Trabalho Prático 1
Métodos Numéricos

Professora Maria João Rodrigues

Elementos do Grupo 8

up202406540 Filipe ZhuHang
up202406887 Jie Chengli
up202303606 Orlando Soares
up202304528 Paulo Lin

Relatório do Trabalho Prático 1

Cálculo Numérico de π e Análise de Convergência de Séries

1. Motivação do Trabalho

O presente trabalho tem como objetivo **explorar métodos numéricos para o cálculo de π**, recorrendo a diferentes **séries matemáticas infinitas** e analisando a **precisão** e **eficiência computacional** das aproximações obtidas.

Para tal, são estudadas **duas séries de naturezas distintas** — uma de **convergência rápida** e outra de **convergência lenta** —, permitindo compreender como a velocidade de convergência influencia o **erro absoluto**, o **número de iterações necessárias** e o **tempo de execução**.

Além do cálculo de π, o trabalho inclui também a **determinação do epsilon máquina (ε_m)**, um conceito fundamental da **aritmética de ponto flutuante (IEEE 754)**, que define o **limite de precisão** numérica dos cálculos realizados por computadores.

Este estudo permite, assim, consolidar conceitos centrais da **análise numérica**, nomeadamente:

- O papel do **epsilon máquina** e suas implicações na precisão computacional
- A **taxa de convergência** de diferentes séries e sua influência no erro
- Os **critérios de paragem** baseados na majoração de erros
- A distinção entre **erro teórico** (majorado) e **erro real** (observado)

A relevância deste trabalho reside na demonstração de que **diferentes formulações matemáticas para o mesmo valor podem apresentar eficiências computacionais significativamente distintas**, evidenciando a importância da **escolha adequada do método numérico** na resolução de problemas científicos e de engenharia.

---

2. Métodos Usados no Trabalho

2.1. Cálculo do Epsilon Máquina

O **epsilon máquina (ε_m)** representa a menor diferença entre `1` e o próximo número maior representável em **aritmética de ponto flutuante**.

Esse valor é importante porque define os **limites da precisão** dos cálculos numéricos.

Programa Desenvolvido

O valor de ε_m foi calculado **iterativamente**, até que a condição $1 + \varepsilon = 1$ seja satisfeita em aritmética de ponto flutuante (IEEE 754). O algoritmo implementado divide sucessivamente o epsilon por 2 até que a adição não seja mais detectável pelo sistema de representação numérica. Resultado obtido: $\varepsilon_m = 2.220446 \times 10^{-16}$ Este valor é idêntico ao retornado por `numpy.finfo(float).eps`, confirmando a correção da implementação. A complexidade do algoritmo é O(log(1/ε)), tornando-o altamente eficiente.

In [2]:
```
import math
import time     # para limitar o tempo maximo de execuçao
```

In [3]:
```
def epsilon_maquina():
    eps = 1.0
    while (1.0 + eps / 2.0) > 1.0: eps = eps / 2.0
    return eps
```

2.2. Cálculo da Série de Convergência Rápida para Aproximação de π

A série utilizada para o cálculo de π é dada por:

$$S = 6 \sum_{k=0}^{\infty} \frac{(2k)!}{4^k (k!)^2 (2k+1)} (0.5)^{2k+1}$$

Esta série é de **convergência rápida**, permitindo obter resultados com elevada precisão após um número reduzido de iterações.

Programa Desenvolvido

O programa implementa a soma iterativa dos termos da série até que o **erro absoluto** seja **inferior ao valor de ε** especificado.

Para melhorar significativamente o desempenho, foram introduzidas **otimizações numéricas** que evitam o recálculo redundante de fatoriais e potências — operações altamente custosas quando calculadas repetidamente.

O algoritmo **armazena e atualiza os valores de**:

- `(2k)!` → atualizado multiplicativamente: `(2k+2)(2k+1)(2k)!`
- `k!` → atualizado como `(k+1)! = (k+1) * k!`
- `4^k` → atualizado multiplicativamente por 4 em cada iteração

Dessa forma, o custo computacional de cada novo termo reduz-se de **O(k)** para **O(1)**, tornando o algoritmo **muito mais eficiente** e numericamente estável.

In [4]:
```python
def calcular_S_eficiente(epsilon):
    res = 0.0
    k = 0

    fact_2k = 1       # (2k)! começando em k=0, (2*0)! = 0! = 1
    fact_k = 1        # k! começando em 0! = 1
    exp_4k = 1        # 4^k começando em k=0

    while True:
        # Calcula termo k usando variáveis armazenadas
        denom = exp_4k * (fact_k**2) * (2*k + 1)
        termo = (fact_2k / denom) * (0.5)**(2*k + 1)

        res += termo

        # Preparar variáveis para próximo termo (k+1)
        k_next = k + 1

        # Atualiza fact_2k para (2(k+1))! = (2k+2)(2k+1)(2k)!
        fact_2k_next = fact_2k * (2*k_next) * (2*k_next - 1)
        # Atualiza k! para (k+1)! = (k+1) * k!
        fact_k_next = fact_k * k_next
        # Atualiza 4^k para 4^(k+1)
        exp_4k_next = exp_4k * 4

        # Calcula próximo termo para checar erro
        denom_next = exp_4k_next * (fact_k_next**2) * (2*k_next + 1)
        termo_next = (fact_2k_next / denom_next) * (0.5)**(2*k_next + 1)

        if abs(termo_next) < epsilon / 6: break
```

```
        # Atualiza variáveis para próxima iteração
        fact_2k = fact_2k_next
        fact_k = fact_k_next
        exp_4k = exp_4k_next
        k = k_next

    S_aproximado = 6 * res
    return S_aproximado, k + 1  # k começa em 0, somamos k+1 termos
```

