

PRÁCTICA 1

MÉTODOS NUMÉRICOS

G12: SANDRA GÓMEZ Y SERGIO CASADO

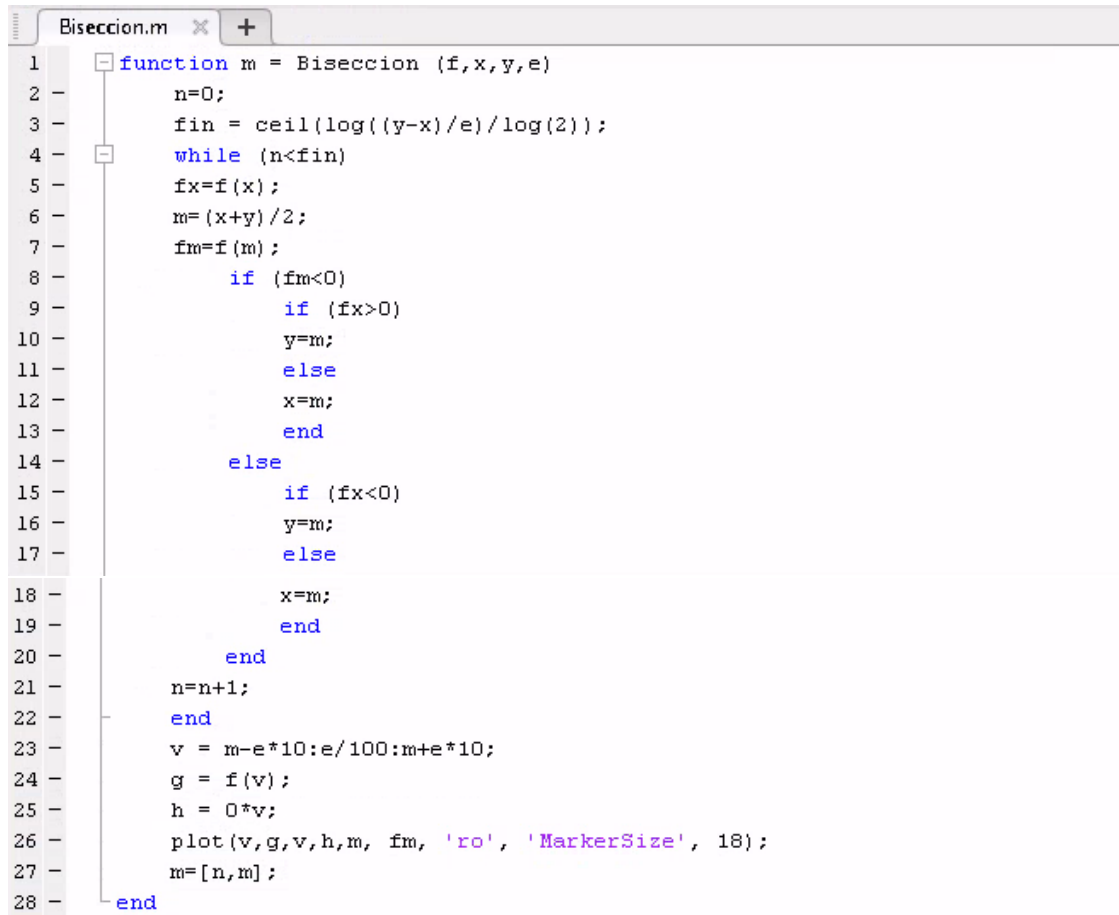
PRÁCTICA 1

PARTE 1: RAÍCES NUMÉRICAS DE FUNCIONES NO LINEALES

1) Programar el método de bisección en Matlab

Biseccion es una función a la que se introduce f, x, y, e . Siendo f la función $f(x)$ con la que trabajaremos, x e y los puntos x_1^0 y x_2^0 respectivamente, y e la tolerancia.

Esta función nos devolverá $m=[n,m]$. Siendo n el número de iteraciones y m la raíz.



```

1 function m = Biseccion (f,x,y,e)
2     n=0;
3     fin = ceil(log((y-x)/e)/log(2));
4     while (n<fin)
5         fx=f(x);
6         m=(x+y)/2;
7         fm=f(m);
8         if (fm<0)
9             if (fx>0)
10                y=m;
11            else
12                x=m;
13            end
14        else
15            if (fx<0)
16                y=m;
17            else
18                x=m;
19            end
20        end
21        n=n+1;
22    end
23    v = m-e*10:e/100:m+e*10;
24    g = f(v);
25    h = 0*v;
26    plot(v,g,v,h,m, fm, 'ro', 'MarkerSize', 18);
27    m=[n,m];
28 end
  
```

2) Programar el método de Newton-Raphson en Matlab

NewtonRaphson es una función a la que introduce f, fd, pto, e . Siendo f la función $f(x)$ con la que trabajaremos, fd la derivada de $f(x)$, pto es el punto x_0^0 , y e es la tolerancia.

Esta función nos devolverá $sol=[y1, "n =", n]$, siendo $y1$ la raíz y n el número de iteraciones.

```

1  function sol = NewtonRaphson(f,fd,pto,e)
2      n=1;
3      y0=pto;
4      y1=y0-(f(y0)/fd(y0));
5      while(abs(y1-y0)>e)
6          y0=y1;
7          n=n+1;
8          y1=y0-(f(y0)/fd(y0));
9      end
10     v = y1-e*10:e/100:y1+e*10;
11     g = f(v);
12     h = 0*v;
13     plot(v,g,v,h,y1, f(y1), 'ro', 'MarkerSize', 18);
14     sol=[y1,"n = ",n];
15 end

```

Aplicar los programas desarrollados a la evaluación de la raíz positiva de la función de una variable real $f: \mathbb{R} \rightarrow \mathbb{R}$ definida por $f(x) = x \sin\left(\frac{1}{2}x^2\right) + e^{-x}$.

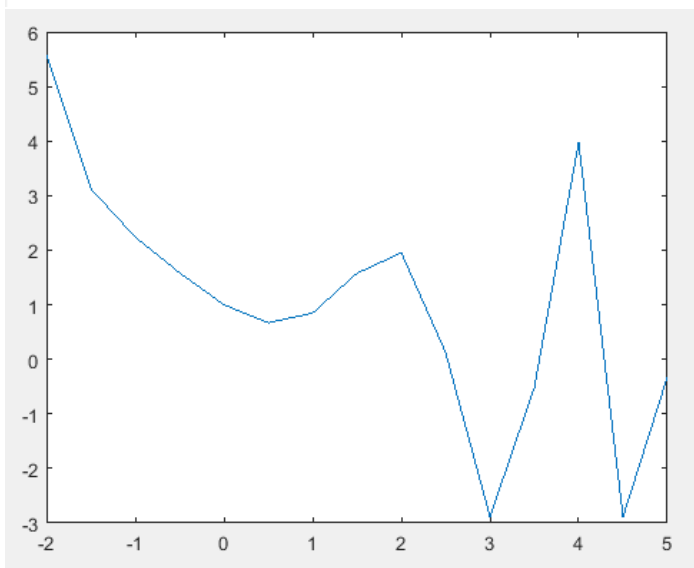
Bisección:

Dibujamos la gráfica por consola para evaluar la raíz positiva:

```

>> x= -2:0.5:5;
>> y = (x.*sin((1/2)*x.^2)+(exp(-x)));
>> plot(x,y);

```



Elegimos los puntos $x_1^0 = 2$ y $x_2^0 = 3$, que serán respectivamente nuestras nuevas x e y que introduciremos por pantalla, y elegiremos (por ejemplo) una tolerancia de $1/1000$.

```
>> f= @(x) x.*sin((1/2)*x.^2)+exp(-x)

f =

    function_handle with value:

    @(x)x.*sin((1/2)*x.^2)+exp(-x)

>> x=2

x =

    2

>> y=3

y =

    3

>> e=1/1000

e =

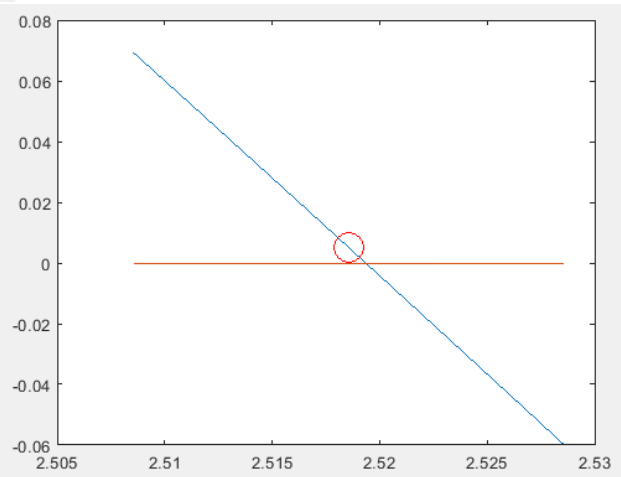
    1.0000e-03
```

Que resulta:

```
>> Biseccion(f,x,y,e)

ans =

    10.0000    2.5186
```



Cambiando e resulta:

```
>> e= 1/100

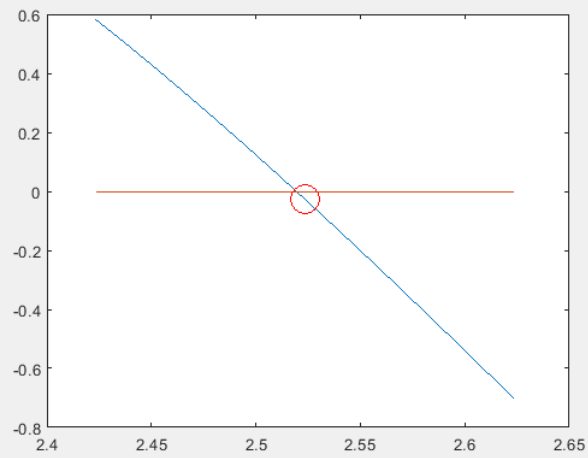
e =

    0.0100

>> Biseccion(f,x,y,e)
```

```
ans =
```

```
7.0000    2.5234
```



```
>> e=1/10000
```

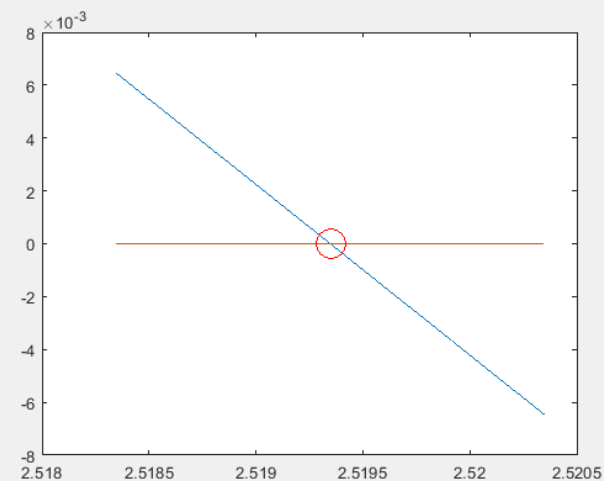
```
e =
```

```
1.0000e-04
```

```
>> Biseccion(f,x,y,e)
```

```
ans =
```

```
14.0000    2.5193
```



Esto ocurre porque al cambiar la cota de error, la aproximación cambia ajustándose más o menos a la raíz exacta.

