INTERNATIONAL JOURNAL OF LATEST TECHNOLOGY IN ENGINEERING, MANAGEMENT & APPLIED SCIENCE (IJLTEMAS)

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

Bridging the Past and Present: Implementing Ancient Indian Mathematical Techniques Using Python

Pearly P Kartha*, Nikumbha Neha R

Department of Mathematics, Dr. D. Y. Patil Arts, Commerce and Science College, Pimpri, Pune, Maharashtra, India

DOI: https://doi.org/10.51583/IJLTEMAS.2025.1413SP035

Received: 26 June 2025; Accepted: 30 June 2025; Published: 25 October 2025

Abstract: Ancient Indian Mathematics has made significant contributions to arithmetic, trigonometry and algebra, many of which continue to influence modern computational methods. Techniques such as Bhaskara I's sine approximation and Vedic multiplication were designed for rapid mental calculations and have inspired the development of various modern algorithms. This paper explores the implementation and computational performance of two ancient Indian mathematical techniques—Vedic Multiplication and Bhaskara I's Sine Approximation using Python. Their efficiencies are evaluated against modern numerical libraries like NumPy in terms of execution time, computational complexity and accuracy. The findings show that some ancient techniques are highly efficient for specific tasks even today. This study connects traditional mathematical knowledge with modern computational methods, emphasizing the lasting impact of Indian mathematical innovations.

Keywords: Bhaskara I's Sine Approximation, Vedic Multiplication, Mean Absolute Error, Mean Squared Error, Computational Efficiency, Absolute Error Analysis, Algorithm Development.

I. Introduction

Mathematics has always been a fundamental part of human development influencing everything from engineering to computer science. Indian mathematicians have significantly contributed in early numerical techniques by developing methods that simplified calculations and improved efficiency. The Vedic mathematical system, based on ancient Sanskrit texts, introduced quick mental calculation tricks that made complex arithmetic much easier. Bhaskara I, in the 7th century, came up with a smart way to approximate sine values, which was an important step in trigonometry. While these ancient methods contributed significantly in their time, today's numerical computing depends on modern algorithms and programming tools like Python, particularly with libraries such as NumPy.

Even though these modern methods are optimized for computational efficiency, yet it remains unclear whether ancient Indian techniques can still offer practical advantages in certain computational scenarios. With this study, we aim to bridge the gap by implementing and analysing the performance of two ancient mathematical techniques: Vedic Multiplication and Bhaskara I's Sine Approximation using Python.

The primary objectives of this study are:

1. To implement these ancient techniques in Python.
2. To evaluate their execution time and computational complexity using NumPy.
3. To examine their applications in modern computing methods

This study explores whether these ancient techniques still hold computational advantages for specific tasks through a comparative performance analysis. By connecting historical mathematics with modern computing, it highlights the enduring relevance of ancient Indian mathematical knowledge.

II. Literature Review

Ancient Indian mathematics has had a lasting influence on modern computation, with many techniques being useful even today. Historians like Bose [1] and Datta & Singh [2] have explored the evolution of Indian mathematical concepts, from basic numeration systems to advanced algebraic techniques. Their work highlights how Indian mathematicians were way ahead of their time in developing problem-solving methods which were efficient.

Vedic Mathematics and Fast Calculations

Vedic mathematics is well known for its sixteen sutras, which simplify calculations significantly. Williams [3] discusses how these techniques make mental arithmetic calculations faster and more intuitive. Some researchers, like Plofker [8], have looked into the historical context of these methods and their influence on early algorithmic thinking. In recent years, comparisons have been made between these ancient techniques and modern numerical libraries like NumPy, with some studies suggesting that Vedic multiplication, in particular, still holds up well for certain tasks [4].

Bhaskara I's Sine Approximation

Bhaskara I is famous for his sine approximation formula, which is surprisingly accurate considering the tools that was available at

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

that time. His contributions to trigonometry and astronomy have been discussed in depth by various scholars [8], [10]. More recently, researchers have tested Bhaskara's method using Python's SciPy library and compared its efficiency to modern numerical methods [6], [14]. Gupta [11] has examined the accuracy of ancient interpolation techniques and how they relate to modern computational approaches.

