**Study of a mathematical approach vs. brute force**

A brute force attack while effective can be quite time consuming. In an attempt to get the number of terms for a harmonic summation when n > 20 we run into computational time that takes minutes or even tens of hours. This is because a brute force approach to harmonic summation has a computational time that is exponentially growing as ‘n’ increases. This may be fine if looking for a summation value (ie. adding the harmonic numbers together until they equal some value) but if we are trying to just see how many iterations it takes to get ‘n’ there is a more elegant and much faster approach.

When we look at the equation for the harmonic summation:

We can see that the summation looks very similar to a Taylor series function. There is also a Taylor series for Euler’s constant that looks like this:

When we study Euler’s equation we can see that there isn’t much work to make it work for us with the harmonic summation when we are looking for the number of iterations ‘n’:

First take the original equation:

Greater than or equal to x comes from the user input number when comparing the summation of values to see if we must stop computing. In the brute force attack we would take the user input, add harmonic values until the accumulator is greater than or equal to the user’s input. That’s when we would know we must exit our loop and the count would be the number of iterations. Lets manipulate this formula:

Lets subtract ln(n) from both sides:

Now our left side is Euler’s constant :

We still need to manipulate this formula to solve for ‘n’ so lets continue by placing ln(n) by itself (add ln(n) and subtract from both sides:

Finally, to isolate ‘n’ we must raise ‘e’ by both sides:

‘e’ and ln() cancel each other out leaving us with:

This equation: , will give us an ‘n’ (iteration) value with an error only from our accuracy with calculating ‘e’. Through trial and error I found that the ‘e’ iteration count is accurate with no error after 100 iterations. This means we can calculate any iteration count after only 100 iterations. Which yielded an average time of 22000 nanoseconds or 0.00002 seconds whether calculating a harmonic sum of 1 or 44.

There are two issues with this approach, one computational and solvable through more study of the ‘LDMXCSR’ register (Load Streaming SIMD Extension Control/Status), and one which is not solvable because of the approach. The ‘LDMXCSR’ or Load Streaming SIMD Extension Control/Status register (32bit) is used for rounding control along with many other tasks. Our concern with this register is that our formula yields an ‘n’ value which is the number of terms in the harmonic summation. No matter what decimals follow the whole value, the whole value is our only concern. This means that we must take the floor or round down no matter what the decimal value. The standard rounding control follows the convention of round up if decimal value is greater than 0.5 and round down if below 0.5.

When we compute this formula we must do it on the SSE/AVX registry because of the FPU values we are dealing with. When we are finished calculating our ‘n’ value we must transfer it back into an ‘int’ register as we are calculating a number or iterations and this is a whole number. We use the instruction ‘cvtsd2si’ to convert our FPU value back onto an int register. This is where our ‘LDMXCSR’ register comes in. The ‘LDMXCSR’ register controls how this value from our SSE/AVX register to the int registery is rounded. This in inadequate for us because when we use the instruction ‘cvtsd2si’, it uses conventional rounding techniques. For us to avoid an possible error of 1 (if our ‘n’ value is say 2.677, this rounds to 3, we need 2) we must set the ‘LDMXCSR’ register to always round down. Bits 13 and 14 act as rounding controls for the ‘LDMXCSR’ register and the setting for truncate is 3 (bits 13 and 14 both set to 1). I was unable to access this register in my code (even though there was much trial unfortunately I found the documentation limiting).

The current program has a possible error of 1 (due to the ‘LDMXCSR’ register not set to truncate) but if we study the quickest approach to brute force we find that vector processing yields a much greater error. To calculate a brute force approach with vector processing we see that the incrimination of the counter will be equal to the amount of parallel processing being done. This means that if your vector processing is doing a 4-way calculation your possible error is +/-4. Moreover, the brute force vector processing will yield an error of (400%) compared to the Euler’s constant approach you are still looking at multiple hours of computation time for a summation of 28 or more where as Euler’s computational time isn’t effected.

As I stated 3 paragraphs ago there are two problems with this approach. The first problem I outlined and showed a solution to and with further research of the ‘LDMXCSR’ register the error can be solved. The other problem is we have no calculated sum. Our formula is geared for one thing and one thing only, to calculate the number of iterations. If we wanted to calculate the actual sum however we still would have to add the harmonic sum one-by-one and see what the yield is. This means that when this program displays the sum the number of iterations is accurate however there is no calculated sum so the current version of the program just spits out the users entered number. The number of iterations yielded however will give you that correct sum.

The project also showed that the largest number we can calculate iterations for is 44.2 because the number of iterations produced is just shy of 2^63-1 which is the maximum value for the int register. 44.2 produces 8813218186917735424 terms and 44.3 overflows the register and the output looks like

this: -9223372036854775808 terms.

This program is ideal for any computation where the number of iterations is necessary. The amount of computational time in the Euler’s constant approach is O(100) where as the brute force attack on the harmonic summation produces a 400% higher error (when doing 4-way parallel vector processing) and a computational time of (exponential). Below I included a comparison of the test cases between brute force and Euler’s constant approach.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | **Brute Force (sec)** | **‘n’ yield**  **BF**  **(Terms)** | **Euler’s Constant (sec)** | **n’ yield**  **Euler’s**  **(Terms)** |
| **2.5** | 0.00001110 | 7 | 0.00002218 | 7 |
| **4.0** | 0.00002337 | 31 | 0.00002226 | 31 |
| **5.0** | 0.00002957 | 83 | 0.00002219 | 83 |
| **7.5** | 0.00013608 | 1015 | 0.00002215 | 1015 |
| **8.5** | 0.00033515 | 2759 | 0.00001714 | 2759 |
| **10.0** | 0.00143118 | 12367 | 0.00001705 | 12367 |
| **15.0** | 0.11435806 | 1835421 | 0.00002200 | 1835421 |
| **20.0** | 15.73389918 | 272400600 | 0.00002212 | 272400600 |
| **24.0** | 860.68477857 | 14872568832 | 0.00001707 | 14872568831 |
| **30.0** | Approx. 20000.0 |  | 0.00002211 | 6000022499693 |
| **35.0** | Approx. 80000.0 |  | 0.00001702 | 890482293866034 |
| **40.0** | Maybe this year |  | 0.00001733 | 132159290357567120 |
| **44.2** | Maybe before i’m 30 |  | 0.00002223 | 8813218186917735424 |