Tarea3Lenin_Amangandi

November 5, 2025

Tarea N3

Métodos Númericos

Nombre:Lenin Amangandi

Enlace GitHub Tarea 3

0.1 Pregunta N1

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

a)

$$\sum_{i=1}^{10} \frac{1}{i^2} = \frac{1}{1^2} + \frac{1}{2^2} + \cdots + \frac{1}{10^2} \quad \text{y luego por} \quad \frac{1}{10^2} + \frac{1}{9^2} + \cdots + \frac{1}{1}$$

[48]:
```python
from math import log10, floor

def truncar(x, cifras=3):
    if x == 0:
        return 0
    decimales = -int(floor(log10(abs(x)))) + (cifras - 1)
    factor = 10 ** decimales
    return floor(x * factor) / factor

def sumar_truncada(n, potencia=2, direccion="asc"):
    suma_trunc = 0
    if direccion == "asc":
        indices = range(1, n+1)
        titulo = "ASCENDENTE"
    elif direccion == "desc":
        indices = range(n, 0, -1)
        titulo = "DESCENDENTE"
    else:
        raise ValueError("direccion debe ser 'asc' o 'desc'")

    print(f"\n{titulo} (1/i^{potencia})")
    print(f"{'i':<5} {'1/i^' + str(potencia):<20} {'Suma parcial':<15}")
```

```
    for i in indices:
        termino = 1 / (i**potencia)
        suma_trunc = truncar(suma_trunc + truncar(termino, 3), 3)
        print(f"{i:<5} {round(termino,10):<20.10f} {suma_trunc:<15}")

    print(f"Suma total= {suma_trunc}")
    return suma_trunc
```

[49]:
```
n = 10
potencia_a = 2

suma_asc_a = sumar_truncada(n, potencia_a, "asc")
suma_desc_a = sumar_truncada(n, potencia_a, "desc")

valor_exacto_a = sum([1/(i**potencia_a) for i in range(1, n+1)])
error_asc_a = abs(valor_exacto_a - suma_asc_a)
error_desc_a = abs(valor_exacto_a - suma_desc_a)
error_rel_asc = (error_asc_a / valor_exacto_a) * 100
error_rel_desc = (error_desc_a / valor_exacto_a) * 100

print(f"\nValor exacto: {valor_exacto_a:.10f}")
print(f"Error ascendente: |S_exacto - S_trunc| = |{valor_exacto_a:.10f} -␣
 ↪{suma_asc_a:.10f}| = {error_asc_a:.10f}")
print(f"Error relativo ascendente: {error_asc_a:.10f} / {valor_exacto_a:.10f} *␣
 ↪100 = {error_rel_asc:.5f}%")
print(f"Error descendente: |S_exacto - S_trunc| = |{valor_exacto_a:.10f} -␣
 ↪{suma_desc_a:.10f}| = {error_desc_a:.10f}")
print(f"Error relativo descendente: {error_desc_a:.10f} / {valor_exacto_a:.10f}␣
 ↪* 100 = {error_rel_desc:.5f}%")
print(f"Método más preciso comparando el porcentaje de error: {'DESCENDENTE' if␣
 ↪error_rel_desc < error_rel_asc else 'ASCENDENTE'} "
      f"(ascendente: {error_rel_asc:.5f}%, descendente: {error_rel_desc:.5f}%)")
```

```
ASCENDENTE (1/i^2)
i     1/i^2                Suma parcial
1     1.0000000000         1.0
2     0.2500000000         1.25
3     0.1111111111         1.36
4     0.0625000000         1.42
5     0.0400000000         1.46
6     0.0277777778         1.48
7     0.0204081633         1.5
8     0.0156250000         1.51
9     0.0123456790         1.52
10    0.0100000000         1.53
Suma total= 1.53
```

```
DESCENDENTE (1/i^2)
i     1/i^2                Suma parcial
10    0.0100000000         0.01
9     0.0123456790         0.0223
8     0.0156250000         0.0379
7     0.0204081633         0.0583
6     0.0277777778         0.0859
5     0.0400000000         0.125
4     0.0625000000         0.187
3     0.1111111111         0.298
2     0.2500000000         0.548
1     1.0000000000         1.54
Suma total= 1.54

Valor exacto: 1.5497677312
Error ascendente: |S_exacto - S_trunc| = |1.5497677312 - 1.5300000000| =
0.0197677312
Error relativo ascendente: 0.0197677312 / 1.5497677312 * 100 = 1.27553%
Error descendente: |S_exacto - S_trunc| = |1.5497677312 - 1.5400000000| =
0.0097677312
Error relativo descendente: 0.0097677312 / 1.5497677312 * 100 = 0.63027%
Método más preciso comparando el porcentaje de error: DESCENDENTE (ascendente:
1.27553%, descendente: 0.63027%)
```

