Pi Calculator

Ways to do:

1- Using the Measurements of a Circle:

By this method we should make a perfect circle and measure circumference of the circle (let’s call it C) and also measure diameter of the circle (let’s call it D) and obtain the Pi number by equation $\pi = \frac{C}{D}$.

This is not a good way for us because it always has a quite big error for measuring circumference and diameter and we can’t use it to do multi thread tasks on java.

2- Using Gregory-Leibniz series:

Mathematicians have found several different mathematical series that, if carried out infinitely, will accurately calculate pi to a great number of decimal places.

For obtaining Pi with this method, we use this equation: $\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots$

This can be a good way for us because it makes possible using multi thread methods in java.

One of disadvantages to this code is that converges very slowly, meaning that it requires a very large number of terms to get accurate results.

After evaluating execution time of code, the following numbers obtained for 2^19 operations and 1000 precision: 827, 878, and 1007 milliseconds.

The way of implementation is that we have 8 threads which runs in a thread pool and each thread calculate 2^16 fractions.

3- Using a Limit:

We pick a large number. The bigger the number, the more accurate our calculation will be.
Plug the number, which we'll call x, into this formula to calculate
$pi = x * sin(\frac{180}{x})$.
The reason this is called a Limit is because the result of it is 'limited' to pi. As we increase our number x, the result will get closer and closer to the value of pi.

This almost likes the first way and it is not possible to use multi thread.

4- Try the Nilakantha series:

This is another infinite series to calculate pi that is fairly easy to understand. While somewhat more complicated, it converges on pi much quicker than the Leibniz formula.

The equation is $\pi = 3 + \frac{4}{2*3*4} - \frac{4}{4*5*6} + \frac{4}{6*7*8} - \frac{4}{8*9*10} + \cdots$

This likes the second way and its advantage is that it converges more quickly than the second way, but in total, it also converges slowly comparing to another algorithms.

After evaluating execution time of code, the following numbers obtained for 2^19 operations and 1000 precision :525, 641, and 656 milliseconds.

The way of implementation with this method likes the Gregory-Leibniz series and a little different in run method.

5- Bailey-Borwein-Plouffe (BBP) Formula:

The formula is:

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} (\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6})$$

The BBP formula is particularly useful for digit extraction algorithms but less efficient for computing large numbers of digits sequentially.

After evaluating execution time of code, the following numbers obtained for 2^10 operations and 1000 precision :313, 309, and 344 milliseconds.

NOTE: for the algorithms 2 and 4 I had a little mistake and 2 tests from 4 tests had failed and it

and it was because I should define every variable in BigDecimal format. I didn't know this problem and I taught I should increase the operations. So, it took a lot of time for computer to execute. So, the durations for algorithms 2 and 4 aren't exactly correct!

The best formula is BBP formula explained above. Because it converges more quickly than others and we can get appropriate result by summation of less number of fractions so it is faster also.

Semaphore

One of ways for handling race condition is using semaphores. But semaphores have a special difference with other ways. The difference is that by semaphores we can access a variable by multiple threads and just not one thread.

The way I implemented semaphore is obvious. I used Semaphore object in Controller and Operator classes and it let 2 threads to run at the same time.