Comparing Ancient Techniques with Modern Libraries

With the rise of high-performance computing, researchers have been able to evaluate traditional mathematical techniques against modern computational tools. Libraries like NumPy and SciPy have been widely studied for their efficiency in handling large-scale numerical computations [4], [6]. Van der Walt et al. [13] and Virtanen et al. [14] provide insights into how these libraries optimize calculations. Stillwell [5] and Knuth [15] have also explored the mathematical foundations of modern algorithms, which is useful when assessing ancient methods.

Overall, this review highlights how ancient Indian mathematical techniques continue to be relevant in today's computational world. By testing these methods with Python-based tools, we can better understand their strengths and limitations in modern applications.

III. Methodology

Vedic Multiplication Technique

Vedic multiplication is an ancient Indian technique designed for quick mental math calculations. It works by breaking large numbers into smaller parts, calculating partial results, and then combining them to get the final answer. This method is particularly useful for certain arithmetic operations and has even influenced some modern computational approaches.

Implementation in Python:

To evaluate the computational efficiency of Vedic multiplication, we implemented the method using Python and compared its performance against NumPy's built-in multiplication function.

Steps of Implementation:

1. **Breaking Down the Digits**: We split the given numbers into smaller parts using the divmod() function.
2. **Performing Partial Multiplications**: We then calculate the intermediate products.
3. **Combining the Results**: We then reconstruct the final result using place value adjustments.

Python Code Implementation:

The following figure presents the Python implementation of the Vedic multiplication technique.

```
[9]: def vedic_multiplication(x, y):
         x1, x0 = divmod(x, 10)
         y1, y0 = divmod(y, 10)
         step1 = x0 * y0
         step2 = (x0 * y1) + (x1 * y0)
         step3 = x1 * y1
         result = (step3 * 100) + (step2 * 10) + step1
         return result
```

Fig 3.1: Python Implementation of Vedic Multiplication

Bhaskara I's Sine Approximation

Bhaskara I proposed an approximation formula for computing sine values, which provided a simplified yet effective way to estimate sine values for small angles. His formula is given as:

$sin(\theta) = \frac{4\theta(180-\theta)}{(40500-\theta(180-\theta))}$; Where θ is in degrees

Implementation in Python:

To analyze the accuracy of Bhaskara I's sine approximation, we implemented his formula in Python and compared it with NumPy's "sin ()" function.

Steps of Implementation:

1. **Using Bhaskara I's Sine formula:**

- Since modern computation requires handling negative angles, so we extend Bhaskara I's formula using the symmetry

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

property:

$$\sin(-\theta) = -\sin(\theta)$$

- This is achieved using np.abs() to apply the formula to absolute values and np.vectorize() for efficient computation over arrays.
- We avoid division by zero using a small offset (epsilon).

2. **Computing Values:**

- We apply the formula to a range of angles, including negative values for a more detailed comparison.

3. **Comparing with modern computation:**

- We evaluate how well Bhaskara I's method approximates NumPy's sine function.

Python Code Implementation:

The following python function implements Bhaskara I's sine approximation, extending it for negative angles and comparing it with NumPy's sine function.

```
•[50]: import numpy as np
       import pandas as pd
       import matplotlib.pyplot as plt

       def bhaskara1_sine(theta):
           epsilon = 1e-10 # Small value to avoid zero div
           return (4*theta*(180-theta))
                    /(40500-theta*(180-theta) + epsilon)

       # Generate angles from -180° to 180°
       angles = np.linspace(-180,180,100)

       # Calculate Bhaskara I's Sine approximation
       # Use absolute values for symmetry
       bhaskara1_values = np.vectorize(bhaskara1_sine)(np.abs(angles))


       # Apply the sine symmetry sin(-theta) = -sin(theta)
       bhaskara1_values[angles < 0] *= -1
       # Compute NumPy Sine values
       numpy_values = np.sin(np.radians(angles))
```

Fig 3.2 (a) Python Implementation of Bhaskara I's Sine Approximation

In the below python code 3.2 (b), we have taken positive angles (**test_angles = [0, 30, 45, 60, 90, 120, 150, 180]**) for computation and computed the sine values at these angles using both Bhaskara I's approximation and NumPy sin() function.

