Chapter 2

Standard methods to produce readable code

When a software is programmed, the layout of the code, its organisation, or its structure does not matter for the computer, because the code is linearly read by the interpreter or the compiler, and its disposition in the interface is not taken into account. Therefore, there is no difference to the computer between the following code:

```
function [res, index]=IsIncludeIn(NumberSearched, Matrix)

n=length(Matrix(:, 1, 1));
m=length(Matrix(1, :, 1));
p=length(Matrix(1, 1, :));

res=false;
index=[];
i=1;

while res==false && i<=n
    j=1;
    while res==false && j<=m
        k=1;
        while res==false && k<=p
            if Matrix(i,j,k)==NumberSearched
                res=true;
                index=[i j k];
            end
            k=k+1;
        end
        j=j+1;
    end
    i=i+1;
end
```

and this code:

```
function [w,g]=u(a,z);o=1;w=false;g=[];while w==false&&...
~(o>length(z(:,1,1)));s=1;while w==false&&~(s>length(z(1,:,1)));
r=1;while w==false&&~(r>length(z(1,1,:)));if z(o,s,r)==a;w=true;
g=[o s r];end;r=r+1;end;s=s+1;end;o=o+1;end;
```

The two programs execute exactly the same algorithm[1], and thus, both versions are acceptable by the computer. However, most pieces of software are developed by a team and they need to understand quickly the codes the other members programmed, which makes the first code presented much more valuable[2], because it takes only few seconds for a programmer to understand what the program does in the first code, whereas the second is completely unreadable. Moreover, the programs often need to be debugged, or be slightly modified during the developing process or the maintenance, which is almost impossible to do if the programs are coded like the second example. Therefore, programmers should always keep in mind that their programs have to be understandable for other people [14]. Based on a number of books and websites, this section tries to define a number of rules to make a code very clear and understandable by humans.

2.1 General Rules

2.1.1 Layout of the code

2.1.1.1 Rules 1.1: Indentation

Indentation [15] involves adding tabulation to the code in order to make its structure emphasised. Therefore, a good indentation is necessary to understand how the program works and what it does, and the programmer should constantly keep in mind that the code has to be well indented. Otherwise, a bad indentation generally makes the code difficult to be read and can even lead to errors or bugs [16], as it can be seen in the following example:

[1] Namely, the two programs look for a number in a three dimensional matrix, return true if the number is included, false otherwise. The indexes of the number are also returned.

[2] It can be nocticed that this code does not include any comment. Indeed, as it is seen in section 2.1.3.2, additionnal comments are required only if the code is not understandable itself, which is not the case of this program.

```
function res=IsEmpty(s)

if not(isa(s,'char'))
    disp('Please enter a string')                    end
    error('Exiting...');

res=length(s)==0;
end
```

The intention of the programmer clearly was to exit the programs if the input entered by the user is not a string, as the indentation suggests. But a deeper look in the code shows that the program will always quit, even if the input entered by the user is a string. These kinds of errors, due to the indentation, can be very difficult to detect, especially in languages such as C or C++ [16] where only brackets are used to specify the beginning and the end of a block. That leads to a first rule concerning the layout that the code should have:

Rule 1.1. *The code of every program should be indented so as to make its structure clearly highlighted.*

There is not one perfect model of indentation and this choice is up to the programmer as long as the code remains clear and organised. However, three different styles are advised in Matlab [17] and can be defined in the interface:

- Classic Style:

  This mode automatically indents every logical block like `for/end`, `if/elseif/else/end`, `while/end`, but no space is added inside the main function block or inside the eventual nested function block, as it can be seen in the example below:

```
function ClassicalExample
% Main function not indented

for i=1:100
    UseNestedFunction(i);
end


    function UseNestedFunction(i)
    % Nested function not indented

    disp(['This is the' num2str(i) 'th line']);

    end

end
```

- Indent Nested Functions Style:

  This mode automatically indents every logical block like `for/end`, `if/elseif/else/end`, `while/end`, and also nested functions, but not the main function, as it can be seen in the example below:

```
function NestedFunctionsExample
% Main function not indented

for i=1:100
    UseNestedFunction(i);
end


    function UseNestedFunction(i)
        % Nested function indented

        disp(['This is the ' num2str(i) 'th line']);

    end

end
```