b)

$$\sum_{i=1}^{10} \frac{1}{i^3} = \frac{1}{1^3} + \frac{1}{2^3} + \cdots + \frac{1}{10^3} \quad \text{y luego por} \quad \frac{1}{10^3} + \frac{1}{9^3} + \cdots + \frac{1}{1}$$

[50]:
```python
potencia_b = 3

suma_asc_b = sumar_truncada(n, potencia_b, "asc")
suma_desc_b = sumar_truncada(n, potencia_b, "desc")

valor_exacto_b = sum([1/(i**potencia_b) for i in range(1, n+1)])
error_asc_b = abs(valor_exacto_b - suma_asc_b)
error_desc_b = abs(valor_exacto_b - suma_desc_b)
error_rel_asc_b = (error_asc_b / valor_exacto_b) * 100
error_rel_desc_b = (error_desc_b / valor_exacto_b) * 100

print(f"\nValor exacto: {valor_exacto_b:.10f}")
print(f"Error ascendente: |S_exacto - S_trunc| = |{valor_exacto_b:.10f} -␣
 ↪{suma_asc_b:.10f}| = {error_asc_b:.10f}")
print(f"Error relativo ascendente: {error_asc_b:.10f} / {valor_exacto_b:.10f} *␣
 ↪100 = {error_rel_asc_b:.5f}%")
print(f"Error descendente: |S_exacto - S_trunc| = |{valor_exacto_b:.10f} -␣
 ↪{suma_desc_b:.10f}| = {error_desc_b:.10f}")
```

```
print(f"Error relativo descendente: {error_desc_b:.10f} / {valor_exacto_b:.10f}
 ↪* 100 = {error_rel_desc_b:.5f}%")
print(f"Método más preciso comparando el porcentaje de error: {'DESCENDENTE' if
 ↪error_rel_desc_b < error_rel_asc_b else 'ASCENDENTE'} "
      f"(ascendente: {error_rel_asc_b:.5f}%, descendente: {error_rel_desc_b:.
 ↪5f}%)")
```

```
ASCENDENTE (1/i^3)
i     1/i^3                Suma parcial
1     1.0000000000         1.0
2     0.1250000000         1.12
3     0.0370370370         1.15
4     0.0156250000         1.16
5     0.0080000000         1.16
6     0.0046296296         1.16
7     0.0029154519         1.16
8     0.0019531250         1.16
9     0.0013717421         1.16
10    0.0010000000         1.16
Suma total= 1.16

DESCENDENTE (1/i^3)
i     1/i^3                Suma parcial
10    0.0010000000         0.001
9     0.0013717421         0.00236
8     0.0019531250         0.0043
7     0.0029154519         0.0072
6     0.0046296296         0.0118
5     0.0080000000         0.0197
4     0.0156250000         0.0353
3     0.0370370370         0.0723
2     0.1250000000         0.197
1     1.0000000000         1.19
Suma total= 1.19

Valor exacto: 1.1975319857
Error ascendente: |S_exacto - S_trunc| = |1.1975319857 - 1.1600000000| =
0.0375319857
Error relativo ascendente: 0.0375319857 / 1.1975319857 * 100 = 3.13411%
Error descendente: |S_exacto - S_trunc| = |1.1975319857 - 1.1900000000| =
0.0075319857
Error relativo descendente: 0.0075319857 / 1.1975319857 * 100 = 0.62896%
Método más preciso comparando el porcentaje de error: DESCENDENTE (ascendente:
3.13411%, descendente: 0.62896%)
```

0.2 Pregunta N2

La serie de Maclaurin para la función arcotangente converge para $-1 < x \leq 1$ y está dada por

