**INFO 6205 Summer 1 2023 Project**

Team 2

Members: Xinzhuo Liu, Chenghao Shi

**Part0 (By Chenghao Shi):**

* Implementation

A screenshot of a computer program

Description automatically generated with medium confidence

Here is my divide method for two BigNumbers. I implemented it with three main steps. Firstly, I transformed two BigNumbers into two BigDecimals and then get the result of their division. After that, I get the whole part and decimal part separately. Then I transform the decimal part into an int array. Finally, decide the sign of result based on the signs of dividend and divisor. After all of the steps, I would get all the parameters to construct the BigNumber result.

* Unit Test

A picture containing text, screenshot, font

Description automatically generated

I added a new test to test the division of 3.333333/9.999999. This test could check if the divide method could get the right answer if the result of a division has infinite decimals. Including this one, all the test for BigNumber class passed.

A screenshot of a computer program

Description automatically generated with medium confidence

**Part1(By Chenghao Shi):**

* Introduction
  + Aim:

develop an implementation of multiplication using Karatsuba’s method and compare the performance of this method with the regular multiplication method.

* Program

A screenshot of a computer program

Description automatically generated with medium confidence

I implemented Karatsuba’s method with three java functions. Multiplication of two decimal numbers could be seen as multiplying two integer numbers and putting the decimal point at an appropriate place. So in the karatsubMultiply function, I transform the multiplicands into two BigIntegers and then multiply them in the multiplyKaratsuba function to get the Integer value of the multiplication result. After that, in the adjustForDecimalPoints function, I construct the final BigNumber result based on the BigInteger result value, decimal places and sign. After complete Karatsuba’s method. I used Benchmark\_Time class to benchmark the time cost of two multiplication method by performing the same task, calculate **𝜋** using Wallis product.

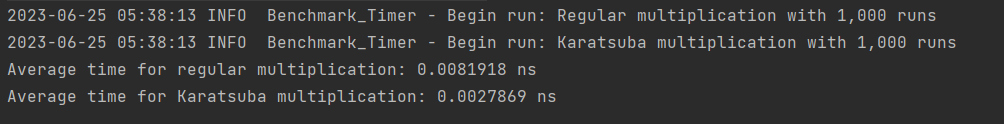
* Observations

When calculating **𝜋** with Wallis product, if just run for small times like 1 or 10 times and calculate 𝜋 with only 10 terms. The time costed by two methods are similar, shown in the below figure.

A screenshot of a computer

Description automatically generated with medium confidence

However, when repeat running for more times like 1000 times, performance of both method get great increase as more running times also means more warm up times. After sufficient warm up, the algorithm performance becomes better. Meanwhile, the difference between the average time of two methods become more obvious.



If I increase the number of terms calculated. The difference between the average time of two methods becomes much more grater.

A screenshot of a computer program

Description automatically generated with low confidence

* Unit tests

All tests passed for Wallis.java.

A screenshot of a computer

Description automatically generated with medium confidence

* Conclusion

With warm up, each method could have a performance increase. However, Karatsuba’s method is much better at dealing with multiplication between two numbers with high digits. In Java, Karatsuba’s method is used in libraries which need to deal with large numbers’ multiplication like BigInteger.

**Part2(By Chenghao Shi):**

* Introduction
  + Aim:

Evaluate the value of 𝜋 with Wallis product based on the code in the part 1.

* Program

A screen shot of a computer program

Description automatically generated with low confidence

I put a main function I the Wallis.java to perform the tasks of benchmark and 𝜋 value evaluation. To make it easy to observe, I only print the **𝜋** to 100 decimal places.

* Observations

This is my value of 𝜋 evaluated with Karatsuba’s method by multiplying 100, 200 and 300 terms. 

100terms



200terms



300terms

It can be detected that, with more terms, the evaluated value is more closer to the given value3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067.

* Unit tests

All tests passed for Wallis.java.

A screenshot of a computer

Description automatically generated with medium confidence

* Conclusion

By calculating more terms, the value of **𝜋** would become more precise.

**Part3 (By Xinzhuo Liu):**

* Introduction
  + Aim:

To find another correction term for MGL series expect for the three terms provided in the requirement.