Newton - Raphson:

Elegimos el punto $x_0^0 = 3$. Y elegiremos (por ejemplo) una tolerancia de $1/1000$.

```

f =
    function_handle with value:
        @(x)x.*sin((1/2)*x.^2)+exp(-x)
>> fd= @(x) sin((x.^2)/2)+(x.^2)*cos((x.^2)/2)-exp(-x)
fd =
    function_handle with value:
        @(x)sin((x.^2)/2)+(x.^2)*cos((x.^2)/2)-exp(-x)
>> pto= 3
pto =
    3
>> e= 1/1000
e =
    1.0000e-03

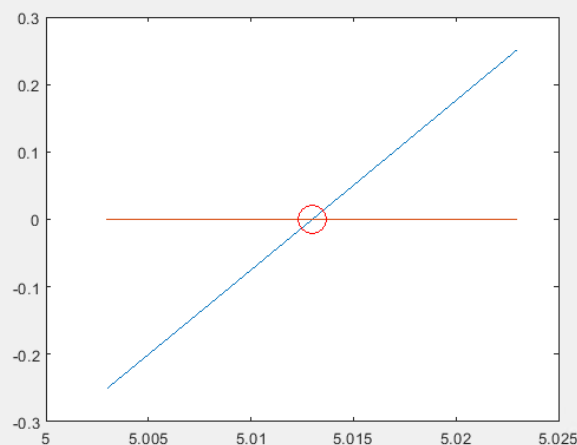
```

Que resulta:

```

>> NewtonRaphson(f,fd,pto,e)
ans =
    1×3 string array
    "5.013"    "n ="    "58"

```



Cambiando e resulta:

```
>> e=1/100

e =

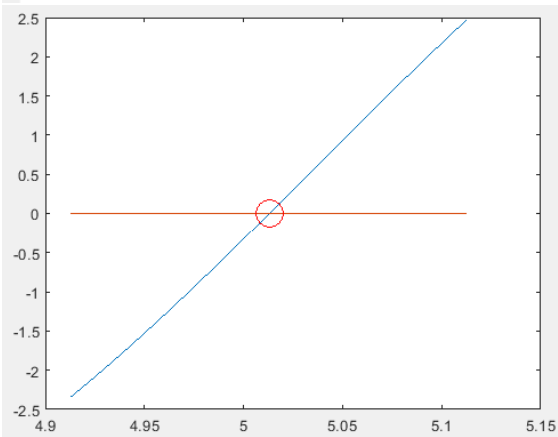
    0.0100

>> NewtonRaphson(f,fd,pto,e)

ans =

    1×3 string array

    "5.013"    "n ="    "58"
```



```
>> e=1/10000

e =

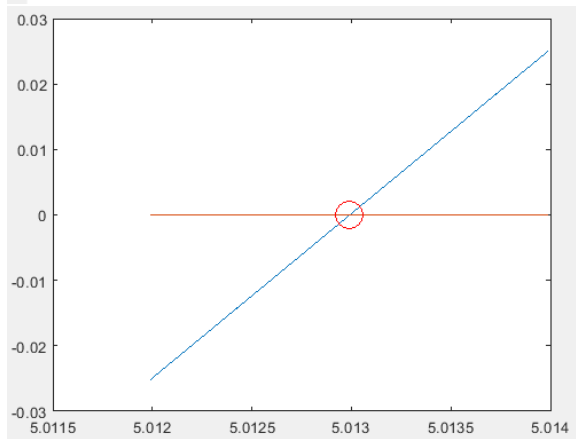
    1.0000e-04

>> NewtonRaphson(f,fd,pto,e)

ans =

    1×3 string array

    "5.013"    "n ="    "59"
```



Comparar los resultados con el resultado de referencia Rr obtenido empleando el comando fzero (help fzero).

```
>> f= @(x) x.*sin((1/2)*x.^2)+exp(-x)

f =

function_handle with value:

@(x) x.*sin((1/2)*x.^2)+exp(-x)

>> fzero(f,2)

ans =

2.5193
```

Mediante fzero con x=2 podemos observar como el método *Bisección* nos da el mismo valor de raíz. Con x=3:

```
>> fzero(f,3)

ans =

2.5193
```

Por tanto, el método *Bisección* nos da una mejor aproximación a la raíz que el método *Newton-Raphson*.

PARTE 2: RAÍCES NUMÉRICAS DE FUNCIONES NO LINEALES

Aplicar los programas desarrollados anteriormente a los problemas siguientes.

Utilizaremos los programas desarrollados en el apartado 1) y 2) anterior.

2.1) $f: \mathbb{R} \rightarrow \mathbb{R}, f(x) = \sin(x) - 0.3e^x$

Raíz positiva >0,6 a partir de los puntos iniciales 1 y 4 (Bisección) y 1 (Newton-Raphson) con tolerancia $\varepsilon=1/100$.

Bisección:

Por consola introducimos:


```
>> f= @(x) sin(x)-0.3*exp(x)

f =

function_handle with value:

    @(x)sin(x)-0.3*exp(x)

>> x=1

x =

    1

>> y=4

y =

    4

>> e=1/100

e =

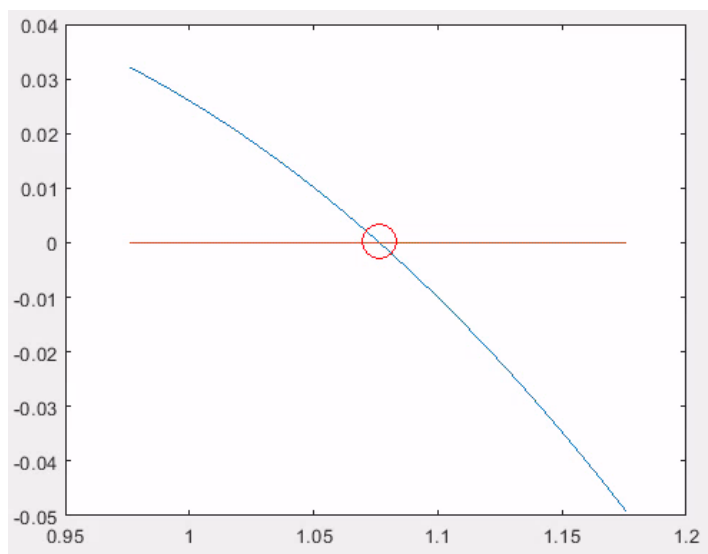
    0.0100

>> Biseccion(f,x,y,e)
```

Que resulta:

```
ans =

    9.0000    1.0762
```



Newton-Raphson:

Por consola introducimos:

```
f =
    function_handle with value:
        @ (x) sin(x)-0.3*exp(x)

>> fd= @ (x) cos(x) - (3*exp(x)/10)

fd =
    function_handle with value:
        @ (x) cos(x) - (3*exp(x)/10)

>> pto=1

pto =
    1

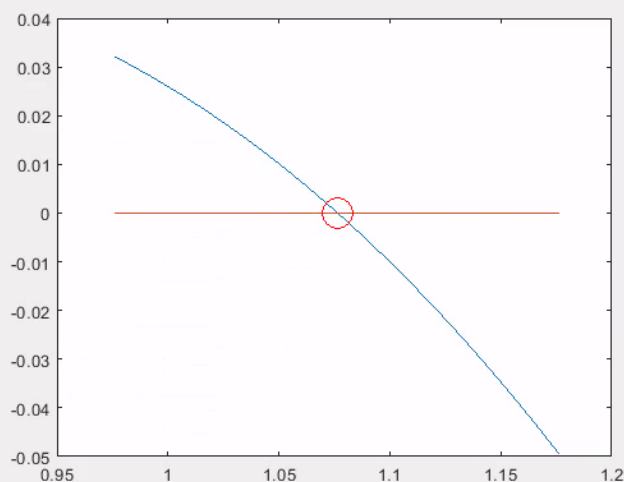
>> e= 1/100

e =
    0.0100
```

Que resulta:

```
>> NewtonRaphson(f,fd,pto,e)

ans =
    1×3 string array
    "1.0765"    "n ="    "3"
```



2.2) $f: [0, \infty) \subset \mathbb{R} \rightarrow \mathbb{R}, f(x) = \sqrt{x} - \cos(x)$

Raíz positiva con tolerancia $\varepsilon=1/1000$ a partir de los puntos iniciales: a) 0 y 4 (Bisección) y 1 (Newton-Raphson); b) 0,5 y 1 (Bisección) y 0,5 (Newton-Raphson).

a)

Bisección:

