**Bounded Angle, Boundless Pi**

Jeerapat Saisuwan

Abstract

The polygonal method for π approximation, using inscribed and circumscribed polygons, provides a geometrically intuitive approach to high-precision computation. Its main challenge is selecting the angle α to achieve a specified number of correct decimal digits, historically hindered by ambiguity. This research introduces a rigorous framework to resolve this issue, enhancing the method’s applicability for extreme-precision tasks.

We propose the Matching-Digit Approximation (MDA), a novel technique deriving a lower bound for α to ensure desired precision. By analyzing the Taylor series’ second term and employing a probabilistic model assuming π’s digits are uniformly distributed, MDA mitigates carry propagation errors, ensuring robust accuracy. Additionally, we developed the Complex Degree Sine (CDS) expression, leveraging Euler’s formula to optimize sine calculations for small angles.

Integrated into a three-step algorithm—specifying precision, estimating carry propagation, and computing π—this approach was implemented in C using MPFR and MPC libraries with dual-thread parallelization. On an Intel i7-4770 (3.4 GHz, 32 GB RAM), it computed π to 14,369,420 digits in 131.1 seconds, verified against the MIT π database. With O(d² log d) complexity, comparable to the Chudnovsky algorithm, our method offers a trigonometric alternative to series-based approaches. Its geometric clarity and parallelization potential make it appealing for numerical analysis. Future enhancements, such as increased parallelization or modern hardware, could further reduce runtimes, establishing this method as a valuable tool for high-precision computations.

Acknowledgments

This project presents a novel trigonometric-based approach for computing π to high precision by combining the Complex Degree Sine (CDS) expression with a rigorously derived error bound. A three-step methodology was developed and successfully implemented in C (using *MPFR* and *MPC*).

I would like to express my sincere thanks to Florian Booneiam for his generous assistance with revising and proofreading this work. His detailed feedback significantly improved the clarity and accuracy of the manuscript.

I am also grateful to Xiao Jingda for his insightful comments on refining the research idea, his guidance on the C implementation, and his thoughtful proofreading. His support—both academically and through our shared moments of relaxation on Roblox—is genuinely appreciated.

Lastly, I am deeply thankful to my parents for their unwavering encouragement, support, and belief in me throughout this journey.

Jeerapat Saisuwan

**Table of Contents**

Abstract

Acknowledgments

Chapter 1: Introduction

1.1 Background and Significance

1.2 Objectives

1.3 Scope of Study

1.4 Definition of Terms Chapter

2: Related Mathematical Concepts

2.1 Trigonometric Foundations

2.1.1 Trigonometric Ratios

2.1.2 Taylor Series Related to Trigonometric Functions

2.2 Limit and Differentiation Tools

2.2.1 Theorems Related to Limits

2.2.2 L’Hôpital’s Rule

2.3 Complex Numbers in Trigonometry

2.3.1 Euler’s Formula and Trigonometric Identities

2.3.2 Complex Logarithm and Exponentiation

2.4 Polygon-Based Approximation of π

2.4.1 Perimeter of Regular Polygons in the Unit Circle

2.4.2 Limit Evaluation Using Radians Chapter

3: Methodology

3.1 Tools Used

3.1.1 C Programming Language

3.1.2 GNU MPFR Library

3.1.3 GNU MPC Library

3.1.4 POSIX Threads (pthreads)

3.2 Steps of Implementation

3.2.1 Derive a Formula for Correct Decimal Digits

3.2.2 Modify the Trigonometric Expression

3.2.3 Compute π to High Precision

3.2.4 Algorithm Overview Chapter

4: Results 4.1 Deriving the Formula for Correct Decimal Places

4.2 Applying Complex Function for π Calculation

4.3 Applying CDS Expression for π Calculation Chapter

5: Conclusion, Discussion, and Recommendations

5.1 Conclusion

5.2 Discussion

5.3 Recommendations References Appendix A: C Code Used for π Calculation

Appendix A: C Code Used for π Calculation

**Chapter 1: Introduction**

**1.1 Background and Significance**

Background

The pursuit of π, the ratio of a circle’s circumference to its diameter, has shaped mathematical history for centuries. Early approaches employed the polygonal method of π approximation, using inscribed and circumscribed regular polygons to estimate π’s value, offering geometric simplicity but limited by imprecise angle selection for high accuracy.

The polygonal method’s challenge lies in determining the angle α to achieve specific precision, often relying on trial-and-error. Over time, infinite series and iterative algorithms surpassed polygonal techniques, enabling computations of π to trillions of digits with methods like Chudnovsky’s, optimized for modern computers.

Inspired by the polygonal method’s geometric elegance, this project leverages trigonometric functions, complex analysis, and Taylor series to refine π approximation. By investigating patterns in decimal digit matching, we aim to develop a systematic, high-precision computational approach, addressing the historical ambiguity in α selection.

Significance

Innovative Methodology (MDA): This research introduces the Matching-Digit Approximation (MDA), which derives a lower bound for α using Taylor series and a probabilistic model to manage carry propagation. This ensures precise digit matching, enhancing the polygonal method’s reliability for extreme-precision tasks.