- Indent All Functions Style:

  In this mode, all the logical blocks like `for/end`, `if/elseif/else/end`, `while/end` are indented, as well as the eventual nested functions and the main function, as it is shown in the example below:

```
function AllFunctionsExample
    % Main function indented

    for i=1:100
        UseNestedFunction(i);
    end


    function UseNestedFunction(i)
        % Nested function indented

        disp(['This is the ' num2str(i) 'th line']);

    end

end
```

There is no best style among these three possibilities and a choice has to be made by the programmer, depending on the style he thinks is the most understandable for others. Note that these options can be changed by going to:
Preferences → Editor/Debugger → Language

In case the code is not correctly indented, another option can be used with Matlab. Indeed, if the whole program is selected, and the user uses Ctrl+i, the whole program is automatically indented according to the style defined in Matlab.

2.1.1.2 Rules 1.2: Consistence

It has been seen in the previous section that several styles could be used for indentation in order to highlight the structure of the code and the choice of the used style does not really matter, as long as the code is easy to be read. However, the user has to keep being consistent when the code is written, which means the chosen style has always to be the same for the whole project, otherwise the code would probably be more complicated to be read [16], as the example below suggests:

```
function NonConsistentExample

    for i=1:100
        UseNestedFunction(i);
    end

    for i=101:200
            AnotherNestedFunction(i);
    end

    for i=201:300
    AThirdNestedFunction(i);
    end

    function UseNestedFunction(i)
        disp(['This is the ' num2str(i) 'th line']);
    end

function AnotherNestedFunction(i)
    disp(['This is the ' num2str(i) 'th line']);
end

    function AThirdNestedFunction(i)
    disp(['This is the ' num2str(i) 'th line']);
    end

end
```

In this example, the rule defined in section 2.1.1.1 has been respected: the indentation is used, but the problem is that the code does not always use the same style of indentation, which makes it unclear. That leads to another rule concerning the layout the code should have:

Rule 1.2. *The indentation style chosen by the user should be the same, for the whole developed project.*

This rule means also that people who are working as a team should agree on the indentation style before starting the project, in order to make the final code clearer, and therefore, to make the maintenance or the modifications easier to perform.

2.1.1.3 Rules 1.3: One Statement per Line

If there is more than one piece of information on one line of the code, it makes the code difficult to follow, because the reader does not know how many pieces of information there are to be read in one line and the logic in the program becomes harder to follow. Thus, using only one statement per line makes the code easier to be read and understood. That leads to another rule concerning the layout of the code: [14]

Rule 1.3. *Each line of the code should contain only one statement.*

2.1.1.4 Rules 1.4: Line Length

Sometimes, for instance, for a complicated logic formula, the line of code that includes this formula may be very long, and this makes the code more difficult to read because our eyes prefer reading narrow lines. It is thus generally advised to limit the length of the line around 80 characters [18][19], which leads to another rule concerning the layout of the code:

Rule 1.4. *The number of characters in one line of code should not exceed more than 80 characters.*

Note that Matlab has a right-side text limit indicator, by default set to 75 characters, that appears as a gray line on the right side of the console and gives a good indication of the readability of the lines of code.

In order to avoid writing too many characters in one line, it is possible in Matlab to keep writing a statement in several lines by using the ellipsis (...), as it can be seen in the example below:

```
1  function res=IsLeap(Year)
2
3  if (mod(Year,4)==0 && mod(Year,100)~=0) || ...
4      (mod(Year,4)==0 && mod(Year,400)==0)
5
6      res=true;
7  else
8      res=false;
9  end
10
11 end
```

The ellipsis (...) used in this program allow the statement in line 3 to continue on line 4 and thus, makes the program easier to read.

