PRACTICE 1

1 MATLAB

MATLAB is an interactive software package which was developed to perform numerical calculations on vectors and matrices. Initially, it was simply a MATrix LABoratory. However, today it is much more powerful.

1.1 Mathematical Functions in MATLAB

In this section, we will study the use of functions in MATLAB. Fist, we will show the standard elementary functions and, then, we will describe how to define our own functions.

1.1.1 Matlab functions

MATLAB provides a large number of standard elementary mathematical functions, including abs, sqrt, exp and sin. Most of them are entered exactly as you would write them mathematically. Taking the square root or logarithm of a negative number is not an error; the appropriate complex result is produced automatically. For a list of the elementary mathematical functions, type `help elfun`.

Note: Mathematical functions cannot usually be applied to matrices (or vectors). For example, e^A and $\sin(A)$ have no meaning unless A is a square matrix. But, in MATLAB, all the elementary mathematical functions are elementwise functions (element by element functions). We can see this in the following example:

```
>> sqrt([1 4 9 16])
ans =
     1     2     3     4
```

1.1.2 Creating functions

Besides the aforementioned functions, we can also define our own functions in MATLAB. It can be *numerical* functions or *symbolic* functions. The first type is used to do fast calculations and the second type is used to perform symbolic calculations such as derivatives, integrals, ...

Symbolic functions:

In order to create the symbolic functions $f(x) = x^2$ and $g(x, y) = x^2 - y^3$, we just execute

```
>> syms x y
>> f(x)=x^2
f(x) =
x^2
>> g(x,y)=x^2-y^3
g(x, y) =
x^2 - y^3
```

and we can obtain the value of a symbolic function at one or more input values as in the examples

```
>> f(1/3)
ans =
1/9
>> f(-2:2)
ans =
[ 4, 1, 0, 1, 4]
>> g(2,1)
ans =
3
```

We can use **double** command to obtain the double-precision value for the symbolic expression:

```
>> double(f(1/3))
ans =
    0.1111
```

We can also do symbolic operations like differentiation and integration.

```
>> f1=diff(f)
f1(x) =
2*x
>> f2=diff(f,2)
f2(x) =
2
```

If the function is a several variables function, it is better to specify the variable by which we are differentiating: `diff(function,variable)` and `diff(function,variable,n)`, computes the derivative and nth derivative of `function` with respect to `variable`, respectively:

```
>> g_x=diff(g,x)
g_x(x, y) =
2*x
>> g_y2=diff(g,y,2)
g_y2(x, y) =
-6*y
```

`int(fun)`, returns the indefinite integral or antiderivative of `fun`,
`int(fun,lower_limit,upper_limit)`, computes the definite integral of `function` from `lower_limit` to `upper_limit`:

```
>> int(f)
ans(x) =
x^3/3
>> double(int(f,0,1))
ans =
    0.3333
```

Numerical Functions or Anonymous Functions:

An *anonymous function* is associated with a variable whose data type is function_handle. Anonymous functions can accept inputs and return outputs, just as standard functions do. However, they can contain only a single executable statement.

The syntax is:

`nameoffunction=@(variables) expression`,

where the variables must be separated by commas and `expression` is the definition of the function depending on the variables.

For example, the anonymous function for $f(x) = x^4$

```
>> f=@(x) x^4
f =
    @(x)x^4
```

and the value of an anonymous function can be obtained:

```
>> f(1/2)
ans =
    0.0625
```

It is easy to convert a function of one type to the other one.

- **Anonymous function** to **Symbolic function**:

  ```
  >> syms x, f_sim(x)=f(x)
  ```

  now, it is possible to differentiate or integrate the function

  ```
  >> diff(f_sim)
  ans(x) =
  4*x^3
  >> int(f_sim)
  ans(x) =
  x^5/5
  ```

- **Symbolic function** to **Anonymous function**: using `matlabFunction`

  ```
  >> syms x;
  >> g(x)=x^3

  >> g_num=matlabFunction(g)
  ```

Anonymous function is an elementwise function (element by element function), for example, we can evaluate the function at 2, 3 and 5 by executing:

Let us create the function:

```
>> g=@(x) x^3
g =
    @(x)x^3
```

when we execute the statement

```
>> g([2 3 5])
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.
Error in @(x)x^3
```

We must use elementwise operators:

```
>> g=@(x) x.^3
g =
    @(x)x.^3
```

but sometimes it is useful to create the symbolic function and, then, convert it to anonymous function.

