PRÁCTICA 3: Ecuaciones no Lineales. Métodos de Intervalo

3.1 Resumen de Teoría

3.1.1 Introducción

En las Prácticas 3 y 4 trataremos sobre uno de los problemas más clásicos de la aproximación numérica: la solución de ecuaciones no lineales. En el campo de la tecnología, y principalmente en la ingeniería, nos encontramos muy frecuentemente con el problema de determinar las raíces de una ecuación no lineal de la forma:

$$f(x) = 0$$

es decir, números x_0 tales que $f(x_0) = 0$. En la mayoría de los casos tienen que usarse métodos de aproximación para encontrar x_0 tal que satisfaga $f(x_0) \approx 0$. Una posible clasificación de los métodos es:

- **Métodos de Intervalo.** Estos métodos se caracterizan por el hecho de que la función cambia de signo al cruzar el eje de abscisas. Por ello es necesario proponer un intervalo donde suceda esto, es decir el intervalo propuesto debe contener la raíz. El más simple es el método de Bisección.
- **Métodos de Punto Fijo.** Estos métodos se caracterizan por el uso de un valor inicial cercano a la raíz, que es usado para encontrar un nuevo valor que puede converger a la raíz o divergir de ésta. Los más simples son: el método de Newton y el de la Secante.

Esta práctica está destinada al estudio del primer tipo de ellos. Pero antes veamos un concepto básico en los algoritmos: la velocidad de convergencia.

3.1.2 Velocidad y Orden de Convergencia

En los procesos de iterativos es interesante estimar la velocidad de convergencia, es decir, la manera en que disminuye el error en cada iteración para valores grandes de n. Una medida habitual de esa velocidad se obtiene a través del orden de convergencia, que definiremos a continuación para cualquier sucesión convergente.

Sea (x_n) una sucesión convergente a un número c y sea $e_n = x_n - c$ que supondremos no nulo para todo n. Entonces se dice que la convergencia de (x_n) es de orden p si se verifica:

$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^i} = 0, \;\; para\; i = 0,1,\dots,p-1; \;\; \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^p} = k \neq 0$$

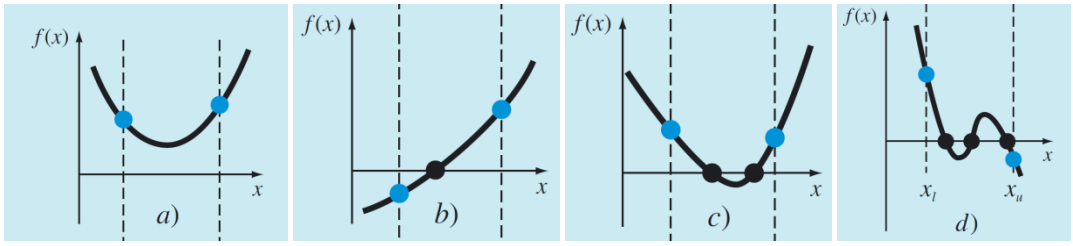
Si $p = 1$ la convergencia se dice lineal, cuando $p = 2$ hablamos de convergencia cuadrática, y cuando $p = 3$ de convergencia cúbica. Hablando de forma simple, converger linealmente es converger muy despacio. Sin embargo, converger cuadráticamente es converger bastante rápido. Aproximadamente en cada iteración duplicamos el número de decimales exactos.

3.1.2 Método de Bisección

Este es uno de los métodos más sencillos e intuitivos para resolver ecuaciones en una variable. Se basa en el *Teorema del valor intermedio*, el cual establece que toda función continua *f* en un intervalo cerrado [*a, b*] toma todos los valores que se hallan entre *f*(*a*) y *f*(*b*).

En caso de que *f*(*a*) y *f*(*b*) tengan signos opuestos, el valor cero sería un valor intermedio, por lo que, simplemente usando el *Teorema de Bolzano*, sabemos que existe un *p* en (*a, b*) que cumple *f*(*p*)=0. De esta forma, se asegura la existencia de al menos una solución de la ecuación *f*(*x*)=0.

