# Capítulo 3

# Raíces de ecuaciones no lineales

## 3.1. Métodos de bisección, regula falsi, secante y Newton Raphson

En los apuntes de teoría se exponen diversos métodos para calcular las raíces de una ecuación no lineal. En el programa *raíces* están programadas las funciones correspondientes para calcular las raíces mediante los métodos de bisección, régula falsi, secante y Newton. El programa principal se compone de la función *main()*, de una serie de variables globales que fijan las precisión de la raíz y la precisión con la que la función se debe de anular y el número máximo de iteraciones, y de la definición de la función cuya raíz buscamos y de su derivada. Hemos construido una función para cada uno de los métodos de cálculo de raíces. Podemos calcular una raíz por varios métodos distintos llamando de forma sucesiva a las funciones que aplican este método. Comentado todas las llamadas menos una calculamos la raíz por el único método llamado. Esta es la estructura del programa, excluídas las funciones donde se programan cada uno de los métodos, que discutimos posteriormente;

```
// Programa de calculo de raices por diversos metodos
#include <cmath>
#include <iomanip>
#include <iostream>
using namespace std;
//prec:tolerancia en el calculo en la raiz
// fprec:valor maximo de la funcion en el valor aceptado para la raiz
// numiter:numero maximo de iteraciones
const double    prec = 1.e-8, fprec = 1.e-8;
const int       nprec = 10, nwidth = 16;
const int       numiter = 100;
const char     *tab = "\t";
//f: funcion, fd: derivada.
double          f(double x);
double          fd(double x);
```

```
int             biseccion(double, double);
int             biseccion2(double, double);
int             secante(double, double);
int             regulafalsi(double, double);
int             newton(double);
int             muller(double, double);
//Definicion de la funcion y derivadas
double
f(double x)
{
    //funcion
    return x * x - 1;
}

double
fd(double x)
{
    //primera derivada
    return 2 * x ;
}

int
main()
{
    /* Programa para comparar diversos métodos de calculo de raices */
    //Poner los puntos de horquillado;
    //x0, x1 puntos de partida
    double          x0 = 0., x1 = 2.;
    //Descomentar los metodos que se deseen utilizar
     biseccion(x0, x1);
     biseccion2(x0, x1);
     secante(x0, x1);
     regulafalsi(x0, x1);
     newton(x1);
     muller(x0,x1);
     return 0;
}
```

### 3.1.1. Método de bisección

La función que calcula la raíz por el método de la bisección la hemos codificado de la siguiente manera:

```
int biseccion(double x0, double x1)
{
    double          f(double x);
    cout << "    Metodo Biseccion" << endl;
    cout.precision(nprec);
    cout << " iteracion" << "             x  " << "  f(x)  " << endl;
    int             n = 1;
    double          x2, f0, f1, f2;
    f0 = f(x0);
    f1 = f(x1);
    f2=f1;

    if (f0 * f1 >= 0) {
        cout << " no hay raiz entre x0 y x1" << endl;
        return 0;
    }
    while (abs(x0 - x1) >= prec) {
        if (n >= numiter) {
            cout << " numero maximo de iteraciones excedido en biseccion"
              << endl;
            return 1;
        }
        x2 = (x0 + x1) / 2;
        f2 = f(x2);
        cout << setw(10) << n << "   " << setw(10) << x2 << "  "
          << f2 << endl;
        if (f2 * f0 < 0) {
            x1 = x2;
            f1 = f2;
        }
        if (f2 * f1 < 0) {
            x0 = x2;
            f0 = f2;
        }
        if(f2==0) break;
        n++;
    }
    cout << "biseccion     " << n - 1 << "  iteraciones   " << "raiz= "
        << setprecision(nprec) << setw(nwidth) << x2 <<
         " funcion= " << f2 << endl<<endl<<endl;
    return 0;
}
```

