Asignatura	Datos del alumno	Fecha
Optimización	**Apellidos:** Avilés Cahill	27/02/2024
	Nombre: Adán	

Laboratorio: Optimización de funciones unidimensionales Laboratorio: Optimización de funciones unidimensionales

Índice

Asignatura	Datos del alumno	Fecha
Optimización	**Apellidos:** Avilés Cahill	27/02/2024
	Nombre: Adán	

1. Comparativa entre métodos

En este trabajo, hemos de integrar en Matlab diferentes métodos para aproximar el mínimo de una función. En el anexo, encontraremos el código usado en cada uno de los métodos. Para comprobar la efectividad de los métodos, serán puestos a prueba con una serie de funciones, además de plotear los resultados y comentar ciertos aspectos que se han ido viendo.

1.1. Caso A

Función a utilizar:

$$f(x) = x^2 + 2x - 1, \quad x \in (-\infty, \infty)$$

En este caso, podemos coger los puntos que queramos $a, b \in (-\infty, \infty)$ para la búsqueda del mínimo. Como sabemos la forma de la función, probaremos con $a = -3$, $b = 3$, exigiendo una tolerancia $tol = 1 \cdot 10^{-5}$.

```
f=@(x) x.^2+2*x-1;
xplot = linspace(-3,3);
yplot = f(xplot);
a=-3;
b=3;
x1=(a+b)/2;
tol=1e-8;
maxiter=100000;
[x_incu,t_incu,iter_incu] = incu(f,a,b,tol,maxiter) ;
[x_budi,t_budi, iter_budi] = budi(f,a,b,tol,maxiter);
syms x
funcion = x^2+2*x+1;
[x_rein,t_rein, iter_rein, cambios_alfa] = rein(funcion,x1,tol, maxiter);
[x_new,t_new, iter_new] = new(funcion,x1,tol,maxiter);
```

```
plot(xplot,yplot, ...
    x_incu, f(x_incu), 'r*', x_budi, f(x_budi), 'g*', ...
     x_rein, f(x_rein), 'b*', x_new, f(x_new), 'y*')
legend('function','incu', 'budi', 'rein', 'new')
```

Podemos ver en la gráfica que todos los puntos coinciden.