La siguiente figura muestra algunas de las formas en las que la raíz puede encontrarse (o no encontrarse) en un intervalo definido [*a, b*]. La figura *b)* representa el caso en que una sola raíz está acotada por los valores positivo y negativo de *f*(*x*). Sin embargo, la figura *d)*, donde *f*(*a*) y *f*(*b*) están también en lados opuestos del eje *x*, muestra tres raíces que se presentan en ese intervalo. En general, si *f*(*a*) y *f*(*b*) tienen signos opuestos, existe un número impar de raíces en el intervalo. Como se indica en las figuras *a)* y *c)*, si *f*(*a*) y *f*(*b*) tienen el mismo signo, no hay raíces o hay un número par de ellas entre los valores.



Como su propio nombre indica es un método en el que el intervalo se divide siempre a la mitad. El método determina:

$$c = \frac{a+b}{2}$$

Y se comprueban las siguientes condiciones:

- Si *f*(*a*).*f*(*c*) <0, entonces *f*(*x*) tiene una raíz en el intervalo [*a, c*]. En este caso tomamos el valor de *c* como *b*.
- Si *f*(*a*).*f*(*c*) >0, entonces *f*(*c*).*f*(*b*) <0 y *f*(*x*) tiene una raíz en el intervalo [*c, b*]. En este caso tomamos el valor de *c* como *a*.
- Si *f*(*a*).*f*(*c*) =0 ó *f*(*c*).*f*(*b*) =0 entonces *f*(*c*) = 0 y hemos encontrado una raíz del polinomio.

En los dos primeros casos se ha determinado un nuevo intervalo que contiene una raíz de la función. Con este nuevo intervalo se continúa sucesivamente encerrando la solución en un intervalo cada vez más pequeño, hasta alcanzar la precisión deseada.

En el tercer caso hemos encontrado en *c* una raíz del polinomio, pero este caso no suele suceder en general, debido a los redondeos. Por esta razón se debe fijar una tolerancia (digamos por ejemplo de 10^{-3}) para finalizar el proceso en este caso. Además insistimos en que debe comprobarse siempre la continuidad de la función *f*(*x*) en el intervalo [*a, b*].

Ahora debemos desarrollar criterios objetivos para decidir cuándo debe terminar el método. Obviamente necesitamos estrategias que no necesiten el conocimiento previo de la raíz.

Una forma sería mediante el llamado *error relativo porcentual* ε_a definido por:

$$\varepsilon_a = \left| \frac{x_r^{nuevo} - x_r^{anterior}}{x_r^{nuevo}} \right| 100\%$$

donde x_r^{nuevo} es la raíz en la iteración actual y $x_r^{anterior}$ el valor de la raíz en la iteración anterior. Se utiliza el valor absoluto, ya que por lo general importa sólo la magnitud de ε_a sin considerar su signo. Cuando ε_a es menor que un valor previamente fijado ε_s, termina el proceso.

Por otro lado debemos tener en cuenta que, si se sabe que la solución está en [*a*, *b*], al aproximarla por el punto medio del intervalo, el *error absoluto máximo* cometido es:

$$\frac{b-a}{2}$$

con lo cual, si el método se aplica *N* veces, la cota del error será:

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

por lo que una forma de garantizar que el error máximo fuese una cantidad *E* prefijada, sería hallar *N* de forma que:

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

y tras redondearlo al entero superior, si el resultado no es exacto, aplicar el método de bisección *N* veces. Este criterio de parada es también muy utilizado.

El método de bisección es menos eficiente que otros métodos que veremos (como el método de Newton), pero es mucho más seguro para garantizar la convergencia. La bisección converge linealmente, por lo cual es un poco lento, sin embargo, se garantiza la convergencia si *f*(*a*) y *f*(*b*) tienen distinto signo.

Si existieran más de una raíz en el intervalo entonces el método sigue siendo convergente pero no resulta tan fácil caracterizar hacia qué raíz converge el método. En lo sucesivo supondremos que la raíz buscada es única.

3.2 Comandos de Maxima para resolver ecuaciones algebraicas

Vamos a presentar, en primer lugar, alguno de los comandos generales de Maxima para resolver ecuaciones algebraicas no lineales.

