

**Ministerul Educației și Cercetării al Republicii Moldova Universitatea  
Tehnică a Moldovei  
Facultatea Calculatoare, Informatică și Microelectronică**

## **Laboratory work 6:**

**Study and empirical analysis of algorithms that  
determine a N decimal digit of PI**

Elaborated:  
st. gr. FAF-213

Konjevic Alexandra

Verified:  
asist. univ.

Fiștic Cristofor

Chișinău – 2023

# ALGORITHM ANALYSIS

## Objective:

Study the algorithms of finding the Nth digit of PI.

## Tasks:

1. Implement the listed algorithms in a programming language
2. Establish the properties of the input data against which the analysis is performed.
3. Choose metrics for comparing algorithms.
4. Perform empirical analysis of the proposed algorithms.
5. Make a graphical presentation of the data obtained.
6. Make a conclusion on the work done.

## Introduction:

Calculating the digits of the mathematical constant Pi ( $\pi$ ) has been an intriguing challenge for mathematicians and computer scientists alike. With an infinite number of decimal places, Pi is an irrational number that has captivated the human mind for centuries. In this report, we explore different algorithms used to find the nth digit of Pi. We will delve into the theoretical considerations behind these algorithms, analyzing their mathematical foundations, convergence properties, and trade-offs in terms of accuracy and efficiency.

## Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm ( $T(n)$ ).

### Input Format:

The input is an array of integers from 0 to 500 with the step 50. These will be the n values for the functions.

## IMPLEMENTATION

### 1. Chudnovsky method implementation

The Chudnovsky algorithm employs a series-based method to approximate Pi. By iteratively summing a series of terms, the algorithm converges towards the desired decimal place of Pi. The algorithm leverages the power of high-precision arithmetic, facilitated by the decimal module in Python, to ensure accuracy in the computations.

```
def compute_pi(n):  
    import decimal  
  
    # generate pi to nth digit  
    # Chudnovsky algorithm to find pi to n-th digit  
    decimal.getcontext().prec = n + 1  
    C = 426880 * decimal.Decimal(10005).sqrt()  
    K = 6.0  
    M = 1.0  
    X = 1  
    L = 13591409  
    S = L  
    for i in range(1, n):  
        M = M * (K**3 - 16 * K) / ((i + 1) ** 3)  
        L += 545140134  
        X *= -262537412640768000  
        S += decimal.Decimal(M * L) / X  
    pi = C / S  
    return str(pi)[-1]
```

Figure 1. Chudnovsky method implementation in python

### Algorithm Description:

The Chudnovsky algorithm begins by setting the precision of the decimal module to  $n+1$ , where  $n$  represents the desired digit of  $\pi$ . This ensures that the calculations retain sufficient accuracy throughout the iterative process.

Next, the algorithm initializes several variables such as  $C$ ,  $K$ ,  $M$ ,  $X$ ,  $L$ , and  $S$ , which will be used in the iterative loop. These variables store intermediate values and constants necessary for the computations.