Computational Achievement: Utilizing the Complex Degree Sine (CDS) expression, implemented in C with MPFR and MPC libraries and dual-thread parallelization, our algorithm computed π to 14,369,420 digits in 131.1 seconds on an Intel i7-4770 (3.4 GHz, 32 GB DDR3 RAM). Verified against the MIT π database, this result confirms the method’s accuracy.

Mathematical and Practical Value: With O(d² log d) complexity, comparable to Chudnovsky’s, our trigonometric approach offers a novel alternative to series-based methods, emphasizing geometric intuition and parallelization potential. For mathematicians, this work provides a fresh perspective on π computation, with applications in numerical analysis and opportunities for optimization on advanced hardware.

**1.2 Objectives**

**1.2.1 Derive a Precision Formula for α**: Develop a formula to determine the number of correct decimal digits of π achievable for a given angle α in the polygonal method of π approximation. By leveraging the Matching-Digit Approximation (MDA) with Taylor series and probabilistic carry propagation analysis, this formula ensures precise angle selection to meet specified accuracy requirements.

**1.2.2 Create an Efficient Sine Computation Algorithm**: Design an algorithm to compute sin(α) for small α in a high-precision environment, optimizing performance for the polygonal method. Utilizing the Complex Degree Sine (CDS) expression based on Euler’s formula, implemented in C with MPFR and MPC libraries, this algorithm enhances computational efficiency for extreme-precision π calculations.

**1.2.3 Compute π to High Precision**: Calculate π to 14,369,420 decimal places using the developed method. By integrating MDA and CDS within a parallelized C implementation, this objective achieves high accuracy, validated against the MIT π database, demonstrating the method’s reliability on modest hardware.

**1.3 Scope of Study**

**1.3.1 Precision Estimation Framework**

* **1.3.1.1** Formulate an exact mathematical inequality to estimate the number of correctly matched decimal digits of π based on a given angle α, using the Matching-Digit Approximation (MDA).
* **1.3.1.2** Derive a simplified version of this inequality to provide a practical and conservative bound for selecting α.

**1.3.2 Complex Function Optimization**

* Develop and apply the Complex Degree Sine (CDS) expression, derived from Euler’s formula, to improve the computation of sin(α) in high-precision environments.

**1.3.3 High-Precision Implementation**

* Implement a full pipeline—from precision estimation to sine evaluation and π computation—using high-precision libraries (MPFR and MPC) and dual-threaded parallelism in C.

1.4 Definition of Terms

1.4.1 Pi (π):  
A mathematical constant defined as the ratio of a circle’s circumference to its diameter. Its decimal representation is non-terminating and non-repeating, and it appears frequently in geometry, trigonometry, and complex analysis.

1.4.2 n-gon:  
A regular polygon with *n* equal sides and *n* equal interior angles. In the context of π approximation, increasing the number of sides of the inscribed or circumscribed *n*-gon improves the accuracy of the π estimate.

1.4.3 α (Alpha):  
The central angle in degrees or radians used in polygonal π approximation. Selecting an appropriate α is critical for achieving the desired number of matching decimal digits.

1.4.4 Matching-Digit Approximation (MDA):  
A derived inequality that provides a lower bound for α to ensure a specific number of correct decimal digits in the computed value of π. It incorporates probabilistic modeling of carry propagation in decimal addition.D

1.4.5 Carry Propagation:  
A phenomenon in arithmetic where the addition of decimal digits may affect adjacent digits through overflow (i.e., when a digit exceeds 9). It plays a key role in the precision analysis of numerical computations involving π.

1.4.6 Complex Degree Sine (CDS):  
A trigonometric expression derived from Euler’s formula and complex exponentials, optimized for computing sin(α) when α is a small angle expressed in degrees. It is designed for use in high-precision arithmetic.

1.4.7 MPFR and MPC:  
High-precision arithmetic libraries used in C. MPFR handles arbitrary-precision real numbers, while MPC extends this to complex numbers, both with correct rounding behavior.

1.4.8 Precision (d):  
The number of decimal digits of π that are targeted in the computation. Higher values of *d* require smaller α and more computational effort.

1.4.9 m:  
An auxiliary integer parameter derived from the desired precision *d*, used to calculate the angle α via the formula ​. A larger *m* implies a smaller α and greater accuracy.

**Chapter 2: Related Mathematical Concepts**

**2.1 Trigonometric Foundations**

2.1.1 Trigonometric Ratios

Trigonometric ratios form the basis of polygonal approximations of π, particularly in the use of sine functions to approximate arc lengths.

Consider a right triangle △ABC, with angle A and the standard labeling:

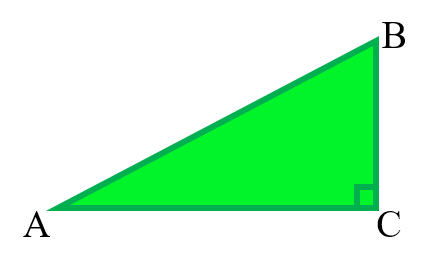


Figure: 1 Triangle ABC

Adjacent side to angle A: AC

Opposite side to angle A: BC

Hypotenuse: AB

The fundamental trigonometric ratios are:

These identities are essential when analyzing geometric methods for π approximation and when connecting real-angle trigonometry to complex representations later in this study.

