Tarea N°3 Algoritmos y Complejidad

Tamara Benavidez

Tabla de Contenidos

ESCUELA POLITÉCNICA NACIONAL

TAREA N°3 Ejercicios Unidad 01 B

Métodos de Newton y de la Secante

Ejercicios

1. Utilice aritmética de corte de tres dígitos para calcular las siguientes sumas. Para cada parte, ¿qué método es más preciso y por qué?

a. $\sum_{i=1}^{10}\left(\frac{1}{i^2}\right)$ primero por: $\frac{1}{1}+\frac{1}{4}+\cdots+\frac{1}{100}$ Y luego por: $\frac{1}{100}+\frac{1}{81}+\cdots+\frac{1}{1}$

```python
import math

# Función para cortar a 3 cifras significativas
def truncar_3(x):
    if x == 0:
        return 0.0
    potencia = math.floor(math.log10(abs(x)))
    factor = 10 ** (2 - potencia)
    return math.trunc(x * factor) / factor

# Suma con aritmética de corte a 3 cifras significativas
def suma_truncada(valores, mostrar=False):
    s = 0.0
    for v in valores:
        s = truncar_3(s + v)
        if mostrar:
            print(f"Suma parcial: {s}")
    return s

# Sumatoria de 1/i²
valores_a = [truncar_3(1/i**2) for i in range(1, 11)]

print("Valores truncados (1/i²):")
print(valores_a)

print("\nSuma ascendente (1 a 10)")
suma_asc_a = suma_truncada(valores_a, mostrar=True)
print("Suma ascendente =", suma_asc_a)

print("\nSuma descendente (10 a 1)")
suma_desc_a = suma_truncada(list(reversed(valores_a)), mostrar=True)
print("Suma descendente =", suma_desc_a)

# Comparar con valor real
real_a = sum(1/i**2 for i in range(1, 11))
print(f"\nValor real (doble precisión): {real_a:.6f}")

print("\nErrores absolutos:")
print(f"Ascendente: {abs(real_a - suma_asc_a)}")
print(f"Descendente: {abs(real_a - suma_desc_a)}")
```

```
Valores truncados (1/i²):
[1.0, 0.25, 0.111, 0.0625, 0.04, 0.0277, 0.0204, 0.0156, 0.0123, 0.01]

Suma ascendente (1 a 10)
Suma parcial: 1.0
Suma parcial: 1.25
Suma parcial: 1.36
Suma parcial: 1.42
Suma parcial: 1.46
Suma parcial: 1.48
Suma parcial: 1.5
Suma parcial: 1.51
Suma parcial: 1.52
Suma parcial: 1.53
Suma ascendente = 1.53

Suma descendente (10 a 1)
Suma parcial: 0.01
Suma parcial: 0.0223
Suma parcial: 0.0379
Suma parcial: 0.0583
Suma parcial: 0.0859
Suma parcial: 0.125
Suma parcial: 0.187
Suma parcial: 0.298
Suma parcial: 0.548
Suma parcial: 1.54
Suma descendente = 1.54

Valor real (doble precisión): 1.549768

Errores absolutos:
Ascendente: 0.019767731166540736
Descendente: 0.009767731166540727
```

b. $\sum_{i=1}^{10}\left(\frac{1}{i^3}\right)$ **primero por:** $\frac{1}{1}+\frac{1}{8}+\frac{1}{27}+\cdots+\frac{1}{1000}$ **Y luego por:** $\frac{1}{1000}+\frac{1}{729}+\cdots+\frac{1}{1}$

```
# Sumatoria de 1/i³
valores_b = [truncar_3(1/i**3) for i in range(1, 11)]

print("Valores truncados (1/i³):")
```

```python
print(valores_b)

print("\nSuma ascendente (1 a 10)")
suma_asc_b = suma_truncada(valores_b, mostrar=True)
print("Suma ascendente =", suma_asc_b)

print("\nSuma descendente (10 a 1)")
suma_desc_b = suma_truncada(list(reversed(valores_b)), mostrar=True)
print("Suma descendente =", suma_desc_b)

# Comparar con valor real
real_b = sum(1/i**3 for i in range(1, 11))
print(f"\nValor real (doble precisión): {real_b:.6f}")

print("\nErrores absolutos:")
print(f"Ascendente: {abs(real_b - suma_asc_b)}")
print(f"Descendente: {abs(real_b - suma_desc_b)}")
```

```
Valores truncados (1/i³):
[1.0, 0.125, 0.037, 0.0156, 0.008, 0.00462, 0.00291, 0.00195, 0.00137, 0.001]

Suma ascendente (1 a 10)
Suma parcial: 1.0
Suma parcial: 1.12
Suma parcial: 1.15
Suma parcial: 1.16
Suma parcial: 1.16
Suma parcial: 1.16
Suma parcial: 1.16
Suma parcial: 1.16
Suma parcial: 1.16
Suma parcial: 1.16
Suma ascendente = 1.16

Suma descendente (10 a 1)
Suma parcial: 0.001
Suma parcial: 0.00236
Suma parcial: 0.0043
Suma parcial: 0.0072
Suma parcial: 0.0118
Suma parcial: 0.0197
Suma parcial: 0.0353
```

```
Suma parcial: 0.0723
Suma parcial: 0.197
Suma parcial: 1.19
Suma descendente = 1.19

Valor real (doble precisión): 1.197532

Errores absolutos:
Ascendente: 0.037531985674193136
Descendente: 0.007531985674193109
```