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

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

[77]:
```
import math
tolerancia_a = 1e-3

print("\nPaso 1: Planteamos la desigualdad de la serie alternante:")
print("|4P_n(1) -  | <= 4/(2n+1) < tolerancia")
print(f"Tolerancia: {tolerancia_a}")

denominador = 4 / tolerancia_a
n_min = (denominador - 1) / 2
n_requerido = int(n_min) + 1

print("\nPaso 2: Despejamos n")
print(f"2n + 1 > 4 / {tolerancia_a} = {denominador}")
print(f"n > ({denominador} - 1)/2 = {n_min}")
print(f"Número mínimo de términos a usar: n = {n_requerido}")

cota_error = 4 / (2 * n_requerido + 1)
print("\nPaso 3: Verificación de la cota de error")
print(f"Cota de error con n={n_requerido}: {cota_error:.6f}")

pi_aprox = 4 * sum([(-1)**(i+1)/(2*i-1) for i in range(1, n_requerido+1)])
error_abs = abs(pi_aprox - math.pi)
error_rel = (error_abs / math.pi) * 100

print("\nPaso 4: Aproximación de  y cálculo de errores")
print(f"Aproximación de  : {pi_aprox:.10f}")
print(f"Error absoluto: | _real -  _aprox| = {error_abs:.10f}")
print(f"Error relativo: ({error_abs:.10f} / {math.pi:.10f}) * 100 = {error_rel:.
↪5f}%")
```

```
Paso 1: Planteamos la desigualdad de la serie alternante:
|4P_n(1) -  | <= 4/(2n+1) < tolerancia
Tolerancia: 0.001

Paso 2: Despejamos n
2n + 1 > 4 / 0.001 = 4000.0
n > (4000.0 - 1)/2 = 1999.5
```

```
Número mínimo de términos a usar: n = 2000

Paso 3: Verificación de la cota de error
Cota de error con n=2000: 0.001000

Paso 4: Aproximación de  y cálculo de errores
Aproximación de  : 3.1410926536
Error absoluto: | _real -  _aprox| = 0.0005000000
Error relativo: (0.0005000000 / 3.1415926536) * 100 = 0.01592%
```

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?

[78]:
```python
tolerancia_b = 1e-10

print("\nPaso 1: Planteamos la desigualdad de la cota de error:")
print("|4P_n(1) -  | <= 4/(2n+1) < tolerancia")
print(f"Tolerancia: {tolerancia_b}")

denominador_b = 4 / tolerancia_b
n_min_b = (denominador_b - 1) / 2
n_requerido_b = int(n_min_b) + 1

print("\nPaso 2: Despejamos n")
print(f"2n + 1 > 4 / {tolerancia_b} = {denominador_b:.2e}")
print(f"n > ({denominador_b:.2e} - 1)/2 = {n_min_b:.2e}")
print(f"Número mínimo de términos a usar: n  {n_requerido_b:,}")
```

```
Paso 1: Planteamos la desigualdad de la cota de error:
|4P_n(1) -  | <= 4/(2n+1) < tolerancia
Tolerancia: 1e-10

Paso 2: Despejamos n
2n + 1 > 4 / 1e-10 = 4.00e+10
n > (4.00e+10 - 1)/2 = 2.00e+10
Número mínimo de términos a usar: n  20,000,000,000
```

0.3 Pregunta N3

Otra fórmula para calcular π se puede deducir a partir de la identidad:

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

Determine el número de términos que se deben sumar para garantizar una aproximación de π dentro de 10^{-3}.

[67]:
```python
import math

def arctan_taylor_iter(x, tol=1e-3):
    suma = 0
    n = 0
    while True:
        termino = (-1)**n * (x**(2*n+1)) / (2*n+1)
        suma += termino
        print(f"n={n}, término={termino:.10f}, suma parcial={suma:.10f}")
        if abs(termino) < tol:
            break
        n += 1
    return suma, n+1

tol = 1e-3

print("Calculando arctan(1/5):")
suma_1, term_1 = arctan_taylor_iter(1/5, tol)

print("\nCalculando arctan(1/239):")
suma_2, term_2 = arctan_taylor_iter(1/239, tol)

pi_aprox = 4*(4*suma_1 - suma_2)
pi_real = math.pi

error_abs = abs(pi_real - pi_aprox)
error_rel = (error_abs / pi_real) * 100

print(f"\nAproximación de  : {pi_aprox:.10f}")
print(f"Valor real de  : {pi_real:.10f}")
print(f"Error absoluto: | _real -  _aprox| = |{pi_real:.10f} - {pi_aprox:.10f}|␣
 ↪= {error_abs:.10f}")
print(f"Error relativo: {error_abs:.10f} / {pi_real:.10f} * 100 = {error_rel:.
 ↪5f}%")
print(f"Términos usados: arctan(1/5)={term_1}, arctan(1/239)={term_2}")
```

```
Calculando arctan(1/5):
n=0, término=0.2000000000, suma parcial=0.2000000000
n=1, término=-0.0026666667, suma parcial=0.1973333333
n=2, término=0.0000640000, suma parcial=0.1973973333

Calculando arctan(1/239):
n=0, término=0.0041841004, suma parcial=0.0041841004
n=1, término=-0.0000000244, suma parcial=0.0041840760

Aproximación de  : 3.1416210293
Valor real de  : 3.1415926536
Error absoluto: | _real -  _aprox| = |3.1415926536 - 3.1416210293| =
```

```
0.0000283757
Error relativo: 0.0000283757 / 3.1415926536 * 100 = 0.00090%
Términos usados: arctan(1/5)=3, arctan(1/239)=2
```