3.2.1. El comando solve

Comenzaremos por la función s**olve** que nos resuelve una ecuación algebraica, o un sistema de ecuaciones algebraicas, de manera exacta:

solve([ecuaciones],[variables])
solve(ecuación, variable)

Cuando sólo es una ecuación no es necesario usar corchetes. Como ya sabemos, no siempre le será posible encontrar solución exacta, como ocurre para las ecuaciones polinómicas de grado mayor a 4, no resolubles por radicales.

```
(%i1)   kill(all)$
(%i1)   solve(x^2-3*x+1=0,x);
```

(%o1) $[x=-\frac{\sqrt{5}-3}{2},x=\frac{\sqrt{5}+3}{2}]$

```
(%i2)   solve(x^5+x^2+1=0,x);
```

(%o2) $[0=x^5+x^2+1]$

También podemos resolver ecuaciones que dependan de algún parámetro, pero en este caso hemos de indicar respecto de que variable resolvemos.

```
(%i3)   eq1:x^3-a*x^2-x^2+2*x=0;
```

(eq1) $x^3-a\,x^2-x^2+2\,x=0$

```
(%i4)    solve(eq1,x);
```

(%o4) $[x=-\frac{\sqrt{a^2+2\,a-7}-a-1}{2},x=\frac{\sqrt{a^2+2\,a-7}+a+1}{2},x=0]$

```
(%i5)   solve(eq1,a);
```

(%o5) $[a=\frac{x^2-x+2}{x}]$

En el caso de ecuaciones de una sola variable nos podemos ahorrar el escribirla:

```
(%i6)   solve(x^2+2*x=0);
(%o6)   [x=-2,x=0]
```

También podemos ahorrarnos el escribir el segundo término de la ecuación si es cero:

```
(%i7)   solve(x^2+2*x);
(%o7)   [x=-2,x=0]
```

Cuando buscamos las raíces de un polinomio a veces tenemos que tener en cuenta la multiplicidad. Por ejemplo:

```
(%i8)    solve(x^7-2*x^6+2*x^5-2*x^4+x^3=0,x);
(%o8)    [x=-%i,x=%i,x=1,x=0]
```

Para conocer la multiplicidad de las raíces se usa el comando **multiplicities**:

```
(%i9)    multiplicities;
(%o9)    [1,1,2,3]
```

Debemos observar también que el resultado de la orden **solve** no es una lista de puntos, es una lista de ecuaciones. Una posible solución consiste en llamar a la salida de alguna manera y "rescatar" con el siguiente comando sus soluciones:

```
(%i10)   s:solve(x^2+2*x=0);
(s)      [x=-2,x=0]
(%i12)   r1:ev(x,s[1]);
         r2:ev(x,s[2]);
(r1)     -2
  (r2)     0
```

El comando **solve** también intenta resolver ecuaciones no algebraicas, pero con resultados muy pobres, como se ve en los siguientes ejemplos:

```
(%i17)   solve([sqrt(x)=4],[x]);
         solve([sqrt(x)=4-x],[x]);
         solve([exp(x)=1],[x]);
         solve([exp(x)=1+x],[x]);
         solve([sin(x)=0],[x]);
(%o13)   [x=16]
  (%o14)   [x=4-sqrt(x)]
  (%o15)   [x=0]
  (%o16)   [x=%e^x-1]
  solve: using arc-trig functions to get a solutior
  Some solutions will be lost.
  (%o17)   [x=0]
```

3.2.2. El comando algsys

La orden **algsys** resuelve exclusivamente ecuaciones o sistemas de ecuaciones algebraicas.

algsys([ecuaciones],[variables])

La primera diferencia de **algsys** con la orden **solve** es pequeña: **algsys** siempre tiene como entradas listas. En otras palabras, aunque sólo tengamos una ecuación tenemos que agrupar la ecuación o ecuaciones entre corchetes igual que las incógnitas.

```
(%i1)    kill(all)$
(%i1)    algsys([x^2-4*x+3=0],[x]);
(%o1)    [[x=3],[x=1]]
```