We finish this paragraph with an example of an anonymous function of several variables. To create the function $h(x,y) = x^2 + y^3$ and to evaluate $h(2,-1)$ it is enough to execute:

```
>> h=@(x,y) x.^2+y.^3
h =
    @(x,y)x.^2+y.^3
>> h(2,-1)
ans =
     3
```

1.2 Functions in files

We can create two kinds of files with the filename extension of .m (program files which that contains a sequence of MATLAB statements):

- **Scripts**, which do not accept input arguments or return output arguments. They operate on data in the workspace.
- **Functions**, which can accept input arguments and return output arguments. Internal variables are local to the function.

1.2.1 Scripts

Scripts are the simplest kind of MATLAB programs. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variable that they create remains in the workspace, so you can use them in further computations. In addition, scripts can produce graphical output using commands like plot.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

1.2.2 Functions

Functions are files that can accept input arguments and return output arguments. The names of the file and the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

A function in a file allows us to define numerical functions similar to anonymous functions with the name of the function, input arguments and output arguments.

The first line of a function in a file must be:

```
function [y_1 y_2 ... y_m]=functionname(x_1,x_2,...,x_n)
```

where `x_1,x_2,...,x_n` are the input arguments and `y_1,y_2,...,y_m` are the output arguments; `functionname` is (as its name indicates) the name we assign to the function and this name must be the same as the file name, that is, "`functionname.m`".

The body of a function can include valid MATLAB expressions, control flow statements, comments, blank lines, and nested functions. Any variable that you create within a function is stored within a workspace specific to that function, which is separate from the base workspace.

Once the file is saved, we call the function with the statement

```
[y_1 y_2 ... y_m]=functionname(x_1,x_2,...,x_n)
```

where the input arguments `x_1,x_2,...,x_n` must be the values we need to obtain the results we wanted.

Example 1.1 *Create a function in a file such that:*

- *the function name must be* `circle`*, then the file name must be "*`circle.m`*",*
- *the input argument must be the radius of a circle,*
- *the output argument must be the area of the circle*

Call the function to calculate the area of a circle with radius 2.

Solution: we create the file circle.m, with the following lines,

```
function a=circle(r)
%     Function a=circle(r) finds the area
%     of a circle with radius r
a=pi*r^2; % This is the body of the function
```

where we can observe that the only line of the body is ended by ";" in order to suppress printing.

Help text appears in the Command Window: "`help circle`"

```
>> help circle
Function a=circle(r) finds the area
of a circle with radius r
```

Finally, find the area of a circle with radius 2.

Note: As there is only one output variable, square brackets are not necessary in the statement to call the function. We execute `area=circle(2)` instead of `[area]=circle(2)`.

Example 1.2 *Create a function in a file such that:*

- *the function name must be* `rectang`*, then the file name must be "*`rectang.m`*",*
- *the input arguments must be the base and the height of a rectangle,*
- *the output arguments must be the area and the perimeter of the rectangle.*

Call the function to calculate the area and the perimeter of a rectangle with base 2 *and height* 3.

1.3 Control flow structures

1.3.1 The structure if/elseif/else/end

The **if** statement evaluates a logical expression and executes a group of statements when the expression is *true*. An expression is true when its result is nonempty and contains only nonzero elements (logical or real numeric). Otherwise, the expression is false.

```
if expression
statements_1 % Executed when expression is true
else
statements_2 % Executed when expression is false
end
```

In this case, the block `statements_1` is executed when `expression` is true, otherwise, the block `statements_2` is executed.