2.1.1 Taylor Series Related to Trigonometric Functions

The first four terms of the Taylor series expansions of common trigonometric functions around are given as follows:

These series expansions are especially relevant in this research, as they form the analytical basis for the Matching-Digit Approximation (MDA) and the high-precision evaluation of sin(α).

**2.2 Limit and Differentiation Tools**

2.2.1 Theorems related to Limits

The following are standard limit theorems that are foundational for calculus and analysis, and are frequently applied in the derivation of trigonometric approximations for π:

Special limits involving trigonometric functions:

2.2.2 L’Hôpital’s Rule

L’Hôpital’s Rule provides a method for evaluating indeterminate forms such as or :

This rule is particularly useful when dealing with trigonometric limits such as or small-angle approximations.

**2.3 Complex Numbers in Trigonometry**

2.3.1 Euler’s Formula and Trigonometric Identities

Euler’s formula establishes a fundamental relationship between exponential and trigonometric functions:

From this identity, the following important trigonometric relationships are derived:

These relationships allow trigonometric functions to be computed through complex exponentials — a key idea used in the Complex Degree Sine (CDS) expression introduced later in this research.

2.3.2 Complex Logarithm and Exponentiation

Complex Logarithm (multi-valued form):

For any non-zero complex number , the logarithm is defined as:

This form reflects the multi-valued nature of the argument function in the complex plane.

principal branch of the complex logarithm

To define a **single-valued function**, we restrict the argument to its principal value:

,

Here, is the **principal argument** of z.

Complex Exponentiation:

Complex exponentiation is defined based on the complex logarithm:

This definition is crucial for expressing terms like, which appear in the derivation of the Complex Degree Sine expression (CDS) used in the π approximation algorithm.

**2.4 Polygon-Based Approximation of π**

This section explores how regular polygons, either drawn inside or outside a circle, are used to estimate π. By increasing the number of polygon sides, the shape gets closer to the circle, improving the accuracy of π’s value. The method calculates the polygon’s perimeter to set upper and lower bounds for π, which narrow as the angle between sides shrinks. This geometric approach, rooted in historical techniques, forms the basis for our method to compute π with high precision, as detailed later in the paper.

2.4.1 Perimeter of Regular Polygons in the Unit Circle

Consider two regular polygons: one **inscribed in** and one **circumscribed around** the unit circle.

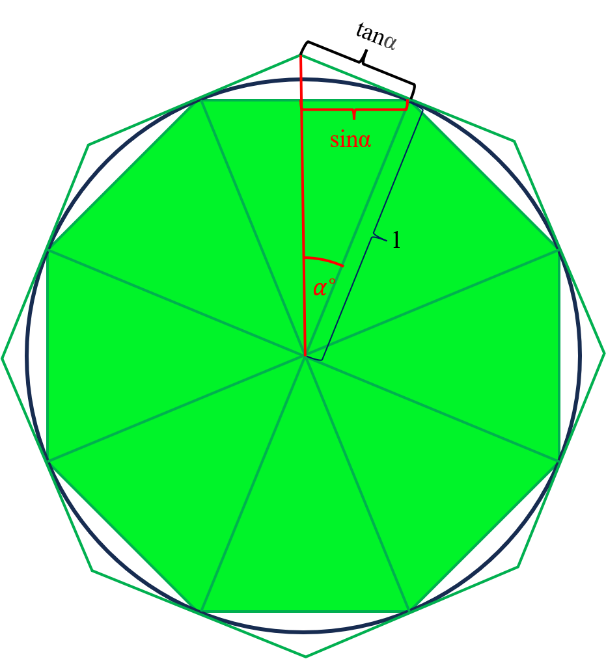


Figure 2: two regular polygons: one **inscribed in** and one **circumscribed around** the unit circle.

For a regular polygon, the perimeter is calculated as:

In the case of polygons constructed relative to a unit circle, the number of sides is ​, where α is the half-angle of one sector.

Inscribed polygon:

Circumscribed polygon:

From this, we obtain the inequality:

which simplifies to:  
This inequality demonstrates that as the angle decreases (i.e., as the number of polygon sides increases), both bounds converge toward π.

2.4.2 Limit Evaluation Using Radians

To analyze the limits as , we first convert degrees to radians:

Lower Bound:

Apply the limit:

Using L’Hôpital’s Rule:  
So,

Upper Bound:

Apply the limit:

Separate the expression for clarity:

=

Using L’Hôpital’s Rule:

So,

**Chapter 3: Methodology**

**3.1 Tools Used**

3.1.1 C Programming Language

C is a general-purpose, procedural programming language known for its speed, portability, and close-to-hardware operations. In this project, C was chosen for implementing a high-performance, limit-based numerical method for approximating π. The program was written entirely in standard C and compiled using the GNU Compiler Collection (GCC), a free and open-source compiler system supporting multiple programming languages (GNU Project, n.d.-a). The implementation also uses the <math.h> library for standard mathematical operations and POSIX-compliant headers such as <pthread.h> and <time.h> for multithreading and high-resolution timing (IEEE, 2018).