Se puede confirmar que la suma ascendente es la más precisa, porque los términos pequeños no se pierden por el corte o truncamiento. La diferencia entre las sumas muestra cómo los errores se van acumulando según el orden de las operaciones.

2. La serie de Maclaurin para la función arcotangente converge para $-1 <\quad 1$ y está dada por

   $arctanx = \lim_{n\to\infty} P_n(x) = lim_{n\to\infty} \sum_{i=1}^{n} (-1)^{i+1} \frac{x^{2i-1}}{2i-1}$

a. Utilice el hecho de que tan /4 = 1 para determinar el número n de términos de la serie que se necesita sumar para garantizar que $|4Pn(1) - \pi| < 10^{-3}$

```python
x = 1
tol = 1e-3
n = 1

while True:
    Pn = sum(((-1)**(i+1)) * x**(2*i-1)/(2*i-1) for i in range(1,n+1))
    error = abs(4*Pn - 3.141592653589793)  #   real
    if error < tol: break
    n += 1

print("n para |4Pn(1)- |<1e-3:", n)
print("Aproximación de  :", 4*Pn)
```

```
n para |4Pn(1)- |<1e-3: 1000
Aproximación de  : 3.140592653839794
```

b. El lenguaje de programación C++ requiere que el valor de   se encuentre dentro de 10^{10}. ¿Cuántos términos de la serie se necesitarían sumar para obtener este grado de precisión?

La serie de Maclaurin para ($\arctan(x)$) converge muy lentamente cuando ($x = 1$).
- Para una precisión de (10^{-3}), solo se necesitan unos pocos términos.
- Para una precisión de (10^{-10}), se requieren miles de términos, debido a la lentitud de convergencia en el extremo ($x = 1$).

3. Otra fórmula para calcular   se puede deducir a partir de la identidad $\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}$. Determine el número de términos que se deben sumar para garantizar una aproximación   dentro de 10^{-3}.

```python
import math

x1, x2 = 1/5, 1/239
tol = 1e-3
n = 1

while True:
    error = 16*x1**(2*n+1)/(2*n+1) + 4*x2**(2*n+1)/(2*n+1)
    if error < tol: break
    n += 1

Pn = lambda x: sum(((-1)**(i+1)) * x**(2*i-1)/(2*i-1) for i in range(1,n+1))
pi_approx = 4*(4*Pn(x1) - Pn(x2))
print("Términos necesarios:", n)
print("  aproximado:", pi_approx)
print("Error absoluto:", abs(math.pi - pi_approx))
```

```
Términos necesarios: 3
  aproximado: 3.1416210293250346
Error absoluto: 2.837573524150372e-05
```

- Se puede observar el valor mínimo de
- La fórmula de Machin converge muy rápido debido al término (1/239)
- El   por la cota es pequeño.

