Pontificia Universidad
JAVERIANA
Bogotá

Pontificia Universidad Javeriana
Departamento de Matemática
Análisis Numérico

Taller 1

Parte 2: Metodos

por

Monica Alejandra Alvarez Carrillo
monica_alvarez@javeriana.edu.co

Santiago Palacios Loaiza
palacios-santiago@javeriana.edu.co

Profesora: Eddy Herrera Daza

Bogotá D.C., 10 de Enero del 2020

1. Tema 1: Teorema de Horner

1.1. Teoría

También conocido como método de multiplicación anidada, es un teorema usado en el análisis numérico creado por el matemático ingles William George Horner en el siglo XVII que consiste en calcular el valor de un polinomio de grado n utilizando el menor numero de multiplicaciones. Esta característica de realizar el menor numero de operaciones posibles es quizás la razón por la que es ampliamente usado, debido a que aumenta en gran medida su eficiencia.

El teorema enuncia que un polinomio de la forma $P(x) = a_0x^n + a_1x^{n-1} + ... + a_n$ puede evaluarse en un número n de multiplicaciones y n adiciones.

1.2. Solución 1

Para demostrar el resultado de la evaluación del polinomio de grado 4

$$P(x) = 2x^4 - 3x^2 + 3x - 4$$

Teniendo en cuenta que el número de multiplicaciones del segundo método está dado por:

$$2n + 1$$

En el caso del polinomio dado, $n = 7$ es decir, $P(x) = 2 * x * (x^3) + 0 * x * (x^2) - 3 * x * (x) + 3 * x - 4$
Se puede demostrar que el teorema se cumple para todo numero n, así:

- Base de inducción
  Para $n = 1$,

  $$2n - 1 = a_0x^n + a_1x^{n-1} + ... + a_n$$

- Paso inductivo
  Para $n = k$,

  $$2k - 1 = a_0x^k + a_1x^{k-1} + ... + a_k$$

  Entonces para $n = k + 1$,

  $$2(k + 1) - 1 = a_0x^{k+1} + a_1x^{(k+1)-1} + ... + a_{k+1}$$
  $$2(k + 1) - 1 = a_0x^{k+1} + a_1x^{k+1-1} + ... + a_{k+1}$$
  $$2(k + 1) - 1 = a_0x^{k+1} + a_1x^k + ... + a_{k+1}$$

  De esta manera podemos concluir que el resultado de evaluar el polinomio con este método es correcto, ya que se evidencian las 7 multiplicaciones que la teoría enuncia.

1.3. Solución 2

El siguiente es el codigo del teorema de Horner implementado en R.

```
Horner <- function(coeficientes, x0){

  valor = coeficientes[1]   # Ajuste de las variables
  multi =0
  sumas=0

  for(i in coeficientes[2:length(coeficientes)]){ # Ciclo que se repite por la cantidad de
    valor <- x0*valor + i                         # terminos del polinomio propuesto.
    multi = multi + 1
    sumas = sumas + 1
  }
  return(cat("El valor del polinomio es: ", round(valor,4), "\nEl total de operaciones es de: ", multi+sumas,"; Multiplicaciones:  ",multi,"; Sumas:  ",sumas))
}
```

El código anterior fue probado con los siguientes valores: coeficientes=[2, 0, -3, 3, -4] y $x_0 = 2$ y dio como resultado un valor para el polinomio de 22, donde se realizaron 4 sumas y 4 multiplicaciones, para un total de 8 operaciones mínimas. Estos resultados nos permiten afirmar que el teorema de Horner fue implementado y probado correctamente en R.