En general, para ecuaciones polinómicas **algsys** nos permite algo más de flexibilidad ya que funciona bien con polinomios de grado alto y, además, permite seleccionar las raíces reales. El comportamiento de **algsys** está determinado por la variable **realonly**. Su valor por defecto es **false**. Esto significa que **algsys** muestra todas las raíces. Si su valor es **true** sólo muestra las raíces reales.

```
(%i2)    eq:x^4-1=0;
(eq)     x^4-1=0
(%i3)    algsys([eq],[x]);
(%o3)    [[x=1],[x=-1],[x=%i],[x=-%i]]
(%i5)    realonly:true$
         algsys([eq],[x]);
(%o5)    [[x=1],[x=-1]]
```

La segunda diferencia es que **algsys** intenta resolver numéricamente la ecuación si no es capaz de encontrar la solución exacta. Consideremos la siguiente ecuación, y vemos como el comando **solve** no es capaz de resolverla de forma exacta:

```
(%i7)    eq:x^6+x+1=0$
         solve(eq,x);
(%o7)    [0=x^6+x+1]
```

Volvamos a poner la variable **realonly** en su valor por defecto.

```
(%i8)    realonly:false$
```

Y comprobemos que **algsys** sí es capaz de resolverla, pero en este caso, de forma numérica:

```
(%i10)   sol:algsys([eq],[x])$
         transpose(%);
(%o10)
[ [x=-1.03838075445846 %i-0.1547351444968429] ]
[ [x=1.03838075445846 %i-0.1547351444968429] ]
[ [x=-0.3005069203095515 %i-0.7906671888144177] ]
[ [x=0.3005069203095515 %i-0.7906671888144177] ]
[ [x=0.9454023333112606-0.6118366937810087 %i] ]
[ [x=0.6118366937810087 %i+0.9454023333112606] ]
```

El commando **transpose** expresa la solución en forma de vector columna, más manejable.

3.2.3. Los comandos allroots y realroots

Las ecuaciones polinómicas se pueden resolver de manera aproximada también mediante los comandos **allroots** y **realroots**, que están especializados en encontrar soluciones racionales aproximadas de polinomios en una variable. La primera de ellas nos da todas las soluciones, y la segunda sólo las reales y pueden ser útiles en polinomios de grado alto. Se escriben en la forma:

allroots(polinomio)
realroots(polinomio)

```
(%i1)   kill(all)$
(%i1)   eq:x^9+x^7-x^4+x=0;
(eq)    x^9+x^7-x^4+x=0
(%i2)   sol:allroots(eq)$
(%i3)   transpose(sol);
                                  [ x = 0.0                                  ]
                                  [                                          ]
                                  [ x = 0.3019050774831237 %i + 0.8440677798277921 ]
                                  [                                          ]
                                  [ x = 0.8440677798277921 - 0.3019050774831237 %i ]
                                  [                                          ]
                                  [ x = 0.8923132916887636 %i - 0.3284644192383591 ]
(%o3)                             [                                          ]
                                  [ x = - 0.8923132916887636 %i - 0.3284644192383591 ]
                                  [                                          ]
                                  [ x = 0.5110407920843032 %i - 0.8098692958948327 ]
                                  [                                          ]
                                  [ x = - 0.5110407920843032 %i - 0.8098692958948327 ]
                                  [                                          ]
                                  [ x = 1.189238256723473 %i + 0.2942659353053997 ]
                                  [                                          ]
                                  [ x = 0.2942659353053997 - 1.189238256723473 %i ]
(%i4)   realroots(eq);
(%o4)   [x = 0]
```

3.2.4 El comando find_root

Para finalizar esta presentación de los comandos de Maxima, indicar que también dispone de la función:

find_root(expr, x, a, b)

que utiliza el método de la bisección para resolver ecuaciones (aunque si la función es suficientemente suave, puede aplicar el método de regula falsi. Este comando aproxima una raíz de expr o de una función *f* en el intervalo cerrado [*a*, *b*], si la función tiene signos diferentes en los extremos del intervalo si no da un mensaje de error.

```
(%i5)   find_root(x^6+x-5, x, 0, 2);
(%o5)   1.246628157210559
```

3.3 El Método de Bisección

En esta sección vamos a programar el método de bisección, tal como lo describimos en la introducción teórica. Como vimos el algoritmo de bisección tiene la ventaja que converge siempre a una solución pero tiene el inconveniente que converge lentamente.