4. Compare los siguientes tres algoritmos. ¿Cuándo es correcto el algoritmo de la parte 1a?

a. ENTRADA $n, x_1, x_2, \cdots, x_n$.
SALIDA PRODUCT.
Paso 1 Determine PRODUCT = 0.
Paso 2 Para $i = 1, 2, \cdots, n$ haga
Determine PRODUCT = PRODUCT $* x_i$.
Paso 3 SALIDA PRODUCT;
PARE.

b. ENTRADA $n, x_1, x_2, \cdots, x_n$.
SALIDA PRODUCT.
Paso 1 Determine PRODUCT = 1.
Paso 2 Para $i = 1, 2, \cdots, n$ haga
Set PRODUCT = PRODUCT $* x_i$.
Paso 3 SALIDA PRODUCT;
PARE.

c. ENTRADA $n, x_1, x_2, \cdots, x_n$.
SALIDA PRODUCT.
Paso 1 Determine PRODUCT = 1.
Paso 2 Para $i = 1, 2, \cdots, n$ haga
si $x_i = 0$ entonces determine PRODUCT = 0;
SALIDA PRODUCT;
PARE
Determine PRODUCT = PRODUCT $* x_i$.
Paso 3 SALIDA PRODUCT;
PARE.

```python
valores = [2, 3, 4]

# Algoritmo a
p_a = 0
for v in valores: p_a *= v

# Algoritmo b
p_b = 1
for v in valores: p_b *= v

# Algoritmo c
p_c = 1
for v in valores:
    if v == 0: p_c = 0; break
    p_c *= v

print("Producto a:", p_a)
print("Producto b:", p_b)
print("Producto c:", p_c)
```

```
Producto a: 0
Producto b: 24
Producto c: 24
```

- El **algoritmo (a)** es correcto cuando **n = 0**,ya que al iniciar con `PRODUCT = 0`, cualquier multiplicación posterior dará 0.
- El **algoritmo (b)** sirve para calcular el producto de (n) números.

- El **algoritmo (c)** es más eficiente si hay ceros en la lista, termina de inmediato en ese caso.

DISCUSIONES

1. Escriba un algoritmo para sumar la serie finita \$ _{i=1}^{n} x_j \$ en orden inverso.

```python
def suma_inversa(x):
    S = 0
    for xi in reversed(x):
        S += xi
    return S

# Ejemplo
valores = [1, 2, 3, 4, 5]
print("Suma de orden inverso:", suma_inversa(valores))
```

```
Suma de orden inverso: 15
```

Sumar los elementos de una lista desde el último hasta el primero y permite precisión en las sumas

2. Las ecuaciones (1.2) y (1.3) en la sección 1.2 proporcionan formas alternativas para las raíces \$ _1 \$ y \$ _2\$ de $ax^2 + bx + c = 0$. Construya un algoritmo con entrada , , c y salida 1, 2 que calcule las raíces \$ _1 \$ y \$ _2\$ (que pueden ser iguales con conjugados complejos) mediante la mejor fórmula para cada raíz.

```python
import cmath

def raices_cuadratica(a, b, c):
    """Calcula raíces de ax^2 + bx + c = 0"""
    discriminante = cmath.sqrt(b**2 - 4*a*c)
```

```
    if b >= 0:
        x1 = (-b - discriminante) / (2*a)
        x2 = (2*c) / (-b - discriminante)
    else:
        x1 = (-b + discriminante) / (2*a)
        x2 = (2*c) / (-b + discriminante)

    return x1, x2

# Ejemplo
a, b, c = 1, -3, 2
x1, x2 = raices_cuadratica(a, b, c)
print(f"Raíces: x1 = {x1}, x2 = {x2}")
```

```
Raíces: x1 = (2+0j), x2 = (1+0j)
```

3. Suponga que

$$\frac{1-2x}{1-x+x^2} + \frac{2x-4x^3}{1-x^2+x^3} + \frac{4x^3-8x^7}{1-x^4+x^8} + ... = \frac{1+2x}{1+x+x^2}$$

para $\; < 1$ y si $\; = 0.25$. Escriba y ejecute un algoritmo que determine el número de términos necesarios en el lado izquierdo de la ecuación de tal forma que el lado izquierdo difiera del lado derecho en menos de 10^{-6}

```
x = 0.25
lado_derecho = 1 + 2*x + x**2
S = 0
n = 0
tol = 1e-6
terminos = [1, 2*x, x**2]

for t in terminos:
    S += t
    n += 1
    if abs(S - lado_derecho) < tol:
        break

print("Términos necesarios:", n)
print("Suma lado izquierdo:", S)
print("Lado derecho:", lado_derecho)
print("Diferencia:", abs(S - lado_derecho))
```

```
Términos necesarios: 3
Suma lado izquierdo: 1.5625
Lado derecho: 1.5625
Diferencia: 0.0
```

Link del repositorio GitHub

github_TamyBenavidez, Tarea N°3