Copyright A.S.M.E. (American Society of Mechanical Engineers)

1. BASIC CONCEPTS OF PROGRAMMING

1.1 Algorithms

The solution of a problem can be developed by means of a mathematical model. This model can be represented as a set of instructions that are realized by an algorithm. Each instruction has a precise meaning and is executed a finite number of times.

1.2 Pseudo language

The pseudo language or pseudo code is a way to describe sequentially the logical procedure that an algorithm must realize. It is also the previous step to the codification or implementation of the algorithm in any programming language. Its use allows the planning of the program, that is to say, the programmer is only interested in the logic and the control structures, not in the syntactic rules of a specific language. This facilitates the correction of possible logical errors that the algorithm may contain.

```
(1)function Find maximum value
(2)
(3)          {Let M = array of real numbers }
(4)          maximum = value of the first position in the array M;
(5)
(6)          for (each position i in array M)
(7)          {
(8)                 if (value i in array M is greater than value stored in maximum)
(9)                 {
(10)                       update the value of maximum;
(11)                }
(12)         }
(13)
(14)         {maximum = maximum value in M}
```

Figure 1. Pseudo code structure

In Figure 1 some important characteristics of a pseudo code are observed. The first of them is that small letters in bold are used for the key words of programming languages. The conditions of flow control (**if, for, while**) are used in the propositions of the pseudo language. The conditional expressions as the one expressed in Figure 1 line (8), may be informal propositions, instead of conditional expressions of a programming language. Notice that the assignment in the Figure 1 line (4) uses an informal expression to the right, and that the cycle **for** of Figure 1 line (6) closes the repetition of the set of instructions under its domain. The domain is a group of instructions that opens and closes the pair of characters "{" and "}" respectively.

1.2.1 Assertion (Pre, Post, Inv)

The status of a program can be described by using propositions located in any place of interest. This type of proposition is called an **assertion**. In the design and interpretation of the different parts of the pseudo code, the assertions are between brackets "{ }". The assertion describes the status of the variables of the program when the execution passes by the point where it is located. Therefore the assertion is **NOT** an executable instruction, but only a description of a status of the program. In conclusion an assertion doesn't have any effect on the execution of the program.

By definition, when describing the state of the execution of the program (with all the possible cases) an assertion is never false. For example, the assertion in Figure 2 indicates that when the execution of the program passes by it, the variable *Interrupter 1* is True and *Interrupter 2* is False.

$$\{\ (\mathit{Interruptor1} == \text{True}\ ) \wedge (\mathit{Interruptor2} == \text{False})\ \}$$

Figure 2. Representation of an assertion

1.2.2 Precondition and postcondition

When one wants to specify the operation of a program, its initial and final status should be described so that its correct execution can be verified. To describe the initial status of the program, an assertion called **precondition** is used and to describe the final status of the program another assertion called **postcondition** is used as well. Retaking the pseudo code from Figure 1, it is established that the precondition and the postcondition are defined as they are shown in Figure 3.

{pre: ***M*** = array of real numbers}

{post: *Maximum* = maximum value in ***M***}

Figure 3. Precondition and postcondition

It is recommended to specify the precondition at the beginning and the postcondition at the end of the code. They should be fulfilled every time that the program is executed

1.3 Executable instructions

1.3.1 Assignment, verification or comparison

The use of variables allows the simulation of a process, task or problem in such a way that the information stored in them can be manipulated according to the user's necessity. An assignment example is observed in Table 1.

Table 1. Graphic Representation of an assignment

a = 1;	Arithmetic assignation
color = 'red';	Character string assignation

The computer executes each assignment sentence in two stages. In the first one, the value of the expression that is written to the right side of the assignment operator is calculated. In the second stage, the value is stored in the variable whose name is written to the left of the assignment operator.

