Reto 1 Análisis Númerico

Edwin Turizo
Juan Pimienta

Marzo 1 2020

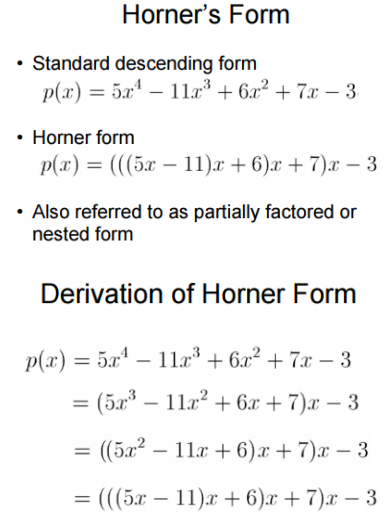
1. Evaluación de un Polinomio

1.1. Problema 1

Implemente en R o Python el método de Horner para evaluar f '(x_0), tenga en cuenta la precisión de la computadora o unidad de redondeo.

Solución

El metodo Horner para derivadas varía de alguna forma, puesto que en un polinomio cuando uno evalua un valor numerico, se reliza en un mismo ciclo teniendo n iteracciones u operaciones, donde n es el grado del polinomio. En el caso de la derivada, la evaluación se realiza en n-1 iteraciones debido a que en una derivada, el polinomio elimina en uno su grado.
Esto hace mucho mas eficiente y menos complejo resolver para una maquina la evaluación de un valor x en una derivada que el polinomio original.
Para explicar de una mejor forma se muestra la siguiente imagen que ejemplifica una manera de entender como funciona Horner.



Esto lo que nos muestra es la recursividad o reutilización de calculos realizados anteriormente al obtener el valor evaluado de f(x) y ahora utilizarlo en la evaluación de f'(x).
Se identifica un patron entonces de que en base a la evaluación realizada en f(x) sí se reevalúa

con los datos tomados se puede conseguir la derivada en ese mismo punto.
El caso que se aplico para probar la funcionalidad del algoritmo fue el siguiente polinomio y por medio de una plataforma web se calculo la derivada en el punto escogido. Lo siguiente se muestra a continuación

$$\text{Derivada de } 3x^3 + 7x^2 + 2x - 5 \text{ cuando } x = 31.9122: \quad 9614.27$$

Ahora, la salida del algoritmo planteado aquí muestra lo siguiente

```
Derivada de f(x) en x=  31.9122  =  9614.267379559998516924679279327392578125  realizada en  3  iteraciones
```

Con esto podemos ver que la aplicación de la precisión doble decimal en los calculos y resultados de la derivada mejoran la representación aumentando el numero de decimales significativos y a continuación se muestra los errores entre estos 2 datos tomados.

- Error Absoluto: 0.0026204400019196328
- Error Relativo: 2.72557355048239e-07

En cuanto a la precisión de la computadora, la implementación en el lenguaje de programación Python nos resuelve de alguna forma dicho problema, puesto que Python tiene en cuenta dicha precisión. Esta precisión utiliza el epsilon de la maquina pero tambien el punto flotante doble o simple según sea el caso.
Algoritmo realizado en Python.

```python
#Función Derivada del metodo de Horner
from decimal import Decimal

def horner(coeficientes, grado, valor):
  iteraciones=1
  y = z = coeficientes[0]
  for j in range(1, grado):
    y = valor * y + coeficientes[j]
    z = valor * z + y
    iteraciones+=1
  return Decimal.from_float(z),iteraciones
coef= [3,7,2,-5]
grado=3
valor=31.9122
[deriv,itera]=horner(coef,grado,valor)
print("Derivada de f(x) en x= ",valor," = ",deriv," realizada en ",itera," iteraciones")
```

1.2. Problema 2

Implemente el método de Horner si se realiza con números complejos, tenga en cuenta la precisión.

Solución

El metodo de Horner para numeros complejos, en especial para polinomios complejos, tiene unas minimas variaciones gracias al lenguaje en que fue implementado: Python. Puesto que

