# Subrutinas en Fortran para la resolución de ecuaciones no lineales de una variable

Pablo Santamaría

v0.5 (Febrero 2021)

## Introducción

En general, las raíces de una ecuación no lineal $f(x) = 0$ no pueden ser obtenidas por fórmulas explícitas cerradas, con lo que no es posible obtenerlas en forma exacta. De este modo, para resolver la ecuación nos vemos obligados a obtener soluciones aproximadas a través de algún método numérico.

Estos métodos son *iterativos*, esto es, a partir de una (o más) aproximación inicial $x_0$ para la raíz, generan una sucesión de aproximaciones $x_1, x_2, \ldots$ que esperamos convergan al valor de la raíz $\alpha$ buscada. El proceso iterativo se continúa hasta que la aproximación se encuentra próxima a la raíz dentro de una tolerancia $\epsilon > 0$ preestablecida. Como la raíz no es conocida, dicha proximidad, medida por el error absoluto $|x_{n+1} - \alpha|$, no puede ser computada. Sin un conocimiento adicional de la función $f(x)$ o su raíz, el mejor criterio para detener las iteraciones consiste en proceder hasta que la desigualdad

$$\frac{|x_{n+1} - x_n|}{|x_{n+1}|} \leq \epsilon$$

se satisfaga, dado que esta condición *estima* en cada paso el error relativo. Ahora bien, puede ocurrir en ciertas circunstancias que la desigualdad anterior nunca se satisfaga, ya sea por que la sucesión de aproximaciones diverge o bien que la tolerancia escogida no es razonable. En tal caso el método iterativo no se detiene nunca. Para evitar este problema consideramos además un número máximo de iteraciones a realizarse. Si este número es excedido entonces el problema debe ser analizado con más cuidado.

¿Cómo se escoge un valor correcto para las aproximaciones iniciales requeridas por los métodos? No existe una respuesta general para esta cuestión. Para el método de bisección es suficiente conocer un intervalo que contenga la raíz, pero para el método de Newton, por ejemplo, la aproximación tiene que estar suficientemente cercana a la raíz para que converja. En cualquier caso primeras aproximaciones iniciales para las raíces pueden ser obtenidas graficando la función $f(x)$.

En las siguientes secciones presentamos implementaciones de los métodos numéricos usuales como subrutinas Fortran. Con el fin de proporcionar subrutinas de propósito general, las mismas tienen entre sus argumentos a la función $f$ involucrada, la cual pude ser entonces implementada por el usuario como un subprograma `function` con el nombre que desee utilizar. Otros argumentos que requieren estas subrutinas son los valores para las aproximaciones iniciales que necesite el método, una tolerancia para la aproximación final de la raíz y un número máximo de iteraciones. La raíz aproximada es devuelta en otro de los argumentos. Dado que el método puede fallar utilizamos también una variable entera como *clave de error* para advertir al programa principal. Por convención tomaremos que si dicha clave es igual a cero, entonces el método funcionó correctamente y el valor devuelto es la raíz aproximada dentro de la tolerancia preescrita. En cambio, si la clave de error es distinta de cero, entonces ocurrió un error. La naturaleza del error dependerá del método, pero un error común a todos ellos es que el número máximo de iteraciones fue alcanzado.

Con el fin de aprovechar la capacidad del Fortran moderno de detectar errores de tipo en los argumentos al llamar a las subrutinas, debemos hacer *explícita* la interface de las mismas. La forma más simple y poderosa de efectuar ésto consiste en agrupar las subrutinas en un módulo, al que denominamos `roots`. Por cuestiones pedagógicas, en el módulo definimos un parámetro lógico, denominado `debug`, tal que cuando es asignado a `.true.` vuelca información sobre las iteracciones realizadas por el método llamado. Para el uso del módulo en programas reales, tal parámetro debería ser asignado siempre a `.false.`. Finalmente, en nuestra implementación todas las cantidades reales son de la clase de *doble precisión*, la cual, para máxima flexibilidad, está definida en forma paramétrica utilizando el modulo intrínseco `iso_fortran_env` y el alias `wp` para asignar la *precisión de trabajo* a dicho valor. Así, el uso del módulo por parte del programa principal,

o la unidad de programa, que realice la llamada a una subrutina del mismo, requiere de las siguientes dos sentencias:

```
use iso_fortran_env, only: wp => real64
use roots, only: nombre_de_la_subrutina
```

El código completo del módulo `roots` es dado al final de estas notas, mientras que las siguientes secciones indican la interfaz de cada subrutina y ejemplos de su uso.

## Método de bisección

El método de bisección comienza con un intervalo $[a, b]$ que contiene a la raíz. Entonces se computa el punto medio $x_1 = (a + b)/2$ del mismo y se determina en cual de los dos subintervalos $[a, x_1]$ o $[x_1, b]$ se encuentra la raíz analizando el cambio de signo de $f(x)$ en los extremos. El procedimiento se vuelve a repetir con el nuevo intervalo así determinado. Es claro que la raíz es acotada en cada paso por el intervalo así generado y que una estimación del error cometido en aproximar la raíz por el punto medio de dicho intervalo es igual a la mitad de la longitud del mismo. Esta estimación es utilizada, en la implementación del método, como criterio de paro para la sucesión de aproximaciones.

En nuestro módulo `roots`, el método de bisección es implementado en la subrutina `biseccion` cuya llamada tiene la forma

```
call biseccion ( f, a, b, n, tol, raiz, clave )
```

donde los argumentos son:

`f`: La función que define la ecuación $f(x) = 0$, la cual es implementada como una `function` de una variable real, de precisión `wp`, que devuelve un valor real, de precisión `wp`.