Taking this into account, we can create the file rectang2.m with the lines

```
function [a p]=rectang2(b,h)
%     Function [a p]=rectang2(b,h) finds the area a and
%     the perimeter p of a rectangle with base b and height h
if b<=0 || h<=0
disp('The input arguments must be positive real numbers')
a=[]; p=[];
else
a=b*h;
p=2*b+2*h;
end
```

```
>> [a p]=rectang2(2,-3)
```

Example 1.3 *Solve a* 2^{nd} *degree equation.*

Flowchart: $coef \to \Delta = b^2 - 4ac \begin{cases} \Delta > 0, & \text{2 raices reales distintas;} \\ \Delta = 0, & \text{raiz Ãºnica;} \\ \Delta < 0, & \text{raices complejas.} \end{cases}$

1.3.2 The structure for/end

Executes a group of statements in a loop for a specified number of times.

```
for k=vector
statements
end
```

Example 1.4 *Create a function in a file such that:*

- *the function name must be* `factorials`*, then the file name must be “*`factorials.m`*”,*
- *the input argument must be a non-negative integer, n,*
- *the output argument must be the factorial of n.*
- *the function must check that n is a non-negative integer, otherwise, an error message shall be displayed.*

Solution: we create the file factorials.m

```
function f=factorials(n)
%     function f=factorials(n) finds the factorial of n
if n==round(n) && n>=0
f=1;
for k=2:n
f=f*k;
end
else disp('ERROR: The input argument must be a non-negative integer')
f=[];
end
```

Now

test the created function:

```
>> factorials(-2)

>> factorials(pi)

>> factorials(5)
```

1.3.3 The structure while/end

Represents a loop that evaluates its body (statements) while a specified condition holds true.

```
while expression
     statements
end
```

Now, we are going to sum a convergent series $\sum_{n=1}^{\infty} a_n$ with MATLAB. Obviously, the problem is to calculate the limit of the partial sums of the series

$$S = \lim_{n\to\infty} S_n, \text{ where } S_n = a_1 + a_2 + \cdots + a_n$$

and the usual procedure is to calculate the elements of the sequence of partial sums S_n and use the condition

$$\frac{|S_{n+1} - S_n|}{|S_n|} \leq T$$

as stopping criterion, being T the relative error we wish. To avoid the division by zero, it is better use the stopping criterion like this: $|S_{n+1} - S_n| \leq T \cdot |S_n|$.

Example 1.5 *Create a function in a file such that:*

- *the function name must be* `series`*, then the file name must be "*`series.m`*",*
- *the input arguments must be an anonymous function a, which represents the nth term of the series a_n and the relative error T.*
- *the output argument must be approximation of the series sum and the numbers of terms have been summed.*

Solution: we create the file series.m

```
function [S n]=series(a,T)
%     Function [S n]=series(a,T) finds de sum of the series a(1)+a(2)+a(3)+...
% INPUT ARGUMENTS:
%   a ...... anonymous function, which represents the nth term
%            of the series
%   T ...... relative error
% OUTPUT ARGUMENTS:
%   S ...... approximation of the series sum
%   n ...... number of addends
n=1; S1=a(1); % n is the counter of addends
stopping_criterion=0;
while stopping_criterion==0
n=n+1;
S=S1+a(n);
stopping_criterion=abs(S-S1)<=T*abs(S1);
S1=S;
end
```

we check the function with the sum of the series:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

```
>> a=@(n) 1/n^2; [s n]=series(a,1e-9)
s =
1.6449
n =
24657
>> s-pi^2/6
ans =
-4.0556e-05
```

Note: If the input argument of a function is a function, like in this case where the function ***a*** is the input argument of the function `series`, MATLAB needs the function ***a*** in the memory like an anonymous function.

If the function is a function in file, the statement to call the function changes from `series(function,T)` to `series(@function,T)`.

For example, going back to the previous series, we can define the sequence in the file z.m, as follows

```
    function y=z(n)
    y=1/n^2;
```

and we obtain an error message by executing

```
>> [s n]=series(z,1e-9)
```

```
Error using z (line 2)
Not enough input arguments.
```

however, if we type

```
>> [s n]=series(@z,1e-9)
```

we obtain the same output as before.