* Program
  + Algorithm

As mentioned in the video, the fourth correction term was given. So I tested the term to see the result. The term is:

A picture containing font, text, screenshot, symbol

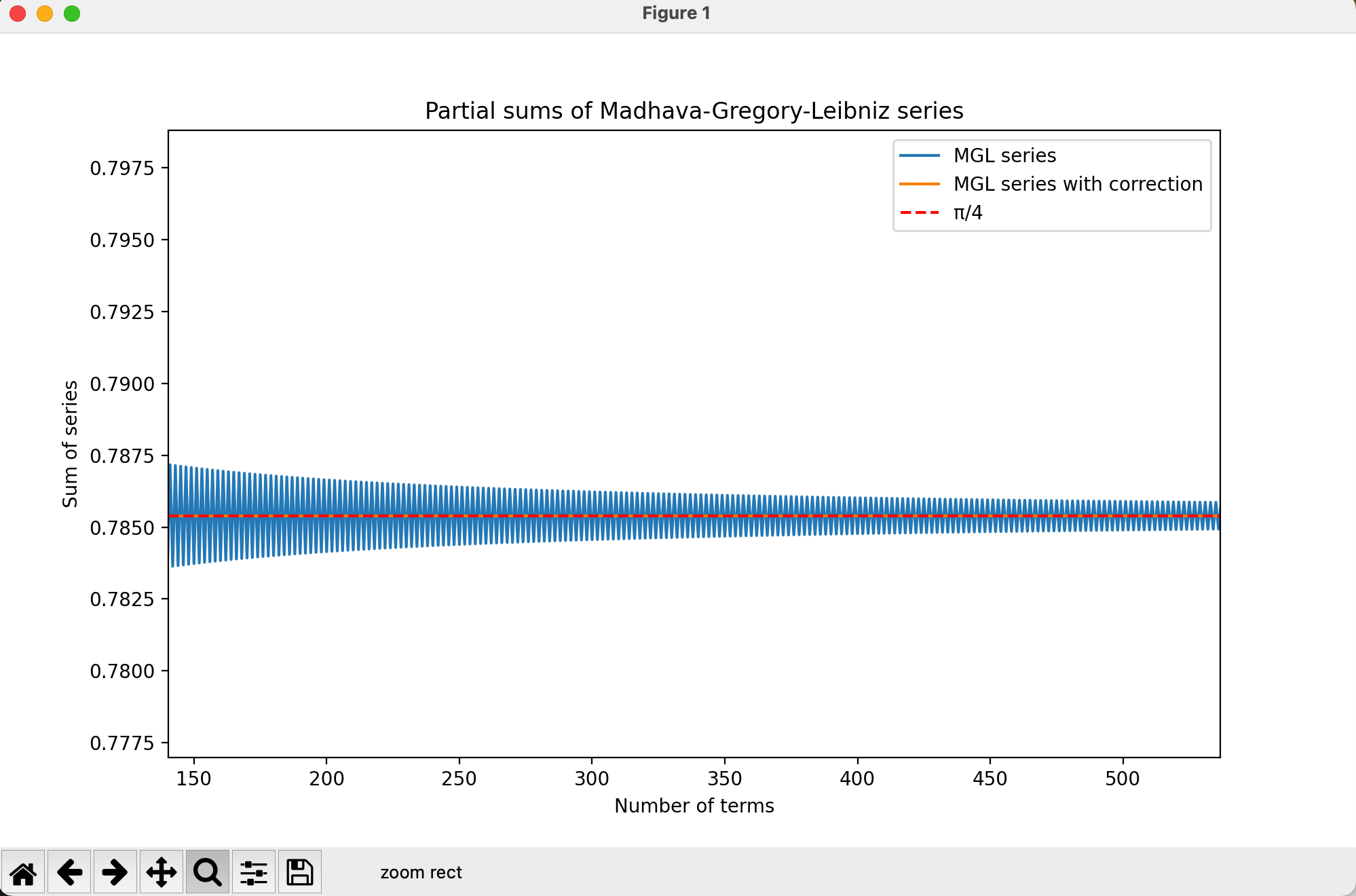
Description automatically generated

* Observations & Graphical Analysis

Using python to draw a graph to compare the difference between MGL series and MGL series after added a correction term we mentioned in the Algorithm, we can clearly see the difference.

A screen shot of a graph

Description automatically generated with medium confidence



A screenshot of a computer

Description automatically generated with medium confidence

So, we can see that the value of MGL series fluctuates up and down from pi/4, and after adding the correction term, the error between value of MGL series and pi/4 greatly reduced. (The black area is the gap/difference between MGL series and MGL series after corrected)

* Results & Mathematical Analysis

Using Java, to ensure that the correction terms make sense to reduce the error between MGL series and pi/4, I compare the MGL series and that after being corrected. I found that the fourth correction term makes sense because the error is smaller than 1E-6 when n is 1000 (such a small number).

