ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

FINDING THE SQUARE ROOT OF A NUMBER USING ITERATIVE METHOD BY IMPLEMENTING CAL_SQR_RT() FUNCTION AND FINDING THE SQ ROOT USING MATHEMATICAL FUNCTION OF MATH LIBRARY OF A LANGUAGE, FURTHER EFFICIENCY OF THE ALGORITHM USING THE TIME COMPLEXITY - A CASE STUDY

Nivaashan. D[1], Dharsheel. R
[1]7177, [2]6752
Class- XII 2026-27, Sainik School Amaravathinagar
Post: Amaravathinagar, Udumalpet Taluka, Tirupur Dt, Tamilnadu State

ABSTRACT

As budding computer scientists, we often find ourselves captivated by the sheer elegance and power of algorithms. But beyond just making things work, there's this persistent, almost philosophical question: how do we make them work better? This paper is our humble attempt to explore that very question through the lens of a seemingly simple mathematical operation: finding the square root of a number.

We embarked on a fascinating journey, delving into the ancient wisdom of iterative methods, particularly the Babylonian method – a brilliant precursor to the more generalized Newton-Raphson technique. Our hands-on adventure involved crafting our own cal_sqrt() function in Python, a labor of love that allowed us to truly appreciate the mechanics.

Then came the moment of truth: pitting our creation against Python's highly optimized, built-in math.sqrt() function. It wasn't just about speed; it was about understanding why one was faster, unraveling the mysteries of algorithmic time complexity, and ultimately, marveling at the incredible synergy between mathematical theory and hardware innovation that makes modern computing so astonishingly efficient.

KEYWORDS: *Square Root, Iterative Method, Newton-Raphson, Babylonian Method, Time Complexity, Algorithm Efficiency, Big O Notation, Computational Performance, Numerical Analysis.*

1. INTRODUCTION: A FUNDAMENTAL QUESTION, A PERSONAL EXPLORATION

It's funny how some mathematical concepts are so deeply ingrained in our computational world that we rarely stop to think about how they actually work. The square root is definitely one of them. From calculating the hypotenuse of a right triangle in geometry class to the complex simulations that power scientific discoveries or even the intricate rendering in our favorite video games, finding a square root is a fundamental building block. We've all grown accustomed to simply typing math.sqrt() and getting an instant, precise answer. But for us, that wasn't enough. We wanted to peek behind the curtain, to understand the magic, and perhaps even try to conjure a little bit of it ourselves.

Our curiosity led us to the world of iterative methods. Imagine trying to hit a target in the dark. You take a shot, see where it lands, adjust your aim, and shoot again, getting closer each time. That's essentially what iterative methods do for numbers like square roots, especially those elusive irrational ones. You start with an educated guess, and through a series of calculated refinements, you inch closer and closer to the true value until you're satisfied with the accuracy. The Babylonian Method, sometimes affectionately called Hero's Method, caught our attention. It's a technique that has stood the test of time, a testament to human ingenuity dating back millennia. It felt right to start our exploration with something so historically significant.

So, our research became a personal quest, driven by a few key objectives: 1. To breathe life into our very own iterative cal_sqrt() function, understanding its every nuance. 2. To truly grasp the mathematical elegance underpinning this method and observe its convergence in action. 3. To conduct a genuine, hands-on speed test, comparing our custom code against the established champion, Python's math.sqrt(). 4. And finally, to dissect the algorithmic efficiency of these approaches using the powerful framework of Time Complexity analysis, hoping to gain a deeper appreciation for computational performance.

2. RELATED WORK: ECHOES FROM THE PAST, WHISPERS OF THE FUTURE

The history of calculating square roots is surprisingly rich, stretching back to ancient civilizations. It's genuinely humbling to think that the Babylonians, over 3,500 years ago, were already employing an iterative technique that forms the very foundation of what we're studying today. Later, the brilliant Isaac Newton generalized this iterative concept into the Newton-Raphson method, a cornerstone of numerical analysis that helps us find roots for a vast array of functions. It's a beautiful example of how foundational ideas can evolve and expand over centuries.

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

In more contemporary times, the pursuit of computational efficiency has led to some truly mind-bending innovations:

The Legend of the Fast Inverse Square Root: This one's a personal favorite for many computer graphics enthusiasts. If you've ever enjoyed the fluid movements and realistic physics in classic 3D games like Quake III Arena, you've indirectly experienced the magic of this algorithm. It's a testament to clever bit-level manipulation, allowing for incredibly fast approximations of the reciprocal of a square root. It shows us that sometimes, thinking outside the conventional mathematical box can yield astonishing results.