In the assertions that evaluate the content of variables in a program the relationship or comparison operators shown in Table 2 are used

Table 2. Comparison symbols

MATLAB	C/C++	MEANING
>	>	Greater than

<	<	Lower than
= =	= =	Equal to
~ =	! =	Different of
<=	<=	Lower or equal to
>=	>=	Greater or equal to

1.3.2 Selection and decision making

It is necessary to incorporate decision structures, so that an algorithm can follow different execution routes. A decision instruction evaluates a condition and in function of the result obtained, the execution branches. The most used structure in programming languages is **if** and **else**. Table 3 illustrates the general form of this instruction and an example of its usage. Generic illustration of conditional instructions. Program that calculates the absolute value of a number by means of decision structures

Table 3. Illustration of Conditionals

if (condition1)
{
 group of commands 1;
}
else if (condition2)
{
 group of commands 2;
}
else if (condition3)
{
 group of commands 3;
}
else
{
 group of commands 4;
{
	{pre: N is a real number}
	
	if ($N > 0$)
	{
	 $ABS = N$
	}
	else if ($N < 0$)
	{
	 $ABS = - N$
	}
	else
	{
	 $ABS = 0$
	}
	
	{post : $ABS = \|N\|$ }

Generic illustration of conditional instructions	*Program that calculates the absolute value of a number by means of decision structures*

The conditions are verified one by one. If the condition is satisfied, the command block in its interior is executed. If it is not satisfied, the other conditions will be verified. An **else** should always be placed at the end to include the pathological or not foreseen cases.

1.4 Iterative commands or cycles

The iterative commands allow the repeated execution of one or several commands, whenever a condition initially established is satisfied. The iterative instructions are also known as **cycles**. Every cycle should be characterized by containing:

1. Preparatory instructions.
2. Condition of authorization for execution of the cycle body.
3. Invariant: an assertion that is true at the beginning of each iteration of the cycle.
4. Cycle body.

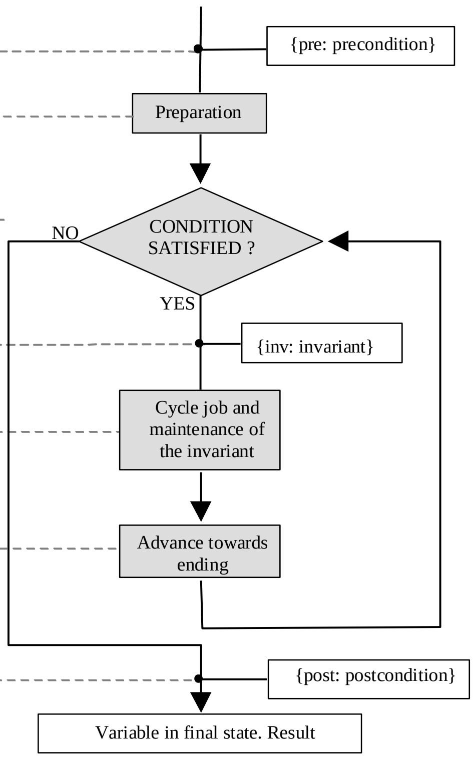
   4.1 Job of the cycle and maintenance of the invariant.

   4.2 Advancing towards cycle ending.

The execution of an iterative command is authorized by a condition that must be evaluated as true. If the condition is true, the body of the cycle is executed. The execution of the iterative command ends at the moment in which the condition becomes false.

Table 4. Structure of an iterative command

Program that finds the maximum value of an array of numbers	**Diagram of the components of a cycle**

Function Find maximum value
{
{ pre: ***M***[1.. *N*]= array of real numbers}
i= 1;
maximum=***M***[1];
while (*i* **<= N)**
{
{inv: maximum = max $_{j=1..i}$ (***M****[j]*)}
if (***M****[i]*≥ *maximum*)
{
change the value of *maximum*
}
i = *i* +1;
}
{Post: maximum = max $_{j=1..N}$ (M[j])}
}
	 {pre: precondition}
	Preparation
	CONDITION SATISFIED ?
	NO
	YES
	{inv: invariant}
	Cycle job and maintenance of the invariant
	Advance towards ending
	{post: postcondition}
	Variable in final state. Result