* Unit tests
* @Test  
  public void testQuarterPi() {  
   *assertEquals*(Rational.*apply*("0"), Madhava.*quarterPi*(1, Madhava::*termFirst*));  
   *assertEquals*(Rational.*apply*("7/6"), Madhava.*quarterPi*(2, Madhava::*termFirst*));  
   *assertEquals*(Rational.*apply*("1/5"), Madhava.*quarterPi*(1, Madhava::*termSecond*));  
   *assertEquals*(Rational.*apply*("58/51"), Madhava.*quarterPi*(2, Madhava::*termSecond*));  
   *assertEquals*(Rational.*apply*("1/9"), Madhava.*quarterPi*(1, Madhava::*termThird*));  
   *assertEquals*(Rational.*apply*("142/123"), Madhava.*quarterPi*(2, Madhava::*termThird*));  
   *assertEquals*(Rational.*apply*("5/27"), Madhava.*quarterPi*(1, Madhava::*termFourth*));  
   *assertEquals*(Rational.*apply*("90/79"), Madhava.*quarterPi*(2, Madhava::*termFourth*));  
  }  
    
  @Test  
  public void testTerm() {  
   *assertEquals*(Rational.*apply*("-22/27"), Madhava.*termFourth*(1));  
   *assertEquals*(Rational.*apply*("112/237"), Madhava.*termFourth*(2));  
   *assertEquals*(Rational.*apply*("-38/117"), Madhava.*termFourth*(3));  
  }  
    
  @Test  
  public void testPi() {  
   *assertEquals*(approximatePi.divide(4).doubleValue(), Madhava.*quarterPi*(1000, Madhava::*termFirst*).toDouble(), 1E-6);  
   *assertEquals*(approximatePi.divide(4).doubleValue(), Madhava.*quarterPi*(1000, Madhava::*termSecond*).toDouble(), 1E-6);  
   *assertEquals*(approximatePi.divide(4).doubleValue(), Madhava.*quarterPi*(1000, Madhava::*termThird*).toDouble(), 1E-6);  
   *assertEquals*(approximatePi.divide(4).doubleValue(), Madhava.*quarterPi*(1000, Madhava::*termFourth*).toDouble(), 1E-6);  
  }  
    
  @Test  
  public void testError() {  
   *assertTrue*(Math.*abs*(Madhava.*mglSeries*(1001).toDouble() - approximatePi.divide(4).doubleValue()) > Math.*abs*(Madhava.*quarterPi*(1000, Madhava::*termFirst*).toDouble() - approximatePi.divide(4).doubleValue()));;  
   *assertTrue*(Math.*abs*(Madhava.*mglSeries*(1001).toDouble() - approximatePi.divide(4).doubleValue()) > Math.*abs*(Madhava.*quarterPi*(1000, Madhava::*termSecond*).toDouble() - approximatePi.divide(4).doubleValue()));;  
   *assertTrue*(Math.*abs*(Madhava.*mglSeries*(1001).toDouble() - approximatePi.divide(4).doubleValue()) > Math.*abs*(Madhava.*quarterPi*(1000, Madhava::*termThird*).toDouble() - approximatePi.divide(4).doubleValue()));;  
   *assertTrue*(Math.*abs*(Madhava.*mglSeries*(1001).toDouble() - approximatePi.divide(4).doubleValue()) > Math.*abs*(Madhava.*quarterPi*(1000, Madhava::*termFourth*).toDouble() - approximatePi.divide(4).doubleValue()));;  
  }
* Conclusion

The fourth correction term should be:

A picture containing font, text, screenshot, symbol

Description automatically generated

* References
  + https://www.youtube.com/watch?v=ypxKzWi-Bwg&t=1375s

**Part4 (By Xinzhuo Liu):**

* Introduction
  + Aim:

To find if there are any other sequences to calculate approximation of pi

* Program
  + Algorithm

After searching, I found the most precise one is Nilakantha’s formula, which looks like:

A picture containing text, font, line, screenshot

Description automatically generated

* Observations & Graphical Analysis

From the unit test, I found that the error between the new sequence and pi is smaller than 1E-9 (much smaller than that of MGL series).

* Results & Mathematical Analysis

With 12 terms, the sequence gives out the result of pi = 3.141479689, which is only accurate to 3 decimal places. So the Nilakantha’s formula must be a reliable one to help approximate the value of pi.

* Unit tests
* @Test  
  public void testPi() {  
   *assertEquals*(approximatePi.doubleValue(), Nilakantha.*calculatePi*(1000).toDouble(), 1E-9);  
   *assertEquals*(approximatePi.doubleValue(), Nilakantha.*calculatePi*(3000).toDouble(), 1E-10);  
  }
* Conclusion

The new sequence is:

A picture containing text, font, line, screenshot

Description automatically generated

* References
  + http://www.maeckes.nl/Formule%20voor%20pi%20%28Nilakantha%29%20GB.html