Sesión 2: Resolución de ecuaciones no lineales.

Versión: 24 de marzo de 2011.

```
--> kill(all) $
```

Queremos resolver la ecuación exp(x) - 2*cos(x) = 0.

En primer lugar haremos la gráfica de f(x) para determinar un intervalo adecuado.

```
--> f(x) := exp(x) - 2*cos(x) $
    wxplot2d(f(x), [x,0,1]) $
```

1 Bisección

1.1 Primer intento

```
--> kill(all) $
```

```
--> f(x) := exp(x) - 2*cos(x) $
    a : 0.0 $
    b : 1.0 $
    m : (a + b)/2.0 $

    for n : 1 thru 15 step 1 do (
        if f(a)*f(m) < 0
            then
                b : m
            else
                a : m,
        m : (a+b)/2,
        print("Paso: ", n, ", Intervalo: [a,b]=", [a, b], " -> Punto medio: m=", m)
        ) $
```

1.2 Inclusión de un criterio de parada

```
--> kill(all) $
```

```
--> f(x) := exp(x) - 2*cos(x) $
    a : 0.0 $
    b : 1.0 $
    m : (a+b)/2.0 $

    tol : 10^(-8) $
    maxn : ceiling(log((b - a)/tol)/log(2)-1) $

    for n : 1 while n <= maxn do (
        if f(a)*f(m) < 0
            then
                b:m
            else
                a:m,
        m : (a+b)/2,
        print("Paso: ", n, ", Intervalo: [a, b]=", [a, b], " -> Punto medio: m=", m)
        ) $
```

Repite el proceso para la función f(x)=2*x-1 en el intervalo [0,1].
¿Observas algún comportamiento inadecuado?

1.3 Observación: error que suele "pasarse"

En las subsecciones 1.1 y 1.2, el programa de bisección diseñado funciona porque el cero no se alcanza en ninguno de los puntos medios calculados. Si aplicamos el programa a la función f(x)=x-0.125 comprobaremos este hecho: debería pararse en el segundo paso y no es así. De hecho, "cae" en un intervalo donde la función es siempre positiva. Por esta razón es conveniente introducir un criterio de parada o alguna orden en el bucle para que reconozca esta situación.

Por cierto, aunque teóricamente podría valer la comparación "f(m)=0", en la práctica se tendrá que trabajar con una comparación del tipo "abs(f(m))<prec" donde prec es un valor "pequeño" prefijado.

```
--> kill(all) $
```

```
--> f(x) := x - 0.125 $
    a : 0.0 $
    b : 1.0 $
    m : (a+b)/2.0 $

    tol : 10^(-8) $
    maxn : ceiling(log((b - a)/tol)/log(2)-1) $
    prec : 10^(-10) $

    for n : 1 while n <= maxn and abs(f(m)) > prec do (
        if f(a)*f(m) < 0
            then
                b:m
            else
                a:m,
        m : (a+b)/2,
        print("Paso: ", n, ", Intervalo: [a, b]=", [a, b], " -> Punto medio: m=", m)
        ) $
```

1.4 Observación: como hacer menos costoso el programa

Si lo pensamos un momento, el producto "f(a)*f(m)" no es necesario en cada iteración. Basta con determinar el signo de f(m) si previamente conocemos el signo de f(a). Para la primera función el programa sería el siguiente.

```
--> kill(all) $
```

```
--> f(x) := exp(x) - 2*cos(x) $
    a : 0.0 $
    b : 1.0 $
    m : (a+b)/2.0 $

    tol : 10^(-8) $
    maxn : ceiling(log((b - a)/tol)/log(2)-1) $
    prec : 10^(-10) $

    for n : 1 while n <= maxn and abs(f(m)) > prec do (
        if f(m) > 0
            then
                b:m
            else
                a:m,
        m : (a+b)/2,
        print("Paso: ", n, ", Intervalo: [a, b]=", [a, b], " -> Punto medio: m=", m)
        ) $
```

Realiza un programa que sea válido tanto para el caso "f(a)>0" como para el caso "f(a)<0".
(Sugerencia: un if en el que se incluyan dos suprogramas muy parecidos).