0.4 Pregunta N4

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

4a.

ENTRADA n, x , x , , x .

SALIDA PRODUCT.

Paso 1 Determine PRODUCT = 0.

Paso 2 Para i = 1, 2, , n haga

Determine PRODUCT = PRODUCT × x .

Paso 3 SALIDA PRODUCT;

PARE.

4b.

ENTRADA n, x , x , , x .

SALIDA PRODUCT.

Paso 1 Determine PRODUCT = 1.

Paso 2 Para i = 1, 2, , n haga

Determine PRODUCT = PRODUCT × x .

Paso 3 SALIDA PRODUCT;

PARE.

4c.

ENTRADA n, x , x , , x .

SALIDA PRODUCT.

Paso 1 Determine PRODUCT = 1.

Paso 2 Para i = 1, 2, , n haga

si x = 0 entonces determine PRODUCT = 0;

SALIDA PRODUCT;

PARE

Determine PRODUCT = PRODUCT × x .

Paso 3 SALIDA PRODUCT;

PARE.

El algoritmo 4a no es válido, ya que comienza el producto con un valor inicial de 0. Dado que cualquier número multiplicado por 0 da como resultado 0, el algoritmo siempre producirá ese valor, sin importar los datos de entrada. Por lo tanto, este método falla en todos los casos, excepto cuando todos los números multiplicados son cero.

Los algoritmos 4b y 4c sí son correctos, ya que inician el producto con 1 y realizan las multiplicaciones de manera adecuada. Además, el algoritmo 4c incorpora una mejora: detecta la presencia de un cero en los datos y detiene el cálculo inmediatamente, evitando operaciones innecesarias.

0.5 Pregunta N5

a. ¿Cuántas multiplicaciones y sumas se requieren para determinar una suma de la forma

$$\sum_{i=1}^{n} a_i b_i$$

?

1. Fórmula general

$$S = \sum_{i=1}^{n} \sum_{j=1}^{i} a_i b_j$$

Cada término (a_i) se multiplica por todos los (b_j) con (j i).

2. Ejemplo con n = 3

$$S = a_1 b_1 + a_2(b_1 + b_2) + a_3(b_1 + b_2 + b_3)$$

3. Conteo de operaciones

Para cada (i):

- hay (i) multiplicaciones y (i) sumas.

$$\text{Multiplicaciones totales} = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$$

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

$$\text{Operaciones totales} = n(n+1)$$

4. Ejemplo numérico

Con (a = [2, 3, 4]) y (b = [1, 2, 3]):

$$S = (2 \times 1) + (3 \times 1 + 3 \times 2) + (4 \times 1 + 4 \times 2 + 4 \times 3) = 2 + 9 + 24 = 35$$

