# Escuela Politécnica Nacional

**Nombre:** Kevin Eduardo Garcia Rodriguez

**Tema:** [Tarea 03] Ejercicios Unidad 01-B

**Repositorio de GitHub:** https://github.com/Nattyrd/Metodos-Numericos-2025B

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: $\frac{1}{1}+\frac{1}{4}+\cdots+\frac{1}{100}$ Y luego por: $\frac{1}{100}+\frac{1}{81}+\cdots+\frac{1}{1}$

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

In [15]:
```python
from decimal import Decimal, getcontext, ROUND_DOWN

#Trunca a tres cifras significativas.
def cortar_tres_digitos(x):
    if x == 0:
        return 0.0

    d = Decimal(str(x))
    exponent = len(str(int(d.copy_abs()))) - 1
    getcontext().prec = 3
    return float(d.scaleb(-exponent).quantize(Decimal('1.00'), rounding=ROUND_DO

#Suma la serie usando corte de 3 cifras significativas en cada paso.
def suma_con_corte(serie, orden='asc'):
    total = 0.0
    datos = serie if orden == 'asc' else reversed(serie)
    for x in datos:
        total = cortar_tres_digitos(total + cortar_tres_digitos(x))
    return total

# Serie a:
# 1/i^2 para i = 1 a 10
serie_a = [1 / (i**2) for i in range(1, 11)]


# Serie
# b: 1/i^3 para i = 1 a 10
serie_b = [1 / (i**3) for i in range(1, 11)]

# Sumas con corte (ascendente y descendente)
resultado_a_asc = suma_con_corte(serie_a, orden='asc')
resultado_a_desc = suma_con_corte(serie_a, orden='desc')

resultado_b_asc = suma_con_corte(serie_b, orden='asc')
resultado_b_desc = suma_con_corte(serie_b, orden='desc')

print(f"Suma A ascendente: {resultado_a_asc}")
```

```
print(f"Suma A descendente: {resultado_a_desc}")

# Serie a:
# valor real a
def corte_3_decimales(x):
    return int(x * 1000) / 1000
sumaA = 0.0
for i in range(1, 11):
    sumaA += corte_3_decimales(1 / i**2)
print(f"Valor real de la serie A: {sumaA}")

print(f"")
print(f"Suma B ascendente: {resultado_b_asc}")
print(f"Suma B descendente: {resultado_b_desc}")

# Serie b:
# valor real b
def corte_3_decimales(x):
    return int(x * 1000) / 1000
sumaB = 0.0
for i in range(1, 11):
    sumaB += corte_3_decimales(1 / i**3)
print(f"Valor real de la serie B: {sumaB}")
```

```
Suma A ascendente: 1.53
Suma A descendente: 1.53
Valor real de la serie A: 1.547

Suma B ascendente: 1.16
Suma B descendente: 1.16
Valor real de la serie B: 1.1939999999999995
```

Para determinar cual método es el más preciso, procedemos a calcular el error relativo para cada método, con la siguiente fórmula:

$$Error_{relativo} = \left| \frac{x - x'}{x} \right|$$

donde:

x = es el valor real.

x'= el valor aproximado.

In [24]:
```
def error_serieA(sumaA, resultado_a_asc):
    return abs((sumaA - resultado_a_asc) / sumaA)
print(f"Error serie A: {error_serieA(sumaA, resultado_a_asc)} ")

print(f"")

def error_serieB(sumaB, resultado_b_asc):
    return abs((sumaB - resultado_b_asc) / sumaB)
print(f"Error serie B: {error_serieB(sumaB, resultado_b_asc)}")
```

```
Error serie A: 0.010989010989010927

Error serie B: 0.028475711892796986
```

Observando los dos errores de la serie A y B, se puede observar que los errores son de la misma magnitud, pero el error de la serie B es mayor que el de la serie A. Por lo que se puede concluir que la serie A es más precisa que la serie B.

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

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

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

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

In [31]:
```
def aproximar_pi_con_error(toleracia_error):
    suma = 0.0
    i = 1
    while True:
        termino = (-1)**(i + 1) / (2 * i - 1)
        suma += termino
        temp_error = 4 * abs(termino)
        if temp_error < toleracia_error:
            break
        i += 1
    pi_aproximado = 4 * suma
    return pi_aproximado, i

pi, n = aproximar_pi_con_error(1e-3)
print(f"π ≈ {pi}, \nel numero de términos requerido en la serie es: {n}")
```

```
π ≈ 3.1420924036835256,
el numero de términos requerido en la serie es: 2001
```