In order to specify the behavior of an iterative command, an assertion or statement (it <u>is not</u> an executable instruction) is identified. This assertion is called **invariant** and represents the status of the execution exactly after the Boolean condition controlling the iterations, and holds invariant and true each time the cycle body is going to be executed (see Table 4). Therefore, the invariant determines (a) the preparation for the loop, (b) the *hard-work* part of the loop, and (c) the advancement towards termination. In addition, the logical equation: "invariant AND not (Boolean condition for the loop)" must coincide with the Post-condition. Because its massive influence in all other instructions of an iterative instruction, the invariant is the single-most important logical predicate to be identified when writing such instructions.

One of the most common iterative commands is **for** (Figure 4). Its operation results from the previous knowledge of the number of iterations (N) required to execute the block of commands. The iterative cycle stops when the execution reaches this predetermined number of iterations. Inside the area of commands the variable "*i"*, which carries the value of the number of iterations, should **NOT** be manipulated, because the cycle **for** automatically updates it in each cycle.

```
for (i ≤ N)
{
    {inv: (invariant)}
    commands
}
```

Figure 4. Iterative structure **for**

While, is another iterative command (Figure 5) whose authorization to be executed is controlled <u>explicitly</u> in its preparation and within the commands in its domain. In these commands, when the execution has reached its objective, it so happens that the condition is not fulfilled anymore and therefore the cycle **while** ends. Then the following instruction to the cycle **while** is executed.

```
while (condition)
{
    {inv: (invariant)}
    commands
    variable = variable ± increment
}
```

Figure 5. Iterative structure **while**

The use of **while** allows the termination of the cycle according to the user's necessities.

1.5 Programming criteria

For the interpretation of a program the use of certain programming norms that allow the follow up of the program become necessary, as well as the detection and correction of errors.

1.5.1 Structure

A program or task can be realized in two ways. In the first, commands should be grouped by specific **functions** and then call them from a main code. By doing so, a short, simple and easy to understand code is achieved. The second alternative is to make a long and complex sequence of commands, therefore it is not advisable.

A function is a short sequence of commands that solves a specific problem. For complex tasks this problem will be divided in conceptually consistent sub problems. Each one will be responsibility of a function. The decomposition of sub-problems in yet other sub-problems follows the same philosophy, until achieving that the biggest problem is solved with the collaboration of several sub routines and small functions.

Figure 6 shows a group of calls and dependencies inside the program or function $\mathbf{F_1}$. An arrow of $\mathbf{F_i}$ toward $\mathbf{F_j}$ indicates that the function $\mathbf{F_i}$ uses the function $\mathbf{F_j}$.

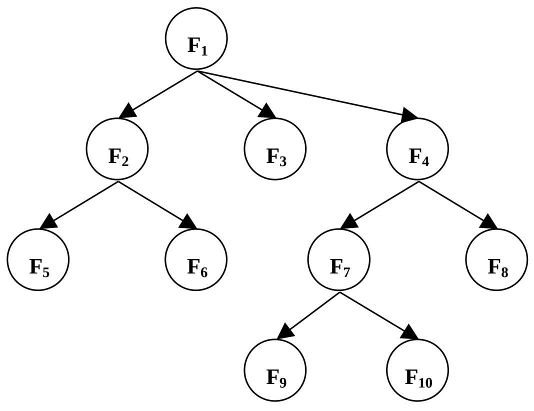


Figure 6. Graph of function calls

It is always suggested to begin to write the functions from the base of the graph of calls ($\mathbf{F_5}$, $\mathbf{F_6}$, $\mathbf{F_9}$, $\mathbf{F_{10}}$, $\mathbf{F_8}$). It allows to write and to prove each function as they end, thus saving time and avoiding errors.

