**INFO 6205 Summer 1 2023 Project**

**Amrita Dubey (002760658)**

**Saniya Athani (002682245)**

● **Introduction**:

The objective of this project is to explore various mathematical series for pi calculation which involves calculation of big numbers. Gain deeper understanding of Karatsuba's algorithm of multiplication and numerical precision. To explore and study the “Madhava-Leibniz-Gregory” series to estimate the approximate value of pi using the corrected terms.

● **Aim**:

The implementation of Karatsuba's multiplication algorithm using the BigNumber class and the calculation of 𝜋 to 100 decimal places using the Wallis product. Additionally, the performance of Karatsuba's method will be benchmarked against traditional multiplication when applied to the Wallis product, and to utilise the “Madhava-Leibniz-Gregory” series to estimate the approximate value of pi with the first three correction terms, Additionally, exploring the possibility of identifying other term for the same series that could aid in the approximation process.

● **Program**

● **Algorithm: Karatsuba’s Algorithm**

● **Observations & Graphical Analysis:**

To benchmark Karatsuba’s multiplication algorithm, we have:

· Doubled the input length which is the length of the multiplicand and the multiplier each iteration.

· The cut-off value was starting from 50 till 1000 by doubling the cutoff value for each iteration.

To test the implementation of the divide and conquer multiplication we have added test cases to validate the product against expected output.

Karatsuba’s multiplication was benchmarked against standard multiplication to validate the performance gain that the algorithm offered by reducing the number of multiplications required. The following graph represents the time complexity of normal and Karatsuba’s multiplication. It is observed that Karatsuba takes significantly less time. The graph for regular multiplication is approximately similar to the quadratic graph.

Experimenting with smaller input length along with varying base conditions, starting from 200 to 3200, it was observed that Karatsuba performs better than standard multiplication if input length is greater than 400. The performance of Karatsuba multiplication below input length of 400 was like or worse than standard multiplication. Based on input length where we begin to see performance gain using Karatsuba multiplication the base condition for recursion was set to 160. If the size of the input is less than 160 or the recursive division break the subproblem to 160 or less the algorithm will perform standard multiplication.

**Less is better, for % higher is better.**

**Experiment Results**

****

**Base condition 20**

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Length** | **Karatsuba Multiplication (ms)** | **Standard Multiplication (ms)** | **Performance gain over normal multiplication (%)** |
| **200** | **2.3** | **1.5** | **-53.3333** |
| **400** | **5.5** | **2.9** | **-89.6552** |
| **800** | **9.9** | **7.7** | **-28.5714** |
| **1600** | **23.0** | **23.7** | **2.953586** |
| **3200** | **49.3** | **55.5** | **11.17117** |

**Base condition 40**

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Length** | **Karatsuba Multiplication (ms)** | **Standard Multiplication (ms)** | **Performance gain over normal multiplication (%)** |
| **200** | **2.5** | **1.6** | **-56.25** |
| **400** | **3.4** | **3.1** | **-9.67742** |
| **800** | **7.8** | **9.1** | **14.28571** |
| **1600** | **15.5** | **23.6** | **34.32203** |
| **3200** | **39.4** | **42.6** | **7.511737** |

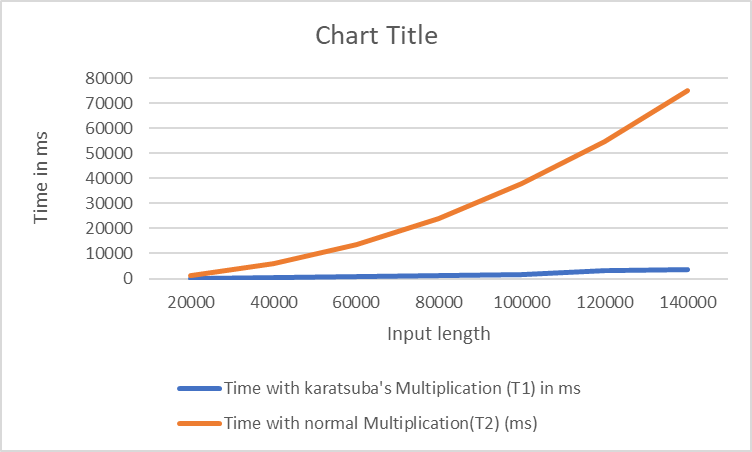
**Base condition 80**

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Length** | **Karatsuba Multiplication (ms)** | **Standard Multiplication (ms)** | **Performance gain over normal multiplication (%)** |
| **200** | **2.1** | **2.0** | **-5** |
| **400** | **4.0** | **4.1** | **2.439024** |
| **800** | **8.7** | **13.1** | **33.58779** |
| **1600** | **12.6** | **24.9** | **49.39759** |
| **3200** | **23.9** | **41.5** | **42.40964** |

