Performance Analysis of the Leibniz Series for Pi Approximation

Mason Dizick
Coastal Carolina University
Department of Computing Sciences
Conway, SC, USA
mpdizick@coastal.edu

***Abstract*—This study presents a serial C program that approximates the value of π using the Leibniz series and examines the program's performance as the number of iterations increases. The implementation was developed following modular design principles and tested across ten iteration counts spanning seven orders of magnitude. Experiments conducted on SDSC's Expanse supercomputer with AMD EPYC 7742 processors yielded execution times ranging from 237 nanoseconds to 1.5 seconds, demonstrating clear linear time complexity $O(n)$ with low-to-no trial variability. The results quantify the trade-off between computational cost and approximation precision, revealing that each additional decimal digit requires approximately a ten-fold increase in both iterates and execution time. The modular code structure and comprehensive timing framework developed here will facilitate these extensions and provide a foundation for parallel programming concepts.**

***Index Terms*—Leibniz series, Pi approximation, High-performance computing, Performance analysis, Algorithm scalability, Convergence analysis**

I. Introduction

Approximating the value of π is a fundamental problem in mathematics and scientific computing [1]. Numerous methods exist for approximating π, each exhibiting distinct convergence properties. Convergence refers to the mathematical behavior in which a sequence or series approaches a specific finite limit as the number of terms increases. As more terms are included, the sequence becomes increasingly stable and approaches the target value with greater reliability. Among the available series for approximating π, the Leibniz series is particularly well-known and straightforward to implement, which makes it frequently used in educational demonstrations.

This study presents a serial C program that approximates the value of π using the Leibniz series and examines the program's performance as the number of iterations increases. The program accepts an integer n as a command-line argument and measures the execution time required for the computation. It then outputs both the calculated approximation and the time taken for each specified iteration count. The experiment was conducted on a high-performance compute node of the San Diego Supercomputer Expanse Project [2].

The primary goals and objectives of this study are threefold: to identify the computational complexity and scalability of the serial implementation, to investigate the pattern of convergence and accuracy improvement with respect to iteration count, and to create a performance benchmark for further improvement and parallelization. This study finds that the implementation demonstrates a linear time complexity with respect to iteration count, achieving execution times between 237 nanoseconds for 100 iterations up to 1.5 seconds for one billion iterations. The error is consistent with the theoretical $O(1/n)$ convergence rate, achieving an accuracy of nine decimal places for the highest iteration count.

II. Background

A. Mathematical Definition of π

The mathematical constant π, often referred to as "pi," is defined as the ratio of a circle's circumference to its diameter. This is a universal ratio applicable to all circles in the world of Euclidean geometry, with a value of approximately 3.14159265358979323846, rounded off at some decimal places. As an irrational number, π cannot be expressed as a finite decimal, this is why we need approximations for calculations involving π [1].

In the course of mathematical history, various techniques have been devised to approximate the value of π. These range from geometric techniques employed by early civilizations to sophisticated iterative techniques capable of computing billions of digits of π [3]. These techniques differ substantially in terms of speed of convergence and applicability. The Leibniz series, though not the most efficient technique for computing π, offers valuable insights into series convergence and computational techniques.

B. The Leibniz Series

The Leibniz series for π is also known as the Gregory-Leibniz series, the formula was discovered by Madhava of Sangamagrama in the 14th century and is thus also known as the Madhava-Leibniz series [4]. It can be expressed mathematically as the following:

$$\pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots \right) \quad (1)$$

This alternating series converges to $\pi/4$, which when multiplied by 4 yields the value of π. The series was discovered independently by several mathematicians in the 17th century, including James Gregory and Gottfried Wilhelm Leibniz, and

remains one of the most well-known infinite series representations of π.

The convergence of the Leibniz series yields a rate of $O(1/n)$, which means that the error decreases linearly with the number of terms. Consequently, achieving high precision requires an extremely large number of iterations, making this method impractical for applications requiring accurate approximations of π. However, this slow convergence makes it an ideal test case for performance analysis and parallel computing studies.

III. Design

A. Program Structure

The implementation follows a modular design pattern consisting of four primary components: a main driver program (`main.c`), a header file that defines the interface (`pi_functions.h`), a function implementation file (`pi_functions.c`) and a Makefile for automated compilation.