1.5.2 Name of functions and variables

In the case of the functions, the name should describe its purpose clearly. The name of the variables has to be of mnemonic type so that it describes as much as possible their content. In order to increase the information provided by the names, it is necessary to implement the use of comments inside the program that give bigger description of use and type of data.

It is not advisable that numeric literals (constants) or others (1.05, 2, -1, ' a', etc) appear in the program. It is necessary to assign the literal to a variable and to use only that variable for that objective during the whole program. In the event of being required the change of such literal, it is easier to change the value in the initial assignment of the variable than to look for that literal in each instruction where it was used and replace the value. See Table 5.

Table 5. Usage of Symbolic Names

NOT RECOMENDED	**OPTIMAL**

if (*N* = = 2)
{
    i = 2; …
    k = *k*/2;
}
	size = 2;
	
	if (*N* = = *size***)**
	{
	    i = 2;
	    k = *k*/*size;*
	}

Notice that the use of constant *2* in the substitution process on the left side of Table 5 is not convenient. One reason is that its process is risky because **NOT** all constants *2* should be changed (See Table 5 right side).

1.5.3 Validation of data

It must be verified that the data input to every function are within the established range and that they have consistent values. There may exist erroneous data which enter the function, if this happens the problem should be solved. If it is not possible the program must be aborted and a report should be made informing the user about the existing error.

1.5.4 Errors

There exist two types of errors:

Of Syntax: they are found in the compilation phase or interpretation phase of the program, they occur due to characteristic causes of the language syntax. They are easy to correct.

Of Logic: they occur during the execution of a program. They are difficult to detect, they may or may not stop the execution of the program, thus producing erroneous results. The following are typical errors:

1. Inconsistent handling of matrix dimensions, operating matrices whose dimension is **NOT** compatible.
2. Omission of the "advancing towards ending" in a **while** cycle.
3. Incorrect calls of functions or disorder in the input parameters.

1.5.5 Association of operations

When using expressions, which involve two or more operators, it is important to apply the priority rules, which govern the order and the precedence of the operations. When a combination of conditions must be fulfilled, it is necessary to use grouping signs in order to ensure that they are executed in the desired sense. (Table 6).

Table 6. Association of conditions

NOT RECOMENDED	IMPROVED

size = 3;
degrees = 2;
if $\sim N < size \wedge M > degrees \vee R == 1$
{
    …
}
	size = 3;
	degrees = 2;
	if $(\sim ((N < size) \wedge (M > degrees))) \vee (R == 1)$
	{
	    …
	}

1.5.6 Indentation

Indentation is a tabulation that is controlled by the user, in which executions or specific tasks are visually grouped. A good indentation improves the design; it facilitates the debugging and modification of a program as well.

The structure of a program that calculates the factorial of a non-negative number is taken as an example (Table 7). It is obvious that the right side code is clearer, since it can be easily established which tasks are executed in the iterative command **while**.

Table 7. Comparison between indented and non-indented code.

NON-INDENTED CODE

```
{pre: N ≥ 0 }

i = 0;
fact = 1;

while (i ≠ N)
{ inv: fact = (i-1)!}
 fact = fact * i;
i = i + 1; }
{post: fact = N! }
```

INDENTED CODE

```
{pre: N ≥ 0 }

i = 0;
fact = 1;

    while (i ≠ N)
    {
            {inv: fact = (i-1)! }
            fact = fact * i;
            i = i + 1;
    }

{post: fact = N! }
```

1.6 EXERCISES - BASIC CONCEPTS OF PROGRAMMING.

A brief explanation about the operating of some MATLAB commands will be found on this chapter, creation of new functions and how to execute them from a main program. The commands are recognized because their characters are in **bold** letters.

1.6.1 The Variable Working Space in MATLAB.

OBJECTIVE:

To learn some basic commands.

