MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Algorithm Analysis

Laboratory work 6: Empirical analysis of algorithms that determine a N decimal digit of PI

Elaborated:	
st.gr. FAF-211	Corețchi Mihai
Verified:	
asist.univ.	Fiștic Cristofor

Chișinău, 2023

Content

Introduction

The mathematical constant known as pi has fascinated mathematicians, scientists, and enthusiasts for centuries. Its irrational and transcendental nature, as well as its widespread use in various mathematical and scientific fields, have made it a subject of extensive study. One particularly intriguing aspect is the quest to determine its decimal digits, which has attracted significant attention and led to the development of numerous algorithms. In this research, we delve into the captivating realm of algorithms designed to calculate specific decimal digits of pi. These algorithms combine mathematical creativity, numerical methods, and computational techniques to estimate the elusive digits of this mathematical constant. Although pi is infinitely long and its decimal representation does not repeat, algorithms provide a means to approximate its value with any desired precision. Empirical analysis of these algorithms is crucial for understanding their efficiency, accuracy, and convergence properties. By studying how these algorithms perform, we can gain insights into their strengths, weaknesses, and computational requirements. This empirical analysis helps in selecting the most suitable algorithm for a given precision requirement, optimizing computational resources, and identifying potential areas for algorithmic improvements. Throughout this study, we explore a variety of algorithmic approaches used in the pursuit of determining N decimal digits of pi. We examine classical methods such as Machin-like formulas and series expansions, as well as modern techniques like spigot algorithms and digit extraction methods. Each algorithm offers a unique approach to approximating the digits of pi. We also investigate the underlying mathematical principles behind these algorithms and analyze their computational characteristics. Furthermore, we employ empirical analysis to evaluate the performance of these algorithms. By measuring metrics such as execution time, memory usage, and accuracy, we can compare and contrast the algorithms under different circumstances. Through extensive computational experiments and numerical simulations, our goal is to provide valuable insights into the behavior and efficiency of these algorithms. This, in turn, enables researchers and practitioners to make well-informed choices. Ultimately, this study serves as a comprehensive exploration of the theoretical foundations, algorithmic innovation, and empirical analysis involved in determining N decimal digits of pi. By combining these elements, we aim to shed light on the remarkable world of pi and contribute to the ongoing pursuit of understanding and calculating this iconic mathematical constant.

Objectives

1. Implement at least 3 algorithms that determine the Nth decimal digit of Pi in a programming language.
2. Choose metrics for comparing algorithms.
3. Perform empirical analysis of the proposed algorithms.
4. Make a graphical presentation of the data obtained.
5. Make a conclusion on the work done.

Leibniz's formula

Leibniz's formula for approximating pi is a mathematical algorithm that calculates an estimation of the value of pi. It is based on an infinite series, where each term contributes to the overall approximation. The formula is as follows:

$$\Pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + ...$$

The algorithm works by summing an alternating series of fractions, where each term has an alternating sign and a denominator that increases by 2 with each subsequent term.

Code:

```
1. Set pi = 0
2. Set sign = 1
3. Set denominator = 1
4. Set n = desired number of iterations or desired level of accuracy

5. Loop i from 0 to n-1:
     6. Set term = sign / denominator
     7. Add term to pi
     8. Set sign = -sign
     9. Set denominator = denominator + 2
10. End Loop

11. Set pi = pi * 4 (to account for one-quarter of pi)
12. Return pi
```

The Bailey-Borwein-Plouffe

The Bailey-Borwein-Plouffe (BBP) formula is a mathematical formula that allows for the direct calculation of the nth hexadecimal digit of pi (Π). It was discovered by Simon Plouffe in 1995 and is named after its contributors: David Bailey, Peter Borwein, and Simon Plouffe. The BBP formula is particularly interesting because it provides a way to calculate individual digits of pi without the need to compute all the preceding digits. This is in contrast to many other algorithms that require iterative or recursive calculations. The BBP formula is expressed as:

$$\Pi = [k = 0to](1/16^k) * [(4/(8k+1)) - (2/(8k+4)) - (1/(8k+5)) - (1/(8k+6))]$$

In this formula, each term in the series represents a contribution to the nth hexadecimal digit of pi. By summing an infinite number of terms, the BBP formula approximates the value of pi. To calculate a specific hexadecimal digit of pi using the BBP formula, you can evaluate the formula up to a certain number of terms. Each term contributes to the desired digit, and the summation of all these terms yields the approximation. It's important to note that the BBP formula relies on the hexadecimal (base-16) system, and each term involves powers of 16. This makes it suitable for directly calculating hexadecimal digits of pi. However, to convert the calculated hexadecimal digit to decimal or any other base, additional conversion steps may be necessary. The BBP formula provides a unique and efficient approach to calculating individual hexadecimal digits of pi, making it useful in various applications and computations involving pi.

