Tarea 03: Ejercicios Unidad 01-B

Métodos Numéricos

Christopher Criollo

2025-11-06

Tabla de Contenidos

1 CONJUNTO DE EJERCICIOS

1.1 Ejercicio de Suma y Aritmética de Corte

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.

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

```python
def three_digit_chop(x):
    if x == 0:
        return 0

    # Para números muy pequeños
    if abs(x) < 0.001:
        s = f"{abs(x):.10f}"
        for i, char in enumerate(s):
            if char not in ['0', '.']:
                significant = s[i:i+3]
                result = float(significant) * (10 ** -(i-2))
```

```
                return result if x > 0 else -result

    # Para números normales
    s = f"{abs(x):.10f}"
    if '.' in s:
        integer_part, decimal_part = s.split('.')
        if integer_part != '0':
            if len(integer_part) >= 3:
                result = float(integer_part[:3])
            else:
                digits_needed = 3 - len(integer_part)
                result = float(integer_part + decimal_part[:digits_needed]) / (10 ** dig
        else:
            count_zeros = 0
            for char in decimal_part:
                if char == '0':
                    count_zeros += 1
                else:
                    break
            significant = decimal_part[count_zeros:count_zeros+3]
            result = float(significant) * (10 ** -(count_zeros + 3))
    else:
        result = float(s[:3]) if len(s) > 3 else float(s)

    return result if x >= 0 else -result

def suma_con_corte(terminos):
    suma = 0
    for termino in terminos:
        termino_chop = three_digit_chop(termino)
        suma_chop = three_digit_chop(suma + termino_chop)
        suma = suma_chop
    return suma

print("LITERAL a:  (1/i²) desde i=1 hasta 10")

terminos_asc_a = [1/(i**2) for i in range(1, 11)]
terminos_desc_a = [1/(i**2) for i in range(10, 0, -1)]

# Valores exactos
suma_exacta_a = sum(terminos_asc_a)

# Sumas con corte
suma_asc_a = suma_con_corte(terminos_asc_a)
```

```python
suma_desc_a = suma_con_corte(terminos_desc_a)

print(f"Suma exacta: {suma_exacta_a:.10f}")
print(f"Orden ascendente: {suma_asc_a:.6f}")
print(f"Orden descendente: {suma_desc_a:.6f}")

# Errores
error_asc_a = abs(suma_exacta_a - suma_asc_a)
error_desc_a = abs(suma_exacta_a - suma_desc_a)

print(f"Error orden ascendente: {error_asc_a:.6e}")
print(f"Error orden descendente: {error_desc_a:.6e}")

if error_asc_a < error_desc_a:
    print("\nMÁS PRECISO: Orden ascendente")
    print("  RAZÓN: Al sumar de menor a mayor, se minimiza la pérdida de")
    print("         dígitos significativos cuando se suman números pequeños")
    print("         a acumulados grandes.")
else:
    print("\nMÁS PRECISO: Orden descendente")
    print("  RAZÓN: Al sumar de mayor a menor, los términos más grandes")
    print("         se suman primero, reduciendo errores de redondeo.")

# Demostración de los términos con corte
print("\nTérminos individuales con corte de 3 dígitos:")
print("Orden ascendente:")
for i, term in enumerate(terminos_asc_a, 1):
    print(f"  1/{i}² = {term:.6f} -> {three_digit_chop(term):.6f}")
```

```
LITERAL a:  (1/i²) desde i=1 hasta 10
Suma exacta: 1.5497677312
Orden ascendente: 1.530000
Orden descendente: 1.540000
Error orden ascendente: 1.976773e-02
Error orden descendente: 9.767731e-03

MÁS PRECISO: Orden descendente
  RAZÓN: Al sumar de mayor a menor, los términos más grandes
         se suman primero, reduciendo errores de redondeo.

Términos individuales con corte de 3 dígitos:
Orden ascendente:
  1/1² = 1.000000 -> 1.000000
  1/2² = 0.250000 -> 0.250000
```

```
1/3² = 0.111111 -> 0.111000
1/4² = 0.062500 -> 0.062500
1/5² = 0.040000 -> 0.040000
1/6² = 0.027778 -> 0.027700
1/7² = 0.020408 -> 0.020400
1/8² = 0.015625 -> 0.015600
1/9² = 0.012346 -> 0.012300
1/10² = 0.010000 -> 0.010000
```

b.

