Reto 1

Luisa Contreras Jairo Vanegas Gabriel Forero

20 de septiembre de 2020

1. Punto uno: Evaluación de las raíces de un Polinomio

1.1. Implementación del algoritmo de Horner

1.1.1. Primera Derivada

El método de Horner, es un método que permite evaluar un polinomio de forma monomio, es decir que lo realiza por medio de multiplicaciones y sumas. Por lo tanto, para hallar Horner en la primera derivada del polinomio se utilizó la división sintética. Lo primero que se realizo fue la comprobación y cancelación de coeficientes en este se busca seleccionar y eliminar los números son suficientemente pequeños, para convertirlos en cero con el fin de que estos no intervengan con el resultado, esto lo realizamos encontrando los valores más pequeños, ubicamos su posición y finalmente convertimos el valor de esta posición en cero. Luego de ello se aplica la división sintética, como ya lo mencionamos, que se realiza reduciendo el primer término, luego multiplicándolo y por último almacenándolo, de esta manera se realiza recursivamente hasta haber divido cada monomio.

1.1.2. Segunda Derivada

Horner nos permite encontrar el valor de una función en un punto determinado garantizando que el número máximo de operaciones corresponde al grado del polinomio. Para la segunda derivada se implementó un arreglo de valores reales donde el índice de la posición. El algoritmo de Horner nos permite encontrar el valor de una función en un punto determinado para asegurar que el número máximo de operaciones y el orden del polinomio considera una serie de valores para su realización. Para lograr su implementación se realiza un arreglo de número reales donde el índice de la posición, que se logra entender de la siguiente manera:

$$P(x) = a_0X^n + a_1X^{n-1} + a_2X^{n-2} + a_3X^{n-3} + ... + a_nX^0 \tag{1}$$

$$[a_0][a_1][a_2][a_3]\dots[a_n] \tag{2}$$

Partiendo de ello, se implementaron un algoritmo que corresponde a la implementación de las derivadas. La función opera el arreglo antes mencionado de la forma $P(x) = a_0X^n + a_1X^{n-1} + a_2X^{n-2} + a_3X^{n-3} + ... + a_nX^0$ donde cada monomio es derivado por medio de la regla de derivación $n * aX^(n-1)$, para después ser sumado. Por esto, se tomó en cuenta que al realizar esta operación que el tamaño del arreglo es variable, entonces después de dicha operación se toma en cuenta que n no es igual al tamaño del arreglo si no, que será de $n = |Coeficientes| - 1$. Finalmente se obtiene un nuevo arreglo que contendrá los coeficientes de la función derivada y con esta se ejecuta el algoritmo de Horner nuevamente y obtendremos el resultado de la derivada del polinomio evaluado en un punto X_i.

1.1.3. Implementación del algoritmo

Algorithm 1: Algoritmo Horner

```
methods.horner <- function(coeficientes, x) {
    n <- length(coeficientes) - 1
    p <- coeficientes[1]
    d1 <- 0
    d2 <- 0
    multi.p <- 0
    multi.d1 <- 0
    multi.d2 <- 0
    for (i in 1:n) {
        d2 <- d2 * x + 2 * d1
        multi.d2 <- multi.d2 + 2
        d1 <- d1 * x + p
        multi.d1 <- multi.d1 + 1
        p <- p * x + coeficientes[i + 1]
        multi.p <- multi.p + 1
    }
    return(list(
        "p" = p,
        "d1" = d1,
        "d2" = d2,
        "multi.p" = multi.p,
        "multi.d1" = multi.p + multi.d1,
        "multi.d2" = multi.p + multi.d1 + multi.d2
    ))
}
```

1.2. Newton-Horner

Para el método de Newton se usa para calcular las raíces de un polinomio. Definida por una función derivable definida en los reales. Donde comenzamos con X_0 como valor inicial y definimos para cada número n, la siguiente recursión:

$$X_{n+1} = X_n - \frac{f(X_n)}{f'(X_n)} \tag{1}$$

Ya considerando ello usaremos Newton para calcular una raíz del polinomio y dividiremos el polinomio con el método primer método de Horner, ya definido. Para el orden de convergencia se tienen dos variables, en las cuales, en una se tiene cuantas veces se usa Horner en el algoritmo y cuantas veces el algoritmo recurre. De esta forma se almacenan de acuerdo van sucediendo luego de cada iteración y de cada proceso para poder determinar las iteraciones totales y las iteraciones del método propio.