PROCEDURE:

To obtain information. Type the command **help** and then the name of the topic or command to be consulted. **help**

1. Eliminate one or all the variables from the workspace. **clear**
2. Clean the command window. **clc**
3. List the variables of the workspace. **who**
4. See the latest computed result. **ans**

The commands used to work on MATLAB may be typed directly on the command window. They can also be previously written on a text file with extension ".m". When typing the file name at the MATLAB prompt the commands are read directly from the file and executed sequentially. In order to create this type of files (also called *script*s), the "New M-file" key is selected on the MATLAB menu (File / New / M-file), allowing the invocation the MATLAB text editor for their creation and debugging. For the correct operation of the *script*s it is necessary that the directory where the file to be executed is located be included on the MATLAB work path directories (see the **path** command).

1.6.2 Scalars, Vector and Matrix Operations in MATLAB.

OBJECTIVE:

Realize the main arithmetic operation among scalars, vector and matrices. To store the history of the session in a log file.

PROCEDURE:

1. Read the help about the **diary** command. Open a diary or log for the work that follows, with the name that you wish. You will need this file at the end of this exercise.
2. Eliminate one or all the variables from the workspace.
3. Create a scalar whose value is 12. Store the value of the variable at esc_1.
4. Create a scalar whose value is 4. Store the value of the variable at esc_2.
5. Add the values of the variable esc_1 and esc_2. Store the result in the variable *sum_esc*.
6. Subtract the values of the variable esc_1 and esc_2. Store the result in the variable *subst_esc*.
7. Multiply the values of the variable esc_1 and esc_2. Store the result in the variable *mult_esc*.
8. Divide the value of the variable esc_1 by the values of the variable esc_2. Store the result in the variable *div_esc*.
9. Raise the value of the variable esc_1 to the value of variable *div_esc*. Store the result in the variable *exp_esc*. **operator ^.**
10. Calculate the square root of the variable *exp_esc*. Store the result in the variable *rz_esc*. **sqrt**
11. Create a 4x4 matrix $\boldsymbol{M_1}$ with all its elements equal to one. **ones**
12. Create a 4x4 matrix $\boldsymbol{M_2}$ with all its elements being random numbers. **rand**
13. Create a 3x4 matrix $\boldsymbol{M_3}$ with all its elements equal to zero. **zeros**
14. Create a 4x4 identity matrix, the matrix name must be ***identity***. **eye**
15. Multiply the value of the variable esc_1 by the matrix ***identity***. Store the result in the variable $\boldsymbol{mult_1}$.
16. Multiply the matrix $\boldsymbol{mult_1}$ by the matrix $\boldsymbol{M_2}$. Store the result in the variable $\boldsymbol{mult_2}$. To calculate M3 = M1*M2 verify that M1 is *m* x *n* and M2 is *n* x *p*.
17. Multiply the value of the variable esc_2 by the matrix $\boldsymbol{M_1}$. Store the result in the variable $\boldsymbol{produc_1}$.
18. Multiply the elements of the matrix $\boldsymbol{produc_1}$ one by one by the elements of the matrix $\boldsymbol{M_2}$. Store the result in the variable $\boldsymbol{produc_2}$.
19. Raise the element of the matrix $\boldsymbol{M_2}$ to the square root. Store the result in the variable ***elev***.
20. Create the vector (1x3) $\boldsymbol{vect_1}$ = [1 2 3].
21. Save the first three values of an $\boldsymbol{M_2}$ column in the variable $\boldsymbol{vect_2}$.
22. Create a vector $\boldsymbol{vect_{50}}$ which values are in a range from 1 to 50 with intervals of 2. Operator **:**
23. Query the size of $\boldsymbol{vect_2}$ and store the result on the variable ***v_size***. **size**
24. Multiply the vector $\boldsymbol{vect_2}$ by transposed vector $\boldsymbol{vect_1}$ and store the result on the variable ***prod***.
25. Transpose the vector $\boldsymbol{vect_1}$ and store the result on the same variable.Close and save the log file.
26. Put the commands of this exercise in the file or *script* "*exercise_CP1_002.m*". Such commands can be found in the log file. Execute the file "*exercise_CP1_002.m*" from the prompt MATLAB. ***script***.

