Ejercicios Bucles, Condiciones y Funciones II

EJERCICIOS

Ejercicio 1: Método de Bisección para Raíces

Define una función que encuentre la raíz de una función continua f(x) en un intervalo [a, b] usando el método de bisección. La función debe incluir verificaciones y funciones internas para evaluar f(x).

1. La función debe aceptar como parámetros:

   • f: la función a la que se le buscar a la raíz.

   • a: el límite inferior del intervalo.

   • b: el límite superior del intervalo.

   • ϵ (tol): el criterio de tolerancia, es decir, la diferencia entre dos iteraciones consecutivas de la raíz encontrada.

   • Max_iter: el número máximo de iteraciones permitidas.

2. La función debe verificar que f(a) y f(b) tengan signos opuestos. Si no los tienen, debe devolver un mensaje de error indicando que no se garantiza que exista una raíz en el intervalo.

3. Implementa la búsqueda de la raíz mediante el siguiente algoritmo:

   • Calcula el punto medio c = (a+b)/ 2 .
   • Si f(c) es suficientemente cercano a cero (según el valor de ϵ), entonces m es la raíz aproximada.
   • Si el signo de f(a) y f(c) son opuestos, la raíz está en el intervalo [a, c]. Actualiza el valor de b = c.
   • Si el signo de f(b) y f(c) son opuestos, la raíz está en el intervalo [c, b]. Actualiza el valor de a = c.
   • Repite el proceso hasta que la diferencia entre a y b sea menor que ϵ o se alcance el número máximo de iteraciones.

Al finalizar las iteraciones, la función debe devolver el valor de la raíz aproximada. Usa una función interna para evaluar f(x) y otra para verificar las condiciones del intervalo. Verifica el funcionamiento de tu código para f(x) = x^3 − 4 en el intervalo [1, 2] y una tolerancia ϵ = 0.001 y un número máximo de iteraciones de 100.

Resolución Ejercicio 1

```
biseccion <- function (f, a, b, tol, max_iter ) {

#creamos una funcion para implementar el método


if (f (a) * f (b) >= 0) {

  cat ( " No se puede aplicar el método en el intervalo dado \n " )

  return ( NA )

}

#este comando if se usa para comprobar que f(a) y f(b) no tienen el mismo signo


c <- (a + b) / 2   #obtenemos así el punto medio

iter <- 0   #Contador de iteraciones


while ((b - a) / 2 > tol && iter < max_iter ) {

  if (f(c) == 0) {

  break


  }

  if (f (a) * f (c) < 0) {

   b <- c

  } else {

   a <- c

  }


  c <- (a + b) / 2

  iter <- iter + 1

  }

#por medio de este while, nos aseguramos de varias cosas:
```

#Primero, de que el intervalo [a,b] es lo suficientemente grande y que supera la tolerancia para así poder calcular la raíz de forma precisa. Si el intervalo es demasiado pequeño porque es menor que la tolerancia, no se puede calcular. Además, otra condición de este while es la que nos obliga a que la iteración no sobrepase el número máximo de repeticiones que nosotros decidimos

#Segundo, dentro de este while aparece 2 comandos if. Vamos a explicarlos: El primero declara que en el caso de que c, el punto medio, es 0, el bucle se rompe, ya que significa que el valor de c es una raíz de la función y devuelve el valor de c como la raíz. El segundo if, que incluye un else también, nos dice que si el producto de f(a) y f(c) es negativo, quiere decir que la raíz está entre a y c, por lo que se actualiza el límite del intervalo y b se cambia por c (b=c). Luego, el else actúa si no se cumple el if, y por lo tanto, si el producto es positivo o cero, significa que la raíz está entre cy b, por lo que se actualiza el límite del intervalo y a se cambia por c: (a=c).