este tiene soporte para operacioens con numeros complejos como la suma, resta, multiplicación y división. Es entonces que la implementación realizada agrupa los casos de evaluación de polinomio complejo y tambien la evaluacíon en la derivada del mismo polinomio complejo de manera correcta.
Como se explico anteriormente, en los calculos de numeros complejos se utilizan metodos de precisión del mismo lenguaje de programación que entiende la maquina lo que facilita el redondeo en los resultados del metodo Horner. En este caso, el tipo de dato Decimal no se puede utilizar puesto que ya se tiene aplicado el tipo de dato Complejo en Python, sin embargo, la representación de decimales por medio de float en la parte real del numero complejo se realiza de forma bastante precisa.
En este caso de numeros complejos se hace necesario implementar adicionalmente una funcion que cuente el numero de operaciones matematicas basicas, puesto que el manejo de estas en numeros complejos varía en gran medida.
Como caso de prueba se tomo un polinomio complejo con la lista de coeficientes: [(2+3i),(0+1i),5,2] y valor de x = (1+1i).
La salida del algoritmo se muestra a continuación

```
Evaluación de f(x) en x =  (1+1j)  =  (-5+3j)
Derivada de f(x) en x =  (1+1j)  =  (-15+14j)
Se realizaron  2  iteraciones en el ciclo.
Se realizaron  20  multiplicaciones y  25  sumas
```

Algoritmo realizado en Python.

```python
#Metodo Horner Numeros Complejos
def contadorOperaciones(sumas,productos,valor,num):
  if type(valor)==complex and type(num)==complex:
    sumas+=3
    productos+=4
  elif (type(num)==complex and type(valor)!=complex) or (type(num)!=complex and type(valor)==complex):
    sumas+=1
    productos+=2
  elif type(num)!=complex and type(valor)!=complex:
    productos+=1
  return sumas,productos

def hornerComplex(coeficientes, grado, valor):
  iteraciones=1
  y = z = coeficientes[0]
  sumas=productos=0
  for j in range(1, grado):
    [sumas,productos]=contadorOperaciones(sumas,productos,valor,y)
    [sumas,productos]=contadorOperaciones(sumas,productos,valor,z)
    y = valor * y + coeficientes[j]
    z = valor * z + y
    sumas+=4
  [sumas,productos]=contadorOperaciones(sumas,productos,valor,y)
  y = coeficientes[-1] + (valor* y)
  sumas+=2
  iteraciones+=1
  return (y,z,productos,sumas,iteraciones)
coefComplex= [(2+3j),(0+1j),5,2]
coefff=[(6+9j),(0+2j),5]
grado=3
valorComplex=1+1j
[normal,deriv,productos,sumas,iteraciones]=hornerComplex(coefComplex,grado,valorComplex)
print("Evaluación de f(x) en x = ",valorComplex," = ",normal)
print("Derivada de f(x) en x = ",valorComplex," = ",deriv)
print("Se realizaron ",iteraciones," iteraciones en el ciclo.")
print("Se realizaron ",productos," multiplicaciones y ",sumas," sumas")
```

2. Optima Aproximación Polinomica

2.1. Problema 1

Aplique una aproximación de Taylor

Solución

Se utilizo una aproximación de taylor basada en el teorema de taylor y su aplicación en la aproximación de polinomios. En este caso se utilizo para la función Sen(x).

$$sen\,x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \cdots \quad , \forall x \in R$$

El algoritmo planteado obtiene los datos del numero de iteraciones especificando el grado del polinomio aproximado obtenido, ademas de el valor aproximado de taylor, el valor exacto calculado de la funcion, y el error entre estos 2 datos.

La salida del algoritmo se muestra a continuación

```
Iteracion                AproxTaylor                         CalcValue                                                  Error
1         0.04091261478765950698743836256         0.0409012022003520658852693259177613072097301483154296875         0.0000114125873074411021
6903664224
2         0.04090120124516676625495408687         0.0409012022003520658852693259177613072097301483154296875         9.5518529963031523904776
13072E-10
3         0.04090120220039013373982636034         0.0409012022003520658852693259177613072097301483154296875         3.8067854557034422238692
79027E-14
Grado del polinomio de la mejor aproximacion 2
Coeficientes del polinomio de la mejor aproximacion de Taylor [Decimal('0.04091261478765950698743836256'), Decimal('0.04090120124516676662
5495408687'), Decimal('0.04090120220039013373982636034')]
```

