ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA EN SISTEMAS

Métodos Numéricos

Tarea No: 3

Algoritmos y Complejidad

DOCENTE

Ing. Jonathan Zea

Nombre

- Sebastián Giovanny Oña Andrade

FECHA DE ENTREGA

Miércoles, 5 de noviembre del 2025

PERIÓDO ACADÉMICO

2025-B

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) $\sum_{i=0}^{10}\left(\frac{1}{i^2}\right)$ primero por $\frac{1}{1}+\frac{1}{4}+\cdots.+\frac{1}{100}$ y luego por $\frac{1}{100}+\cdots+\frac{1}{4}+\frac{1}{1}$

- **Orden ascendente:** 1.55
- **Orden descendente:** 1.55
- **Valor real:** 1.5497677

b) $\sum_{i=0}^{10}\left(\frac{1}{i^3}\right)$ primero por $\frac{1}{1}+\frac{1}{8}+\cdots.+\frac{1}{1000}$ y luego por $\frac{1}{1000}+\frac{1}{729}+\cdots+\frac{1}{1}$

- **Orden ascendente:** 1.19
- **Orden descendente:** 1.2
- **Valor real:** 1.19753

```
# Ejercicio 1: sumas con aritmética de 3 dígitos
def round3(x):
    """Redondea a tres cifras significativas."""
    return float(f"{x:.3g}")
# (a) Suma de 1/i^2
# (b) Suma de 1/i^3
s1, s2 = 0.0, 0.0
# ascendente
for i in range(1, 11):
    s1 = round3(s1 + round3(1 / i**2))
# descendente
for i in range(10, 0, -1):
    s2 = round3(s2 + round3(1 / i**2))
```

2. La serie de Maclaurin para la función arco tangente converge para $-1 < x \leq 1$ y está dada por:

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

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

$$n \geq 2000$$

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

- $n \approx 2 \times 10^{10}\ terminos$, pero el método converge lentamente.

```
# Ejercicio 2: Determinar n mínimo para error < tolerancia
from math import pi
def n_para_tolerancia(tol):
    n = 1
    while 4 / (2 * n + 1) > tol:
        n += 1
    return n
print("Para error < 1e-3: n =", n_para_tolerancia(1e-3))
print("Para error < 1e-10: n =", n_para_tolerancia(1e-10))


print("Ascendente =", s1)
print("Descendente =", s2)
```

3. Otra fórmula para calcular π 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 π dentro de 10^{-3}.

Usando la serie:

$$\arctan(x) = \sum_{k=1}^{n} (-1)^{k+1} \frac{x^{2k-1}}{2k-1}$$

Se cumple $|R_n| \leq \frac{x^{2n+1}}{2n+1}$.

Para π con error $< 10^{-3}$: $n = 3$ es suficiente. Converge mucho más rápido que la serie $\arctan(1)$.

```
# Ejercicio 3: Determinar n mínimo para |error| < 1e-3 usando Machin
def machin_error(n):
    from math import pi
    def arctan_series(x, n):
        return sum(((-1)**(k+1)) * (x**(2*k-1)) / (2*k-1) for k in range(1, n+1))
    pi_aprox = 4 * (4 * arctan_series(1/5, n) - arctan_series(1/239, n))
    return abs(pi - pi_aprox)


for n in range(1, 6):
    print(f"n={n}, error={machin_error(n):.2e}")
```

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.

El correcto algoritmo es del literal a.

5. a. ¿Cuántas multiplicaciones y sumas se requieren para determinar una suma de la forma $\sum_{i=1}^{n}\sum_{j=1}^{i} a_i b_j$?

En el cálculo original, para cada valor de i se realiza una suma de i productos:

S = $a_1 b_2 + a_2(b_1 + b_2) + a_3(b_1 + b_2 + b_3) + \cdots + a_n(b_1 + b_2 + \cdots + b_n)$

Para cada i, multiplicamos (cada $a_i b_j$) y $i - 1$ sumas para agregarlas

Multiplicaciones = $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Sumas = $\frac{n(n+1)}{2} - 1$

El algoritmo directo requiere un número cuadrático de operaciones ($O(n^2)$), es decir, crece rápidamente al aumentar n.

b. Modifique la suma en la parte a) a un formato equivalente que reduzca el número de cálculos.