$$\sum_{i=1}^{10}\left(\frac{1}{i^3}\right) \text{ primero por } \frac{1}{1}+\frac{1}{8}+\frac{1}{27}+\cdots+\frac{1}{1000} \text{ y luego por } \frac{1}{1000}+\frac{1}{729}+\cdots+\frac{1}{1}$$

```python
def three_digit_chop(x):
    if x == 0:
        return 0

    # Para números muy pequeños
    if abs(x) < 0.001:
        s = f"{abs(x):.10f}"
        for i, char in enumerate(s):
            if char not in ['0', '.']:
                significant = s[i:i+3]
                result = float(significant) * (10 ** -(i-2))
                return result if x > 0 else -result

    # Para números normales
    s = f"{abs(x):.10f}"
    if '.' in s:
        integer_part, decimal_part = s.split('.')
        if integer_part != '0':
            if len(integer_part) >= 3:
                result = float(integer_part[:3])
            else:
                digits_needed = 3 - len(integer_part)
                result = float(integer_part + decimal_part[:digits_needed]) / (10 ** dig
        else:
            count_zeros = 0
            for char in decimal_part:
                if char == '0':
                    count_zeros += 1
                else:
                    break
            significant = decimal_part[count_zeros:count_zeros+3]
```

```
            result = float(significant) * (10 ** -(count_zeros + 3))
    else:
        result = float(s[:3]) if len(s) > 3 else float(s)

    return result if x >= 0 else -result

def suma_con_corte(terminos):
    suma = 0
    for termino in terminos:
        termino_chop = three_digit_chop(termino)
        suma_chop = three_digit_chop(suma + termino_chop)
        suma = suma_chop
    return suma

print("LITERAL b:  (1/i³) desde i=1 hasta 10")

terminos_asc_b = [1/(i**3) for i in range(1, 11)]
terminos_desc_b = [1/(i**3) for i in range(10, 0, -1)]

# Valores exactos
suma_exacta_b = sum(terminos_asc_b)

# Sumas con corte
suma_asc_b = suma_con_corte(terminos_asc_b)
suma_desc_b = suma_con_corte(terminos_desc_b)

print(f"Suma exacta: {suma_exacta_b:.10f}")
print(f"Orden ascendente: {suma_asc_b:.6f}")
print(f"Orden descendente: {suma_desc_b:.6f}")

# Errores
error_asc_b = abs(suma_exacta_b - suma_asc_b)
error_desc_b = abs(suma_exacta_b - suma_desc_b)

print(f"Error orden ascendente: {error_asc_b:.6e}")
print(f"Error orden descendente: {error_desc_b:.6e}")

if error_asc_b < error_desc_b:
    print("\nMÁS PRECISO: Orden ascendente")
    print("  RAZÓN: Para esta serie donde los términos decrecen rápidamente,")
    print("         sumar del más pequeño al más grande reduce los errores")
    print("         de redondeo al evitar sumar números muy pequeños")
    print("         a acumulados grandes.")
else:
```

```
    print("\nMÁS PRECISO: Orden descendente")
    print("  RAZÓN: Los términos más significativos se suman primero,")
    print("         proporcionando una mejor aproximación inicial.")

# Demostración de los términos con corte
print("\nTérminos individuales con corte de 3 dígitos:")
print("Orden ascendente:")
for i, term in enumerate(terminos_asc_b, 1):
    print(f"  1/{i}³ = {term:.8f} -> {three_digit_chop(term):.8f}")
```

```
LITERAL b:  (1/i³) desde i=1 hasta 10
Suma exacta: 1.1975319857
Orden ascendente: 1.160000
Orden descendente: 1.190000
Error orden ascendente: 3.753199e-02
Error orden descendente: 7.531986e-03

MÁS PRECISO: Orden descendente
  RAZÓN: Los términos más significativos se suman primero,
         proporcionando una mejor aproximación inicial.

Términos individuales con corte de 3 dígitos:
Orden ascendente:
  1/1³ = 1.00000000 -> 1.00000000
  1/2³ = 0.12500000 -> 0.12500000
  1/3³ = 0.03703704 -> 0.03700000
  1/4³ = 0.01562500 -> 0.01560000
  1/5³ = 0.00800000 -> 0.00800000
  1/6³ = 0.00462963 -> 0.00462000
  1/7³ = 0.00291545 -> 0.00291000
  1/8³ = 0.00195312 -> 0.00195000
  1/9³ = 0.00137174 -> 0.00137000
  1/10³ = 0.00100000 -> 0.00100000
```