1.4. Solución 3

Para la solución del punto 3 se agregaron un pequeño conjunto de elementos al código de la implementación anterior para el calculo del error con respecto a la formula alternativa propuesta. El nuevo código solución se muestra a continuación:

```
Horner_Error <- function(coeficientes, x0){
  valor = coeficientes[1] # Ajuste de las variables
  multi =0
  sumas=0
  for(i in coeficientes[2:length(coeficientes)]){ # Ciclo que se repite por la cantidad de
    valor <- x0*valor + i                         # terminos del polinomio propuesto.
    multi = multi + 1
    sumas = sumas + 1

  }

  teoria= ((x0^51)-1)/(x0-1) # Formula alternativa propuesta
```

```
13    print(teoria)
14
15
16
17    errorabs = (teoria-valor)   # Calculo de los errores
18    errorrela = (errorabs/valor)*100
19    return(cat("El valor del polinomio es: ", round(valor,4), "\nEl total de
          operaciones es de: ", multi+sumas,"; Multiplicaciones:  ",multi,";
          Sumas:  ",sumas, " \n El valor teorico es ",round(teoria,4), ", el
          error absoluto es de ",round(errorabs,4),"y el error relativo de ",
          round(errorrela,4),"%"))
20
21  }
```

El código anterior fue probado con los siguientes valores propuestos en el enunciado: El polinomio tiene un 1 en todos sus coeficientes, por lo tanto el vector que se pasa como parámetro contiene 50 veces el numero 1 y $x_0 = 1{,}0001$ y dio como resultados un valor para el polinomio de 50.1227, donde se realizaron 49 sumas y 49 multiplicaciones, para un total de 98 operaciones mínimas. El error absoluto dio un valor de 1.0050 mientras que se tuvo un error relativo de 2,0051 % Estos resultados nos permiten afirmar que el teorema de Horner fue implementado y probado correctamente en R.

2. Tema 2: Números Binarios

2.1. Teoría

El sistema de números binarios cuenta con solo dos dígitos, el uno (1) y el cero(0), es la combinación de estos la que permite a este sistema numérico representar todos los números. Los números binarios son muy usados en el mundo de la computación y la electrónica debido a que su reducida cantidad de caracteres es fácil de representar con voltajes, como alto (1) o bajo (0) o también encendido(1) y apagado(0).

2.2. Solución 1

Para hallar los primeros 15 bits de π se convertirá el numero π mediante una herramienta digital y se tomaran los primeros 15bis decimales. Los métodos para convertir números decimales fraccionales a números binarios no serán explicados en este punto, pero se explican el punto de solución siguiente.

Para fines del ejercicio se tomara un valor de π igual a 3,1415926:

$\pi_2 = 11{,}001001000011111$

Se puede apreciar claramente en la parte entera del numero con los primeros dígitos que representan un 3, y luego en la parte fraccional se ve que no existe un patrón que se repita, esto debido a la naturaleza irracional del numero π

2.3. Solución 2 y 3

En este punto el enunciado pide convertir una serie de números en sistema binario a números del sistema decimal, a continuación se muestran los resultados:

Resultados:

Sistema Binario	Sistema Decimal
1010101	85
1011.101	11.6250
10111.010101	23.3281...
111.11111	7.9999...

Tabla de resultados sección 2

Sistema Decimal	Sistema Binario
11.25	1011.01
0.66...	0.10101010...
30.6	11110.10011001...
99.9	110011.11111111...

Tabla de resultados sección 3

A continuación una explicación teórica de como escribir los números binarios y decimales con partes fracciónales en un sistema numérico o en el otro. Se asume que ya se tienen los conocimientos sobre las partes enteras de los mismos números en ambos sistemas.

Para pasar fracciónales de **Decimal** a **Binario**:

- Se toma solo la parte fraccional del decimal, por ejemplo en el numero (10,3125 $\Leftrightarrow$ 0,3125), (4,67 $\Leftrightarrow$ 0,67), (0,2 $\Leftrightarrow$ 0,2)
- Luego la parte que tomamos la multiplicamos por 2 hasta que el resultado sea un 1 sin parte fraccional, durante este proceso se dan dos casos; en el primero el resultado de multiplicar por 2 es menor que 1, en este caso se escribe un 0; en el segundo caso al multiplicar obtenemos un numero mayor que 1 con parte fraccional, en este caso escribimos un 1 y tomamos la parte fraccional para la siguiente iteración. Lo siguiente es un ejemplo:
- 0,3125 (decimal) =>0,0101 (binario).