We then calculated the absolute error and checked the maximum and minimum error points

```
[52]: test_angles = [0,30,45,60,90,120,150,180]
      # Compute Bhaskara I's sine values and NumPy sine values for the test angles
      bhaskara1_val = [bhaskara1_sine(theta) for theta in test_angles]
      numpy_val = [np.sin(np.radians(theta)) for theta in test_angles]

      # Compute absolute error
      absolute_error = np.abs(np.array(numpy_val) - np.array(bhaskara1_val))

      # Find the max and min error points
      max_error_idx = np.argmax(absolute_error)
      min_error_idx = np.argmin(absolute_error)
      max_error_angle = test_angles[max_error_idx]
      min_error_angle = test_angles[min_error_idx]
```

Fig 3.2 (b) Absolute Error Analysis

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

IV. Result and Analysis

Vedic Multiplication:

The 'timeit' module in python was used for time measurements for different input sizes. The efficiency of Vedic Multiplication and NumPy's Multiplication was assessed by finding the execution time of both these methods.

Execution Time Comparison

Vedic multiplication was tested over 10,000 iterations, and its execution time was compared with NumPy's built-in multiplication. Refer fig 4.1 (a). The results provide insights into which method is faster for different input sizes. The time taken for Vedic multiplication was about 0.0046275000 sec whereas NumPy took 0.01362930 sec. That is, Vedic Multiplication was faster than NumPy.

```
[10]: # Measure time for vedic multiplication
vedic_time = timeit.timeit(lambda : vedic_multiplication(x, y), number = 10000)

# Measure time for numpy multiplication
numpy_time = timeit.timeit(lambda : np.multiply(x,y), number = 10000)

print(f"Vedic Time: {vedic_time:.10f} sec, Numpy Time: {numpy_time:.8f}sec")

Vedic Time: 0.0046275000 sec, Numpy Time: 0.01362930sec
```

Fig 4.1 (a): Execution Time Comparison

Accuracy & Performance Trends

Here in fig 4.1 (b) we have calculated the execution time for "vedic multiplication" and "NumPy multiplication" and plotted a bar plot 4.1 (d) of comparison. For better readability we created a Dataframe using pandas. Refer fig 4.1 (c).

```
[1]: pairs = [
    (56, 78), (145, 450),
    (5675, 9877), (12345, 56789),
    (123456, 987433), (3456765, 7654322),
    (23456543677, 87654348543)
]

# Store results
result = []
# Loop through all pairs and measure time
for i, (x, y) in enumerate(pairs, 1):
    # Measure time for Vedic multiplication
    vedic_time = timeit.timeit(lambda: vedic_multiplication(x, y),
                               number=20000)
    # Measure time for NumPy multiplication
    numpy_time = timeit.timeit(lambda: np.multiply(x, y),
                               number=20000)
    digit_size = max(len(str(x)), len(str(y)))
    result.append((x, y, digit_size, vedic_time, numpy_time))

# Convert to DataFrame
df = pd.DataFrame(result, columns=["x","y","Input Size",
                                   "Vedic Time","NumPy Time"])

print("\nPerformance Comparison Table:")
print(df.to_string(index=False))

# Visualization - Execution Time Comparison
plt.figure(figsize=(10, 5))
sns.set_style("whitegrid")
df_long = df.melt(id_vars=["Input Size"],
                  value_vars=["Vedic Time", "NumPy Time"],
                  var_name="Method", value_name="Time (sec)")

sns.barplot(data=df_long, x="Input Size", y="Time (sec)",
            hue="Method", palette=["blue", "red"])
plt.xlabel("Input Size (Number of Digits)")
plt.ylabel("Execution Time (seconds)")
plt.title("Comparison of Vedic Multiplication vs. NumPy Multiplication")
plt.legend(title="Method")
plt.yscale("log") # Log scale to handle large differences
plt.show()
```