The criterion to stop the iteration in the previous example, can be the number of summands to be used:

```
    function [S n]=series2(a,T,N)
    %     Function [S n]=series2(a,T,N) finds de sum of the series a(1)+a(2)+a(3)+...
    % INPUT ARGUMENTS:
    %   a ...... anonymous function, which represents the nth term
    %            of the series
    %   T ...... relative error
    %   N ...... Maximum number of addends to use
    % OUTPUT ARGUMENTS:
    %   S ...... approximation of the series sum
    %   n ...... number of addends

    n=1; S1=a(1); % n is the counter of addends
    stopping_criterion=0;
    while stopping_criterion==0 && n<N
    n=n+1;
    S=S1+a(n);
    stopping_criterion=abs(S-S1)<=T*abs(S1);
    S1=S;
    end
    if stopping_criterion==0
    disp(['It doesn' char(39) 't converges with the requested accuracy.'])
    disp('The result is not the solution but the latter amount calculated.')
    end
```

For example, the series

$$\sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{n} = \ln 2$$

converges very slowly, so we avoid using more than 50.000 addends by using

```
>> b=@(n) (-1)^(n-1)/n
b =
@(n)(-1)^(n-1)/n


>> [s n]=series2(b,1e-9,50000)
It doesn't converge with the requested accuracy.
The result is not the solution but the latter amount calculated.
s =
0.6931
n =
50000
```

1.4 Accuracy of calculations

- **eps:** shows the accuracy of real numbers. In MATLAB, the accuracy of real numbers (internal representation) is always the same: between 15 and 16 digits. The accuracy is the distance from abs(X) to the next larger floating point number (in magnitude) of the same precision as X. For example, if we execute

  ```
  >> eps(10)
  ans =
  1.7764e-15
  ```

  so, the next floating point number to `10` is `10+1.7764e-015`, but, if we calculate

  ```
  >> 10+10^(-16)
  ans =
  10
  ```

  we obtain a wrong result.

  With the following commands, we can see the evolution of `eps(x)` and the resulting loss of accuracy, as `x` grows:

  ```
  >> eps(0)
  ans =
  4.9407e-324
  >> eps(1)
  ans =
  2.2204e-16
  >> eps % observe that it is the same as eps(1)
  ans =
  2.2204e-16
  >> eps(1000)
  ans =
  1.1369e-13
  >> eps(10^16)
  ans =
  2
  ```

- **realmax:** gives the largest number that MATLAB can handle:

  ```
  >> realmax
  ans =
  1.7977e+308
  ```

  and this limitation may result, with large numbers calculations, in errors or not valid mathematical properties.

  ```
  >> 10^308*2*0.1
  ans =
  Inf
  >> 10^308*(2*0.1)
  ans =
  2.0000e+307
  ```

- **format:** changes the number of digits that MATLAB shows, normaly 5 significant digits.

  ```
  >> format long
  >> pi
  ans =
  3.141592653589793
  ```

  we can go back to the default format

  ```
  >> format
  ```

- **fprintf:** Regardless of the format we use, we can always decide the number of decimal places that we want to display, using the command **fprintf**.

  The simplest use of this function is to show text (which must be entered with single quotation marks `'` around it) as it is shown

  ```
  >> fprintf('Text of the example\n')
  Text of the example
  ```

  the string `\n` means break line.

  Another way to use this command is to display text and variable values: by executing `fprintf('... % ... % ... % ...',x1,x2,x3, ...)`, MATLAB substitutes the first symbol `%` with variable value `x1`, the second one with variable value `x2` and so on, showing these values with the format indicated below the symbol `%`.

  As an example, we see how to show the length of a circumference of radius $\sqrt{2}$:

  ```
  >> r=sqrt(2), l=2*pi*r
  r =
  1.4142
  l =
  8.8858
  >> fprintf('The length of a circumference of radius %.2f is %.7f\n',r,l)
  The length of a circumference of radius 1.41 is 8.8857659
  ```

  Notice that with the expressions `%.2f` and `%.7f` the variables `r` and `l` are shown with 2 and 7 decimal digits, respectively.