El método de bisección pertenece a la categoría de los procesos iterativos, donde para buscar la solución aproximada se repite el método hasta obtener una buena aproximación. La pregunta clave es cuándo debe terminar el método.

Criterio1. Como en cualquier proceso iterativo una posibilidad es finalizar el cálculo cuando el *error relativo porcentual* ε_a sea menor que un valor previamente fijado de *tolerancia* ε_s:

$$\varepsilon_a = \left| \frac{x_r^{(k)} - x_r^{(k-1)}}{x_r^{(k)}} \right| 100\% < \varepsilon_s$$

donde $x_r^{(k)}$ es el valor de la iteración *k*-ésima y $x_r^{(k-1)}$ el valor de la iteración anterior. Este método puede, en algunos casos, dar problemas.

Criterio 2. Pero en el método de bisección tenemos la ventaja de que conocemos una estimación del *error absoluto máximo* cometido. Como vimos, si se sabe que la solución está en [*a, b*], una forma de garantizar que el error absoluto máximo sea una cantidad *E* prefijada, es hallar *N* de forma que:

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

y tras redondearlo al entero superior, si el resultado no es exacto, aplicar el método de bisección *N* veces. Este criterio de parada es muy utilizado.

Criterio 3. Pero aún tenemos una posibilidad más. Y en que en este tipo de problemas de hallar la raíz de una ecuación, sí que conocemos el valor de la función para la solución exacta: *f*(*x*)=0. Por tanto podemos imponer como condición de parada que:

$$|f(c) - 0| < \varepsilon$$

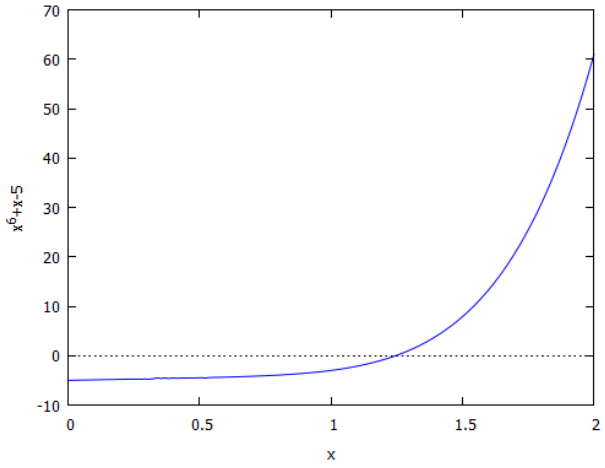
siendo ε la tolerancia deseada. Este criterio también es ampliamente utilizado.

Cuando programemos los distintos métodos utilizaremos la notación anterior para distinguir el criterio de parada, acabando el nombre del método respectivamente en 1, 2 o 3. Veamos un ejemplo. Consideremos de nuevo la ecuación que previamente resolvimos mediante el comando **find_root**:

$$f(x) = x^6 + x - 5 = 0$$

Para buscar el intervalo [*a, b*], tal que *f*(*a*).*f*(*b*) < 0, es conveniente ver la gráfica de la función. Se pueden ir tanteando distintos intervalos (comenzando por intervalos muy amplios) hasta localizar un cambio de signo, por ejemplo, en el intervalo [0, 2].

```
(%i1)     kill(all)$
(%i1)    plot2d(x^6+x-5,[x,0,2])$
```




```
(%i2)    f(x):=x^6+x-5$
```

Si prescindimos de la gráfica podemos ver que $f(0) = -5$ y $f(2) = 61$, y por tanto $f(x)$ posee al menos una raíz en dicho intervalo. Pero como por otro lado, la derivada $f'(x) = 6x^5 + 1 > 0$ para todo $x \in [0, 2]$, se deduce que existe una única raíz en dicho intervalo.