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

In [1]:
```
import math

# Valores dados por la identidad de Machin:
# π/4 = 4 arctan(1/5) - arctan(1/239)
x1 = 1/5
x2 = 1/239
target = 1e-3  # Queremos que el error en π sea menor que 10^-3


def remainder_bound(x, n):
    """
    Cota superior del resto (error) de la serie de arctan(x)
    después de sumar hasta el término n (0-based).
```

```
    R_n <= x^(2n+3) / (2n+3)
    """
    return x**(2*n+3) / (2*n+3)


# Buscamos los mínimos n1 y n2 tales que el error total en π <= 10^-3
min_solution = None
for n1 in range(0, 50):
    for n2 in range(0, 50):
        R1 = remainder_bound(x1, n1)
        R2 = remainder_bound(x2, n2)
        err_bound = 16 * R1 + 4 * R2  # error máximo en π
        if err_bound <= target:
            min_solution = (n1, n2, err_bound)
            break
    if min_solution:
        break

# Función para calcular la suma parcial de la serie de arctan
def arctan_partial(x, n):
    """
    Calcula la suma parcial de la serie de arctan(x)
    desde k = 0 hasta k = n (inclusive).
    """
    s = 0.0
    for k in range(n + 1):
        term = ((-1)**k) * (x**(2*k + 1)) / (2*k + 1)
        s += term
    return s


# Aplicamos la fórmula de Machin con los n1 y n2 hallados
n1, n2, bound_est = min_solution
S1 = arctan_partial(x1, n1)
S2 = arctan_partial(x2, n2)
pi_approx = 16 * S1 - 4 * S2
pi_true = math.pi
error_real = abs(pi_approx - pi_true)

# Mostramos resultados
print(f"Se necesitan N1 = {n1+1} términos para arctan(1/5)")
print(f"Se necesitan N2 = {n2+1} términos para arctan(1/239)")
print(f"Cota teórica del error en π: {bound_est:.3e}")
print(f"Aproximación de π: {pi_approx:.10f}")
print(f"Valor real de π:    {pi_true:.10f}")
print(f"Error real:         {error_real:.3e}")
```

```
Se necesitan N1 = 3 términos para arctan(1/5)
Se necesitan N2 = 1 términos para arctan(1/239)
Cota teórica del error en π: 2.935e-05
Aproximación de π: 3.1416209317
Valor real de π:    3.1415926536
Error real:         2.828e-05
```

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

**1° Algoritmo**

| ENTRADA $n$, $x1$, $x2$, $\cdots$ , $xn$. SALIDA PRODUCT. Paso 1 Determine PRODUCT = 0. Paso 2 Para $i$ = 1, 2, $\cdots$ , $n$ haga Determine PRODUCT = PRODUCT $*$ $xi$. Paso 3 SALIDA PRODUCT; PARE. |

**2° Algoritmo**

| ENTRADA $n$, $x1$, $x2$, $\cdots$ , $xn$. SALIDA PRODUCT. Paso 1 Determine PRODUCT = 1. Paso 2 Para $i$ = 1, 2, $\cdots$ , $n$ haga Set PRODUCT = PRODUCT $*$ $xi$. Paso 3 SALIDA PRODUCT; PARE. |

**3° Algoritmo**

| ENTRADA $n$, $x1$, $x2$, $\cdots$ , $xn$. SALIDA PRODUCT. Paso 1 Determine PRODUCT = 1. Paso 2 Para $i$ = 1, 2, $\cdots$ , $n$ haga si $xi$ = 0 entonces determine PRODUCT = 0; SALIDA PRODUCT; PARE Determine PRODUCT = PRODUCT $*$ $xi$. Paso 3 SALIDA PRODUCT; PARE. |

El b) algoritmo es el correcto, cumple con enunciado

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

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

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

In [33]:
```python
def suma_original(a, b):
    """Calcula la suma usando el orden original."""
    n = len(a)
    total = 0
    for i in range(n):
        for j in range(i + 1):
            total += a[i] * b[j]
    return total

def suma_optimizada(a, b):
    """Calcula la suma con menos operaciones usando la transformación equivalent
    n = len(a)
    suma_ai = [0] * n
    # Calcula sumas parciales acumuladas desde a[j] hasta a[n-1]
    acumulado = 0
    for j in reversed(range(n)):
        acumulado += a[j]
        suma_ai[j] = acumulado

    total = 0
    for j in range(n):
        total += b[j] * suma_ai[j]
    return total

# Ejemplo
```

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8]

print("Suma original:", suma_original(a, b))
print("Suma optimizada:", suma_optimizada(a, b))
```

```
Suma original: 185
Suma optimizada: 185
```

# Discuciones

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

ENTRADA: n, x_1, x_2, ..., x_n

SALIDA: suma_total

Paso 1: Inicializar suma_total = 0

Paso 2: Para i desde n hasta 1 ( decrementa -1):

```
    suma_total = suma_total + x_i
```

Paso 3: Devolver suma_total

FIN

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 $x1$, $x2$ que calcule las raíces $x_1$ y $x_2$ (que pueden ser iguales con conjugados complejos) mediante la mejor fórmula para cada raíz.

```
    plaintext
    ENTRADA: a, b, c
    SALIDA: x_1 , x_2

    ENTRADA: a, b, c
    SALIDA: x1, x2

    Paso 1: Calcular discriminante D = b² - 4ac

    Paso 2: Si D ≥ 0:
        Si b ≥ 0:
            x1 = [2c] / [-b - √D]  # Fórmula (1.3) para raíz más
    pequeña
            x2 = [-b - √D] / [2a]   # Fórmula (1.2) para raíz más
    grande
        Sino:
            x1 = [-b + √D] / [2a]   # Fórmula (1.2) para raíz más
    grande
            x2 = [2c] / [-b + √D]   # Fórmula (1.3) para raíz más
```

```
pequeña
Sino (D < 0):
    Calcular parte real = -b/(2a)
    parte imaginaria = √|D|/(2a)
    x1 = parte real + parte imaginaria*i
    x2 = parte real - parte imaginaria*i
FIN
```

3. suponga que

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

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 $10^{-6}$

In [1]:
```
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
```

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