1.2.1. Implementación del algoritmo

Algorithm 2: Algoritmo Newton-Horner

```
methods.newton_horner <- function(coeficientes, x, tolerancia) {
    i <- 0
    multi <- 0
    p <- 10000
    while (abs(p) > tolerancia) {
        horner <- methods.horner(coeficientes, x)
        p <- horner$p
        pD1 <- horner$d1
        multi <- multi + horner$multi.d1
        x2 <- x - (p / pD1)
        x <- x2
        horner <- methods.horner(coeficientes, x)
        p <- horner$p
        multi <- multi + horner$multi.d1
        i <- i + 1
    }
    return(list(
        "r" = x,
        "iter" = i,
        "multi" = multi
    ))
}
```

1.2.2. Método de Laguerre

Para implementar el algoritmo de Laguerre se hizo una intensiva búsqueda de la manera de como implementar el algoritmo, así que los siguieron los siguientes pasos: Sea P(x) un polinomio real cualquiera en una sola variable x de la forma:

$$P(x) = X^n + a_{n-1}^{n_1} + ... + a_1 X + a_0 \tag{1}$$

Luego de ello se ha determinado $a_n = 1$ y cuando $P(x) = 0$ se transforma en una ecuación algebraica con n raíces que se pueden denotar como $z_1, z_2, ..., z_{n-1}, Z_n$, de manera que se usan las siguientes fórmulas para su solución.

$$G = \frac{P'(X_k)}{P(x_k)} \tag{2}$$

$$H = G^2 - \frac{P''(X_k)}{P(x_k)} \tag{3}$$

$$S_k = \frac{n}{G + 0)\sqrt{(n-1)(nH - G^2)}} \tag{4}$$

$$X_{k+1} = X_k - S_k \tag{5}$$

1.2.3. Implementación del algoritmo

Algorithm 3: Algoritmo Laguerre

```
methods.laguerre <- function(coeficientes, x, tolerancia) {
    multi <- 0
    i <- 0
    p <- 10000
    n <- length(coeficientes) - 1
    while (abs(p) > tolerancia) {
        horner <- methods.horner(coeficientes, x)
        p <- horner$p
        pD1 <- horner$d1
        pD2 <- horner$d2
        multi <- multi + horner$multi.d2
        G <- pD1 / p
        H <- G^2 - (pD2 / p)
        factorInterno <- (n - 1) * ((n * H) - G^2)
        raiz <- sqrt(as.complex(as.numeric(factorInterno)))
        x2 <- 0
        if (abs(G + raiz) > abs(G - raiz)) {
            x2 <- x - (n / (G + raiz))
        } else {
            x2 <- x - (n / (G - raiz))
        }
        x <- x2
        i <- i + 1
        horner <- methods.horner(coeficientes, x)
        p <- horner$p
        multi <- multi + horner$multi.p
    }
    return(list(
        "r" = x,
        "iter" = i,
        "multi" = multi
    ))
}
```

1.3. Resultados

Tolerancia: 10^{-16}

X	Iteraciones Newton	Multiplicaciones Newton	Newton Resultados
48.430278	17	272	2.00000000000000000
6.818189	2	112	2.00000000000000000
31.538218	12	192	2.00000000000000000
7.267539	7	112	2.00000000000000000
-31.774527	11	176	-5.00000000000000000

