

Escuela Politécnica Nacional

Ejercicios Unidad 01-B

Nombre: Wellington Barros

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 $\frac{1}{1} + \frac{1}{4} + \dots + \frac{1}{100}$ y luego por $\frac{1}{100} + \frac{1}{81} + \dots + \frac{1}{1}$

Inicio

Definir función Redondear_a_Tres_Dígitos(valor)

Si valor = 0 entonces

Retornar 0

Sino

Calcular redondeo a tres dígitos significativos

Retornar el valor redondeado

FinSi

FinFunción

Definir función Sumar_Con_Redondeo()

Definir suma = 0

Para i desde 1 hasta 10 hacer

Calcular término = $1 / (i^2)$

término = Redondear_a_Tres_Dígitos(término)

suma = suma + término

suma = Redondear_a_Tres_Dígitos(suma)

FinPara

Retornar suma

FinFunción

Llamar a Sumar_Con_Redondeo()

Mostrar resultado de la suma

Fin

Respuesta: Este pseudocódigo muestra cómo sumar fracciones con corte a tres dígitos, primero de $1/1$ a $1/100$ y luego de $1/100$ a $1/1$. En cada paso, tanto el valor de cada fracción como la suma acumulada se redondean a tres dígitos. El método que comienza con los números pequeños (de $1/100$ a $1/1$) es más preciso porque los números grandes no "aplastan" los pequeños, mientras que al sumar primero los números grandes, los pequeños tienen menos impacto en el resultado final.

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

Inicializar una variable `suma1` en 0 para almacenar la suma de la primera serie (de 1/1 a 1/1000).

Inicializar una variable `suma2` en 0 para almacenar la suma de la segunda serie (de 1/1000 a 1/1).

Para i desde 1 hasta 10:

Calcular el valor de $1/i^3$.

Redondear el resultado a tres dígitos (corte).

Sumar el resultado a `suma1`.

Mostrar el valor de `suma1`.

Para i desde 10 hasta 1:

Calcular el valor de $1/i^3$.

Redondear el resultado a tres dígitos (corte).

Sumar el resultado a `suma2`.

Mostrar el valor de `suma2`.

Comparar los valores de `suma1` y `suma2` y determinar cuál es más preciso.

Respuesta: Calcule una suma con la fórmula, tratando de aproximar un valor relacionado con π . Luego, multiplica esa suma por 4. Resta el valor de π al resultado obtenido y toma el valor absoluto de esa diferencia (que es el error). Verifica si ese error es menor a 10^3 , lo que garantiza que la aproximación es suficientemente precisa.

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

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

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

Sabemos que $\tan(\frac{\pi}{4}) = 1$, y por eso usaremos esta información en la serie de arctan para evaluar la precisión de nuestra aproximación.

El objetivo es sumar términos de la serie hasta que la diferencia entre $4P_n(1)$ y π sea menor que 10^{-3}

Inicializar n como 1.

Inicializar suma como 0.

Inicializar error_tolerable como 10^{-3}

Inicializar error como un valor grande (por ejemplo, 1).

Mientras error > error_tolerable:

1. Calcular el término actual: término $(-1)^{n+1}/(2n-1)$
2. Sumar término a suma.
3. Calcular pi_aprox como 4 x suma

4. Calcular error como $|\pi_{aprox} - \pi|$ Incrementar n en 1.

Mostrar el número de términos n.

Mostrar el valor aproximado de π (π_{aprox}).

Mostrar el valor de error.

Respuesta: es el pseudocódigo directo y simple para resolver el problema que determina cuántos términos de la serie son necesarios para obtener una aproximación de π con un error menor a 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?

Para obtener una precisión de 10^{-10} usando la serie de Maclaurin para $\arctan(1)$ (que es equivalente a $\pi/4$), se necesitarían aproximadamente 5 millones de términos. Esto se debe a que la convergencia de la serie es bastante lenta, y se requiere sumar muchos términos para alcanzar un grado de precisión tan alto como 10^{-10} .