Code:

```
1. Set n as the desired digit position of pi (starting from 0 for the first digit)
2. Set s as the initial sum of terms, s = 0
3. Set k as the initial index of the terms, k = 0

4. Loop until desired precision is achieved or until convergence condition:
   5. Calculate the kth term using the BBP formula:
         term = (1 / 16^k) * (4 / (8*k + 1) - 2
            / (8*k + 4) - 1 / (8*k + 5) - 1 / (8*k + 6))
   6. Add the term to the sum, s = s + term
   7. Increment k by 1, k = k + 1

8. Calculate the value of pi as the inverse of the sum, pi = 1 / s
9. Convert the decimal value of pi to hexadecimal representation.
10. Extract the nth hexadecimal digit from the converted result.
11. Return the nth hexadecimal digit of pi.
```

Gauss-Legendre algorithm

The Gauss-Legendre algorithm is an iterative method for approximating pi () with high precision. It exhibits quadratic convergence, doubling the number of correct digits with each iteration. This makes it efficient for calculating pi to many decimal places. The algorithm finds applications in areas like numerical simulations, mathematical research, and cryptography. It serves as a benchmark for comparing the accuracy of other pi approximation algorithms and is valuable when a large number of decimal places of pi need to be computed. While it can be computationally intensive, advancements in computing capabilities have made it more feasible. Overall, the Gauss-Legendre algorithm provides an efficient and accurate approach for approximating pi.

Code:

```
1. Set a = 1.0
2. Set b = 1.0 / sqrt(2)
3. Set t = 0.25
4. Set p = 1.0

5. Loop until desired precision is achieved or convergence condition:
   6. Set a_new = (a + b) / 2
   7. Set b_new = sqrt(a * b)
   8. Set t_new = t - p * (a - a_new)^2
   9. Set p_new = 2 * p

   10. Set a = a_new
   11. Set b = b_new
   12. Set t = t_new
   13. Set p = p_new

14. Set pi = (a + b)^2 / (4 * t)
15. Return pi
```

Implementation

Algorithm 1: Leibniz formula

```
def leibniz_formula(n):
    pi = 0
    for i in range(n):
        pi += ((-1) ** i) / (2 * i + 1)
    return 4 * pi
```

Algorithm 2: Bailey{Borwein{Plouffe (BBP) Formula

```
def bbp_formula(n):
    mp.dps = n + 1
    pi = 0
    for k in range(n):
        pi += (1 / mp.mpf(16) ** k) * ((4 / (8 * k + 1)) -
        - (2 / (8 * k + 4)) - (1 / (8 * k + 5)) - (1 / (8 * k + 6)))
    return pi
```