`a,b`: Datos de entrada reales, de precisión `wp`, que constituyen los extremos del intervalo inicial $[a, b]$ con $f(a)f(b) < 0$.

`n`: Dato de entrada/salida entero. Como dato de entrada indica el número de iteraciones máximo a realizar y, como dato de salida, el número de iteraciones realizadas.

`tol`: Es un dato de entrada real, de precisión `wp`, que da una tolerancia para el error absoluto.

`raiz`: Es un dato de salida real, de precisión `wp`, que retorna la aproximación buscada a la raíz.

`clave`: Es un dato de salida entero que permite al usuario verificar si el método funcionó correctamente. Los valores posibles y su significado son los siguientes:

- 0. El método convergió correctamente: el valor retornado en `raiz` es una raíz de $f$ dentro de la tolerancia especificada,
- $> 0$. El número de iteraciones máximo se ha alcanzado. Por lo tanto, el valor devuelto en `raiz` no es un dato útil en este caso y el usuario debe examinar con más cuidado la ecuación.
- $< 0$. El método no puede proceder debido a que $f$ es de igual signo en $a$ y $b$.

## Método de Newton–Raphson

El método de Newton comienza con una aproximación inicial $x_0$ dada, a partir de la cual se genera la sucesión de aproximaciones $x_1, x_2, \ldots$, siendo $x_{n+1}$ la abscisa del punto de intersección del eje $x$ con la recta tangente a $f(x)$ que pasa por el punto $(x_n, f(x_n))$. Esto conduce a la fórmula de iteración

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \ldots$$

En nuestro módulo `roots`, el método de Newton es implementado en la subrutina `newton` cuya llamada se describe a continuación. Nótese que se requiere también como argumento no sólo la funcion $f(x)$, sino también su derivada $f'(x)$, la cual debe ser implementada por el usuario, al igual que $f(x)$, como una `function`.

```
call newton ( f, df, x0, n, tol, raiz, clave )
```

Los argumentos son:

`f`: La función que define la ecuación $f(x) = 0$, la cual es implementada como una `function` de una variable real. de precisión `wp`, que devuelve un valor real, de precisión `wp`.

`df`: La derivada de la función que define la ecuación $f(x) = 0$, la cual es implementada como una `function` de una variable real, de precisión `wp`, que devuelve un valor real, de precisión `wp`.

`x0`: Dato de entrada real, de precisión `wp`, que constituye una aproximación inicial para la raíz a determinar.

`n`: Dato de entrada/salida entero. Como dato de entrada indica el número de iteraciones máximo a realizar y, como dato de salida, el número de iteraciones realizadas.

`tol`: Es un dato de entrada real, de precisión `wp`, que da una tolerancia para el error relativo.

`raiz`: Es un dato de salida real, de precisión `wp`, que retorna la aproximación buscada a la raíz.

`clave`: Es un dato de salida entero que permite al usuario verificar si el método funcionó correctamente. Los valores posibles y su significado son los siguientes:

- 0. El método convergió correctamente: el valor retornado en `raiz` es una raíz de $f$ dentro de la tolerancia especificada,
- $> 0$. El número de iteraciones máximo se ha alcanzado. Por lo tanto, el valor devuelto en `raiz` no es un dato útil en este caso y el usuario debe examinar con más cuidado la ecuación.

## Método de Newton para ecuaciones algebraicas

En el caso particular en que $f(x)$ es un polinomio, el método de Newton puede ser eficientemente implementado si la evaluación de $f(x_n)$ (y su derivada) es realizada por el *método iterativo de Horner*. En efecto, supongamos que $f(x)$ es un polinomio de grado $m$:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m,$$

la evaluación de $f(x_n)$ por la regla de Horner procede computando

$$\begin{cases} b_m = a_m \\ b_k = a_k + b_{k+1} x_n \quad k = m-1, \ldots, 0 \end{cases}$$

siendo, entonces

$$b_0 = f(x_n).$$

En tanto que $f'(x_n)$ es computada haciendo

$$\begin{cases} c_m = b_m \\ c_k = b_k + c_{k+1} x_n \quad k = m-1, \ldots, 1 \end{cases}$$

siendo, entonces

$$c_1 = f'(x_n).$$

El método de Newton se reduce así a

$$x_{n+1} = x_n - \frac{b_0}{c_1}$$

El procedimiento resultante se conoce a menudo como *método de Birge–Vieta*.

En nuestro módulo `roots`, el método es implementado en la subrutina `birge_vieta` cuya llamada se describe a continuación Nótese que los coeficientes del polinomio son pasados en un arreglo $a$ de $(m+1)$ componentes, siendo $m$ el grado del polinomio, de manera tal que $a(0) = a_0, a(1) = a_1, \ldots, a(m) = a_m$.

```
call birge_vieta ( a, x0, n, tol, raiz, clave )
```

Los argumentos son:

`a`: Dato de entrada consistente en un vector de $m+1$ valores reales, de precisión `wp`, que contiene los coeficientes del polinomio de grado $m$, de manera que $a(i) = a_i$ para $i = 0, \ldots, m$.

`x0`: Dato de entrada real, de precisión `wp`, que constituye una aproximación inicial para la raíz a determinar.

`n`: Dato de entrada/salida entero. Como dato de entrada indica el número de iteraciones máximo a realizar y, como dato de salida, el número de iteraciones realizadas.

`tol`: Es un dato de entrada real, de precisión `wp`, que da una tolerancia para el error relativo.

`raiz`: Es un dato de salida real, de precisión `wp`, que retorna la aproximación buscada a la raíz.