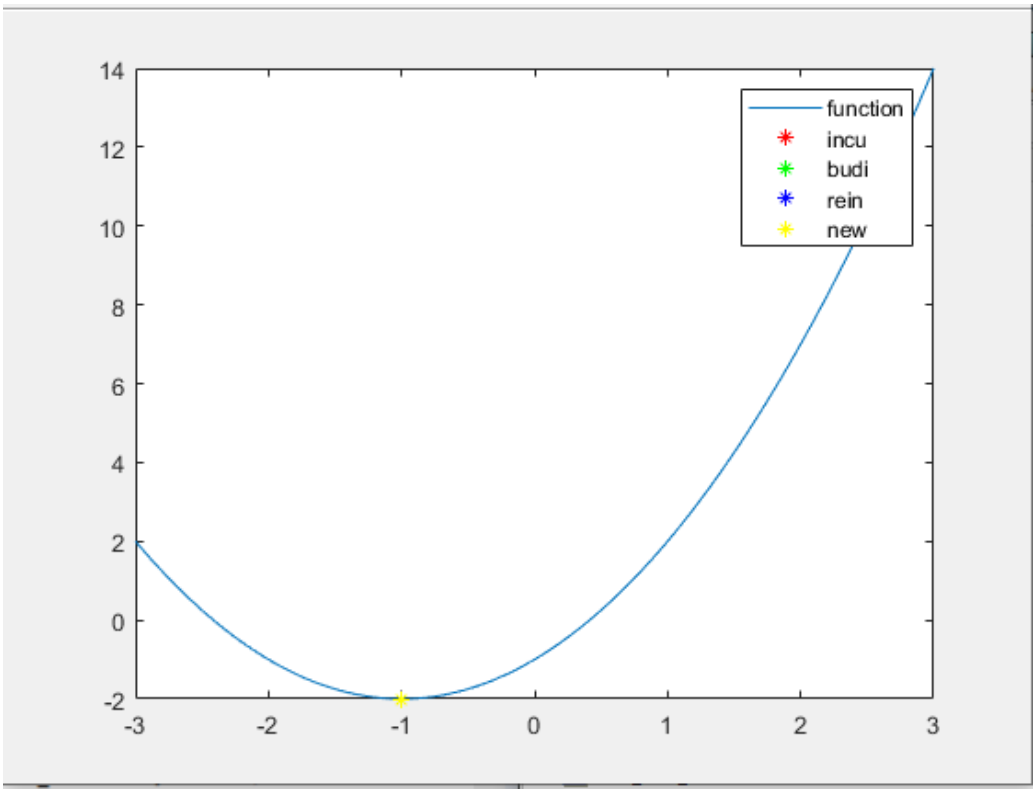

Figura 1: Gráfica del caso A

Para generar la tabla de resultados, utilizaremos el código siguiente (usaremos el mismo en los cuatro casos, así que solo se mostrará una vez)

```
legends = ["incu"; "budi"; "rein" ;"new"];
x_values = [x_incu, x_budi,x_rein,x_new]';
y_values = [f(x_incu), f(x_budi), f(x_rein),f(x_new)]';
time_seconds = [t_incu, t_budi,t_rein,t_new]';
```

```
iters = [iter_incu, iter_budi,iter_rein,iter_new]';
table(legends, x_values, y_values, time_seconds, iters)
```

Que nos devuelve

legends	x_values	y_values	times	iters
"incu"	-1	-2	0.0002964	2
"budi"	-0.999999999997294	-2	0.009523	100000
"rein"	-1	-2	0.0613942	1
"new"	-1	-2	0.0513656	2

Figura 2: Tabla de resultados del caso A

Como podemos observar, todos los métodos llegan a una situación muy parecida, siendo la interpolación cuadrática el más rápido de ellos, pero teniendo más iteraciones que las rectas inexactas. Los pasos tan pequeños que (hemos usado en *budi* $epsilon = 0{,}005$) dan lugar a un número de iteraciones muy elevado.

1.2. Caso B

Función a utilizar:

$$f(x) = \sin(x) + x^2, \quad x \in (-\infty, \infty)$$

Como en el caso anterior, podemos coger los puntos que queramos $a, b \in (-\infty, \infty)$ para la búsqueda del mínimo. Probaremos con $a = -3$, $b = 3$, exigiendo una tolerancia $tol = 1 \cdot 10^{-5}$.

```
f=@(x) sin(x)+x.^2;
xplot = linspace(-0.45018360,-0.45018362);
yplot = f(xplot);
a=-1;
b=1;
```

```
x1=(a+b)/2;
tol=1e-8;
maxiter=100000;

[x_incu,t_incu,iter_incu] = incu(f,a,b,tol,maxiter) ;
[x_budi,t_budi, iter_budi] = budi(f,a,b,tol,maxiter);

syms x
funcion = sin(x)+x^2;
[x_rein,t_rein, iter_rein, cambios_alfa] = rein(funcion,x1,tol, maxiter);
[x_new,t_new, iter_new] = new(funcion,x1,tol,maxiter);
```