Por consola introducimos:

```
>> f= @(x) sqrt(x)-cos(x)

f =

    function_handle with value:

    @(x) sqrt(x)-cos(x)

>> e=1/1000

e =

    1.0000e-03

>> x=0

x =

    0

>> y=4

y =

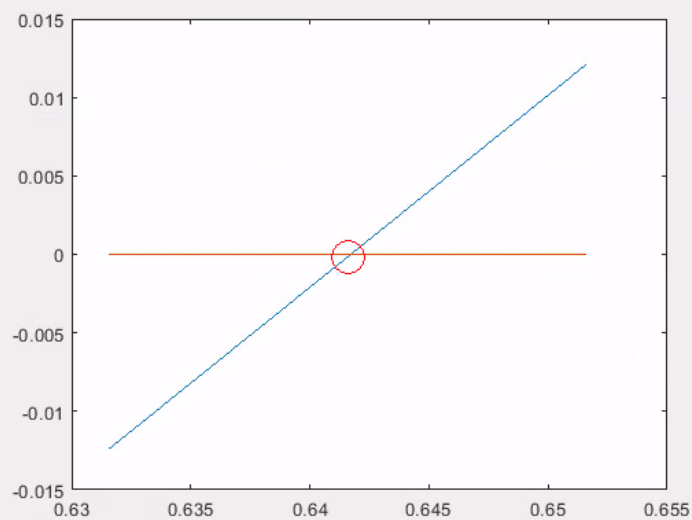
    4

>> Biseccion(f,x,y,e)
```

Que resulta:

```
ans =

    12.0000    0.6416
```

*Newton-Raphson:*

Por consola introducimos:

```
>> f= @(x) sqrt(x)-cos(x)

f =

    function_handle with value:

    @(x) sqrt(x)-cos(x)

>> fd=@(x) sin(x)+(1/(2*sqrt(x)))

fd =

    function_handle with value:

    @(x) sin(x)+(1/(2*sqrt(x)))

>> pto=1

pto =

    1

>> e= 1/1000

e =

    1.0000e-03
```

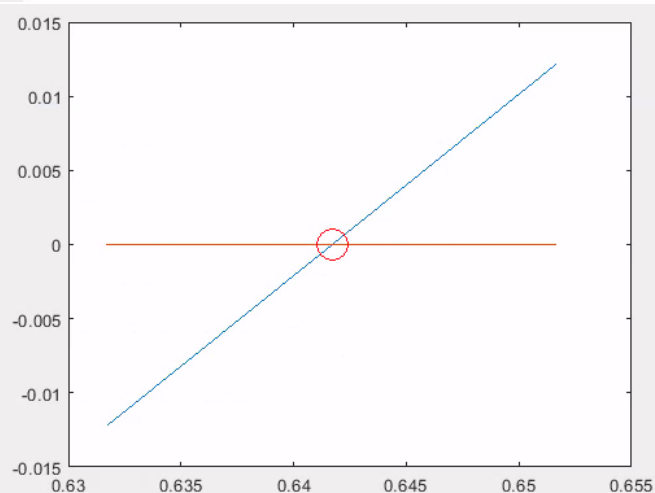
Que resulta:

```
>> NewtonRaphson(f,fd,pto,e)

ans =

    1×3 string array

    "0.64171"    "n ="    "3"
```



b)

Bisección:

Por consola introducimos:

```
>> f= @(x) sqrt(x)-cos(x)

f =

    function handle with value:

    @(x) sqrt(x)-cos(x)

>> e=1/1000

e =

    1.0000e-03

>> x=0.5

x =

    0.5000

>> y=1

y =

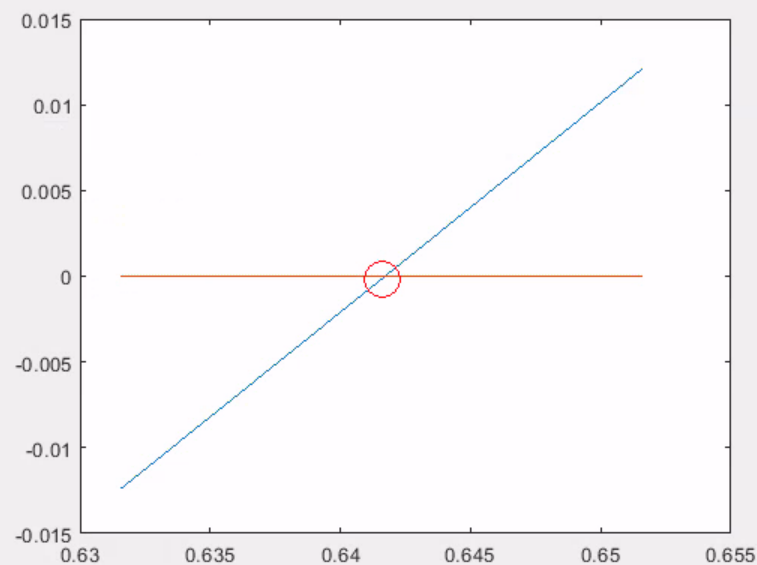
    1
```

Que resulta:

```
>> Biseccion(f,x,y,e)

ans =

    9.0000    0.6416
```



Newton-Raphson:

Por consola introducimos:

```
>> f=@(x) sqrt(x)-cos(x)

f =

    function_handle with value:

    @(x) sqrt(x)-cos(x)

>> fd=@(x) sin(x)+(1/(2*sqrt(x)))

fd =

    function_handle with value:

    @(x) sin(x)+(1/(2*sqrt(x)))

>> pto= 0.5

pto =

    0.5000

>> e= 1/1000

e =

    1.0000e-03
```

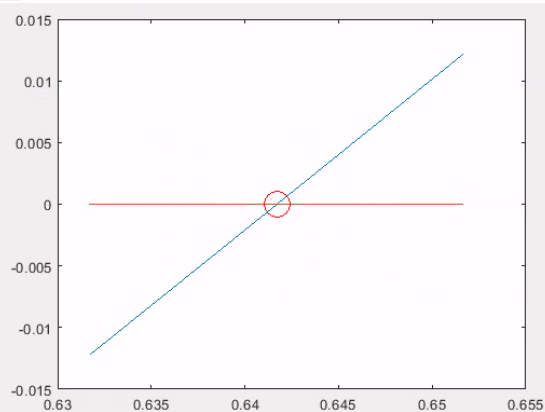
Que resulta:

```
>> NewtonRaphson(f,fd,pto,e)

ans =

    1×3 string array

    "0.64171"    "n ="    "3"
```



2.3) $f: \mathbb{R} \rightarrow \mathbb{R}, f(x) = 2x^3 - 11,7x^2 + 17,7x - 5$

Raíces a partir de los puntos iniciales: a) 0 y 1, 1 y 3, 3 y 5 (Bisección con tolerancia $\varepsilon=1 \times 10^{-3}$); b) 0,5 , 1,5 , 4 (Newton-Raphson con tolerancia $\varepsilon=1 \times 10^{-10}$).

a)

Bisección 0 y 1:

Por consola introducimos:

```
>> f= @(x) 2*x.^3-11.7*x.^2+17.7*x-5

f =

function_handle with value:

    @(x) 2*x.^3-11.7*x.^2+17.7*x-5

>> x=0

x =

    0

>> y=1

y =

    1

>> e=1*10^(-3)

e =

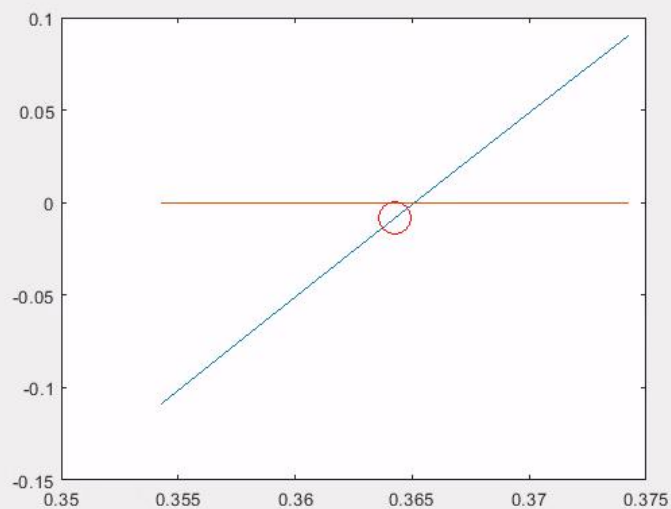
    1.0000e-03
```

Que resulta:

```
>> Biseccion(f,x,y,e)

ans =

    10.0000    0.3643
```



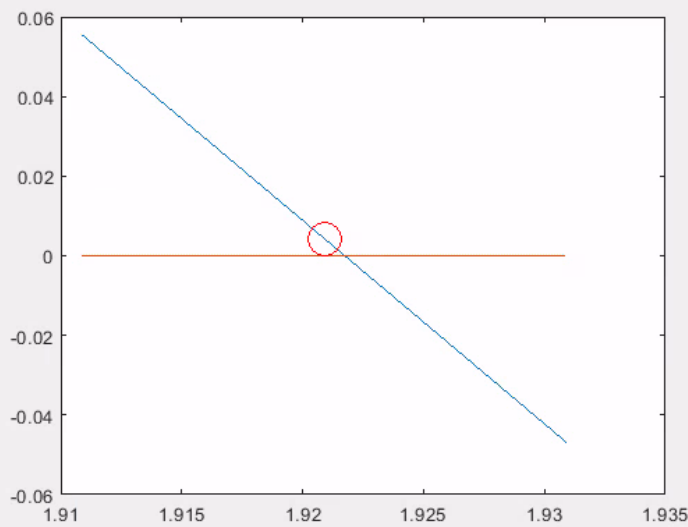
Bisección 1 y 3:

Por consola introducimos:

```
>> x=1  
  
x =  
  
    1  
  
>> y=3  
  
y =  
  
    3
```

Que resulta:

```
>> Biseccion(f,x,y,e)  
  
ans =  
  
    11.0000    1.9209
```

*Bisección 3 y 5:*

Por consola introducimos:

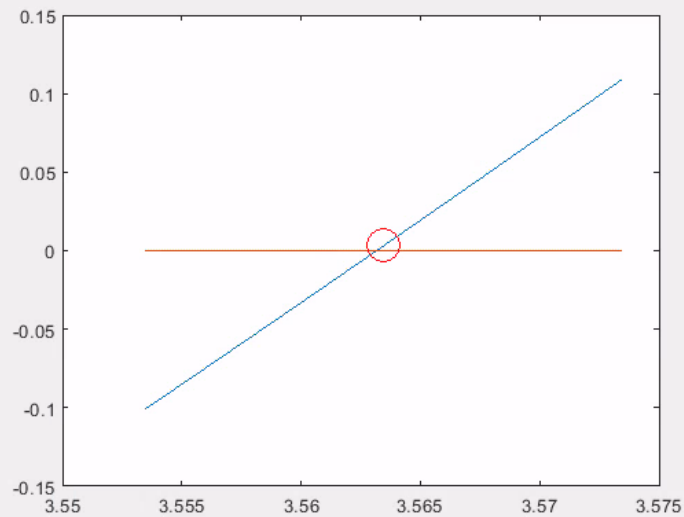
```
>> x=3  
  
x =  
  
    3  
  
>> y=5  
  
y =  
  
    5
```

Que resulta:


```
>> Biseccion(f,x,y,e)
```

```
ans =
```

```
11.0000    3.5635
```



Newton-Raphson 0.5:

Por consola introducimos:

```
>> f= @(x) 2*x.^3 -11.7*x.^2 +17.7*x -5
```

```
f =
```

```
function_handle with value:
```

```
@(x) 2*x.^3-11.7*x.^2+17.7*x-5
```

```
>> fd= @(x) 6*x.^2-(117*x)/5 +177/10
```

```
fd =
```

```
function_handle with value:
```

```
@(x) 6*x.^2-(117*x)/5+177/10
```

```
>> e= 1*10^(-10)
```

```
e =
```

```
1.0000e-10
```

```
>> pto=0.5
```

```
pto =
```

```
0.5000
```

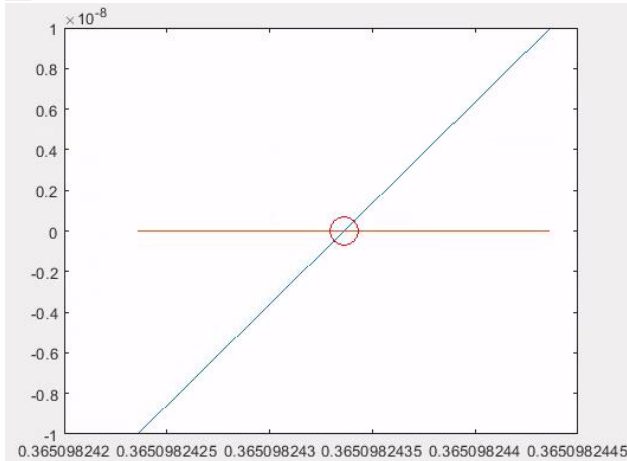
Que resulta:

```
>> NewtonRaphson(f,fd,pto,e)

ans =

1×3 string array

    "0.3651"    "n ="    "5"
```



Newton-Raphson 1.5:

Por consola introducimos:

```
>> pto=1.5

pto =

    1.5000
```

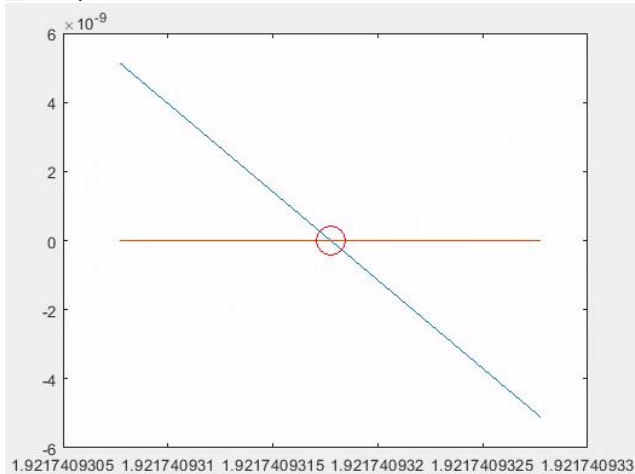
Que resulta:

```
>> NewtonRaphson(f,fd,pto,e)

ans =

1×3 string array

    "1.9217"    "n ="    "5"
```



Newton-Raphson 4:

Por consola introducimos:

```
>> pto=4

pto =

     4
```

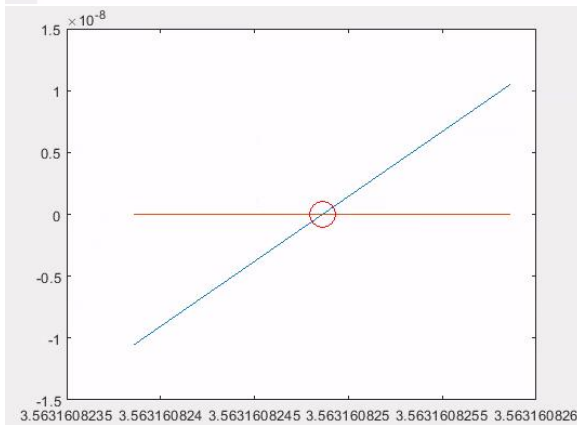
Que resulta:

```
>> NewtonRaphson(f,fd,pto,e)

ans =

1×3 string array

    "3.5632"    "n ="    "6"
```



2.4) $f: \mathbb{R} \rightarrow \mathbb{R}, f(x) = e^{\frac{1}{2}x} + 5x - 5$

Raíz con tolerancia $\varepsilon=1 \times 10^{-3}$ a partir de los puntos iniciales: a) 0 y 4 (Bisección) y 1 (Newton-Raphson).

Bisección:

Por consola introducimos:

```
>> f= @(x) exp((1/2)*x)+5*x-5

f =

    function handle with value:

    @(x)exp((1/2)*x)+5*x-5

>> x=0

x =

    0

>> y=4

y =

    4

>> e= 1*10^(-3)

e =

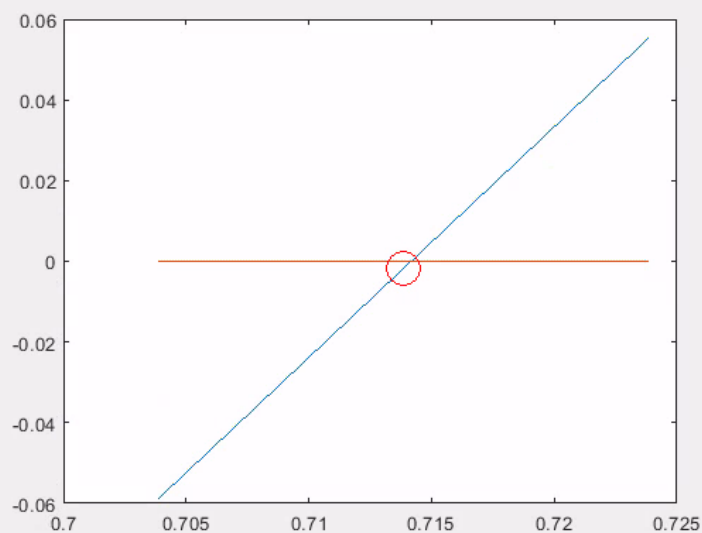
    1.0000e-03
```

Que resulta:

```
>> Biseccion(f,x,y,e)

ans =

    12.0000    0.7139
```



Newton-Raphson:

Por consola introducimos:

```
>> f= @(x) exp((1/2)*x)+5*x-5

f =

function_handle with value:

    @(x)exp((1/2)*x)+5*x-5

>> fd= @(x) (exp(x/2)/2)+5

fd =

function_handle with value:

    @(x)(exp(x/2)/2)+5

>> pto=1

pto =

    1

>> e= 1*10^(-3)

e =

    1.0000e-03
```

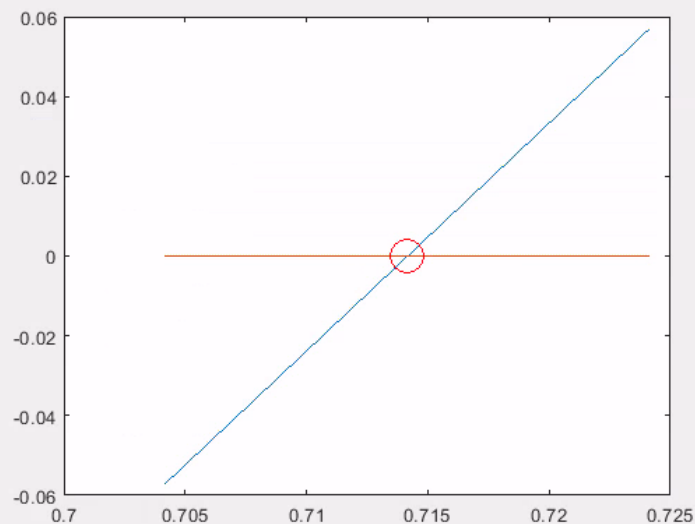
Que resulta:

```
>> NewtonRaphson(f,fd,pto,e)

ans =

1×3 string array

    "0.71417"    "n ="    "3"
```



PARTE 3: RAÍCES NUMÉRICAS DE FUNCIONES NO LINEALES

Convergencia – Newton-Raphson (Ilustración del resultado teórico. TRABAJO PRÁCTICO)

El objetivo de esta práctica es comprobar el orden de convergencia del Método de Newton-Raphson (convergencia cuadrática).

Para ello, consideramos la función $f(x) = x^3 - 3x + 2$, aproximamos la raíz $x_0 = -2$ a partir de la estimación inicial $x_0^0 = -3$ con la tolerancia $\varepsilon = 1 \times 10^{-6}$ mediante el programa elaborado en la Parte 1 de la Práctica 1, y evaluamos:

- el error cometido en la iteración n , $E_n = x_n - x_0$;
- el valor $|E_{n+1}|/|E_n|^k$ para $k = 1$, $k = 2$ y $k = 3$,

para rellenar la tabla siguiente y así poder observar el comportamiento cuando n crece de las columnas con los tres valores de k :

	x_0^0	E_n	$ E_{n+1} / E_n ^k$ $k=1$	$ E_{n+1} / E_n ^k$ $k=2$	$ E_{n+1} / E_n ^k$ $k=3$
$n=0$	-3	?	?	?	?
1	?	?	?	?	?
2	?				
3					
.					
.					
.					
.					
.					