`clave`: Es un dato de salida entero que permite al usuario verificar si el método funcionó correctamente. Los valores posibles y su significado son los siguientes:

- 0. El método convergió correctamente: el valor retornado en `raiz` es una raíz de $f$ dentro de la tolerancia especificada,
- $> 0$. El número de iteraciones máximo se ha alcanzado. Por lo tanto, el valor devuelto en `raiz` no es un dato útil en este caso y el usuario debe examinar con más cuidado la ecuación.

## Método de la secante

El método de la secante procede a partir de *dos* aproximaciones iniciales, $x_{-1}$ y $x_0$, obteniendo la aproximación $x_{n+1}$ como la abscisa del punto de intersección del eje $x$ con la recta secante que pasa por los puntos $(x_{n-1}, f(x_{n-1}))$ y $(x_n, f(x_n))$. La fórmula de iteración es entonces

$$x_{n+1} = x_n - f(x_n)\frac{(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}, \quad n = 0, 1, 2, \ldots$$

El método de la secante, si bien no converge tan rápido como Newton, tiene la gran ventaja de no requerir la derivada de la función. Eso sí, ahora se necesitan dos aproximaciones iniciales para iniciar el método.

En nuestro módulo `roots`, el método de la secante es implementado en la subrutina `secante` cuya llamada tiene la forma:

```
call secante ( f, x0, x1, n, tol, raiz, clave )
```

Los argumentos son:

`f`: La función que define la ecuación $f(x) = 0$, la cual es implementada como una `function` de una variable real. de precisión `wp`, que devuelve un valor real, de precisión `wp`.

`df`: La derivada de la función que define la ecuación $f(x) = 0$, la cual es implementada como una `function` de una variable real, de precisión `wp`, que devuelve un valor real, de precisión `wp`.

`x0, x1`: Datos de entrada reales, de precisión `wp`, que constituye dos aproximaciones iniciales para la raíz a determinar.

`n`: Dato de entrada/salida entero. Como dato de entrada indica el número de iteraciones máximo a realizar y, como dato de salida, el número de iteraciones realizadas.

`tol`: Es un dato de entrada real, de precisión `wp`, que da una tolerancia para el error relativo.

`raiz`: Es un dato de salida real, de precisión `wp`, que retorna la aproximación buscada a la raíz.

`clave`: Es un dato de salida entero que permite al usuario verificar si el método funcionó correctamente. Los valores posibles y su significado son los siguientes:

- 0. El método convergió correctamente: el valor retornado en `raiz` es una raíz de $f$ dentro de la tolerancia especificada,
- $> 0$. El número de iteraciones máximo se ha alcanzado. Por lo tanto, el valor devuelto en `raiz` no es un dato útil en este caso y el usuario debe examinar con más cuidado la ecuación.

## Iteración de punto fijo

El método de punto fijo requiere que se reescriba la ecuación $f(x) = 0$ en la forma

$$x = \phi(x)$$

Entonces, a partir de una aproximación inicial $x_0$, se obtiene la sucesión de aproximaciones $x_1, x_2, ...$ según

$$x_{n+1} = \phi(x_n), \quad n = 0, 1, \ldots$$

En nuestro módulo `roots`, el método de punto fijo es implementado en la subrutina `punto_fijo` cuya llamada se describe a continuación. Nótese que en la implementación de la función de iteración la que debe pasarse por argumento.

```
call punto_fijo ( f, x0, n, tol, raiz, clave )
```

Los argumentos son:

`f`: La función que define la ecuación de iteración $x = f(x)$, la cual es implementada como una `function` de una variable real. de precisión `wp`, que devuelve un valor real, de precisión `wp`.

`x0`: Dato de entrada real, de precisión `wp`, que constituye una aproximación inicial para la raíz a determinar.

`n`: Dato de entrada/salida entero. Como dato de entrada indica el número de iteraciones máximo a realizar y, como dato de salida, el número de iteraciones realizadas.

`tol`: Es un dato de entrada real, de precisión `wp`, que da una tolerancia para el error relativo.

`raiz`: Es un dato de salida real, de precisión `wp`, que retorna la aproximación buscada a la raíz.

`clave`: Es un dato de salida entero que permite al usuario verificar si el método funcionó correctamente. Los valores posibles y su significado son los siguientes:

- 0. El método convergió correctamente: el valor retornado en `raiz` es una raíz de $f$ dentro de la tolerancia especificada,

- $> 0$. El número de iteraciones máximo se ha alcanzado. Por lo tanto, el valor devuelto en `raiz` no es un dato útil en este caso y el usuario debe examinar con más cuidado la ecuación.

## Método de propósito general

Un método de propósito general que permita determinar aproximaciones precisas a cada raíz de una ecuación no lineal, $f(x) = 0$, debería poder acotar la raíz dentro intervalos sucesivos cada vez menores y, a la vez, ser rápidamente convergente. Sabemos que el método de bisección acota sucesivamente la raíz dentro de un intervalo cada vez menor, pero la convergencia es lenta. Por otra parte, el método de Newton o el método de la secante convergen más rápidamente que el método de bisección, pero abandonan toda posible acotación de la raíz y dicha convergencia ocurrirá sólo si la aproximación inicial está suficientemente próxima a la raíz buscada. Es claro entonces que el algoritmo deseado debe ser un procedimiento híbrido que combine las ventajas de ambos métodos. En particular, uno de tales algoritmos, es el llamado *método de Dekker*, el cual combina el método de bisección con el método de la secante de forma tal que si la siguiente aproximación dada por el algoritmo de la secante cae fuera de un intervalo alrededor de la solución de longitud menor al intervalo del paso anterior, entonces se utiliza la aproximación correspondiente al método de bisección para dicha iteración.