Si ploteamos los resultados, veremos

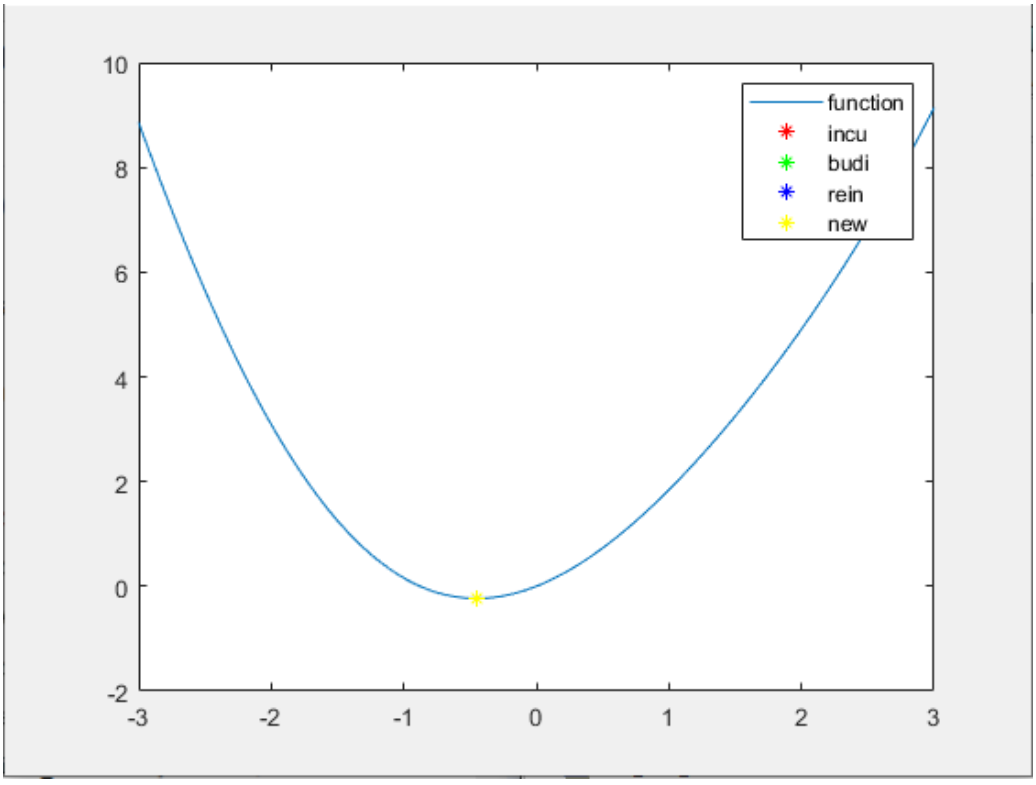

Figura 3: Gráfica del caso B

con tabla de resultados

legends	x_values	y_values	time_seconds	iters
"incu"	-0.450183609831387	-0.232465575158216	0.0018606	12
"budi"	-0.450183611288684	-0.232465575158216	0.0133968	100000
"rein"	-0.450183615088463	-0.232465575158216	1.0448071	10
"new"	-0.450183611294874	-0.232465575158216	0.180731	5

Figura 4: Tabla de resultados del caso B

Aquí podemos observar que el tiempo se dispara en las rectas inexactas, y que, como antes, el método más rápido es el de interpolación cuadrática, pero el que menos iteraciones requiere es el de Newton. Podemos hacer una ampliación del plot para ver que aunque los puntos sean muy cercanos, no son coincidentes.