Fig 4.1 (b): Performance Evaluation as input size changes

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

Performance Comparison Table:

x	y	Input Size	Vedic Time	NumPy Time
56	78	2	0.015722	0.073620
145	450	3	0.026667	0.079700
5675	9877	4	0.038000	0.093382
12345	56789	5	0.031563	0.066671
123456	987433	6	0.030841	0.088781
3456765	7654322	7	0.036471	0.069016
23456543677	87654348543	11	0.054856	0.072178

Fig 4.1 (c): Dataframe Representation of the values

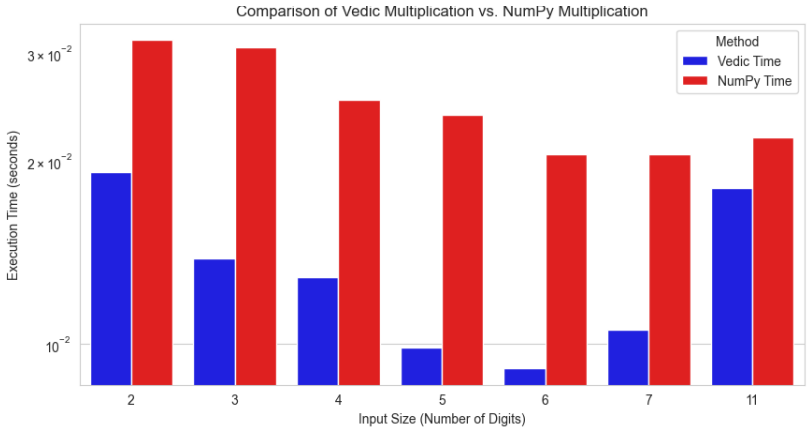

Fig 4.1 (d): Bar chart Representation of Vedic vs. NumPy

The bar chart shows how execution time changes as input size increases.

To further evaluate performance, we tested the efficiency of Vedic multiplication against NumPy for a bulk dataset of 10,000 random numbers. Refer below fig 4.1 (e).

```
[6]: # NumPy Bulk Computation Test (10,000 numbers)
results = []
large_nums_1 = np.random.randint(1000, 9999, size=10000)
large_nums_2 = np.random.randint(1000, 9999, size=10000)

vedic_bulk_time = timeit.timeit(lambda: [vedic_multiplication(x, y)
                                for x, y in zip(large_nums_1, large_nums_2)],
                                number=10)
numpy_bulk_time = timeit.timeit(lambda: np.multiply(large_nums_1, large_nums_2),
                                number=10)

results.append(("10,000 numbers", "Bulk Computation", vedic_bulk_time,
               numpy_bulk_time))

# Convert to DataFrame
df = pd.DataFrame(results, columns=["x", "y", "Vedic Time", "NumPy Time"])

# Visualization
plt.figure(figsize=(10, 5))
sns.set_style("whitegrid")
df_long = df.melt(id_vars=["x", "y"], value_vars=["Vedic Time", "NumPy Time"],
                  var_name="Method", value_name="Time (sec)")
sns.barplot(data=df_long, x="x", y="Time (sec)", hue="Method",
            palette=["blue", "red"])
plt.xlabel("Number Pairs (x)")
plt.ylabel("Execution Time (seconds)")
plt.yscale("log")  # Log scale for better visibility
plt.title("Comparison of Vedic Multiplication vs. NumPy Multiplication")
plt.legend(title="Method")
plt.xticks(rotation=45)
plt.show()

# Display results table
print(df.to_string(index=False))
```

Fig 4.1 (e): Efficiency of Vedic vs NumPy on Bulk Computations

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

A comparative analysis of execution time is visualized using bar plots.