2.3. Cálculo Numérico de π com a Série de Leibniz

A série de **Leibniz** para π é dada por:

$$S = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

ou seja:

$$\pi \approx 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots\right).$$

Esta série **converge lentamente**. Portanto, para que o erro seja menor que ε, é necessário um número de termos da ordem de O(1/ε), o que torna esta série impraticável para ε muito pequenos.

Programa Desenvolvido

Implementámos a soma iterativa com um **controlo de tempo (timeout)** para evitar execuções excessivamente longas:

In [5]:
```
def calcular_S_leibniz(epsilon, timeout=3):
    res = 0.0
    k = 0
    tempo_inicio = time.time()
    sinal = 1

    while True:
        termo = sinal / (2*k + 1)

        # Critério de paragem
        if abs(termo) < epsilon/4:
            break

        res += termo
        k += 1
        sinal *= -1

        # Verifica timeout
        if k % 100000 == 0:
            if time.time() - tempo_inicio > timeout:
                S_aproximado = 4 * res
                return S_aproximado, k, True
```

```
    S_aproximado = 4 * res
    return S_aproximado, k, False
```

2.4. Análise Global e Comparação dos Métodos

Para concluir o trabalho, foi desenvolvido um bloco de código responsável por **executar todas as funções implementadas**, **avaliar o desempenho** e **comparar os resultados numéricos obtidos**.

O objetivo da função `analise_completa()` é integrar os diferentes métodos — desde o cálculo do **ε-máquina**, até à **aproximação de π** através das séries de **convergência rápida** e de **Leibniz** —, apresentando de forma sistemática os **valores calculados**, **erros absolutos**, **número de iterações** e **tempo de execução**.

In [6]:
```
def analise_completa():
    """
    Executa todos os cálculos e apresenta resultados formatados.
    """
    print("=" * 80)
    print("TRABALHO PRÁTICO 1 - CÁLCULO NUMÉRICO DE π")
    print("=" * 80)

    # Problema 1
    print("\n1. EPSILON MÁQUINA")
    print("-" * 80)
    eps_maquina = epsilon_maquina()
    print(f"Epsilon máquina calculado: {eps_maquina:.6e}")
    print(f"(Verificação: 1 + eps = {1.0 + eps_maquina})")

    pi_exato = math.pi  # Valor exato de π (para comparação)

    # Problemas 2, 3 e 4
    epsilons = [1e-5, 1e-10, 1e-15]

    print("\n2. SÉRIE DE CONVERGÊNCIA RÁPIDA")
    print("-" * 80)
    print(f"{'ε':<12} {'n termos':<12} {'S calculado':<20} {'Erro |π-S|':<15} {'
    print("-" * 80)

    for eps in epsilons:
        tempo_inicio = time.time()
        S, n = calcular_S_eficiente(eps)
        tempo_fim = time.time()
        tempo_exec = tempo_fim - tempo_inicio
        erro = abs(pi_exato - S)

        aviso = " ⚠ DEMORADO!" if tempo_exec > 10 else ""
        print(f"{eps:<12.0e} {n:<12d} {S:<20.16f} {erro:<15.6e} {tempo_exec:<12.

    print("\n3. SÉRIE DE LEIBNIZ")
    print("-" * 80)
    print(f"{'ε':<12} {'n termos':<12} {'S calculado':<20} {'Erro |π-S|':<15} {'
    print("-" * 80)

    for eps in epsilons:
```

```
        tempo_inicio = time.time()
        S, n, timeout_atingido = calcular_S_leibniz(eps, timeout=1000)
        tempo_fim = time.time()
        tempo_exec = tempo_fim - tempo_inicio
        erro = abs(pi_exato - S)

        if timeout_atingido:
            aviso = " ⚠ TIMEOUT! (cálculo interrompido)"
            print(f"{eps:<12.0e} {n:<12d} {S:<20.16f} {erro:<15.6e} {tempo_exec:
        else:
            aviso = " ⚠ DEMORADO!" if tempo_exec > 3 else ""
            print(f"{eps:<12.0e} {n:<12d} {S:<20.16f} {erro:<15.6e} {tempo_exec:

    print("\n4. ANÁLISE COMPARATIVA")
    print("-" * 80)
    print(f"Valor exato de π: {pi_exato:.16f}")
    print("=" * 80)
```

