Subrutinas en Fortran 95 para la resolución de ecuaciones no lineales de una variable

Pablo Santamaría

v0.3.1 (Mayo 2014)

1. 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 aproximaciones iniciales para la raíz, generan una sucesión de aproximaciones $x_0, x_1, x_2, \ldots$ que esperamos convergan al valor de la raíz α 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

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

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 funcione [1]. 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 quiera. 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 de Fortran 95 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 mismas en un módulo, al que denominaremos `roots`. En nuestra implementación todas las cantidades reales serán de la clase de *doble precisión*, la cual, para máxima flexibilidad, está definida en forma paramétrica utilizando el modulo intrínseco `iso_fortran_env`.

[1] De hecho, efectuando algunas iteraciones del método de bisección podemos obtener una buena aproximación para iniciar el método de Newton.

Código 1. Implementación del módulo `roots`

```
MODULE roots

    USE, intrinsic :: iso_fortran_env, ONLY: WP => REAL64
    IMPLICIT NONE

CONTAINS

    SUBROUTINE biseccion(f,a,b,n,tol,raiz,clave)
        ...
    END SUBROUTINE biseccion

    SUBROUTINE newton(f,df,x0,n,tol,raiz,clave)
        ...
    END SUBROUTINE newton

    SUBROUTINE birge_vieta(a,m,x0,n,tol,raiz,clave)
        ...
    END SUBROUTINE birge_vieta

    SUBROUTINE secante(f,x0,x1,n,tol,raiz,clave)
        ...
    END SUBROUTINE secante

    SUBROUTINE punto_fijo(f,x0,n,tol,raiz,clave)
        ...
    END SUBROUTINE punto_fijo

END MODULE roots
```

El código correspondiente a cada subrutina, que debe insertarse donde se encuentran los puntos suspensivos, será discutido por separado en las siguientes secciones.

2. Método de bisección

El método de bisección comienza con un intervalo $[a, b]$ que contiene a la raíz. Entonces se computa el punto medio $x_0 = (a + b)/2$ del mismo y se determina en cual de los dos subintervalos $[a, x_0]$ o $[x_0, b]$ 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 siguiente implementación del método, como criterio de paro para la sucesión de aproximaciones.

Código 2. Implementación del método de bisección

```
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.
    ! ------------------------------------------------
    ! Bloque de declaración de argumentos
    ! ------------------------------------------------
    INTERFACE
       REAL(WP) FUNCTION f(x)         ! Función que define la ecuación
         IMPORT :: WP
         IMPLICIT NONE
         REAL(WP), INTENT(IN) :: x
       END FUNCTION f
    END INTERFACE
    REAL(WP), INTENT(IN)    :: a     ! Extremo izquierdo del intervalo inicial
    REAL(WP), INTENT(IN)    :: b     ! Extremo derecho del intervalo inicial
```

```
      INTEGER,  INTENT(INOUT) :: 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)
      ! ------------------------------------------------
      ! Bloque de declaración de variables locales
      ! ------------------------------------------------
      INTEGER  :: i
      REAL(WP) :: xl, xr, error
      ! ------------------------------------------------
      ! Bloque de procesamiento
      ! ------------------------------------------------
      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
      DO i=1,n
         error = (xr-xl)*0.5_WP
         raiz  = xl + error
         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
```

3. 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, ..., 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

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

Nuestra implementación de la subrutina correspondiente requiere que se pase 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`.