  Proceso:

$$0,3125 * 2 = 0,625 \rightarrow 0$$
$$0,625 * 2 = 1,25 \rightarrow 1$$

$$0,25 * 2 = 0,5 \rightarrow 0$$
$$0,5 * 2 = 1 \rightarrow 1$$
$$\text{En orden: } 0101 \rightarrow 0,0101 \text{ (binario)}$$

Para pasar fracciónales de **Binario** a **Decimal**:

- Una vez mas, se toma solo la parte fraccional del binario, por ejemplo en el numero (0,0110 $\Leftrightarrow$ 0,0110), (100110,111 $\Leftrightarrow$ 0,111), (0,1010 $\Leftrightarrow$ 0,1010)
- En este caso resulta útil usar la siguiente formula:

$$\sum_1^n X * \frac{1}{2^n}$$

- Donde X corresponde al n-enésimo numero de la secuencia de dígitos fracciónales binario s a convertir. A continuación un ejemplo con el numero binario 0.101 que corresponde a 0.625 en decimal:

$$1 * \tfrac{1}{2^1} + 0 * \tfrac{1}{2^2} + 1 * \tfrac{1}{2^3} = 0{,}625$$

3. Tema 3: Representación del Punto Flotante de los Números Reales, Épsilon de una máquina

3.1. Teoría

Se debe tener en cuenta que las computadores manejan un sistema aritmético binario lo cual representa cada número como una potencia de 2. En ese sentido ciertos números se pueden vislumbrar de manera exacta, pero por otro lado existen números como pi que llegan a tener una secuencia de decimales infinita, las cuales trascienden del sistema de representación binaria. Por eso mismo los computadores usan dos formatos diferentes de número: Punto fijo y punto flotante.

El estándar IEEE 754 establecido desde 1985, especifica todo aquello relacionado con la aritmética computacional de puntos flotantes, tanto formatos como reglas de redondeo y el correcto manejo de algunas excepciones. Según el estándar un punto flotante está compuesto del signo, la mantisa y un exponente y de acuerdo a lo anterior se definen tres niveles de precisión:

- Sencilla (32 bits)
- Doble (64 bits)
- Completa (80 bits)

3.2. Solución 1

El ajuste de un numero binario infinito en un numero finito de bits depende de la propiedad infinita del mismo numero binario,esto es, si el componente infinito del numero se ubica en un fraccional periódico entonces se debe colocar, una vez, el periodo tras el punto decimal y se debe agregar alguna seña que indique la secuencia infinita, como por ejemplo unos puntos suspensivos "...". Por otro lado, cuando el numero infinito que intentamos escribir de manera finita no presenta ningún patrón, ya sea porque es un numero infinitamente grande o porque es de naturaleza irracional se debería simplemente elegir una cantidad de bits a representar y redondear a esa cantidad.

Otra situación es si queremos representar el numero infinito en el estándar de IEEE, en este caso se colocan todos los bits del exponente mínimo en 1 y se usa el bit de signo para especificar si se trata de un infinito positivo o negativo.

3.3. Solución 2

El redondeo consiste en aproximar las cifra, si ésta es menor que 4 se deja igual, pero si es mayor o igual a 5 se le suma una unidad. Mientras que trucar un número consiste en reemplazar por ceros las cifras de orden inferior.

3.4. Solución 4

Cálculo del número de punto flotante (IEEE) para $fl(x)$ dónde $x = 0,4$:

- Conversión del número decimal a binario:

$$\begin{gathered} 0,4 * 2 = 0,8 \rightarrow 0 \\ 0,8 * 2 = 1,6 \rightarrow 1 \\ 0,6 * 2 = 1,2 \rightarrow 1 \\ 0,2 * 2 = 0,4 \rightarrow 0 \\ 0,4 * 2 = 0,8 \rightarrow 0 \\ \ldots \end{gathered}$$

- Entonces,

$$0{,}0110011001100110011001100110011\overline{0011}00$$

  Se puede observar que el número es infinito y además presenta periodicidad.