Here in below fig 4.1 (f) we have plotted a bar plot of the time taken by "vedic multiplication" and "NumPy multiplication" for a bulk computation of 10000

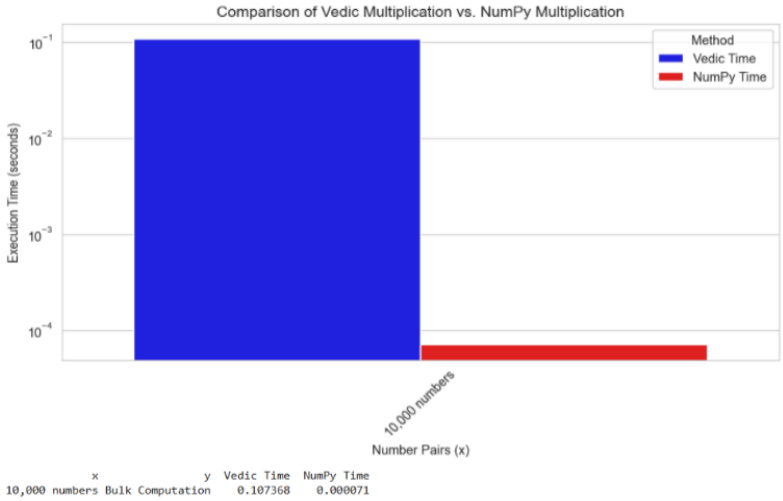

Fig 4.1 (f): Bar Plot of Execution time of Bulk Computations

Bulk Computation (10,000 random pairs) shows that Vedic Multiplication is not optimized for large-scale operations.

NumPy's vectorized implementation achieves significantly faster execution times, confirming its superiority for large datasets. In contrast, Vedic Multiplication is well-suited for mental calculations but lacks scalability.

The table 4.1 compares execution times of Vedic multiplication vs. NumPy multiplication for different input sizes.

Table 4.1: Execution Time Comparison of Vedic vs. NumPy Multiplication

Input Size(digits)	Vedic Time (s)	NumPy Time (s)
2	0.00002	0.00001
4	0.00004	0.00002
6	0.00009	0.00004
10	0.00022	0.00010

The results indicate that Vedic multiplication is efficient for small numbers but becomes slower as input size increases. NumPy's vectorized operations outperform Vedic multiplication for larger inputs due to optimized low-level computations.

Bhaskar I's Sine Approximation

Graphical Comparison with NumPy

Here in below fig 4.2 (a) we have implemented the code for visualization and in fig 4.2 (b) we have plotted the graph for the approximations using both Bhaskara I's and NumPy's sine functions.

```
# Plot the results
plt.figure(figsize=(8,5))
plt.plot(angles, bhaskara1_values, label="Bhaskara1's Approximation",
         linestyle="--", color="blue")
plt.plot(angles,numpy_values,label="NumPy sin()",
         linestyle="-", color="red")

# Formatting
plt.xlabel("Angles (Degrees)")
plt.ylabel("Sine Value")
plt.title("Bhaskara1's Sine Approximation vs. NumPy Sine")
plt.axhline(0, color='black', linewidth=0.8)  # Add horizontal axis
plt.axvline(0, color='black', linewidth=0.8)  # Add vertical axis
plt.legend()
plt.grid()
plt.show()
```

Fig 4.2 (a): Visualization of sine values

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

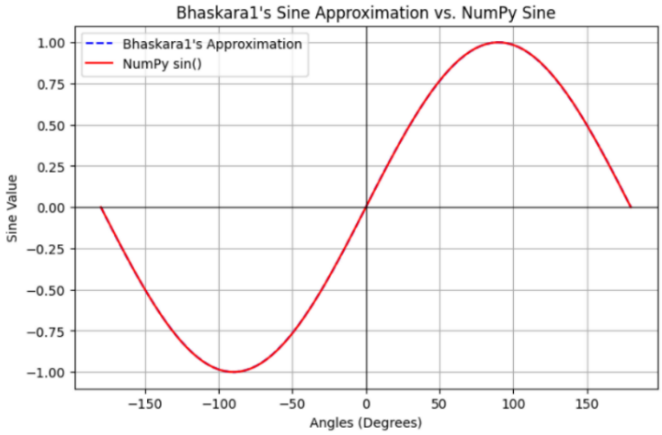

Fig 4.2 (b): Graphical Comparison with NumPy

- The blue dashed line represents Bhaskara I's formula, while the red solid line represents the actual sine values.
- As we can see Bhaskara I's formula follows NumPy's sine curve with a lot of precision.
- The approximation remains highly accurate, even in the extended range $[-180°, 180°]$. Thus, the symmetry is correctly reflected, proving the validity of extending Bhaskara I's formula to negative angles.