En este método, el único criterio de convergencia ha sido la precisión de cálculo de la raíz. En algunas circunstancias interesa que la función se anule con una determinada precisión `fprec`. En este caso se puede poner una condición adicional como en el siguiente programa

```
int biseccion2(double x0, double x1)
{
    double          f(double x);
    int             n = 1;
    double          f0 = f(x0), f1 = f(x1), x2 = x1, f2;
    cout << "    Metodo Biseccion 2" << endl;

    if (f0 * f1 >= 0) {
        cout << " no hay raiz entre x0 y x1" << endl;
        return 0;
    }
    while (abs(x0 - x1) >= prec || abs(0.5 * (f0 + f1)) >= fprec) {
        if(n>=numiter) {
    cout<<"numero maximo de iteraciones excedido en biseccion 2"<<endl;
            return 1;
        }
        x2 = (x0 + x1) / 2;
        f2 = f(x2);
        if (f2 * f1 < 0) {
            x0 = x2;
            f0 = f2;
        }
        if (f2 * f0 < 0) {
            x1 = x2;
            f1 = f2;
        }
if(f2==0) break;
        n++;
    }

    cout << "biseccion 2    " << n - 1 << "  iteraciones   "
 << "raiz= " << x2 << " funcion= " << f2 << endl<<endl<<endl;
    return 0;
}
```

El método de bisección es robusto pero lento. Es necesario horquillar las raíces, para lo cual hace falta tener un conocimiento gráfico del comportamiento de la función. Para ello, lo mejor es dibujar la función con un programa de representación gráfica como *gnuplot*. Primero se dibuja la función en un dominio grande y verificamos que la función tiene un comportamiento

asintótico que garantiza que no hay raíces fuera del diominio dibujado. Luego procedemos a dibujar la función en intervalos más pequeños hasta que veamos la posición de cada una de las raíces lo suficientemente clara como para dar un intervalo de horquillado relativamente pequeño, y estemos seguros de que sólo hay una raíz en ese intervalo.

**Ejercicio 1.** *Localizad todas las raíces reales de las siguientes funciones*

1. $f(x) = e^{-x} - 2x^2$
2. $f(x) = x^3 + 3x^2 - 1$

### 3.1.2. Métodos de la régula falsi y de la secante

El método de la régula falsi es un método de horquillado como el anterior pero converge mucho más rápidamente como vimos en la parte teórica. Hemos codificado el método en la siguiente forma

```
int regulafalsi(double x0, double x1)
{
    double          f(double x);
    int             n = 1;
    double          x2 = 0,x2old=1;
    cout << "    Metodo Regula Falsi" << endl;
    cout.precision(nprec);
    cout << " iteracion" << "             x  " << "  f(x)  " << endl;
    double f0=f(x0), f1=f(x1), f2=1.;
    if (f0 * f1 >= 0) {
        cout << " no hay raiz entre x0 y x1" << endl;
        return 0;
    }
    while ((abs(f2) >= fprec)||(abs(x2-x2old)>prec)) {
        if (n >= numiter) {
            cout << " numero maximo de iteraciones excedido en regulafalsi" << endl;
            return 1;
        }
        x2old=x2;
        x2 = x1 - f1 * (x1 - x0) / (f1 - f0);
        f2=f(x2);
        cout << setw(10) << n << "   " << setw(10) << x2 <<
             "  " << f2 << endl;
        if (f2 * f0 < 0){
            x1 = x2;
            f1=f2;
}
```

```
        if (f2 * f1 < 0){
    f0=f2;
            x0 = x2;
}
        if (f2==0) break;
        n++;
    }
    cout << "regulafalsi   " << n - 1 << "  iteraciones   "
          << "raiz= " << x2 <<
        " funcion= " << f2 << endl<<endl<<endl;
    return 0;
}
```

El método de la secante es una variación del método de la régula falsi, que consiste en no requerir horquillado y tomar las aproximaciones sucesivas a la raíz. Hemos codificado este método en la siguiente forma

```
int secante(double x0, double x1)
{
    double          f(double x);
    cout << "    Metodo secante" << endl;
    cout.precision(nprec);
    cout << "   x0= " << x0 << " x1= " << x1 << endl;
    cout << " iteracion" << "             x  " << "  f(x)  " << endl;
    int             n = 1;
    double          x2 = 0;
    double f0=f(x0), f1=f(x1), f2=1;
    while ((abs(f2) >= fprec)||abs(x2-x1)>=prec) {
        if(n>=numiter) {
    cout<<"numero maximo de iteraciones excedido en secante"<<endl;
            return 1;
        }
        x2 = x1 - f1 * (x1 - x0) / (f1 - f0);
        f2=f(x2);
        cout << setw(10) << n << "   " << setw(10) << x2 << "  "
          << f2 << endl;
        x0 = x1;
        x1 = x2;
        f0=f1;
        f1=f2;
        n++;
    }
    cout << "secante       " << n - 1 << "  iteraciones   "
```

```
      << "raiz= " << x2 << " funcion= " << f2 << endl<<endl<<endl;
    return 0;
}
```

