tarea-03-ejercicios-unidad-01-b

May 19, 2025

ESCUELA POLITÉCNICA NACIONAL FACULTAD DE INGENIERÍA DE SISTEMAS MÉTODOS NUMÉRICOS

Conjunto de Ejercicios 1.3

Richard Tipantiza 2025-05-11

0.1 CONJUNTO DE EJERCICIOS 1.3

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. $\sum_{i=1}^{10} \frac{1}{i^2}$ primero por: $\frac{1}{1} + \frac{1}{4} + \cdots + \frac{1}{100}$ y luego por: $\frac{1}{100} + \frac{1}{81} + \cdots + \frac{1}{1}$
   b. $\sum_{i=1}^{10} \frac{1}{i^3}$ primero por: $\frac{1}{1} + \frac{1}{8} + \frac{1}{27} + \cdots + \frac{1}{1000}$ y luego por: $\frac{1}{1000} + \frac{1}{729} + \cdots + \frac{1}{1}$

- Pseudocódigo:

INICIO
Para el literal a y b:
1. Definir la secuencia de términos en orden ascendente y descendente
2. Implementar función de suma con aritmética de 3 dígitos
3. Calcular las sumas
4. Comparar con el valor exacto
5. Determinar qué método es más preciso
FIN

[34]:
```python
import numpy as np

# Implementación en Python
def tres_digitos(x):
    """Redondea a 3 dígitos significativos"""
    if x == 0:
        return 0
    # Calculamos la potencia de 10 para mantener 3 dígitos significativos
    potencia = 3 - 1 - int(np.floor(np.log10(abs(x))))
    return round(x, potencia)

def suma_ascendente(terminos):
    """Suma los términos en orden ascendente (del más pequeño al más grande)"""
    suma = 0
```

```
    # Ordenamos los términos en orden ascendente
    terminos_ordenados = sorted(terminos)
    for term in terminos_ordenados:
        suma = tres_digitos(suma + tres_digitos(term))
    return suma

def suma_descendente(terminos):
    """Suma los términos en orden descendente (del más grande al más pequeño)"""
    suma = 0
    # Ordenamos los términos en orden descendente
    terminos_ordenados = sorted(terminos, reverse=True)
    for term in terminos_ordenados:
        suma = tres_digitos(suma + tres_digitos(term))
    return suma

# Parte a: Σ(1/i²) para i=1 a 10
terminos_a = [1/(i**2) for i in range(1, 11)]
suma_asc_a = suma_ascendente(terminos_a)
suma_desc_a = suma_descendente(terminos_a)
exacto_a = sum(1/(i**2) for i in range(1, 11))

# Parte b: Σ(1/i³) para i=1 a 10
terminos_b = [1/(i**3) for i in range(1, 11)]
suma_asc_b = suma_ascendente(terminos_b)
suma_desc_b = suma_descendente(terminos_b)
exacto_b = sum(1/(i**3) for i in range(1, 11))

# Resultados
print("Parte a (Σ 1/i²):")
print(f"Ascendente (menor a mayor): {suma_asc_a}")
print(f"Descendente (mayor a menor): {suma_desc_a}")
print(f"Valor exacto: {exacto_a}")
print(f"Error ascendente: {abs(suma_asc_a - exacto_a):.2e}")
print(f"Error descendente: {abs(suma_desc_a - exacto_a):.2e}")

print("\nParte b (Σ 1/i³):")
print(f"Ascendente (menor a mayor): {suma_asc_b}")
print(f"Descendente (mayor a menor): {suma_desc_b}")
print(f"Valor exacto: {exacto_b}")
print(f"Error ascendente: {abs(suma_asc_b - exacto_b):.2e}")
print(f"Error descendente: {abs(suma_desc_b - exacto_b):.2e}")

# Análisis adicional: mostrar los términos para entender mejor
print("\nAnálisis de términos para 1/i²:")
print("Términos ordenados (ascendente):", [f"{x:.6f}" for x in␣
 ↪sorted(terminos_a)])
```

```
print("Términos ordenados (descendente):", [f"{x:.6f}" for x in␣
 ↪sorted(terminos_a, reverse=True)])

print("\nAnálisis de términos para 1/i³:")
print("Términos ordenados (ascendente):", [f"{x:.6f}" for x in␣
 ↪sorted(terminos_b)])
print("Términos ordenados (descendente):", [f"{x:.6f}" for x in␣
 ↪sorted(terminos_b, reverse=True)])
```

```
Parte a (Σ 1/i²):
Ascendente (menor a mayor): 1.55
Descendente (mayor a menor): 1.55
Valor exacto: 1.5497677311665408
Error ascendente: 2.32e-04
Error descendente: 2.32e-04

Parte b (Σ 1/i³):
Ascendente (menor a mayor): 1.2
Descendente (mayor a menor): 1.19
Valor exacto: 1.1975319856741933
Error ascendente: 2.47e-03
Error descendente: 7.53e-03

Análisis de términos para 1/i²:
Términos ordenados (ascendente): ['0.010000', '0.012346', '0.015625',
'0.020408', '0.027778', '0.040000', '0.062500', '0.111111', '0.250000',
'1.000000']
Términos ordenados (descendente): ['1.000000', '0.250000', '0.111111',
'0.062500', '0.040000', '0.027778', '0.020408', '0.015625', '0.012346',
'0.010000']

Análisis de términos para 1/i³:
Términos ordenados (ascendente): ['0.001000', '0.001372', '0.001953',
'0.002915', '0.004630', '0.008000', '0.015625', '0.037037', '0.125000',
'1.000000']
Términos ordenados (descendente): ['1.000000', '0.125000', '0.037037',
'0.015625', '0.008000', '0.004630', '0.002915', '0.001953', '0.001372',
'0.001000']
```