2 Secante

2.1 Fallo aparente

```
--> kill(all) $
```

```
--> f(x) := exp(x) - 2*cos(x) $
    x0 : 0.0 $
    x1 : 1.0 $

    for n : 2 thru 12 step 1 do (
        x2 : x0 - f(x0)*(x1-x0)/(f(x1) - f(x0)),
        print("Solución aproximada: x", n, "=", x2),
        x0 : x1,
        x1 : x2
        ) $
```

2.2 Criterio de parada en secante

```
--> kill(all) $
```

```
--> f(x) := exp(x) - 2*cos(x) $
    x0 : 0.0 $
    x1 : 1.0 $

    tol : 10^(-8) $

    for n : 2 while abs(x1-x0) > tol and n <= 25 step 1 do (
        x2 : x0 - f(x0)*(x1-x0)/(f(x1) - f(x0)),
        print("Solución aproximada: x", n, "=", x2),
        x0 : x1,
        x1 : x2
        ) $
```

2.3 Disminución del coste

Si queremos disminuir el número de evaluaciones, la manera obvia de proceder sería la siguiente.

```
--> kill(all) $
```

```
--> f(x) := exp(x) - 2*cos(x) $
    x0 : 0.0 $
    x1 : 1.0 $
    fx0 : f(x0) $
    fx1 : f(x1) $

    tol : 10^(-8) $

    for n : 2 while abs(x1-x0) > tol and n <= 25 step 1 do (
        x2 : x0 - fx0*(x1-x0)/(fx1 - fx0),
        print("Solución aproximada: x", n, "=", x2),
        x0 : x1,
        x1 : x2,
        fx0 : fx1,
        fx1 : f(x2)
        ) $
```

Hábilmente se puede reducir una evaluación más.

```
--> kill(all) $
```

```
--> f(x) := exp(x) - 2*cos(x) $
    x0 : 0.0 $
    x1 : 1.0 $
    fx0 : f(x0) $

    tol : 10^(-8) $

    for n : 2 while abs(x1-x0) > tol and n <= 25 step 1 do (
        fx1 : f(x1),
        x2 : x0 - fx0*(x1-x0)/(fx1 - fx0),
        print("Solución aproximada: x", n, "=", x2),
        x0 : x1,
        x1 : x2,
        fx0 : fx1
        ) $
```

En toda esta sección hemos trabajado con la expresión

```
x2 : x0 - f(x0)*(x1-x0)/(f(x1) - f(x0))
```

Pero también podemos utilizar la expresión equivalente

```
x2 : (f(x0)*x1-f(x1)*x0)/(f(x0) - f(x1))
```

¿Cuál de las dos crees que será más eficiente?

3 Newton-Raphson

3.1 Primer intento

```
--> f(x) := exp(x) - 2*cos(x) $
    define(d1f(x),diff(f(x),x,1)) $

    x0 : 0.5 $

    for n : 1 thru 7 step 1 do (
        x0 : x0 - f(x0)/d1f(x0),
      print("Solución aproximada: x", n, "=", x0)
        )$
```

3.2 Criterio de parada en Newton-Raphson

```
--> kill(all);
```

```
--> f(x) := exp(x) - 2*cos(x) $
    define(d1f(x),diff(f(x),x,1)) $

    x0 : 0.0 $
    x1 : 0.5 /*este es el verdadero valor inicial x0*/$

    tol : 10^(-15) $

    for n : 1 while abs(x1-x0) > tol and n <= 100 step 1 do (
        x0 : x1,
        x1 : x0 - f(x0)/d1f(x0),
        print("Solución aproximada: x", n, "=", x1)
        )$
```