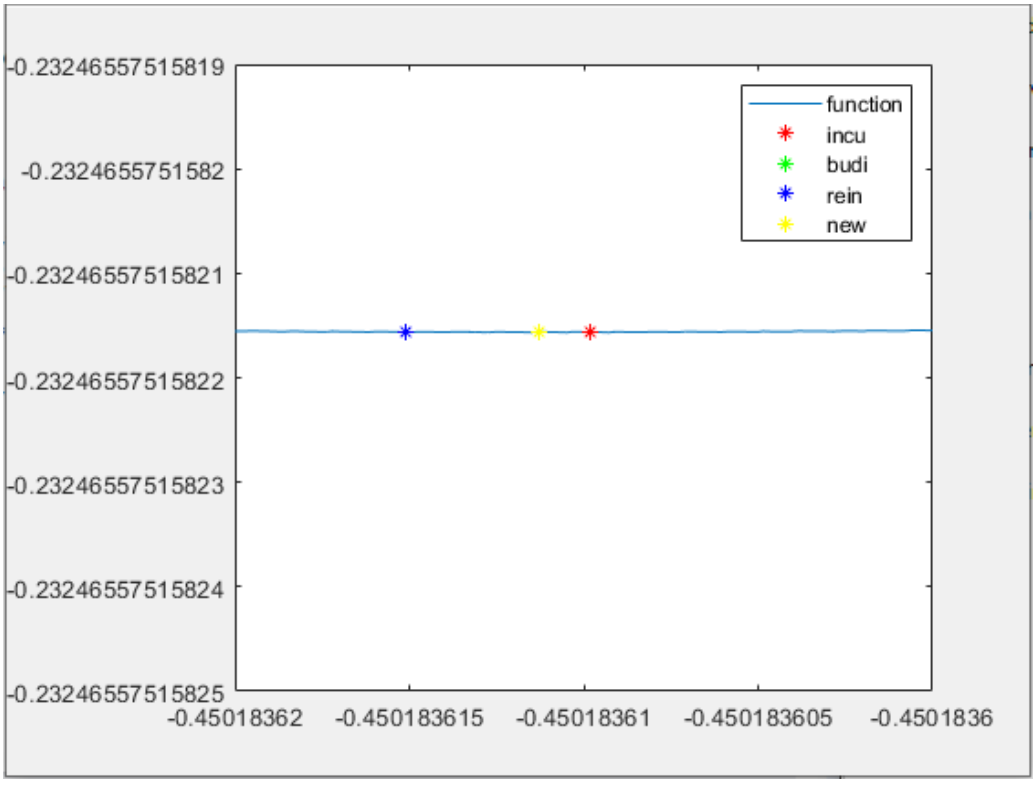

Figura 5: Ampliación de la gráfica de C

1.3. Caso C

Función a utilizar:

$$f(x) = \cos(x) - \ln(x), \quad x \in (0, 5]$$

Hemos de tener cuidado ahora, pues la función no está definida en el 0, así pues, probaremos con $a = 1 \cdot *10^{-3}$, $b = 5$, teniendo a muy próximo al 0

```
f=@(x) cos(x)-log(x)
xplot = linspace(0,6);
yplot = f(xplot);
a=0.001;
b=5;
x1=(a+b)/2;
tol=1e-6;
maxiter=100000;

[x_incu,t_incu,iter_incu] = incu(f,a,b,tol,maxiter) ;
[x_budi,t_budi, iter_budi] = budi(f,a,b,tol,maxiter);

syms x
funcion = cos(x)-log(x);
[x_rein,t_rein, iter_rein, cambios_alfa] = rein(funcion,x1,tol, maxiter);
[x_new,t_new, iter_new] = new(funcion,x1,tol,maxiter);
```