2 Zeros of Nonlinear Equations I

A fundamental principle in computer science is iteration. As the name suggests, the process is repeated until an answer is achieved.

Iterative techniques are used to find roots of equations, solutions of linear and nonlinear systems of equations and solutions of differential equations.

We will now begin the analysis of two methods to find numerical approximations for the roots of an equation: fixed point iteration method and bisection method of Bolzano.

2.1 Bisection method of Bolzano

We must start with a real function $f(x)$, of variable $x \in \mathbb{R}$, we want to find the roots of f, that is, to solve the equation $f(x) = 0$.

The bisection method, based on Bolzano's theorem, consists of finding an interval $[a, b]$ where $f(a)$ and $f(b)$ have opposite signs. Since the graph $y = f(x)$ of a continuous function is unbroken, it will cross the x-axis at a zero $z = r$ that lies somewhere in the interval $[a, b]$. The bisection method systematically moves the end points of the interval closer and closer together until we obtain an interval of arbitrarily small width that brackets the zero. The decision step for this process of interval halving is first to choose the midpoint of the interval $[a, b]$, $x = \frac{a+b}{2}$ and then to analyze the three possibilities that might arise:

1. if $f(a)$ and $f(x)$ have opposite signs, a zero lies in $[a, x]$.
2. if $f(x)$ and $f(b)$ have opposite signs, a zero lies in $[x, b]$.
3. if $f(x) = 0$, then the zero is x.

If either case (1) or (2) occurs, we have found an interval half as wide as the original interval that contains the root, and we are "squeezing down on it". By repeating the process, we can obtain an interval as small as desired and, ultimately, bring the solution to the desired accuracy.

Since both the zero r and the midpoint x lie in the interval $[a, b]$, the distance between them can't be greater than half the width of this interval, that is,

$$|x - r| \leq \frac{b - a}{2}$$

.

By applying the method N times, the error bound should be $\frac{b-a}{2^N}$, that is,

$$|x - r| \leq \frac{b - a}{2^N}$$

so a way to make sure that the maximum error was a preset amount E, would be to find N such that

$$\frac{b - a}{2^N} = E$$

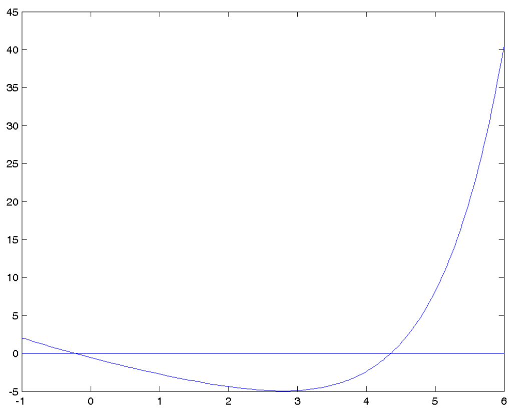
and after rounding to the nearest integer greater than or equal to it, if the result is not exact, apply the bisection method N times.

Create a function in file, named bisection.m to compute the zeros of a function using the bisection methos.

Example 2.1 *Find an approximation of the roots of* $f(x) = e^{x-2} - \ln(x+2) - 2x$ *that lie in the interval* $[-1, 6]$*, with a maximum absolute error of* 10^{-12}*:*

Solution: First, we define the function and represent it graphically, by running the statements

```
>> f=@(x) exp(x-2)-log(x+2)-2*x
f =
@(x)exp(x-2)-log(x+2)-2*x
>> fplot(f,[-1 6]);
>> hold on; fplot(@(x) 0,[-1 6]);
```



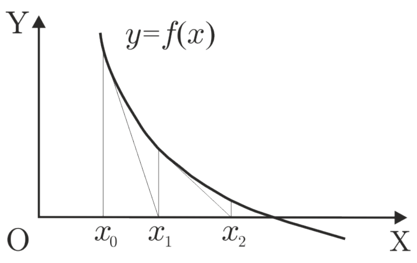
and we see that the function has two roots, the first one lies in the interval $[-1, 0]$ and the second one in the interval $[4, 5]$, so, to calculate with the required precision, we simply do:

```
>> x1=bisection(f,-1,0,1e-12)
x1 =
-0.2314
>> x2=bisection(f,4,5,1e-12)
x2 =
4.3575
```

2.2 Newton-Raphson's method

We use this method when the exact differential may be computed.