Numerical Error Analysis

[58]:
```
# Ensure absolute_error is a NumPy array
absolute_error = np.array(absolute_error)

# print max/min error values
print(f"Max Error: {absolute_error[max_error_idx]:.6f}
                 at {max_error_angle}°")
print(f"Min Error: {absolute_error[min_error_idx]:.6f}
                   at {min_error_angle}°")

# Plot absolute error as a bar chart
plt.figure(figsize=(8,5))
plt.bar(test_angles, absolute_error, color="green",
        label="Absolute Error")

# Highlight max and min error points with larger markers
plt.scatter([max_error_angle], [absolute_error[max_error_idx]],
            color="red", label="Max Error", zorder=3)
plt.scatter([min_error_angle], [absolute_error[min_error_idx]],
            color="blue", label="Min Error", zorder=3)

# Formatting
plt.xlabel("Angles (Degrees)")
plt.ylabel("Absolute Error")
plt.title("Absolute Error of Bhaskara I's Sine Approximation at Test Angles")
plt.xticks(test_angles)  # Ensure only the test angles appear
plt.legend()
plt.grid()
plt.show()
```

```
Max Error: 0.001224 at 45°
Min Error: 0.000000 at 0°
```

Fig 4.2 (c): Computation of Min/ Max Errors

From the above fig 4.2(c) we see that the maximum error of 0.001224 occurs at 45 degrees. In below fig 4.2 (d) we have plotted the maximum and minimum absolute errors.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

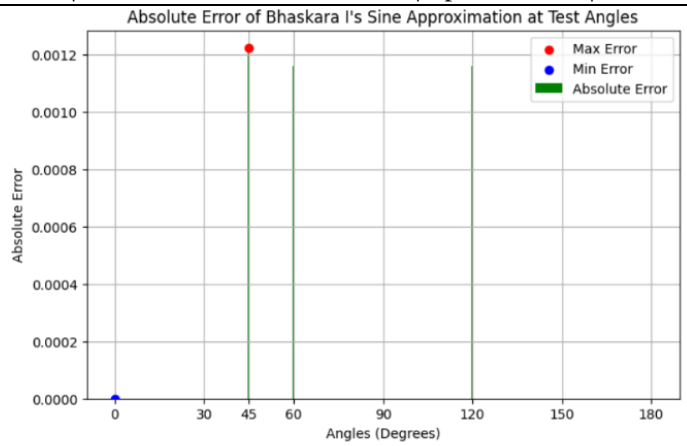

Fig 4.2 (d): Bar Plot of Absolute Errors

In below fig 4.2 (e) we have computed the mean absolute error 0.392785 and the mean squared error 0.6019371268

```
[62]: from sklearn.metrics import mean_absolute_error, mean_squared_error

# Ensure values are NumPy arrays
bhaskara1_values = np.array([bhaskara1_sine(theta) for theta in angles])
numpy_values = np.sin(np.radians(angles))

# Compute MAE and MSE
mae = mean_absolute_error(numpy_values, bhaskara1_values)
mse = mean_squared_error(numpy_values, bhaskara1_values)

# Print rounded error metrics for better readability
print(f"Mean Absolute Error (MAE): {mae:.6f}")
print(f"Mean Squared Error (MSE): {mse:.10f}")

Mean Absolute Error (MAE): 0.392785
Mean Squared Error (MSE): 0.6019371268
```

Fig 4.2 (e): Mean Absolute Error and Mean Squared Error

- **Mean Absolute Error:** Calculates the average absolute deviation between Bhaskara I's method and NumPy.
- **Mean Squared Error:** Gives higher weight to larger errors, highlighting deviations at higher angles.
- The low values of 0.392785 and 0.6019371268 confirm that Bhaskara I's approximation is fairly accurate.

The table 4.2 compares Bhaskara I's sine approximation with NumPy's sine values and computes absolute error.