Cuadro 1: Resultados del algoritmo de Newton

X	Iteraciones Laguerre	Multiplicaciones Laguerre	Laguerre Resultados
48.430278	6	120	2.00000000000000000
6.818189	5	100	2.00000000000000000
31.538218	8	160	2.00000000000000000
7.267539	4	80	2.00000000000000000
-31.774527	3	60	-5.00000000000000000

Cuadro 2: Resultados del algoritmo de Laguerre

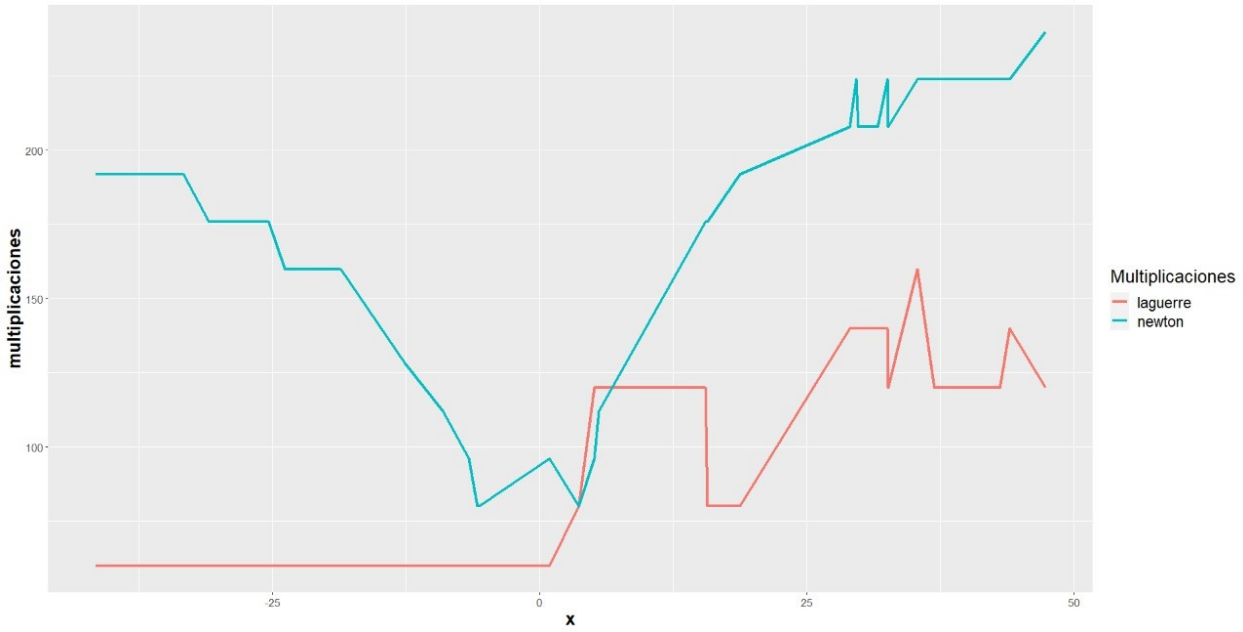

Figura 1: Resultados del algoritmo de Brent

1.3.1. Conclusiones

Luego de observar los resultados junto con la gráfica de Multiplicaciones, la cual corresponde como se puede observar a la variación entre las iteraciones que se deben realizar para llegar

al resultado final, de la cuál podemos concluir que es demasiado variable y no podemos determinar un factor de cambio como tal. Aunque en promedio las iteraciones de Newton son mayores a las de Laguerre. También podemos afirmar que no existe diferencia en los resultados obtenidos por ambos métodos, pues los dos llegan al mismo. Sin embargo, el método de Laguerre al interactuar con él es mucho más complicado y puede llegar a tener problemas con los números imaginarios por su forma, la cual corresponde a una raíz en el denominador de una división.