- Start with x_0 close to the root r, define x_1 to be the point of intersection of the x-axis and the tangent line to the curve at the point $(x_0, f(x_0))$.
- The process is repeated to obtain a sequence $\{x_n\}$ that converges to r.



The line tangent to the curve at the point $(x_0, f(x_0))$ is $y - y_0 = m(x - x_0)$, thus:

$$y - f(x_0) = f'(x_0)(x - x_0)$$

and the intersection with the x-axis $(y = 0)$ is

$$-f(x_0) = f'(x_0)(x - x_0) \implies -\frac{f(x_0)}{f'(x_0)} = x - x_0 \implies x_0 - \frac{f(x_0)}{f'(x_0)} = x$$

therefore,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

and in general, we obtain the sequence:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

We will use the stopping criterion

$$\frac{|x_{n+1} - x_n|}{|x_n|} \leq T$$

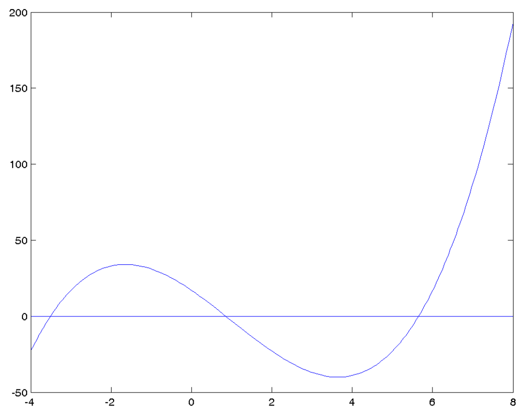
being T the relative error (known as tolerance). To prevent possible divisions by numbers close to zero, we will raise it as $|x_{n+1} - x_n| \leq T \cdot |x_n|$.

Example 2.2 *Check graphically that the equation $x^3 - 3x^2 - 18x + 17 = 0$ has its three solutions at the interval $[-4, 8]$. Find them by applying the Newton-Raphson method, with relative tolerance 10^{-6}.*

Solution: We define $f(x) = x^3 - 3x^2 - 18x + 17$ as symbolic function, so we can obtain the derivative with MatLab

```
>> syms x; f(x)=x^3-3*x^2-18*x+17
f(x) =
x^3 - 3*x^2 - 18*x + 17
>> fd=diff(f)
fd(x) =
3*x^2 - 6*x - 18
```

and we convert them to numerical functions and plot the graph of $y = f(x)$:



Finally, we apply the **newton** function, and the three solutions are:

```
>> x1=newton(f_num,fd_num,-3,1e-6,200)
x1 =
-3.5094
>> x2=newton(f_num,fd_num,2,1e-6,200)
x2 =
0.8570
>> x3=newton(f_num,fd_num,6,1e-6,200)
x3 =
5.6524
```

Example 2.3 *Create the function* `newton2` *by changing the function* `newton`*, so that the output variable contains the values of the iterations of the Newton-Raphson method. Then, apply* `newton2`*, for solving the equation of the previous example.*

and we find the solutions of the equation of the previous example,

```
>> y1=newton2(f_num,fd_num,-3,1e-6,200)
y1 =
-3.6296   -3.5140   -3.5094   -3.5094   -3.5094
>> y2=newton2(f_num,fd_num,2,1e-6,200)
y2 =
0.7222    0.8576    0.8570    0.8570
>> y3=newton2(f_num,fd_num,6,1e-6,200)
y3 =
5.6852    5.6527    5.6524    5.6524
```