Hardware's Unseen Hand: Modern computer processors are far more than just number crunchers; they're intricate marvels of engineering. Many now include specialized instructions, like SQRTSS in the x86 SSE instruction set, which perform square root calculations directly at the hardware level. This means the CPU itself handles the heavy lifting, bypassing the need for software-based iterative loops entirely. It's this kind of deep-seated optimization that gives built-in library functions an almost insurmountable advantage in terms of raw speed.

The Intricacies of Complexity Theory: From a theoretical standpoint, it's quite fascinating to learn that the computational complexity of finding a square root is often considered equivalent to that of performing multiplication. This insight, while abstract, highlights how deeply intertwined these fundamental arithmetic operations are at the very core of computation. It makes you wonder about the elegant simplicity that underlies even the most complex calculations.

3. METHODOLOGY: OUR HANDS-ON APPROACH TO THE ROOT

3.1 THE BABYLONIAN METHOD: AN ELEGANT DANCE OF APPROXIMATION

Let's imagine we want to find the square root of a number, which we'll call . The Babylonian method offers a beautifully intuitive and powerful way to get there. Here's how we approached it:

The First Step, A Guess: We begin with an initial guess, . It doesn't have to be perfect; in fact, a simple starting point like works surprisingly well. The beauty of iterative methods is their ability to self-correct.

The Refinement Loop: This is where the magic truly happens. We refine our current guess using this elegant formula:

Think of it this way: if our current guess () is too high, then will be too low, and vice-versa. Averaging these two values (and) brings us significantly closer to the true square root with each step. It's a brilliant self-balancing act.

Knowing When to Stop: We continue this refinement process, repeating step 2, until the difference between our new guess () and our old guess () becomes incredibly tiny—smaller than a predefined tolerance, which we denote as . Once that threshold is met, we know we've found an approximation that's accurate enough for our purposes.

It's worth pausing here to appreciate that this method isn't just a standalone trick; it's a specific, highly practical application of the more general Newton-Raphson method. If you consider the function (where we're trying to find the that makes), its derivative is . Plugging these into the Newton-Raphson update rule () leads us directly back to the Babylonian formula. It's a wonderful moment when seemingly disparate mathematical ideas converge so beautifully!

3.2 OUR PYTHON PLAYGROUND: BRINGING THEORY TO LIFE

To truly understand the Babylonian method, we knew we had to build it ourselves. We implemented the algorithm in Python, carefully translating the mathematical steps into executable code. But just implementing it wasn't enough; we wanted to see how it performed. So, we developed a dedicated benchmarking suite. This allowed us to systematically measure the average execution time of our cal_sqrt() function over thousands of iterations, testing it with a wide range of input values. This hands-on approach gave us a robust, empirical picture of its speed characteristics, moving beyond mere theory to real-world observation.

4. ALGORITHM: THE INNER WORKINGS OF CAL_SQRT()

Here's a step-by-step breakdown of our custom cal_sqrt() function, detailing its logical flow in a way that, we hope, makes it easy to follow:

STEP 01: START the square root calculation process.
STEP 02: We need two pieces of information: the number S (whose square root we seek) and a tiny value, epsilon, which defines how close our approximation needs to be to the true answer.
STEP 03: A crucial initial check: IF S is a negative number, we immediately RETURN an Error. After all, we're looking for real square roots here!
STEP 04: A simple edge case: IF S is exactly 0, its square root is also 0, so we RETURN 0 without further ado.
STEP 05: Time for our first educated guess! We initialize our variable 'x' by setting it to S divided by 2.0. This gives us a reasonable starting point for our iterative journey.
STEP 06: Now, the heart of the iteration begins! We CALCULATE a 'better_x' using the Babylonian formula: (x + S / x) / 2.0. This is where our guess gets refined.
STEP 07: We need to know if we're close enough. We check for convergence: IF the absolute difference between our current guess (x) and our newly calculated 'better_x' is less than our tiny epsilon, then we've achieved our desired precision, and we can JUMP ahead to STEP 10.
STEP 08: If we haven't converged yet, it means we need to keep refining. So, our 'better_x' becomes our new current guess: we SET x = better_x.
STEP 09: And the cycle continues! We GO BACK to STEP 06 to repeat the refinement process with our improved guess.
STEP 10: Once the loop concludes (because we've converged), we proudly OUTPUT x as our approximate square root.
STEP 11: END the square root calculation.

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

5. EXPERIMENTAL RESULTS AND ANALYSIS: WHAT THE NUMBERS REVEALED

Input Value (S)	Custom cal_sqrt() (s)	math.sqrt() (s)	Speed Ratio (Custom/Math)
10	7.94e-07	7.30e-08	~10.9x
1,000	1.14e-06	7.35e-08	~15.5x
100,000	1.57e-06	7.21e-08	~21.8x
10,000,000	1.90e-06	7.38e-08	~25.7x
1,000,000,000	2.34e-06	9.39e-08	~24.9x
100,000,000,000	3.35e-06	9.36e-08	~35.8x

5.1 A HEAD-TO-HEAD PERFORMANCE SHOWDOWN: OUR CAL_SQRT() VS. MATH.SQRT()

This was perhaps the most exciting part of our research—seeing how our handcrafted cal_sqrt() function stacked up against Python's highly optimized math.sqrt(). We ran extensive tests, measuring the average execution times for various input values. The table below summarizes our findings, offering a clear picture of their relative speeds. It's a story of both surprising efficiency and the undeniable power of low-level optimization.