2.1.1.5 Rules 1.5: Code Grouping

As an article would present an idea with one paragraph, it is also advised to split the code into "paragraphs" to make the code clearer and more understandable [16][18]. If a task requires several lines of code, separating the different tasks, by adding space between them, emphasises the succession of tasks done by the program, as it can be seen in the following example:

```
1  function PlotSphere(R, Xc, Yc, Zc)
2
3  % Creation of a sphere of radius 1
4  [X, Y, Z]=sphere();
5
6  % Increasing the radius of the sphere
7  X = R * X;
8  Y = R * Y;
9  Z = R * Z;
10
11 % Translating the sphere
12 X = X - Xc;
13 Y = Y - Yc;
14 Z = Z - Zc;
15
16 %Plotting the sphere
17 surf(X, Y, Z);
18 axis equal;
19
20 end
```

Thanks to the groups created, it is very easy to understand that the program firstly creates a sphere of radius 1, then increases the radius of the sphere, then translates the sphere to the center input by the user, and finally plots the sphere. That leads to another rule for the layout of the code:

Rule 1.5. *The different lines of code in a program should be regrouped, each group representing a task or an idea.*

2.1.2 Names

2.1.2.1 Rules 2.1: Names of the Variables

The choice of the names of the variables is maybe one of the most important things to make a code understandable, because a right and reasonable name for a variable tells a lot about its function, its utility, and its aim, and therefore, gives a lot of pieces of information to the reader about the program. That is why a good name for a variable is generally more helpful and more interesting than a comment [15][16][19].

Let the example provided in the introduction of the current section be taken again, where the variables are randomly named:

```
function [w, g]=u(a, z)

b=length(z(:, 1, 1));
e=length(z(1, :, 1));
x=length(z(1, 1, :));

w=false;
g=[];
o=1;

while w==false && o<=b
    s=1;
    while w==false && s<=e
        r=1;
        while w==false && r<=x
            if z(o,s,r)==a
                w=true;
                g=[o s r];
            end
            r=r+1;
        end
        s=s+1;
    end
    o=o+1;
end
```

This code is in compliance with the rules established for the layout, but it is still difficult to understand what the program does, because the name of the variable does not give any information about their goals or their structures. The following example shows the same program with names carefully chosen, which gives a lot of details on the aim of the program:

```
function [res, index]=IsIncludeIn(NumberSearched, Matrix)

n=length(Matrix(:, 1, 1));
m=length(Matrix(1, :, 1));
p=length(Matrix(1, 1, :));

res=false;
index=[];
i=1;

while res==false && i<=n
    j=1;
    while res==false && j<=m
        k=1;
        while res==false && k<=p
            if Matrix(i,j,k)==NumberSearched
                res=true;
                index=[i j k];
            end
            k=k+1;
        end
        j=j+1;
    end
    i=i+1;
end
```

With the right names, it is almost immediately clear that the program tries to find the number `NumberSearched` in the three dimensional matrix `Matrix`. That thus leads to a fist rule concerning the names of variables:

Rule 2.1. *The variables should have names that provide details about their roles, their structures and their aims in the program.*

2.1.2.2 Rules 2.2: Name Scheme

The name of a variable or a function needs also to be easily read, and if there are several words in this name, the beginning of the different words must be emphasised, in order them to be simply read. Two options [19] are generally used:

- camelCase The first letter of each word is a capital letter: `MyWonderfulVariable`
- underscores

  The different words in the name of the variables are separated by an underscore: `my_wonderful_variable`

That leads to another rule concerning the name of the variables or functions:

Rule 2.2. *The different words included in the names of the variables of a program should be emphasised so as to be easily read, and one of the options among camelCase style or underscores style is recommended.*

2.1.2.3 Rules 2.3: Magic Number

A magic number is a number that appears directly in the source code without any explanations and that comes out of nowhere, like the number 18 in the example below:

```
function [res,YearsRemaining]=IsAllowedToDrink(Age)

if age<18
    res=false;
    YearsRemaining=18-Age;
else
    res=true;
    YearsRemaining=0;
end
```

This program tells the user if he is allowed to drink alcohol and if not, how many years he has to wait. The problem is that the role of the number 18 in the program does not clearly appear. It is more convenient to use a variable for this magic number [15][19], as the following example shows:

```
function [res,YearsRemaining]=IsAllowedToDrink(Age)

AgeLimit=18;

if age<AgeLimit
    res=false;
    YearsRemaining=AgeLimit-Age;
else
    res=true;
    YearsRemaining=0;
end
```