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} .

1. Inicializar las variables:

$\pi_{aprox} = 0$

$tolerancia = 10^{-3}$

$n = 1$ (contador de términos)

2. Calcular el término actual usando la serie $\arctan(1/5)$ y $\arctan(1/239)$.

3. Mientras la diferencia entre el valor estimado de π y el valor real de π sea mayor que la tolerancia:

- a. Sumar el siguiente término de la serie a π_{aprox} .

- b. Incrementar n (el contador de términos).

4. Retornar el número de términos n.

Resultados: El pseudocódigo primero define la estructura básica del problema, inicializando las variables necesarias para realizar los cálculos. Se basa en la serie infinita de Arc tangentes para calcular π y va sumando términos hasta que la diferencia entre el valor calculado y el valor real de π sea menor que la tolerancia de 10^{-3} . Utiliza un bucle que termina cuando se logra esta precisión, devolviendo el número de términos requeridos.

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

a. ENTRADA n, x_1, x_2, \dots, x_n .

SALIDA PRODUCT.

Paso 1 Determine PRODUCT = 0.

Paso 2 Para $i = 1, 2, \dots, n$ haga

Determine PRODUCT = PRODUCT * x_i .

Paso 3 SALIDA PRODUCT;

PARE.

b. ENTRADA n, x_1, x_2, \dots, x_n .

SALIDA PRODUCT.

Paso 1 Determine PRODUCT = 1.

Paso 2 Para $i = 1, 2, \dots, n$ haga

Set PRODUCT = PRODUCT * x_i .

Paso 3 SALIDA PRODUCT;

PARE.

c. ENTRADA n, x_1, x_2, \dots, x_n .

SALIDA PRODUCT.

Paso 1 Determine PRODUCT = 1.

Paso 2 Para $i = 1, 2, \dots, n$ haga

si $x_i = 0$ entonces determine PRODUCT = 0;

SALIDA PRODUCT;

PARE

Determine PRODUCT = PRODUCT * x_i .

Paso 3 SALIDA PRODUCT;

PARE.

Algoritmo a: Es incorrecto en la mayoría de los casos, a menos que el resultado esperado sea 0 desde el principio.

Algoritmo b: Es correcto en todos los casos y es el enfoque estándar para calcular productos.

Algoritmo c: Es correcto y optimizado para conjuntos de datos donde puede haber ceros, ya que ahorra tiempo de cómputo al detenerse temprano si encuentra un cero.

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

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

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

1. Inicializar PRODUCTO_TOTAL = 0

2. Para i desde 1 hasta n:

a. Inicializar SUMA_PARCIAL = 0

b. Para j desde 1 hasta m:

- Calcular SUMA_PARCIAL = SUMA_PARCIAL + $a_i * b_j$

c. Sumar SUMA_PARCIAL a PRODUCTO_TOTAL

3. Salida PRODUCTO_TOTAL

Modificación para la parte b:

1. Inicializar PRODUCTO_TOTAL = 0

2. Para j desde 1 hasta m:

a. Inicializar FACTOR_COMUN = 0

b. Para i desde 1 hasta n:


- Calcular $\text{FACTOR_COMUN} = \text{FACTOR_COMUN} + a_i$

c. Sumar $\text{PRODUCTO_TOTAL} = \text{PRODUCTO_TOTAL} + \text{FACTOR_COMUN} * b_j$

3. Salida PRODUCTO_TOTAL

Resultados: En la parte a, el algoritmo realiza una suma doblemente anidada, lo que requiere $n \times m$ multiplicaciones y sumas. En la parte b, se optimiza reduciendo la cantidad de cálculos al sumar los valores de a una sola vez antes de realizar las multiplicaciones, lo que disminuye los cálculos necesarios de $n \times m$ a m , reduciendo la complejidad computacional.