#Finalmente, escribe la fórmula del punto medio de nuevo porque en cada vuelta del bucle while el valor de c cambia, y así podemos seguir dividiendo el intervalo. También, aumenta el número de las iteraciones por 1 en cada vuelta que da el bucle while

```
 if (iter == max_iter) {
 cat ( "Se alcanzó el número máximo de iteraciones \n" )
 }
```

#la función acaba cuando se ha alcanzado el número máximo de iteraciones (repeticiones), y en este caso no se ha encontrado la solución

```
 return (c)  #devuelve el valor de la raíz aproximada
}
```

Ejemplo de uso

```
f <- function (x) x^3 – 4
```

#creamos la función del enunciado para aplicar en ella nuestro método

#Ahora declaramos los valores del enunciado “Verifica el funcionamiento de tu código para f(x) = x^3 − 4 en el intervalo [1, 2] y una tolerancia ϵ = 0.001 y un número máximo de iteraciones de 100.”

a <- 1

b <- 2

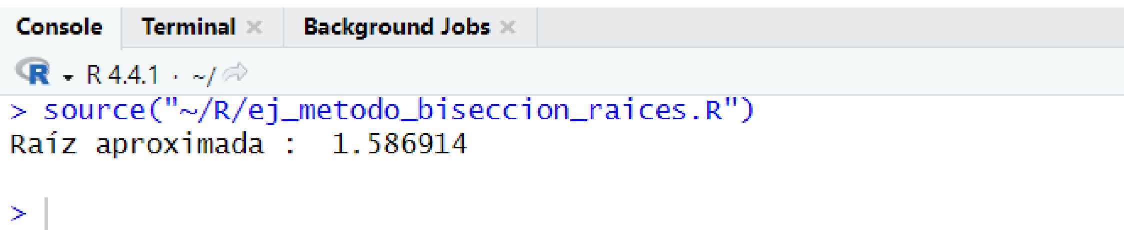
tol <- 0.001

max_iter <- 100

resultado_raiz <- biseccion (f, a, b, tol, max_iter)

cat("Raíz aproximada : " , resultado_raiz , "\n ")

#creamos el parámetro resultado_raiz para aplicar nuestro ejemplo del método de bisección y le damos el valor de la función creada. Por último, pedimos que nos la enseñe en la consola



Ejercicio 2: Serie de Fibonacci por Condicional

Define una función que calcule los primeros n términos de la serie de Fibonacci modificada, donde:

$$F(i) = \begin{cases} F(i-1) + F(i-2) & \text{si } i \text{ es par} \\ F(i-1) - F(i-2) & \text{si } i \text{ es impar} \end{cases}$$

La función debe usar una función interna para calcular los términos. Recuerda que la serie de Fibonacci toma estos valores iniciales F(1) = 0 y F(2) = 1. Verifica la función para n = 10.

Resolución Ejercicio 2

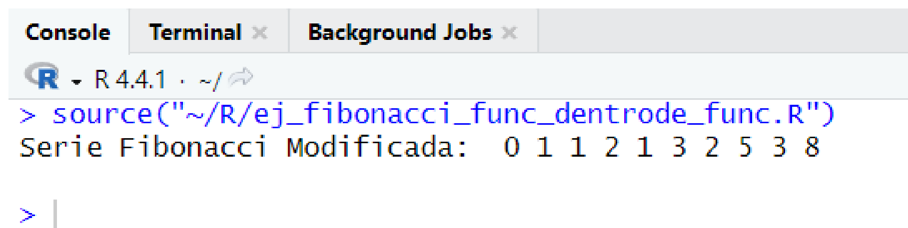
```
fibonacci_modificado <- function (n) {

#creamos una función que depende de la variable n, esta función creará la serie

   fib <- function ( i ) {

#creamos otra función dentro de la anterior que calcula los valores i-esimos.
Esta función solo puede ser usada dentro de la principal y no podrá ser llamada

   if ( i == 1) return (0)

   if ( i == 2) return (1)  #este if y el anterior nos lo proporciona el enunciado

   if ( i %% 2 == 0) {              #este if else corresponde a F(i) del enunciado. %%
                                    devuelve el resto de la división

   return (fib(i - 1) + fib( i - 2))   #si es par, hará esta operación

   } else {

  return (fib(i - 1) - fib(i - 2))     #en cambio, si es impar hará esta operación

  }

}         #cierra la función interna ya que se han hecho todos los cálculos
          necesarios para obtener los valores, ahora solo queda colocarlos
```

```
resultado <- numeric (n)      #crea un vector de longitud n y 0 de valor inicial, es
                              decir, un vector vacío.

for (i in 1:n) {

resultado [i] <- fib (i)

  }
```