Podemos reescribir la suma así: **S =** $\sum_{i=1}^{n} a_i B_i$, donde $\boldsymbol{B_i} = \sum_{j=1}^{i} b_j$.

Esto significa que:

- Primero calculamos los valores acumulados B_i de los b_j.
- Luego multiplicamos cada a_i por su correspondiente B_i y los sumamos.

Para calcular todos los B_i:

$B_i = b_i,\ B_i = B_{i-1} + b_i, para\ i = 2, \ldots \ldots n.$ (Se necesita $n - 1$ sumas.)

Para la suma final:

S = $a_1 B_1 + a_2 B_2 + \cdots + a_n B_n$, se necesita n multiplicaciones y $n - 1$ sumas.

Optimizado

Multiplicaciones = n

Sumas = $2n - 2$

En la versión optimizada, se usa la variable acumulativa $\boldsymbol{B_i} = \sum_{j=1}^{i} b_j$, lo que reduce los cálculos a solo n multiplicaciones y $(2n - 2)$ sumas, con crecimiento lineal.

Por tanto, la segunda forma es mucho más eficiente, especialmente para valores grandes de n.

DISCUSIONES

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

```
INPUT: n, x[1..n]
S := 0
FOR i := n DOWNTO 1 DO
    S := S + x[i]
OUTPUT: S
```

2. **Las ecuaciones (1.2) y (1.3) en la sección 1.2 proporcionan formas alternativas para las raíces x_1 y x_2 de $ax^2 + bx + c = 0$. Construya un algoritmo con entrada a, b, c c y salida x_1, x_2 que calcule las raíces x_1 y x_2 (que pueden ser iguales con conjugados complejos) mediante la mejor fórmula para cada raíz.**

```
INPUT: a,b,c
IF a = 0 THEN
    IF b = 0 THEN OUTPUT: "Sin solución o infinitas" ELSE OUTPUT: x = -c/b
ELSE
    D := b*b - 4*a*c
    r := sqrt(D)          # real o compleja
    q := -0.5*(b + sign(b)*r)
    x1 := q/a
    x2 := c/q             # estable numéricamente
    OUTPUT: x1, x2
```

3. **Suponga que**

$$\frac{1-2x}{1-x+x^2} + \frac{2x-4x^3}{1-x^2+x^4} + \cdots = \frac{1+2x}{1+x+x^2}$$

para $x < 1$ y si $x = 0.25$. 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}.

```
INPUT: x, tol
S := 0
k := 1
REPEAT
    num := 2^(k-1)*x^(2^(k-1)-1) - 2^k*x^(2^k - 1)
    den := 1 - x^(2^(k-1)) + x^(2^k)
    S := S + num/den
    k := k + 1
UNTIL abs(S - (1+2*x)/(1+x+x^2)) < tol
OUTPUT: (k-1)   # número de términos
```

```python
# Ejercicio 3: Determinar cuántos términos se requieren para error < 1e-6

def serie(x, tol=1e-6):
    rhs = (1 + 2*x) / (1 + x + x**2)
    S = 0
    k = 1
    while True:
        num = 2**(k-1) * x**(2**(k-1) - 1) - 2**k * x**(2**k - 1)
        den = 1 - x**(2**(k-1)) + x**(2**k)
        S += num / den
        if abs(S - rhs) < tol:
            break
        k += 1
    return k, S, rhs

# Ejemplo
n, S, rhs = serie(0.25)
print(f"Se requieren {n} términos.")
print(f"Suma parcial = {S}, Valor real = {rhs}")
```

```
ERMINAL  PROBLEMS  GITLENS  OUTPUT  DEBUG CONSOLE  PORTS  JUPYTER
 0ms  C:\Universidad_Seb\S-5\MN\Tareas\Tarea_2  master  26
"C:\Program Files\Python313\python.exe" c:/Universidad_Seb/S-5/MN/Tareas/Tarea_2/Ejercicio6.py
e requieren 4 términos.
uma parcial = 1.1428571279559818, Valor real = 1.1428571428571428
 257ms  C:\Universidad_Seb\S-5\MN\Tareas\Tarea_2  master  26
```