Aproximação de π com Análise de Performance

Descrição da Tarefa

Este projeto implementa um programa em C que calcula aproximações de π usando séries matemáticas, variando o número de iterações e medindo o tempo de execução. O objetivo é comparar os valores obtidos com o valor real de π e analisar como a acurácia melhora com mais processamento computacional.

> **Objetivo Principal:** Demonstrar a relação fundamental entre esforço computacional e precisão numérica, um princípio que governa desde simulações científicas até inteligência artificial moderna.

□ Métodos de Aproximação de π

Comparação dos Dois Métodos

Série de Leibniz: π/4 = 1 - 1/3 + 1/5 - 1/7 + ...

- **Vantagem:** Muito simples de implementar
- **Desvantagem:** Convergência lenta (precisa de muitas iterações)
- **Característica:** Usa denominadores ímpares (1, 3, 5, 7...)

Série de Nilakantha: π = 3 + 4/(2×3×4) - 4/(4×5×6) + ...

- **Vantagem:** Convergência muito mais rápida
- **Desvantagem:** Cálculo ligeiramente mais complexo
- **Característica:** Usa produtos de três números consecutivos

Resultado: Nilakantha atinge alta precisão com muito menos iterações

que Leibniz

⚙️ Funcionalidades Implementadas

Análise de Performance

- **measure_time():** Mede tempo de execução com precisão de clock
- **calculate_error():** Calcula erro absoluto comparado ao valor real
- **print_results():** Formata resultados em tabela organizada

Testes Automatizados

O programa realiza testes com 6 diferentes números de iterações, variando de 100 a 10,000,000, permitindo análise detalhada da convergência e performance.

📊 Resultados da Execução

🔢 Resultados Completos - Série de Leibniz

Iterações	π Aproximado	Tempo (s)	Erro	Precisão
100	3.131592903559	0.000000	1.00e-02	99.6817%
1,000	3.140592653840	0.000003	1.00e-03	99.9682%
10,000	3.141492653590	0.000026	1.00e-04	99.9968%
100,000	3.141582653590	0.000290	1.00e-05	99.9997%
1,000,000	3.141591653590	0.002561	1.00e-06	100.0000%
10,000,000	3.141592553590	0.028028	1.00e-07	100.0000%

⚡ Resultados Completos - Série de Nilakantha

Iterações	π Aproximado	Tempo (s)	Erro	Precisão
100	3.141592410972	0.000027	2.43e-07	100.0000%
1,000	3.141592653341	0.000003	2.49e-10	100.0000%
10,000	3.141592653590	0.000032	2.55e-13	100.0000%
100,000	3.141592653590	0.000293	6.66e-15	100.0000%
1,000,000	3.141592653590	0.002945	6.22e-15	100.0000%
10,000,000	3.141592653580	0.031370	9.45e-12	100.0000%

🏆 Comparação Direta (1,000,000 iterações)

Método	π Aproximado	Tempo (s)	Erro	Diferença de Precisão
Leibniz	3.141591653590	0.002805	1.00e-06	Base
Nilakantha	3.141592653590	0.002891	6.22e-15	**160.000x mais precisa**

🔍 Análise dos Resultados

📊 Principais Descobertas:

- **Nilakantha é dramaticamente superior:** Com apenas 100 iterações, já atinge erro de 2.43e-07
- **Leibniz precisa de 10 milhões de iterações** para atingir erro similar (1.00e-07)
- **Tempos similares:** Ambos métodos têm performance temporal parecida
- **Precision ceiling:** Limitação da precisão do tipo double em ~10^-15

⚖️ Trade-offs Observados:

- **Simplicidade vs Eficiência:** Leibniz é mais simples, Nilakantha é mais eficiente
- **Iterações vs Precisão:** Nilakantha atinge alta precisão com 100x menos iterações
- **Tempo de desenvolvimento vs Performance:** Nilakantha compensa complexidade adicional

🌍 O Padrão de Precisão Crescente em Aplicações Reais

O comportamento observado neste projeto - onde maior esforço computacional resulta em precisão incrementalmente melhor - é um padrão fundamental que se repete em diversas aplicações críticas da computação moderna.