Los métodos de la regula falsi y de la secante son conceptualmente distintos, pues el primero es un método de horquillado que siempre converge mientras que el segundo es un método iterativo de dos puntos que puede diverger, aunque cuando converge lo puede hacer más rápidamente que el método de la régula falsi.

**Ejercicio 2.** *Estudiar todas las raíces reales de las dos funciones propuestas en el ejercicio 1 por los métodos de la regula falsi y la secante.*

### 3.1.3. Método de Newton

El método de Newton es un método que converge muy rápidamente ya que es de segundo orden. Sin embargo puede diverger rápidamente si el punto inicial no es adecuado. Su principal inconveniente es que requiere el cálculo de la derivada de la función, lo cual puede ser tedioso si la función es complicada. Hemos codificado el método de Newton de la siguiente forma:

```
int newton(double x1)
{
    int             n = 0;
    double          x2 = x1+10;
    double          f1 = f(x1);
    cout << "    Metodo de Newton" << endl;
    cout.precision(nprec);
    cout << " iteracion" << "             x  " << "  f(x)  " << endl;
    cout << setw(10) << 0 << "   " << setw(10) << x1 <<
         "  " << f1 << endl;
    while ((abs(f1) >= fprec)||abs(x2-x1)>=prec) {
      if(n>=numiter) {
cout<<"numero maximo de iteraciones excedido en newton"<<endl;
        return 1;
        }
        x2 = x1 - f1 / fd(x1);
        x1 = x2;
        f1 = f(x1);
        cout << setw(10) << n+1 << "   " << setw(10) <<
             x1 << "  " << f1 << endl;
        n++;
    }
    cout << "newton        " << n  << "  iteraciones   " <<
      "raiz= " << x1 << " funcion= " << f1 << endl<<endl<<endl;
    return 0;
```

```
}
```

Cuando el método de Newton diverge, una posibilidad es pasar a un método de horquillado como bisección, régula falsi o Muller. El método de Newton sirve también para calcular raíces complejas, como veremos posteriormente.

**Ejercicio 3:** *Estudiar todas las raíces reales de las funciones propuestas en el ejercicio 1 por el Método de Newton.*

## 3.2. Método de Müller

Un método que converge casi tan deprisa como el de Newton y que además es robusto, es el método de Müller, que consiste en interpolar tres puntos cercanos a la raíz por un polinomio de segundo grado y calcular como nueva aproximación de la raíz, la raíz de este polinomio. Hemos codificado este método en la siguiente forma:

```
int muller(double x0, double x1)
{
    double          f(double x);
    cout << "    Metodo Muller" << endl;
    cout.precision(nprec);
    cout << " iteracion" << "             x  " << "  f(x)  " << endl;

    int             n = 1;
    double          x2 = 0, x3 = 0;
    double          f0 = f(x0), f1 = f(x1), f2 = f1, a, b, c, d, h;

    x2 = x1 - f1 * (x1 - x0) / (f1 - f0);
    f2 = f(x2);
    cout << setw(10) << n << "   " << setw(10) << x2 <<
        "  " << f2 << endl;
    n = 2;
    while ((abs(f2) >= fprec)||abs(x2-x1)>=prec) {
        if (n >= numiter) {
            cout << " numero maximo de iteraciones excedido en muller" << endl;
            return 1;
        }
        a = (x1 - x0) * f2 + (x0 - x2) * f1 + (x2 - x1) * f0;
        a = a / ((x2 - x1) * (x1 - x0) * (x1 - x0));
        b = (x1 - x0) * (2 * x2 - x1 - x0) * f2
         - (x2 - x0) * (x2 - x0) * f1 + (x2 - x1) * (x2 - x1) * f0;
        b = b / ((x2 - x1) * (x1 - x0) * (x1 - x0));
```

```
        c = (x2 - x0) * f2 / (x1 - x0);
        d = sqrt(b * b - 4 * a * c);
        if (b > 0)
            h = -2 * c / (b + d);
        else
            h = -2 * c / (b - d);
        x3 = x2 + h;
        x0 = x1;
        f0 = f1;
        x1 = x2;
        f1 = f2;
        x2 = x3;
        f2 = f(x2);
        cout << setw(10) << n << "   " << setw(10) << x2
            << "  " << f2 << endl;
        n++;
    }
    cout << "muller       " << n - 1 << "  iteraciones   " <<
            "raiz= " << x2 << " funcion= " << f2 << endl<<endl<<endl;
    return 0;
}
```

