INFO 6205 Summer 1 2023 Project

Team 2
Members: Xinzhuo Liu, Chenghao Shi

Part0 (By Chenghao Shi):

- Implementation

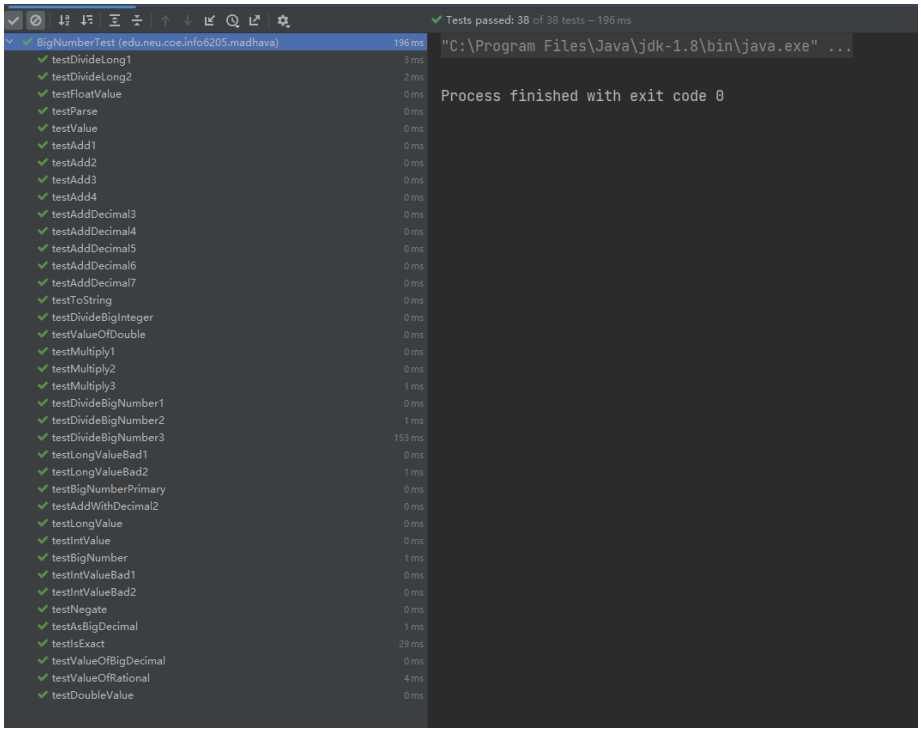
```
public BigNumber divide(BigNumber x) {
    if (x.decimals.length > 0) {
        // TODO implement long division.
          throw new BigNumberException("divide is not supported for division by non-whole numbers");
        BigDecimal dividend = toBigDecimal();
        BigDecimal divisor = x.toBigDecimal();
        BigDecimal res;
        try {
            res = dividend.divide(divisor);
        }catch (Exception e){
            res = dividend.divide(divisor, scale: 1000, RoundingMode.DOWN);
        }

        BigInteger resWhole = BigInteger.valueOf(res.longValue());
        BigDecimal dec = res.remainder(BigDecimal.ONE);

        if (dec.compareTo(BigDecimal.valueOf(0))>0){
            String decimal = dec+"";

            int[] decimals = new int[decimal.length()-2];
            for (int i = 2; i<decimal.length(); i++){
                decimals[i-2] = Integer.valueOf( s: decimal.charAt(i)+"");
            }

            return new BigNumber(resWhole, decimals, sign: sign==x.sign);
        }

        return new BigNumber(resWhole, new int[]{}, sign: sign==x.sign);
    } else if(x.isWhole() && x.whole.equals(BigInteger.valueOf(0))){
        throw new BigNumberException("divisor can't be 0");
    } else {
        BigNumber quotient = divide(x.whole);
        return x.sign ? quotient : quotient.negate();
    }
}
```

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

```
@Test
public void testDivideBigNumber3() {
    BigNumber target = BigNumber.value( whole: 3,  decimals: 333333,  sign: true);
    BigNumber divisor = BigNumber.value( whole: 9,  decimals: 999999,  sign: true);
    assertEquals(BigNumber.value(Rational.apply(BigInt.apply( i: 3), BigInt.apply( i: 9))), target.divide(divisor));
}
```

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.



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.