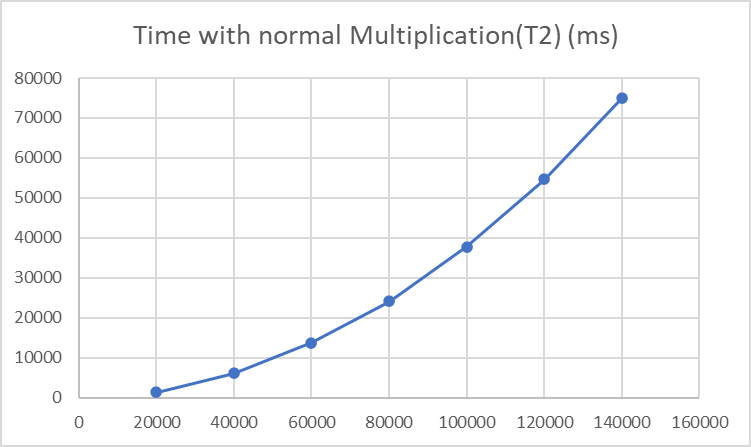
**Base condition 160**

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Length** | **Karatsuba Multiplication (ms)** | **Standard Multiplication (ms)** | **Performance gain over normal multiplication (%)** |
| **200** | **1.1** | **1.5** | **26.66666666666666** |
| **400** | **2.4** | **3.4** | **29.411764705882355** |
| **800** | **5.6** | **13.4** | **58.2089552238806** |
| **1600** | **9.0** | **24.8** | **63.7096774193** |
| **3200** | **17.7** | **40.9** | **56.723716381418086** |

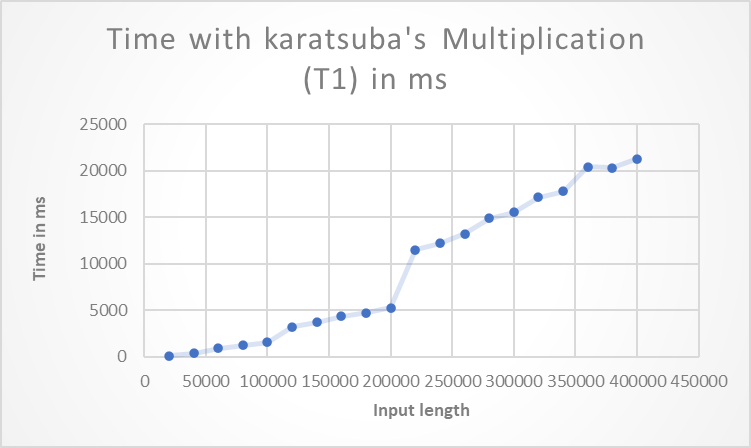
The following graph represents the time complexity of Karatsub’s multiplication with standard multiplication. For this experiment we checked the time taken to complete multiplication of large decimals starting from 20k to140K. Standard multiplication slowed down the computation significantly making it tough to benchmark standard multiplication beyond 140K.

****

**For better visualisation for both standard and Karatsuba’s multiplication we have added the following graph:**

****

Our observation from the above graph shows that standard multiplication takes quadratic time O(n^2), when the size of the input is increased by 20K for every experimental run.

The following graph represents the time taken by Karatsub’s multiplication, the following graph looks like it falls between O(N^2) and O(N):****

● **Analysis of Karatsuba method in Java runtime library**

BigInteger class provides a built-in method for multiplication "multiply()" to handle large number multiplication. Multiply method is based on Toom-Cook and Karatsuba algorithm, these algorithms leverage the divide and conquer principle by breaking down large numbers into smaller subproblems by using recursion, digit wise multiplication and partial product accumulation. Focusing more on Karatsuba multiplication, this approach reduces the number of recursive multiplication required to perform large number multiplication. The algorithm uses the following mathematical expression:

**a \* b = (a1 \* b1) \* B^2 + ((a1 + a0) \* (b1 + b0) - (a1 \* b1) - (a0 \* b0)) \* B + a0 \* b0**

**a** and **b** are large input numbers and **a1, a0, b1,** and **b0** represent their high and low parts after splitting. The algorithm recursively calculates immediate results using three multiplications instead of four that results in a reduced time complexity.

Toom-Cook multiplication builds on Karatsuba multiplication, the algorithm breaks down the problem into multiple sub-problems instead of just two (High and Low) that Karatsuba does. It leverages the concept of polynomial interpolation and evaluation to multiply these parts efficiently. The mathematical expression can be given by

**a \* b = (a1 \* B + a0) \* (b1 \* B + b0) = (a1 \* b1) \* B^2 + ((a1 + a0) \* (b1 + b0) - (a1 \* b1) - (a0 \* b0)) \* B + a0 \* b0**

**a \* b = (a1 \* B + a0) \* (b1 \* B + b0) = (a1 \* b1) \* B^2 + ((a1 + a0) \* (b1 + b0) - (a1 \* b1) - (a0 \* b0)) \* B + a0 \* b0**

Where **a1, a0, b1,** and **b0** represent the different parts obtained after splitting the input numbers.