1.6.3 Introduction to User Commands for Session Control in MATLAB.

OBJECTIVE:

Create applications to work with users interface commands. Enter the input data with the keyboard or the mouse. This exercise must be executed from a *script* or a MATLAB file commands (*.m).

PROCEDURE:

1. Eliminate all the variables from the workspace
2. Clean the command window.
3. Create a menu window (for example to choose a day from the week) and store the result on the variable *day*. **menu**.
4. Make a pause and then continue with the execution. **pause**
5. Display a message on the MATLAB command line. It should request the user to press any key to continue with the execution program. **disp, pause**
6. Display a message on MATLAB command line. It should request the user to type a number. Store the result on the variable *number*. **Input**

1.6.4 2D Plotting in MATLAB.

OBJECTIVE:

Plot the expression "$\boldsymbol{y}$ = sin($\boldsymbol{x}$)". The $\boldsymbol{x}$ values are in a range [0, 2π] with intervals of π/12. This exercise must be executed from a *script* or a MATLAB file commands (*.m).

PROCEDURE:

1. Eliminate all the variables from the workspace
2. Request a MATLAB graphic window. **figure**
3. Clean the graphic window. **clf**
4. Create a row vector $\boldsymbol{x}$ with values in [0, 2π] with intervals of π/12.
5. Calculate $\boldsymbol{y}$ vector as "$\boldsymbol{y}$ = sin($\boldsymbol{x}$)". **sin( )**
6. Plot $\boldsymbol{x}$ vs $\boldsymbol{y}$. **plot**
7. Name the axis "X" and "Y" on the graphic. **xlabel**, **ylabel**
8. Title the graphic. **title**
9. Turn on the graphic grid. **grid**
10. Make a pause during the execution program.
11. Change the axis limits of the graphic. Left inferior point = (-1, -2), right superior point = (3π,2). **axis**
12. Make a pause during the execution.
13. Turn off the graphic grid.
14. Display in the graphic window the text '(1.0, 1.1)' on the window position (1.0, 1.1). **text**
15. Put on the graphic window labels for the left inferior point and right superior point.
16. Display a message on MATLAB command line. It should request the user to pick a point with the mouse, by using the graphic window. **ginput**
17. Display a text on the graphic window, use the mouse to pick its position. **gtext**

1.6.5 3D Plotting in MATLAB.

OBJECTIVE:

Calculate and plot the expression "***z*** = cos(***x***) + sin(***y***)" for values of ***x*** and ***y*** in a range [0, 10] with intervals of 0.05. This exercise must be executed from a *script* or a MATLAB file commands (*.m).

PROCEDURE:

1. Eliminate all the variables from the workspace.
2. Request a MATLAB graphic window.
3. Clean the graphic window.
4. Create a row vector ***x*** which values are in [0, 10] with intervals of 0.05.
5. Create a row vector ***y*** which values are in [0, 10] with intervals of 0.05.
6. Calculate ***z*** vector as function of "***z*** = cos(***x***) + sin(***y***)". ***cos( ), sin().***
7. Plot ***x***, ***y***, ***z***. ***plot3***
8. Place axes labels "X" , "Y" and "Z" on the graphic window.
9. Title the graph.
10. Turn on the graphic grid.

1.6.6 Plotting of 2D Polygonal Regions in MATLAB.

OBJECTIVE:

Use the mouse to capture points from the screen. Data input with keyboard or the mouse. This exercise must be executed from a *script* or a MATLAB file commands (*.m).