2. Punto dos: Algoritmo de Brent

El algoritmo de Brent busca las raíces de una función combinando tres métodos: Bisección, Secante e interpolación lineal. Logrando tener la confiabilidad de bisección, en menor cantidad de tiempo. A continuación, se explicará como se usó el algoritmo de Brent para encontrar las raíces de la función:
$F(x) = X^3 - 2X^2 + \frac{4X}{3} - \frac{8}{27}$
Para comprobar la veracidad del método se usó Wólfram con el fin de obtener la respuesta y así poder compararlo con el algoritmo de Brent. Teniendo como resultado que la raíz es:
$X = \frac{2}{3}$

2.1. Pseudocódigo

Las condiciones iniciales del algoritmo son:
1. La función o polinomio
2. El limite superior e inferior del intervalo
3. Tolerancia permitida
El algoritmo de Brent se implementó en Python, a continuación, se muestra el pseudocódigo:
Se definen los intervalos de la función [a,b]
Calcular f(a)
Calcular f(b)
if f(a)*f(b)$\geq$ 0 then
Sale de la función por que la raíz no esta en el intervalo
end if
if $|f(a)| \leq |f(b)|$ then
Se intercambian los valores a y b
end if
c = a
repeat until f(b) = 0 or f(s) = 0 or $|b - a| \geq$ tolerancia (converge)
if f(a) $\neq$ f(c) and f(b) $\neq$ f(c) then
s= Se usa la formula de la interpolación lineal para calcular la raiz
else
s= Se usa la formula de la secante para calcular la raiz
end if
if s is not between $\frac{3a+b}{4}$ and b or
$(|s - b| \geq \frac{|b-c|}{2})$ or

($|s-b| \geq \frac{|c-d|}{2}$) or
($|b-c| \leq |s|$) or
($|c-d| \leq |s|$) then
s=Se usa la formula del metodo bisección para calcular la raiz
end if
calculate f(s)
d = c (d no se usa en la primera iteración)
c = b
if f(a)*f(s) $\leq$ 0 then
b = s
else
a = s
end if
if $|f(a)| \leq |f(b)|$ then
Se intercambian los valores a y b
end if
end repeat
La raíz se encuentra en el valor de s

2.2. Resultados

Los resultados que se obtuvieron con una tolerancia de ϵ^{-14} fueron los siguientes:

Iteración	Raíz
1	-0,007936507936508
2	0,488095238095238
3	0,736111111111111
4	0,612103174603174
5	0,674107142857142
6	0,643105158730158
7	0,658606150793650
8	0,666356646825397
9	0,670231894841270
10	0,668294270833333
11	0,667325458829365
12	0,666841052827381
13	0,666598849826389
14	0,666719951326885
15	0,666659400576637

Cuadro 3: Resultados del algoritmo

Al graficar obtenemos la siguiente aproximación a la raíz:

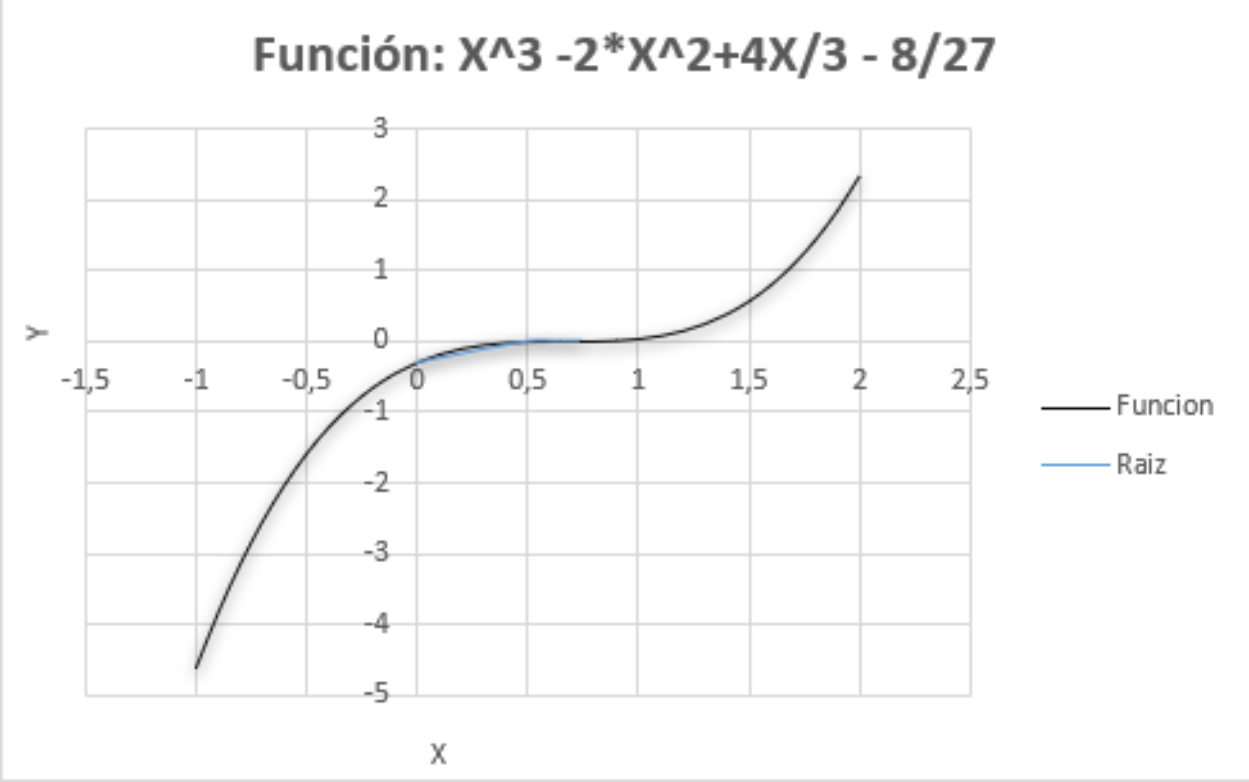

Figura 2: Resultados del algoritmo de Brent

2.3. Pruebas

Las pruebas se realizaran con diferentes significancias, ya que variar los intervalos no tiene mucho impacto. Pues si la raíz esta en el intervalo la encontrara y sino simplemente se saldrá. Los resultados son los siguientes

Significancia	Raíz
ϵ^{-7}	0.6666652
ϵ^{-10}	0.6666652109
ϵ^{-15}	0.6666652109770974

Cuadro 4: Pruebas

A partir de la tabla 2 se concluye que a medida que aumenta la tolerancia el resultado de la raíz es más exacto y tiene una mejor aproximación

2.4. Comparación con el método secante

Se comparo el método de Brent con el método de la secante implementado en el taller pasado. A partir de esto se concluyo que el método de Brent es mas rápido y a su vez mas exacto que el método de la secante, esto se puede apreciar en la tabla 3, donde Brent llega a una solución con 35 iteraciones menos.

Concepto	Secante	Brent
Iteraciones	50	15
Resultado	0.6666698708192892	0.6666652109770974

Cuadro 5: Comparación del método de la secante y el método de Brent

3. Punto tres: Óptima Aproximación Polinómica

3.1. Aproximación por el algoritmo modificado de Remez

Como podemos ver en el documento adjunto al reto [1], el algoritmo de Remez es un algoritmo iterativo que aproxima una función cualquiera usando el método de aproximación mínima, por el que se minimiza la función de error en diferentes puntos que se escogen entre el conjunto de números de Chebyshev relevantes para la función a aproximar. Para el alcance de este documento, no se implementó el algoritmo, en cambio se usó la solución encontrada en el documento [1] para la función

$$f(x) = e^{sin(x)-cos(x^2)} \tag{3.1}$$

en el intervalo $[-2^{-8}; 2^{-8}]$ y un error menor de 2^{-90}.