[81]:
```python
def suma_doble(a, b):

    total = 0
    multiplicaciones = 0
    sumas = 0

    print("Iteraciones:")

    for i in range(len(a)):
        for j in range(i + 1):
            producto = a[i] * b[j]
            multiplicaciones += 1

            anterior = total
            total += producto
            sumas += 1

            print(f"i={i+1}, j={j+1} → {a[i]}×{b[j]} = {producto:>4} | "
                  f"Total: {anterior} + {producto} = {total}")

    print(f"Multiplicaciones totales: {multiplicaciones}")
    print(f"Sumas totales: {sumas}")
    print(f"Operaciones totales: {multiplicaciones + sumas}")
    print(f"Resultado final: {total}")

    return total
a = [2, 3, 4]
b = [1, 2, 3]

suma_doble(a, b)
```

```
Iteraciones:
i=1, j=1 → 2×1 =    2 | Total: 0 + 2 = 2
i=2, j=1 → 3×1 =    3 | Total: 2 + 3 = 5
i=2, j=2 → 3×2 =    6 | Total: 5 + 6 = 11
i=3, j=1 → 4×1 =    4 | Total: 11 + 4 = 15
i=3, j=2 → 4×2 =    8 | Total: 15 + 8 = 23
i=3, j=3 → 4×3 =   12 | Total: 23 + 12 = 35
Multiplicaciones totales: 6
Sumas totales: 6
Operaciones totales: 12
Resultado final: 35
```

[81]: 35

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

La suma doble original

$$S = \sum_{i=1}^{n} \sum_{j=1}^{i} a_i b_j$$

puede simplificarse usando la propiedad distributiva de la suma, ya que (a_i) no depende de (j). Así se obtiene

$$S = \sum_{i=1}^{n} a_i \Big(\sum_{j=1}^{i} b_j \Big).$$

Luego, definiendo una suma acumulada (B_i = _{j=1}^{i} b_j), la expresión se reduce a

$$S = \sum_{i=1}^{n} a_i B_i,$$

lo que evita recalcular la suma interna cada vez y disminuye las operaciones de (O(n^2)) a (O(n)).

Por ejemplo, si (a = [2, 3, 4]) y (b = [1, 2, 3]), primero calculamos las sumas acumuladas (B_i): (B_1 = 1,; B_2 = 1+2=3,; B_3 = 1+2+3=6.)
Luego aplicamos la fórmula

$$S = \sum_{i=1}^{3} a_i B_i = 2(1) + 3(3) + 4(6) = 35.$$

Así se obtiene el mismo resultado que con la suma doble, pero con menos operaciones.

0.6 DISCUSIONES

1. Sumar la serie finita $\sum_{i=1}^{n} x_i$ en orden inverso (de x_n a x_1).

[1]:
```
def sumar_serie_inversa(X):
    suma = 0
    for i in range(len(X)-1, -1, -1):
        suma += X[i]
    return suma

X = [1, 2, 3, 4, 5]
resultado = sumar_serie_inversa(X)
print("La suma inversa es:", resultado)
```

```
La suma inversa es: 15
```

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.

[2]:
```
import cmath

def resolver_cuadratica(a, b, c):
    if a == 0:
        raise ValueError("El coeficiente 'a' no puede ser cero")
```

```
    delta = cmath.sqrt(b**2 - 4*a*c)
    if b >= 0:
        x1 = (-b - delta) / (2*a)
    else:
        x1 = (-b + delta) / (2*a)

    x2 = c / (a * x1)

    return x1, x2

a, b, c = 1, -3, 2
x1, x2 = resolver_cuadratica(a, b, c)
print("Raíces:", x1, x2)

a, b, c = 1, 2, 5
x1, x2 = resolver_cuadratica(a, b, c)
print("Raíces complejas:", x1, x2)
```

```
Raíces: (2+0j) (1+0j)
Raíces complejas: (-1-2j) (-1+2j)
```

3. **3.** Suponga que

$$\frac{1-2x}{1+x^2+x^4}+\frac{2x-4x^3}{1+x^4+x^8}+\frac{4x^3-8x^7}{1+x^8+x^{16}}+\cdots=\frac{1+2x}{1+x+x^2}$$

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}.

[12]:
```
x = 0.25
diferencia = 1e-6
right_side = (1 + 2 * x) / (1 + x + x ** 2)

sum_left = 0
n = 0
while True:
    num = (2**n) * (x**(2**n - 1)) - (2**(n + 1)) * (x**(2**(n + 1) - 1))
    den = 1 - x**(2**n) + x**(2**(n + 1))
    term = num / den
    sum_left += term

    if abs(sum_left - right_side) < diferencia:
        break

    n += 1
```

```
print(f"Se necesitan {n + 1} términos para que la suma difiera del lado derecho␣
  ↪en menos de {diferencia:.0e}")
```

```
Se necesitan 4 términos para que la suma difiera del lado derecho en menos de
1e-06
```