Algoritmo realizado en Python.

```python
 #Aproximacion Taylor Sin(x)
import math
import numpy as np
import matplotlib.pyplot as plt
import os
from decimal import Decimal

def func_sin(x, n):
    sin_aprox = 0
    coefs=[]
    for i in range(n):
        coef = (-1)**i
        num = x**(2*i+1)
        denom = math.factorial(2*i+1)
        sin_aprox += (Decimal.from_float(coef)) * ( Decimal.from_float((num))/Decimal.from_float((denom)))
        coefs.append(sin_aprox)
    return (sin_aprox,coefs)

errorEsperado=1e-12
li=-((math.pi)/64)
ls=((math.pi)/64)

angles = np.arange(li,ls,0.01)
p_sin = np.sin(angles)
fig, ax = plt.subplots()
ax.plot(angles,p_sin)

iteracion=1
print("Iteracion ","\t\t\tAproxTaylor","\t\t\tCalcValue","\t\t\tError")
while True:
    t_sin=[]
    sin_exp=[]
    errores=[]
    for angle in angles:
      sinAprox,coefs=func_sin(angle,iteracion)
      t_sin.append(sinAprox)
      sinExp=Decimal.from_float(math.sin(angle))
      sin_exp.append(sinExp)

    error=abs(t_sin[-1] - sin_exp[-1])
    print(iteracion,"\t\t",t_sin[-1],"\t\t",sin_exp[-1],"\t\t",error)
    ax.plot(angles,t_sin)
    if error < errorEsperado:
        break
    iteracion+=1

print("Grado del polinomio de la mejor aproximacion",iteracion-1)
print("Coeficientes del polinomio de la mejor aproximacion de Taylor",coefs)
#ax.set_ylim([-7,4])
#ax.set_xlim([-np.pi/64,np.pi/64])

legend_lst = ['sin() function']
for i in range(1,iteracion+1):
    legend_lst.append(f'Taylor Series - {i} terms')
ax.legend(legend_lst, loc=3)
#plt.savefig(os.path.join(os.path.dirname(os.path.realpath(__file__)),'taylor_sin_approximation.png'))
```

2.2. Problema 2

Implemente el metodo de Remez

Solución

Para la implementación del metodo de Remez se utilizaron los polinomios de Chebyshev y en el caso de la aproximación MinMax.

Algoritmo realizado en Pyton.

```python
#Metodo de Remez
import math
import numpy as np
from matplotlib import pyplot as plt
import os

#Nodos de chebyshev
def _get_chebyshev_nodes(n, a, b):
  nodes = [.5 * (a + b) + .5 * (b - a) * math.cos((2 * k + 1) / (2. * n) * math.pi) for k in range(n)]
return nodes

def _get_errors(exactvals, polycoeff, nodes):
  ys = np.polyval(polycoeff, nodes)
for i in range(len(ys)):
  ys[i] = abs(ys[i] - exactvals[i])
return ys

def run_remez(fun, a, b, d, error):
  finished = False
cn = _get_chebyshev_nodes(d + 2, a, b)
cn2 = _get_chebyshev_nodes(100 * d, a, b)

iteracion = 0
while not finished and len(cn) == d + 2 and iteracion < 50:
  iteracion += 1
b = np.array([fun(c) for c in cn])
A = np.matrix(np.zeros([d + 2,d + 2]))
for i in range(d + 2):
  x = 1.
x *= cn[i]
for j in range(d + 2):
  A[i, j] = x
x *= cn[i]
A[i, -1] = (-1)**(i + 1)
res = np.linalg.solve(A, b)

revlist = reversed(res[0:-1])
sccoeff = []
for c in revlist:
  sccoeff.append(c)
errs = _get_errors([fun(c) for c in cn2], sccoeff, cn2)
maximum_indices = []

if errs[0] > errs[1]:
  maximum_indices.append(0)
for i in range(1, len(errs) - 1):
  if errs[i] > errs[i-1] and errs[i] > errs[i+1]:
  maximum_indices.append(i)
if errs[-1] > errs[-2]:
  maximum_indices.append(-1)

finished = True
for idx in maximum_indices[1:]:
  if abs(errs[idx] - errs[maximum_indices[0]]) > error:
  finished = False
cn = [cn2[i] for i in maximum_indices]

return (list(reversed(res[0:-1])))

def f(x):
  return math.sin(x)
error=1e-12

a = -math.pi/64
b = math.pi/64
degree = 3
angles = np.arange(a,b,0.01)
err, coeffs = run_remez(f, a, b,degree,error)
print("Coefficients{}: {}", list(reversed(coeffs)))
```

3. Conclusiones

Para la realización de estos algoritmos se utilizo el lenguaje de programación Python, el cual representa los numeros decimales a traves del tipo de dato float a bajo nivel, es decir, utilizando 64 bits, luego en Python siempre se utiliza doble precisión, y en concreto se sigue el estándar IEEE 754: 1 bit para el signo, 11 para el exponente, y 52 para la mantisa. Esto significa que los valores que puede representar van desde 2,2250738585072020 x 10-308 hasta 1,79769313486231571030 8. Pero esto tiene sus limitaciones, impuestas por el hardware. Por eso Python implemento un nuevo tipo de dato "Decimal", para el caso de que se necesite representar fracciones o numeros decimales de forma más precisa. En cada uno de los algoritmos presentados se hizo uso del tipo de dato Decimal para la representación precisa de los decimales y reducir el error en los calculos.