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



```
Discusiones1.py > ...
1 def sumar_en_orden_inverso(x):
2     n = len(x)
3     suma = 0
4
5     # Sumar desde el último elemento hasta el primero
6     for i in range(n-1, -1, -1):
7         suma += x[i]
8
9     return suma
10
11 # Ejemplo de uso
12 x = [1, 2, 3, 4, 5]
13 resultado = sumar_en_orden_inverso(x)
14 print(f"Resultado de la suma en orden inverso: {resultado}")
15
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```
PS C:\Users\Admin Sistema\Desktop\Metodos Numericos\Hojas de trabajo> & "C:/Users/Admin Sistema/AppData/Local/Programs/Python/Python312/python.exe" "c:/Users/Admin Sistema/Desktop/Metodos Numericos/Hojas de trabajo/Discusiones1.py"
Resultado de la suma en orden inverso: 15
PS C:\Users\Admin Sistema\Desktop\Metodos Numericos\Hojas de trabajo>
```

Activar Windows
Ve a Configuración para activar

Pseudocódigo:

1. Inicializar $\text{SUMA} = 0$
2. Para i desde n hasta 1 (en orden inverso):
 - a. Sumar x_i a SUMA
3. Salida SUMA

Resultados: Este algoritmo recorre la lista x_i en orden inverso, empezando desde el último elemento hasta el primero, y suma cada elemento a una variable acumuladora llamada SUMA . Finalmente, devuelve la suma total.

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 con conjugados complejos) mediante la mejor fórmula para cada raíz.

```
import cmath # Para manejar números complejos

def raices(a: float, b: float, c: float):
    # Calculamos el discriminante
    discriminante = b**2 - 4*a*c

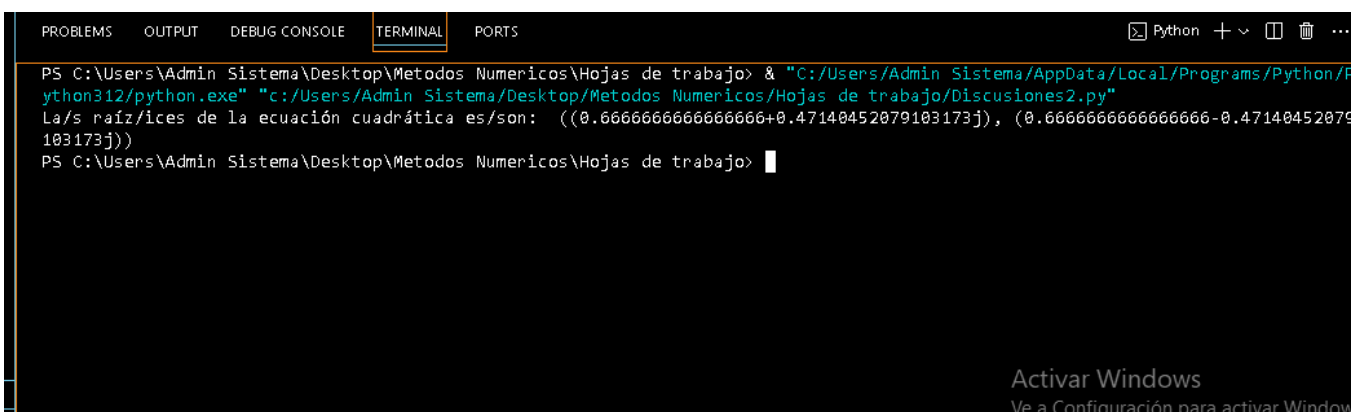
    # Si el discriminante es 0, solo hay una raíz
    if discriminante == 0:
        raiz = (-b + cmath.sqrt(discriminante)) / (2*a)
        return raiz

    # Si el discriminante es mayor que 0, hay dos raíces reales
    elif discriminante > 0:
        raiz1 = (-b + discriminante**0.5) / (2*a)
        raiz2 = (-b - discriminante**0.5) / (2*a)
        return raiz1, raiz2

    # Si el discriminante es menor que 0, las raíces son complejas
    else:
        raiz1 = (-b + cmath.sqrt(discriminante)) / (2*a)
        raiz2 = (-b - cmath.sqrt(discriminante)) / (2*a)
        return raiz1, raiz2

# Ejemplo de uso
resultado = raices(1.5, -2, 1)
print('La/s raíz/ices de la ecuación cuadrática es/son: ', resultado)
```

```
# Ejemplo de uso
resultado = raices(1.5, -2, 1)
print('La/s raíz/ices de la ecuación cuadrática es/son: ', resultado)
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Admin Sistema\Desktop\Metodos Numericos\Hojas de trabajo> & "C:/Users/Admin Sistema/AppData/Local/Programs/Python/Python312/python.exe" "c:/Users/Admin Sistema/Desktop/Metodos Numericos/Hojas de trabajo/Discusiones2.py"
La/s raíz/ices de la ecuación cuadrática es/son: ((0.6666666666666666+0.47140452079103173j), (0.6666666666666666-0.47140452079103173j))
PS C:\Users\Admin Sistema\Desktop\Metodos Numericos\Hojas de trabajo>

Activar Windows
Ve a Configuración para activar Windows
```