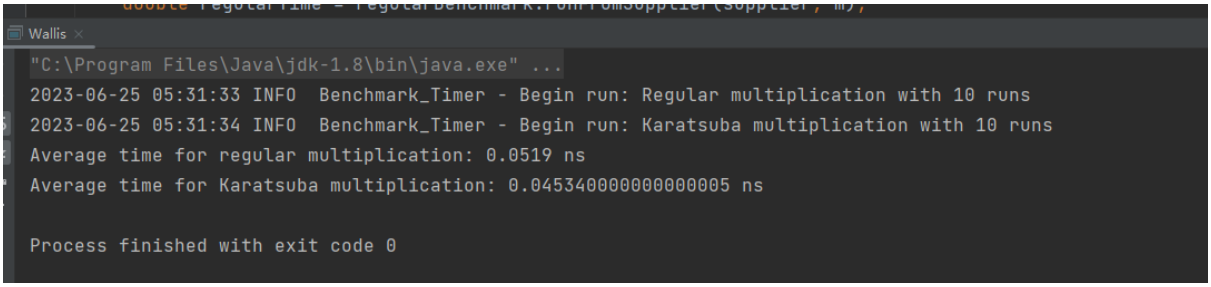
- Program

```
public BigNumber karatsubaMultiply(BigNumber that) {
    int decimalPoints = this.decimals.length + that.decimals.length;

    BigInteger x = new BigInteger(this.toString().replace( target: ".",  replacement: ""));
    BigInteger y = new BigInteger(that.toString().replace( target: ".",  replacement: ""));

    return adjustForDecimalPoints(multiplyKaratsuba(x, y), decimalPoints,  sign: sign==that.sign);
}

4 usages
public static BigInteger multiplyKaratsuba(BigInteger x, BigInteger y) {
    int N = Math.max(x.bitLength(), y.bitLength());
    if (N <= 2000) return x.multiply(y); // fallback to standard multiplication for small input numbers to avoid overhead of recursion

    N = (N / 2) + (N % 2); // bitLength/2 rounded up

    // x = a + 2^N b,   y = c + 2^N d
    BigInteger b = x.shiftRight(N);
    BigInteger a = x.subtract(b.shiftLeft(N));
    BigInteger d = y.shiftRight(N);
    BigInteger c = y.subtract(d.shiftLeft(N));

    // compute sub-expressions
    BigInteger ac = multiplyKaratsuba(a, c);
    BigInteger bd = multiplyKaratsuba(b, d);
    BigInteger abcd = multiplyKaratsuba(a.add(b), c.add(d));

    return ac.add(abcd.subtract(ac).subtract(bd).shiftLeft(N)).add(bd.shiftLeft( n: 2*N));
}

// support method to deal with decimals and sign of multiplication result
1 usage
private BigNumber adjustForDecimalPoints(BigInteger number, int decimalPoints, boolean sign) {
    String strNumber = number.toString();

    if (strNumber.length() <= decimalPoints) {
        while (strNumber.length() < decimalPoints)
            strNumber = "0" + strNumber;
        strNumber = "0." + strNumber;
    } else {
        strNumber = strNumber.substring(0, strNumber.length() - decimalPoints) + "." + strNumber.substring( beginIndex: strNumber.length() - decimalPoints);
    }

    return sign ? BigNumber.parse(strNumber) : BigNumber.parse(strNumber).negate();
}
```

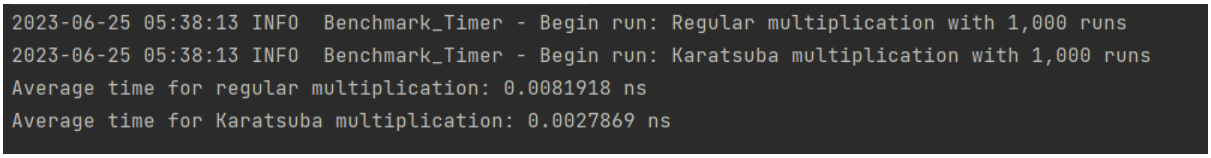
I implemented Karatsuba's method with three java functions. Multiplication of two decimal numbers could be seen as multiplying two integer numbers and putting the decimal point at an appropriate place. So in the karatsubMultiply function, I transform the multiplicands into two BigIntegers and then multiply 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

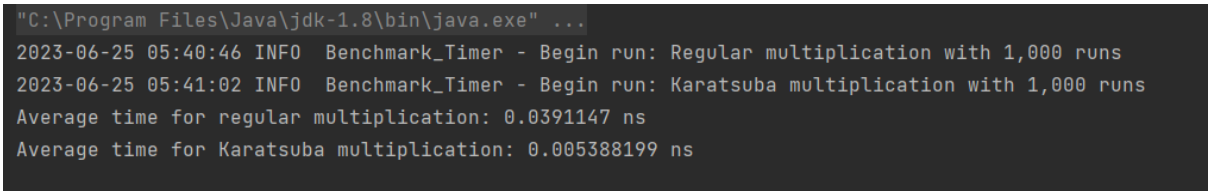
  When calculating π with Wallis product, if just run for small times like 1 or 10 times and calculate π with only 10 terms. The time costed by two methods are similar, shown in the below figure.