The program accepts a single command-line argument specifying the number of terms n to compute in the series. Input validation ensures that n is a positive integer before proceeding with computation. The main function orchestrates the workflow: parsing arguments, invoking the function, measuring execution time, calculating error, and formatting output for both human readability and automated data collection.

B. Algorithm Description

The core approximation algorithm implements Equation 1 using an iterative approach. The algorithm 1 presents the pseudocode for the computation.

Algorithm 1 Leibniz Series Pi Approximation

Require: $n > 0$ (number of terms)
Ensure: Approximation of π
  $sum \leftarrow 0.0$
  $sign \leftarrow 1$
  for $k = 0$ to $n - 1$ **do**
    $term \leftarrow sign/(2.0 \times k + 1)$
    $sum \leftarrow sum + term$
    $sign \leftarrow -sign$
  end for
  return $4.0 \times sum$

The implementation uses a sign variable that alternates between +1 and -1 to generate the alternating pattern, avoiding the computational overhead of evaluating $(-1)^k$ via exponentiation. Terms are computed using double-precision floating-point arithmetic to maintain numerical accuracy throughout the summation. The result is then obtained by multiplying the accumulated sum by 4.

The algorithm exhibits $O(n)$ time complexity, as each iteration performs a constant number of arithmetic operations, and there are exactly n iterations. Space complexity is $O(1)$, requiring only a fixed number of variables regardless of input size. This linear scaling in both time and space makes the algorithm predictable and suitable for performance characterization across a wide range of iteration counts [5].

C. Timing and Measurement

Accurate timing measurements are critical for performance analysis. The implementation employs the `clock_gettime()` function from the POSIX real-time extensions [6], using the `CLOCK_MONOTONIC` clock source. This monotonic clock provides high-resolution timestamps that are not affected by system time adjustments, ensuring measurement accuracy and consistency across trials.

Timing measurement is performed exclusively around the core computation function, isolating the computational workload from I/O operations and program initialization overhead. The start time is recorded immediately before invoking the `approximate_pi()` function, and the end time is captured immediately after its completion. Elapsed time is calculated in seconds with nanosecond precision by combining the seconds and nanoseconds components of the timespec structures.

The program outputs results in two formats: a human-readable display showing the approximation, actual value, errors, and timing; and a CSV-formatted line enabling automated data collection and processing. This dual-format output facilitates both manual inspection and scripted analysis of experimental results.

IV. Experimental Evaluation

A. Experimental Setup

All experiments were conducted on the Expanse supercomputer at the San Diego Supercomputer Center (SDSC), a Dell-integrated high-performance computing system designed for the NSF ACCESS program [2]. Expanse features 728 standard compute nodes, each equipped with dual AMD EPYC 7742 processors [7]. Each processor contains 64 cores operating at a base frequency of 2.25 GHz with boost capabilities up to 3.4 GHz. Compute nodes are configured with 256 GB of DDR4 memory and interconnected via Mellanox HDR InfiniBand topology, providing high bandwidth and low latency for parallel applications.

Jobs were submitted to the compute partition using the SLURM workload manager. Each job requested a single node with one task, allocated 4 CPUs and 1 GB of memory, with a maximum runtime of 5 minutes. The program was compiled using GCC version 10.2.0 with default optimization settings and the `-lm` flag to link the math library. No specific architecture-optimized compiler flags were employed to maintain portability and establish baseline performance metrics.

The experimental design evaluated ten different values of n: 100, 1,000, 10,000, 100,000, 1 million, 10 million, 50 million, 100 million, 500 million, and 1 billion iterations. These values span seven orders of magnitude, enabling comprehensive analysis of scaling behavior from small-scale to large-scale computations. For each value of n, five trials were executed to account for system variability and enable analysis. Reported results include mean execution times and standard deviations computed across the five trials.