3.1.2 GNU MPFR Library

To ensure high-precision floating-point arithmetic, the project employs the GNU MPFR library, which extends the capabilities of standard C by offering correctly rounded, arbitrary-precision floating-point computations (Fousse et al., 2007). This allows for accurate manipulation of mathematical constants and limits.

3.1.3 GNU MPC Library

For complex-number calculations with arbitrary precision, the GNU MPC library is used. Built on top of GMP and MPFR, MPC supports complex floating-point arithmetic with well-defined rounding (Enge et al., 2012). This was essential for computing values like and in the π approximation algorithm.

3.1.4 POSIX Threads (pthreads)

The program utilizes POSIX threads (pthreads) to parallelize the computation of complex exponentials. This multithreading approach improves performance and reduces execution time, especially for high-precision calculations (IEEE, 2018).

**3.2 Steps of Implementation**

3.2.1 Derive a formula that determines the number of correct decimal digits of π for any given α. A mathematical inequality, referred to as the Matching-Digit Approximation (MDA), is derived to estimate the number of correctly matched decimal digits of π based on a given angle α. This involves analyzing the second term of the Taylor series expansion and incorporating probabilistic modeling of carry propagation to ensure precision.

3.2.2 Modify the expression sin(α) and evaluate it using the C programming language and high-precision libraries. The Complex Degree Sine (CDS) expression, derived from Euler’s formula, is developed to compute sin(α) efficiently for small angles α. This expression is implemented in C using the MPFR and MPC libraries to handle high-precision arithmetic, optimizing performance for the polygonal method of π approximation.

3.2.3 Compute the value of π to 14,369,420 decimal digits using the derived method. The derived MDA and CDS are integrated into a C program that computes π to 14,369,420 decimal digits. The implementation employs dual-threaded parallelism with POSIX threads and high-precision libraries, The results are checked against a trusted π database to make sure they are correct.

3.2.4 Algorithm Overview The algorithm calculates π to a chosen number of decimal places in three steps. First, it picks a suitable angle by figuring out how many digits are needed, including extra ones to avoid errors from numbers carrying over during addition. Second, it calculates a trigonometric value for that angle using special high-precision math tools in the C program to ensure accuracy. Third, it splits the work into two parts, running them at the same time with separate threads, to combine the results and produce π. The C program (shown in Appendix A) takes the number of digits as input, processes the calculations, and saves π to a file with the time it took.

**Chapter 4: Results**

**4.1 Deriving the Formula for Correct Decimal Places**

As mentioned in the section Related Mathematical Concepts, both  and approach pi as approach 0. However, the specific value of α required for a given level of accuracy has not been determined. To identify the pattern, multiple values of α in degrees —typically in powers of 10 —are substituted.

substitute , matching digit:

substitute , matching digit:

substitute , matching digit: 9

substitute , matching digit:

substitute , matching digit:

Next, consider the expression which can be rewritten using a trigonometric identity as:

=

Both terms contain the common factor ; therefore, we now turn our attention to, which will be analyzed using its Taylor series expansion.

Given that is a decimal whose digits are denoted by where and each , define this quantity as b for convenience. Then:

Consider:

For clarity of presentation, the addition of is shown in columnar format, as illustrated below. Let denote the changed digits after addition, where and each.

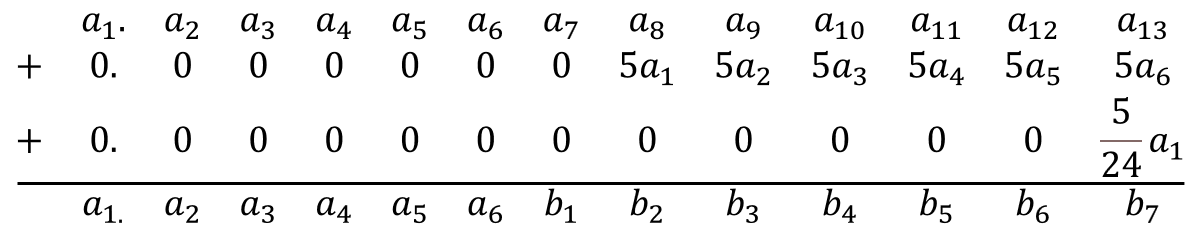


Figure 3: Columnar addition of (in this case ).

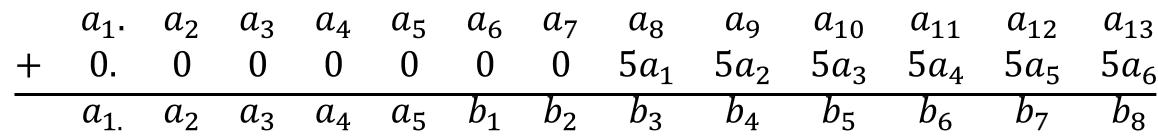
It is evident that the matching decimal digits are primarily influenced by the second term in the Taylor series. Therefore, considering only the first two terms is sufficient for futher effect on digits analysis as shown below.

Figure 4: Columnar addition of illustrating a 2-digit carry propagation effect (in this case ).