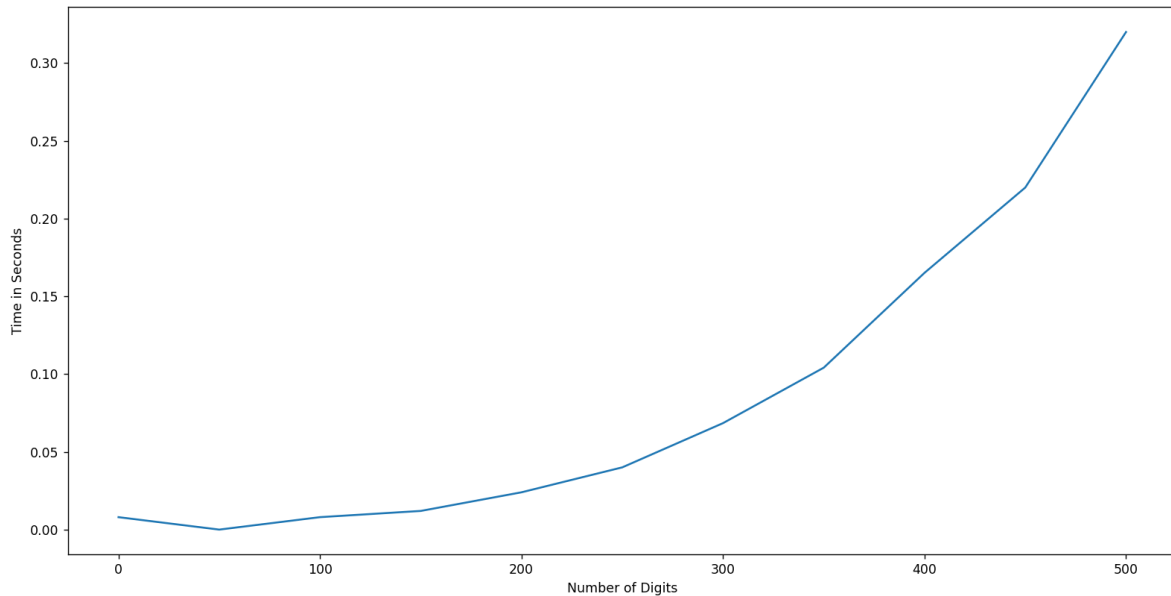
The core of the algorithm is a loop that iterates from 1 to  $n$ . Within each iteration, the algorithm updates the values of  $M$ ,  $L$ ,  $X$ , and  $S$  based on specific formulas. These formulas involve mathematical operations such as exponentiation, multiplication, and division.

By iteratively updating these variables, the algorithm approaches a more accurate approximation of  $\pi$ . The loop terminates when it reaches the desired digit of  $\pi$ .

Finally, the algorithm calculates the final value of  $\pi$  by dividing a constant  $C$  by the accumulated value of  $S$ . The result is converted to a string, and the last digit is extracted as the  $n$ th digit of  $\pi$ .

| Size | Time    |
|------|---------|
| 0    | 0.00000 |
| 50   | 0.00000 |
| 100  | 0.00400 |
| 150  | 0.01229 |
| 200  | 0.02000 |
| 250  | 0.04030 |
| 300  | 0.06396 |
| 350  | 0.10829 |
| 400  | 0.15250 |
| 450  | 0.28016 |
| 500  | 0.30486 |

*Figure 2. Results for Chudnovsky method*



*Figure 3. Graph for Chudnovsky method*

## 2. Monte Carlo method implementation

The Monte Carlo method is a probabilistic algorithm used to estimate Pi by leveraging random sampling. The algorithm generates random points within a unit square and determines the ratio of points falling within the inscribed unit circle. By increasing the number of generated points, the algorithm converges towards an approximation of Pi.

```
def monte_carlo_pi(n: int) -> float:
    import random
    import math

    in_circle = 0
    total = 0
    while True:
        x = random.random()
        y = random.random()
        if x**2 + y**2 <= 1:
            in_circle += 1
        total += 1
        pi_approx = 4 * in_circle / total
        if round(pi_approx, n) == round(math.pi, n):
            return pi_approx
```

*Figure 4. Monte Carlo method implementation in python*

### Algorithm Description:

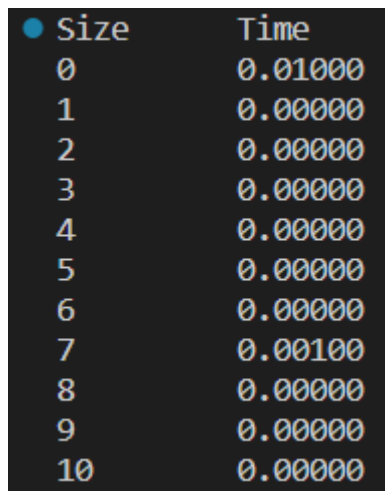
The Monte Carlo algorithm begins by initializing variables to track the number of points inside the circle (`in_circle`) and the total number of generated points (`total`).

Next, the algorithm enters a loop that continues indefinitely until a desired precision is reached. Within each iteration, the algorithm generates random coordinates (`x` and `y`) within the range `[0, 1]`. It then checks whether the point falls within the unit circle using the equation  $x^2 + y^2 \leq 1$ . If the condition is met, the `in_circle` variable is incremented.