2. La serie de Maclaurin para la función arcotangente converge para $-1 \leq x \leq 1$ 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

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

```python
import math

def arctan_maclaurin(x, n_terms):
    result = 0.0
    for i in range(1, n_terms + 1):
        term = ((-1) ** (i + 1)) * (x ** (2 * i - 1)) / (2 * i - 1)
        result += term
    return result

def encontrar_terminos_para_precision_10_3():
    print("LITERAL 2a: Términos para |4P_n(1) -  | < 10^{-3}")

    pi_real = math.pi
    objetivo_error = 1e-3
    n = 1

    print(f"  real: {pi_real:.10f}")
    print(f"Error objetivo: {objetivo_error}")
    print("\nBuscando número de términos necesarios...")
    print("-" * 60)

    while True:
        # Calcular arctan(1) con n términos
        arctan_approx = arctan_maclaurin(1, n)
        # Calcular  aproximado: 4 * arctan(1)
        pi_approx = 4 * arctan_approx
        # Calcular error
        error = abs(pi_approx - pi_real)

        print(f"n = {n:3d}:  _approx = {pi_approx:.8f}, error = {error:.2e}")

        # Verificar si cumple con el criterio de error
        if error < objetivo_error:
            print("-" * 60)
            print(f"  RESULTADO: Se necesitan {n} términos")
            print(f"  Error final: {error:.2e} < {objetivo_error}")
            print(f"    aproximado: {pi_approx:.8f}")
            print(f"    real:       {pi_real:.8f}")
            break

        n += 1

        # Prevenir bucle infinito
        if n > 1000:
```

```
            print("Advertencia: No se alcanzó la precisión en 1000 términos")
            break

    return n

# Ejecutar el análisis
n_terminos_10_3 = encontrar_terminos_para_precision_10_3()

# Demostración detallada de los primeros términos
print("\n" + "=" * 60)
print("DEMOSTRACIÓN DETALLADA DE LOS PRIMEROS TÉRMINOS")
print("=" * 60)

print("n\tTérmino\t\tSuma Parcial\t4*Suma\t\tError")
print("-" * 80)

suma_parcial = 0
for i in range(1, min(11, n_terminos_10_3 + 3)):
    termino = ((-1) ** (i + 1)) * (1 ** (2 * i - 1)) / (2 * i - 1)
    suma_parcial += termino
    pi_aprox = 4 * suma_parcial
    error = abs(pi_aprox - math.pi)

    print(f"{i}\t{termino:+.6f}\t{suma_parcial:.6f}\t{pi_aprox:.6f}\t{error:.6f}")
```

```
import math

def terminos_para_precision_10_3():
    n = 1
    while True:
        arctan_approx = sum(((-1)**(i+1)) / (2*i-1) for i in range(1, n+1))
        pi_approx = 4 * arctan_approx
        if abs(pi_approx - math.pi) < 1e-3:
            return n
        n += 1

resultado_2a = terminos_para_precision_10_3()
print(f"Literal 2a: Se necesitan {resultado_2a} términos para |4P_n(1) -  | < 10 ³")
```

```
Literal 2a: Se necesitan 1000 términos para |4P_n(1) -  | < 10 ³
```