Utilizamos el siguiente programa basado en el programa ya explicado anteriormente:

```
function sol = NewtonRaphson2(f,fd,pto,e,x0)
T=[ " " "Xn" "En" "|En+1|/|En|^1" "|En+1|/|En|^2" "|En+1|/|En|^3" ];
disp(T);
n=1;
y0=pto;
y1=y0-(f(y0)/fd(y0));
T=[ "0" num2str(pto) num2str(Error(x0,y0)) num2str(abs(Error(x0,y1))/abs(Error(x0,y0))) num2str(abs(Error(x0,y1))/abs(Error(x0,y0))^2) num2str(abs(Error(x0,y1))/abs(Error(x0,y0)^3)) ];
disp(T);
while(abs(y1-y0)>e)
y0=y1;
n=n+1;
y1=y0-(f(y0)/fd(y0));
T=[ num2str(n-1) num2str(pto) num2str(Error(x0,y0)) num2str(abs(Error(x0,y1))/abs(Error(x0,y0))) num2str(abs(Error(x0,y1))/abs(Error(x0,y0))^2) num2str(abs(Error(x0,y1))/abs(Error(x0,y0)^3)) ];
disp(T);
end
sol=[y1,"n=",n];
end
```

Primero introducimos los datos:

```
>> f=@(x) x^3-3*x+2
```

f =

[function handle](#) with value:

$\theta(x) x^3 - 3x + 2$

```
>> fd=@(x) 3*x^2-3
```

fd =

[function handle](#) with value:

$\theta(x) 3x^2 - 3$

```
>> pto=-3
```

pto =

-3

```
>> x0=-2
```

x0 =

-2

```
>> e=0.000001
```

e =

1.0000e-06

Al utilizar el programa con los datos anteriores, nos sale el siguiente resultado en tabla, y la última línea donde pone ans, nos da el resultado $x_6 = -2$ y $n = 6$:

```
>> NewtonRaphson2(f,fd,pto,e,x0)
      ""      "Xn"      "En"      "|En+1|/|En|^1"      "|En+1|/|En|^2"      "|En+1|/|En|^3"

      "0"      "-3"      "-1"      "0.33333"      "0.33333"      "0.33333"

      "1"      "-3"      "-0.33333"      "0.16667"      "0.5"      "1.5"      ""

      "2"      "-3"      "-0.055556"      "0.035088"      "0.63158"      "11.3684"      ""

      "3"      "-3"      "-0.0019493"      "0.001297"      "0.66537"      "341.3346"      ""

      "4"      "-3"      "-2.5283e-06"      "1.6855e-06"      "0.66666"      "263679.0893"      ""

      "5"      "-3"      "-4.2615e-12"      "0"      "0"      "0"      ""

ans =

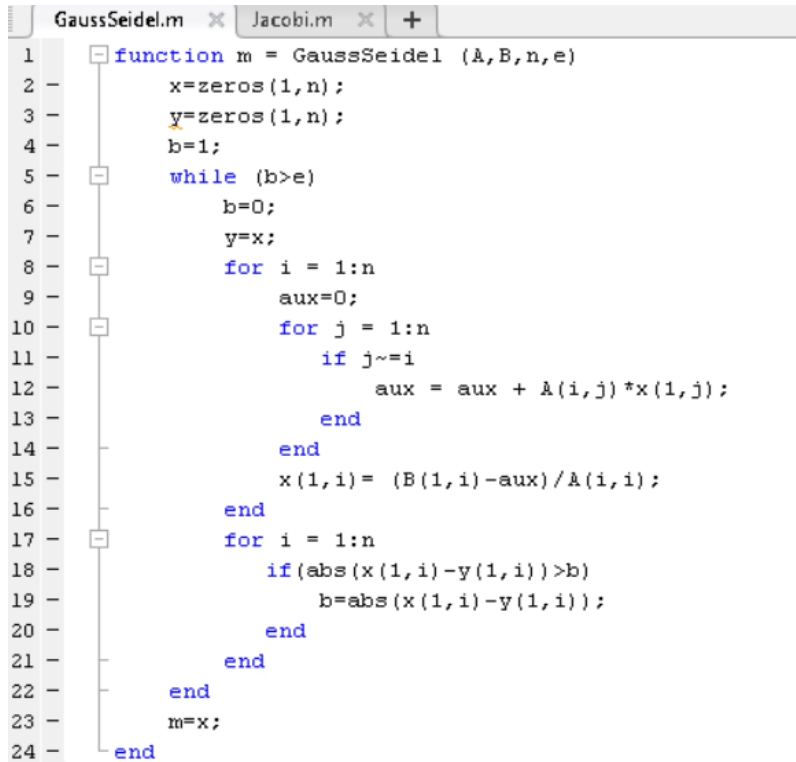
1x3 string array

      "-2"      "n ="      "6"
```


PARTE 4: RESOLUCIÓN NUMÉRICA DE SISTEMAS LINEALES

Programar los métodos de Gauss-Seidel y de Jacobi.

El siguiente código es el correspondiente a el método de Gauss-Seidel para resolución numérica de sistemas lineales, se le debe introducir una matriz "A" y una "B", el número de incógnitas en la variable "n" y la cota de error en la variable "e". El sistema a resolver es de la forma $AX=B$.



```

1  function m = GaussSeidel (A,B,n,e)
2      x=zeros(1,n);
3      y=zeros(1,n);
4      b=1;
5      while (b>e)
6          b=0;
7          y=x;
8          for i = 1:n
9              aux=0;
10             for j = 1:n
11                 if j~=i
12                     aux = aux + A(i,j)*x(1,j);
13                 end
14             end
15             x(1,i)= (B(1,i)-aux)/A(i,i);
16         end
17         for i = 1:n
18             if (abs(x(1,i)-y(1,i))>b)
19                 b=abs(x(1,i)-y(1,i));
20             end
21         end
22     end
23     m=x;
24 end

```

El siguiente código es el correspondiente a el método de Jacobi para resolución numérica de sistemas lineales, se le debe introducir una matriz "A" y una "B", el número de incógnitas en la variable "n" y la cota de error en la variable "e". El sistema a resolver es de la forma $AX=B$.

```

GaussSeidel.m  x  Jacobi.m  x  +
1  function m = Jacobi (A,B,n,e)
2  -      x=zeros(1,n);
3  -      y=zeros(1,n);
4  -      b=1;
5  -      while (b>e)
6  -          b=0;
7  -          y=x;
8  -          for i = 1:n
9  -              aux=0;
10 -             for j = 1:n
11 -                 if j~=i
12 -                     aux = aux + A(i,j)*y(1,j);
13 -                 end
14 -             end
15 -             x(1,i) = (B(1,i)-aux)/A(i,i);
16 -         end
17 -         for i = 1:n
18 -             if(abs(x(1,i)-y(1,i))>b)
19 -                 b=abs(x(1,i)-y(1,i));
20 -             end
21 -         end
22 -     end
23 -     m=x;
24 - end

A =

    3.0000    -0.1000    -0.2000
    0.1000     7.0000    -0.3000
    0.3000    -0.2000    10.0000

>> B=[7.85 -19.3 71.4]

B =

    7.8500   -19.3000    71.4000

>> n=3

n =

     3

>> e=0.5

e =

    0.5000

>> GaussSeidel(A,B,n,e)

ans =

    2.9906   -2.4996     7.0003

>> Jacobi(A,B,n,e)

ans =

    3.0008   -2.4885     7.0064

```

Así es como se debe introducir los datos para el funcionamiento del programa, este ejercicio es el número uno:

$$1) \begin{cases} 3x_1 - 0,1x_2 - 0,2x_3 = 7,85 \\ 0,1x_1 + 7x_2 - 0,3x_3 = -19,3 \\ 0,3x_1 - 0,2x_2 + 10x_3 = 71,4 \end{cases} \text{ por los métodos de: Gauss-Seidel, Jacobi, Gauss-Seidel}$$

~~SOR~~ ($\omega=1,5$), considerando el vector inicial nulo $x_i^0=0$ y una tolerancia que se irá variando. Comparar la solución obtenida con la solución exacta $x_1 = 3$; $x_2 = -2,5$; $x_3 = 7$.

$$2) \begin{cases} 5x_1 + 2x_2 - 1x_3 + 1x_4 = 12 \\ 1x_1 + 7x_2 + 3x_3 - 1x_4 = 2 \\ -1x_1 + 4x_2 + 9x_3 + 2x_4 = 1 \\ 1x_1 - 1x_2 + 1x_3 + 4x_4 = 3 \end{cases} \text{ por los métodos de: Gauss-Seidel, Jacobi, Gauss-Seidel SOR}$$

($\omega=1.5$), considerando el vector inicial nulo $x_i^0=0$ y una tolerancia que se irá variando.

Comparar la solución obtenida con la solución exacta

$x_1 = 2,7273$; $x_2 = -0,4040$; $x_3 = 0,6364$; $x_4 = -0,1919$.