En ambos casos, el método de sumar en orden descendente es más preciso porque minimiza el error por truncamiento al sumar primero los términos más pequeños.

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

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

- Pseudocódigo:

INICIO
Parte a:
1. Definir función P_n(x) que calcula la suma de la serie hasta n términos
2. Inicializar n = 1
3. Mientras $|4 * P_n(1) - \pi| \geq 10^{-3}$
- Incrementar n
- Calcular nuevo $P_{n(1)}$
4. Devolver n
Parte b:
1. El mismo proceso pero con tolerancia 10^{-10}
2. Encontrar n donde $|4 * P_n(1) - \pi| < 10^{-10}$
FIN

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 $|4P_n(1) - \pi| < 10^{-3}$

[35]:
```python
# Implementación en Python para literal a
import math

def calcular_terminos_a():
    """Calcula el número de términos para precisión 10^-3"""
    def P_n(n):
        return sum((-1)**(i+1) / (2*i-1) for i in range(1, n+1))

    n = 1
    while True:
        pi_approx = 4 * P_n(n)
        error = abs(pi_approx - math.pi)
        if error < 1e-3:
            return n
        n += 1
        if n > 10000:  # Límite seguro para esta precisión
            break
    return n

# Calcular y mostrar resultados
n_a = calcular_terminos_a()
pi_approx_a = 4 * sum((-1)**(i+1) / (2*i-1) for i in range(1, n_a+1))
error_a = abs(pi_approx_a - math.pi)

print("Literal a:")
print(f"Número de términos requeridos: {n_a}")
print(f"Aproximación de  : {pi_approx_a}")
print(f"Error absoluto: {error_a}")
print(f"¿Cumple con 10^-3? {'Sí' if error_a < 1e-3 else 'No'}")
```

```
Literal a:
```

```
Número de términos requeridos: 1000
Aproximación de  : 3.1405926538397932
Error absoluto: 0.0009999997499998692
¿Cumple con 10^-3? Sí
```

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?