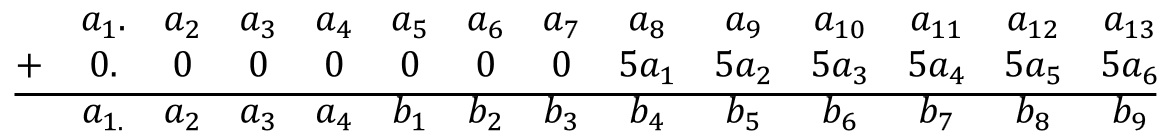


Figure 5: Columnar addition of illustrating a 3-digit carry propagation effect.(in this case ).

The influence of the second term can theoretically spread indefinitely if each digit results in a carry to the next place value. However, such behavior is highly unlikely to occur when computing the value of pi, due to the nature of its decimal expansion and the diminishing magnitude of higher-order terms.

Now consider the effect of the approximation on the value of taking into account both integer and decimal digits, under the condition that approaches 0, and assuming no carry propagation occurs.

substituting will guarantee at least 1 correct digit.

substituting will guarantee at least 2 correct digit.

substituting will guarantee at least correct digit.

Define k is number of the digits effected by carry propagation and it is obvious that:

substituting will guarantee at least 1 correct digit.

substituting will guarantee at least 2 correct digit.

substituting will guarantee at least correct digit.

We now turn our attention to determining a practical value of , the number of digits to which a carry might propagate, in order to compute the first digits of pi with high precision.

To ensure our π computation is accurate, we analyze how addition errors, called carry propagation, affect the digits. Since π’s digits (e.g., 3.14159…) are believed to behave like random numbers from 0 to 9, each digit has an equal chance (1/10) of appearing. When adding numbers to compute π, a carry occurs if the sum of two digits exceeds 9 (e.g., 3 + 7 = 10, causing a carry to the next digit). For any digit, if one number has a digit of 7, 8, or 9 (three out of ten possible values), the sum with another digit may exceed 9, leading to a carry. This gives a carry probability of approximately 3/10 = 0.3 per digit. This assumption, supported by studies showing π’s digits are nearly random (Bailey, 1997), helps us predict how many extra digits to compute to avoid errors.

To refine this estimate, we then consider the influence of successive digits of pi. By analyzing digit patterns that trigger a carry, we obtain the following:

For 2 digits (e.g., ...31...), there is one triggering case: ...69.... This contributes a probability of . The cumulative probability becomes .

For 3 digits (e.g., ...314...), there are four triggering patterns: ...686..., ...687..., ...688..., and ...689.... Each occurs with probability , contributing , for a cumulative probability of .

For 4 digits (e.g., ...3141...), one relevant case is ...6859..., contributing , giving a cumulative probability of .

Continuing this process, we observe that the cumulative contribution from digit-triggering patterns converges to:

where denotes the th digit of pi. This expression represents the expected probability of a carry being triggered at any digit, under the assumption of uniform randomness and digit overlap behavior.

We now examine the role of preceding digits in enabling a carry to propagate across multiple digit positions. For a carry to propagate through digits before the target digit, each of the preceding preceding digits must be 9, since only the sum causes a carry to continue. Incorporating both the triggering and propagation behavior — that is, accounting for a triggering digit followed by a chain of 9s — we analyze typical patterns of carry propagation. Let denote the digit that either initiates a carry or receives one due to previous propagation. Then the probability patterns are as follows:

* ...1x... → carry propagates 1 digit on preceding digits with probability
* ...9x... → carry propagates 2 digits on preceding digits with probability
* ...99x..., → carry propagates 3 digits on preceding digits with probability
* → carry propagates k digits on preceding digits with probability

Thus, the probability of a carry chain propagating through exactly digits decrease exponentially with .

Since we aim to compute digits of pi, and the probability that any single digit triggers a carry chain of length is , the expected number of such events over all digits is:

To ensure that no carry-induced errors affect the computed digits, we impose the condition:

This inequality provides a practical bound on to guarantee accuracy across digits. Solving for , we obtain:

Taking logarithms base 10 on both sides:

Since must be an integer, we take the floor of the logarithm.

To simplify computations and provide a conservative overestimate, we approximate , giving:

This provides a concrete and practical bound for choosing sufficiently large to avoid carry-induced errors in the computation of digits of pi.

Now that the lower bound of k has been established, attention is turned to the inequality . Let , and consider the inequality in the form:

Since is taken to be positive in this context, we restrict the domain to:

To convert from radians to degrees, we multiply by ​:

However, this method is not yet practical, as the expression still contains the constant π. Therefore, an approximation technique is applied to eliminate π from the expression as follows.

Multiply both sides by :

The substitution of into the expression demonstrating that , therefore:

Thus, we obtain:

Terms that cannot be explicitly calculated are omitted from the analysis:

The expression is then rewritten in the form of a power of two.

Apply the inequality :

Ignore :

Define , the expression are obtained.

This inequality will be referred to as the Matching-Digit Approximation (MDA) expression.

**4.2 applying complex function for calculation.**

To compute , Euler's formula is applied:

To express this in degrees, convert radians to degrees using :

Then, the expression can be rewritten in terms of powers of , by noting that . Using the identity , and selecting the principal branch of the complex logarithm, where , we define:

This allows us to rewrite the expression as:

This expression is refer to as **Complex Degree Sine (CDS) expression**.