**Evaluation of 𝜋 using the Wallis product**

● **Observations and analysis**

While calculating Wallis product it has been observed that when number of terms is smaller till 1000 Karatsuba performs well, but after that it takes significant time. The increase in time is a resultant of increase in recursion depth and multiple array creation for storing the result of each recursion. Since input size is increasing by 1000 times in each iteration, for 20,000 terms total input length would be of 20 million. Which means recursion depth = lg(10000000) which is approximately 24. Due to this overhead the performance of the algorithm reduces.

**Calculation of pi using Walli’s product, product calculated using Karatsuba’s method.**

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Terms | Input Length | Time | Approximation of **𝜋** |
| 5 | 5000 | 0.243 | 3.0021759545569069378593188116997640807164616688426212235736045259854783664307473831283355092878902402 |
| 10 | 10000 | 0.409 | 3.0677038066434989710316881013507949531410011157812743262537743378242104467713918505245860136557913701 |
| 20 | 20000 | 1.286 | 3.1035169615392332650699784041307704915358741372044805584345276033252795748829606139614612223891315624 |
| 40 | 40000 | 3.146 | 3.1222603264214369745506401285472667940517831681780566984020750141510046112613297294234149117234982952 |
| 100 | 100000 | 12.34 | 3.13378749062816224125103608246234081156608595602494902564208742075050356140893730528171250988894920467 |

**Calculating pi using walli’s product, product calculated using BigInterger multiplication.**

|  |  |  |
| --- | --- | --- |
| Number of Terms | Input Length | Approximation of **𝜋** |
| 1000 | 1000000 | 3.1414355935899081924826868168434560782933074973551337070711883072045299573049622162806563754695757347 |
| 5000 | 5000000 | 3.1414355935899081924826868168434560782933074973551337070711883072045299573049622162806563754695757347 |
| 10000 | 10000000 | 3.1415141186819220469785580507138775513425133394700024394455997602027357506449179433722141754241393658 |
| 20000 | 20000000 | 3.1415533849087742495740774966468064224058479387166511355064078607430829866544842086105866456953642677 |

It has been observed that Walli’s convergence to pi is very slow and time consuming. We were able to calculate pi approximately to 20,000 terms. Beyond 20k terms the time taken to calculate the approximation of pi is huge, which is why the observation has not been captured beyond that.

● **Results & Mathematical Analysis**

**Part - 3(Madhava-Leibniz-Gregory):**

We utilised the "BigDecimal" class in Java to execute this task, which provides precise decimal values with arbitrary precision. Our approach involves using corrected Madhava terms to modify the series of pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + 4/13 - 4/15 + 4/17 - 4/19… to generate approximations of the value of pi. We tested this method using 4 terms, 3 of which were already provided in the project. The table below displays a comparison between the value of pi and the approximated pi value (with Madhava term), highlighting the differences between them.

**1st term: 1/N**

|  |  |  |  |
| --- | --- | --- | --- |
| Number Of Terms | Basic Pi value | Approximate Pi Value | Difference |
| 100 | 3.1**3**1592903558552764307414238276920516403054384406575651389662076501524624 | 3.1**4**1592903558552764307414238276920516403054384406575651389662076501524624 | 0.010000000000000000000000000000000000000000000000000000000000000000000000 |
| 10000 | 3.141**4**92653590043238459518383374815378787013642744180460513479805474395852 | 3.141**5**92653590043238459518383374815378787013642744180460513479805474395852 | 0.000100000000000000000000000000000000000000000000000000000000000000000000 |
| 1000000 | 3.14159**1**653589793238712643383279190384197170352500105815564788342357154440 | 3.14159**2**653589793238712643383279190384197170352500105815564788342357154440 | 0.000001000000000000000000000000000000000000000000000000000000000000000000 |
| 100000000 | 3.1415926**4**3589793238462643633279502884197138149375105820984475842307825632 | 3.1415926**5**3589793238462643633279502884197138149375105820984475842307825632 | 1.0000000000000000000000000000000000000000000000000000000000000000E-8 |

Based on the observation above,the First corrected term provide correct approximation value for the 100000000 number of terms until 18 decimal places 3.1415926**5**358979323846264...

while for the 100 terms the approximation is until 6 decimal places 3.141592..