Código 3. Implementación del método de Newton–Raphson

```
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.
    ! ------------------------------------------------
    ! Bloque de declaración de argumentos
    ! ------------------------------------------------
    INTERFACE
       REAL(WP) FUNCTION f(x)         ! Función que define la ecuación
         IMPORT :: WP
         IMPLICIT NONE
         REAL(WP), INTENT(IN) :: x
       END FUNCTION f
       REAL(WP) FUNCTION df(x)        ! Derivada de la función
         IMPORT :: WP                 ! que define a la ecuación
         IMPLICIT NONE
         REAL(WP), INTENT(IN) :: x
       END FUNCTION df
    END INTERFACE
    REAL(WP), INTENT(IN)    :: x0    ! Aproximación inicial a la raíz
    INTEGER,  INTENT(INOUT) :: n     ! Límite de iteraciones/iteraciones realizadas
    REAL(WP), INTENT(IN)    :: tol   ! Tolerancia para el error relativo
    REAL(WP), INTENT(OUT)   :: raiz  ! Aproximación a la raíz
    INTEGER,  INTENT(OUT)   :: clave ! Clave de éxito:
                                     !   0 : éxito
                                     !  >0 : iteraciones excedidas
    ! ------------------------------------------------
    ! Declaración de variables locales
    ! ------------------------------------------------
    INTEGER  ::  i
    REAL(WP) :: xx0
    ! ------------------------------------------------
    ! Bloque de procesamiento
    ! ------------------------------------------------
    clave = 1
    xx0   = x0
    DO i=1,n
       raiz  = xx0 - f(xx0)/df(xx0)
       IF (ABS(raiz-xx0) < tol*ABS(raiz) ) THEN
          clave = 0
          n     = i
          EXIT
       ENDIF
       xx0 = raiz
    END DO
    ! ------------------------------------------------
END SUBROUTINE newton
```

4. 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.

Nuestra implementación en la siguiente subrutina pasa los coeficientes del polinomio en un arreglo a de $(m+1)$ componentes, siendo m el grado del polinomio (valor que también es requerido como argumento). En la subrutina, para simplificar el tratamiento de los subíndices, el arreglo es declarado con el índice inferior 0, no 1, de manera que $a(0) = a_0, a(1) = a_1, \ldots, a(M) = a_m$.

Código 4. Implementación del método de Birge–Vieta

```
SUBROUTINE birge_vieta(a,m,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 evalua-
    ! ción del polinomio y su derivada.
    ! -------------------------------------------------
    ! Bloque de declaración de argumentos
    ! -------------------------------------------------
    INTEGER,  INTENT(IN)    :: m     ! Grado del polinomio
    REAL(WP), INTENT(IN)    :: a(0:m)! 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(INOUT) :: n     ! Limite 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
    ! -------------------------------------------------
    ! Bloque de declaración de variables locales
    ! -------------------------------------------------
    INTEGER :: i, j
    REAL(WP):: xx0,b,c
    ! -------------------------------------------------
    ! Bloque de procedimiento
    ! -------------------------------------------------
    clave = 1
    xx0   = x0
    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 (ABS(raiz-xx0) < tol*ABS(raiz)) THEN
           clave = 0
           n     = i
           EXIT
        END IF
        xx0 = raiz
     END DO
     ! -------------------------------------------------
END SUBROUTINE birge_vieta
```

5. Método de la secante

El método de la secante procede a partir de *dos* aproximaciones iniciales 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

$$x_{n+1} = x_n - f(x_n)\frac{(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}, \quad n = 2, 3, \ldots$$

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 arrancar el método.

Código 5. Implementación del método de la secante

```
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.
    ! -------------------------------------------------
    ! Bloque de declaración de argumentos
    ! -------------------------------------------------
    INTERFACE
       REAL(WP) FUNCTION f(x)       ! Función que define la ecuación
         IMPORT :: WP
         IMPLICIT NONE
         REAL(WP), INTENT(IN) :: x
       END FUNCTION f
    END INTERFACE
    REAL(WP), INTENT(IN)    :: x0,x1 ! Aproximaciones iniciales a la raíz
    INTEGER,  INTENT(INOUT):: 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
    ! -------------------------------------------------
    ! Bloque de declaración de variables locales
    ! -------------------------------------------------
    INTEGER :: i
    REAL(WP):: xx0, xx1, fx0, fx1
    ! -------------------------------------------------
    ! Bloque de procesamiento
    ! -------------------------------------------------
    clave = 1
    xx0   = x0
    xx1   = x1
    fx0   = f(x0)
```

```
      fx1   = f(x1)
      DO i= 2,n
         raiz  = xx1 - fx1*((xx1-xx0)/(fx1-fx0))
         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
```

6. Iteración de punto fijo

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

$$x = \phi(x)$$

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

$$x_{n+1} = \phi(x_n), \quad n = 1, 2, \ldots$$

En la siguiente implementación, es importante recordar que la función que es pasada por argumento es ahora $\phi(x)$ y no $f(x)$, función que debe ser implementada por el usuario como una `FUNCTION`.

