (M)
% ...
    s = 0;
    for i = 1:rows(M)
        for j = 1:columns(M)
            s = s + M(i,j);
        end
    end
end
```

Nested FORs

- Again the behaviour of this function may be debugged:

```
>> A =[ 2 6 3; 1 0 8]
 A = 2 6 3
     1 0 8
>> x = name_7(Z)
 s = 0
 s = 2    % + M(1,1)
 s = 8    % + M(1,2)
 s = 11   % + M(1,3)
 x = 12   % + M(2,1)
 x = 12   % + M(2,2)
 x = 20   % + M(2,3)
 s = 20
```

```
function s = name_7 (M)
% ...
   s = 0;
   for i = 1:rows(M)
      for j = 1:columns(M)
         s = s + M(i,j)
      end
   end
end
```

Nested FORs

- Actually the same result could be obtained by summing the elements of the matrix by columns:

```
>> A =[ 2 6 3; 1 0 8]
 A = 2 6 3
     1 0 8
>> x = name_7(Z)
 s = 0
 s = 2    % + M(1,1)
 s = 3    % + M(2,1)
 s = 9    % + M(1,2)
 x = 9    % + M(2,2)
 x = 12   % + M(1,3)
 x = 20   % + M(2,3)
 s = 20
```

```
function s = name_7 (M)
% ...
   s = 0;
   for j = 1:columns(M)
      for i = 1:rows (M)
         s = s + M(i,j)
      end
   end
end
```