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?**

```python
import math

def arctan_maclaurin_optimizado(x, n_terms):
    result = 0.0
    x_power = x  # x^(2i-1)

    for i in range(1, n_terms + 1):
        term = ((-1) ** (i + 1)) * x_power / (2 * i - 1)
        result += term
        x_power *= x * x  # Actualizar potencia: x^(2i+1) = x^(2i-1) * x^2
    return result

def encontrar_terminos_para_precision_10_10():
    print("LITERAL 2b: Términos para |4P_n(1) -  | < 10^{-10}")
    print("=" * 60)

    pi_real = math.pi
    objetivo_error = 1e-10
    n = 1

    print(f"  real: {pi_real:.15f}")
    print(f"Error objetivo: {objetivo_error:.1e}")
    print("\nBuscando número de términos necesarios...")
    print("-" * 80)

    # Solo mostrar algunos puntos para no saturar
    puntos_mostrar = [1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]

    while True:
        # Calcular aproximación
        arctan_approx = arctan_maclaurin_optimizado(1, n)
        pi_approx = 4 * arctan_approx
        error = abs(pi_approx - pi_real)

        # Mostrar progreso en puntos específicos
        if n in puntos_mostrar or error < objetivo_error:
            print(f"n = {n:5d}:  _approx = {pi_approx:.12f}, error = {error:.2e}")

        # Verificar criterio
        if error < objetivo_error:
            print("-" * 80)
            print(f"  RESULTADO: Se necesitan {n} términos")
            print(f"  Error final: {error:.2e} < {objetivo_error:.1e}")
            print(f"    aproximado: {pi_approx:.12f}")
```

```
            print(f"    real:      {pi_real:.12f}")

            # Análisis adicional
            print(f"\n  Último término: {1/(2*n-1):.2e}")
            print(f"  El término n={n} es aproximadamente {1/(2*n-1):.2e}")
            break

        n += 1

        # Límite de seguridad
        if n > 1000000:
            print("Advertencia: No se alcanzó la precisión en 1,000,000 términos")
            break

    return n

# Ejecutar el análisis
n_terminos_10_10 = encontrar_terminos_para_precision_10_10()

# Análisis teórico del error
print("\n" + "=" * 60)
print("ANÁLISIS TEÓRICO DEL ERROR")
print("=" * 60)

print("Para la serie alternante arctan(1) = 1 - 1/3 + 1/5 - 1/7 + ...")
print("El error después de n términos está acotado por el primer término omitido:")
print("|Error|   1/(2n+1)")

# Calcular n teórico para error < 1e-10
n_teorico = (1 / (2 * 1e-10) - 1) / 2
print(f"\nPara error < 1e-10, necesitamos 1/(2n+1) < 1e-10")
print(f"Esto implica: 2n + 1 > 1e10")
print(f"n > (1e10 - 1)/2   {n_teorico:.0f}")

print(f"\nComparación:")
print(f"  n encontrado: {n_terminos_10_10}")
print(f"  n teórico: ~{n_teorico:.0f}")
```

```
def terminos_para_precision_10_10():
    # Análisis teórico: error   1/(2n+1) < 10 ¹
    n_teorico = int((1e10 - 1) / 2)
    return n_teorico

resultado_2b = terminos_para_precision_10_10()
```

```
print(f"Literal 2b: Se necesitan ~{resultado_2b:,} términos para |4P_n(1) -  | < 10 ¹ ")
```

```
Literal 2b: Se necesitan ~4,999,999,999 términos para |4P_n(1) -  | < 10 ¹
```