4 Un procedimiento para implementar el método de bisección

Recordemos el programa con inclusión de un criterio de parada que realizamos al principio de esta sesión.

```
--> f(x) := exp(x) - 2*cos(x) $
    a : 0.0 $
    b : 1.0 $
    m : (a+b)/2.0 $

    tol: 10^(-3) $
    maxn : ceiling(log((b - a)/tol)/log(2) - 1) $

    for n : 1 while n <= maxn do (
        if f(a)*f(m)<0
            then
                b:m
            else
                a:m,
        m:(a+b)/2,
        print("Paso: ", n, ", Intervalo: [a,b]=", [a, b], " -> Punto medio: m=", m)
        ) $
```

Realicemos una pequeña modificación para simplificar el trabajo a realizar.

```
--> f(x) := exp(x) - 2*cos(x) $
    a : 0.0 $
    b : 1.0 $
    m : (a + b)/2.0 $

    tol: 10^(-3) $

    for n : 1 while abs(b-a) >= tol and n <= 100 step 1 do (
        if f(a)*f(m)<0 then b:m else a:m,
        m:(a+b)/2,
        print(n, ":", [a, b], " -> ", m)
    ) $
```

¿Por qué hace una iteración más que en el proceso anterior?

Diseñamos el procedimiento: comando "block".

4.1 Primer intento: fijando un número determinado de pasos.

```
--> biseccion(ley, var, a1, b1) := block( [a, b, c],
        a : float(a1),
        b : float(b1),
        for n:1 thru 15 step 1 do (
            c: (a+b)/2,
            if subst(a,var,ley)*subst(c,var,ley) < 0
                then
                    b : c
                else
                    a : c
            ),
        c
        )$
```

```
--> biseccion(f(x),x,0,1);
```

```
--> biseccion(f(x),x,1,2);
```

```
--> biseccion(f(x),x,-1,0);
```

```
--> biseccion(f(x),x,-2,-1);
```

```
--> biseccion(f(x),x,-1,-2);
```

4.2 Alternativa para no abusar del comando "subst": definición de funciones locales

```
--> biseccion(ley, var, a1, b1) := block( [a, b, c],
        local(f),
        define(f(x),subst(x,var,ley)),
        a : float(a1),
        b : float(b1),
        for n:1 thru 15 step 1 do (
            c: (a+b)/2,
            if f(a)*f(c) < 0
                then
                    b : c
                else
                    a : c
           ),
        c
        )$
```

```
--> f(x) := exp(x) - 2*cos(x) $
```

```
--> biseccion(f(x),x,0,1);
```

```
--> biseccion(f(x),x,1,2);
```

```
--> biseccion(f(x),x,-1,0);
```

```
--> biseccion(f(x),x,-2,-1);
```

```
--> biseccion(f(x),x,-1,-2);
```

La definición de funciones locales se puede (o debe) hacer en todos los ejemplos que siguen a continuación.

4.3 Controlemos el intervalo.

```
--> biseccion(ley, var, a1, b1) := block( [a, b, c],
        a : float(a1),
        b : float(b1),
        if subst(a,var,ley)*subst(b,var,ley) > 0
            then
                return ("Intervalo no adecuado"),
        for n:1 thru 15 step 1 do (
            c: (a+b)/2,
            if subst(a,var,ley)*subst(c,var,ley)<0
                then
                    b:c
                else
                    a:c
            ),
            c
        ) $
```

```
--> biseccion(f(x),x,0,1);
```

```
--> biseccion(f(x),x,1,2);
```

```
--> biseccion(f(x),x,-1,0);
```

```
--> biseccion(f(x),x,-2,-1);
```

```
--> biseccion(f(x),x,-1,-2);
```

4.4 Controlemos el número de iteraciones.

```
--> kill(all)$
```

```
--> f(x) := exp(x) - 2*cos(x) $
```

```
--> biseccion(ley, var, a1, b1) := block( [a, b, c, m, maxiter:300, tol:10^(-15), prec:10^(-10)],
        a : float(a1),
        b : float(b1),
        if subst(a,var,ley)*subst(b,var,ley)>0
            then
      return ("Intervalo no adecuado"),
        for n:1 while (n<= maxiter and abs(b-a)> tol and abs(subst((a+b)/2,var,ley))> prec) step 1 do (
            m:n,
            c: (a+b)/2,
            if subst(a,var,ley)*subst(c,var,ley)<0
                then
                    b:c
                else
                    a:c
            ),
        [m,{c}]
        ) $
```

```
--> biseccion(f(x),x,0,1);
```

```
--> biseccion(f(x),x,1,2);
```

```
--> biseccion(f(x),x,-1,0);
```

```
--> biseccion(f(x),x,-2,-1);
```

```
--> biseccion(f(x),x,-1,-3);
```