En circunstancias normales, la ecuación no lineal a resolver por el método de Dekker está dada por una función $f(x)$ continua sobre un intervalo $[a, b]$ para el cual $f(a)f(b) < 0$, lo cual asegura la existencia de, al menos, una raíz en dicho intervalo. El método genera entonces, en cada iteración, un nuevo intervalo $[a, b]$ con $f(a)f(b) < 0$, que decrece en longitud respecto de la iteración anterior, y tal que $a$ constituye una mejor aproximación a la raíz que $b$, en el sentido de que $|f(a)| \leq |f(b)|$, y una mejor aproximación que el punto medio $m = (a + b)/2$. Las iteraciones proceden hasta que se satisface el criterio de convergencia:

$$\frac{|b - a|}{2} \leq \text{máx}\,\{\epsilon_{\text{abs}}, \epsilon_{\text{rel}}|a|\},$$

donde $\epsilon_{\text{abs}}$ y $\epsilon_{\text{rel}}$ son tolerancias prefijadas para el error absoluto y relativo, respectivamente. Para comprender lo que este test significa, supongamos primero que $\epsilon_{\text{rel}} = 0$, entonces la condición se reduce a

$$\frac{|b - a|}{2} \leq \epsilon_{\text{abs}}$$

la cual es la condición usual para asegurar que el punto medio $m$ no está alejado de la raíz en más de $\epsilon_{\text{abs}}$. Pero, puesto que se asume $a$ es una mejor aproximación a la raíz que $m$, el criterio proporciona también un test para la precisión $a$ como aproximación de la raíz. Nótese que si, en circunstancias desfavorables, $a$ no es mejor aproximación que $m$, el test implica que $a$ está alejado de la raíz no mas de $2\,\epsilon_{\text{abs}}$. Si por otra parte, suponemos ahora que $\epsilon_{\text{abs}} = 0$ y el criterio fuera

$$\frac{|b - a|}{2} \leq |m|\epsilon_{\text{err}}$$

el mismo sería un test para el error relativo para la aproximación $m$ de la raíz. Como se asume que $a$ es una mejor aproximación que $m$, el criterio es utilizado con $a$ en vez de $m$. De este modo la condición mixta de testeo del error absoluto y relativo proporciona robustez al método puesto que si la raíz es próxima a cero, un test basado únicamente en el error relativo no resulta apropiado. Además, para evitar un bucle infinito en circunstancias desfavorables, las iteraciones son detenidas si el criterio no se satisface después de un determinado número máximo de evaluaciones de $f(x)$. El algoritmo es suficientemente robusto para tratar con casos excepcionales e informar en consecuencia, como ser la presencia, en el intervalo considerado, de un polo o un mínimo local en vez de una raíz.

En nuestro módulo, el método de Dekker es implementado por la subrutina `fzero` cuya llamada tiene la forma:

```
call fzero ( f, a, b, eps_rel, eps_abs, clave )
```

Los argumentos son:

`f`: La función que define la ecuación $f(x) = 0$, la cual es implementada como una `function` de una variable real. de precisión `wp`, que devuelve un valor real, de precisión `wp`.

`df`: La derivada de la función que define la ecuación $f(x) = 0$, la cual es implementada como una `function`

`a, b`: Datos de entrada y salida reales, de precisión `wp`. Como datos de entrada constituyen los extremos del intervalo inicial $[a, b]$ con $f(a)f(b) < 0$. Como datos de salida, constituyen un intervalo que encierra a la raíz, siendo $a$ una mejor aproximación a $b$ de la raíz, en el sentido de que $|f(a)| \leq |f(b)|$ (si éste no es el caso, la subrutina intercambia los valores de $a$ y $b$ de manera tal que dicha condición se mantenga).

`eps_rel`: es un dato de entrada real, de precisión `wp`, que da una tolerancia para el error relativo. La subrutina se asegura que el valor ingresado nunca sea menor que $2\,\epsilon_{\mathrm{M}}$, siendo $\epsilon_{\mathrm{M}}$ el *epsilon de la máquina*.

`eps_abs`: es un dato de entrada real, de precisión `wp`, que da una tolerancia para el error absoluto. Si el intervalor inicial $[a, b]$ contiene al cero, entonces este valor no debe ser nulo.

`clave`: es un dato de salida entero que permite al usuario verificar si el método funcionó correctamente. Los valores posibles y su significado son los siguientes:

- 0. OK. El método funcionó correctamente: el valor retornado en $a$ es una raíz de $f$ dentro de la tolerancia especificada, la raíz se encuentra dentro del intervalo $[a, b]$ devuelto, la función cambia de signo en los extremos de dicho intervalo y $f(x)$ decrece en magnitud conforme el intervalo decrece su longitud.
- 1. OK. El valor retornado en $a$ cumple la condición $f(a) = 0$, pero el intervalo $[a, b]$ no necesariamente encierra a la raíz dentro de la tolerancia prefijada (por lo tanto, el valor devuelto en $b$ no es un dato útil en este caso).
- $-1$. Problemas. El intervalo retornado $[a, b]$ se encuentra dentro de la tolerancia prefijada y la función cambia de signo en los extremos de dicho intervalo, pero su valor se fue incrementando conforme los intervalos $[a, b]$ se achicaban. Es probable que $a$ se encuentre próximo a un punto singular de $f(x)$.
- $-2$. Problemas. El intervalo retornado $[a, b]$ se encuentra dentro de la tolerancia prefijada pero la función no cambia de signo en sus extremos. El usuario debe entonces examinar con más cuidado si $a$ está cerca de un mínimo local de $f(x)$ o de una de una raíz de multiplicidad par, o bien ninguna de estas situaciones.
- $-3$. Problemas. Se alcanza un número máximo de evaluaciones ($> 500$) de $f(x)$ sin poder determinar la raíz. El usuario debe considerar un intervalo inicial más chico o determinar si realmente hay una raíz en él.