Because the magic number is now included in a well-named name, the reader can immediately understand what its role is in the program, which thus becomes easier to understand. Moreover, if the law changes and the age limit to buy a drink is set to 21 years, the program requires one minor change, whereas in the other case it would have required several changes, which could have produced bugs, because the number 18 is duplicated[3] and the programmer that makes an update to the program can forget to change one of these duplications. Therefore, including magic numbers into variables makes the code easier to understand and prevents the program from any bugs when it is updated. That leads to another rule concerning the names of variables:

Rule 2.3. *Magic numbers should not appear directly in the code but should be included in variables with names that explain the roles of these numbers in the program.*

2.1.3 Explanations on the code

2.1.3.1 Rules 3.1: Documentation of the code

One thing that makes the code easier to understand is also to add a brief description of what the program does at the beginning of the code, which is called documentation. Thus, thanks to the documentation, the reader do no longer have to understand what the code does, but only how it does it, and therefore a lot of time can be saved during the reading [18]. That thus leads to a fist rule concerning the explanations on the code:

Rule 3.1. *Each code should start by a documentation that at least contains:*

- *An explanation of what the code does*
- *An explanation of the inputs*
- *An explanation of the outputs*

The following code shows an example of the style that can be used but once again, there is no best style as long as the documentation appears clear and readable.

```
1 function res=IsLeap(y)
2
3 % IsLeap.m : Test if a year is leap or not
4 %
5 % PROTOTYPE :
6 %   res=IsLeap(y)
7 %
8 % DESCRIPTION
```

[3] See section 2.1.4.1.

```
9  %   Test a year is leap or not.
10 %
11 %   Note : Leap if : -it is divisible by 4
12 %                    -it is divisible by 400
13 %          Not leap if: - it is not divisible by 4
14 %                       - it is divisible by 100 but not by 400
15 %
16 % INPUT
17 %   y: a year (integer)
18 %
19 % OUTPUT
20 %   res: a boolean: true if y is leap, false if it is not
21
22 ...
```

In this documentation style are included:

- The name of the file and one descriptive sentence about the code
- The line of command to enter to execute the function
- A more complete description of what the code does
- The signification of the inputs
- The signification of the outputs

The other advantage of a documentation is that for Matlab and other interfaces, it can be also accessed as a help, as it can be seen in Figure 2.1:

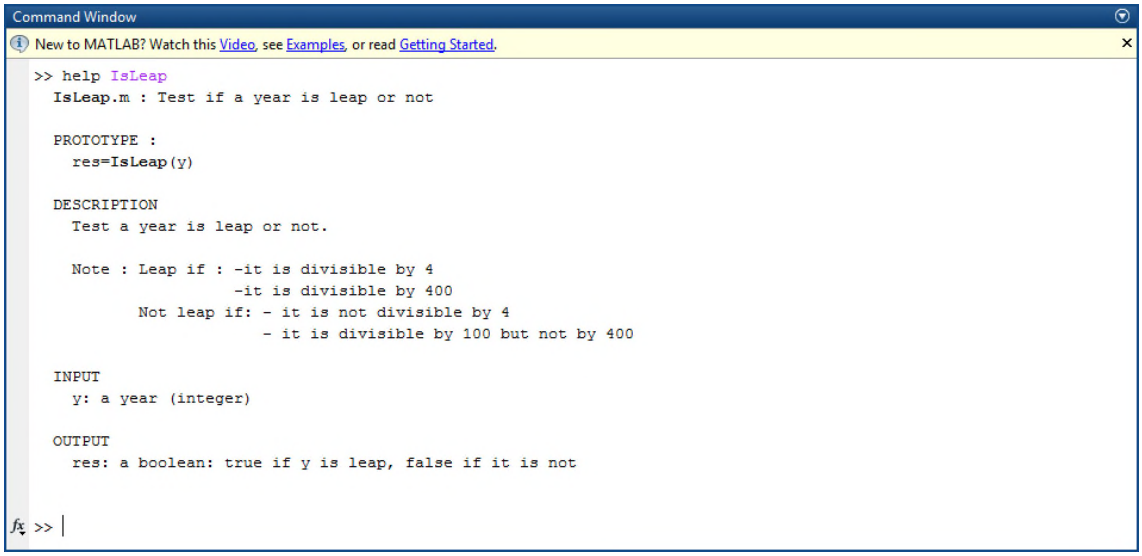

Figure 2.1: Documentation example in Matlab

2.1.3.2 Rules 3.2: Comments on the code

Codes need generally to be commented, in order to help the reader when the algorithms used in the program can be complex and difficult to understand. Therefore comments are supposed to add value to the code and make the reading easier. However, if a program is badly commented, it can have the opposite effect: the code can be harder to follow and thus difficult to read [16]. This section tries to give some keys about comments to make sure the readability of the code is not lowered because of the comments.

- Avoid obvious comments:

```
function [res,YearsRemaining]=IsAllowedToDrink(Age)

AgeLimit=18; % set the age limit to 18


if age<AgeLimit % if the age is lower than the age limit
    res=false; % the result is false
    YearsRemaining=AgeLimit-Age; % the years remaining are
                                 % the difference between
                                 % the age limit and the age

else % if the age is greater than the age limit
    res=true; % the result is true
    YearsRemaining=0; % the year remaining is 0
end
```

  A good code should normally be clear enough to be understandable, and obvious comments will just repeat the code a second time, which just makes the code more difficult to read, as a newspaper would be if every sentence are repeated twice. Only complex algorithms generally need to be commented. That is why, before writing a comment, it has to be ensured that it provides value to the code [15][16].

- Rewrite the code instead of commenting:

  If a part of the code seems unclear and the programmer thinks it should be explained, it is generally because the code is not clear or the programmer does not really understand what he wrote, and in that case, that means there is probably a better way to write the code, and rewriting the code generally clarifies it more than adding comments that can be hard to understand [16].

- Do not contradict the code:

  The comments generally agree with what the program does, but it has to be kept in mind that a program always evolves, or is transformed to remove any

remaining bugs, and then the code and the comments can become contradictory, which can easily confuse the reader. By checking the comments each time the program is updated, this problem should be avoided [15].

- No commented code in the program:

  It is sometimes easier to comment a part of a code to debug it and to write a new version below, but the commented code should be removed because it can be very confusing for the reader [14].

- A line of comment before a block of statements is enough:

  According to rule 1.5[4], the statements in the code should be regrouped where each group represents a task or an idea. When this idea or this task is a little bit complex, one line of comment before each block does not distort the layout of the program and makes the code even easier to understand. [18]

```
function PlotSphere(R, Xc, Yc, Zc)

[X, Y, Z]=sphere();

% Increasing the radius of the sphere
X = R * X;
Y = R * Y;
Z = R * Z;

% Translating the sphere
X = X - Xc;
Y = Y - Yc;
Z = Z - Zc;

surf(X, Y, Z);
axis equal;

end
```

With this layout, it is very easy to understand what the program does. To be more precise, it has been pointed out by Capers Jones [14] that one comment for ten statements were probably the best compromise between comments and clarity.

- Indentation

  The fact that comments have to match the indentation style has not to be forgotten, because, if the comments are not indented in the same way as the rest of the code, the structure of the code would not appear any more [14].

This leads to another concerning the explanations on the code:

[4] See section 2.1.1.5.

Rule 3.2. *The parts of the code* ***that need to be explained*** *should clearly be commented, and comments should be written with a corresponding indentation and only when they add value to the code.*

2.1.4 Structuring the code

2.1.4.1 Rules 4.1: DRY Principle

The DRY (Don't Repeat Yourself) principle, also known as the DIE (Duplication Is Evil) principle, has been formulated by Andy Hunt and Dave Thomas in the book *The Pragmatic Programmer* [18] and is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system". This is probably one of the most important rules in this chapter, but also maybe one of the most difficult to apply, because programmers can duplicate codes, even without knowing it.