[48]:
```python
def expresion(n):
    return (-1)**(n + 1) / (2*n - 1)

def error_Absoluto(valor_aproximado):
    import math
    return abs(math.pi - valor_aproximado)

# Implementación en Python para literal b
def sumatoriaConLimitacion(error_maximo):
    sumatoria = 0
    contador = 1
    mensaje = ""

    while True:  # Ejecutar hasta alcanzar el error deseado
        sumatoria += expresion(contador)
        if error_Absoluto(4 * sumatoria) < error_maximo:  # Evaluar error sobre␣
 ↪el valor de pi
            mensaje = f"Número de términos necesarios: {contador}"
            break
        contador += 1
        if contador >= 1000000:
            mensaje = f"Número de iteraciones máximo alcanzado: {contador}"
            break

    return sumatoria, mensaje

# Parámetro de error para literal b
error_maxi = 1e-10
suma_aproxi, num_termi = sumatoriaConLimitacion(error_maxi)
error_b = error_Absoluto(4 * suma_aproxi)

print("\nLiteral b:")
print(num_termi)
print(f"Valor aproximado de  : {4 * suma_aproxi}")
print(f"Error absoluto: {error_b}")
print(f"¿Cumple con 10^-10? {'Sí' if error_b < 1e-10 else 'No'}")
```

```
Literal b:
Número de iteraciones máximo alcanzado: 1000000
```

```
Valor aproximado de  : 3.1415936535907742
Error absoluto: 1.0000009811328425e-06
¿Cumple con 10^-10? No
```

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

-Pseudocódigo

INICIO
1. Definir función arctan_series(x, n) que calcula $arctan(x)$ usando n términos
2. Definir función calcular_pi(n1, n2) usando la fórmula dada
3. Inicializar n = 1
4. Mientras $error > 10^{-23}$:
- Calcular π aproximado con n términos
- Calcular error absoluto
- Incrementar n
5. Devolver n requerido
FIN

[37]:
```python
import math

def arctan_series(x, n_terms):
    """Calcula arctan(x) usando series de Maclaurin con n términos"""
    return sum((-1)**k * x**(2*k + 1) / (2*k + 1) for k in range(n_terms))

def calcular_pi(n_terms):
    """Calcula  usando la fórmula dada con n términos en cada serie"""
    term1 = 4 * arctan_series(1/5, n_terms)
    term2 = arctan_series(1/239, n_terms)
    return 4 * (term1 - term2)

def encontrar_terminos_requeridos():
    """Determina el número de términos necesario para precisión 10^-23"""
    n = 1
    while True:
        pi_approx = calcular_pi(n)
        error = abs(pi_approx - math.pi)
        if error < 1e-23:
            return n
        n += 1
        # Prevención para no exceder límites computacionales
        if n > 1000:
            print("Advertencia: Convergencia no alcanzada en 1000 términos")
            return n

# Calcular términos requeridos
n_requerido = encontrar_terminos_requeridos()
```

```
pi_aproximado = calcular_pi(n_requerido)
error = abs(pi_aproximado - math.pi)

# Resultados
print(f"Número de términos requeridos: {n_requerido}")
print(f"Aproximación de  : {pi_aproximado}")
print(f"Error absoluto: {error}")
print(f"¿Cumple con 10^-23? {'Sí' if error < 1e-23 else 'No'}")
```

```
Advertencia: Convergencia no alcanzada en 1000 términos
Número de términos requeridos: 1001
Aproximación de  : 3.1415926535897936
Error absoluto: 4.440892098500626e-16
¿Cumple con 10^-23? No
```

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

a. Producto iniciando en 0

Entrada $ n, x_1, x_2, ..., x_n $
Salida PRODUCT
Paso 1 Establecer PRODUCT = 0
Paso 2 Para $ i = 1, 2, ..., n $, hacer:
PRODUCT = PRODUCT * x_i
Paso 3 Salida: PRODUCT
Parar

[38]:
```
def algoritmo_a(x):
    product = 0
    for num in x:
        product = product * num
    return product
```

b. Producto iniciando en 1

Entrada $ n, x_1, x_2, ..., x_n $
Salida PRODUCT
Paso 1 Establecer PRODUCT = 1
Paso 2 Para $ i = 1, 2, ..., n $, hacer:
PRODUCT = PRODUCT * x_i
Paso 3 Salida: PRODUCT
Parar

