Métodos numéricos de resolución de ecuaciones

5

5.1 Introducción al análisis numérico 81   5.2 Resolución numérica de ecuaciones con *Maxima* 85   5.3 Breves conceptos de programación 89   5.4 El método de bisección 93   5.5 Métodos de iteración funcional 102

En este capítulo vamos a ver cómo encontrar soluciones aproximadas a ecuaciones que no podemos resolver de forma exacta. En la primera parte, presentamos algunos de los comandos incluidos en *Maxima* para este fin. En la segunda parte, mostramos algunos métodos para el cálculo de soluciones como el método de bisección o el de Newton-Raphson

Comenzamos la primera sección hablando sobre las ventajas e inconvenientes de trabajar en modo numérico.

5.1 Introducción al análisis numérico

Los ordenadores tienen un capacidad limitada para almacenar cada número real por lo que en un ordenador únicamente pueden representarse un número finito de números reales: los números máquina. Si un número real no coincide con uno de estos números máquina, entonces se aproxima al más próximo. En este proceso se pueden producir, y de hecho se producen, errores de redondeo al eliminar decimales. También se pueden introducir errores en la conversión entre sistema decimal y sistema binario: puede ocurrir que un número que en sistema decimal presente un número finito de dígitos, en sistema binario presente un número infinito de los mismos.

Como consecuencia de esto, algunas propiedades aritméticas dejan de ser ciertas cuando utilizamos un ordenador.

La precisión de un número máquina depende del número de bits utilizados para ser almacenados.

Puede producirse una severa reducción en la precisión si al realizar los cálculos se restan dos números similares. A este fenómeno se le conoce como cancelación de cifras significativas. Lo que haremos para evitar este fenómeno será reorganizar los cálculos en un determinado desarrollo.

5.1.1 Números y precisión

Todos los números que maneja Maxima tienen precisión arbitraria. Podemos calcular tantos decimales como queramos. Si es posible, Maxima trabaja de forma exacta

```
(%i1)   sqrt(2);
(%o1)   √2
```

o podemos con la precisión por defecto

```
(%i2)  sqrt(2),numer;
(%o2)  1.414213562373095
```

Cuando decimos que $\sqrt{2}$ es un número de precisión arbitraria no queremos decir que podamos escribir su expresión decimal completa (ya sabes que es un número irracional) sino que podemos elegir el número de digitos que deseemos y calcular su expresión decimal con esa precisión.

```
(%i3)  fpprec:20;
(%o3)  20
(%i4)  bfloat(sqrt(2));
(%o4)  1.4142135623730950488b0
```

Vamos a comentar un par de detalles que tenemos que tener en cuenta en este proceso.

Errores de redondeo

Si sólo tenemos 5 dígitos de precisión, ¿cómo escribimos el número 7.12345? Hay dos métodos usuales: podemos truncar o podemos redondear. Por truncar se entiende desechar los dígitos sobrantes. El redondeo consiste en truncar si los últimos dígitos están entre 0 y 4 y aumentar un dígito si estamos entre 5 y 9. Por ejemplo, 7.46 se convertiría en 7.4 si truncamos y en 7.5 si redondeamos. El error es siempre menor en utilizando rendondeo. ¿Cuál de las dos formas usa Maxima? Puedes comprobarlo tu mismo.

```
(%i5)  fpprec:5;
(%o5)  5
(%i6)  bfloat(7.12345);
(%o6)  7.1234b0
```

¿Qué pasa si aumentamos la precisión en lugar de disminuirla?

```
(%i7)  fpprec:20;
(%o7)  20
(%i8)  bfloat(0.1);
(%o8)  1.0000000000000000555b-1
```

¿Qué ha pasado? 0.1 es un número exacto. ¿Porqué la respuesta no ha sido 0.1 de nuevo? Fíjate en la siguiente respuesta

```
(%i9)  bfloat(1/10);
```

```
(%o9)   1.0b-1
```

¿Cuál es la diferencia entre una otra? ¿Porqué una es exacta y la otra no? La diferencia es el error que se puede añadir (y acabamos de ver que se añade) cuando pasamos de representar un número en el sistema decimal a binario y viceversa.

5.1.2 Aritmética de ordenador

Sabemos que el ordenador puede trabajar con números muy grandes o muy pequeños; pero, por debajo de cierto valor, un número pequeño puede hacerse cero debido al error de redondeo. Por eso hay que tener cuidado y recordar que propiedades usuales en la aritmética real (asociatividad, elemento neutro) no son ciertas en la aritmética de ordenador.

Elemento neutro

Tomamos un número muy pequeño, pero distinto de cero y vamos a ver cómo *Maxima* interpreta que es cero:

```
(%i10)  h:2.22045*10^(-17);
(%o10)  2.22045 10^-17
```

Y si nos cuestionamos si h funciona como elemento neutro:

```
(%i11)  is(h+1.0=1.0);
(%o11)  true
```

la respuesta es que sí que es cierto que h+1.0=1.0, luego h sería cero.
Por encima, con números muy grandes puede hacer cosas raras.

```
(%i12)  g:15.0+10^(20);
(%o12)  1.10^20
(%i13)  is(g-10^(20)=0);
(%o13)  false
(%i14)  g-10^(20);
(%o14)  0.0
```

Aquí no sale igual, pero si los restáis cree que la diferencia es cero.

Propiedad asociativa de la suma

Con aritmética de ordenador vamos a ver que no siempre se cumple que: $(a+b)+c = a+(b+c)$

```
(%i15)  is((11.3+10^(14))+(-(10)^14)=11.3+(10^(14)+(-(10)^14)));
(%o15)  false
```

Si ahora trabajamos con números exactos, vamos a ver qué pasa:

```
(%i16)  is((113/10+10^(14))+(-(10)^14)=113/10+(10^(14)+(-(10)^14)));
(%o16)  true
```

5.1.3 Cancelación de cifras significativas

Como hemos visto, uno de los factores que hay que tener en cuenta a la hora de realizar cálculos, son aquellas operaciones que involucren valores muy grandes o cercanos a cero. Esta situación se presenta por ejemplo, el cálculo de la diferencia de los cuadrados de dos números muy similares

```
(%i17)  a:1242123.78$
        b:1242123.79$
        a^2-b^2;
        (a-b)*(a+b);
(%o18)  -24842.4755859375
(%o18)  -24842.47572313636
```

¿Por cierto? ¿Cuál es el resultado correcto? Probemos de otra forma

```
(%i19)  a:124212378$
        b:124212379$
        a^2-b^2;
        (a-b)*(a+b);
(%o20)  -248424757
(%o21)  -248424757
```

Parece que el resultado correcto es -24842.4757. Ninguno de los dos anteriores. Vale. Veamos otro ejemplo usando basado en la misma idea. Fijemos la precisión 40 y consideremos el número a:

```
(%i22)  fpprec:40;
(%o22)  40
(%i23)  a:bfloat(1-(10)^(-30));
```

```
(%o23)   9.999999999999999999999999999b-1
```

Ahora vamos a calcular: $1+a$ y $(a^2-1)/(a-1)$. Deberían ser iguales, ya que ambas expresiones matemáticamente son equivalentes:

$$1+a=\frac{(a-1)(a+1)}{a-1}=\frac{a^2-1}{a-1},$$

en cambio,

```
(%i24)   b:1+a$
(%i25)   c:(a^2-1)/(a-1)$
(%i26)   is(b=c);
(%o26)   false
```

No las reconoce como iguales. Este es el resultado del efecto de cancelación de cifras significativas que tiene lugar cuando se restan dos cantidades muy parecidas. En este caso es claro cuál de ambas formas de realizar el cálculo es mejor.