Duplication is often used by programmers because it is easier to copy and paste a code instead of creating a function that takes the slight differences between the codes as inputs. The result is that maintenances or little changes can become horrible because the people that perform the maintenances have to know where all the duplicated codes are and if they forget just one, it can generate bugs in the software. That is why it is always better to create nested functions rather than duplicating codes. With this style of programming, if there is an update of the software that has to be made, changes will be done only in one location, which saves a lot of time and allows the system not to have bugs due to the maintenance [15][16]. That leads to a first rule concerning the structure that the code should have:

Rule 4.1. ***Never ever duplicate code sections*** *when another choice can be made and choose always to factor them into a common function when the slight differences are expressed thanks to the inputs of this function.*

The code below shows a simple example of duplication:

```
function [res,YearsRemaining]=IsAllowedToDrink(Age)

if age<18
    res=false;
    YearsRemaining=18-Age;
else
    res=true;
    YearsRemaining=0;
end
```

The aim of this code is to know if the user can buy alcohol, and if he cannot, the code provides also the number of years he has to wait before being allowed to

buy it. The age limit to buy alcohol is set to 18, according to the law. In this code, the programmer duplicated this number because he wrote it twice in the code. Now, imagine that the law changes and the age limit is now set to 19 years. The programmer, updating quickly his program, make this change:

```
function [res,YearsRemaining]=IsAllowedToDrink(Age)

if age<19
    res=false;
    YearsRemaining=18-Age;
else
    res=true;
    YearsRemaining=0;
end
```

The new program seems to work correctly, because it says that a person cannot buy alcohol if he his younger than 19 years old, but the program does not output the correct number of years the user has to wait before buying alcohol anymore. If the event of a user that is incapable of calculating this number[5], the only way he can find this bug is to enter the number 18, because in this case, the program outputs that the user is not allowed to buy alcohol and he has to wait 0 years before being allowed to buy it.

If the programmer had not made the error of duplicate this number and had stored it in a variable like in the following example, when he would have changed the age limit to 19, there would have been no bug in this program:

```
function [res,YearsRemaining]=IsAllowedToDrink(Age)

AgeLimit=19;

if age<AgeLimit
    res=false;
    YearsRemaining=AgeLimit-Age;
else
    res=true;
    YearsRemaining=0;
end
```

This was an example for just one number duplicated once, whereas programmer can duplicate hundreds of lines of code several times in a same program. A lot of bugs in maintenance can thus come for these duplications, that can be very

[5]This is clearly not going to happen in this example, but there can be similar cases for very complex calculations.

difficult to detect and also difficult to solve. Therefore, factorisations of the codes into functions are strongly recommended rather than duplicating these codes, and using these functions, if they are well-named and well-documented[6], increases the readability of the code.

2.1.4.2 Rules 4.2: Deep Nesting

Deep nesting is generally hard to follow for the user, and it is better to reduce it, especially for the logic [18], which leads to another concerning the structure the code should have:

Rule 4.2. *Deep nesting should be avoided as much as the programmer can, especially for logic expressions, because there is generally another way to program that reduces the deep of nesting.*

As an example, the following code:

```
function res=IsLeap(Year)

if mod(Year,4)==0
    if mod(Year,100)==0
        if mod(Year,400)==0
            res=true;
        else
            res=false;
        end
    else
        res=true;
    end
else
    res=false;
end
```

can be reduced to:

[6]Generally, it is easy to access the documentation with the interface used, for example, in Matlab, clicking on the name of the function and then pressing the F1 key displays the documentation of the function.

```
1  function res=IsLeap(Year)
2
3  if (mod(Year,4)==0 && mod(Year,100)~=0) || ...
4      (mod(Year,4)==0 && mod(Year,400)==0)
5
6      res=true;
7  else
8      res=false;
9  end
10
11 end
```

which makes it much easier to read and understand.

2.1.4.3 Rules 4.3: Dead Code

A dead code is a part of the code that is never used in the program and that can appear in the code because of carelessness from the programmer or updates made to the code [16], as the following example suggests:

```
1  function Mod=Module(X, Y)
2
3  z=X+Y*1i;
4  zconj=X-Y*1i;
5
6  Mod=sqrt(X^2 + Y^2);
7
8  end
```

In this code, the programmer probably wanted to use the formula $|z| = z\overline{z}$ to calculate the module of the vector $\begin{bmatrix} X \\ Y \end{bmatrix}$, but he finally chose the formula $\sqrt{X^2+Y^2}$ and the complex numbers `z` and `zconj` are not used in this program, which makes the lines 3 and 4 completely useless.