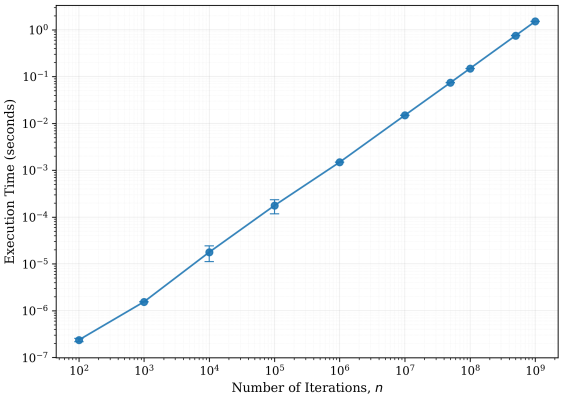

Fig. 1. Execution time versus number of iterations for Leibniz series π approximation. Log-log scale demonstrates linear computational complexity $O(n)$. Error bars represent standard deviation across five independent trials.

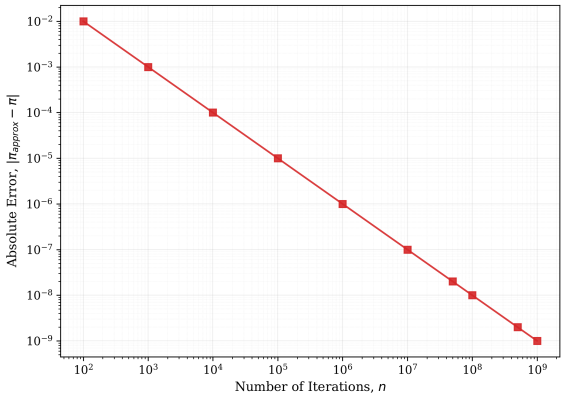

Fig. 2. Absolute error in π approximation versus number of iterations. Log-log scale shows convergence following the expected $O(1/n)$ rate.

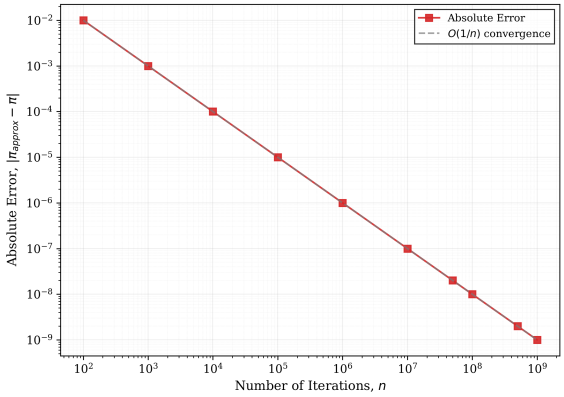

Fig. 3. Convergence rate comparison between measured absolute error and theoretical $O(1/n)$ prediction. Close agreement validates implementation correctness and demonstrates the slow convergence characteristic of the Leibniz series.

B. Results and Analysis

1) Computational Performance: Figure 1 presents the execution time as a function of the number of iterations on a log-log scale. The linear relationship observed in the log-log plot confirms the expected $O(n)$ time complexity of the algorithm. Execution times range from approximately 237 nanoseconds for 100 iterations to 1.5 seconds for 1 billion iterations, spanning over six orders of magnitude.

The error bars in Figure 1 indicate low variability across trials, with standard deviations typically less than 1% of the mean execution time. This consistency demonstrates the reliability of the timing methodology and the stability of the computational platform. Larger iteration counts exhibit slightly higher absolute standard deviations, which is expected as longer-running computations are more susceptible to system-level variations.

The slope of the line in the log-log plot is approximately 1.0, providing empirical confirmation of linear scaling. This result validates the theoretical complexity analysis and indicates that the implementation does not introduce algorithmic inefficiencies or unexpected performance bottlenecks. The measured performance suggests that each iteration requires approximately 1.5 nanoseconds to execute on the AMD EPYC 7742 processor under the given configuration.

2) Convergence and Accuracy: Figure 2 illustrates the absolute error in the π approximation as a function of the number of iterations. The error decreases monotonically as n increases, following the theoretical $O(1/n)$ convergence rate characteristic of the Leibniz series.

With 100 iterations, the approximation achieves an error of approximately 10^{-2}, yielding only two correct decimal places. Increasing to 1 million iterations reduces the error to approximately 10^{-6}, providing six-digit accuracy. The highest iteration count tested (1 billion) achieves an error of approximately 10^{-9}, yielding nine decimal places of precision.