PROCEDURE:

1. Eliminate one or all the variables from the workspace
2. Display a message on MATLAB command line. It should request the user to type the number of points to be captured from the screen. Store the result on the variable *N*
3. Request a MATLAB graphic window.
4. Clean the graphic window.Create the vector ***coordX*** = [1,2,2,1].
5. Create the vector ***coordY*** = [2,2,3,3].
6. Plot the vector ***coordY*** vs ***coordX***.
7. Change the axis limits of the graphic. Left inferior point (0,1.5), right superior point (3,3.5).
8. Label the axis X” and “Y” on the graphic window.
9. Make a pause during the execution program.
10. Request a new MATLAB graphic window.
11. Extend the vector ***coordX*** and ***coordY*** in such a way that the command to plot ***coordY*** vs ***coordX*** generates a closed rectangle. Plot again ***coordY*** vs ***coordX***.
12. Freeze the graphic. **hold**
13. Change the axis limits of the graphic. Left inferior point (0,1.5), right superior point (3,3.5).
14. Fill the closed polygon. **fill**
15. Display a message on MATLAB command line to request the user to catch the *N* points from the screen.
16. Use the mouse to catch *N* points from the screen. Store the result on the array ***xy*** (*N*x2) .
17. Plot the second column of ***xy*** vs. the first one.

1.6.7 Generation and Plotting of 3D Meshes in MATLAB.

OBJECTIVE:

Plot a surface with different colors, by using the *colormap*. This exercise must be executed from a *script* or a MATLAB command file (*.m).

PROCEDURE:

1. Eliminate the variables from the work space.
2. Request a MATLAB graphic window.
3. Clean the graphic window.
4. Create a row vector ***x*** which values are in [-3, 3] with intervals of 0.5.
5. Create a row vector ***x*** which values are in [-2, 2] with intervals of 0.05.
6. Calculate ***X*** and ***Y*** matrices that define a grid, based on ***x*** and ***y***. ***meshgrid***
7. Calculate the ***Z*** matrix in function of ***X*** and ***Y***. ***peaks***
8. Plot ***Z***. **surf**
9. Define a gray color map. ***colormap***
10. Title the graphic.
11. Label the axis “X”, “Y” and “Z” on the graphic.
12. Request a new MATLAB graphic window.
13. Plot ***Z***.
14. Define a different color map.
15. Title the graphic.
16. Name the axis “X”, “Y” and “Z” on the graphic.

1.6.8 Menu Management in MATLAB.

OBJECTIVE:

To create an application by using the graph command **figure** and the interface command **menu**. This exercise must be executed from a *script* or a MATLAB command file (*.m).

PROCEDURE:

1. Eliminate all the variables from the work space.
2. Clean the command window
3. Create a window menu with the following options: yellow, blue, red, green and exit.
4. Iteratively, place a text (see table) in the graphic window, according to the user selection. To do that, you must request and clean a MATLAB graphic window inside of the iterative cycle. ***text, if, while***

Selection	Text	Position in graphic window
‘yellow’	‘yellow’	Lower right corner
‘blue’	‘blue’	Upper right corner
‘red’	‘red’	Upper left corner
‘green’	‘green’	Lower left corner
‘exit’	--	--

5. The program must end when the user selects the option *exit*. ***while***

1.6.9 Invariants and Iterative Cycles (summation of array contents) in MATLAB.

OBJECTIVE:

To calculate the summation of the values of a column vector ***M***. This exercise must be executed from a *script* or a MATLAB command file (*.m).

PROCEDURE:

1. Eliminate all the variables from the workspace.
2. Prompt the user for the number of rows of vector ***M***. Store that value in the variable *N*.
3. Generate the vector ***M*** of *N*x1. This vector must be filled with random values.
4. Calculate the summation of entries in ***M*** by using an iterative cycle **while** with a control variable *i*. A variable *sum* serves as a partial accumulator along the cycle execution.
5. Store the final result of *sum* in a new variable called *Sumator*.
6. Display on the command line the value of *Sumator*.

1.6.10 Invariants and Iterative Cycles (average values of array contents) in MATLAB.

OBJECTIVE:

To calculate the average of the values of a column vector ***M***. This exercise must be executed from a *script* or a MATLAB command file (*.m).