Introducimos inicialmente los datos en las variables de la siguiente manera:

```
>> A = [ 5 2 -1 1; 1 7 3 -1; -1 4 9 2; 1 -1 1 4 ]
```

```
A =
```

```

     5     2    -1     1
     1     7     3    -1
    -1     4     9     2
     1    -1     1     4
```

```
>> B=[12 2 1 3]
```

```
B =
```

```

    12     2     1     3
```

```
>> n=4
```

```
n =
```

```

     4
```

```
>> e = 0.5
```

```
e =
```

```

    0.5000
```

Luego usamos los programas para calcular la solución numérica:

```
>> GaussSeidel(A,B,n,e)
```

```
ans =
```

```

    2.4965   -0.2387    0.4868   -0.0555
```

```
>> Jacobi(A,B,n,e)
```

```
ans =
```

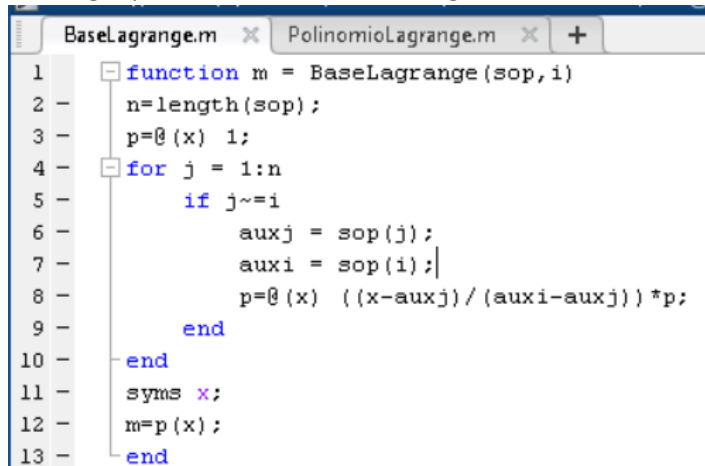
```

    2.3771   -0.0310    0.3068    0.1901
```

PARTE 5: INTERPOLACIÓN NUMÉRICA (LAGRANGE)

1) Programar la expresión de los polinomios de base de Lagrange

El código que vamos a utilizar es el siguiente:

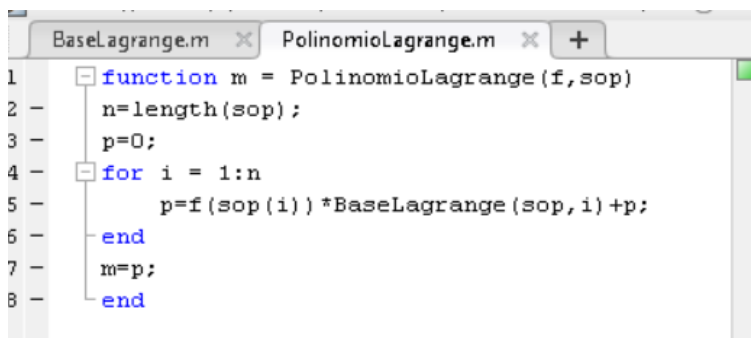


```

1 function m = BaseLagrange(sop,i)
2     n=length(sop);
3     p=@(x) 1;
4     for j = 1:n
5         if j~=i
6             auxj = sop(j);
7             auxi = sop(i);
8             p=@(x) ((x-auxj)/(auxi-auxj))*p;
9         end
10    end
11    syms x;
12    m=p(x);
13 end
  
```

En este primer código, le introducimos todos los valores del soporte en un array en la variable “sop”, y en la variable “i” le introducimos el orden de la base de Lagrange menos uno. Y nos devuelve una base de Lagrange.

2) Programar la expresión del polinomio interpolador de Lagrange



```

1 function m = PolinomioLagrange(f,sop)
2     n=length(sop);
3     p=0;
4     for i = 1:n
5         p=f(sop(i))*BaseLagrange(sop,i)+p;
6     end
7     m=p;
8 end
  
```

En este segundo código, utilizamos el anterior para ir calculando cada una de la bases para ir incluyendo en el polinomio interpolador. Le tenemos que meter una función de la forma $f = @(x) \dots$. También le tenemos que introducir todos los valores del soporte en un array en la variable “sop”. Con un soporte de dos valores, [0,2], y la función, el polinomio interpolador resultante es:

```

>> f=@(x) exp(-x)+cos(4*x/pi)

f =

function_handle with value:

    @(x)exp(-x)+cos(4*x/pi)

>> PolinomioLagrange(f,sop)

ans =

2 - (24254131799228981*x)/18014398509481984
  
```

Si cambiamos el soporte a [0,0.5,1,1.5,2] nos sale la siguiente aproximación:

$$(372146326806111*x*(2*x - 1)*(2*x - 3)*(x - 2))/562949953421312 + (3944195464726781*x*(2*x - 2)*(2*x - 4)*(x - 1/2))/54043195528445952 - (6239733289746997*x*(2*x - 3)*((2*x)/3 - 1/3)*(x -$$

$$1)) / 18014398509481984 - (6352959975846737 * x * (2 * x - 2) * ((2 * x) / 3 - 4 / 3) * (x - 3 / 2)) / 2251799813685248 + 2 * (2 * x - 1) * (x / 2 - 1) * ((2 * x) / 3 - 1) * (x - 1)$$

Si cambiamos el soporte a uno que vaya desde 0 hasta 2 con un paso de 0.1, nos sale la aproximación:

$$\begin{aligned} & 2 * (2 * x - 1) * (x / 2 - 1) * ((2 * x) / 3 - 1) * (5 * x - 1) * ((5 * x) / 2 - 1) * ((5 * x) / 3 - 1) * ((5 * x) / 4 - 1) * (10 * x - 1) * ((5 * x) / 6 - 1) * ((5 * x) / 7 - 1) * ((5 * x) / 8 - 1) * ((10 * x) / 3 - 1) * ((5 * x) / 9 - 1) * ((10 * x) / 7 - 1) * ((10 * x) / 9 - 1) * ((10 * x) / 11 - 1) * ((10 * x) / 13 - 1) * ((10 * x) / 17 - 1) * ((10 * x) / 19 - 1) * (x - 1) + (30992864632301945 * x * ((5 * x) / 2 - 2) * ((5 * x) / 3 - 1) * (5 * x - 5) * ((5 * x) / 2 - 4) * ((5 * x) / 3 - 3) * (5 * x - 7) * ((5 * x) / 4 - 1 / 2) * (2 * x - 7 / 5) * ((5 * x) / 4 - 5 / 2) * ((10 * x) / 3 - 3) * ((10 * x) / 3 - 5) * (10 * x - 11) * (10 * x - 13) * ((10 * x) / 9 - 1 / 3) * (2 * x - 17 / 5) * ((10 * x) / 7 - 5 / 7) * ((10 * x) / 11 - 1 / 11) * ((10 * x) / 7 - 19 / 7) * (x - 1 / 5)) / 108086391056891904 + \\ & (372146326806111 * x * (2 * x - 1) * (2 * x - 3) * (5 * x - 4) * (5 * x - 6) * ((5 * x) / 2 - 3 / 2) * ((5 * x) / 3 - 2 / 3) * ((5 * x) / 4 - 1 / 4) * ((5 * x) / 2 - 7 / 2) * ((5 * x) / 3 - 8 / 3) * (10 * x - 9) * (10 * x - 11) * ((5 * x) / 4 - 9 / 4) * ((10 * x) / 3 - 7 / 3) * ((10 * x) / 7 - 3 / 7) * ((10 * x) / 3 - 13 / 3) * ((10 * x) / 9 - 1 / 9) * ((10 * x) / 7 - 17 / 7) * ((10 * x) / 9 - 19 / 9) * (x - 2)) / 562949953421312 + \\ & (1192445375670355 * x * (5 * x - 2) * (2 * x - 1 / 5) * (5 * x - 4) * ((5 * x) / 2 - 1 / 2) * ((5 * x) / 3 - 2) * ((5 * x) / 2 - 5 / 2) * ((10 * x) / 3 - 1) * (10 * x - 5) * ((5 * x) / 6 - 3 / 2) * ((10 * x) / 3 - 3) * (10 * x - 7) * (2 * x - 11 / 5) * ((5 * x) / 4 - 7 / 4) * ((10 * x) / 9 - 5 / 3) * ((5 * x) / 7 - 10 / 7) * ((10 * x) / 7 - 13 / 7) * ((10 * x) / 11 - 17 / 11) * ((10 * x) / 13 - 19 / 13) * (x - 8 / 5)) / 562949953421312 + (10965076979671585 * x * (5 * x - 3) * ((5 * x) / 2 - 1) * (2 * x - 3 / 5) * (5 * x - 5) * ((5 * x) / 2 - 3) * ((5 * x) / 4 - 2) * ((5 * x) / 3 - 1 / 3) * (10 * x - 7) * ((5 * x) / 3 - 7 / 3) * (10 * x - 9) * ((5 * x) / 6 - 5 / 3) * (2 * x - 13 / 5) * ((10 * x) / 3 - 5 / 3) * ((10 * x) / 7 - 1 / 7) * ((10 * x) / 3 - 11 / 3) * ((10 * x) / 7 - 15 / 7) * ((10 * x) / 9 - 17 / 9) * ((10 * x) / 11 - 19 / 11) * (x - 9 / 5)) / 9007199254740992 + (3944195464726781 * x * (2 * x - 2) * (2 * x - 4) * ((5 * x) / 3 - 3 / 2) * ((5 * x) / 6 - 1 / 4) * ((10 * x) / 3 - 4) * ((10 * x) / 3 - 6) * (5 * x - 13 / 2) * ((5 * x) / 2 - 11 / 4) * (5 * x - 17 / 2) * ((5 * x) / 4 - 7 / 8) * (10 * x - 14) * ((10 * x) / 9 - 2 / 3) * (10 * x - 16) * ((5 * x) / 7 - 1 / 14) * ((5 * x) / 2 - 19 / 4) * ((10 * x) / 7 - 8 / 7) * ((10 * x) / 11 - 4 / 11) * ((10 * x) / 13 - 2 / 13) * (x - 1 / 2)) / 54043195528445952 + \\ & (820478150644545 * x * (5 * x - 6) * (5 * x - 8) * ((5 * x) / 2 - 5 / 2) * ((5 * x) / 3 - 4 / 3) * (2 * x - 9 / 5) * ((5 * x) / 4 - 3 / 4) * ((5 * x) / 2 - 9 / 2) * ((5 * x) / 6 - 1 / 6) * ((10 * x) / 7 - 1) * ((5 * x) / 3 - 10 / 3) * (10 * x - 13) * (10 * x - 15) * (2 * x - 19 / 5) * ((10 * x) / 3 - 11 / 3) * ((10 * x) / 3 - 17 / 3) * ((10 * x) / 9 - 5 / 9) * ((10 * x) / 11 - 3 / 11) * ((10 * x) / 13 - 1 / 13) * (x - 2 / 5)) / 31525197391593472 - (44633470468070265 * x * ((5 * x) / 2 - 3) * ((5 * x) / 4 - 1) * (5 * x - 7) * ((5 * x) / 2 - 5) * (5 * x - 9) * ((5 * x) / 6 - 1 / 3) * ((5 * x) / 3 - 5 / 3) * (2 * x - 11 / 5) * ((5 * x) / 7 - 1 / 7) * ((2 * x) / 3 - 1 / 15) * (10 * x - 15) * (10 * x - 17) * ((10 * x) / 3 - 13 / 3) * ((10 * x) / 7 - 9 / 7) * ((10 * x) / 3 - 19 / 3) * ((10 * x) / 9 - 7 / 9) * ((10 * x) / 11 - 5 / 11) * ((10 * x) / 13 - 3 / 13) * (x - 3 / 5)) / 288230376151711744 - (18437326707188215 * x * (2 * x - 4 / 5) * ((5 * x) / 3 - 1 / 2) * (5 * x - 7 / 2) * ((5 * x) / 3 - 5 / 2) * ((10 * x) / 3 - 2) * ((5 * x) / 2 - 5 / 4) * ((10 * x) / 3 - 4) * (5 * x - 11 / 2) * ((5 * x) / 4 - 1 / 8) * (10 * x - 8) * (10 * x - 10) * (2 * x - 14 / 5) * ((10 * x) / 9 - 2) * ((5 * x) / 2 - 13 / 4) * ((10 * x) / 7 - 2 / 7) * ((5 * x) / 4 - 17 / 8) * ((10 * x) / 7 - 16 / 7) * ((10 * x) / 11 - 20 / 11) * (x - 19 / 10)) / 20266198323167232 - \\ & (5571667004306525 * x * ((5 * x) / 3 - 2) * ((2 * x) / 3 - 1 / 5) * (5 * x - 8) * ((5 * x) / 6 - 1 / 2) * (5 * x - 10) * ((5 * x) / 2 - 7 / 2) * ((5 * x) / 4 - 5 / 4) * ((10 * x) / 3 - 5) * (2 * x - 13 / 5) * ((10 * x) / 9 - 1) * ((5 * x) / 7 - 2 / 7) * ((5 * x) / 8 - 1 / 8) * (10 * x - 17) * (10 * x - 19) * ((10 * x) / 7 - 11 / 7) * ((10 * x) / 11 - 7 / 11) * ((10 * x) / 13 - 5 / 13) * ((10 * x) / 17 - 1 / 17) * (x - 4 / 5)) / 20266198323167232 + (8688615196656625 * x * (5 * x - 1) * (5 * x - 3) * ((5 * x) / 2 - 2) * (10 * x - 3) * ((5 * x) / 4 - 3 / 2) * (10 * x - 5) * (2 * x - 9 / 5) * ((5 * x) / 3 - 5 / 3) * ((10 * x) / 3 - 1 / 3) * ((5 * x) / 6 - 4 / 3) * ((5 * x) / 8 - 5 / 4) * ((10 * x) / 3 - 7 / 3) * ((5 * x) / 7 - 9 / 7) * ((10 * x) / 7 - 11 / 7) * ((2 * x) / 3 - 19 / 15) * ((10 * x) / 9 - 13 / 9) * ((10 * x) / 11 - 15 / 11) * ((10 * x) / 13 - 17 / 13) * (x - 7 / 5)) / 2251799813685248 - (6352959975846737 * x * (2 * x - 2) * (5 * x - 3 / 2) * ((2 * x) / 3 - 4 / 3) * ((5 * x) / 2 - 1 / 4) * (5 * x - 7 / 2) * (10 * x - 4) * (10 * x - 6) * ((10 * x) / 3 - 2 / 3) * ((5 * x) / 2 - 9 / 4) * ((10 * x) / 3 - 8 / 3) * ((5 * x) / 3 - 11 / 6) * ((5 * x) / 4 - 13 / 8) * ((10 * x) / 7 - 12 / 7) * ((5 * x) / 6 - 17 / 12) * ((10 * x) / 9 - 14 / 9) * ((5 * x) / 7 - 19 / 14) * ((10 * x) / 11 - 16 / 11) * ((10 * x) / 13 - 18 / 13) * (x - 3 / 2)) / 2251799813685248 - \\ & (6239733289746997 * x * (2 * x - 3) * ((2 * x) / 3 - 1 / 3) * ((5 * x) / 2 - 4) * (5 * x - 9) * ((5 * x) / 4 - 3 / 2) * ((5 * x) / 6 - 2 / 3) * ((5 * x) / 3 - 7 / 3) * ((5 * x) / 8 - 1 / 4) * ((5 * x) / 7 - 3 / 7) * ((5 * x) / 9 - 1 / 9) * (10 * x - 19) * ((10 * x) / 3 - 17 / 3) * ((10 * x) / 7 - 13 / 7) * ((10 * x) / 9 - 11 / 9) * ((10 * x) / 11 - 9 / 11) * ((10 * x) / 13 - 7 / 13) * ((10 * x) / 17 - 3 / 17) * ((10 * x) / 19 - 1 / 19) * (x - 1)) / 18014398509481984 - (176725084298555 * x * (2 * x - 6 / 5) * (5 * x - 9 / 2) * ((5 * x) / 2 - 7 / 4) * ((5 * x) / 3 - 5 / 6) * (5 * x - 13 / 2) * ((5 * x) / 4 - 3 / 8) * (10 * x - 10) * (10 * x - 12) * (2 * x - 16 / 5) * ((10 * x) / 3 - 8 / 3) * ((5 * x) / 2 - 15 / 4) * ((10 * x) / 7 - 4 / 7) * ((10 * x) / 3 - 14 / 3) * ((10 * x) / 9 - 2 / 9) * ((5 * x) / 3 - 17 / 6) * ((5 * x) / 4 - 19 / 8) * ((10 * x) / 7 - 18 / 7) * ((10 * x) / 9 - 20 / 9) * (x - 1 / 10)) / 387028092977152 - \end{aligned}$$