Si ploteamos los resultados, veremos

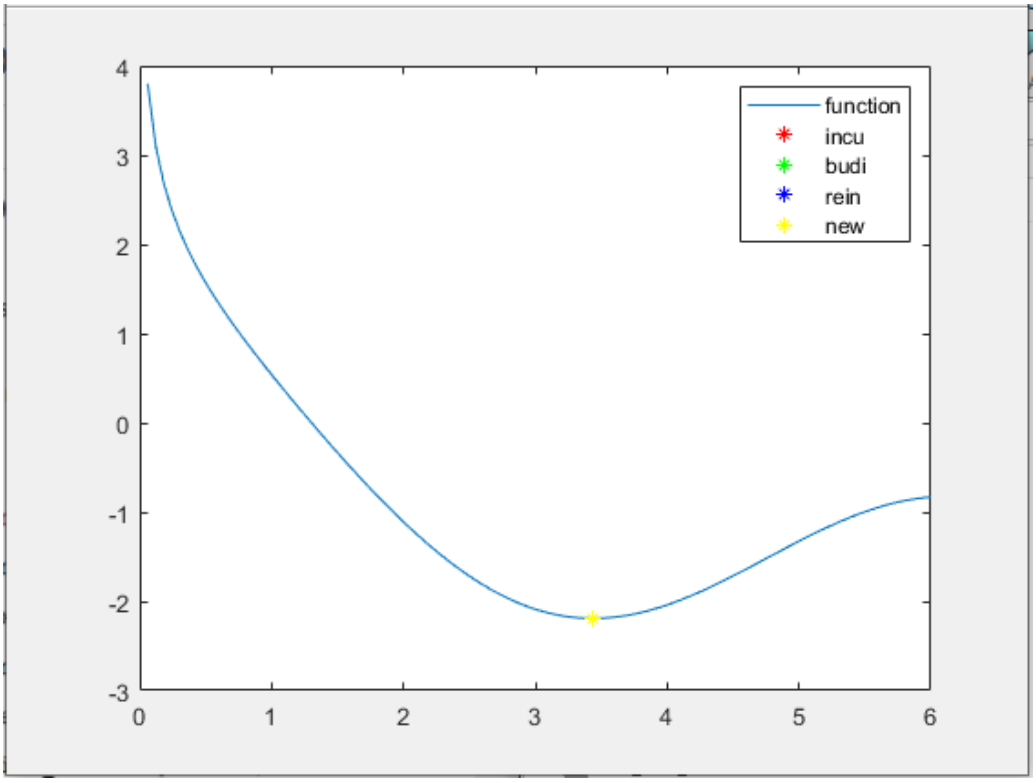

Figura 6: Gráfica del caso C

Con tabla de resultados:

legends	x_values	y_values	time_seconds	iters
"incu"	3.43682901068556	-2.19128264421716	0.0083661	9
"budi"	3.43780875	-2.19128214435852	0.0215106	100000
"rein"	3.43682888793945	-2.19128264421716	0.7982933	8
"new"	3.4368289123266	-2.19128264421716	0.1433965	4

Figura 7: Tabla de resultados del caso C

1.4. Caso D

En este caso usaremos una una función definida por código Matlab (y por su definición, , $x \in [1, 4]$) , así que donde necesitemos la derivada tendremos que editar un poco nuestras funciones. Se nos da un fichero $funcionrr$ que devuelve el resultado, en este caso, hemos creado otro llamado $derivada\_funcionrr$ que nos devuelve el resultado de la derivada.

```
function df = derivada_funcionrr ( x )
    xini=[x 0];
    t=[0 2*pi];
    options=odeset('RelTol',1e-13,'AbsTol',1e-13);
    [ts,xs]=ode45(@campo_pendulo,t,xini,options);
     df=xs(end,2);
end
```

Sin embargo, no he conseguido hacerlo funcionar para el método de Newton, y pese a considerar que el método de las rectas inexactas estaba bien programado, se queda estancado en los cambios de α así que solo presentaremos resultados para los primeros dos métodos, dejando igualmente el código de las rectas inexactas.

```
a=1;
b=4;
x1=(a+b)/2;
tol=1e-6;
maxiter=100000;


[x_incu,t_incu,iter_incu] = incu('funcionrr',a,b,tol,maxiter) ;
[x_budi,t_budi, iter_budi] = budi('funcionrr',,a,b,tol,maxiter);
[x_rein,t_rein, iter_rein, cambios_alfa] = rein('funcionrr', 'derivada_funcionrr',x1,tol
```

La tabla de resultados para los dos primeros métodos es

legends	x_values	y_values
"incu"	2.55744085961097	0.972578423669069
"budi"	2.55744895782471	0.972578424439682

Figura 8: Tabla de resultados del caso D

2. Resumen final

Como resultado final, podemos observar que en general, todos los métodos converge a la solución con más o menos iteraciones, siendo la búsqueda dicotómica el que más iteraciones necesita, y las rectas inexactas el que más tiempo consume, pues dentro de sus iteraciones no estamos teniendo en cuenta todos los cambios que sufre α. Como nota, decir que en cuanto a tiempo e iteraciones el mejor método parece ser el de inteprolación cuadrática. Por otro lado, notar que si los a, b tomados eran demasiado grandes o la tolerancia era demasiado pequeña, los métodos de $budi$ e $incu$ no convergían tan bien, lo cual es un problema si no escogemos los a, b adecuados.