A continuación vamos a programar el método de bisección y vamos a comenzar utilizando el Criterio 2 de parada, determinando el número máximo de pasos a realizar N para garantizar que nuestro *error absoluto máximo* es menor que un valor dado E. Sabemos que si realizamos el método N veces:

$$\frac{b-a}{2^N} < E \quad \rightarrow 2^N > \frac{b-a}{E}$$

Tomando logaritmos en base 2:

$$N > log_2 \frac{b-a}{E}$$

Por ejemplo:

$$N = \left[log_2 \frac{b-a}{E}\right] + 1$$

denotando por $[x]$ la parte entera del número real x.

Para calcular N en Maxima, basta con utilizar el comando **floor(*x*)** que nos devuelve el mayor entero que es menor o igual que el número real x. Además fijémonos en como calcular logaritmos en una base distinta del número e.

Volvamos a nuestro ejemplo: $f(x) = x^6 + x - 5 = 0$ y supongamos que elegimos $E = 10^{-6}$. Como los extremos del intervalo son $a = 0$ y $b = 2$ tenemos que:

```
(%i3)    log2(x):=log(x)/log(2)$
(%i7)    a:0$
         b:2$
         E:10^(-6)$
         N:floor(log2((b-a)/E))+1;
(N)      21
```

El programa más simple en Maxima puede escribirse como sigue:

```
(%i9)    for i:1 thru N do (c:(a+b)/2,
         if f(a)*f(c)<0 then b:c else a:c)$
         print("La aproximación buscada es c = ",float(c))$
La aproximación buscada es c =  1.246628761291504
```

El programa anterior, puede mejorarse y escribirse como una función mediante un **block**. Comenzaremos introduciendo como comentarios las variables de entrada y salida del programa. Esto es fundamental en programación para conseguir una fácil lectura por otro usuario del programa. El programa aproxima la raíz de una ecuación $f(x) = 0$, cuando f es continua y toma signos opuestos en los extremos. Si $f(x)$ toma el mismo signo se sale con un mensaje de error. Usamos el comando **sign** de Maxima. El criterio de salida es una cota del error absoluto. La salida devuelve la aproximación y el número de iteraciones realizadas. Dentro del bloque la función se llamará **f_** para no coincidir con ninguna función definida fuera.

```
(%i1)    kill(all)$
(%i1)    /* Función biseccion2(f_,a,b,E) */
         /* Método de Bisección para resolver f(x) = 0 */
         /* ARGUMENTOS DE ENTRADA: */
         /* f ... Función de la que se buscan los ceros */
         /* a,b . Extremos del intervalo de búsqueda */
         /* E ... Cota del error absoluto: |c-raiz exacta|<= E */
         /* ARGUMENTOS DE SALIDA: */
         /* c ... Valor aproximado de la raíz */;
         /* N ... Número de Iteraciones */;
         biseccion2(f_,a,b,E):=
         block([numer],numer:true,
         if (sign(f_(a)) = sign(f_(b))) then (print(" Atención,
         f tiene el mismo signo en ",a," y ",b,"!"), return(false)),
         log2(x):=log(x)/log(2),
         N:floor(log2((b-a)/E))+1,
         for i:1 thru N do (
                 c:(a+b)/2,
                 if f_(a)*f_(c)<=0 then b:c else a:c
                                 ),
         print("La aproximación buscada es c = ",c),
             print("Número de Iteraciones = ",N))$
```

Y sin más ya podemos aplicarlo:

```
(%i2)    f(x):=x^6+x-5$
(%i3)    biseccion2(f,0,2,10^-6)$
La aproximación buscada es c =  1.246628761291504
  Número de Iteraciones =  21

(%i4)    g(x):=3-x^2$
(%i5)    biseccion2(g,0,2,10^-6)$
La aproximación buscada es c =  1.732050895690918
  Número de Iteraciones =  21
```