5.2 Resolución numérica de ecuaciones con *Maxima*

Las ecuaciones polinómicas se pueden resolver de manera aproximada. Los comandos `allroots` y `realroots` están especializados en encontrar soluciones racionales aproximadas de polinomios en una variable.

`allroots(`*polinomio*`)`	soluciones aproximadas del polinomio
`bfallroots(`*polinomio*`)`	soluciones aproximadas del polinomio con precisión arbitraria
`realroots(`*polinomio*`)`	soluciones aproximadas reales del polinomio
`realroots(`*polinomio, error*`)`	soluciones aproximadas reales del polinomio con cota del error
`nroots(`*polinomio*`,`*a*`,`*b*`)`	número de soluciones reales del polinomio entre a y b
`algsys([`*ecuaciones*`],[`*variables*`])`	resuelve la ecuación o ecuaciones

Estás órdenes nos dan todas las soluciones reales y complejas de un polinomio en una variable y son útiles en polinomios de grado alto cuando falla la orden `solve`. La primera de ellas, `allroots`, nos da las soluciones con la precisión por defecto

allroots

```
(%i27)   eq:x^9+x^7-x^4+x$
(%i28)   allroots(eq);
```

```
(%o28) [x=0.0,x=0.30190507748312%i+0.8440677798278,
       x=0.8440677798278-0.30190507748312%i,
       x=0.8923132916888%i-0.32846441923834,
       x=-0.8923132916888%i-0.32846441923834,
       x=0.51104079208431%i-0.80986929589487,
       x=-0.51104079208431%i-0.80986929589487,
       x=1.189238256723466%i+0.29426593530541,
       x=0.29426593530541-1.189238256723466%i]
```

bfallroots Si queremos una precisión determinada, usamos la orden `bfallroots`.

```
(%i29) fpprec:6$
       bfallroots(eq);
(%o29) [x=0.0b0,x=3.0191b-1%i+8.44063b-1,
       x=8.44063b-1-3.0191b-1%i,x=8.92279b-1%i-3.28481b-1,
       x=-8.92279b-1%i-3.28481b-1,x=5.11037b-1%i-8.09838b-1,
       x=-5.11037b-1%i-8.09838b-1,x=1.18924b0%i+2.94256b-1,
       x=2.94256b-1-1.18924b0%i]
```

realroots Si sólo nos interesan las soluciones reales, la orden `realroots` calcula soluciones racionales aproximadas del polinomio.

```
(%i30) eq1:x^4-3*x^3+x^2-4*x+12$
(%i31) realroots(eq1);
(%o31) [x=2,x=81497599/33554432]
```

Si comparas con la salida de `allroots`, comprobarás que 2 es solución, pero que $\frac{81497599}{33554432}$ sólo es una solución aproximada. La precisión con la que se realiza la aproximación se puede controlar con un segundo parámetro. Por ejemplo, si queremos que el error sea menor que 10^{-5}, escribimos lo siguiente.

```
(%i32) realroots(eq1,10^(-5));
(%o32) [x=2,x=636699/262144]
```

Recuerda que la variable `multiplicities` guarda la multiplicidad de cada una de las raíces de la última ecuación que has resuelto.

```
(%i33) realroots((x-2)^2*eq1,10^(-5));
(%o33) [x=2,x=636699/262144]
```

```
(%i34) multiplicities;
(%o34) [3,1]
```

Por último, comentar que es posible saber el número de raíces de un polinomio en una variable en un intervalo concreto[8]

```
(%i35) nroots(eq1,0,2);
(%o35) 1
```

eso sí, ten cuidado porque se cuentan raíces con su multiplicidad

```
(%i36) nroots((x-2)^2*eq1,0,2);
(%o36) 3
```

El teorema de los ceros de Bolzano

Uno de los primeros resultados que aprendemos sobre funciones continuas es que si cambian de signo tienen que valer cero en algún momento. Para que esto sea cierto nos falta añadir un ingrediente: la funciones tienen que estar definidas en intervalos. Este resultado se conoce como teorema de los ceros de Bolzano y es una variante del teorema del valor intermedio.

Teorema de los ceros de Bolzano

Teorema 5.1. *Sea $f : [a,b] \to \mathbb{R}$ una función continua verificando que $f(a)f(b) < 0$, entonces existe $c \in]a,b[$ tal que $f(c) = 0$.*

Ejemplo 5.2. Una de las utilidades más importantes del Teorema de los ceros de Bolzano es garantizar que una ecuación tiene solución. Por ejemplo, para comprobar que la ecuación $e^x + \log(x) = 0$ tiene solución, estudiamos la función $f(x) = e^x + \log(x)$: es continua en $\mathbb{R}^+$ y se puede comprobar que $f(e^{-10}) < 0$ y $0 < f(e^{10})$. Por tanto, la ecuación $e^x + \log(x) = 0$ tiene al menos una solución entre e^{-10} y e^{10}. En particular, tiene solución en $\mathbb{R}^+$.

`find_root(f(x),x,a,b)`	solución de f en $[a,b]$

find_root

El comando `find_root` encuentra una solución de una función (ecuación) continua que cambia de signo por el método de bisección, esto es, dividiendo el intervalo por la mitad y quedándose con aquella mitad en la que la función sigue cambiando de signo. En realidad el método que utiliza *Maxima* es algo más elaborado pero no vamos a entrar en más detalles.

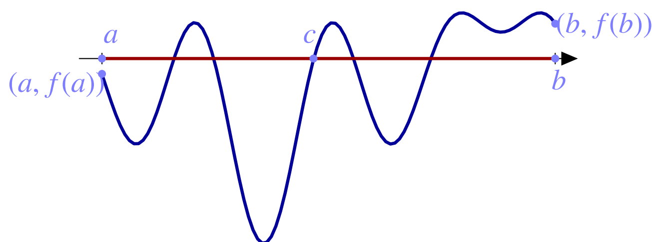


Figura 5.1 Teorema de los ceros de Bolzano

```
(%i37) f(x):=exp(x)+log(x);
```

```
(%o37)  f(x):=exp(x)+log(x)
```

Buscamos un par de puntos donde cambie de signo

```
(%i38)  f(1)
(%o38)  %e
(%i39)  f(exp(-3));
(%o39)  %e^(%e^-3)+3
```

¿Ese número es negativo?

```
(%i40)  is(f(exp(-3))<0);
(%o40)  true
```

o bien,

```
(%i41)  f(exp(-3)),numer;
(%o41)  -1.948952728663784
```

Vale, ya que tenemos dos puntos donde cambia de signo podemos utilizar `find_root`:

```
(%i42)  find_root(f(x),x,exp(-3),1);
(%o42)  0.26987413757345
```

Observación 5.3. Este método encuentra *una* solución pero no nos dice cuántas soluciones hay. Para eso tendremos que echar mano de otras herramientas adicionales como, por ejemplo, el estudio de la monotonía de la función.

5.2.1 Ejercicios

Ejercicio 5.1. Calcula las soluciones de $8\operatorname{sen}(x) + 1 - \frac{x^2}{3} = 0$.

Ejercicio 5.2. Encuentra una solución de la ecuación $\tan(x) = \frac{1}{x}$ en el intervalo $\left]0, \frac{\pi}{2}\right[$.

Ejercicio 5.3. ¿Cuantas soluciones tiene la ecuación $\frac{e^x}{2} - 2\operatorname{sen}(x) = 1$ en el intervalo $[-3, 3]$?

[8] Se admiten $\pm\infty$ como posibles extremos del intervalo

5.3 Breves conceptos de programación

Hemos visto cómo resolver ecuaciones y sistemas de ecuaciones con *Maxima* mediante la orden `solve` o `algsys`. La resolución de ecuaciones y sistemas de ecuaciones de manera exacta está limitada a aquellas para los que es posible aplicar un método algebraico sencillo. En estas condiciones, nos damos cuenta de la necesidad de encontrar o aproximar soluciones para ecuaciones del tipo $f(x) = 0$, donde, en principio, podemos considerar como f cualquier función real de una variable. Nuestro siguiente objetivo es aprender a "programar" algoritmos con *Maxima* para aproximar la solución de estas ecuaciones.