3.2. Aproximación por el algoritmo de Taylor

Para la aproximación por el algoritmo de Taylor utilizamos la función taylor() que podemos encontrar en la librería pracma [2] para el lenguaje de programación R. La función taylor() trabaja con una función derivable, en este caso (3.1), el punto x_0 donde se aproximara el polinomio, y el grado del polinomio de Taylor, esta librería solo permite polinomios de grado $n \leq 4$. El resultado de esta función es un vector con los coeficientes del polinomio aproximado en la forma:

$$\sum_{i=0}^{n} C_i * x^{n-i} \tag{3.2}$$

3.3. Procedimiento

Para proveer la precisión necesaria para la comparación de las dos aproximaciones, la de Taylor y la de Remez modificado, utilizamos la librería Rmpfr [3] de R. Debido a que las significancias de los números en este experimento están en base 2, se usan los exponentes como los bits significativos a la hora de convertir los números en sus equivalentes números de precisión múltiple.
Para comparar las aproximaciones se procede a usar el algoritmo de Horner, implementado en el punto (1), para evaluar las aproximaciones en un valor seleccionado aleatoriamente dentro del rango tal que

$$x \in [-2^{-8}; 2^{-8}] \tag{3.3}$$

estos valores evaluados en cada aproximación se comparan entonces con el valor real entregado por la evaluación directa de la función (3.1), para calcular así el error absoluto de acuerdo

a la siguiente ecuación

$$e = |X_r - X_a| \tag{3.4}$$

con X_r siendo el valor evaluado directamente y X_a el valor evaluado en la aproximación correspondiente.

3.4. Resultados

Se procede a realizar este experimento con 100 valores de x y con estos valores del error obtenemos la siguiente gráfica:

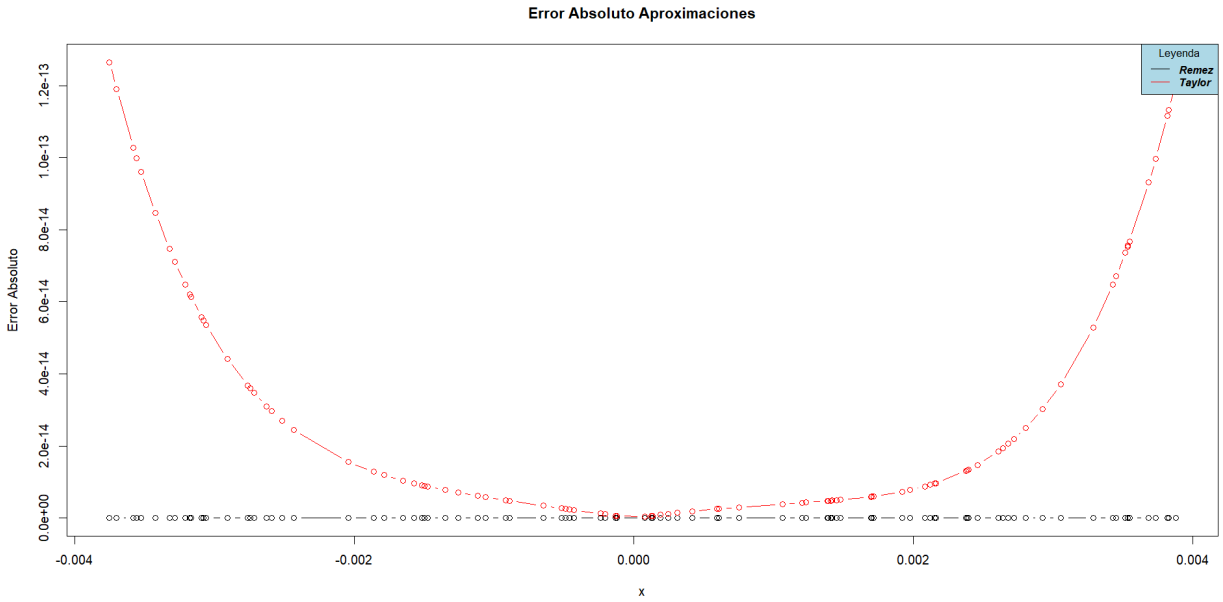

Figura 3: Remez Vs Taylor

De esta gráfica podemos ver como el error absoluto de la aproximación de Remez es tan bajo que pareciera que es constante en comparación con el error de la aproximación de Taylor la cual disminuye a medida que se acerca al punto x_0, en este caso $x_0 = 0$, en donde fue aproximado y vuelve a crecer a medida que se aleja de dicho punto. Esta muy claro que para este problema en específico la aproximación de Remez se presenta como la mejor aproximación con valores de error ínfimos respecto al valor real de la función. Los valores exactos se pueden obtener al correr el algoritmo en R adjunto a este documento. Sin embargo como se puede ver en la siguiente gráfica

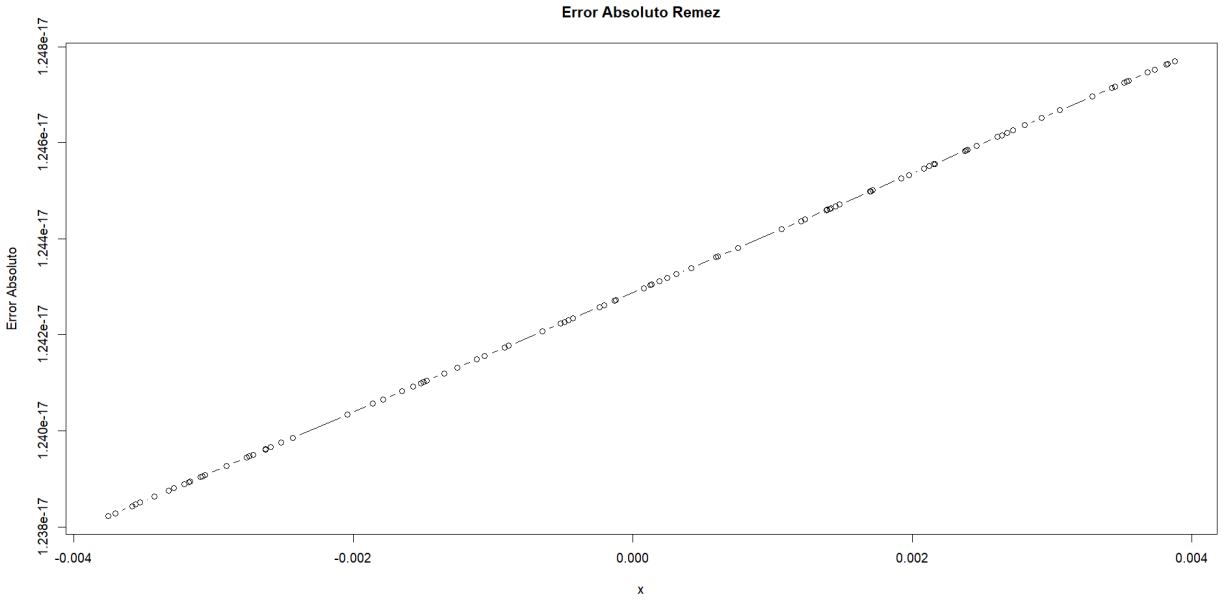

Figura 4: Errores absolutos Remez

El error absoluto en la aproximación de Remez crece de forma lineal a medida que avanzamos en la recta numérica, este comportamiento puede explicarse debido a que a medida que nos alejamos del nodo de en donde se encontró el minimax, el error crece naturalmente hasta que se acerque al siguiente nodo en donde empezara a disminuir se intuye de manera también lineal.

Referencias

[1] F. De Dinechin and C. Q. Lauter, "Optimizing polynomials for floating-point implementation," *arXiv:0803.0439 [cs]*, Mar. 2008. arXiv: 0803.0439.

[2] H. W. Borchers, "pracma: Practical Numerical Math Functions," Dec. 2019.

[3] M. Maechler, R. M. Heiberger (formatHex(), *Bin, and *Dec), "Rmpfr: R MPFR - Multiple Precision Floating-Point Reliable," Jan. 2020.