Otra posibilidad es programar el Criterio 3 de parada. Es decir, iremos subdividiendo el intervalo por su punto medio, hasta que el valor de $f(x)$ en dicho punto medio sea muy pequeño, es decir tan cercano a cero como deseemos. Debemos en este caso introducir esta tolerancia y también el número máximo de iteraciones que estamos dispuestos a realizar para obtener la solución aproximada, pues este valor no es ahora conocido a priori, como antes.

```
(%i1)    kill(all)$
(%i1)    /* Función biseccion3(f_,a,b,tol,maxiter) */
         /* Método de Bisección para resolver f(x) = 0 */
         /* ARGUMENTOS DE ENTRADA: */
         /* f ... Función de la que se buscan los ceros */
         /* a,b . Extremos del intervalo de búsqueda */
         /* tol ..Tolerancia en la aproximación |f(c)|<tol */
         /* maxiter ..Número máximo de iteraciones */
         /* ARGUMENTOS DE SALIDA: */
         /* c ... Valor aproximado de la raíz */;
         /* N ... Número de Iteraciones */;
         biseccion3(f_,a,b,tol,maxiter):=
         block([numer],numer:true,
         if (sign(f_(a)) = sign(f_(b))) then (print(" Atención,
         f tiene el mismo signo en ",a," y ",b,"!"), return(false)),
         for i:1 thru maxiter do (
                 c:(a+b)/2,
                 if abs(f_(c))<tol then
                 (print("[Aproximación, Número de iteraciones]"),
                 return([c,i])
                 )
                 else(if f_(a)*f_(c)<0 then b:c else a:c))
         )$
```

Obteniendo en este caso:

```
(%i2)    f(x):=x^6+x-5$
(%i3)    biseccion3(f,0,2,10^-6,100);
[Aproximación, Número de iteraciones]
 (%o3)    [1.246628165245056,24]
```

Si comparamos ambas soluciones vemos cuál es mejor: la del Criterio 3. Razonar porqué.

```
(%i4)    f(1.246628165245056);
(%o4)    1.531770026375057 10^-7
(%i5)    f(1.246628761291504);
(%o5)    1.151676477917363 10^-5
```

Es interesante comentar que si introducimos como tol en el programa anterior el épsilon de máquina, en este caso diríamos que es la "solución exacta de máquina".

```
(%i6)    biseccion3(f,0,2,2.22*10^-16,100);
[Aproximación, Número de iteraciones]
 (%o6)    [1.246628157210559,53]
```

Calculemos ahora la raíz con el comando **realroots** (que da aproximaciones racionales de las raíces reales de un polinomio).

```
(%i7)    realroots(f(x)=0),numer;
(%o7)    [x=-1.361203998327255,x=1.246628135442734]
```

Si comparamos ambas soluciones vemos que nuestro método da mejor solución que el que lleva incorporado Maxima, a no ser que se modifique la variable opcional **rootsepsilon**, cuyo valor por defecto es 1.10^{-7}.

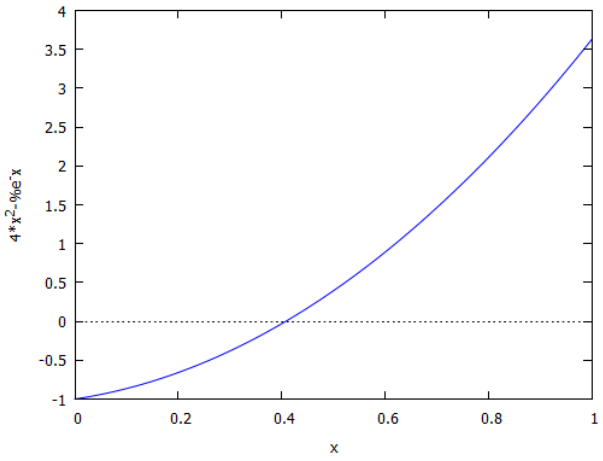
```
(%i9)    abs(f(1.246628157210559));
         abs(f(1.246628135442734));
(%o8)    3.552713678800501 10^-15
(%o9)    4.15001704112683 10^-7
```