Lo primero que tenemos que tener en cuenta es que no existe ningún método general para resolver todo este tipo de ecuaciones en un número finito de pasos. Lo que sí tendremos es condiciones para poder asegurar, bajo ciertas hipótesis sobre la función f, que un determinado valor es una aproximación de la solución de la ecuación con un error prefijado.

El principal resultado para asegurar la existencia de solución para la ecuación $f(x) = 0$ en un intervalo $[a, b]$, es el Teorema de Bolzano que hemos recordado más arriba. Dicho teorema asegura que si f es continua en $[a, b]$ y cambia de signo en el intervalo, entonces existe al menos una solución de la ecuación en el intervalo $[a, b]$.

Vamos a ver dos métodos que se basan en este resultado. Ambos métodos nos proporcionan un algoritmo para calcular una sucesión de aproximaciones, y condiciones sobre la función f para poder asegurar que la sucesión que obtenemos converge a la solución del problema. Una vez asegurada esta convergencia, bastará tomar alguno de los términos de la sucesión que se aproxime a la sucesión con la exactitud que deseemos.

5.3.1 Bucles

Antes de introducirnos en el método teórico de resolución, vamos a presentar algunas estructuras sencillas de programación que necesitaremos más adelante.

La primera de las órdenes que vamos a ver es el comando `for`, usada para realizar bucles. Un bucle es un proceso repetitivo que se realiza un cierto número de veces. Un ejemplo de bucle puede ser el siguiente: supongamos que queremos obtener los múltiplos de siete comprendidos entre 7 y 70; para ello, multiplicamos 7 por cada uno de los números naturales comprendidos entre 1 y 10, es decir, repetimos 10 veces la misma operación: multiplicar por 7.

`for` *`var:valor1`* `step` *`valor2`* `thru` *`valor3`* `do` *`expr`*	bucle `for`
`for` *`var:valor1`* `step` *`valor2`* `while` *`cond`* `do` *`expr`*	bucle `for`
`for` *`var:valor1`* `step` *`valor2`* `unless` *`cond`* `do` *`expr`*	bucle `for`

En un bucle `for` nos pueden aparecer los siguientes elementos (no necesariamente todos)

a) *var:valor1* nos sitúa en las condiciones de comienzo del bucle.

b) *cond* dirá a *Maxima* el momento de detener el proceso.

c) *step valor2* expresará la forma de aumentar la condición inicial.

d) *expr* dirá a *Maxima* lo que tiene que realizar en cada paso; *expr* puede estar compuesta de varias sentencias separadas mediante punto y coma.

En los casos en que el paso es 1, no es necesario indicarlo.

`for var:valor1 thru valor3 do expr`	bucle `for` con paso 1
`for var:valor1 while cond do expr`	bucle `for` con paso 1
`for var:valor1 unless cond do expr`	bucle `for` con paso 1

Para comprender mejor el funcionamiento de esta orden vamos a ver algunos ejemplos sencillos.

En primer lugar, generemos los múltiplos de 7 hasta 70:

```
(%i43)  for i:1 step 1 thru 10 do print(7*i)
        7
        14
        21
        28
        35
        42
        49
        56
        63
        70
(%o43)  done
```

Se puede conseguir el mismo efecto sumando en lugar de multiplicando. Por ejemplo, los múltiplos de 5 hasta 25 son

```
(%i44)  for i:5 step 5 thru 25 do print(i);
        5
        10
        15
        20
        25
(%o44)  done
```

Ejemplo 5.4. Podemos utilizar un bucle para sumar una lista de números pero nos hace falta una variable adicional en la que ir guardando las sumas parciales que vamos obteniendo. Por ejemplo, el siguiente código suma los cuadrados de los 100 primeros naturales.

```
(%i45)  suma:0$
        for i:1 thru 100 do suma:suma+i^2$
        print("la suma de los cuadrados de los 100 primeros
          naturales vale ",suma);
        la suma de los cuadrados de los 100 primeros
(%o45)  naturales vale 338350
```

print(*expr1*,*expr2*,...)	escribe las expresiones en pantalla

En la suma anterior hemos utilizado la orden `print` para escribir el resultado en pantalla. La orden `print` admite una lista, separada por comas, de literales y expresiones. **print**

Por último, comentar que no es necesario utilizar una variable como contador. Podemos estar ejecutando una serie de expresiones mientras una condición sea cierta (bucle `while`) o mientras sea falsa (bucle `unless`). Incluso podemos comenzar un bucle infinito con la orden `do`, sin ninguna condición previa, aunque, claro está, en algún momento tendremos que ocuparnos nosotros de salir (recuerda el comando `return`).

while *cond* do *expr*	bucle while
unless *cond* do *expr*	bucle unless
do *expr*	bucle for
return (*var*)	bucle for

Este tipo de construcciones son útiles cuando no sabemos cuántos pasos hemos de dar pero tenemos clara cuál es la condición de salida. Veamos un ejemplo bastante simple: queremos calcular $\cos(x)$ comenzando en $x = 0$ e ir aumentando de 0.3 en 0.3 hasta que el coseno deje de ser positivo.

```
(%i46)  i:0;
(%o46)  0
(%i47)  while cos(i)>0 do (print([i,cos(i)]),i:i+0.3);
        0 1
        [0.3, 0.95533648560273]
        [0.6, 0.82533560144755]
        [0.9, 0.62160994025671]
        [1.2, 0.36235771003359]
        [1.5, 0.070737142212368]
(%o47)  done
```

5.3.2 Condicionales

La segunda sentencia es la orden condicional `if`. Esta sentencia comprueba si se verifica una condición, después, si la condición es verdadera *Maxima* ejecutará una *expresión1*, y si es falsa ejecutará otra *expresión2*.

if *condición* then *expr1* else *expr2*	condicional if-then-else
if *condición* then *expr*	condicional if-then

La expresiones 1 y 2 pueden estar formadas por varias órdenes separadas por comas. Como siempre en estos casos, quizá un ejemplo es la mejor explicación:

```
(%i48)  if log(2)<0 then x:5 else 3;
```

```
(%o48)   3
```

Observa que la estructura if-then-else devuelve la expresión correspondiente y que esta expresión puede ser una asignación, algo más complicado o algo tan simple como "3".

La última sentencia de programación que vamos a ver es la orden `return(var)` cuya única finalidad es la de interrumpir un bucle en el momento que se ejecuta y devolver un valor. En el siguiente ejemplo se puede comprender rápidamente el uso de esta orden.

```
(%i49)   for i:1 thru 10 do
           (
           if log(i)<2 then print("el logaritmo de",i,"es menor
             que 2")   else return(x:i)
           )$
         print("el logaritmo de",x,"es mayor que 2")$
         el logaritmo de 1 es menor que 2
         el logaritmo de 2 es menor que 2
         el logaritmo de 3 es menor que 2
         el logaritmo de 4 es menor que 2
         el logaritmo de 5 es menor que 2
         el logaritmo de 6 es menor que 2
         el logaritmo de 7 es menor que 2
         el logaritmo de 8 es mayor que 2
```

Observación 5.5. La variable que se utiliza como contador, `i` en el caso anterior, es siempre local al bucle. No tiene ningún valor asignado fuera de él. Es por esto que hemos guardado su valor en una variable auxiliar, `x`, para poder usarla fuera del bucle.

5.3.3 Ejercicios

Ejercicio 5.4. Usa el comando `for` en los siguientes ejemplos:
a) Sumar los números naturales entre 400 y 450.
b) Calcula la media de los cuadrados de los primeros 1000 naturales.