[39]:
```
def algoritmo_b(x):
    product = 1
    for num in x:
        product = product * num
    return product
```

c. Producto con verificación de ceros

Entrada $ n, x_1, x_2, ..., x_n $
Salida PRODUCT
Paso 1 Establecer PRODUCT = 1
Paso 2 Para $ i = 1, 2, ..., n $, hacer:
Si $x_i = 0$, entonces:
Establecer PRODUCT = 0
Salida: PRODUCT
Parar
PRODUCT = PRODUCT * x_i
Paso 3 Salida: PRODUCT
Parar

[40]:
```python
def algoritmo_c(x):
    product = 1
    for num in x:
        if num == 0:
            return 0
        product = product * num
    return product
```

[41]:
```python
# Casos de prueba
test_cases = [
    ([1, 2, 3, 4], 24),
    ([2, 2, 2], 8),
    ([5, 0, 3], 0),
    ([], 1),          # Producto vacío
    ([0.5, 2], 1.0),
    ([-1, 1, -1], -1)
]

# Evaluación de los algoritmos
print("Comparación de algoritmos:\n")
print("{:<15} {:<15} {:<15} {:<15}".format(
    "Caso de prueba", "Algoritmo A", "Algoritmo B", "Algoritmo C"))

for nums, expected in test_cases:
    a = algoritmo_a(nums)
    b = algoritmo_b(nums)
    c = algoritmo_c(nums)

    print("{:<15} {:<15} {:<15} {:<15}".format(
        str(nums),
        str(a) + ("  " if a == expected else "  "),
        str(b) + ("  " if b == expected else "  "),
        str(c) + ("  " if c == expected else "  ")))
```

```
Comparación de algoritmos:
```

```
Caso de prueba  Algoritmo A     Algoritmo B     Algoritmo C
[1, 2, 3, 4]    0               24              24
[2, 2, 2]       0               8               8
[5, 0, 3]       0               0               0
[]              0               1               1
[0.5, 2]        0.0             1.0             1.0
[-1, 1, -1]     0               1               1
```

- Para cualquier caso general, los algoritmos B y C son correctos.
- El algoritmo C es mejor en casos con ceros.
- El algoritmo B es más simple para listas pequeñas.

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

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

- Pseudocódigo

INICIO
1. Para cada par (i, j) se realiza 1 multiplicación: $a_i \cdot b_j$
2. Se requieren $(n \cdot l - 1)$ sumas para agregar todos los términos
3. Total: $n \cdot l$ multiplicaciones + $(n \cdot l - 1)$ sumas
FIN

[42]:
```python
def operaciones_original(n, l):
    mult = n * l
    sumas = n * l - 1
    return mult, sumas

n, l = 5, 3
mult, sumas = operaciones_original(n, l)
print(f"Para n={n}, l={l}:")
print(f"Multiplicaciones: {mult}")
print(f"Sumas: {sumas}")
print(f"Total operaciones: {mult + sumas}")
```

```
Para n=5, l=3:
Multiplicaciones: 15
Sumas: 14
Total operaciones: 29
```

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

INICIO
1. Calcular suma_A = $\sum a_i$ con $(n - 1)$ sumas
2. Calcular suma_B = $\sum b_j$ con $(l - 1)$ sumas
3. Multiplicar suma_A $\cdot$ suma_B (1 operación)
4. Total: 1 multiplicación + $(n + l - 2)$ sumas
FIN

[43]:
```python
def operaciones_optimizadas(n, l):
    sumas = (n - 1) + (l - 1)
    mult = 1
    return mult, sumas

n, l = 5, 3
mult, sumas = operaciones_optimizadas(n, l)
print(f"Para n={n}, l={l}:")
print(f"Multiplicaciones: {mult}")
print(f"Sumas: {sumas}")
print(f"Total operaciones: {mult + sumas}")
```

```
Para n=5, l=3:
Multiplicaciones: 1
Sumas: 6
Total operaciones: 7
```

0.2 DISCUSIONES

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

- Pseudocódigo

INICIO
1. Leer lista de números $x = [x_1, x_2, \dots, x_n]$
2. Inicializar suma $= 0$
3. Para cada elemento en orden inverso (de x_n a x_1):
a. Sumar elemento al acumulador
4. Devolver resultado
FIN

[44]:
```python
def suma_inversa(x):
    suma = 0
    for elemento in reversed(x):
        suma += elemento
    return suma

datos = [1, 2, 3, 4, 5]
resultado = suma_inversa(datos)
print(f"Lista original: {datos}")
print(f"Suma en orden inverso: {resultado}")
```

```
Lista original: [1, 2, 3, 4, 5]
Suma en orden inverso: 15
```

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 $ y salida $ x_1, x_2 $ que calcule las raíces $ x_1 $ y $ x_2 $ (que pueden ser iguales o conjugados complejos) mediante la mejor fórmula para cada raíz.