In [7]:
```
analise_completa()
```

```
================================================================================
TRABALHO PRÁTICO 1 - CÁLCULO NUMÉRICO DE π
================================================================================

1. EPSILON MÁQUINA
--------------------------------------------------------------------------------
Epsilon máquina calculado: 2.220446e-16
(Verificação: 1 + eps = 1.0000000000000002)

2. SÉRIE DE CONVERGÊNCIA RÁPIDA
--------------------------------------------------------------------------------
ε            n termos     S calculado          Erro |π-S|      Tempo (s)
--------------------------------------------------------------------------------
1e-05        7            3.1415894253191219   3.228271e-06    0.0000
1e-10        14           3.1415926535153385   7.445466e-11    0.0000
1e-15        22           3.1415926535897936   4.440892e-16    0.0000

3. SÉRIE DE LEIBNIZ
--------------------------------------------------------------------------------
ε            n termos     S calculado          Erro |π-S|      Tempo (s)
--------------------------------------------------------------------------------
1e-05        200000       3.1415876535897618   5.000000e-06    0.1281
1e-10        3149000000   3.1415926532706604   3.191327e-10    1000.0124    ⚠ TI
MEOUT! (cálculo interrompido)
1e-15        3136300000   3.1415926532693952   3.203979e-10    1000.0498    ⚠ TI
MEOUT! (cálculo interrompido)

4. ANÁLISE COMPARATIVA
--------------------------------------------------------------------------------
Valor exato de π: 3.1415926535897931
================================================================================
```

3. Melhorias possíveis para a Série de Leibniz

1. **A série de Leibniz, embora simples, é extremamente lenta para pequenas tolerâncias (ε):**
   A convergência é linear e requer um número **astronómico de termos** para

alcançar uma precisão razoável. Mesmo com optimizações básicas, o crescimento do erro relativo torna-se rapidamente incontrolável.

2. **A redução do número de termos é uma estratégia eficaz, mas limitada:**
   A compressão 2 a 2 (i.e., $1/a - 1/(a+2) = 2/(a(a+2))$) permite **reduzir pela metade** o número de divisões — a operação mais dispendiosa — sem perda significativa de precisão.
   Contudo, compressões superiores (4 a 4, 8 a 8, etc.) tornam os termos muito mais complexos e pouco vantajosos, dado o custo elevado das divisões face às multiplicações nos processadores modernos.

3. **Transformações e pré-cálculos matemáticos podem melhorar a complexidade temporal:**
   Métodos como **Leibniz–Euler** e **Leibniz–Aitken** aceleram a convergência ao custo de cálculos mais pesados por termo.
   No caso da formulação de Euler, a complexidade pode ser reduzida de $\mathbf{O(n^3)}$ para $\mathbf{O(n^2)}$ ao pré-calcular coeficientes binomiais com o triângulo de Pascal, tornando-a comparável em desempenho à versão de Aitken.

4. **A velocidade de execução depende fortemente do ambiente de execução:**
   O interpretador **CPython** é intrinsecamente lento devido à ausência de compilação nativa.
   Ferramentas como **Numba** e **Cython** permitem compilar o código Python em funções nativas, obtendo ganhos de desempenho de **até 100×**.
   Alternativamente, a **vetorização com NumPy** permite processar milhões de termos simultaneamente, reduzindo drasticamente o tempo de execução.

---

4. Conclusões Finais

Este trabalho demonstrou de forma clara e quantitativa que:

1. **A escolha da formulação matemática é determinante:**
   Duas séries que convergem para o mesmo valor de π podem diferir em **ordens de magnitude** quanto à eficiência computacional, chegando a variações superiores a 10^5 vezes no número de iterações necessárias.

2. **A série de convergência rápida revela-se ideal para aplicações práticas:**
   Atinge **precisão de máquina** com apenas **22 termos**, apresentando tempos de execução praticamente nulos.

3. **A série de Leibniz, embora de grande valor pedagógico, é computacionalmente ineficiente:**
   Para **precisões superiores a $\mathbf{10^{-5}}$**, o número de termos torna-se proibitivo, mesmo com tentativas de **aceleração de convergência** (como o método de Euler).

4. **O epsilon máquina define um limite intransponível de precisão:**
   Em aritmética de **dupla precisão**, não é possível obter erros inferiores a aproximadamente $\mathbf{10^{-16}}$, independentemente da formulação matemática utilizada.

5. **A introdução de mecanismos de timeout é essencial:**
   Garante a **robustez prática dos algoritmos**, prevenindo execuções indefinidas em métodos que, apesar de corretos em teoria, são **ineficientes na prática**.

O problema proposto foi **resolvido com sucesso** para a **série de convergência rápida** em todos os valores de ε e **parcialmente resolvido** para a **série de Leibniz** (com convergência apenas até $\varepsilon = 10^{-5}$), evidenciando na prática as limitações de métodos com **convergência lenta**.

Referências

- Burden, R. L., & Faires, J. D. (2010). *Numerical Analysis* (9th ed.). Brooks/Cole.
- IEEE Standard 754-2008 for Floating-Point Arithmetic.
- Python Software Foundation. *Python 3 Documentation* – `math` module.
- NumPy Documentation – Data types and numerical precision.