Código 6. Implementación del método de punto fijo

```
   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.
      ! -------------------------------------------------
      ! Bloque de declaración de argumentos
      ! -------------------------------------------------
      INTERFACE
         REAL(WP) FUNCTION f(x)       ! Función que define la ecuación
           IMPORT :: WP
           IMPLICIT NONE
           REAL(WP), INTENT(IN) :: x
         END FUNCTION f
      END INTERFACE
      REAL(WP), INTENT(IN)    :: x0    ! Aproximación inicial a la raíz
      INTEGER,  INTENT(INOUT):: n      ! Limite 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
      ! -------------------------------------------------
      ! Bloque de declaración de variables locales
      ! -------------------------------------------------
      INTEGER  :: i
      REAL(WP) :: xx0
      ! -------------------------------------------------
      ! Bloque de procesamiento
      ! -------------------------------------------------
      clave = 1
      xx0   = x0
```

```
      DO i=1,n
         raiz  = f(xx0)
         IF (ABS(raiz-xx0) < tol*ABS(raiz)) THEN
            clave = 0
            n     = i
            EXIT
         ENDIF
         xx0 = raiz
      END DO
      ! ------------------------------------------------
  END SUBROUTINE punto_fijo
```

7. Ejemplo

Como 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 $\frac{1}{2} \times 10^{-6}$ y un número máximo de, digamos, 100 iteraciones. El siguiente programa implementa lo requerido.

Código 7. Determinación de la raíz de la ec. $\cos x - x = 0$

```
PROGRAM ejemplo
  USE, intrinsic :: iso_fortran_env, ONLY: WP => REAL64
  USE roots, ONLY: biseccion

  IMPLICIT NONE
  REAL(WP) :: a, b, tol, raiz
  INTEGER :: n, clave

  ! Datos de iniciales
  a   = 0.6_WP
  b   = 0.8_WP
  tol = 0.5E-6_WP
  n   = 100

  ! Determinar la raíz
  CALL biseccion(f,a,b,n,tol,raiz,clave)
  IF (clave == 0) THEN
     WRITE(*,*) 'Raíz = ', raiz
     WRITE(*,*) 'Número de 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 ejemplo
```

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

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

Su ejecución arroja entonces:

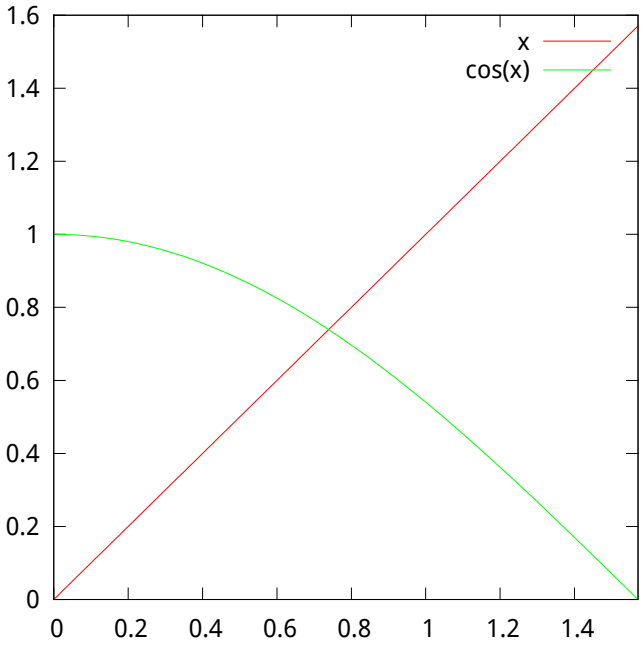

Figura 1. Determinación de una aproximación inicial de la raíz de la función $\cos x - x$.

```
$ ./ejemplo
Raíz =   0.73908500671386723
Número de iteraciones realizadas =          19
```

Así, pues, la raíz buscada, con seis decimales correctos, es 0.739085.