Ejercicio 5.5. Dado un número positivo x, se puede conseguir que la suma

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

sea mayor que x tomando un número n suficientemente grande. Encuentra la forma de calcular dicho número de manera general. ¿Cuál es el valor para $x = 10, 11$ y 13?

Ejercicio 5.6. Calcula las medias geométricas de los n primeros naturales y averigua cuál es el primer natural para el que dicha media sea mayor que 20.

Ejercicio 5.7. Calcula la lista de los divisores de una natural n.

5.4 El método de bisección

El método de bisección es una de las formas más elementales de buscar una solución de una ecuación. Ya sabemos que si una función continua cambia de signo en un intervalo, entonces se anula en algún punto. ¿Cómo buscamos dicho punto?

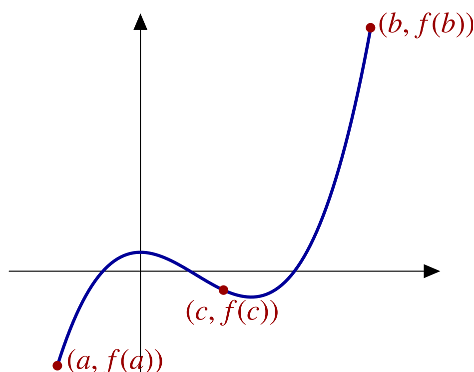


Figura 5.2 Método de bisección

Comencemos con una función $f : [a,b] \to \mathbb{R}$ continua y verificando que tiene signos distintos en los extremos del intervalo. La idea básica es ir dividiendo el intervalo por la mitad en dos subintervalos, $[a, \frac{1}{2}(a+b)]$ y $[\frac{1}{2}(a+b), b]$, y elegir aquel en el que la función siga cambiando de signo. Si repetimos este proceso, obtenemos un intervalo cada vez más pequeño donde se encuentra la raíz.

Más concretamente el proceso sería,

Datos iniciales: función f, números a, b
Se verifica que: $f(a)f(b) < 0$
bucle
- ▷ Calculamos el punto medio $c = (a+b)/2$,
- ▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
- ▷ y elegimos aquél donde la función cambie de signo

final bucle

5.4.1 Un ejemplo concreto

Vamos a aplicar este método a la función $f(x) = x^6 + x - 5$ en el intervalo $[0,2]$.
Definiremos en primer lugar la función y el intervalo y luego un bucle que nos calcula

```
(%i50)  f(x):=x^6+x-5;
        a:0.0;
        b:2.0;
(%o51)  x^6+x-5
(%o52)  0.0
(%o53)  2.0
```

Observa que hemos declarado `a` y `b` como valores numéricos. Comprobamos que la función cambia de signo.

```
(%i54)  f(a)*f(b);
(%o54)  -305.0
```

Ahora el bucle,[9]

```
(%i55)  for i:1 thru 10 do
          (
          c:(a+b)/2, /* calculamos el punto medio */
          if f(a)*f(c)<0 /* ¿cambia de signo en [a,c]? */
            then b:c /* elegimos [a,c] */
            else a:c, /* elegimos [c,b] */
          print(a,b) /* escribimos los resultados por pantalla */
          )$
        1.0 2.0
        1.0 1.5
        1.0 1.25
        1.125 1.25
        1.1875 1.25
(%o55)  1.21875 1.25
        1.234375 1.25
        1.2421875 1.25
        1.24609375 1.25
        1.24609375 1.248046875
```

Fíjate que ya sabemos que la solución es aproximadamente 1.24. No podemos estar seguros todavía del tercer decimal. Quizá sería mejor repetir el bucle más de diez veces, pero, ¿cuantas? Podemos establecer como control que la distancia entre los extremos sea pequeña. Eso sí, habría que añadir un tope al número de pasos asegurarnos de que el bucle termina.

Observación 5.6. Si no se quiere ralentizar mucho la ejecución de este bucle y del resto de programas en el resto del tema, es conveniente trabajar en modo númerico. Recuerda que este comportamiento se controla con la variable `numer`. Puedes cambiarlo en el menú **Numérico→Conmutar salida numérica** o directamente estableciendo el valor de `numer` en verdadero.

```
(%i56)  numer:true;
(%o56)  true
```

[9] Los símbolos /* y */ indican el principio y el fin de un comentario. No se ejecutan.

Control del error

En este método es fácil acotar el error que estamos cometiendo. Sabemos que la función f se anula entre a y b. ¿Cuál es la mejor elección sin tener más datos? Si elegimos a la solución, teóricamente, podría ser b. El error sería en este caso $b - a$. ¿Hay alguna elección mejor? Sí, el punto medio $c = (a+b)/2$. ¿Cuál es el error ahora? Lo peor que podría pasar sería que la solución fuera alguno de los extremos del intervalo. Por tanto, el error sería como mucho $\frac{b-a}{2}$. En cada paso que damos dividimos el intervalo por la mitad y al mismo tiempo también el error cometido que en el paso n-ésimo es menor o igual que $\frac{b-a}{2^n}$.

A partir de aquí, podemos deducir el número de iteraciones necesarias para obtener una aproximación con un error o exactitud prefijados. Si notamos por "`err`" a la exactitud prefijada, entonces para conseguir dicha precisión, el número "n" de iteraciones necesarias deberá satisfacer

$$\frac{b-a}{2^n} \leq err$$

así,

$$n \geq \log_2\left(\frac{b-a}{err}\right) = \frac{\log\left(\frac{b-a}{err}\right)}{\log(2)}.$$

`ceiling(a)`	menor entero mayor o igual que `a`

La orden `ceiling(x)` nos da el menor entero mayor o igual que x. Bueno, ya sabemos cuántos pasos tenemos que dar. Reescribimos nuestro algoritmo con esta nueva información:

ceiling

Datos iniciales: función f, números a, b, error err, contador i
Se verifica que: $f(a)f(b) < 0$
▷ Calculamos el número de pasos
para i:1 hasta número de pasos **hacer**
 ▷ Calculamos el punto medio $c = (a+b)/2$,
 ▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
 ▷ y elegimos aquél donde la función cambie de signo
final bucle

Volviendo a nuestro ejemplo, nos quedaría algo así.

```
(%i57)  f(x):=x^6+x-5;
        a:0$
        b:2$
        err:10^(-6)$
        log2(x):=log(x)/log(2)$ /* hay que definir el logaritmo en
        base 2 */
        pasos:ceiling(log2((b-a)/err))$
        for i:1 thru pasos do
          (
          c:(a+b)/2,
          if f(a)*f(c)<0
            then b:c
            else a:c,
          print(a,b) /* escribimos los resultados por pantalla */
          )$
```

¿Y si hay suerte?

Si encontramos la solución en un paso intermedio no habría que hacer más iteraciones. Deberíamos parar y presentar la solución encontrada. En cada paso, tenemos que ir comprobando que $f(c)$ vale o no vale cero. Podríamos comprobarlo con una orden del tipo `is(f(c)=0)`, pero recuerda que con valores numéricos esto puede dar problemas. Mejor comprobemos que es "suficientemente" pequeño.