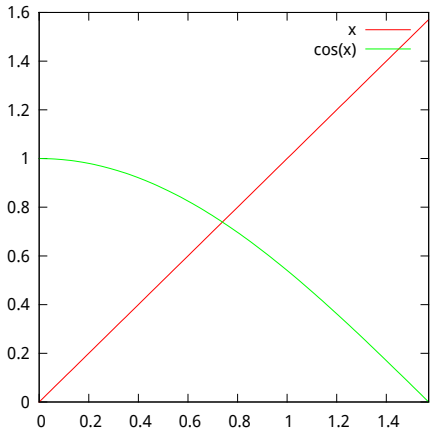


**Figura 1.** Determinación de una aproximación inicial de la raíz de la función $\cos x - x$.

# Ejemplos

Como primer ejemplo del uso de nuestro módulo `roots`, consideremos el problema de determinar la raíz de la ecuación

$$\cos x - x = 0.$$

Al graficar las curvas $y = x$ e $y = \cos x$, vemos en la fig. 1, que la interesección de las mismas ocurre dentro del intervalo $[0.6, 0.8]$. Así pues, la raíz buscada se encuentra en el interior de este intervalo y con él podemos proceder a calcular la raíz con el método de bisección asumiendo una tolerancia para el error absoluto de $\frac{1}{2} \times 10^{-8}$ y un número máximo de, digamos, 100 iteraciones. El siguiente programa implementa lo requerido.

**Código 1. Determinación de la raíz de la ec.** $\cos x - x = 0$

```
program ejemplo1
  use iso_fortran_env, only: wp => real64
  use roots, only: biseccion
  implicit none (type, external)
  real(wp) :: a, b, tol, raiz
  integer  :: n, clave

  ! Datos de iniciales
  a   = 0.6_wp
  b   = 0.8_wp
  tol = 0.5e-8_wp
  n   = 100

  ! Determinar la raíz
  call biseccion(f,a,b,n,tol,raiz,clave)
  if (clave == 0) then
     write(*,*) 'Raíz =', raiz
     write(*,*) 'Iteraciones realizadas =', n
  else
     write(*,*) 'Error =', clave
  endif

contains

  real(wp) function f(x)
    ! Función que define la ecuación
    real(wp), intent(in) :: x
    f = cos(x) - x
  end function f

end program ejemplo1
```

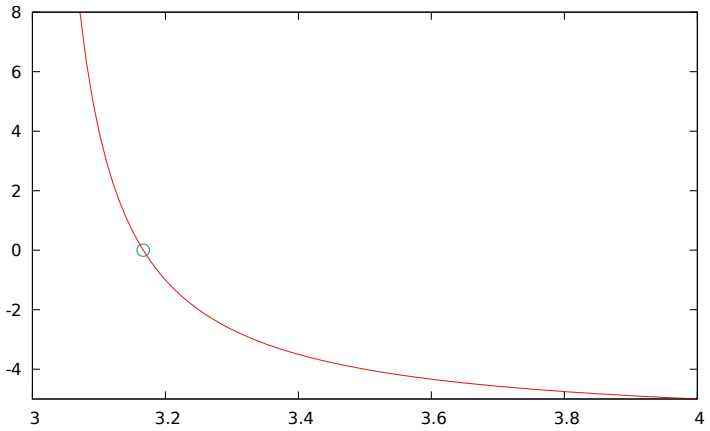


**Figura 2.** La función $1/(x-3)-6$ en el intervalo $[3,4]$ y su cero.

La compilación del mismo procede en la línea de comandos como sigue:

```
$ gfortran -Wall -o ejemplo1 roots.f90 ejemplo1.f90
```

Su ejecución arroja entonces:

```
$ ./ejemplo1
Raíz = 0.73908513486385341
Iteraciones realizadas = 26
```

Así, pues, la raíz buscada, con ocho decimales correctos, es 0.73908513.

Como segundo ejemplo, consideremos la ecuación

$$\frac{1}{x-3}+6=0.$$

Esta función es intencionalmente escogida para testear el desempeño de `fzero`, ya que la misma tiene un polo simple en $x=3$, próximo a la raíz buscada $x=3+1/6$, y la función es positiva sólo sobre el intervalo que comprende a dichos puntos (ver su gráfica en 2). Nótese que el valor 3 puede ser utilizado como uno de los extremos del intervalo inicial ya que, en la aritmética de punto flotante implementada en las computadoras, la función toma allí el valor `+Infinity`. El siguiente programa implementa la resolución del problema.

**Código 2. Determinación de la raíz de la ec. $1/(x-3)-6=0$**

```
program ejemplo2
  use iso_fortran_env, only: wp => real64
  use roots, only: fzero
  implicit none (type, external)

  real(wp) :: a,b
  real(wp) :: rerror, aerror
  integer  :: code

  a = 3.0_wp
  b = 4.0_wp

  rerror = 5.0e-8_wp
  aerror = 0.0_wp

  call fzero ( f, a, b, rerror, aerror, code )

  write(*,'(a,i0)') 'Code = ', code
  write(*,'(a,g0)') 'Root = ', a
  write(*,*)
```

```
  contains

    real(wp) function f(x)
      real(wp), intent(in) :: x
      f = 1.0_wp/( x - 3.0_wp ) - 6.0_wp
    end function f

  end program ejemplo2
```

La compilación del mismo procede en la línea de comandos como sigue:

```
 $ gfortran -Wall -o ejemplo2 roots.f90 ejemplo2.f90
```

Su ejecución arroja entonces:

```
 $ ./ejemplo2
Code = 0
Root = 3.1666666669771066
```

Vemos, pues, que el código arroja el valor correcto de la raíz, dentro de la tolerancia especificada.