**4.3 applying CDS expression for calculation.**

From CDS expression:

Substitude :

Simplifying the expression:

It is clear that and thus:

Applying the CDS expression, the first 14,369,420 decimal digits of π were computed in 131.1 seconds on an Intel i7-4770 processor using a C implementation with MPFR and MPC libraries for high-precision arithmetic and POSIX threads for parallel computation of exponential terms. The digits were verified against the MIT π database (MIT SIPB, n.d.) and y-cruncher v0.8.6’s Chudnovsky output (Yee, n.d.), confirming exact agreement via character-by-character string comparison. Full implementation details and source code are in Appendix A.

**Chapter 5: Conclusion, Discussion, and Recommendations**

**5.1 Conclusion**

This research developed a three-step method to compute the value of π to a specified number of decimal places:

1. **Specify the desired precision**: Determine the number of decimal digits d to compute.

2. Estimate the number of digits affected by carry propagation using the inequality:

Then, compute , and determine the smallest integer satisfying:

3. Substitute the resulting value of into the following expression and evaluate it using high-precision numerical computation:

The result approximates pi with the desired number of correct decimal digits.

**5.2 Discussion**The key contributions of this work are:

* Matching-Digit Approximation (MDA): A systematic lower bound for angle selection based on Taylor expansion and probabilistic carry propagation modeling.
* Complex Degree Sine (CDS): A novel method for computing sin(α) using complex exponentials derived from Euler’s formula.
* Numerical Implementation in C: An algorithm implemented in C using MPFR and MPC libraries for high-precision arithmetic, parallelized with POSIX threads.  
  Unlike series-based algorithms like Chudnovsky’s, our trigonometric and geometric approach offers a conceptually elegant alternative with comparable asymptotic complexity .

Our method computed π to 14,369,420 digits in 131.1 seconds on an Intel i7-4770 processor. In contrast, y-cruncher’s optimized Chudnovsky implementation achieved the same precision in 1.492 seconds (start-to-end wall time) using 8 threads [Yee, n.d.]. Our method takes longer because it uses detailed sine calculations, but it offers a new geometric view and works well on basic computers with just two threads, unlike Chudnovsky's method, which uses Fast Fourier Transform (FFT) to speed up calculations on systems with multiple cores.  
However, our method has trade-offs:

* It is less efficient at extremely high precision due to the computational cost of evaluating sin(α) for very small α.
* It relies on statistical assumptions about carry propagation, which could introduce rare edge cases in certain digit sequences.
* It requires arbitrary-precision libraries to avoid floating-point underflow for small α.

Despite these challenges, this work shows that a geometric approach to π computation is feasible and accurate on standard desktop hardware, with proper carry propagation estimation and high-precision libraries.

**5.3 Recommendations**

**5.3.1 Optimize Sine Computation in C**

Improve the efficiency of computing in high-precision C implementations by:

* Exploring Chebyshev approximations or truncated Taylor series customized for small-angle input.
* Reducing the number of terms dynamically based on estimated error bounds.
* Avoiding unnecessary overhead by inlining operations or optimizing polynomial evaluation (e.g., Horner’s method).

Since α changes with each desired digit level and becomes very small, lookup tables are not applicable.

**5.3.2 Refine Carry Propagation Model**

This research assumes that the probability of a carry being triggered by a given digit is approximately . To improve this model:

* Perform empirical testing on real π digit sequences to validate or refine the assumption.
* Study actual carry behavior through simulated additions with various tail sequences.
* Investigate whether the carry probability varies across digit positions or correlates with π’s known statistical properties.

Improving this model can tighten the MDA bound and improve reliability for edge cases.

**5.3.3 Improve Arbitrary-Precision Integration in C**

While MPFR and MPC are effective, further improvements could include:

* Custom small-angle trigonometric functions for fixed patterns like , where expressions can be simplified in code.
* Implementing adaptive precision strategies: increase precision only where needed, reducing overhead.
* Direct integration of critical arithmetic routines instead of general-purpose APIs where predictable patterns exist.

**5.3.4 Enhance Parallelization Beyond Two Threads**

Currently, two threads are used to compute and . To improve performance:

* Parallelize the power series computation itself (e.g., computing separate terms in parallel).
* Explore OpenMP or POSIX thread pools for managing multiple tasks within CDS evaluation.
* Evaluate potential for GPU acceleration for high-precision arithmetic, although it requires careful management of rounding.

**5.3.5 Develop a Configurable Command-Line Interface**

Turn the implementation into a modular CLI tool that supports:

* User-defined digit targets.
* Output format selection (plain text, verbose, raw binary).
* Adjustable thread counts and precision settings.
* Progress tracking and estimated runtime display.

Such a tool could aid both research users and educators interested in π computation.