  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.

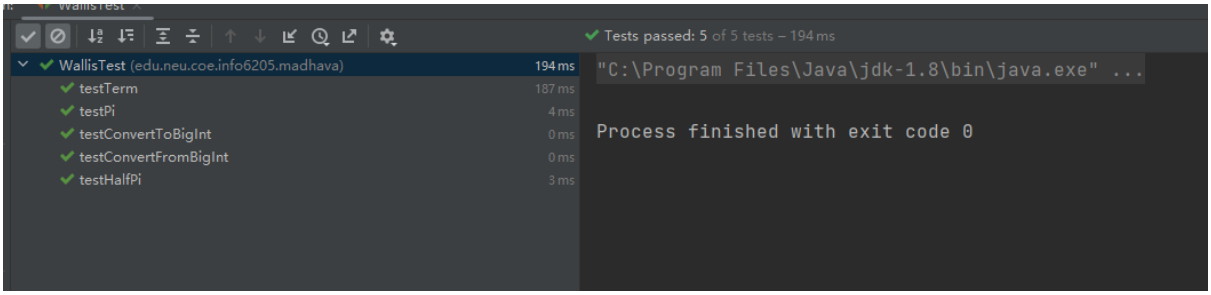


  If I increase the number of terms calculated. The difference between the average time of two methods becomes much more grater.



- Unit tests

  All tests passed for Wallis.java.



- 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

```
public static void main(String[] args) {
    // Create the BigNumber instances from 1 to 100 in a array
    BigNumber[] inputs = new BigNumber[100];
    for (int i = 0; i < 100; i++){
        inputs[i] = BigNumber.value(BigInteger.valueOf(i+1));
    }

    AtomicInteger j = new AtomicInteger( initialValue: 1);
    // Consumer function for regular multiplication
    Consumer<BigNumber[]> regularMultiply = arr -> {
        BigNumber res = BigNumber.parse( s: "2");
        while(j.get()<100){
            res = res.multiply(BigNumber.value(term(j.getAndIncrement())));
        }
    };
    AtomicInteger i = new AtomicInteger( initialValue: 1);
    // Consumer function for Karatsuba multiplication
    Consumer<BigNumber[]> karatsubaMultiply = arr -> {
        BigNumber res = BigNumber.parse( s: "2");
        while(i.get()<100){
            res = res.karatsubaMultiply(BigNumber.value(term(i.getAndIncrement())));
        }
    };

    // Create a supplier that returns an array of BigNumber instances
    Supplier<BigNumber[]> supplier = () -> inputs;

    // Now we create two Benchmark_Timer instances
    Benchmark_Timer<BigNumber[]> regularBenchmark = new Benchmark_Timer<>( description: "Regular multiplication", regularMultiply);
    Benchmark_Timer<BigNumber[]> karatsubaBenchmark = new Benchmark_Timer<>( description: "Karatsuba multiplication", karatsubaMultiply);

    // Run the benchmarks
    int m = 1000; // Number of times to run the benchmarks
    double regularTime = regularBenchmark.runFromSupplier(supplier, m);
    double karatsubaTime = karatsubaBenchmark.runFromSupplier(supplier, m);

    System.out.println("Average time for regular multiplication: " + regularTime + " ns");
    System.out.println("Average time for Karatsuba multiplication: " + karatsubaTime + " ns");

    BigNumber res = BigNumber.parse( s: "2");
    int k = 1;
    while(k<101){
        res = res.multiply(BigNumber.value(term(k++)));
    }
    System.out.println("The value of \uD835\uDF0B is: "+res.toString().substring(0,103));
}
```

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 $\boldsymbol{\pi}$ to 100 decimal places.