# Código del módulo **roots**

**Código 3. Módulo roots**

```
module roots

  use, intrinsic :: iso_fortran_env, only: wp => real64
  implicit none

  logical, private, parameter :: debug = .true.

  abstract interface
     real(wp) function func(x)
       import :: wp
       implicit none
       real(wp), intent(in) :: x
     end function func
  end interface

contains

  subroutine biseccion ( f, a, b, n, tol, raiz, clave )

    ! METODO DE BISECCION para encontrar una solución
    ! de f(x)=0 dada la función continua f en el intervalo
    ! [a,b] donde f(a) y f(b) tienen signos opuestos.

    ! Argumentos
    procedure(func)          :: f     ! Función que define la ecuación
    real(wp), intent(in)     :: a     ! Extremo izquierdo del intervalo inicial
    real(wp), intent(in)     :: b     ! Extremo derecho del intervalo inicial
    integer,  intent(in out) :: n     ! Límite de iteraciones/iteraciones
        realizadas
    real(wp), intent(in)     :: tol   ! Tolerancia para el error absoluto
    real(wp), intent(out)    :: raiz  ! Aproximación a la raiz
    integer,  intent(out)    :: clave ! Clave de éxito:
                                      !   0 : éxito
                                      !  >0 : iteraciones excedidas
                                      !  <0 : no se puede proceder (f de
                                      !       igual signo en a y b)
    ! Variables locales
    integer  :: i
    real(wp) :: xl, xr, error

    clave = 1
    xl    = a
    xr    = b
    if ( sign( 1.0_wp , f(xl) ) * sign ( 1.0_wp, f(xr) ) > 0.0_wp ) then
       clave = -1
       return
    endif

    if ( debug ) write(*,'(a)') '    i          x_i'

    do i=1,n

       error = (xr-xl)*0.5_wp
       raiz  = xl + error

       if ( debug ) write(*,'(i5,2x,g0)') i, raiz

       if ( error <= tol ) then
          clave = 0
          n     = i
```

```
         exit
      endif

      if ( sign( 1.0_wp, f(xl) ) * sign( 1.0_wp, f(raiz) ) > 0.0_wp ) then
         xl = raiz
      else
         xr = raiz
      endif

   enddo

end subroutine biseccion


subroutine newton ( f, df, x0, n, tol, raiz, clave )

   ! Metodo DE NEWTON-RAPHSON para encontrar una
   ! solución de f(x)=0 dada la función derivable
   ! f y una aproximación inicial x0.

   ! Argumentos
   procedure(func)          :: f     ! Función que define la ecuación
   procedure(func)          :: df    ! Derivada de la función que define la
       ecuación
   real(wp), intent(in)     :: x0    ! Aproximación inicial a la raíz
   integer,  intent(in out) :: n     ! Límite de iteraciones/iteraciones
       realizadas
   real(wp), intent(in)     :: tol   ! Tolerancia para el error relativo
   real(wp), intent(out)    :: raiz  ! Aproximación a la raiz
   integer,  intent(out)    :: clave ! Clave de éxito:
                                     !   0 : éxito
                                     !  >0 : iteraciones excedidas
   ! Variables locales
   integer  ::  i
   real(wp) :: xx0

   clave = 1
   xx0   = x0

   if ( debug ) write(*,'(a)') '    i          x_i'

   do i=1,n
      raiz  = xx0 - f(xx0)/df(xx0)

      if ( debug ) write(*,'(i5,2x, g0)') i, raiz

      if ( abs( raiz-xx0 ) <= tol * abs( raiz ) ) then
         clave = 0
         n     = i
         exit
      endif

      xx0 = raiz

   end do

end subroutine newton


subroutine secante ( f, x0, x1, n, tol, raiz, clave )

   ! ALGORITMO DE LA SECANTE para encontrar una solución
   ! de f(x)=0, siendo f una función continua, dada las
   ! aproximaciones iniciales x0 y x1.
```

```
  ! Argumentos
  procedure(func)          :: f     ! Función que define la ecuación
  real(wp), intent(in)     :: x0,x1 ! Aproximaciones iniciales a la raíz
  integer,  intent(in out) :: n     ! Límite de iteraciones/iteraciones
      realizadas
  real(wp), intent(in)     :: tol   ! Tolerancia para el error relativo
  real(wp), intent(out)    :: raiz  ! Aproximación a la raiz
  integer,  intent(out)    :: clave ! Clave de éxito:
                                    !   0 : éxito
                                    !  >0 : iteraciones excedidas

  ! Variables locales
  integer :: i
  real(wp):: xx0, xx1, fx0, fx1

  clave = 1
  xx0   = x0
  xx1   = x1
  fx0   = f(x0)
  fx1   = f(x1)

  if ( debug ) write(*,'(a)') '    i          x_i'

  do i= 1,n

     raiz  = xx1 - fx1 * ( ( xx1-xx0 ) / ( fx1-fx0 ) )

     if ( debug ) write(*,'(i5,2x, g0)') i, raiz

     if ( abs( raiz-xx1 ) <= tol * abs( raiz ) ) then
        clave = 0
        n     = i
        exit
     endif

     xx0 = xx1
     fx0 = fx1
     xx1 = raiz
     fx1 = f(raiz)

  end do

end subroutine secante


subroutine birge_vieta ( a, x0, n, tol, raiz, clave )

  ! METODO DE BIRGE-VIETA para resolver ECUACIONES ALGEBRAICAS:
  ! P(x) = 0 donde P es un polinomio de grado m de coeficientes reales.
  ! El método se basa en el método de Newton-Raphson implementando el
  ! esquema de Horner para la evaluación del polinomio y su derivada.

  ! Argumentos
  real(wp), intent(in)     :: a(0:)  ! Vector de m+1 elementos conteniendo