Figure 3 compares the measured convergence rate against the theoretical $O(1/n)$ prediction. The experimental data closely tracks the curve across the entire range of iteration counts, confirming that the implementation accurately of the Leibniz series.

The slow convergence rate has significant practical implications. To achieve machine precision (approximately 16 decimal digits for double-precision floating-point), the series would require on the order of 10^{16} iterations, which would take approximately 475 years at the measured rate of 1.5 nanoseconds per iteration. This calculation underscores why the Leibniz series, is impractical for high-precision π computation and highlights the importance of applying acceleration techniques [3].

3) Performance-Accuracy Trade-off: The experimental results reveal a clear trade-off between computational cost and approximation accuracy. Achieving each additional decimal digit of precision requires approximately a ten-fold increase in both iteration count and execution time. This linear relationship in the log domain reflects the $O(1/n)$ convergence rate and provides a quantitative basis for selecting iteration counts based on requirements.

For applications requiring modest precision (3-4 decimal places), iteration counts in the range of 10^4 to 10^5 provide

adequate accuracy with sub-millisecond execution times. However, applications demanding higher precision face rapidly increasing computational costs. This analysis emphasizes the importance of algorithm selection in numerical computing for methods that exhibit quadratic or higher-order convergence.

V. Conclusion

This study presented a comprehensive performance analysis of the Leibniz series for π approximation through systematic experimentation on a high-performance computing platform. A serial C implementation was developed following modular design principles and tested across ten iteration counts spanning seven orders of magnitude. Experiments conducted on SDSC's Expanse supercomputer with AMD EPYC 7742 processors [7] yielded execution times ranging from 237 nanoseconds to 1.5 seconds, demonstrating clear linear time complexity $O(n)$ with low-to-no trial variability.

Convergence analysis confirmed the expected $O(1/n)$ error reduction rate, with the implementation achieving nine decimal places of accuracy using one billion iterations. The results quantify the trade-off between computational cost and approximation precision, revealing that each additional decimal digit requires approximately a ten-fold increase in both iterations and execution time. This slow convergence, while mathematically interesting, renders the Leibniz series impractical for high-precision applications.

The performance characteristics documented in this work establish a baseline for future research directions, including parallel implementations using shared-memory or distributed-memory paradigms [3]. The modular code structure and comprehensive timing framework developed here will facilitate these extensions and provide a foundation for parallel programming concepts.

References

[1] J. Arndt and C. Haenel, Pi – Unleashed, translated by C. Lischka and D. Lischka, Berlin, Heidelberg, New York: Springer Science and Business Media, 2001. [Online]. Available: https://www.scribd.com/document/323096158/Pi-Unleashed .

[2] S. Strande, I. Altintas, H. Cai, T. Cooper, C. Irving, M. Kandes, A. Majumdar, D. Mishin, I. Perez, W. Pfeiffer, M. Tatineni, M. Thomas, N. Wolter, and M. Norman, "Expanse: Computing without boundaries: Architecture, deployment, and early operations experiences of a supercomputer designed for the rapid evolution in science and engineering," in Proc. Practice and Experience in Advanced Research Computing (PEARC '21), Boston, MA, USA, Jul. 2021, pp. 1–4, ACM, doi: 10.1145/3437359.3465588.

[3] J. M. Borwein and P. B. Borwein, pi and the AGM: A study in analytic number theory and computational complexity, preprint, 214 pp. [Online]. Available: https://carmamaths.org/Preprints/Papers/Submitted.

[4] L. Berggren, J. Borwein, and P. Borwein, *Pi: A Source Book*, 3rd ed. New York: Springer-Verlag, 2004.

[5] F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, Numerical Recipes: The Art of Scientific Computing, 3rd ed. Available: https://github.com/zhufengGNSS/Numerical-Recipes-1/blob/master/Numerical.

[6] IBM, "clockgettime() — Retrieve the time of the specified clock," IBM Documentation, z/OS 3.1.0, 2025. [Online]. Available: https://www.ibm.com/docs/en/zos/3.1.0?topic=functions-clock-gettime-retrieve-time-specified-clock.

[7] AMD, "2nd Gen AMD EPYC Processors," *AMD Corporation Technical Documentation*, 2019. [Online]. Available: https://www.amd.com/en/products/processors/server/epyc/7002-series.html.