**2nd term:1/(N+1(1/(4\*N)))**

|  |  |  |  |
| --- | --- | --- | --- |
| Number Of Terms | Basic Pi value | Approximate Pi Value | Difference |
| 100 | 3.1**3**1592**90**35**5855**2**76430741423827692051640305438440**6**575**6**51389662076501524**624 | 3.1**4**1592**65**35**6480**2**60806132039062311186161942397516**6**806**6**45614806447892239**624 | 0.009999750006249843753906152346191345216369590760230994225144371390715000 |
| 10000 | 3.141**4**926535**9004**32384**59518**38337**48153**787**870**1**364**27**4**4**180460**5**13479**8**05474**3**95**852 | 3.141**5**926535**8979**32384**60143**38337**32528**787**909**198927**3**4**414835**5**37893**8**67913**3**60**852 | 0.000099999999750000000624999998437500003906249990234375024414062438965000 |
| 1000000 | 3.14159**1**653589793238**71**2643383279**1903**841971703**52500**10581556**478834**235715**4**440 | 3.14159**2**653589793238**46**2643383279**2528**841971703**36875**10581556**869459**235715**3**440 | 9.99999999999750000000000062499999999984375000000003906249999999000E-7 |

The second corrected term above provide correct approximation value for the 100000000 number of terms until 30 decimal places 3.141592653589793238462643383279…

while for the 100 terms the approximation value is until 10 decimal places 3.1415926535…

**3rd Term:1/(N+(1/(4\*N+(4/n))))**

|  |  |  |  |
| --- | --- | --- | --- |
| Number Of Terms | Basic Pi value | Approximate Pi Value | Difference |
| 100 | 3.1**3**1592**90**35**585527**64307**41**4**238276920516403**0**54**38**4**406**57**56**5138966207650**1**524**624 | 3.1**4**1592**65**35**897988**58545**63**4**460749111492531**0**38**38**6**406**32**56**8263575631472**1**746**624 | 0.009999750031246094238220222472190976127984001999750031246094238220222000 |
| 10000 | 3.141**4**926535**9004**323845951838337481537878701364274418046051347980547439**5**852 | 3.141**5**926535**8979**323846264338333575287927529488**6**64066491180742416430009**1**852 | 0.000099999999750000003124999960937500488281243896484451293944358825696000 |
| 1000000 | 3.14159**1**653589793238**71**2643383279**1903**841971**70352500**10581**556478834**235**715**4440 | 3.14159**2**653589793238**46**2643383279**5028**841971**69961875**10581**605306959**235**654**4440 | 9.99999999999750000000000312499999999609375000000488281249999390000E-7 |
| 100000000 | 3.1415926**4**3589793238462643**63**32795028841971**3814**937510582098**447584**2307825632 | 3.1415926**5**3589793238462643**38**32795028841971**6939**937510582098**056959**2307825632 | 9.999999999999999750000000000000031249999999999996093750000000000E-9 |

Based on the observation above,the 3rd corrected term provide good approximation value for the 100000000 number of terms until 55 decimal places 3.1415926535897932384626433832795028841971693993751058209…

while for the 100 terms the approximation value is until 14 decimal places 3.14159265358979

**4th term: 1 / (N + (1 / (4 \* N + (4 / (N + (9 / (4 \* N)))))))**

|  |  |  |  |
| --- | --- | --- | --- |
| Number Of Terms | Basic Pi value | Approximate Pi Value | Difference |
| 100 | 3.1**3**1592**90**35**585527643074**1**423827692**0**516403054**3**844065**7**5651389662076501524**624 | 3.1**4**1592**65**35**897932362165**1**839733254**0**154841740**3**780911**7**7326327549394414638**624 | 0.009999750031223620086879496561236060410268115950118519639702612928440000 |
| 10000 | 3.141**4**926535**9004**32384**5951**83**83374815378**7**87013642744180460513**4**79**8**0**54**74395**852 | 3.141**5**926535**8979**32384**626433**83**27950288194**7**169787500043341299**4**95**8**3**54**46547**852 | 0.000099999999750000003124999735937526363278607177999332517381801918515000 |
| 1000000 | 3.14159**1**653589793238**71**2643383279**1903**841971**70352500**10581**556**4**78834**23**57154**440 | 3.14159**2**653589793238**46**2643383279**5028**841971**69399375**10581**872**4**94459**23**46630**440 | 9.99999999999750000000000312499999997359375000026363281249735718000E-7 |
| 100000000 | 3.1415926**4**3589793238462643**63**32795028841971**3814**93751058209**8**4**47584**2307825632 | 3.1415926**5**3589793238462643**38**32795028841971**6939**93751058209**7**4**94459**2307825632 | 9.999999999999999750000000000000031249999999999973593750000000000E-9 |

Based on the observation above, the 3rd corrected term provide correct approximation value for 100000000 number of terms until 67 decimal places 3.1415926535897932384626433832795028841971693993751058209749445923078..

For the 100 terms the correct approximation value is until 16 decimal places 3.1415926535897932..