**Ejercicio 4:** *Estudiar todas las raíces reales de las funciones propuestas en el ejercicio 1 por el Método de Müller.*

## 3.3. Números complejos en C++

Para poder utilizar números complejos en C++ hay que incluir la cabecera *complex:*

```
#include <complex>
```

El tipo *complex* se comporta como un un tipo nativo, aunque en realidad es una clase, o sea algo muy similar a las estructuras del C. Además, es lo que se denomina una clase patrón (template class), que permite el definir números complejos cuyas partes real e imaginaria pueden ser de cualquiera de los tipos nativos: *int, double*,... No vamos a entrar por ahora en como se construye o define una clase, sino sólo en como se utiliza la clase *complex*. En primer lugar definimos un número complejo $c = x + iy$, donde $x$ *e* $y$ son las partes real e imaginarias, respectivamente. Las funciones incluidas en la librería estándar son las siguientes:

1. Las operaciones aritméticas usuales: +, -, *, /.

2. Las operaciones aritméticas compuestas: +=, -=, *=, /=.

3. Los operadores lógicos: == , !=.

4. Las funciones que dan el módulo, argumento y partes real e imaginaria de un número complejo: *norm(c)*=$x^2+y^2$, *abs(c)*=$\sqrt{x^2+y^2}$, *conj*(c)=$x-iy$, *real(c), imag(c), arg(c).*

5. Las funciones exponencial y logarítmica: *exp(c), log(c), log10(c).*

6. Las funciones de potencia y raíz cuadrada: *pow(c1,c2), pow(c1,real), pow(c1,imaginario), sqrt(c).*

7. Las funciones trigonométricas: *sin(c), cos(c), tan(c).*

8. Las funciones hiperbólicas: *sinh(c), cosh(c), tanh(c).*

La existencia de todas estas funciones deben de verificarse para cada sistema operativo y compilador dados, puesto que las realizaciones de los compiladores no suelen estar todavía completamente de acuerdo con el estándar. La versión 2.95 del compilador gcc no tiene definidas las funciones t*an(c), tanh(c)* y *log10(c)*. Tampoco admite la salida de un número complejo con *cout*.

La forma de utilizar los números complejos es la siguiente: Supongamos que queremos declarar números complejos formados por parejas de números enteros, c1 y c2, otros formados por parejas de números reales en simple precisión c3 y c4, y finalmente otros números complejos formados por parejas de números en doble precisión c5 y c6. Los declaramos de la siguiente forma:

```
complex<int> c1,c2;
complex<float> c3, c4;
complex<double> c5, c6;
```

La presencia de los paréntesis angulares <> indica que complex es una clase patrón, que se puede especificar para un tipo dado, poniendo dicho tipo entre paréntesis angulares <>.

Para inicializar c3= 3.+4.i, por ejemplo, escribimos

```
c3 = complex<float>(3.,4.)
```

Igualmente podemos inicializar

```
c5 = complex<double>(6.,4.)
c2 = complex<int>(1,2)
```

Damos a continuación algunos ejemplos de utilización de funciones:

```
c4 = sin(c3)*sqrt(c3);
c5 = pow(c4,c3);
c6 = log(c5);
complex<double> c7 = c6/c5*c4;
```

Las funciones de la clase *complex* están definidas para el compilador *gcc* en *la librería /usr/include/g++-3/std/* en los ficheros *complex.h* y *complex.cc*.