Datos iniciales: función f, números a, b, error *err*, contador i, precisión *pr*
Se verifica que: $f(a)f(b) < 0$
▷ Calculamos el número de pasos
para i:1 hasta número de pasos **hacer**
  ▷ Calculamos el punto medio $c = (a + b)/2$,
  si f(c)<pr **entonces**
    ▷ La solución es c
  en otro caso
    ▷ Comparamos los signos de $f(a)$, $f(c)$ y $f(b)$
    ▷ y elegimos aquél donde la función cambie de signo
  final si
  ▷ La solución aproximada es c
final del bucle

En nuestro ejemplo, tendríamos lo siguiente

```
(%i58)  f(x):=x^6+x-5;
        a:0$
        b:2$
        err:10^(-6)$
        pr:10^(-5)$
        log2(x):=log(x)/log(2)$
        pasos:ceiling(log2((b-a)/err))$
        for i:1 thru pasos do
          (
          c:(a+b)/2,
          if abs(f(c))<pr
            then (print("La solucion es exacta"), return(c))
            else if f(a)*f(c)<0
              then b:c
              else a:c
          )$
        print("la solucion es ",c)$ /* aproximada o exacta, es la
        solución */
```

¿Se te ocurren algunas mejoras del algoritmo? Algunas ideas más:

a) el cálculo de $f(a)f(c)$ en cada paso no es necesario: si sabemos el signo de $f(a)$, sólo necesitamos saber el signo de $f(c)$ y no el signo del producto,

b) habría que comprobar que $f(a)$ y $f(b)$ no son cero (eso ya lo hemos hecho) ni están cerca de cero como hemos hecho con c.

c) Si queremos trabajar con una precisión mayor de 16 dígitos, sería conveniente utilizar numéros en coma flotante grandes.

5.4.2 Funciones y bloques

Una vez que tenemos más o menos completo el método de bisección, sería interesante tener una forma cómoda de cambiar los parámetros iniciales: la función, la precisión, los extremos, etc. Un bloque es la estructura diseñada para esto: permite evaluar varias expresiones y devuelve el último resultado salvo petición expresa.

La forma más elemental de "programa" en *Maxima* es lo que hemos hecho dentro del cuerpo del bucle anterior: entre paréntesis y separados por comas se incluyen comandos que se ejecutan sucesivamente y devuelve como salida la respuesta de la última sentencia.

```
(%i59)  (a:3,b:2,a+b);
(%o59)  5
```

Variables y funciones locales

Es conveniente tener la precaución de que las variables que se utilicen sean locales a dicho programa y que no afecten al resto de la sesión. Esto se consigue agrupando estas órdenes en un bloque

```
(%i60)  a:1;
(%o60)  1
(%i61)  block([a,b],a:2,b:3,a+b);
(%o61)  5
(%i62)  a;
(%o62)  1
```

Como puedes ver, la variable `a` global sigue valiendo uno y no cambia su valor a pesar de las asignaciones dentro del bloque. Esto no ocurre con las funciones que definamos dentro de un bloque. Su valor es global a menos que lo declaremos local explícitamente con la sentencia `local`. Observa la diferencia entre las funcione f y g.

local

```
(%i63)  block([a,b],
          local(g),g(x):=x^3,
          f(x):=x^2,
          a:2,b:3,g(a+b));
(%o63)  125
```

Si preguntamos por el valor de f o de g fuera del bloque, f tiene un valor concreto y g no:

```
(%i64)  f(x);
(%o64)  x^2
(%i65)  g(x);
(%o65)  g(x)
```

`local(`*`funciones`*`)`	declara funciones locales a un bloque
`return(`*`expr`*`)`	detiene la ejecución de un bloque y devuelve *expr*
`block([`*`var1,var2,..`*`],`*`expr1,expr2,..`*`)`	evalúa *expr1, expr2,...* y devuelve la última expresión evaluada

El último paso suele ser definir una función que permite reutilizar el bloque. Por ejemplo, el factorial de un número natural n se define recursivamente como

$$1! = 1, \quad (n+1)! = (n+1) \cdot n!.$$

Podemos calcular el factorial de un natural usando un bucle: usaremos la variable f para ir acumulando los productos sucesivos y multiplicamos todos los naturales hasta llegar al pedido.

```
(%i66)  fact(n):=block([f:1], for k:1 thru n do f:f*k,f );
(%o66)  fact(n):=block([f:1],for k thru n do f:f*k,f)
(%i67)  fact(5);
(%o67)  120
```

¿Y si quiero acabar antes?

Si queremos salir de un bloque y devolver un resultado antes de llegar a la última expresión, podemos usar la orden `return`. Por ejemplo, recuerda la definición que hicimos en el primer tema de la función logaritmo con base arbitraria.

```
(%i68)  loga(x):=log(x)/log(a)$
```

Esto podemos mejorarlo algo utilizando dos variables:

```
(%i69)  loga(x,a):=log(x)/log(a)$
```

pero deberíamos tener en cuenta si a es un número que se puede tomar como base para los logaritmos. Sólo nos valen los números positivos distintos de 1. Vamos a utilizar un bloque y un condicional.

```
(%i70)  loga(x,a):=block(
          if a<0 then print("La base es negativa"),
          if a=1 then print("La base es 1"),
          log(x)/log(a)
        )$
```

Si probamos con números positivos

```
(%i71)  loga(3,4);
         log(3)
(%o71)  ------
         log(4)
```

Funciona. ¿Y si la base no es válida?

```
(%i72)  log(3,-1);
        No calculamos logaritmos con base 1
          log(3)
(%o72)  -------
         log(-1)
```

Fíjate que no hemos puesto ninguna condición de salida en el caso de que la base no sea válida. Por tanto, *Maxima* evalúa una tras otra cada una de las sentencias y devuelve la última. Vamos a arreglarlo.

```
(%i73)  loga(x,a):=block(
          if a<0 then
            (print("La base es negativa"),return()),
          if a=1 then
            (print("La base es 1"),return()),
          log(x)/log(a)
        )$
```

Parámetros opcionales

Para redondear la definición de la función logaritmo con base cualquiera, podría ser interesante que la función "loga" calcule el logaritmo neperiano si sólo ponemos una variable y el logaritmo en base a si tenemos dos variables.

Las entradas opcionales se pasan a la definición de una función entre corchetes. Por ejemplo, la función

```
(%i74)  f(a,[b]):=block(print(a),print(b))$
```

da por pantalla la variable `a` y el parámetro o parámetros adicionales que sean. Si solo escribimos una coordenada

```
(%i75)  f(2);
        2
        []
(%o75)  []
```

nos devuelve la primera entrada y la segunda obviamente vacía en este caso. Pero si añadimos una entrada más

```
(%i76)  f(2,3);
        2
        [3]
(%o76)  [3]
```

o varias

```
(%i77)  f(2,3,4,5);
        2
        [3,4,5]
```

```
(%o77)  [3,4,5]
```

Como puedes ver, "[b]" en este caso representa una lista en la que incluimos todos los parámetros opcionales que necesitemos. Ahora sólo es cuestión de utilizar las sentencias que nos permiten manejar los elementos de una lista para definir la función logaritmo tal y como queríamos.

```
(%i78)  loga(x,[a]):=block([res],/* variable para el resultado */
          if length(a)=0
            then return(res:log(x))
            else (
              if a[1]<0 then (print("La base es negativa"),return()),
              if a[1]=1 then (print("La base es 1"),return()),
              log(x)/log(a[1])
            )
        )$
```

Observación 5.7. Se puede salir del bloque de definición de la función usando la sentencia `error(`*`mensaje`*`)` en los casos en que la base no sea la adecuada.

```
(%i79)  loga(x,[a]):=block([res],/* variable para el resultado */
          if length(a)=0
            then return(res:log(x))
            else (
              if a[1]<0 then error("Cambia la base"),
              if a[1]=1 then error("La base es 1"),
              log(x)/log(a[1])
            )
        )$
```

5.4.3 De nuevo bisección

Si unimos todo lo que hemos aprendido, podemos definir una función que utilice el método de bisección. Hemos usado la sentencia `subst` para definir la función a la que aplicamos bisección dentro del bloque. La orden `subst(a,b,c)` sustituye `a` por `b` en `c`.

```
biseccion(expr,var,ext_inf,ext_sup):=
    block(
        [a,b,c,k,err:10^(-8),prec:10^(-9)],
        /* extremos del intervalo */
        a:ext_inf,
        b:ext_sup,

        /* número de pasos */
        local(log2,f),
        define(log2(x),log(x)/log(2)),
```

```
 define(f(x),subst(x,var,expr)),
 pasos:ceiling(log2((b-a)/err)),

/* comprobamos las condiciones iniciales */
 if f(a)*f(b)>0 then error("Error: no hay cambio de signo"),

 /* ¿se alcanza la solución en los extremos? */
 if abs(f(a)) < prec then return(a),
 if abs(f(b)) < prec then return(b),

 for k:1 thru pasos do
    (
    c:(a+b)/2,
    if abs(f(c))< prec then return (c),
    if f(a)*f(c)< 0 then b:c else a:c
    ),
 c);
```

A partir de este momento, podemos utilizarlo usando

```
(%i80) biseccion(x^2-2,x,0.0,3.0);
(%o80) 1.414213562384248
```

Observa que las cotas del error y la precisión la hemos fijado dentro del bloque. Prueba a añadirlo como valores opcionales.