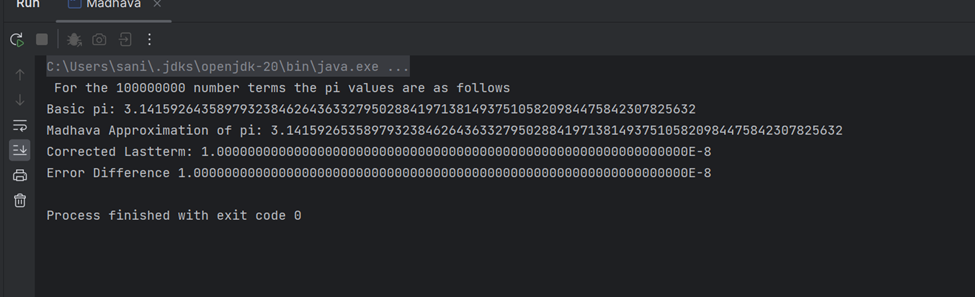
● **Conclusion**

For all the corrected terms the approximate value is getting closer by increasing the number of terms, which can be noticeable with the highlighted numbers,the difference decreases as the number of terms increases .

By looking at the observation we can say that the 4th term provides a good approximation value compared to the other 3 terms.

The first term does not provide much approximation to the pi value as much as 2nd, 3rd and 4th does, which can be noticeable in the tables.

The corrected term is same as the Difference error value, which you can see in the below Snapshot:



We have also experimented pi value using 100000000 terms up to 200 decimal places,our finding indicates that 4th term provide a more accurate approximation compared to the other correction terms.(3rd)

Evidence is represented by the highlighted digits(below).

**4th term:** 3.14159265358979323846264338327950288419716939937510582097494459230781640403620899862803870659211706797583604549765731872877426453139601929990162903110949732869260667279731129879045909512015888860850516

**3rd term:**

3.14159265358979323846264338327950288419716939937510582098056959230781640087605274862803975903352331797554953915000106881512247009780224313238301330846111033806913254936842953056383625960215678384907924

**PART 4;(Neelkanth Series)**

We found another series which approximates the value of pi for the specified number of times which is Known as “Neelkanth Series”. It is relatively easy to implement in a code. The series looks something like this

Pi = 3+4/(2\*3\*4)-4/(4\*5\*6)+4/(6\*7\*8)-4/(8\*9\*10)+……

The denominator is multiplication of 3 consecutive numbers starting with a multiple of 2 except for the first term.

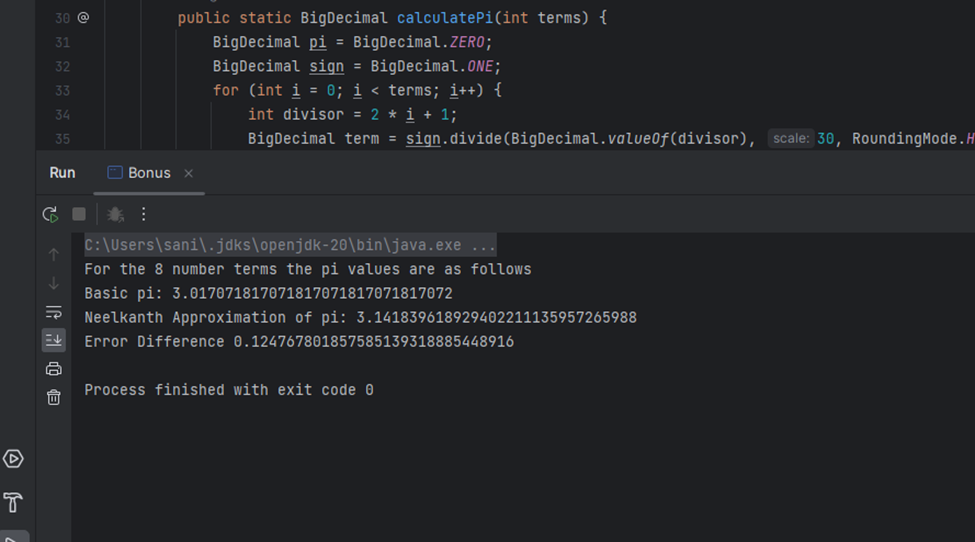
|  |  |  |  |
| --- | --- | --- | --- |
| Number Of Terms | Basic Pi value | Approximate Pi Value | Difference |
| 100 | 3.1**3**1592**903558552764307414238**2**769205**1**640305438440**6**57565138966207650152462**4 | 3.1**4**1592**889142081080420272145**2**070626**1**956444245996**6**18704097895121469926381**4 | 0.009999985583528316112857906930142103161388075559611389589289138197739190 |
| 10000 | 3.141**4**92653590043**23**8**45**95**18383374815378787013642744180460513479805474**3**95852** | 3.141**5**92653590043**08**8**51**95**01512498877145993511008838308173907914453809**3**24221** | 0.000099999999999850059983129124061767206497366094127713394434648334928369 |
| 1000000 | 3.14159**1**65358979323871264**33**832**79**1903**8419717035**2**5**0**010581556**4**788342357154440** | 3.14159**2**65358979323871264**18**832**85**1903**6732221160**2**4**0**626227031**4**353226078323693** | 9.99999999999999998500005999983125041249906156454749564883721169253E-7 |
| 100000000 | 3.1415926**4**3589793238462643633279**502**884197**1**381493**75105**82**09844**758**4**2**307825632** | 3.1415926**5**3589793238462643633279**487**884197**7**381493**58230**82**13969**758**3**2**923444741** | 9.999999999999999999999985000000599999983125000412499990615619109E-9 |