- Representación de este número en punto flotante con precisión doble:
  Signo (1 bit): 0
  Exponente (11 bits): $(01111111101)_2$
  Mantisa(52 bits):(1001100110011001100110011001100110011001100110011001)

  De esta manera podemos concluir que $fl(0,4)$ en el estándar IEEE 754 es aproximadamente $(0,3999999999999999966693309261245)_{10}$

3.5. Solución 5

Teniendo en cuenta que en el estándar IEEE 754 el error de redondeo relativo se denota como:

$$\frac{fl(x)-x}{|x|} <= \frac{1}{2}\epsilon_{maq}$$

El error de redondeo de para 0,4 es:

$$\frac{0{,}3999999999999999966693309261245-0{,}4}{|0{,}4|} <= \frac{2^{-52}}{2}$$

$$\frac{0{,}3999999999999999966693309261245-0{,}4}{|0{,}4|} <= 2^{-53}$$

$$8,32667268468875 * 10^{-17} <= 1,110223 * 10^{-16}$$

Entonces el error es aproximadamente de $8,327 * 10^{-17}$

3.6. Solución 6

En el caso del lenguaje Python, los tipos de datos básicos manejan los siguientes niveles de precisión:

- Entero (int) - precisión sencilla.
- Flotante (float) - precisión doble.
- Enteros largos (long) - puede manejar números con cualquier nivel de precisión.

Para el caso de R, los tipos de datos básicos como enteros, reales, enteros long e incluso números complejos se manejan con precisión doble.

3.7. Solución 7

Número de máquina en hexadecimal del número 9.4

- Conversión del número decimal a binario:

$$9 \rightarrow 1001$$

Por procedimientos anteriores sabemos que

$$0,4 \rightarrow 0,0110011001100110011001\overline{1100}$$

Entonces,

$$9,4 \rightarrow 1001,0110011001100110011001\overline{1100}$$

- Representación en número de máquina hexadecimal: 0x9

4. Convergencia Acelerada

4.1. Solución 1

Para la solución de este punto se creo un vector con los números del 1 al 15 y luego se le aplico la formula propuesta por el enunciado del punto, esta corresponde a : $x_n = cos(\frac{1}{n})$. Tras aplicar esta operación se obtuvo un vector con valores de x_n sobre el cual se va aplicar la siguiente secuencia:

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

Al aplicar la formula anterior sobre el arreglo de valores creados anteriormente se obtienen los valores consignados en la tabla siguiente:

Convergencia Acelerada
2.155016
2.155016
1.980651
1.972808
1.977327
1.982020
1.985675
1.988413
1.990472
1.992045

Tabla de resultados

El código realizado en R se encuentra en el mismo repositorio que este documento, bajo el nombre de "Aitken.R"

4.2. Solución 2

Para la solución de este ejercicio comenzamos igualando las dos funciones propuestas en el enunciado, para formar una solución sobre la cual trabajar:

$$3 * (\sin(x))^3 - 1 = 4 * \sin(x) * \cos(x)$$
$$3 * (\sin(x))^3 = 4 * \sin(x) * \cos(x) + 1$$
$$0 = 4 * \sin(x) * \cos(x) - 3 * (\sin(x))^3$$

Luego con las funciones igualadas, vamos a usar el método de bisección para obtener los siguientes valores de las raíces que en este caso corresponden a: 1.186495; 1.638489, se necesitaron un total de 28 iteraciones y el error fue igual a 0.0074505. Seria posible crear una convergencia acelerada con estos valores en el caso de que fueran mas, pero al solo obtener dos raíces no podemos aplicar la secuencia anterior.

5. Bibliografía

- https://cran.r-project.org/doc/contrib/R-intro-1.1.0-espanol.1.pdf