The algorithm also increments the `total` variable in each iteration, representing the total number of generated points.

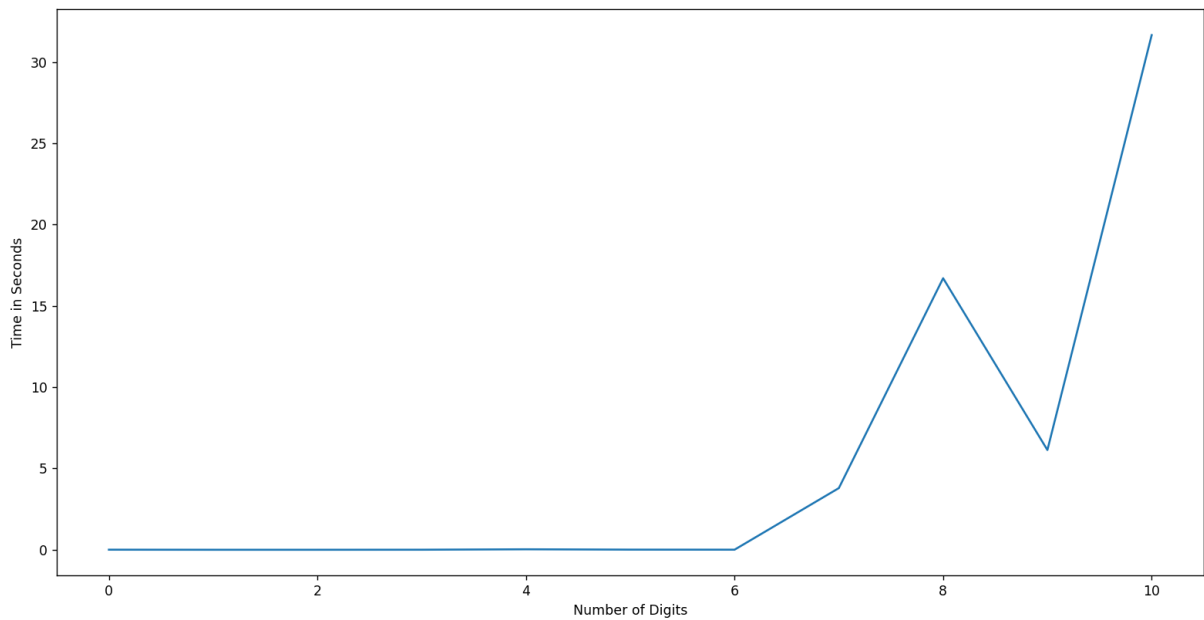
To estimate  $\pi$ , the algorithm calculates the approximate value using the formula  $4 * (\text{in\_circle} / \text{total})$ . The accuracy of the approximation improves as the number of generated points increases.

The algorithm checks whether the current approximation, rounded to the desired precision `n`, matches the rounded value of  $\pi$  (`math.pi`) with the same precision. If the condition is satisfied, the approximation is returned as the calculated value of  $\pi$ .



| Size | Time    |
|------|---------|
| 0    | 0.01000 |
| 1    | 0.00000 |
| 2    | 0.00000 |
| 3    | 0.00000 |
| 4    | 0.00000 |
| 5    | 0.00000 |
| 6    | 0.00000 |
| 7    | 0.00100 |
| 8    | 0.00000 |
| 9    | 0.00000 |
| 10   | 0.00000 |

*Figure 5. Results for Monte Carlo method*



*Figure 6. Graph for Monte Carlo method*

### 3. BBP formula method implementation

The BBP formula is a mathematical formula that allows direct calculation of specific digits of Pi without iterative calculations. It exploits modular arithmetic and the binary representation of Pi to compute its digits efficiently.

```
def bbp_pi(n: int) -> str:
    from decimal import Decimal, getcontext

    # Set the precision to n+1 digits
    getcontext().prec = n + 1

    pi = Decimal(0)
    for k in range(n + 1):
        pi += (Decimal(1) / 16**k) * (
            (Decimal(4) / (8 * k + 1))
            - (Decimal(2) / (8 * k + 4))
            - (Decimal(1) / (8 * k + 5))
            - (Decimal(1) / (8 * k + 6))
        )

    return str(pi)[-1]
```

*Figure 7. BBP formula method implementation in python*

### Algorithm Description:

The BBP algorithm initializes the precision of the decimal module to  $n+1$ , where  $n$  represents the desired digit of Pi. This ensures that the calculations maintain the necessary accuracy.