Based on the observation above for the Neelkanth Series, correct approximation value for 100000000 number of terms until 25 decimal places 3.1415926**5**35897932384626433..

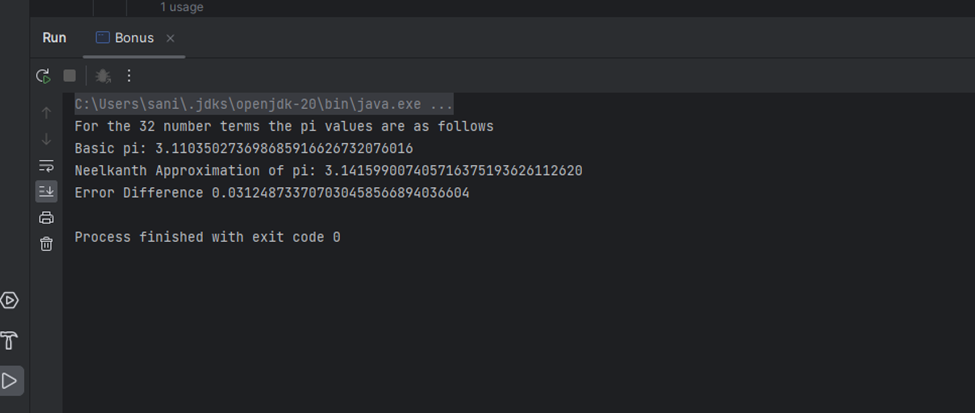
For the 100 terms the correct approximation value is until 6 decimal places 3.141592..

This series required a larger number of terms to approximate pi value.

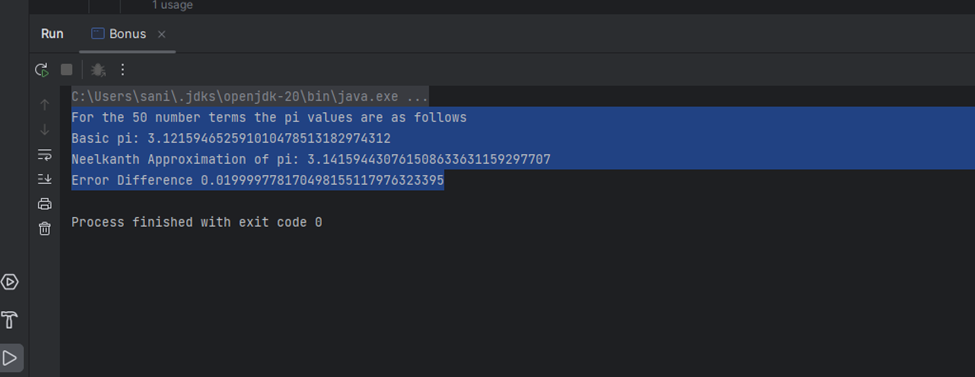
below are the snapshots for the smaller number of terms up to 20000, the approximated values are highlighted green.

****

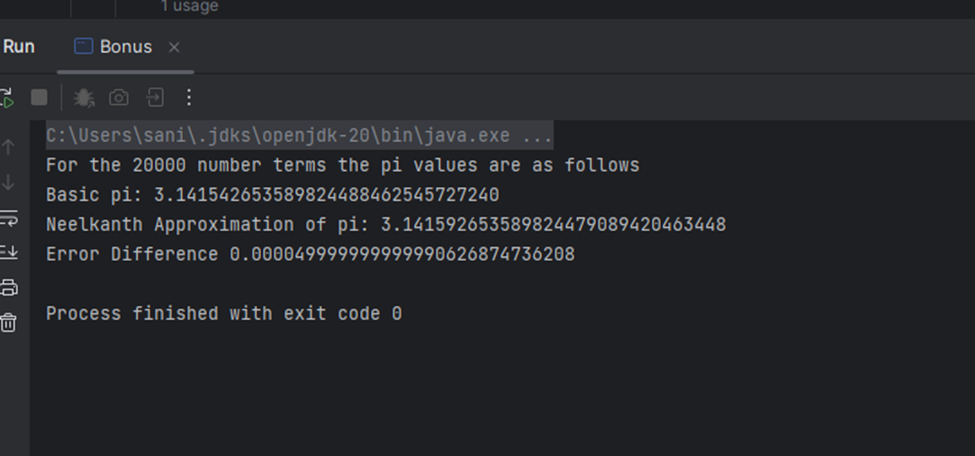
**Neelkanth Approximation of pi(8 terms): 3.141839618929402211135957265988**