1. Simulações Físicas de Alta Fidelidade

🛩️ Dinâmica de Fluidos Computacional (CFD)

- **10^3 células:** estimativa grosseira do arrasto
- **10^6 células:** captura turbulência básica
- **10^9 células:** resolve detalhes críticos para segurança
- **Custo:** Dias de supercomputador para cada incremento
- **Impacto:** Diferença entre aprovação e rejeição em certificação

🏗️ Simulações Estruturais (Elementos Finitos)

- **Malha grosseira:** tendências gerais de tensão
- **Malha refinada:** identifica pontos de falha críticos
- **Trade-off:** Cada refinamento dobra o tempo de computação
- **Consequência:** Erro de 1% pode significar colapso estrutural

2. Inteligência Artificial e Aprendizado de Máquina

🤖 Treinamento de Modelos de Linguagem

- **GPT-1 (117M parâmetros):** texto básico
- **GPT-3 (175B parâmetros):** capacidades emergentes

- **GPT-4 (~1.7T parâmetros):** raciocínio sofisticado
- **Custo:** Crescimento exponencial de recursos

🎯 Algoritmos de Busca (AlphaGo)

- **1.000 simulações:** jogada razoável
- **100.000 simulações:** nível profissional
- **10.000.000 simulações:** superhumano
- **Escalabilidade:** Cada ordem de magnitude requer 10x mais hardware

3. Computação Científica e Pesquisa

🧬 Descoberta de Medicamentos

- **Nanosegundos:** movimentos locais
- **Microsegundos:** mudanças conformacionais
- **Milisegundos:** dobramento completo de proteínas
- **Desafio:** Cada escala temporal requer ordens de magnitude mais computação

4. Padrões Comuns e Lições Aprendidas

Lei dos Retornos Decrescentes Universais

Precisão ∝ log(Recursos Computacionais)

🔬 Conceitos de Programação Paralela

Este projeto demonstra conceitos fundamentais para programação paralela:

1. Paralelização Potencial:

- Loop principal pode ser dividido entre threads
- Redução paralela para somar termos
- Independência entre iterações

2. Medição de Performance:

- Benchmarking preciso com clock()
- Análise de escalabilidade
- Profiling de algoritmos

3. Trade-offs Computacionais:

- Tempo vs Precisão
- Memória vs Velocidade
- Algoritmo vs Hardware

Conclusões

Este projeto ilustra princípios fundamentais da computação científica:

1. Algoritmos diferentes têm características de convergência distintas

2. Existe sempre um trade-off entre precisão e performance

3. A escolha do algoritmo pode ser mais importante que recursos computacionais

4. Medição rigorosa é essencial para otimização

Estes conceitos são aplicáveis em simulações reais, IA, computação financeira e qualquer área que demande cálculos iterativos de alta precisão.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>     // Para fabs() - cálculo do erro absoluto
#include <time.h>     // Para medição de tempo com clock()

#ifdef _WIN32
    #include <windows.h>
#endif

#define PI_REAL 3.14159265358979323846  // Valor de referência de π com alta precisão

// Série de Leibniz: convergência lenta mas simples π/4 = 1 - 1/3 + 1/5 - 1/7 + ...
double calculate_pi_leibniz(long long iterations) {
    double pi_approx = 0.0;
    int sign = 1;                    // Alterna entre +1 e -1 para os sinais da série

    for (long long i = 0; i < iterations; i++) {
        pi_approx += sign * (1.0 / (2 * i + 1));  // Denominadores ímpares: 1, 3, 5, 7...
        sign *= -1;                              // Alterna o sinal a cada iteração
    }

    return 4.0 * pi_approx;          // Multiplica por 4 pois calculamos π/4
}

// Série de Nilakantha: convergência mais rápida π = 3 + 4/(2*3*4) - 4/(4*5*6) + 4/(6*7*8) - ...
double calculate_pi_nilakantha(long long iterations) {
    double pi_approx = 3.0;          // Começa com 3, valor base da série
    int sign = 1;                    // Alterna sinais: +, -, +, -, ...