3. Anexos: Código

3.1. Búsqueda dicotómica

```
function [x1, t, I] = budi(f,a,b,tol, maxiter)
%Metodo de la busqueda dicotomica
tic
epsilon = 1e-5;
I=0;
while abs(b-a)>tol && I < maxiter
    x1=(a+b)/2;
    xa=x1-epsilon;
    xb=x1+epsilon;
    fa=feval(f,a);
    fb=feval(f,b);
    if fa<fb
        b=xb;
    elseif fa>fb
        a=xa;
    else
        a=xa;
        b=xb;
    end
    I=I+1;
end
t=toc;
end
```

3.2. Interpolación cuadrática

Para este método, nos hemos ayudado del libro (2) y de su descripción de los algoritmos.

```
function [x_gorro, t, I] = incu(f,a,b,tol, maxiter)
tic;
x1 = a;
x3 = b;
x2 = (x1+x3)/2;
x0 = 1e10;
f1=feval(f,x1);
f2=feval(f,x2);
f3=feval(f,x3);
numerador = (x2^2-x3^2)*f1+(x3^2-x1^2)*f2+(x1^2-x2^2)*f3;
denominador = 2*((x2-x3)*f1+(x3-x1)*f2+(x1-x2)*f3);
x_gorro = numerador/denominador;
f_gorro = feval(f,x_gorro);
I=1; %Aqui ya tendria una iteracion para el minimo
while abs(x_gorro-x0)>tol && I<maxiter
    if (x1<x_gorro) && (x_gorro<x2) % x1<x_gorro<x2
        if f_gorro<=f2
            x3=x2;
            f3=f2;
            x2=x_gorro;
            f2=f_gorro;
        else
            x1=x_gorro;
            f1=f_gorro;
        end
    elseif  (x2<x_gorro) && (x_gorro<x3) % x2<x_gorro<x3
```

```
        if f_gorro<=f2
           x1=x2;
           f1=f2;
           x2=x_gorro;
           f2=f_gorro;
        else
            x3=x_gorro;
            f3=f_gorro;
        end
    end
    I=I+1;
    x0=x_gorro;
    numerador = (x2^2-x3^2)*f1+(x3^2-x1^2)*f2+(x1^2-x2^2)*f3;
    denominador = 2*((x2-x3)*f1+(x3-x1)*f2+(x1-x2)*f3);
    x_gorro = numerador/denominador;
    f_gorro = feval(f,x_gorro);
end
t=toc;
end
```

3.3. Rectas inexactas

Notar que en este método, para no calcular nosotros la derivada, hemos decidido que $f(x)$ sea definida mediante x variable simbólica, lo cual nos permite utilizar $diff$ para diferenciar.

```
function [xk,t, I, cambios_alfa] = rein(f,x1,tol, maxiter)
% Espera una funcion con x simbolica
%Metodo de las rectas inexactas
tic
c1=0.1;
c2=0.9;
xk=x1;
I=0;
cambios_alfa=0;
% Definimos los ai
al = 0; % a_l
ar = 1000;
ak = al+2;
f_prima = eval(subs(diff(f),xk));
while abs(f_prima)>tol && I<maxiter
    % Defino todos los valores necesarios
    fxk = eval(subs(f,xk));
    f_prima_xk = eval(subs(diff(f),xk));
    dk = -sign(f_prima_xk);
    fxk_ak = eval(subs(f,xk+ak*dk));
    f_prima_xk_ak = eval(subs(diff(f),xk+ak*dk));
    % Primera condicion de wolfe, mientras no se cumpla sigo
    while fxk_ak > (fxk+c1*ak*dk*f_prima_xk)
       % si no se cumple la condicion de wolfe
        ar=ak;
```

```
        ak = (al+ar)/2;
        fxk = eval(subs(f,xk));
        f_prima_xk = eval(subs(diff(f),xk));
        dk = -sign(f_prima_xk);
        fxk_ak = eval(subs(f,xk+ak*dk));
        f_prima_xk_ak = eval(subs(diff(f),xk+ak*dk));
        cambios_alfa=cambios_alfa+1;
    end
     % Segunda condicion de wolfe, mientras no se cumpla sigo
    while abs(dk*f_prima_xk_ak)/abs(dk*f_prima_xk) > c2
       % si no se cumple la condicion de wolfe
        al=ak;
        ak = (al+ar)/2;
        if ar==10000
            ak = 2*ak;
        end
        f_prima_xk_ak = eval(subs(diff(f),xk+ak*dk));
        cambios_alfa=cambios_alfa+1;
    end
    % Llegados a este punto se cumplen todas las condiciones
    I=I+1;
    xk=xk+ak*dk;
    f_prima = eval(subs(diff(f),xk));
end
t=toc;
end
```