****

**Neelkanth Approximation of pi(32 terms): 3.141599007405716375193626112620**

****

**Neelkanth Approximation of pi(50 terms): 3.141594430761508633631159297707**

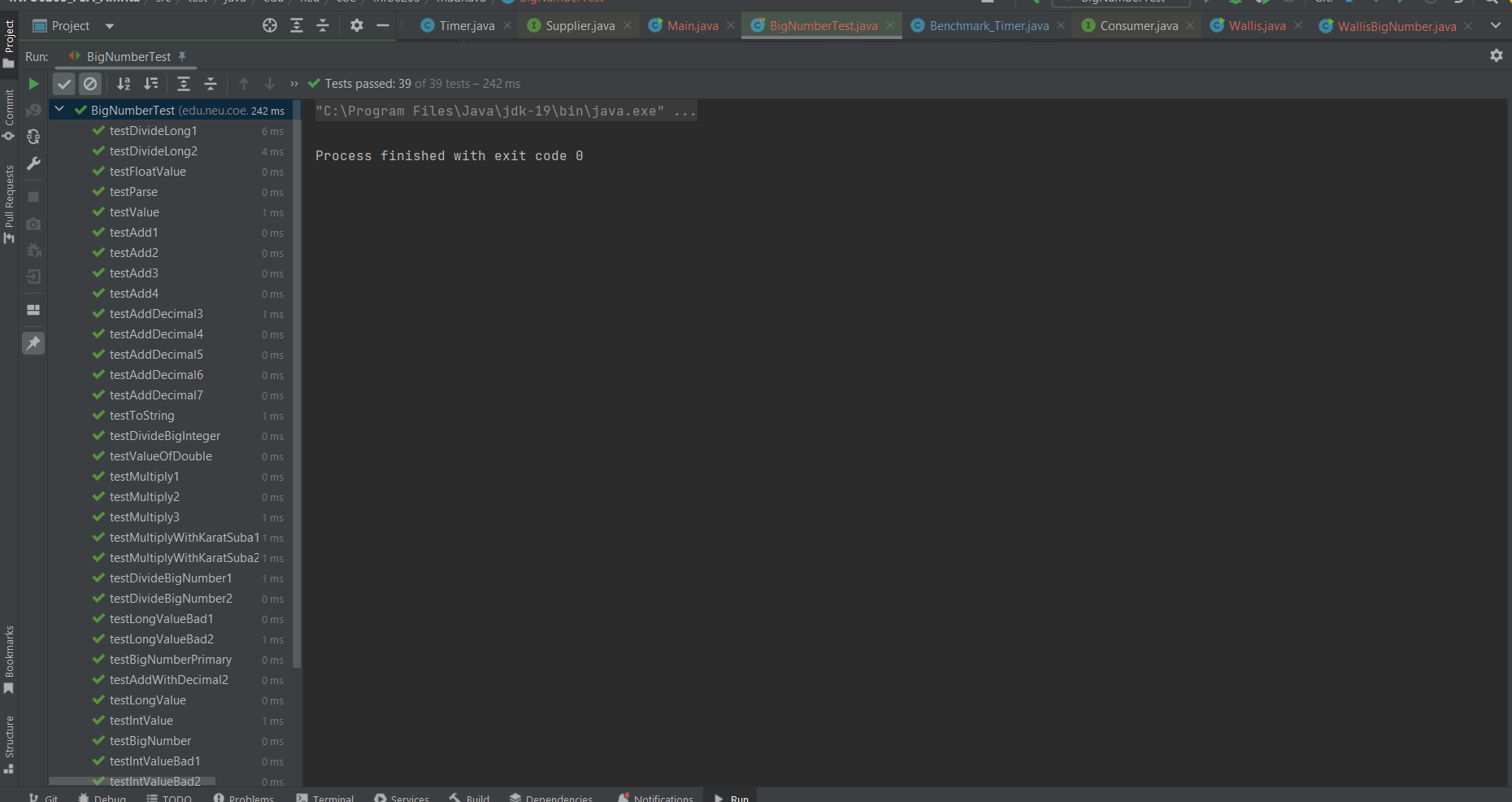
****

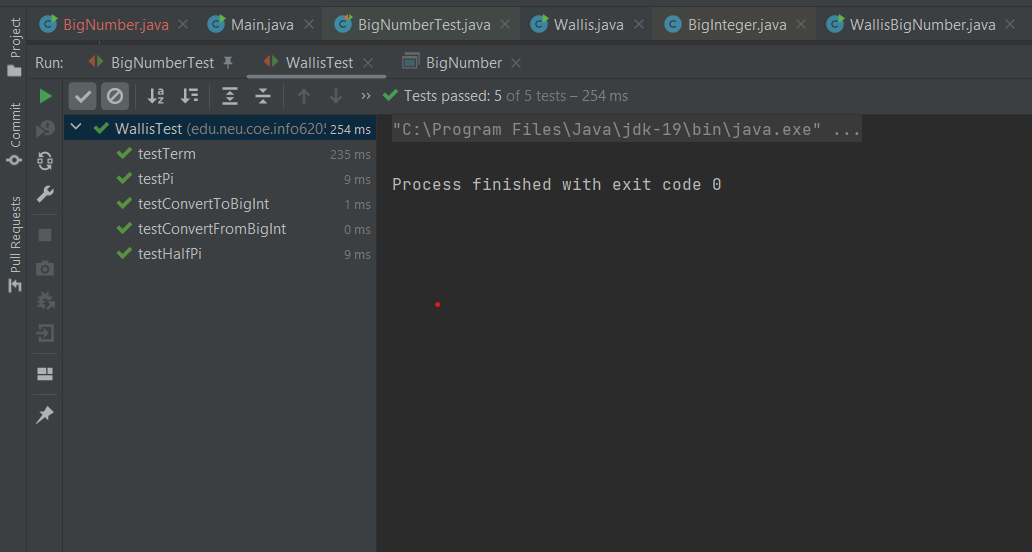
**Neelkanth Approximation of pi(20000terms):3.141592653589824479089420463448**

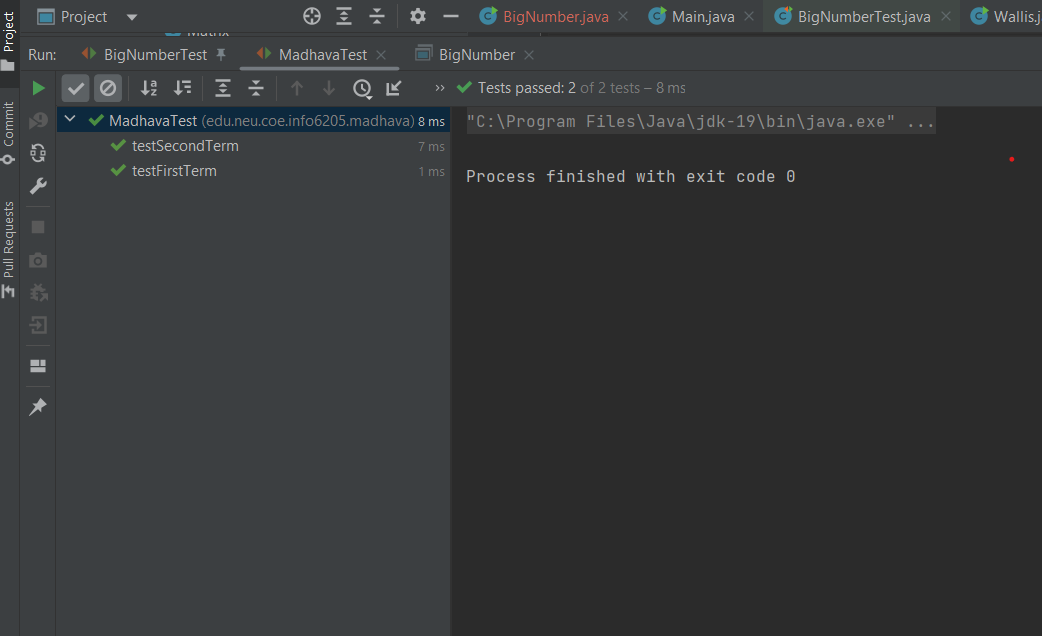
**Conclusion**

By observing the above table it can be seen that Neelkanth series provides an approximate value with fewer decimal places , as the number of terms increases the approximate value is also increased by the number of decimal places,When compared to the Madhava-Leibniz series, the Neelkanth series is not as effective for a small number of terms. The Neelkanth series requires a larger and larger number of terms to provide the best approximate value.

* **Unit tests**

The following unit testes were run :****

****

****

* **References**

<https://vixra.org/pdf/2302.0056v1.pdf>

<https://en.wikipedia.org/wiki/Karatsuba_algorithm>

<https://en.wikipedia.org/wiki/Wallis_product>

[**https://en.wikipedia.org/wiki/Madhava%27s\_correction\_term**](https://en.wikipedia.org/wiki/Madhava%27s_correction_term)