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

```
The value of π is: 3.133787490628162241251036082462340811566085956024949025642087420750503561408937305281712509888949204 67
```

100terms

```
The value of π is: 3.137677900950936093927530537246158310670574988978468845069501252043128393147085270684013726539737690 83
```

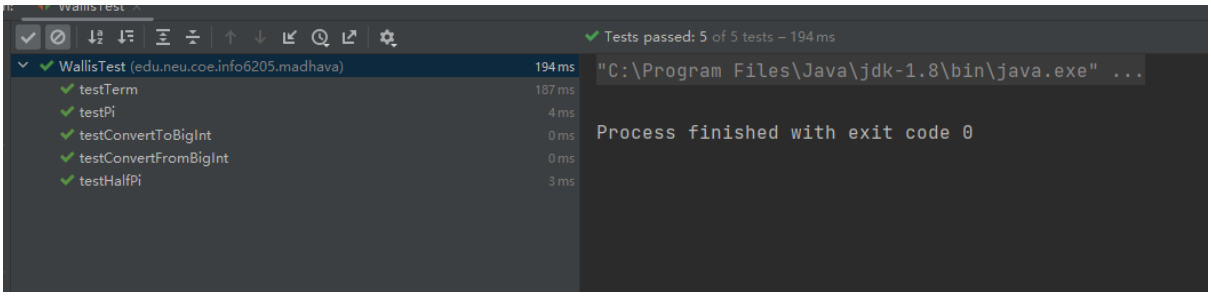
200terms

```
The value of π is: 3.13898010388212809523319278945689778970452320341354087784458768804057451961876014454526201796359606804
```

300terms

It can be detected that, with more terms, the evaluated value is more closer to the given value3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 7.

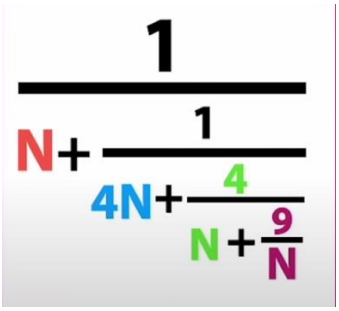
- Unit tests
  All tests passed for Wallis.java.



- 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:



- 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.