5.4.4 Ejercicios

Prueba a cambiar la función, los extremos del intervalo (en los cuales dicha función cambia de signo), así como la exactitud exigida. Intenta también buscar un caso simple en el que se encuentre la solución exacta en unos pocos pasos. Por último, intenta usar el algoritmo anterior para calcular $\sqrt[3]{5}$ con una exactitud de 10^{-10}.

5.5 Métodos de iteración funcional

En esta sección tratamos de encontrar una solución aproximada de una ecuación de la forma $x = f(x)$. Es usual referirse a dichas soluciones como *puntos fijos* de la función f. Los puntos fijos de una función f no son más los puntos de intersección de las gráficas de la función f y de la identidad. Por ejemplo, la función de la Figura 5.3 tiene tres puntos fijos.

Hay algunas condiciones sencillas que nos garantizan que una función tiene un único punto fijo.

Teorema 5.8. *Sea $f : [a,b] \to [a,b]$.*
a) Si f es continua, f tiene al menos un punto fijo.
b) Si f es derivable y $|f'(x)| \le L < 1$, $\forall x \in [a,b]$, entonces tiene un único *punto fijo.*

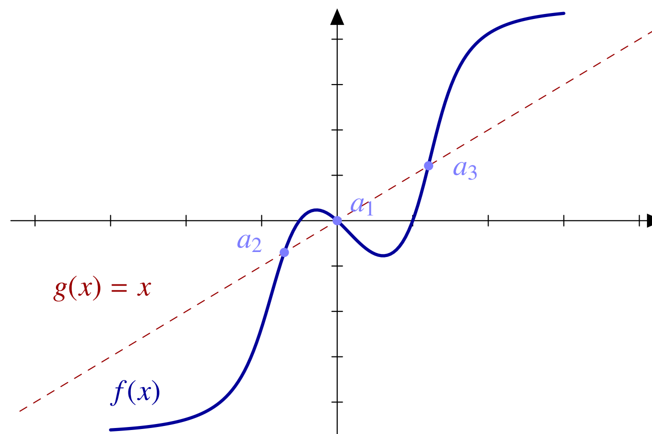


Figura 5.3 Puntos fijos de una función

Iteración funcional

Para buscar un punto fijo de una función, se elige un punto inicial $x_1 \in [a,b]$ cualquiera y aplicamos la función repetidamente. En otras palabras, consideramos la sucesión definida por recurrencia como

$$x_{n+1} = f(x_n)$$

para $n \geq 1$. Si la sucesión es convergente y llamamos s a su límite, entonces

$$s = \lim_{n\to\infty} x_{n+1} = \lim_{n\to\infty} f(x_n) = f(s),$$

o, lo que es lo mismo, s es un punto fijo de la función f. Más aún, en algunos casos es posible controlar el error.

Teorema 5.9. *Sea $f : [a,b] \to [a,b]$ derivable verificando que $|f'(x)| \leq L < 1$, $\forall\, x \in [a,b]$. Sea s el único punto fijo de la función f. Sea $x_1 \in [a,b]$ cualquiera y $x_{n+1} = f(x_n)$, entonces*

$$|x_n - s| \leq \frac{L}{1-L}\,|x_n - x_{n-1}| \leq \frac{L^{n-1}}{1-L}|x_2 - x_1|.$$

El método de construcción de la sucesión es lo que se conoce como un método de iteración funcional.

Ejemplo 5.10. Consideremos la función $f(x) = \frac{1}{4}\left(\cos(x) + x^2\right)$ con $x \in [0,1]$. Acotemos la derivada,

$$\left|f'(x)\right| = \left|\frac{-\operatorname{sen}(x) + 2x}{4}\right| \leq \left|\frac{\operatorname{sen}(x)}{4}\right| + \left|\frac{2x}{4}\right| \leq \frac{1}{4} + \frac{2}{4} = \frac{3}{4} < 1.$$

Por tanto, la función f tiene un único punto fijo en el intervalo $[0,1]$. Podemos encontrar el punto fijo resolviendo la correspondiente ecuación.

```
(%i81)   find_root((cos(x)+x^2)/4-x,x,0,1);
(%o81)   .2583921443715997
```

También podemos calcular las iteraciones comenzando en un punto inicial dado de manera sencilla utilizando un bucle

```
(%i82)  define(f(x),(cos(x)+x^2)/4)$
(%i83)  x0:0;
        for i:1 thru 10 do(
          x1:f(x0),print("Iteración",i,"vale", x1),x0:x1
        );
        0
        Iteración 1 vale 0.25
        Iteración 2 vale .2578531054276612
        Iteración 3 vale .2583569748525884
        Iteración 4 vale .2583898474528139
(%o83)  Iteración 5 vale .2583919943502456
        Iteración 6 vale .2583921345730372
        Iteración 7 vale .2583921437316118
        Iteración 8 vale .2583921443297992
        Iteración 9 vale .2583921443688695
        Iteración 10 vale .2583921443714213
```

Observación 5.11. Existen muchas formas de cambiar una ecuación de la forma $f(x) = 0$ en un problema de puntos fijos de la forma $g(x) = x$. Por ejemplo, consideremos la ecuación $x^2 - 5x + 2 = 0$.

a) Sumando x en los dos miembros

$$x^2 - 5x + 2 = 0 \iff x^2 - 4x + 2 = x,$$

y las soluciones de f son los puntos fijos de $g_1(x) = x^2 - 4x + 2$ (si los tiene).

b) Si despejamos x,

$$x^2 - 5x + 2 = 0 \iff x = \frac{x^2 + 2}{5}$$

y, en este caso, los puntos fijos de la función $g_2(x) = \frac{x^2+2}{5}$ son las soluciones buscadas.

c) También podemos despejar x^2 y extraer raíces cuadradas

$$x^2 - 5x + 2 = 0 \iff x^2 = 5x - 2 \iff x = \sqrt{5x - 2}\,.$$

En este caso, nos interesan los puntos fijos de la función $g_3(x) = \sqrt{5x - 2}$.

Como puedes ver, la transformación en un problema de puntos fijos no es única. Evidentemente, algunas de las transformaciones mencionadas antes dependen de que x sea distinto de cero, mayor o menor que $2/5$, etc. Además de eso las funciones g_i pueden tener mejores o peores propiedades, algunas verificarán las condiciones del teorema anterior y otras no.

5.5.1 Ejercicios

Ejercicio 5.8. Escribe un programa que dada una función, un punto inicial y un número de iteraciones, devuelva la última de ellas.

Ejercicio 5.9. Utiliza el método de iteración con las 3 funciones anteriores empezando en cada uno de los puntos 0.5, 1.5 y 6. ¿En cuáles obtienes convergencia a un punto fijo? ¿Es siempre el mismo?