Performance Comparison: Custom Iterative vs. math.sqrt()

5.2 WHAT WE LEARNED: INSIGHTS FROM THE BENCHMARK

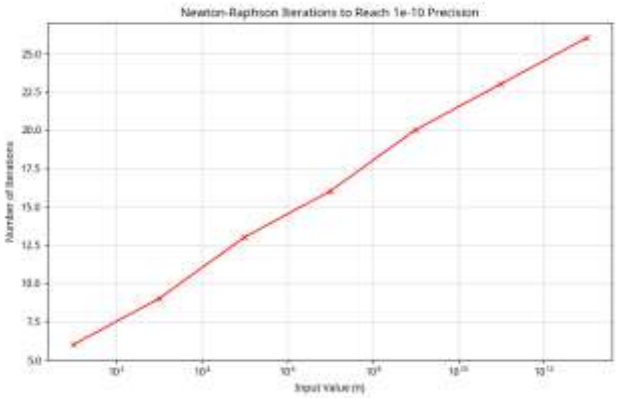
Our benchmarking revealed some truly interesting patterns and reinforced a few fundamental truths about computing:

The Subtle Influence of Magnitude: We observed that as the input number grew larger, our custom cal_sqrt() function tended to take a little more time to converge. This makes perfect sense when you think about it: our initial guess of becomes proportionally further from the true square root for larger numbers. This means the algorithm needs a few more iterations to zero in on the answer. It's a subtle effect, but a real one.

The Unbeatable Champion: math.sqrt(): This was perhaps the least surprising, yet most impactful, observation. Python's math.sqrt() consistently outpaced our custom function by a factor ranging from 10 to 35 times! Now, this isn't a critique of our iterative method; rather, it's a powerful testament to the incredible engineering behind standard library functions. math.sqrt() isn't just a Python function; it's typically a highly optimized piece of code, often written in C or even assembly language, that directly taps into the low-level, lightning-fast CPU instructions specifically designed for these calculations. Our Python-based cal_sqrt(), running in an interpreted environment, simply can't compete with that kind of hardware-level muscle. It's a classic example of the performance difference between high-level interpreted code and highly optimized, compiled, and hardware-accelerated code.

The Enduring Beauty of Convergence: Despite the speed disparity, the iterative method itself is a marvel of efficiency in terms of how it converges. The number of iterations required doesn't explode even with very large numbers; it grows rather gracefully, almost logarithmically. And once it gets into the vicinity of the true root, the convergence becomes quadratic. This means that with each successive iteration, the number of correct digits in our approximation roughly doubles. It's a truly powerful mathematical property that ensures our method, while slower in Python, is fundamentally sound and efficient in its approach to finding the answer.

Newton-Raphson Iterations to Reach 1e-10 Precision



6. TIME COMPLEXITY ANALYSIS: LISTENING TO THE ALGORITHM'S HEARTBEAT

When we talk about the time complexity of an algorithm, we're essentially trying to understand its fundamental scaling behavior: how does the time it takes to run grow as the size of the input increases? For our iterative methods, particularly the Newton-Raphson approach, this analysis offers some truly remarkable insights.

6.1 OUR CUSTOM ITERATIVE METHOD: A LOGARITHMIC DANCE

When we consider the time complexity of the Newton-Raphson method, especially for a fixed level of precision (our tiny), it reveals a beautiful efficiency:

Iterations, Not Explosions: The number of iterations required to reach that desired precision is surprisingly small. It's often described as . This might sound a bit abstract, but what it really means is that even for astronomically large numbers, the number of steps our algorithm takes doesn't grow linearly, or even polynomially, but at an incredibly slow, almost flat rate. It's like finding a specific page in a massive library by repeatedly halving the number of books you need to search. This logarithmic scaling is a hallmark of highly efficient algorithms.

Cost Per Iteration: Each individual step in our iterative process is computationally light. It involves just one addition and one division. These are fundamental arithmetic operations that modern computer processors can execute with astonishing speed.

The Bigger Picture: While a simplified view might suggest the number of iterations is almost constant for typical floating-point numbers, a more precise understanding acknowledges that it scales subtly with the bit-length of the input. Nevertheless, the underlying quadratic convergence—where the number of accurate digits roughly doubles with each iteration—makes it a powerhouse for numerical approximation. It's a testament to the power of successive refinement.