**Ejercicio:** *Escribid un programa en el que se verifiquen las operaciones con números complejos y las diferentes funciones de la librería matemática descritas en este apartado, utilizando las igualdades matemáticas de vuestra preferencia (p. e.* $cos^2(z)+sin^2(z)=1$ *).*

## 3.4. Aplicación de los métodos de Newton y Müller a raíces complejas

Los métodos de Newton y Müller son válidos para refinar raíces en el plano complejo. Hay que dar el punto o los puntos iniciales (Müller utiliza dos puntos iniciales) suficientemente próximos a la raíz y complejos. La localización gráfica de las raíces es complicada en este caso. La representación gráfica del módulo de la función en tres dimensiones puede ser de gran ayuda para encontrar los ceros. En el programa `raicescomplejas.cpp` se codifican estos dos métodos:

```
// Programa de calculo de raices complejas por los metodos de Newton y Muller
#include <math.h>
#include <iomanip>
#include <iostream>
#include <complex>
using namespace std;
//prec:error en la raiz
// fprec:valor maximo de la funcion en el valor aceptado para la raiz
// numiter:numero maximo de iteraciones
const double    prec = 1.e-12, fprec = 1.e-12;
const int       nprec = 10, nwidth = 16;
const int       numiter = 100;
const char     *tab = "\t";
//f: funcion, fd:derivada.

complex < double >f(complex < double >);
complex < double >fd(complex < double >);
int             muller(complex < double >, complex < double >);
int             newton(complex < double >);

complex < double > f(complex < double >x)
{
return complex < double >(16, 0) * x * x * x * x -
                complex < double >(40, 0) * x * x * x +
                complex < double >(5, 0) * x * x +
                complex < double >(20, 0) * x +
                complex < double >(6, 0);
}

complex < double > fd(complex < double >x)
{
return complex < double >(64, 0) * x * x * x -
                complex < double >(120, 0) * x * x +
                complex < double >(10, 0) * x +
```

```
                complex < double >(20, 0);
}


int main()
{
/*
 * Programa para comparar diversos métodos de calculo de raices
 * complejas
 */
//x0, x1 puntos iniciales Muller; x1 punto inicial Newton


        complex < double > x0 = complex < double >(0.5, 0.5),
                    raiz;
muller(x0, x1);
newton(x0);
return 0;
}


int newton(complex < double >x1)
{
int             n = 1;
complex < double >x2;
complex < double >f1 = f(x1);
cout << "    Metodo de Newton" << endl;
cout << " Punto inicial= " << x1 << endl;
cout.precision(nprec);
cout << " iteracion" << "             x  " << "  f(x)  " << endl;
cout << setw(10) << 0 << "   " << setw(10) << x1 << "  " << f1 << endl;
while ((abs(f1) >= fprec) || (abs(x2 - x1) > prec)) {
if (n >= numiter) {
cout << " numero maximo de iteraciones excedido en muller" << endl;
return 1;
}
x2 = x1 - f1 / fd(x1);
x1 = x2;
f1 = f(x1);
cout << setw(10) << n << "   " << setw(10) << x1
<< "  " << f1 << endl;
n++;
}
```

```
cout << "newton        " << n - 1 << "  iteraciones   " << "raiz= "
<< x1 << " funcion= " << f1 << endl;
return 0;
}


int muller(complex < double >x0, complex < double >x1)
{
complex < double >f(complex < double >x);
cout << "    Metodo Muller" << endl;
cout.precision(nprec);
cout << " iteracion" << "             x  " << "  f(x)  " << endl;

int             n = 1;
complex < double >x2 = (0, 0), x3 = (0, 0);
complex < double >f0 = f(x0), f1 = f(x1), f2 = f1, a, b, c, d,
                h;


x2 = x1 - f1 * (x1 - x0) / (f1 - f0);
f2 = f(x2);
cout << setw(10) << n << "   " << setw(10) << x2 << "  " << f(x2) << endl;
n = 2;
while ((abs(f2) >= fprec) || (abs(x2 - x1) >= prec)) {
if (n >= numiter) {
cout << " numero maximo de iteraciones excedido en muller" << endl;
return 1;
}
a = (x1 - x0) * f2 + (x0 - x2) * f1 + (x2 - x1) * f0;
a = a / ((x2 - x1) * (x1 - x0) * (x1 - x0));
b = (x1 - x0) * (complex < double >(2, 0) * x2 - x1 - x0) *
f2 - (x2 - x0) * (x2 - x0) * f1 + (x2 - x1) * (x2 - x1) * f0;
b = b / ((x2 - x1) * (x1 - x0) * (x1 - x0));
c = (x2 - x0) * f2 / (x1 - x0);
d = sqrt(b * b - complex < double >(4, 0) * a * c);
if (abs(b + d) > abs(b - d))
h = -complex < double >(2, 0) * c / (b + d);
else
h = complex < double >(-2, 0) * c / (b - d);
x3 = x2 + h;
x0 = x1;
f0 = f1;
x1 = x2;
```

```
f1 = f2;
x2 = x3;
f2 = f(x2);
cout << setw(10) << n << "   " << setw(10) << x2 << "  " << f2 << endl;
n++;
}
cout << "muller       " << n - 1 << "  iteraciones   " <<
"raiz= " << x2 << " funcion= " << f2 << endl;
return 0;
}
```