                                     ! los coeficientes del polinomio
  real(wp), intent(in)     :: x0     ! Aproximación inicial a la raíz
  real(wp), intent(in)     :: tol    ! Tolerancia para el error relativo
  integer,  intent(in out) :: n      ! Límite de iteraciones/iteraciones
      realizadas
  real(wp), intent(out)    :: raiz   ! Aproximación a la raiz
  integer,  intent(out)    :: clave  ! Clave de éxito:
                                     !   0 : éxito
                                     !  >0 : iteraciones excedidas
  ! Variables locales
  integer :: m, i, j
```

```
  real(wp):: xx0,b,c

  clave = 1
  xx0   = x0
  m     = size(a)-1

  if ( debug ) write(*,'(a)') '    i          x_i'

  do i=1,n

     ! Esquema de Horner
     b = a(m)
     c = a(m)
     do j=m-1,1,-1
        b = b*xx0+a(j)
        c = c*xx0+b
     enddo
     b = b*xx0+a(0)

     ! Método de Newton
     raiz  = xx0 - b/c

     if ( debug ) write(*,'(i5,2x,g0)') i, raiz

     if ( abs( raiz-xx0 ) <= tol * abs( raiz ) ) then
        clave = 0
        n     = i
        exit
     end if
     xx0 = raiz
  end do

end subroutine birge_vieta


subroutine punto_fijo ( f, x0, n, tol, raiz, clave )

  ! ALGORITMO DE PUNTO FIJO o DE APROXIMACIONES SUCESIVAS
  ! para encontrar una solución de x=f(x) dada una
  ! aproximación inicial x0.

  ! Argumentos
  procedure(func)          :: f     ! Función que define la ecuación
  real(wp), intent(in)     :: x0    ! Aproximación inicial a la raíz
  integer,  intent(in out) :: n     ! Límite de iteraciones/iteraciones
      realizadas
  real(wp), intent(in)     :: tol   ! Tolerancia para el error relativo
  real(wp), intent(out)    :: raiz  ! Aproximación a la raiz
  integer,  intent(out)    :: clave ! Clave de éxito:
                                    !   0 : éxito
                                    !  >0 : iteraciones excedidas

  ! Variables locales
  integer  :: i
  real(wp) :: xx0

  clave = 1
  xx0   = x0

  if ( debug ) write(*,'(a)') '    i          x_i'

  do i=1,n
     raiz  = f(xx0)

     if ( debug ) write(*,'(i5,2x,g0)') i, raiz
```

```
      if ( abs( raiz-xx0 ) <= tol * abs( raiz ) ) then
         clave = 0
         n     = i
         exit
      endif
      xx0 = raiz
    end do

end subroutine punto_fijo


subroutine fzero( f, b, c, re, ae, iflag )

  ! Search for a zero of a function F(X) in a given interval
  ! (B,C).  It is designed primarily for problems where F(B)
  ! and F(C) have opposite signs.
  !
  ! Original author:
  !
  !  Shampine, L. F., (SNLA)
  !  Watts, H. A., (SNLA)
  !
  ! Modified by
  !
  !  Pablo Santamaría
  !
  ! Description
  !
  !     FZERO searches for a root of the nonlinear equation F(X) = 0
  !     (when F(X) is a real function of a single real variable X),
  !     between the given values B and C until the width
  !     of the interval (B,C) has collapsed to within a tolerance
  !     specified by the stopping criterion,
  !
  !                     ABS(B-C) <= 2*MAX(RW*ABS(B),AE).
  !
  !     The method used is an efficient combination of bisection and the
  !     secant rule and is due to T. J. Dekker.
  !
  ! Arguments
  !
  !   f     :EXT   - Name of the function subprogram defining F(X).
  !                  This subprogram should have the form
  !                   REAL(WP) FUNCTION F(X)
  !                      USE iso_fortan_env, ONLY: WP => REAL64
  !                      IMPLICIT NONE
  !                      REAL(WP), INTENT(IN) :: X
  !                      F = ...
  !                   END FUNCTION F
  !                  An explicit interface should be provided
  !                  in the calling program.
  !
  !   b     :INOUT - One end of the interval (B,C).  The
  !                  value returned for B usually is the better
  !                  approximation to a zero of F.
  !
  !   c     :INOUT - The other end of the interval (B,C)
  !
  !
  !   re    :IN    - Relative error used for RW in the stopping criterion.
  !                  If the requested RE is less than machine precision,
  !                  then RW is set to approximately machine precision.
  !
  !   ae    :IN    - Absolute error used in the stopping criterion.  If
```

```
!                  the given interval (B,C) contains the origin, then a
!                  nonzero value should be chosen for AE.
!
!   iflag :OUT   - A status code.  User must check IFLAG after each
!                  call.  Control returns to the user from FZERO in all
!                  cases.
!
!                0  B is within the requested tolerance of a zero.
!                   The interval (B,C) collapsed to the requested
!                   tolerance, the function changes sign in (B,C), and
!                   F(X) decreased in magnitude as (B,C) collapsed.
!
!                1  F(B) = 0.  However, the interval (B,C) may not have
!                   collapsed to the requested tolerance.
!
!               -1  B may be near a singular point of F(X).