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

$$\pi/4 = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

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

Para resolver este ejercicio, despejamos el número 4 de la izquierda, quedándonos:

$$\pi = \frac{4\arctan\frac{1}{5} - \arctan\frac{1}{239}}{4}$$

Ahora, el siguiente código cumple con la suma de esta funcion con un parámetro n

```
from numpy import arctan
sum = 0
n = 16
for i in range (1, n + 1):
    sum += (4*arctan(1 / 5) - arctan(1/239)) / 4
print(sum)
arc16=(math.pi - sum)
print(abs(arc16))

print(arc16 < 10**(-3))
```

```
3.1415926535897944
1.3322676295501878e-15
True
```

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

a. ENTRADA □,□1,□2,□,□□. SALIDA PRODUCT. Paso 1 Determine PRODUCT = 0. Paso 2 Para □ = 1,2,□,□ haga Determine PRODUCT = PRODUCT□□□. Paso 3 SALIDA PRODUCT; PARE.

b. ENTRADA □,□1,□2,□,□□. SALIDA PRODUCT. Paso 1 Determine PRODUCT = 1. Paso 2 Para □ = 1,2,□,□ haga Set PRODUCT = PRODUCT□□□. Paso 3 SALIDA PRODUCT; PARE.

c. ENTRADA □,□1,□2,□,□□. SALIDA PRODUCT. Paso 1 Determine PRODUCT = 1. Paso 2 Para □ = 1,2,□,□ haga si □□=0 entonces determine PRODUCT = 0; SALIDA PRODUCT; PARE Determine PRODUCT = PRODUCT□□□. Paso 3 SALIDA PRODUCT; PARE.

Respuesta: El algoritmo "a" es el más acertado para el algoritmo de la parte 1a

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

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

Respuesta:

Para esta suma doble: * Se requieren **multiplicaciones**:

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

* Se requieren **sumas**:

$$\left(\sum_{i=1}^{n} i\right) - 1 = \frac{n(n+1)}{2} - 1$$

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

Respuesta:

1. Linealidad de sumatoria: $ = ∑{i=1}^{n} (a_i ∑{j=1}^{i} b_j)$
2. Suma de una progresión aritmética: $\sum_{i=1}^{n} a_i \cdot \frac{i(i+1)}{2}$
3. Reagrupación: $\sum_{i=1}^{n} \frac{i(i+1)}{2} a_i$

1.2 DISCUSIONES

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

```
SInversa = 0
for i in range(5, 0, -1):
    SInversa += i

print("Resultado al inverso:", SInversa)
```

```
Resultado al inverso: 15
```

2. Las ecuaciones (1.2) y (1.3) en la sección 1.2 proporcionan formas alternativas para las raíces □1 y □2 de □□ 2 + □□ + □ = 0. Construya un algoritmo con entrada a, b y 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.

```
def raices(a : float, b : float, c : float) -> tuple[float, float] | float | tuple[compl
    discriminante = b**2 - 4*a*c

    if (discriminante == 0):
        raiz = (-b + (discriminante)**0.5) / 2*a
        return raiz

    elif (discriminante > 0):
        raiz1 = (-b + (discriminante)**0.5) / 2*a
        raiz2 = (-b - (discriminante)**0.5) / 2*a
        return raiz1, raiz2

    else:
        raiz1 = complex(-b / 2*a, (abs(discriminante))**0.5 * 100 // 2*a / 100)
        raiz2 = complex(-b / 2*a, -(abs(discriminante))**0.5 * 100 // 2*a / 100)
        return raiz1, raiz2


resultado = raices(1.5, -2, 1)
print('La/s raiz/ices de la ecuacion cuadratica es/son: ', resultado)
```

```
La/s raiz/ices de la ecuacion cuadratica es/son:  ((1.5+1.05j), (1.5-1.065j))
```

3. Suponga que

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

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}.

```
def calcular_lado_izquierdo(x):
    suma_lado_izquierdo = 0.0
    termino = 1
    denominador = 1.0
    while True:
```

```
        suma_lado_izquierdo += (2.0**(termino - 1) * x**(2*termino - 1)) / denominador
        if abs(suma_lado_izquierdo - (1 + 2*x) / (1 + x + x**2)) < 1e-6:
            break
        termino += 1
        denominador *= 1 - x**(2*termino)
    return termino
```