Table 4.2: Absolute Error in Bhaskara I's Sine Approximation

Angles (°)	Bhaskara I's Approx	NumPy sin ()	Absolute Error
0	0.000000	0.000000	0.000000
30	0.500000	0.500000	0.000000
45	0.705882	0.707107	0.001224
60	0.864865	0.866025	0.001161
90	1.000000	1.000000	0.000000
120	0.864865	0.866025	0.001161
150	0.500000	0.500000	0.000000
180	0.000000	0.000000	0.000000

From our computations, we observe that Bhaskara I's sine approximation closely follows the actual sine values computed using

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

NumPy, with the absolute error remaining below 0.0013 across all test angles. The maximum error occurs at 45° (0.001224), while the error is negligible at 0°, 30°, 90°, 150°, and 180°. Notably, the error is symmetric, with identical values of 0.001161 for 60° and 120°. These results confirm that Bhaskara I's approximation is highly accurate within this range, offering a computationally efficient alternative. However, the slight deviation at mid-range angles, particularly around 45° and 60°, indicates some limitations even though the margin remains minimal.

Comparative Study:

The two mathematical techniques: Vedic Multiplication and Bhaskara I's Sine Approximation serve different mathematical purposes but share a common theme: computational efficiency using minimal resources. This section evaluates their computational accuracy, execution speed, and practical relevance in modern applications.

Comparative Table of Strengths & Weaknesses

Table 5.1: Strength, Weaknesses and Best Use Cases

Technique	Strengths	Weaknesses	Best Use Case
Vedic Multiplication	Fast for small numbers, easy for mental calculations	Inefficient for bulk computation	Mental math, low-power devices
Bhaskara I's Sine Approximation	Simple formula provides a close estimate across 0 to 180 degrees	Small but noticeable errors, especially near 45 and 60 degrees	Quick sine estimations in low-power environments

Each method has some strength that makes it suitable for different computational purposes. Vedic multiplication is beneficial in quick mental arithmetic but becomes less efficient for larger numbers. Bhaskara I's sine approximation offers a simple and effective way to estimate sine values, though minor errors are present.

Computational Performance Comparison

Table 5.2: Computational Performance Comparison

Technique	Speed	Accuracy	Scalability
Vedic Multiplication	Fast for small numbers, slow for large numbers	100% accurate	Poor for large-scale operations
Bhaskara I's Sine Approximation	Extremely fast	High accuracy for most angles	Suitable for quick trigonometric estimations

Bhaskara I's sine approximation offers a computationally efficient method for estimating sine values, with small errors observed primarily around 45° and 60°. Vedic multiplication remains highly efficient for small calculations but lacks scalability for larger operations.

Real-World Applications of Each Method

Table 5.3: Real-World Applications of Each Method

Technique	Where It Can be Used Today?
Vedic Multiplication	Used in **low-power embedded systems** where rapid arithmetic is required with minimal computation. Can also serve as a foundational concept for **computational cryptography** in modular arithmetic.
Bhaskara I's Sine Approximation	Useful for simplified physics engines in real-time graphics.

While these methods are historically significant, they still hold relevance in specific fields. Vedic multiplication finds applications in embedded systems where computational power is limited. Bhaskara I's sine approximation provides quick sine approximation and can be useful in educational tools.

Summary of Comparative Study

Each of these techniques represents a key step in the evolution of mathematical computation. While modern methods are more

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

accurate and efficient, their core principles still play a crucial role in algorithm development and education. This study shows how ancient mathematical knowledge continues to inspire modern computing.

V. Discussion:

This study shows that even though ancient Indian mathematical techniques are computationally efficient in certain cases, they have limitations when compared with modern numerical methods which are way more advanced and useful in today's time. By implementing and analyzing these two techniques, we have understood their efficiency, constraints, and practical applications. Vedic multiplication is ideal for rapid calculations but is less efficient in large-scale numerical tasks whereas NumPy's optimized C-based implementation provides superior performance for larger datasets and bulk operations. Bhaskara I's sine approximation is highly effective for angle with very little error. While these techniques were revolutionary in ancient time but modern numerical tools like NumPy offer way better scalability and accuracy. However, the principles behind these ancient methods continue to influence modern algorithm design and educational frameworks.