[45]:
```python
import cmath
import math

def raices_cuadraticas(a, b, c):
    """Calcula las raíces de ax² + bx + c = 0 con máxima precisión"""
    if a == 0:
        raise ValueError("El coeficiente 'a' no puede ser cero")

    D = b**2 - 4*a*c
    sqrt_D = cmath.sqrt(D) if D < 0 else math.sqrt(D)

    # Evitar resta catastrófica
    if b >= 0:
        x1 = (-b - sqrt_D)/(2*a)
    else:
        x1 = (-b + sqrt_D)/(2*a)

    # Usar relación de Vieta para la segunda raíz
    x2 = (c/a)/x1 if x1 != 0 else (-b)/(a)

    return x1, x2

casos_prueba = [
    (1, 5, 6),    # Raíces reales distintas
    (1, -4, 4),   # Raíz real doble
    (1, 0, 1),    # Raíces complejas
    (1, 1e10, 1), # Caso de resta catastrófica
    (1, -1e5, 1)  # Otro caso delicado
]

print("Resultados para diferentes casos:")
print("{:<30} {:<30} {:<30}".format("Coeficientes", "Raíz x ", "Raíz x "))
for a, b, c in casos_prueba:
    x1, x2 = raices_cuadraticas(a, b, c)
    print("{:<30} {:<30} {:<30}".format(
        f"a={a}, b={b}, c={c}",
        f"{x1:.15g}",
        f"{x2:.15g}"))
```

```
Resultados para diferentes casos:
Coeficientes                   Raíz x                         Raíz x
a=1, b=5, c=6                  -3                             -2
a=1, b=-4, c=4                 2                              2
a=1, b=0, c=1                  0-1j                           -0+1j
a=1, b=10000000000.0, c=1      -10000000000                   -1e-10
a=1, b=-100000.0, c=1          99999.99999                    1.0000000001e-05
```

3. Suponga que:

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

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} $.

[46]:
```python
def termino_serie(n, x):
    """Calcula el n-ésimo término de la serie izquierda"""
    exponente = 2**(n-1)
    numerador = (2**(n-1) * x**(exponente - 1)) - (2**n * x**(2*exponente - 1))
    denominador = 1 - x**exponente + x**(2*exponente)
    return numerador / denominador

def valor_derecho(x):
    """Calcula el valor del lado derecho de la ecuación"""
    return (1 + 2*x) / (1 + x + x**2)

def calcular_terminos_necesarios(x, tolerancia):
    """Determina cuántos términos se necesitan para alcanzar la precisión␣
 ↪deseada"""
    suma = 0.0
    n = 1
    objetivo = valor_derecho(x)

    while True:
        suma += termino_serie(n, x)
        if abs(suma - objetivo) < tolerancia:
            return n
        n += 1
        # Prevención para evitar bucles infinitos
        if n > 100:
            raise ValueError("No se alcanzó la convergencia en 100 términos")
```

[47]:
```python
x = 0.25
error_limite = 1e-6

# Cálculo de resultados
n_final = calcular_terminos_necesarios(x, error_limite)
suma_final = sum(termino_serie(i, x) for i in range(1, n_final+1))
valor_exacto = valor_derecho(x)

# Presentación de resultados
print(f"Número de términos necesarios: {n_final}")
print(f"Valor de la serie: {suma_final:.8f}")
print(f"Valor exacto: {valor_exacto:.8f}")
print(f"Diferencia: {abs(suma_final - valor_exacto):.2e}")
```

```
Número de términos necesarios: 4
Valor de la serie: 1.14285713
Valor exacto: 1.14285714
Diferencia: 1.49e-08
```