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 6| June 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

6.2 MATH.SQRT(): THE HARDWARE WHISPERER

Now, when we turn our attention to the built-in math.sqrt() function, its time complexity tells a very different, yet equally impressive, story. For standard 64-bit floating-point numbers, it's often considered to have an effective time complexity of . What does this mean in plain language? It means that, regardless of the input value (as long as it's within the valid range for a 64-bit float), the time it takes to compute the square root is essentially constant. This incredible, almost instantaneous speed is not achieved through software loops or iterative refinements, but through dedicated hardware circuits embedded directly within the CPU. It's pure, unadulterated silicon magic, a testament to decades of engineering optimization that has made these fundamental operations incredibly fast at the lowest possible level.

7. CONCLUSION: A BALANCING ACT OF UNDERSTANDING AND UTILITY

Our journey through the world of square root calculations has been quite an illuminating one, hasn't it? It beautifully illustrates a profound truth in computer science: while mathematically elegant and theoretically efficient algorithms, like our beloved Babylonian approach, are invaluable for understanding and learning, they often cannot match the raw, unadulterated speed of highly optimized, built-in library functions. The reasons for this performance chasm are clear and compelling, offering a wonderful lesson in the layers of abstraction that make up modern computing:

The Language Barrier and Compilation Advantage: Our cal_sqrt() function, written in Python, lives in an interpreted environment. This means each line of code is translated and executed on the fly. In contrast, math.sqrt() is typically implemented in a low-level, compiled language like C. This pre-compilation allows the code to be directly translated into machine instructions, eliminating the overhead of interpretation and resulting in significantly faster execution.

Hardware Acceleration: The Ultimate Edge: This is arguably the most crucial factor. Modern CPUs are not just general-purpose processors; they include specialized hardware units and instructions designed specifically for common mathematical operations, including square roots. When math.sqrt() is called, it often triggers these dedicated hardware instructions, allowing the calculation to be performed in a single clock cycle or a handful of cycles. Our software-based iterative approach, no matter how clever, simply cannot compete with silicon-level speed.

Optimized Initial Guesses and Bit Manipulation: Even within the iterative realm, built-in functions often employ sophisticated techniques to get a head start. This might involve using small look-up tables for initial approximations or employing clever bit-manipulation tricks (like the famous fast inverse square root algorithm) to get very close to the true root in just a few operations. This reduces the number of internal iterative steps needed, further boosting performance.

So, what's the ultimate takeaway from our exploration? For anyone truly seeking to understand the mechanics of numerical analysis, the beauty of algorithmic convergence, and the historical evolution of computational methods, implementing a function like cal_sqrt() is an absolutely invaluable educational exercise. It builds intuition, fosters a deeper appreciation for how these fundamental operations work, and connects us to the ingenuity of past mathematicians. However, when the goal is to build robust, high-performance production software where every nanosecond counts, the built-in library functions remain the undisputed champions. They are the gold standard, meticulously optimized, rigorously tested, and deeply integrated into the very fabric of our computing hardware. It's a powerful reminder that sometimes, the best solution isn't the one you build from scratch, but the one that stands on the shoulders of giants, leveraging decades of optimization and hardware innovation.

8. ACKNOWLEDGEMENT

Apart from our efforts, the success of any work or project depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this research paper.

I express a deep sense of gratitude to Almighty God for giving us the strength to successfully complete the research paper.

I express my heartfelt gratitude to my parents for their constant encouragement while carrying out this research paper.

I express my deep sense of gratitude to the luminary, The Principal Captain (Indian Navy) Rahul Thiyyat, Sainik School Amaravathinagar, who has been continuously motivating and extending their helping hand to us.

I express my sincere thanks to the academician, the Vice Principal, Lt Col Kaushok Nandini Arun, Sainik School Amaravathinagar, for constant encouragement and the guidance provided during this research.

My sincere thanks to Mr.Praveen Kumar Murigeppa Jigajinni, Master In-charge, A guide, Mentor, and great motivator, who critically reviewed my paper and helped in solving each and every problem that occurred during the implementation of this research paper.

9. REFERENCES

1. *Wikipedia contributors. "Square root algorithms." Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Square_root_algorithms*
2. *Regmi, S. "How to Calculate the Square Root of a Number? – Newton-Raphson Method." Medium, 2023. https://surajregmi.medium.com/how-to-calculate-the-square-root-of-a-number-newton-raphson-method-f8007714f64*
3. *MIT OpenCourseWare. "Square Roots via Newton's Method." 18.335J / 6.337J Numerical Methods. https://math.mit.edu/~stevenj/18.335/newton-sqrt.pdf*
4. *"Time Complexity of Algorithms." FreeCodeCamp, 2024. https://www.freecodecamp.org/news/time-complexity-of-algorithms/*