**References**

Abramowitz, M., & Stegun, I. A. (Eds.). (1964). *Handbook of mathematical functions with formulas, graphs, and mathematical tables* (Vol. 55). U.S. Government Printing Office. <https://personal.math.ubc.ca/~cbm/aands/abramowitz_and_stegun.pdf>

Agarwal, R. P., Agarwal, H., & Sen, S. K. (2013). Birth, growth and computation of pi to ten trillion digits. Retrieved from <https://link.springer.com/content/pdf/10.1186/1687-1847-2013-100.pdf>

Arndt, J. (2010). Matters computational. Retrieved from <https://jjj.de/fxt/fxtbook.pdf>

Badger, R. (1990). Lazzarini’s lucky approximation. The Mathematical Association of America. Retrieved from <http://www.maa.org/sites/default/files/pdf/upload_library/22/Allendoerfer/1995/Badger.pdf>

Bailey, D. H. (1997). The quest for pi. Retrieved from <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/pi-quest.pdf>

Borwein, J. M. (2011). The life of pi. Retrieved from <https://carmamaths.org/resources/jon/piday-16-sm.pdf>

Chan, H. H. (2012). Rational analogues of Ramanujan’s series for 1/π. Retrieved from [https://web.archive.org/web/20191219004315/http://pdfs.semanticscholar.org/6810/c8ac1ad01821a504e66f5d8665aeeae93cab.pdf](https://web.archive.org/web/20191219004315/http:/pdfs.semanticscholar.org/6810/c8ac1ad01821a504e66f5d8665aeeae93cab.pdf)

Complex Analysis. (n.d.). Roots of complex number. Retrieved from <https://complex-analysis.com/content/roots_complex_numbers.html>

Damini, D. B., & Dhar, A. (2020). How Archimedes showed that π is approximately equal to 22/7. arXiv. Retrieved from <https://arxiv.org/abs/2008.07995>

Enge, A., Hanrot, G., & Théobald, C. (2012). *MPC – A library for multiprecision complex arithmetic with exact rounding* (Version 1.0.2) [Software]. GNU Project. <https://www.multiprecision.org/mpc/>

Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., & Zimmermann, P. (2007). *MPFR: A multiple-precision binary floating-point library with correct rounding*. ACM Transactions on Mathematical Software, 33(2), 1–15. https://doi.org/10.1145/1236463.1236468

Galperin, G. (2003). Playing pool with π. Retrieved from <https://www.maths.tcd.ie/~lebed/Galperin.%20Playing%20pool%20with%20pi.pdf>

GNU Project. (n.d.-a). *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/

Haber, H. (2011). Taylor series expansions. Retrieved from <http://scipp.ucsc.edu/~haber/ph116A/taylor11.pdf>

IEEE. (2018). *The Open Group Base Specifications Issue 7, 2018 edition* (POSIX.1-2017). IEEE Std 1003.1-2017. https://pubs.opengroup.org/onlinepubs/9699919799/

Johansson, F., & contributors. (2007–present). mpmath. Retrieved from <https://mpmath.org/>

Limit and continuity. (2014). Retrieved from <https://anakkamatee.files.wordpress.com/2014/08/ch1.pdf>

Mathai, A. M. (1999). An introduction to geometrical probability. Gordon and Breach Science Publishers.

McKeeman, B. (2016). The computation of pi by Archimedes. Retrieved from <https://www.mathworks.com/matlabcentral/fileexchange/29504-the-computation-of-pi-by-archimedes>

Milla, L. (2021). A detailed proof of the Chudnovsky formula with means of basic complex analysis. arXiv. Retrieved from <https://arxiv.org/pdf/1809.00533>

MIT SIPB. (n.d.). *The first billion digits of pi*. Massachusetts Institute of Technology. Retrieved from <https://stuff.mit.edu/afs/sipb/contrib/pi/pi-billion.txt>

mpmath.org. (2023). Trigonometric function. Retrieved from <https://mpmath.org/doc/current/functions/trigonometric.html>

Needham, T. (1997). *Visual Complex Analysis*. Oxford University Press

O’Connor, J. J., & Robertson, E. F. (2000). Madhava of Sangamagramma. Retrieved from <https://mathshistory.st-andrews.ac.uk/Biographies/Madhava/>

O’Connor, J. J., & Robertson, E. F. (2003). Liu Hui. Retrieved from <https://mathshistory.st-andrews.ac.uk/Biographies/Liu_Hui/>

3Blue1Brown. (2019, February 8). The most unexpected answer to a counting puzzle [Video]. YouTube. Retrieved from <https://www.youtube.com/watch?v=HEfHFsfGXjs>

Trigonometry of right triangle. (2012). Retrieved from <https://books.google.co.th/books?id=uJqaBAAAQBAJ&pg=PA448>

Wästlund, J. (2010). Summing inverse squares by Euclidean geometry. Retrieved from <http://www.math.chalmers.se/~wastlund/Cosmic.pdf>

Xu, Z. (2022). A new form of the Machin-like formula for π by iteration with increasing integers. arXiv. Retrieved from <https://arxiv.org/pdf/2108.07718.pdf>

Yee, A. (n.d.). y-cruncher: A multi-threaded pi and constants calculator. Retrieved from http://www.numberworld.org/y-cruncher/

**Appendix A: C Code Used for π Calculation**

The following C program computes π using a trigonometric approach based on the Matching-Digit Approximation (MDA) and Complex Degree Sine (CDS) expressions. It employs MPFR and MPC libraries for high-precision arithmetic and POSIX threads for parallelizing complex exponential computations.