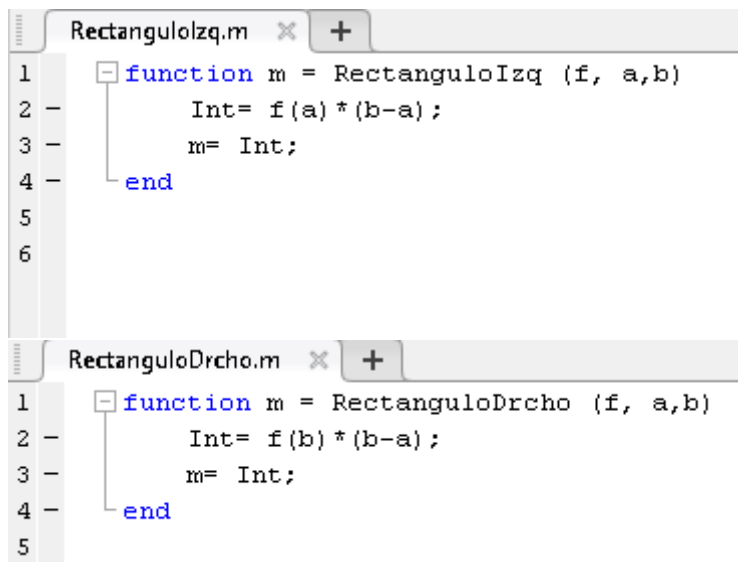
$$\begin{aligned}
& (3166626548798675 * x * (2 * x - 2/5) * (5 * x - 5/2) * ((5 * x)/2 - 3/4) * ((5 * x)/3 - 1/6) * (5 * x - 9/2) * (10 * x - \\
& 6) * (10 * x - 8) * (2 * x - 12/5) * ((10 * x)/7 - 2) * ((10 * x)/3 - 4/3) * ((5 * x)/2 - 11/4) * ((10 * x)/3 - 10/3) * ((5 * x)/3 - \\
& 13/6) * ((5 * x)/4 - 15/8) * ((5 * x)/6 - 19/12) * ((10 * x)/9 - 16/9) * ((10 * x)/11 - 18/11) * ((10 * x)/13 - 20/13) * (x - \\
& 17/10)) / 1970324836974592 - (33906148123244585 * x * (2 * x - 8/5) * (5 * x - 11/2) * ((5 * x)/2 - \\
& 9/4) * ((5 * x)/3 - 7/6) * (5 * x - 15/2) * ((5 * x)/4 - 5/8) * (10 * x - 12) * (10 * x - 14) * ((5 * x)/6 - 1/12) * (2 * x - \\
& 18/5) * ((10 * x)/3 - 10/3) * ((5 * x)/2 - 17/4) * ((10 * x)/7 - 6/7) * ((10 * x)/3 - 16/3) * ((10 * x)/9 - 4/9) * ((5 * x)/3 - \\
& 19/6) * ((10 * x)/11 - 2/11) * ((10 * x)/7 - 20/7) * (x - 3/10)) / 234187180623265792 + \\
& (20114008205015035 * x * (5 * x - 2) * (10 * x - 1) * ((5 * x)/2 - 3/2) * (10 * x - 3) * (2 * x - 7/5) * ((5 * x)/3 - \\
& 4/3) * ((5 * x)/4 - 5/4) * ((10 * x)/3 - 5/3) * ((5 * x)/6 - 7/6) * ((5 * x)/7 - 8/7) * ((5 * x)/8 - 9/8) * ((5 * x)/9 - \\
& 10/9) * ((10 * x)/7 - 9/7) * ((2 * x)/3 - 17/15) * ((10 * x)/9 - 11/9) * ((10 * x)/11 - 13/11) * ((10 * x)/13 - \\
& 15/13) * ((10 * x)/17 - 19/17) * (x - 6/5)) / 2251799813685248 - (37576901437273555 * x * (5 * x - 1/2) * (5 * x - \\
& 5/2) * (10 * x - 2) * ((5 * x)/3 - 3/2) * (10 * x - 4) * (2 * x - 8/5) * ((10 * x)/3 - 2) * ((2 * x)/3 - 6/5) * ((5 * x)/2 - \\
& 7/4) * ((5 * x)/6 - 5/4) * ((10 * x)/9 - 4/3) * ((5 * x)/4 - 11/8) * ((10 * x)/7 - 10/7) * ((5 * x)/7 - 17/14) * ((10 * x)/11 - \\
& 14/11) * ((5 * x)/8 - 19/16) * ((10 * x)/13 - 16/13) * ((10 * x)/17 - 20/17) * (x - 13/10)) / 6755399441055744 + \\
& (2120954471629175 * x * (2 * x - 12/5) * ((2 * x)/3 - 2/15) * (5 * x - 15/2) * ((5 * x)/2 - 13/4) * ((5 * x)/3 - \\
& 11/6) * (5 * x - 19/2) * ((5 * x)/4 - 9/8) * (10 * x - 16) * (10 * x - 18) * ((5 * x)/6 - 5/12) * ((5 * x)/7 - 3/14) * ((5 * x)/8 - \\
& 1/16) * ((10 * x)/3 - 14/3) * ((10 * x)/7 - 10/7) * ((10 * x)/3 - 20/3) * ((10 * x)/9 - 8/9) * ((10 * x)/11 - \\
& 6/11) * ((10 * x)/13 - 4/13) * (x - 7/10)) / 9570149208162304 + (27049863061981845 * x * (2 * x - \\
& 14/5) * ((2 * x)/3 - 4/15) * (5 * x - 17/2) * ((5 * x)/2 - 15/4) * ((5 * x)/3 - 13/6) * ((5 * x)/4 - 11/8) * (10 * x - 18) * (10 * x \\
& - 20) * ((5 * x)/6 - 7/12) * ((5 * x)/7 - 5/14) * ((5 * x)/8 - 3/16) * ((10 * x)/3 - 16/3) * ((5 * x)/9 - 1/18) * ((10 * x)/7 - \\
& 12/7) * ((10 * x)/9 - 10/9) * ((10 * x)/11 - 8/11) * ((10 * x)/13 - 6/13) * ((10 * x)/17 - 2/17) * (x - \\
& 9/10)) / 85568392920039424 - (5338855982479255 * x * (5 * x - 3/2) * (10 * x - 2) * (2 * x - 6/5) * ((5 * x)/2 - \\
& 5/4) * ((10 * x)/3 - 4/3) * ((5 * x)/3 - 7/6) * ((5 * x)/4 - 9/8) * ((10 * x)/7 - 8/7) * ((2 * x)/3 - 16/15) * ((5 * x)/6 - \\
& 13/12) * ((10 * x)/9 - 10/9) * ((5 * x)/7 - 15/14) * ((10 * x)/11 - 12/11) * ((5 * x)/8 - 17/16) * ((10 * x)/13 - \\
& 14/13) * ((5 * x)/9 - 19/18) * ((10 * x)/17 - 18/17) * ((10 * x)/19 - 20/19) * (x - 11/10)) / 281474976710656
\end{aligned}$$