Algo similar ocurre con el comando **algsys**, pero no con el comando **find_root** el cual nos devuelve la misma aproximación que nosotros hemos llamado "solución exacta de máquina" obtenida mediante biseccion3.

```
(%i11)   realonly:true;
         algsys([f(x)],[x]);
(realonly) true
(%o11)   [[x=1.246628131021195],[x=-1.361204013377926]]
(%i12)   find_root(f(x),0,2);
(%o12)   1.246628157210559
```

Veamos un ejemplo más de aplicación. Podemos cambiar el nombre de la función, sin problema.

```
(%i13)   g(x):=(2*x)^2-exp(-x)$
(%i14)   plot2d(g(x),[x,0,1])$
```




```
(%i15)   biseccion3(g,0,1,2.22*10^-16,100);
[Aproximación, Número de iteraciones]
(%o15)   [0.4077767094044804,54]
```

3.5 Ejercicios Propuestos

Ejercicio 1. Resolver mediante el método de bisección las siguientes ecuaciones:

a) $\cos(x).\cosh(x)+1 = 0$

b) $e^{x-2} - \ln(x + 2) - 2x = 0$

con una tolerancia ε de 10^{-6} en $|f(c) - 0| < \varepsilon$, localizando previamente intervalos adecuados.

Ejercicio 2. Sea la ecuación:

$$\text{sen}(x) - 2\cos(2x) + x^2 = \pi^2 - 2$$

(a) Demostrar, gráficamente y con Bolzano, que admite al menos una raíz en el intervalo $[0, 3\pi]$.
(b) Utilizar el comando **find_root** para aproximar la raíz.
(c) ¿Se obtiene raíz exacta? ¿Con qué error ha sido aproximada?.

Ejercicio 3. Aplica reiteradamente el método de bisección para hallar, con una tolerancia ε de 10^{-3}, en $|f(c) - 0| < \varepsilon$, la raíz de:

$$x = \tan(x)$$

que está entre 4 y 4.5. Compara con la solución obtenida por **find_root**.

Ejercicio 4. Usa el método de bisección para aproximar la raíz de:

a) $f(x) = \text{sqrt}(x^2 + 1) - \tan x$, comenzando en el intervalo [0.5, 1]

b) $f(x) = \exp(-x^3) - 2x + 1$ comenzando en el intervalo [0.75, 1]

Usar como criterio de parada que el error absoluto máximo sea menor de 10^{-12}.

Ejercicio 5. Demostrar que la siguiente función tiene una única raíz real:

$$f(x) = x^3 + e^x - 10$$

(a) Determinar un intervalo en el cual esté la raíz.
(b) Dibujar la gráfica de la función.
(c) Calcular el número de iteraciones para que el error absoluto máximo sea menor de 10^{-10} con el algoritmo de bisección. Hallar la raíz aproximada.

Ejercicio 6. Aplica el método de bisección para hallar, con una tolerancia ε de 10^{-16} en $|f(c) - 0| < \varepsilon$, la raíz que está entre 0 y 1 de:

$$x = \cos(x)$$

Compara con la solución obtenida por **find_root**.

Ejercicio 7. Aplicar el método de bisección a la función:

$$f(x) = x^6 - 1$$

Comienza en el intervalo [0, 1.5] hasta que la tolerancia ε en el valor de la función sea menor de 10^{-12}

Ejercicio 8. Vamos a comprobar la influencia del tamaño del intervalo [*a*, *b*] en el número de iteraciones a realizar por el algoritmo de bisección. Utilizaremos para ello la ecuación:

$$x^3 - 2x = 0$$

Toma intervalos cada vez más grandes [1, *k*] y observa el resultado.

Ejercicio 9. Hacer una variante del programa biseccion2 que muestre el valor obtenido en cada iteración. Probarlo.

Ejercicio 10. Vamos a comparar los criterios de parada 2 y 3. Para ello consideremos la función:

$$f(x) = (1/x) - 1000$$

Usar el criterio 2 con error absoluto máximo de 10^{-1} y el criterio 3 con tolerancia ε de 10^{-3}. Compara las soluciones obtenidas. ¿Qué observas?

Ejercicio 11. Hacer una variante del programa y llamarlo biseccion1 que utilice el criterio de parada1. Probarlo y comparar con los otros criterios.