The dead code is not used in the program, which make the code more difficult to read and understand. It also adds additional and useless calculations that increase the time of execution of the program. Note that Matlab automatically identifies dead code in the program and gives a warning to the programmer. That leads to another concerning the structure the code should have:

Rule 4.3. *If there is a dead code in the program, it should be deleted.*

2.1.4.4 Rules 4.4: Compacting Logic

There are generally possibilities to reduce conditional statements in just one line, which makes the code much easier to read, to follow and to understand, as it becomes much more compact [16]. As an example, the following code:

```
function res=IsEqualTo3Or5Or7(n)

if n==3
    res=true;
else
    if n==5
        res=true;
    else
        if n==7
            res=true;
        else
            res=false;
        end
    end
end

end
```

can be compacted in:

```
function res=IsEqualTo3Or5Or7(n)

res=(n==3 || n==5 || n==7);

end
```

which is much easier to read and understand. This example is very simple, but similar cases are generally present in a lot of programs. That leads to another concerning the structure the code should have:

Rule 4.4. *The logic should be compacted as much as possible.*

2.1.4.5 Rules 4.5: Negation in Logic

Generally, negations in logic formulas are less obvious to understand than affirmations, and the positive form should always be preferred, as it is illustrated in the following code:

```
1  function res=SearchElement(Table, Element)
2
3  n=length(Table);
4  res=false;
5  i=1;
6
7  while res==false && ~(i>n)
8      res=(Table(i)==Element);
9      i=i+1;
10 end
11
12 end
```

When line 7 is read, it is not immediately clear that the conditional statement requires to have `i` smaller than `n`. Turning this statement into:

```
1  while res==false && (i<=n)
```

makes the reading easier to understand. That leads to another concerning the structure the code should have:

Rule 4.5. *Negations in logic should be avoided when it is possible.*

2.1.4.6 Rules 4.6: Parenthesis in Logic

Even if, in Matlab, the logical operator `&&` has a higher precedence than the operator `||`[7], it is always recommended to use parenthesis, because it makes the logical statement much clearer and therefore much easier to understand [15]. For instance, the following code:

```
1  function res=Is12Or21(Table)
2
3  if Table(1)==1 && Table(2)==2 || Table(1)==2 && Table(2)==1
4      res=true;
5  else
6      res=false;
7  end
```

gives the same result as this one:

[7] There are precedences for all operators that can be used in Matlab, including `<`, `<=`, `==`, `!=`, `>=`, `>`, `=`, ...

```
function res=Is12Or21(Table)

if (Table(1)==1 && Table(2)==2) || (Table(1)==2 && Table(2)==1)
    res=true;
else
    res=false;
end
```

However, the addition of parenthesis in the second code clarifies the logic, and thus, does not confuse the reader. That leads to another concerning the structure the code should have:

Rule 4.6. *Logical statements should always include parenthesis.*

2.1.4.7 Rules 4.7: Idioms

Idioms are natural ways to express something and, as they are included in speaking languages, they are also included in programming languages [15]. These following codes are all doing the same thing and however, only the last one has a correct writing:

```
n=100;
Table=zeros(1,n);

for i=n:(-1):1
    Table(i)=i;
end
```

```
n=100;
Table=zeros(1,n);

i=11;
while i<=n
    Table(i)=i;
    i=i+1;
end
```

```
n=100;
Table=zeros(1,n);

for i=1:n
    Table(i)=i;
end
```

The last version is the only acceptable one because it corresponds to the common language used by programmers and, therefore, is the most readable, and also because it is the most obvious one: each cell of `Table` are read, from the first to the last one. That leads to another concerning the structure the code should have:

Rule 4.7. *Idioms should be used as much as possible.*

2.1.4.8 Rules 4.8: Folder Structure

Software development can include the creation of hundred, even thousands different functions and, if they are not properly organised, it can be very difficult to understand the structure of the software. Using a folders tree structure where the different functions are regrouped by tasks helps to understand how the software works and gives an easy access to all the created functions [18], as it can be seen in Figure 2.2.