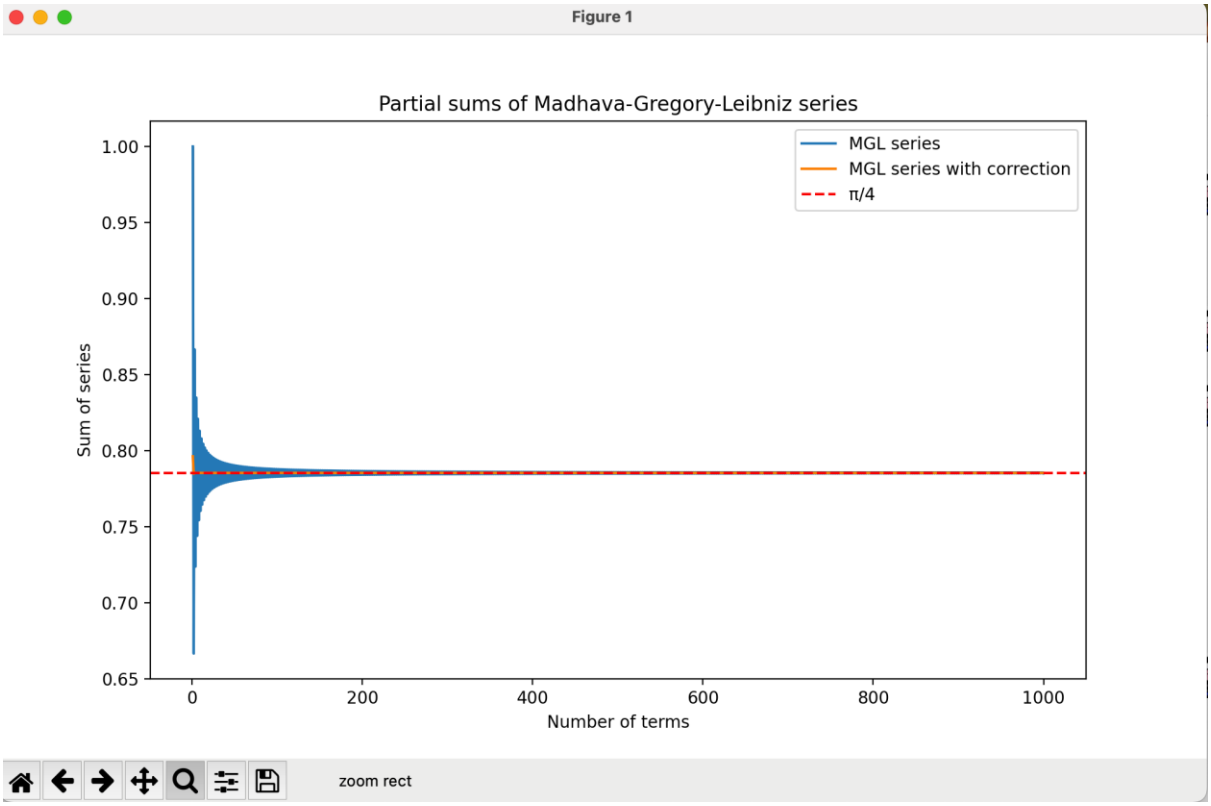


Figure 1
Partial sums of Madhava-Gregory-Leibniz series
MGL series
MGL series with correction
π/4
Sum of series
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
200
400
600
800
1000
Number of terms
zoom rect

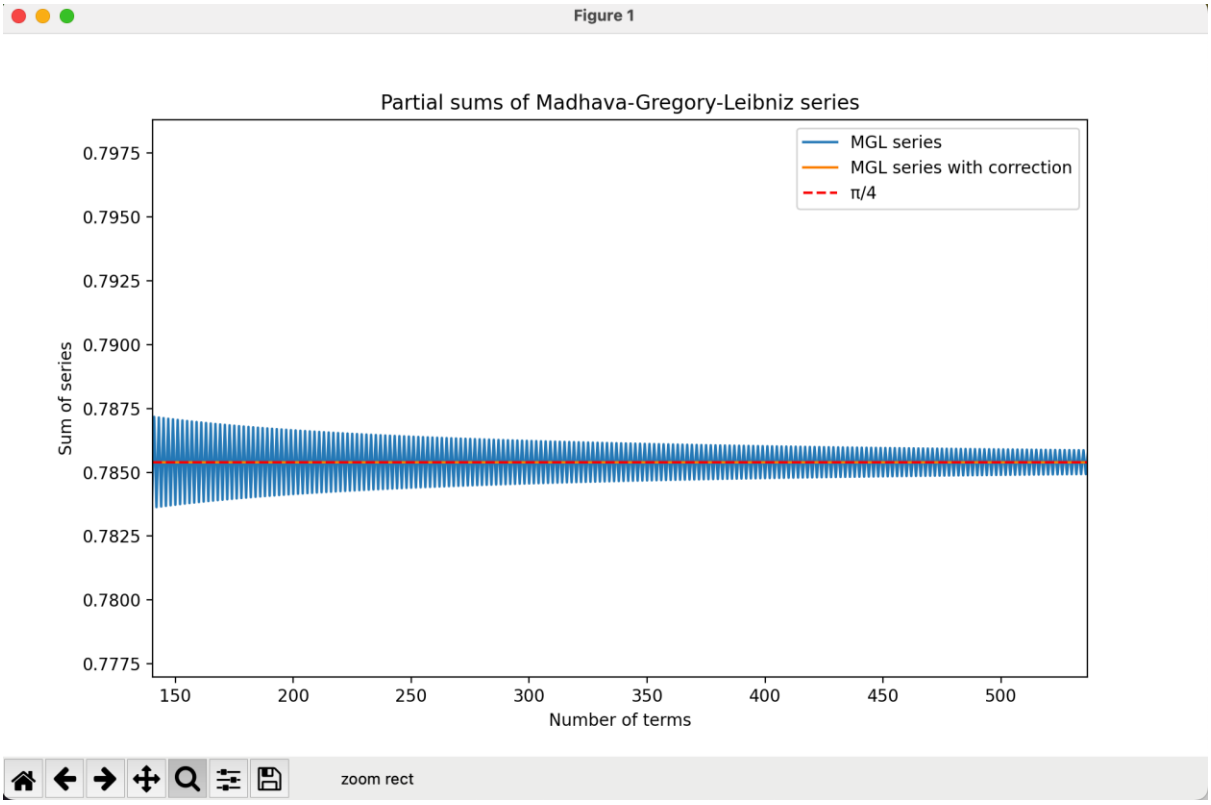
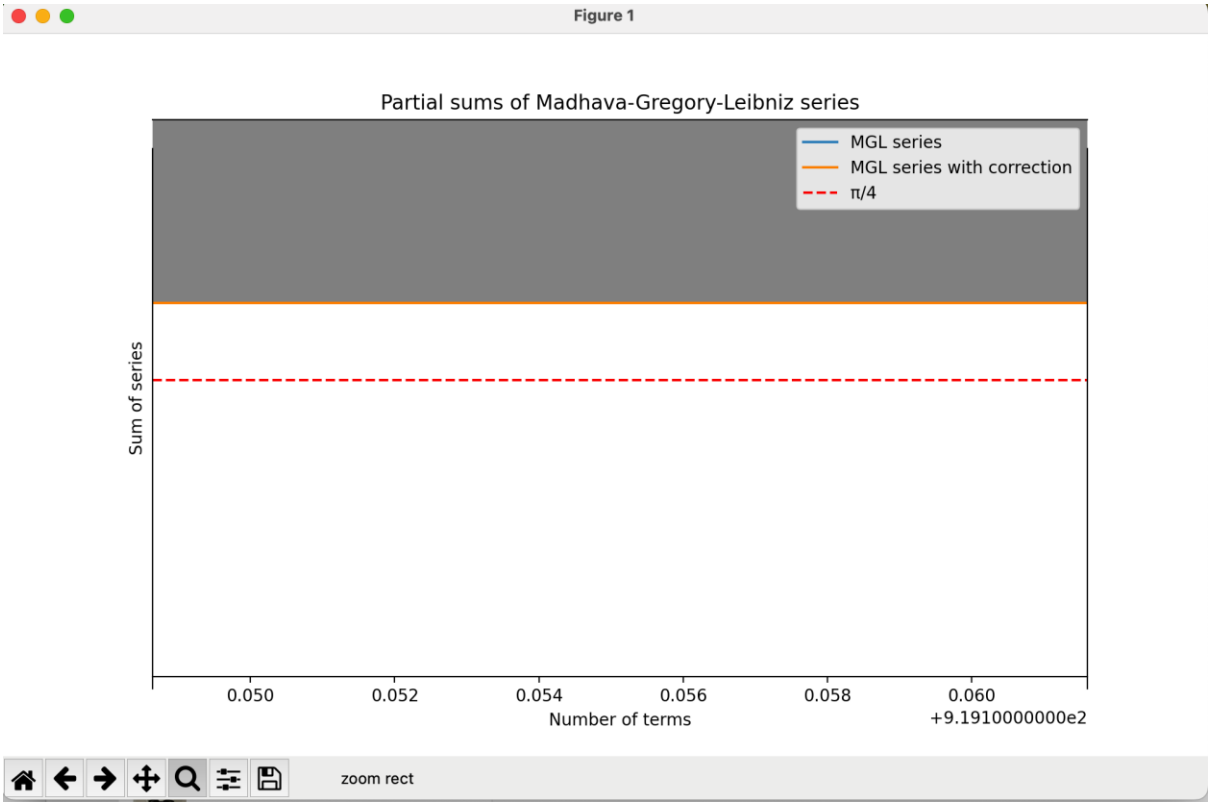