#ahora rellenará este vector vacío usando un bucle for y la función secundaria

```
return (resultado)

}   #aquí termina la función principal


n <- 10

resultado_fib <- fibonacci_modificado (n)

cat( "Serie Fibonacci Modificada: " ,resultado_fib , "\n " )
```

#siguiendo el enunciado, probamos que nuestro código funciona correctamente, implementando una longitud de 10 términos a la Serie. No es necesario llamar a la función interna porque los valores se calculan dentro de la función primaria. Le damos nombre a nuestra función particular y pedimos que la imprima por la consola. Obtenemos este resultado:



Ejercicio 3: Integración Numérica por Sumas de Riemann

Desarrolla una función en R que realice la integración numérica aproximada de una función f(x) definida por el usuario mediante el método de Sumas de Riemann usando puntos medios.

Instrucciones:

1. La función debe aceptar como argumentos:
    - f: la función a integrar.
    - a: el límite inferior de integración.
    - b: el límite superior de integración.
    - n: el número de subintervalos en los que se dividirá [a, b].
2. La función calculará la integral definida como:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f(x_i)\Delta x$$

   donde:
   • $\Delta x = (b-a) / n$ es el ancho de cada subintervalo.
   • $x_i = a + (i - 0.5)\Delta x$ es el punto medio de cada subintervalo.

3. Implementa la solución usando un bucle for y evita el uso de funciones predefinidas avanzadas como sum y prod.

4. Incluye una función interna dentro de la función principal que evalúe los puntos medios xi de cada subintervalo.

Fórmulas a emplear:

- **Ancho del subintervalo:**

$$\Delta x = \frac{b-a}{n}$$

- **Punto medio del subintervalo i:**

$$x_i = a + (i - 0.5)\Delta x$$

- **Suma acumulada:**

$$\text{suma} = \sum_{i=1}^{n} f(x_i)\Delta x$$

Verifica el código para f(x)) = x^2, a = 0, b = 1 y n = 1000

Resolución Ejercicio 3

```
suma_rieman = function (f, a, b, n) {    #se crea la función para calcular la suma
  #ahora se aplican las fórmulas proporcionadas en el enunciado
  dx = (b-a)/n          #se usa para dividir el intervalo [a,b] en subintervalos
                        iguales. Se hace para calcular la aproximación de la
                        integral, sumando las áreas de los rectángulos formados
                        por los subintervalos bajo la curva de la función.
  suma = 0              #iniciamos un sumatorio para la suma acumulada y dentro
                        calculamos xi ya que es el argumento del sumatorio,
                        siempre siguiendo las fórmulas dadas arriba
  for (i in 1:n){
  x_sub_i = a + (i-0.5)*dx
  suma = suma + f(x_sub_i) * dx    #evalúa la función en el punto xi
  }
  return (suma)
}
#ahora, comprobamos que funciona nuestro código
f <- function (x)  x ^2   #esta es la función a integrar
a <- 0
b <- 1
n <- 1000
result = suma_rieman (f,a,b,n)    #damos el nombre de “result” a la aplicación
                                  de los parámetros en la función que hemos
                                  creado, y luego pedimos que nos imprima el
                                  valor resultante

print (result)
```

```
Console   Terminal ×   Background Jobs ×
R 4.4.1 · ~/
> source("~/R/ej_suma_rieman_funciones_y_llamar_funciones.R")
[1] 0.3333332
>
```