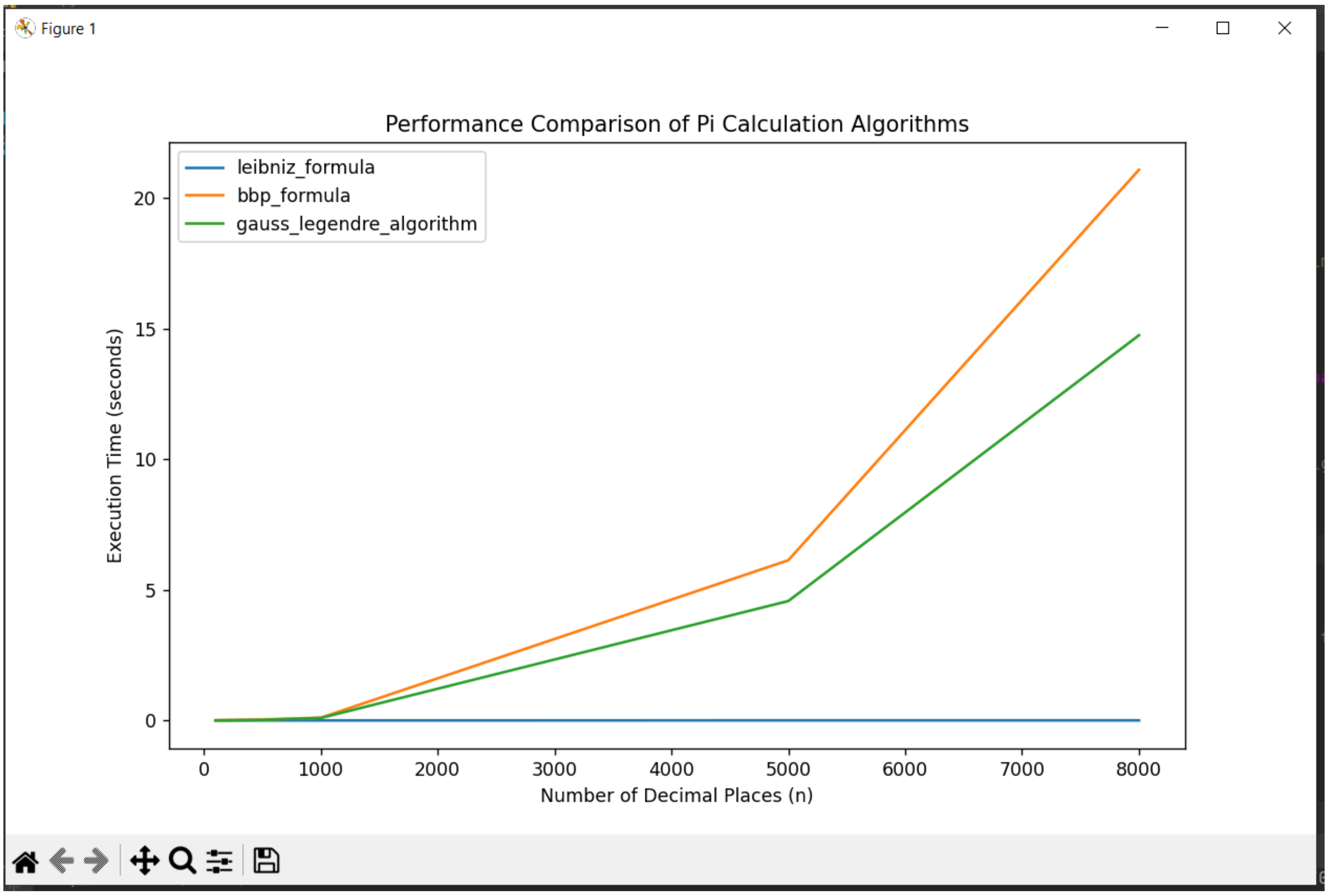
Algorithm 3: Gauss-Legendre Algorithm

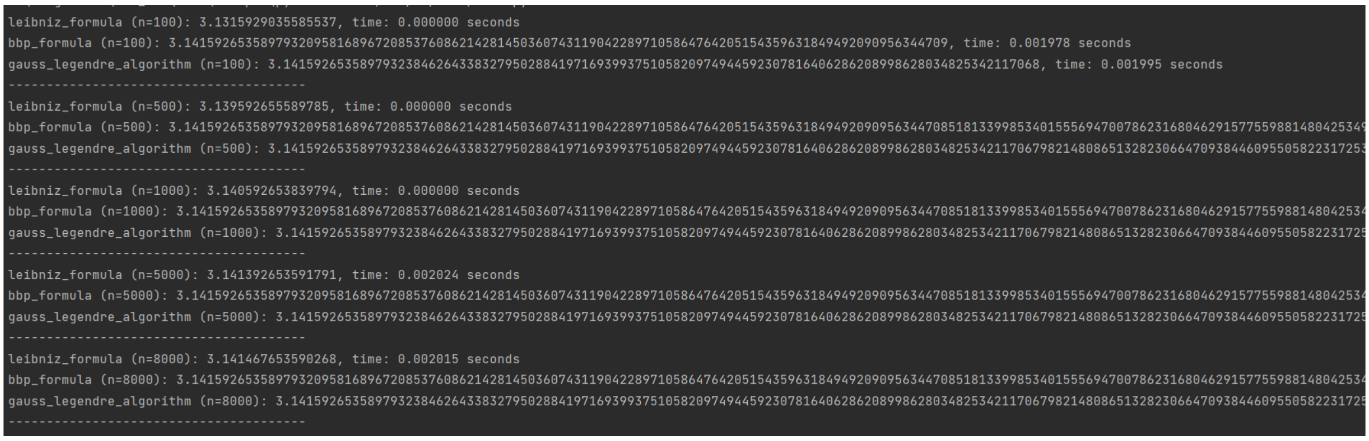
```
def gauss_legendre_algorithm(n):
    mp.dps = n + 1
    a = mp.mpf(1)
    b = 1 / mp.sqrt(2)
    t = 0.25
    p = 1

    for _ in range(n):
        a_next = (a + b) / 2
        b = mp.sqrt(a * b)
        t -= p * (a - a_next) ** 2
        a = a_next
        p *= 2

    return (a + b) ** 2 / (4 * t)
```