5.5.2 Representación gráfica con el paquete *dynamics*

Para representar gráficamente los puntos de la sucesión, comenzamos con el primer punto de la sucesión $(x_1, f(x_1))$ y, a partir de ese momento, nos vamos moviendo horizontalmente hasta cruzar la bisectriz y verticalmente hasta encontrar de nuevo la gráfica de la función. Más concretamente,

a) comenzamos con $(x_1, f(x_1))$;
b) nos movemos horizontalmente hasta cortar la bisectriz. El punto de corte será $(f(x_1), f(x_1))$;
c) nos movemos verticalmente hasta cortar a la gráfica de f o, lo que es lo mismo, tomamos $x_2 = f(x_1)$ y le calculamos su imagen. El punto de corte será esta vez $(x_2, f(x_2))$.
d) Repetimos.

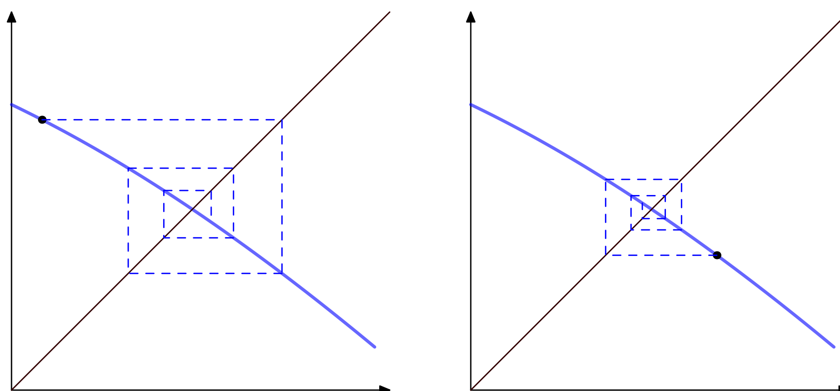


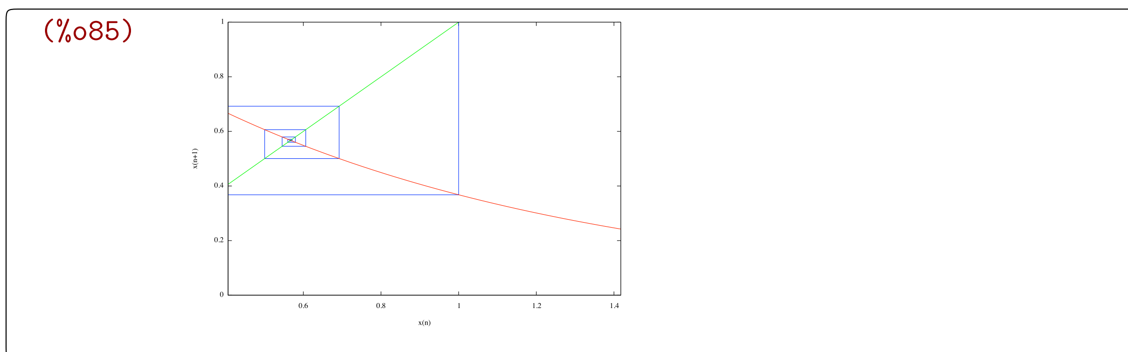
Figura 5.4 Método de iteración funcional

El paquete *dynamics* permite hacer estas representaciones de forma muy sencilla. Primero lo cargamos

```
(%i84)  load(dynamics)$
```

y luego podemos usar los comandos `evolution` y `staircase` indicando el punto inicial y el número de iteraciones. Por ejemplo, el diagrama de escalera para la función e^{-x} tomando como punto inicial 1 y 10 pasos

```
(%i85)  staircase(exp(-x),1,10,[y,0,1]);
```



Observa que hemos añadido `[y,0,1]` para indicar un rango más apropiado que el se dibuja por defecto.

`evolution(`*`func,pto1,pasos,opciones`*`)`	Gráfico de las iteraciones de *func*
`staircase(`*`func,pto1,pasos,opciones`*`)`	Gráfico de escalera de las iteraciones de *func*

5.5.3 Criterios de parada

Suele cuando trabajamos con métodos iterativos que tenemos una sucesión que sabemos que es convergente, pero no conocemos cuál es el valor exacto de su límite. En estos casos lo que podemos hacer es sustituir el valor desconocido del límite por uno de los términos de la sucesión que haría el papel de una aproximación de dicho límite. Por ejemplo, si consideramos el término general de una sucesión $\{a_n\}_{n\in\mathbb{N}}$ dada, con la ayuda del ordenador podemos calcular un número finito de términos. La idea es pararse en los cálculos en un determinado elemento a_{k_0} para que haga el papel del límite. Se impone entonces un *criterio de parada* para que dicho valor sea una buena aproximación del límite de la sucesión.

Tolerancia

Una forma de establecer un criterio de parada es considerar un número pequeño, al que llamaremos *tolerancia* y denotaremos por T, y parar el desarrollo de la sucesión cuando se de una de las dos circunstancias siguientes:

a) $|a_n - a_{n-1}| < T$,

b) $\dfrac{|\,a_n - a_{n-1}\,|}{|a_n|} < T$.

La primera es el error absoluto y la segunda el error relativo. Suele ser mejor utilizar esta última.

5.5.4 Ejercicios

Ejercicio 5.10. Añade una condición de parada al método de iteración.

5.5.5 El método de Newton-Raphson

El método de Newton-Raphson nos proporciona un algoritmo para obtener una sucesión de puntos que aproxima un cero de una función dada.

La forma de construir los términos de la sucesión de aproximaciones es sencilla. Una vez fijado un valor inicial x_1, el término x_2 se obtiene como el punto de corte de la recta tangente a f en x_1 con el eje OX. De la misma forma, obtenemos x_{n+1} como el punto de corte de la recta tangente a f en el punto x_n con el eje OX. De lo dicho hasta aquí se deduce:

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

Como observarás se trata de un método de iteración funcional. Para comprender el algoritmo observa la Figura 5.5 donde se ve cómo se generan los valores de las aproximaciones.

Para asegurar la convergencia de la sucesión (hacia la solución de la ecuación) usaremos el siguiente resultado.

Teorema 5.12. *Sea f una función de clase dos en el intervalo $[a, b]$ que verifica:*
a) $f(a)f(b) < 0$,
b) $f'(x) \neq 0$, *para todo* $x \in [a, b]$,
c) $f''(x)$ *no cambia de signo en* $[a, b]$.
Entonces, tomando como primera aproximación el extremo del intervalo $[a, b]$ donde f y f'' tienen el mismo signo, la sucesión de valores x_n del método de Newton-Raphson es convergente hacia la única solución de la ecuación $f(x) = 0$ en $[a, b]$.