PROCEDURE:

1. Eliminate all the variables from the work space.
2. Prompt the user for the number of rows of vector ***M***. Store that value on the variable *N*.
3. Generate the vector ***M*** of *N*x1. This vector must be filled with random values.
4. Calculate the summation of entries in ***M*** by using an iterative cycle **while** with a control variable *i*. A variable *sum* serves as a partial accumulator along the cycle execution.
5. Calculate the average of the ***M*** values. This result must be store in the variable *mean*.

1.6.11 Invariants and Iterative Cycles (sorting or array contents) in MATLAB.

OBJECTIVE:

To sort the values of a row vector ***a*** in decreasing order. This exercise must be executed from a *script* or a MATLAB command file (*.m).

PROCEDURE:

1. Write a function **[*b*]** = ***swap*(*a* ,** i_1 , i_2 , i_3 , i_4**)**. This function swaps two values in a matrix. The inputs are: matrix ***a*** and the locations of the two values, *(i1 , i2)* and *(i3 , i4)* . The output parameter is the matrix ***b***, equal to ***a***, except for the swapped values.

2. Write a function **[*b*] = *my_sort*(*a*)**. The input parameter to this function is the vector (1x*N* or *N*x1) ***a*** in arbitrary order. The output parameter ***b*** is the sorted copy of **a**, with its values in decreasing order. This function should contain:

   2.1. A counter *cont1*, started at one.

   2.2. An iterative cycle **while** which covers all the positions of vector ***a***. Use the counter *cont1* and the variable *N* to control the iterative cycle operation. Create inside this cycle another iterative cycle **while**. It must be controlled by another counter called *cont2* and by the variable *N*. This last cycle ensures the organizing of the sub-array ***a***(1 : *cont1*) by calling function ***swap***.

1.6.12 Invariants and Iterative Cycles (minima and maxima of array contents) in MATLAB

OBJECTIVE:

To identify the maximum, minimum and the locations of those values within a vector ***M*** (1x*N* or *N*x1). This exercise must be executed from a *script* or a MATLAB command file (*.m).

PROCEDURE:

1. Eliminate all the variables from the work space.
2. Prompt the user for the number of rows of vector **M**. Store, that value in the variable *N*.
3. Generate the vector ***M*** of *N*x1. This vector must be filled with random values.
4. Initialize and use the variables:

   posmax: index where the maximum value is stored.

   posmin: index where the minimum value is stored.

   Max: maximum value of the vector.

   Min: minimum value of the vector.

5. Execute the iterative cycle **for** controlled by *i* and *N* to calculate the maximum and minimum values and its positions.
6. When the iterative cycle is completed store the final result is as it appears on Figure 7, in a matrix called ***Minmax***

$$Minmax = \begin{bmatrix} Min & posmin \\ Max & posmax \end{bmatrix}$$

Figure 7. Store the Final Results.

1.6.13 Functions and Sub-Routines in MATLAB.

OBJECTIVE:

To build a MATLAB program which calculates the average, addition, min and max, of the values of a matrix ***A***. This exercise must be executed from a *script* or a MATLAB command file (*.m).

PROCEDURE:

1. Eliminate all the variables from the work space.
2. Prompt the user for the number of rows and columns of matrix ***A***. Store these values in the variable *M* and *N* respectively.
3. Generate a matrix ***A*** of *M*x*N*. This matrix must be filled with random values.
4. Write a function [*sum*] = ***sumt (A)*** that calculates the total addition (*sum*) of the elements of matrix ***A***.
5. Write a function [*avg*] = ***average (A)*** that calculates the average value (*avg*) of the elements of matrix ***A***.
6. Write a function [*max_v* , ***posmax***] = ***maximum (A)*** that finds the maximum value (*max_v*) of the matrix ***A*** and stores its location in vector (*1*x*2*) ***posmax***.
7. Write a function [*min_v* , ***posmin***] =***minimum (A)*** that finds the minimum value (*vmin_v*) of the matrix ***A*** and stores its location in vector (*1*x*2*) ***posmin.***
8. Write a main program called ***exercise_1_13***, where each one of the developed functions is used.