**Usage**:  
Compile with:  
gcc compute\_pi.c -lmpfr -lmpc -lpthread -lm -o compute\_pi

The program outputs π to pi\_result.txt, including runtime. Requires MPFR and MPC installed.

#define \_POSIX\_C\_SOURCE 199309L

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <pthread.h>

#include <mpc.h>

#include <mpfr.h>

#include <time.h>

#define THREAD\_COUNT 2

// Thread data structure

typedef struct {

mpc\_t result;

mpfr\_t angle;

int sign; // +1 or -1

int prec;

} ThreadData;

// Exponential calculation thread

void\* compute\_exponential(void\* arg) {

ThreadData\* data = (ThreadData\*)arg;

mpc\_t j;

mpc\_init2(j, data->prec);

mpc\_set\_ui\_ui(j, 0, 1, MPC\_RNDNN); // j = i

mpfr\_t temp\_angle;

mpfr\_init2(temp\_angle, data->prec);

if (data->sign == -1)

mpfr\_neg(temp\_angle, data->angle, MPFR\_RNDN);

else

mpfr\_set(temp\_angle, data->angle, MPFR\_RNDN);

mpc\_pow\_fr(data->result, j, temp\_angle, MPC\_RNDNN);

mpc\_clear(j);

mpfr\_clear(temp\_angle);

pthread\_exit(NULL);

}

int main() {

// High-resolution timer start

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC, &start);

// === Step 1: Get digit input from user ===

int digits;

printf("Enter number of digits you want: ");

scanf("%d", &digits);

// === Step 2: Compute m from digits using custom formula ===

int k = (int)(log10(digits \* 0.32)) + 2;

int n = digits + k;

int m = (5 \* n + 1 + 2) / 3; // ceiling((5n+1)/3)

printf("Using d = %d, k = %d, n = %d ⇒ m = %d\n", digits, k, n, m);

// === Step 3: Compute required precision ===

double bits\_estimate = ceil(digits \* log2(10));

int prec\_bits = (int)bits\_estimate;

mpfr\_set\_default\_prec(prec\_bits);

// === Step 4: Setup angles and values ===

mpfr\_t angle, two\_pow\_m\_minus\_1, real\_part, final\_result;

mpfr\_inits2(prec\_bits, angle, two\_pow\_m\_minus\_1, real\_part, final\_result, NULL);

mpfr\_ui\_pow\_ui(angle, 2, m - 1, MPFR\_RNDN); // angle = 2^(m-1)

mpfr\_ui\_div(angle, 1, angle, MPFR\_RNDN); // angle = 1 / 2^(m-1)

mpfr\_ui\_pow\_ui(two\_pow\_m\_minus\_1, 2, m - 1, MPFR\_RNDN); // 2^(m-1)

// === Step 5: Launch threads to compute exp(±i \* angle) ===

pthread\_t threads[THREAD\_COUNT];

ThreadData data[THREAD\_COUNT];

for (int i = 0; i < THREAD\_COUNT; i++) {

mpc\_init2(data[i].result, prec\_bits);

mpfr\_init2(data[i].angle, prec\_bits);

mpfr\_set(data[i].angle, angle, MPFR\_RNDN);

data[i].sign = (i == 0) ? +1 : -1;

data[i].prec = prec\_bits;

pthread\_create(&threads[i], NULL, compute\_exponential, &data[i]);

}

for (int i = 0; i < THREAD\_COUNT; i++) {

pthread\_join(threads[i], NULL);

}

// === Step 6: Combine results to get final pi approximation ===

mpc\_t numerator, result, j;

mpc\_init2(numerator, prec\_bits);

mpc\_init2(result, prec\_bits);

mpc\_init2(j, prec\_bits);

mpc\_sub(numerator, data[0].result, data[1].result, MPC\_RNDNN); // exp(iθ) - exp(-iθ)

mpc\_set\_ui\_ui(j, 0, 1, MPC\_RNDNN); // j = i

mpc\_div(result, numerator, j, MPC\_RNDNN); // (exp(iθ) - exp(-iθ)) / i

mpc\_real(real\_part, result, MPFR\_RNDN); // Real part of result

mpfr\_mul(final\_result, real\_part, two\_pow\_m\_minus\_1, MPFR\_RNDN);// × 2^(m - 1)

// === Step 7: Output to file ===

FILE\* f = fopen("pi\_result.txt", "w");

if (f) {

mpfr\_out\_str(f, 10, 0, final\_result, MPFR\_RNDN);

fprintf(f, "\n");

clock\_gettime(CLOCK\_MONOTONIC, &end);

double elapsed = (end.tv\_sec - start.tv\_sec) +

(end.tv\_nsec - start.tv\_nsec) / 1e9;

fprintf(f, "Execution Time: %.6f seconds\n", elapsed);

fclose(f);

}

// === Step 8: Cleanup ===

for (int i = 0; i < THREAD\_COUNT; i++) {

mpc\_clear(data[i].result);

mpfr\_clear(data[i].angle);

}

mpc\_clear(j);

mpc\_clear(numerator);

mpc\_clear(result);

mpfr\_clears(angle, two\_pow\_m\_minus\_1, real\_part, final\_result, NULL);

return 0;

}