Las imágenes fractales se obtienen estudiando la convergencia en el plano complejo del método de Newton. Por ejemplo, el conocido conjunto de Mandelbrot es la zona de convergencia de l método de Newton para las raíces de la ecuación $z^2 + c = 0$. Cuando se colorean las zonas de divergencia de acuerdo con la velocidad de divergencia se obtienen llamativas imágenes.

**Ejercicio:** *Compilad y ejecutad el programa. Cambiad la función incluidas en el programa por otra función de vuestra preferencia (que tenga raíces) y calculadlas.*

## 3.5. Ejercicios a presentar como memoria

1. Obtener todas las raíces de la ecuación

$$x^5 - x^4 - 5x^3 + 5x^2 + 6x - 6 = 0$$

   por los métodos de bisección, regula falsi, secante y Newton, con una precisión de $10^{-10}$ para la raíz. La función debe de anularse con la misma precisión. Localizad las raíces aproximadamente utilizando gnuplot.

2. Obtener todas las raíces de la ecuación

$$x^2 - \cos(x) = 0$$

   por los métodos de bisección, régula falsi, secante y Newton, con una precisión de $10^{-10}$ para la raíz. La función debe de anularse con la misma precisión. Localizad las raíces aproximadamente utilizando gnuplot.

3. Obtener una raíz de la ecuación $\arctan(x) = 0$ en el intervalo $[-5, 5]$ mediante los métodos de bisección, régula falsi y Newton con una precisión de $10^{-9}$. Comentad y justificad el resultado.

4. Cuando el método de Newton-Raphson falla, en el sentido de que una iteración del método produce un valor fuera del intervalo donde se encuentra la raíz, un procedimiento posible es realizar una iteración del método de bisección, y volver a realizar una iteración de Newton. Si el método de Newton produce un valor que de nuevo está fuera del nuevo intervalo

obtenido por bisección, realizamos una nueva iteración de bisección, hasta que el método de Newton produzca una serie convergente. Programad este método mediante una función que llamaréis newtonbisec y aplicadlo a la ecuación del ejercicio anterior.

5. El método de Steffensen es un método iterativo similar al de Newton-Raphson, pero que no utiliza la derivada, que se calcula aproximadamente como un cociente incremental. Este método viene definido por la relación de recurrencia

$$x_{n+1} = x_n - \frac{f^2(x_n)}{f(x_n + f(x_n)) - f(x_n)}$$

Programad este método y aplicadlo a la ecuación del segundo ejercicio. Haced un comentario de un máximo de 4 líneas sobre la comparación de este método y el de Newton.

6. Encontrar las raíces complejas de la ecuación

$$x^4 + 1 = 0$$

mediante el método de Newton o el de Müller. Tomad diferentes puntos de partida para obtener cada una de las raíces.