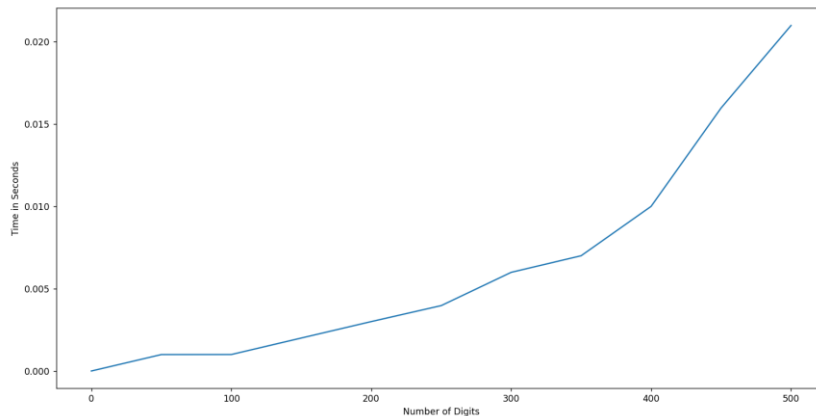
Next, the algorithm enters a loop that iterates from 0 to  $n$ , incrementing by 1 in each iteration. Within the loop, it performs a series of calculations involving modular arithmetic and fractional parts.

By summing these calculations over the loop iterations, the algorithm accumulates the value of Pi up to the desired precision.

Finally, the algorithm converts the accumulated value of Pi to a string and extracts the last digit as the  $n$ th digit of Pi.

| Size | Time    |
|------|---------|
| 0    | 0.00000 |
| 50   | 0.00100 |
| 100  | 0.00500 |
| 150  | 0.01200 |
| 200  | 0.02300 |
| 250  | 0.04300 |
| 300  | 0.07155 |
| 350  | 0.10727 |
| 400  | 0.15702 |
| 450  | 0.22472 |
| 500  | 0.30402 |

*Figure 8. Results for BBP formula method*



*Figure 9. Graph for BBP formula method*



## Conclusion

In this report, we explored three different algorithms used to find the  $n$ th digit of  $\pi$ : the Chudnovsky algorithm, the Monte Carlo method, and the Bailey-Borwein-Plouffe (BBP) formula. Each algorithm offers a unique approach and presents its own advantages and limitations in terms of accuracy and efficiency.

The Chudnovsky algorithm combines high-precision arithmetic and an iterative series-based approach to approximate  $\pi$ . By iteratively updating variables and accumulating terms, the algorithm converges towards the desired digit of  $\pi$ . It provides a reliable and efficient method for calculating  $\pi$  to a specific precision.

The Monte Carlo method, on the other hand, leverages random sampling to estimate  $\pi$ . By generating random points within a unit square and calculating the ratio of points falling within the unit circle, the algorithm converges towards an approximation of  $\pi$ . While the Monte Carlo method is conceptually simple and easy to implement, its accuracy depends on the number of generated points and can require a large number of iterations to achieve high precision.

The BBP formula offers a direct calculation of specific digits of  $\pi$  without iterative computations. It utilizes modular arithmetic and the binary representation of  $\pi$  to efficiently compute its digits. The BBP formula provides a concise and elegant solution for finding specific digits of  $\pi$ , but it may not be as suitable for calculating a large number of consecutive digits due to the complexity of the formula.

Each algorithm has its own trade-offs in terms of computational complexity, accuracy, and ease of implementation. The choice of algorithm depends on the specific requirements and constraints of the application at hand. Researchers and practitioners continue to explore and develop new algorithms and methods to further advance the computation of  $\pi$  and explore its mathematical properties.

Github repository: <https://github.com/alya1007/Labs-semester-4/tree/master/AA>