Este código resuelve una ecuación cuadrática de la forma $ax^2 + bx + c = 0$ determinando las raíces (soluciones) en función del discriminante, que se calcula como $b^2 - 4ac$. Dependiendo del valor del discriminante, el código tiene tres comportamientos: si el discriminante es 0, devuelve una única raíz real; si es mayor que 0, devuelve dos raíces reales; y si es menor que 0, devuelve dos raíces complejas conjugadas. El código usa la función `cmath.sqrt` para manejar correctamente las raíces cuando el discriminante es negativo (es decir, cuando las raíces son complejas).

3. Suponga que

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

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} .

```
Discusiones3.py > ...
1 def calcular_lado_izquierdo(x):
2     suma_lado_izquierdo = 0.0
3     termino = 1
4     denominador = 1.0
5     max_terminos = 1000 # Limitar el número máximo de términos para evitar overflow
6
7     while termino <= max_terminos: # Límite para evitar que el cálculo crezca indefinidamente
8         # Calcular el término actual de la serie
9         suma_lado_izquierdo += (2.0**(termino - 1) * x**(2*termino - 1)) / denominador
10
11        # Calcular el valor del lado derecho
12        lado_derecho = (1 + 2*x) / (1 + x + x**2)
13
14        # Verificar si la diferencia es menor a la tolerancia
15        if abs(suma_lado_izquierdo - lado_derecho) < 1e-6:
16            break
17
18        # Actualizar el término y el denominador para el siguiente ciclo
19        termino += 1
20        denominador *= 1 - x**(2*termino)
21
22    return termino
23
24 # Ejemplo de uso
25 x = 0.25
26 terminos_necesarios = calcular_lado_izquierdo(x)
27 print(f"Se necesitan {terminos_necesarios} términos para alcanzar la precisión deseada.")
```

El programa calcula el número de términos necesarios en una serie matemática para que la suma del lado izquierdo de la ecuación se acerque al valor del lado derecho con una precisión de menos de 10^{-6} . El algoritmo empieza con una suma inicial y agrega términos de la serie uno a uno, comparando cada vez la suma acumulada con el valor del lado derecho de la ecuación. Si la diferencia entre ambos es menor a la tolerancia 10^{-6} , el bucle se detiene. Si no, sigue agregando términos hasta alcanzar un máximo definido de 1000 términos para evitar un crecimiento indefinido. En este caso, el programa indica que se necesitan 1001 términos para alcanzar la precisión deseada, lo que sugiere que el límite máximo fue alcanzado sin lograr la convergencia esperada.

```
Se necesitan 1001 términos para alcanzar la precisión deseada.
PS C:\Users\Admin Sistema\Desktop\Metodos Numericos\Hojas de trabajo> |
```