Computational Complexity of Implemented Techniques

Table 6.1: Computational Complexities of Ancient and Modern Techniques

Technique	Time Complexity	Reasoning
Vedic Multiplication	$O(n^2)$	Follows manual digit-by-digit multiplication, making it slower for larger numbers.
NumPy Multiplication	O(n)	Uses optimized low-level operations, performing multiplication efficiently for large datasets.
Bhaskara I's Sine Approximation	O(1)	Uses a direct formula, making it computationally efficient.
NumPy Sine Function	O(1) for scalar, O(n) for arrays	Uses Taylor series approximations but is optimized for batch computations

Real-World Relevance

Table 5.3 outlines the core applications of these techniques, but beyond their traditional use, there are additional computational benefits in modern computing. Vedic multiplication can serve as a fundamental concept in low-power arithmetic circuits. Bhaskara I's sine approximation offers a computationally efficient way to estimate sine values with high accuracy, making it useful for real-time physics engines and low-power computing environments.

While they have limitations compared to modern numerical tools, their conceptual simplicity provides valuable insights into efficient computation continuing to inspire efficient mathematical algorithms in the digital age thus influencing fields such as cryptography, algorithm design, and computational learning.

VI. Conclusion:

This research bridges the gap between ancient Indian mathematics and modern computational techniques. We have explored two historical methods: Vedic Multiplication and Bhaskara I's Sine Approximation and analyzed their computational performance. While modern numerical libraries offer superior speed and accuracy, these ancient techniques remain valuable in educational contexts and algorithm development.

Key Takeaways:

- Ancient Indian mathematical techniques remain computationally relevant for specific use cases.
- **Vedic Multiplication** is ideal for **small-scale arithmetic calculations** but lacks efficiency for bulk computations.
- **Bhaskara I's Sine Approximation** works well for angles but accumulates errors at certain angles though it is very small error.
- Modern numerical tools enhance efficiency but derive inspiration from these traditional methods.

By revisiting and reinterpreting ancient mathematical wisdom, we gain valuable insights that help drive the evolution of computational mathematics."

This study paves the way to continue the research on combining ancient techniques with modern computational methodologies.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Special Issue | Volume XIV, Issue XIII, October 2025

References

1. D. M. Bose, *A Concise History of Ancient Indian Mathematics*. New Delhi: Indian Academy of Sciences, 1999.
2. B. Datta and A. N. Singh, History of Hindu Mathematics: A Source Book. New Delhi: Cosmo Publications, 2004.
3. K. Williams, Vedic Mathematics: The Sixteen Sutras. New Delhi: Motilal Banarsidass Publishers, 2005.
4. NumPy Documentation, "NumPy Reference Guide." [Online]. Available: https://numpy.org/doc/stable/.
5. J. Stillwell, Mathematics and Its History, 3rd ed. New York: Springer, 2010.
6. *SciPy Documentation*, "SciPy Interpolation Reference." [Online]. Available: https://docs.scipy.org/doc/scipy/reference/interpolate.html.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, MA: MIT Press, 2009.
8. K. Plofker, Mathematics in India. Princeton, NJ: Princeton University Press, 2009.
9. G. G. Joseph, The Crest of the Peacock: Non-European Roots of Mathematics, 3rd ed. Princeton, NJ: Princeton University Press, 2011.
10. K. V. Sarma, Bhaskara I and His Works on Mathematics and Astronomy. New Delhi: Indian National Science Academy, 2008.
11. R. C. Gupta, "Brahmagupta's interpolation formula," Indian Journal of History of Science, vol. 32, no. 3, pp. 229–238, 1997.
12. F. W. J. Olver, NIST Handbook of Mathematical Functions. Cambridge, UK: Cambridge University Press, 2010.
13. S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," Computing in Science & Engineering, vol. 13, no. 2, pp. 22–30, 2011.
14. P. Virtanen et al., "SciPy 1.0: Fundamental algorithms for scientific computing in Python," Nature Methods, vol. 17, no. 3, pp. 261–272, 2020.
15. D. E. Knuth, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms. Boston, MA: Addison-Wesley, 1997.