PARTE 6: INTEGRACIÓN NUMÉRICA

1) Programar las 3 fórmulas del Rectángulo en Matlab

Las 3 fórmulas del rectángulo son: Rectángulo con el punto en el extremo izquierdo, rectángulo con el punto en el extremo derecho, rectángulo con el punto en el punto medio.

A cada función *RectanguloIzq*, *RectanguloDrcho*, *RectanguloMedio* se la introducen f , a , b , siendo f la función correspondiente a la integral, y a , b los puntos de la integral. Todas las funciones devuelven m que es el resultado de la integral.



```

RectanguloIzq.m
1 function m = RectanguloIzq (f, a,b)
2     Int= f(a) *(b-a) ;
3     m= Int;
4 end

RectanguloDrcho.m
1 function m = RectanguloDrcho (f, a,b)
2     Int= f(b) *(b-a) ;
3     m= Int;
4 end
  
```

```

RectanguloMedio.m  x  +
1  function m = RectanguloMedio (f, a,b)
2  -     Int= f((a+b)/2)*(b-a);
3  -     m= Int;
4  - end

```

2) Programar la fórmula del Trapecio en Matlab

A la función *Trapecio* se la introducen f , a , b , siendo f la función correspondiente a la integral, y a , b los puntos de la integral. Todas las funciones devuelven m que es el resultado de la integral.

```

Trapecio.m  x  +
1  function m = Trapecio (f, a,b)
2  -     Int= (b-a)*((f(a)+f(b))/2);
3  -     m= Int;
4  - end

```

3) Programar la fórmula de Simpson 1/3 en Matlab

A la función *Simpson13* se la introducen f , a , b , siendo f la función correspondiente a la integral, y a , b los puntos de la integral. Todas las funciones devuelven m que es el resultado de la integral.

```

Simpson13.m  x  +
1  function m = Simpson13 (f, a,b)
2  -     Int= ((b-a)/6)*(f(a)+4*f((a+b)/2)+f(b));
3  -     m= Int;
4  - end
5

```

$$\int_0^{2\pi} \cos(x^2 - 1) dx.$$

Aplicar los programas desarrollados al cálculo de la integral

Introducimos por consola:

```

>> f= @(x) cos((x.^2)-1)

f =

function_handle with value:

    @(x) cos((x.^2)-1)

>> a=0

a =

    0

>> b=2*pi

b =

    6.2832

```

Rectángulo Extremo Izquierdo:


```
>> RectanguloIzq(f,a,b)

ans =

    3.3948
```

Rectángulo Extremo Derecho:

```
>> RectanguloDrcho(f,a,b)

ans =

    4.4699
```

Rectángulo Punto Medio:

```
>> RectanguloMedio(f,a,b)

ans =

   -5.3395
```

Trapezio:

```
>> Trapecio(f,a,b)

ans =

    3.9323
```

Simpson 1/3:

```
>> Simpson13(f,a,b)

ans =

   -2.2489
```

Comparar los resultados con el resultado de referencia I_r obtenido empleando el comando `trapz(x,f)` o el comando `@(x)` (error relativo $|I-I_r|/I_r$)

```
>> x = linspace(0,2*pi,2000);
>> y = cos((x.^2)-1);
>> a = trapz(x,y);
>> a

a =

    0.9211
```

Como se puede comprobar, las soluciones aproximadas obtenidas por nuestros programas son bastante dispares unas de otras y, además, están bastante lejanas de la solución obtenida por la función “trapz”. Esto se debe a que todas son aproximaciones o cálculos numéricos de las integrales y no cálculos analíticos de la integral, y en este caso se nota bastante ya que la función oscila y según nos vamos separando del $x=0$, la velocidad a la que oscila va aumentando rápidamente, esto da lugar a que una aproximación en la que cogemos solo una cierta cantidad de puntos, puedan no ser suficientes para que la solución obtenida se acerque mucho a la real.

▪ **Pregunta optativa:**

Deducir la fórmula de Milne compuesta y la expresión de su error. Programar la fórmula de Milne compuesta en Matlab.

Formula de Milne Compuesta:

$$h = \frac{a+b}{4};$$

$$\int_a^b f(x) dx = \int_a^{x_1} f(x) dx + \dots + \int_{x_n}^b f(x) dx = \frac{(x_1-a)}{90} \sum_{i=0}^4 c_i f(x_i) + \dots + \frac{(b-x_n)}{90} \sum_{i=0}^4 c_i f(x_i);$$

Expresión de su Error:

$$R_4(x) = Kh^7 f^{(6)}(\epsilon_x);$$

$$\text{Siendo } K = 8/945 \text{ O}(h^7);$$

Para programar la formula de Milne hemos utilizado dos funciones: *MilneCompuesta* y *MilneNoCompuesta*.

MilneCompuesta es una función a la que se le introduce f , a , b , n . Siendo f la función de la integral, a y b los puntos extremos de la integral, y n los subintervalos de la integral. Esta función nos devuelve m que es el resultado de la integral utilizando *MilneCompuesta* en n subintervalos.

MilneNoCompuesta es una función a la que se le introduce f , a , b . Siendo f la función de la integral, y a y b los puntos extremos de la integral. Esta función nos devuelve m que es el resultado de la formula de Milne en el subintervalo dado.

```

1 function m = MilneCompuesta(f, a,b, n)
2     aux=b-a;
3     subinterv= aux/n;
4     result=0;
5     auxa=a;
6     auxb=b;
7     for i=auxa:subinterv:auxb
8         a=i;
9         b=i+subinterv;
10        result=result+ MilneNoCompuesta(f,a,b);
11    end
12
13    m= result;
14
15 end
16

1 function m = MilneNoCompuesta(f, a,b)
2     h= (b-a)/4;
3     Int1= ((b-a)/90) * ( 7*f(a) )+(32*f(a+h) )+(12*f(a+h+h) )+(32*f(a+h+h+h) )+(7*f(a+h+h+h+h) ) ;
4     m=Int1;
5 end
  
```

$$\int_1^3 (\cos(x) - x \sin(x)) dx$$

Aplicar el programa desarrollado al cálculo de la integral
intervalo [1, 3] en N sub-intervalos y dando a N los valores: 2, 4, 8, 16, 32, ...

dividiendo el

Introducimos por consola:

```
>> f= @(x) cos(x)-x.*sin(x)

f =

    function_handle with value:

    @(x) cos(x)-x.*sin(x)

>> a=1

a =

    1

>> b=3

b =

    3
```

Para N=2:

```
>> n=2

n =

    2

>> MilneCompuesta(f,a,b,n)

ans =

   -3.1549
```

Para N=4:

```
>> n=4

n =

    4

>> MilneCompuesta(f,a,b,n)

ans =

   -3.8179
```

Para N=8:

```
>> n=8  
  
n =  
  
      8  
  
>> MilneCompuesta(f,a,b,n)  
  
ans =  
  
    -3.7712
```

Para N=16:

```
>> n=16  
  
n =  
  
    16  
  
>> MilneCompuesta(f,a,b,n)  
  
ans =  
  
    -3.6649
```

Para N=32:

```
>> n=32  
  
n =  
  
    32  
  
>> MilneCompuesta(f,a,b,n)  
  
ans =  
  
    -3.5932
```

Para N=64:

```
>> n=64  
  
n =  
  
    64  
  
>> MilneCompuesta(f,a,b,n)  
  
ans =  
  
    -3.5531
```

Para N= 128:

```
>> n=128  
  
n =  
  
    128  
  
>> MilneCompuesta(f,a,b,n)  
  
ans =  
  
   -3.5320
```