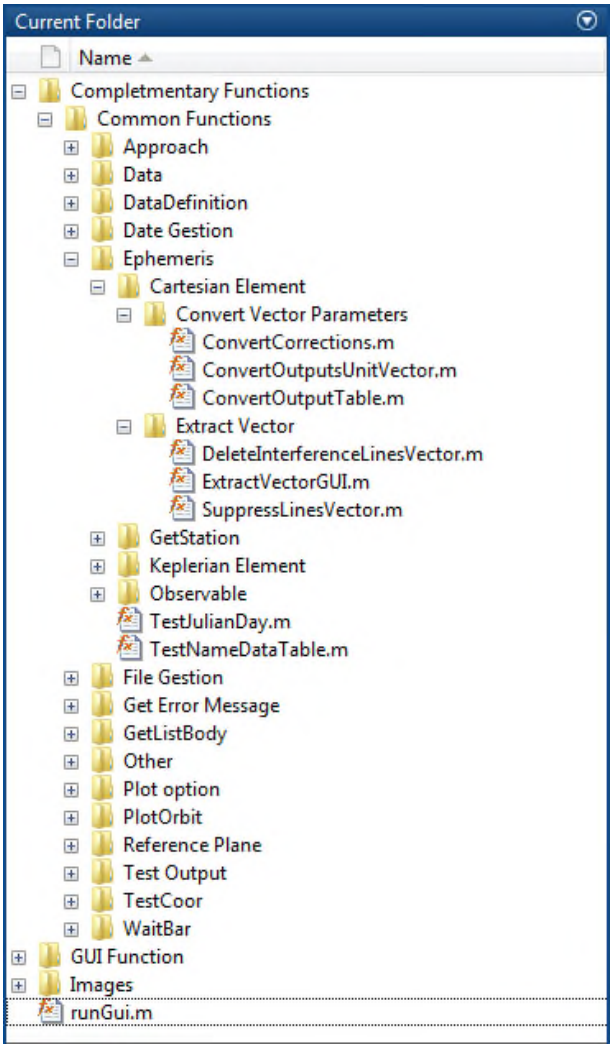

Figure 2.2: Example of folder tree structure

That leads to another concerning the structure the code should have:

Rule 4.8. *The functions should be organised in a folders tree structure where the functions that have a common objective are regrouped in a same folder.*

2.2 Rules Overview

The following table summarises the different rules stated in this section:

Rule 1.1	*The code of every program should be indented so as to make its structure clearly highlighted.*
Rule 1.2	*The indentation style chosen by the user should be the same, for the whole developed project.*
Rule 1.3	*Each line of the code should contain only one statement.*
Rule 1.4	*The number of characters in one line of code should not exceed more than 80 characters.*
Rule 1.5	*The different lines of code in a program should be regrouped, each group representing a task or an idea.*
Rule 2.1	*The variables should have names that provide details about their roles, their structures and their aims in the program.*
Rule 2.2	*The different words included in the names of the variables of a program should be emphasised so as to be quickly read, and one of the options among camelCase style or underscores style is recommended.*
Rule 2.3	*Magic numbers should not appear directly in the code but should be included in variables with names that explain the roles of these numbers in the program.*
Rule 3.1	*Each code should start by a documentation that at least contains an explanation of what the code does, an explanation on the inputs, an explanation on the outputs.*
Rule 3.2	*The parts of the code* ***that need to be explained*** *should clearly be commented, and comments should be written with a corresponding indentation and only when they add value to the code.*
Rule 4.1	***Never ever duplicate code sections*** *when another choice can be made and choose always to factor them into a common function when the slight differences are expressed thanks to the inputs of this function.*
Rule 4.2	*Deep nesting should be avoided as much as the programmer can, especially for logic expressions, because there is generally another way to program that reduces the deep of nesting.*
Rule 4.3	*If there is a dead code in the program, it should be deleted.*
Rule 4.4	*The logic should be compacted as much as possible.*
Rule 4.5	*Negations in logic should be avoided when it is possible.*
Rule 4.6	*Logical statements should always include parenthesis.*
Rule 4.7	*Idioms should be used as much as possible.*
Rule 4.8	*The functions should be organised in a folders tree structure where the functions that have a common objective are regrouped in a same folder.*