    for (long long i = 1; i <= iterations; i++) {
        long long n = 2 * i;         // Gera números pares: 2, 4, 6, 8...
        pi_approx += sign * (4.0 / (n * (n + 1) * (n + 2)));  // Produto de 3 números consecutivos
        sign *= -1;                  // Alterna sinal para próxima iteração
    }

    return pi_approx;                // Retorna aproximação direta de π
}

// Medição de tempo usando ponteiros para função - permite testar qualquer algoritmo
double measure_time(double (*func)(long long), long long iterations) {
    clock_t start = clock();         // Marca tempo inicial em ticks do processador
    double result = func(iterations);  // Executa a função passada como parâmetro
    clock_t end = clock();           // Marca tempo final

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;  // Converte para segundos
    return time_taken;
}

// Calcula diferença absoluta entre valor real e aproximação
double calculate_error(double approximation) {
    return fabs(PI_REAL - approximation);  // fabs() garante valor positivo
}

// Formatação padronizada dos resultados em tabela organizada
void print_results(const char* method, long long iterations, double pi_approx, double time_taken) {
    double error = calculate_error(pi_approx);                    // Erro absoluto
    double accuracy_percentage = (1.0 - (error / PI_REAL)) * 100.0;  // Precisão percentual

    printf("%-15s | %12lld | %15.12f | %10.6f | %12.2e | %8.4f%%\n",
           method, iterations, pi_approx, time_taken, error, accuracy_percentage);
}

int main() {
    printf("Cálculo de Aproximações de π\n");
    printf("Valor real de π: %.15f\n\n", PI_REAL);

    // Teste com diferentes escalas para analisar convergência e performance
    long long test_iterations[] = {100, 1000, 10000, 100000, 1000000, 10000000};
    int num_tests = sizeof(test_iterations) / sizeof(test_iterations[0]);  // Número de testes

    printf("%-15s | %12s | %15s | %10s | %12s | %8s\n",
           "Método", "Iterações", "π Aproximado", "Tempo (s)", "Erro", "Precisão");
    printf("----------------|--------------|-----------------|------------|--------------|----------\n");

    // Análise da série de Leibniz - convergência lenta mas conceptualmente simples
    printf("\nSérie de Leibniz (π/4 = 1 - 1/3 + 1/5 - 1/7 + ...):\n");
    for (int i = 0; i < num_tests; i++) {
```

```
        double pi_approx = calculate_pi_leibniz(test_iterations[i]);       // Calcula aproximação
        double time_taken = measure_time(calculate_pi_leibniz, test_iterations[i]);  // Mede tempo
        print_results("Leibniz", test_iterations[i], pi_approx, time_taken);
    }

    // Análise da série de Nilakantha - convergência mais rápida
    printf("\nSérie de Nilakantha (π = 3 + 4/(2×3×4) - 4/(4×5×6) + ...):\n");
    for (int i = 0; i < num_tests; i++) {
        double pi_approx = calculate_pi_nilakantha(test_iterations[i]);    // Calcula aproximação
        double time_taken = measure_time(calculate_pi_nilakantha, test_iterations[i]);  // Mede tempo
        print_results("Nilakantha", test_iterations[i], pi_approx, time_taken);
    }

    // Comparação direta entre os dois métodos com mesmo número de iterações
    printf("\n=== ANÁLISE COMPARATIVA ===\n");

    long long comparison_iterations = 1000000;  // Número fixo para comparação justa
    printf("\nComparação com %lld iterações:\n", comparison_iterations);

    // Testa Leibniz com medições independentes
    double leibniz_pi = calculate_pi_leibniz(comparison_iterations);
    double leibniz_time = measure_time(calculate_pi_leibniz, comparison_iterations);
    double leibniz_error = calculate_error(leibniz_pi);

    // Testa Nilakantha com medições independentes
    double nilakantha_pi = calculate_pi_nilakantha(comparison_iterations);
    double nilakantha_time = measure_time(calculate_pi_nilakantha, comparison_iterations);
    double nilakantha_error = calculate_error(nilakantha_pi);

    printf("\nLeibniz    : π ≈ %.12f | Erro: %.2e | Tempo: %.6f s\n",
           leibniz_pi, leibniz_error, leibniz_time);
    printf("Nilakantha : π ≈ %.12f | Erro: %.2e | Tempo: %.6f s\n",
           nilakantha_pi, nilakantha_error, nilakantha_time);
    return 0;
}
```

Projeto desenvolvido para demonstrar conceitos de aproximação numérica, análise de performance e fundamentos de programação paralela.

Projeto educacional - uso livre para fins acadêmicos.