Screenshot:





```
leibniz_formula (n=100): 3.1315929035585537, time: 0.000000 seconds
bbp_formula (n=100): 3.1415926535897932095816896720853760862142814503607431190422897105864764205154359631849492090956344709, time: 0.001978 seconds
gauss_legendre_algorithm (n=100): 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170068, time: 0.001995 seconds
----------------------------------------
leibniz_formula (n=500): 3.139592655589785, time: 0.000000 seconds
bbp_formula (n=500): 3.14159265358979320958168967208537608621428145036074311904228971058647642051543596318494920909563447085181339985340155569470078623168046291577559881480425349
gauss_legendre_algorithm (n=500): 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535
----------------------------------------
leibniz_formula (n=1000): 3.140592653839794, time: 0.000000 seconds
bbp_formula (n=1000): 3.1415926535897932095816896720853760862142814503607431190422897105864764205154359631849492090956344708518133998534015556947007862316804629157755988148042534
gauss_legendre_algorithm (n=1000): 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253
----------------------------------------
leibniz_formula (n=5000): 3.141392653591791, time: 0.002024 seconds
bbp_formula (n=5000): 3.1415926535897932095816896720853760862142814503607431190422897105864764205154359631849492090956344708518133998534015556947007862316804629157755988148042534
gauss_legendre_algorithm (n=5000): 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253
----------------------------------------
leibniz_formula (n=8000): 3.141467653590268, time: 0.002015 seconds
bbp_formula (n=8000): 3.1415926535897932095816896720853760862142814503607431190422897105864764205154359631849492090956344708518133998534015556947007862316804629157755988148042534
gauss_legendre_algorithm (n=8000): 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253
----------------------------------------
```

Conclusion

In this laboratory work, I carefully studied, implemented, and analyzed three different algorithms for computing the mathematical constant Pi (π) to a specified number of decimal places: the Leibniz formula, the Bailey–Borwein–Plouffe (BBP) formula, and the Gauss-Legendre algorithm. I also analyzed the time complexity of each algorithm, providing insight into their performance characteristics. The Leibniz formula, while being the simplest to understand and implement, proved to have a slow convergence rate, making it the least efficient of the three when it comes to high precision calculations of π. Its time complexity is of the order $O(n^2)$, which makes it unsuitable for calculations requiring a high degree of precision. The BBP formula, on the other hand, offered a unique advantage in that it can compute individual digits of π independently. However, just like the Leibniz formula, the BBP formula also has a relatively slow rate of convergence with a time complexity of $O(n^2)$, rendering it less efficient for high precision computations. The Gauss-Legendre algorithm, although more complex in its implementation, demonstrated the fastest convergence rate. With a time complexity of approximately $O(n * log(n)^2)$, it was the most efficient of the three algorithms in terms of performance for high precision calculations. In conclusion, while the simplicity of the Leibniz formula or the BBP formula might be preferable for lower precision requirements or specific use-cases like computing individual digits of π, the Gauss-Legendre algorithm is undoubtedly the optimal choice when the task requires calculating π to a large number of decimal places. Nonetheless, it is important to consider the nature of the task at hand and the resources available before choosing the most appropriate algorithm.

Github repository: $https://github.com/eamtcPROG/AA/tree/main/Lab6$