Figure 1
Partial sums of Madhava-Gregory-Leibniz series
MGL series
MGL series with correction
π/4
Sum of series
0.7975
0.7950
0.7925
0.7900
0.7875
0.7850
0.7825
0.7800
0.7775
150
200
250
300
350
400
450
500
Number of terms
zoom rect



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)

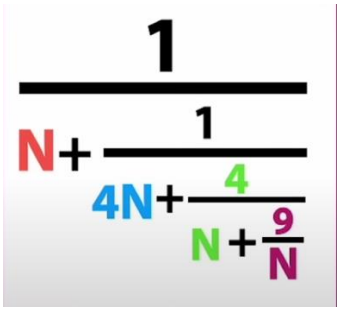
- Results & Mathematical Analysis
  Using Java, to ensure that the correction terms make sense to reduce the error between MGL series and pi/4, I compare the MGL series and that after being corrected. I found that the fourth correction term makes sense because the error is smaller than 1E-6 when n is 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:



- 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:

$$\frac{\pi-3}{4}=\frac{1}{2\times3\times4}-\frac{1}{4\times5\times6}+\frac{1}{6\times7\times8}-\cdots$$

- 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:

$$\frac{\pi-3}{4}=\frac{1}{2\times3\times4}-\frac{1}{4\times5\times6}+\frac{1}{6\times7\times8}-\cdots$$

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