3.4. Rectas inexactas adaptado a derivadas

En este caso sí usamos como parámetros la función y su derivada.

```
function [xk,t, I, cambios_alfa] = rein_con_derivada(f,df,x1,tol, maxiter)
%Metodo de las rectas inexactas, donde df es la derievada de la funcion f
tic
c1=0.1;
c2=0.9;
xk=x1;
I=0;
cambios_alfa=0;
% Definimos los ai
al = 0; % a_l
ar = 1000;
ak = al+2;
f_prima = feval(df,xk);
while abs(f_prima)>tol && I<maxiter && cambios_alfa<10000
    % Defino todos los valores necesarios
    fxk = feval(f,xk);
    f_prima_xk = feval(df,xk);
    dk = -sign(f_prima_xk);
    fxk_ak = feval(f,xk+ak*dk);
    f_prima_xk_ak = feval(df,xk+ak*dk);
    % Primera condicion de wolfe, mientras no se cumpla sigo
    while fxk_ak > (fxk+c1*ak*dk*f_prima_xk)
       % si no se cumple la condicion de wolfe
        ar=ak;
        ak = (al+ar)/2;
        fxk = eval(subs(f,xk));
```

```
        f_prima_xk = feval(df,xk);
        dk = -sign(f_prima_xk);
        fxk_ak = feval(f,xk+ak*dk);;
        f_prima_xk_ak = feval(df,xk+ak*dk);
        cambios_alfa=cambios_alfa+1;
    end
     % Segunda condicion de wolfe, mientras no se cumpla sigo
    while abs(dk*f_prima_xk_ak)/abs(dk*f_prima_xk) > c2
       % si no se cumple la condicion de wolfe
        al=ak;
        ak = (al+ar)/2;
        if ar==10000
            ak = 2*ak;
        end
        f_prima_xk_ak = feval(df,xk+ak*dk);
        cambios_alfa=cambios_alfa+1;
    end
    % Llegados a este punto se cumplen todas las condiciones
    I=I+1;
    xk=xk+ak*dk;
    f_prima = feval(df,xk);
end
t=toc;
end
```

3.5. Newton

Al igual que en el método anterior, utilizamos $f(x)$ simbólica.

```
function [x,t,iter] = new(f,x0,tol,maxiter)
tic
x(1) = x0;
n1 = tol+1;
h = tol+1;
iter = 0;
while (abs(n1) > tol && abs(h) > tol) && iter < maxiter
    n1 = eval(subs(diff(f),(x(iter+1))));
    ddf = eval(subs(diff(f,2),(x(iter+1))));
    h = n1/ddf;
    x(iter+2) = x(iter+1)-h;
    iter = iter+1;
end
x = x(end);
t = toc;
end
```

4. Bibilografia

Referencias

[1] Apuntes de la asignatura de Optimización

[2] Andreas Antoniou y Wu Sheng-Lu, *Practical Optimization*, págs 95-97