Teorema de Newton-Raphson

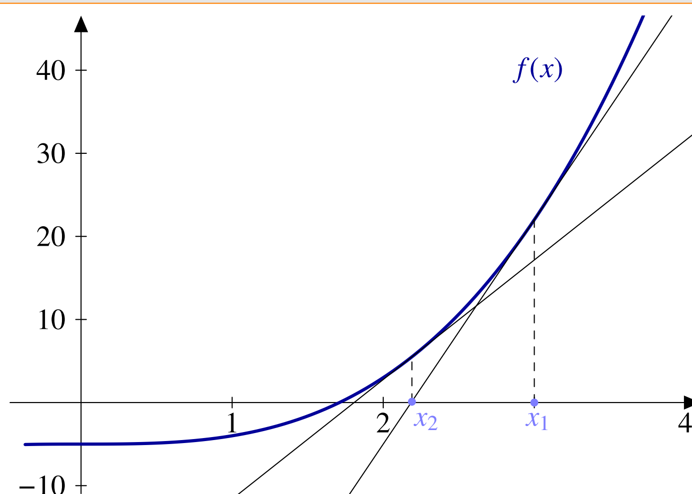


Figura 5.5 Método de Newton-Raphson

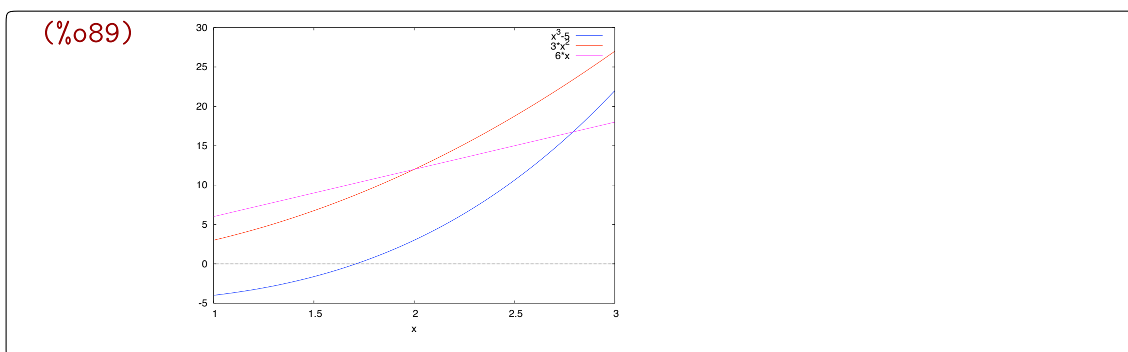
Una vez que tenemos asegurada la convergencia de la sucesión hacia la solución de la ecuación, deberíamos decidir la precisión. Sin embargo, veremos que el método es tan rápido en su convergencia que por defecto haremos siempre 10 iteraciones. Otra posibilidad sería detener el cálculo de cuando el valor absoluto de la diferencia entre x_n y x_{n+1} sea menor que la precisión buscada (lo cual no implica necesariamente que el error cometido sea menor que la precisión).

Utilizaremos ahora *Maxima* para generar la sucesión de aproximaciones. Resolvamos de nuevo el ejemplo de $x^3 - 5 = 0$ en el intervalo $[1, 3]$.

Podemos comprobar, dibujando las gráficas de $f(x) = x^3 - 5$, $f'(x)$ y $f''(x)$ en el intervalo $[1, 3]$, que estamos en las condiciones bajo las cuales el Teorema de Newton-Raphson nos asegura convergencia.

```
(%i86)  f(x):=x^3-5$
(%i87)  define(df(x),diff(f(x),x))$
(%i88)  define(df2(x),diff(f(x),x,2))$
(%i89)  plot2d([f(x),df(x),df2(x)],[x,1,3]);
```

(%o89)



A continuación, generaremos los términos de la sucesión de aproximaciones mediante el siguiente algoritmo. Comenzaremos por definir la función f y el valor de la primera aproximación. Inmediatamente después definimos el algoritmo del método de Newton-Raphson, e iremos visualizando las sucesivas aproximaciones. Como dijimos, pondremos un límite de 10 iteraciones, aunque usando mayor precisión decimal puedes probar con un número mayor de iteraciones.

```
(%i90)  y:3.0$
        for i:1 thru 10 do
          (y1:y-f(y)/df(y),
          print(i,"- aproximación",y1),
          y:y1
          );
        1 - aproximación 2.185185185185185
        2 - aproximación 1.80582775632091
        3 - aproximación 1.714973662124988
        4 - aproximación 1.709990496694424
        5 - aproximación 1.7099759468005
        6 - aproximación 1.709975946676697
        7 - aproximación 1.709975946676697
        8 - aproximación 1.709975946676697
        9 - aproximación 1.709975946676697
        10 - aproximación 1.709975946676697
```

Observarás al ejecutar este grupo de comandos que ya en la séptima iteración se han "estabilizado" diez cifras decimales. Como puedes ver, la velocidad de convergencia de este método es muy alta.

El módulo mnewton

El método que acabamos de ver se encuentra implementado en *Maxima* en el módulo `mnewton` de forma mucho más completa. Esta versión se puede aplicar tanto a funciones de varias variables, en otras palabras, también sirve para resolver sistemas de ecuaciones.

Primero cargamos el módulo

```
(%i91)  load(mnewton)$
```

y luego podemos buscar una solución indicando función, variable y punto inicial

```
(%i92)  mnewton(x^3-5,x,3);
(%o92)  [[x=1.709975946676697]]
```

5.5.6 Ejercicios

Ejercicio 5.11.
El método de regula falsi o de la falsa posición es muy parecido al método de bisección. La única diferencia es que se cambia el punto medio por el punto de corte del segmento que une los puntos $(a, f(a))$ y $(b, f(b))$ con el eje de abscisas.
Escribe un programa que utilice este método. Para la función $f(x) = x^2 - 5$ en el intervalo $[0, 4]$, compara los resultados obtenidos. ¿Cuál es mejor?

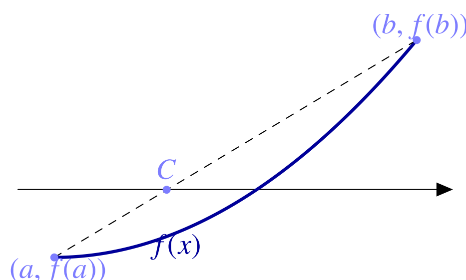


Figura 5.6 Método de regula falsi

Ejercicio 5.12. Reescribe el método de Newton-Raphson añadiendo una condición de salida (cuándo el error relativo o absoluto sea menor que una cierta cantidad) y que compruebe que la primera derivada está "lejos" de cero en cada paso.

Ejercicio 5.13.
El método de la secante evita calcular la derivada de una función utilizando recta secantes. Partiendo de dos puntos iniciales x_0 y x_1, el siguiente es el punto de corte de la recta que pasa por $(x_0, f(x_0))$ y $(x_1, f(x_1))$ y el eje de abscisas. Se repite el proceso tomando ahora los puntos x_1 y x_2 y así sucesivamente.
La convergencia de este método no está garantizada, pero si los dos puntos iniciales están próximos a la raíz no suele haber problemas. Su convergencia es más lenta que el método de Newton-Raphson aunque a cambio los cálculos son más simples.

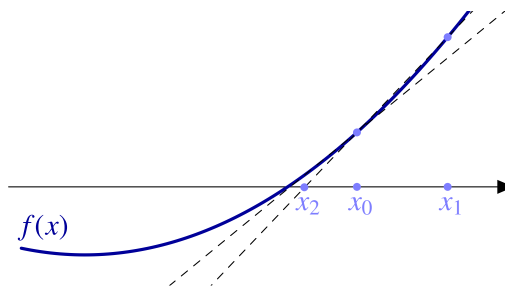


Figura 5.7 Método de la secante

Escribe un programa que utilice este método. Para la función $f(x) = x^2 - 5$, compara los resultados obtenidos con el método de Newton-Raphson. ¿Cuál es mejor?

Ejercicio 5.14. Resuelve las ecuaciones
a) $e^{-x} + x^2 - 3\,\text{sen}(x) = 0$,
b) $e^{|x|} = \arctan(x)$,
c) $x^{15} - 2 = 0$
utilizando los métodos que hemos estudiado. Compara cómo se comportan y decide en cuál la convergencia es más rápida.

Ejercicio 5.15.

a) Considérese la ecuación $e^{(x^2+x+1)} - e^{x^3} - 2 = 0$. Calcular programando los métodos de bisección y de Newton-Raphson, la solución de dicha ecuación en el intervalo $[-0.3, 1]$ con exactitud 10^{-10}.

b) Buscar la solución que la ecuación $\tan(x) = \frac{1}{x}$ posee en el intervalo $[0, \frac{\pi}{2}]$ usando los métodos estudiados.