¿Qué pasa si, en el intervalo inicial, uno de los extremos es solución?
¿Y si la solución es justo el punto medio? (Recordemos la Observación 1.3)

```
--> f(x) := x-1 $
```

```
--> biseccion(f(x),x,0,1);
```

```
--> biseccion(f(x),x,0,2);
```

```
--> biseccion(f(x),x,1,2);
```

4.5 Versión definitiva.

Salvo mejora a realizar teniendo en cuenta las Observaciones dadas en 1.4 y 4.2.

```
--> kill(all) $
```

```
--> biseccion(ley,var,a1,b1):= block( [a, b, c, m, maxiter:300, tol:10^(-15), prec:10^(-10)],
        a : float(a1),
        b : float(b1),
        c : (a+b)/2,
        if abs(subst(a,var,ley)) < prec then return([0,{a}]),
        if abs(subst(b,var,ley)) < prec then return([0,{b}]),
        if subst(a,var,ley)*subst(b,var,ley) > 0 then return ("Intervalo no adecuado"),
        if abs(subst(c,var,ley)) < prec then return([1,{c}]),
        for n : 1 while (n<= maxiter and abs(b-a)> tol and abs(subst(c,var,ley))> prec) step 1 do (
            m : n,
            c : (a+b)/2,
            if subst(a,var,ley)*subst(c,var,ley) < 0 then b : c else a : c
          ),
        [m,{c}]
    ) $
```

```
--> f(x) := x^2-1 $
```

```
--> biseccion(f(x),x,0,2);
```

```
--> biseccion(f(x),x,0,1);
```

```
--> biseccion(f(x),x,-1,0);
```

```
--> biseccion(f(x),x,0,5);
```

```
--> biseccion(f(x),x,5,2);
```

```
--> biseccion(f(x),x,1,-1);
```

```
--> g(x):=x^4-2;
    biseccion(g(x),x,-0.5,2.5);
```

4.6 Cálculo de derivadas

```
--> kill(all) $
```

```
--> f(x):=x^3-5 $
```

```
--> diff(f(x),x);
```

Comprobemos que con este cálculo de la derivada, no obtenemos una expresión evaluable.

```
--> fd(x):=diff(f(x),x);
```

```
--> fd(3);
```

¿Cómo podemos solventar este problema?

```
--> fd1(x):=''(diff(f(x),x));
```

```
--> define(df2(x),diff(f(x),x));
```

```
--> fd3:diff(f(x),x);
```

¿Cuáles de las anteriores expresiones proporcionan una expresión evaluable de la derivada?
En los casos en los casos en los que no es evaluable, ¿hay alguna alternativa para realizar la evaluación?
(Sugerencia: recuerda cómo hemos usado la función "subst" en la implementación del método de bisección).

¿Qué ocurre si la variable "x" tiene preasignado un valor concreto?
¿Cómo podemos solventar este problema?
(Sugerencia: busca en la ayuda información sobre el operador " ' ").

4.7 Ejercicios

Realiza un procedimiento para implementar el método de la secante.

Realiza un procedimiento para implementar el método de Newton-Raphson.
(Indicación: para la definición de la derivada, parece que la forma más adecuada es la que hemos usado como fd3).

Realiza un procedimiento para implementar el método de regula-falsi.
Previamente, diseña el programa correspondiente a dicho método teniendo en cuenta todas las observaciones y comentarios de esta sesión.

En esta sesión no hemos cuidado mucho el aspecto visual de la salida de los resultados, en particular de las tablas. Modifícalas tal como se hizo en la sesión 1.