!                   The interval (B,C) collapsed to the requested tol-
!                   erance and the function changes sign in (B,C), but
!                   F(X) increased in magnitude as (B,C) collapsed, i.e.
!                     ABS(F(B out)) > MAX(ABS(F(B in)),ABS(F(C in)))
!
!               -2  No change in sign of F(X) was found although the
!                   interval (B,C) collapsed to the requested tolerance.
!                   The user must examine this case and decide whether
!                   B is near a local minimum of F(X), or B is near a
!                   zero of even multiplicity, or neither of these.
!
!               -3  Too many (> 500) function evaluations used.
!
! References
!
!      L. F. Shampine and H. A. Watts, FZERO, a root-solving
!             code, Report SC-TM-70-631, Sandia Laboratories,
!             September 1970.
!      T. J. Dekker, Finding a zero by means of successive
!             linear interpolation, Constructive Aspects of the
!             Fundamental Theorem of Algebra, edited by B. Dejon
!             and P. Henrici, Wiley-Interscience, 1969.
!
! Notes
!
! This implementation is a Fortran 90 refactoring from the original
! dfzero.f subroutine:
!
! * http://www.netlib.org/slatec/src/dfzero.f
!
! with some changes from:
!
! * ftp://ftp.wiley.com/public/college/math/sapcodes/f90code/saplib.f90
!

! Arguments
procedure(func)          :: f
real(wp), intent(in out) :: b
real(wp), intent(in out) :: c
real(wp), intent(in)     :: re
real(wp), intent(in)     :: ae
integer,  intent(out)    :: iflag

! Local variables
real(wp) :: a, acbs, acmb, cmb
real(wp) :: fa, fb, fc, fx
real(wp) :: p, q
real(wp) :: aw, rw, tol
integer  :: ic, kount, max_feval
```

```
      logical  :: force_bisect
      character(7) :: method


      ! Set error tolerances
      rw = max( re, epsilon(1.0_wp) )
      aw = max( ae, 0.0_wp )

      ! Initialize
      max_feval = 500
      ic = 0

      fb = f( b )
      fc = f( c )
      kount = 2

      a = c
      fa = fc
      acbs = abs( b-c )
      fx = max( abs(fb), abs(fc) )

      if ( debug ) then
         write(*,'(a,1x,a,11x,a,19x,a,17x,a,15x,a)') 'F-count', &
              & 'method', 'b', 'c', 'f(b)', 'f(c)'
         method = 'initial'
      end if

      ! Iteration loop
      do

         ! Perform interchange such abs(f(b)) < abs(f(c))
         if ( abs( fc ) < abs( fb ) ) then
            a  = b
            fa = fb
            b  = c
            fb = fc
            c  = a
            fc = fa
         end if

         if ( debug ) write(*,'(i5,*(2x,g0))') kount, method, b, c, fb, fc

         cmb  = 0.5_wp * (c-b)
         acmb = abs( cmb )
         tol  = max( rw*abs(b), aw ) ! or tol = rw*abs(b) + aw

         ! Test stopping criterion and function count
         ! Set iflag appropriately
         if ( acmb <= tol ) then
            if ( sign( 1.0_wp, fb ) * sign( 1.0_wp, fc ) > 0.0_wp ) then
               iflag = -2
            else if ( abs( fb ) > fx ) then
               iflag = -1
            else
               iflag = 0
            end if
            exit
         endif

         if ( abs( fb ) <= 0.0_wp ) then
            iflag = 1
            exit
         end if

         if ( kount >= max_feval ) then
```

```
            iflag = -3
            exit
         end if

         ! Calculate new iterate implicitly as b+p/q, where we arrange
         ! p >= 0.  The implicit form is used to prevent overflow.
         p = (b-a) * fb
         q = fa - fb

         if ( p < 0.0_wp ) then
            p = -p
            q = -q
         end if

         ! Update a and check for satisfactory reduction in the size of the
         ! bracketing interval every four increments.
         ! If not, force bisection.
         a  = b
         fa = fb
         ic = ic + 1
         force_bisect = .false.

         if ( ic >= 4 ) then

            if ( 8.0_wp * acmb >= acbs ) then
               force_bisect = .true.
            else
               ic   = 0
               acbs = acmb
            end if

         end if

         ! Get new iterate b
         if ( force_bisect ) then
            b = b + cmb         ! Use bisection (c+b)/2
            if (debug) method = 'bisect*'
         else if ( p <=  abs(q) * tol ) then ! If to small, increment by tolerance
            b = b + sign( tol, cmb )
            if ( debug ) method = 'minimal'
         else if ( p < cmb * q ) then ! Root ought to be between b and (c+b)/2
            b = b + p/q  ! Use secant rule
            if ( debug ) method = 'secant'
         else
            b = b + cmb  ! Use bisection (c+b)/2
            if ( debug ) method = 'bisect'
         end if

         ! Have completed computation for new iterate b.
         fb = f( b )
         kount = kount + 1

         ! Decide whether next step is interpolation or extrapolation.
         if ( sign( 1.0_wp, fb ) * sign( 1.0_wp, fc ) > 0.0_wp ) then
            c  = a
            fc = fa
         end if

      end do

   end subroutine fzero

end module roots
```