**COMPENG 3SK3 – Project 1: Numerical Precision**

Professor: Dr. Xiaolin Wu

Khaled Hassan, hassak9, 400203796

Due: November 13, 2022

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is my own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario. Submitted by Khaled Hassan (hassak9, 400203796).

1. Pseudocode

|  |
| --- |
| System:   1. Define the number of terms to include in the alternating harmonic series summation operation, *N*. 2. 4 processors each call the summation algorithm function on N/4 terms. 3. Add the partial sums from processors 1 and 2 together, as well as those from processors 3 and 4 together into next stage, half-sum processors. 4. Add both half sums together to achieve final sum of harmonic series approximation and output approximation, error. |
| Summation Algorithm: An implementation of Kahan Summation. This method was chosen as it accounts for It is implemented as follows:   1. Define a lower bound and upper bound, with the *nth* term representing the term if *n* is odd, and if *n* is even, in the series. 2. Define variables for the partial sum, error, sign and a temporary variable to be used in the summation procedure. 3. A for-loop iterates through *n*, from the lower bound to the upper bound.    1. First, the sign is determined by taking the modulo of n by 2.    2. The current *nth* term is calculated, accounting for the accumulated error.    3. The temporary variable holds the sum of the current partial sum and the current term. The accumulated error is reassigned to the difference of the temporary variable from the partial sum and the current term.    4. The accumulated partial sum is assigned the current value of the temporary variable.   Finally, the final partial sum for that segment of terms is returned. |

This can be summarized in the flowchart below:

Diagram

Description automatically generated

2. Documentation

Since I chose to write my program in C, the entire program is defined in one module, called <*main.c*>. At the header of this file, I include the following two standard libraries: <*stdio.h*> and <*math.h*> and define the function prototype to the partial Kahan sum algorithm implementation function called *calculatePartialSum*\_*kahan*. Next, the total number of terms in the series *N* is defined, along with variables to store the quarter point, the halfway point and three fourths point. Different values of *N* were experimented with. The optimal (minimal) error was achieved without slowing system performance by setting *N = .* Next, 4 processors and 2 half-sum variables are instantiated, each holding a partial sum as follows:

* 1. Processor 1 holds the sum of terms from 1 to – 1.
  2. Processor 2 holds the sum of terms from to – 1.
  3. Processor 3 holds the sum of terms from to – 1.
  4. Processor 4 holds the sum of terms from to N.
  5. Half-sum 1 holds the sum of processor 1 and 2’s sums.
  6. Half-sum 2 holds the sum of processor 3 and 4’s sums.

The partial sums assigned to the processors are found by calling the *calculatePartialSum*\_*kahan()* function, and by passing it the values of the upper and lower bounds for that range of terms as defined above. Finally, a line-breaks and new-lines are used to format the output to the Terminal.

The *calculatePartialSum*\_*kahan()* function is defined as follows:

It requires 2 integer arguments: a *lower\_bound* and an *upper\_bound*. The function instantiates the following variables: *partial\_sum*, *error*, *temp* and *sign*. The function implements Kahan summation for the *n* terms between *upper\_bound* and *upper\_bound* as follows:

1. First, the *sign* variable is assigned a value of +1 or –1, depending on if *n* is even or odd.
2. The current term is calculated by: (*float*) (1 / (*float*) *n \* sign – error*. Explicit type casting is performed by adding the (*float*) keyword in front of *n*, as well as in front of the overall term expression to prevent implicit typecasting to *int*, which would result in an aggregated partial sum of 0.
3. *Temp* is assigned the sum of the accumulated *partial\_sum* with the current *term*.
4. The new accumulated error is found by subtracting the *partial\_sum* and the current *term* from *temp*.
5. Finally, the new *partial\_sum* value assigned is the current value of *temp*. This concludes a single for-loop iteration, and *n* is incremented.
6. Finally, once the for-loop breaks, we return the float variable *partial\_sum*.

Each of the 4 calls to this function calculate the sum of terms from *lower\_limit* to *upper\_limit – 1*. *Upper\_limit* is accounted for in the next function call.

C Code: The code used to implement my parallel simulation algorithm can be found below, as well as in a file named “3SK3\_P1\_hassak9\_main.c” (submitted alongside this report).

|  |
| --- |
| **// Khaled Hassan, hassak9, 400203796**  **#include <stdio.h>**  **#include <math.h> // only for log calculation for error comparison**  **float calculatePartialSum\_kahan(int lower\_limit, int upper\_limit);**  **void main()**  **{**  **unsigned int N = 1e9; // number of terms**  **int quarter = N / 4;**  **int halfway = N / 2;**  **int three\_fourths = 3 \* N / 4;**  **float processor\_1 = 0.0;**  **float processor\_2 = 0.0;**  **float processor\_3 = 0.0;**  **float processor\_4 = 0.0;**  **float half\_sum\_1 = 0.0;**  **float half\_sum\_2 = 0.0;**  **processor\_1 = calculatePartialSum\_kahan(1, quarter);             // from 2 to quarter - 1**  **processor\_2 = calculatePartialSum\_kahan(quarter, halfway);       // from quarter to halfway - 1**  **processor\_3 = calculatePartialSum\_kahan(halfway, three\_fourths); // from halfway to three\_fourths - 1**  **processor\_4 = calculatePartialSum\_kahan(three\_fourths, N + 1);   // from three\_fourths to n**  **half\_sum\_1 = processor\_1 + processor\_2;**  **half\_sum\_2 = processor\_3 + processor\_4;**  **float sum = half\_sum\_1 + half\_sum\_2;**  **printf("\n");**  **printf("------------------------------------------------------\n");**  **printf("\n");**  **printf("Ln2 Kahan approximation with %d terms: %.10f\n", N, sum);**  **printf("------------------------------------------------------\n");**  **printf("\n");**  **printf("Ln2 approximation using C built-in log function: %.10f\n", log(2));**  **printf("\n");**  **printf("------------------------------------------------------\n");**  **printf("\n");**  **printf("Error: %.10f\n", (float)fabs(sum - log(2)));**  **}**  **float calculatePartialSum\_kahan(int lower\_limit, int upper\_limit)**  **{**  **float partial\_sum = 0.0;**  **float error = 0.0;**  **float temp = 0.0;**  **int sign;**  **for (int n = lower\_limit; n < upper\_limit; n++)**  **{**  **if (n % 2 == 0)**  **{**  **sign = -1;**  **}**  **else**  **{**  **sign = 1;**  **}**  **float term = (float)((1 / (float)n) \* sign - error);**  **temp = partial\_sum + term;**  **error = (temp - partial\_sum) - term;**  **partial\_sum = temp;**  **}**  **return partial\_sum;**  **}** |

3. Design Decisions

Throughout the development of this system, several design decisions were taken to achieve the highest attainable precision while fulfilling project requirements. A key decision was the value of N. Through trial and error, N was gradually increased until a value of was reached, keeping in mind the trade-off between speed of program execution and approximation precision. Naturally, as N increases, the estimation is more precise as more terms used reduces the error. However, this also leads to a slower program as the computer has to deal with many more calculations. An acceptable trade-off was achieved at the aforementioned value of N. Another decision taken was to use the Kahan summation algorithm to calculate partial sums. This algorithm was chosen due to its simplicity of implementation, and due to the precision it reaches as it factors accumulated error into calculations. Finally, the program was structured according to the pseudocode defined in section 1, as that defines my approach to the instructions outlined in the project specification document. C was chosen to implement this program instead of Matlab due to its speed and ease of use.

4. Questions:

1. Can you achieve, in the IEEE 32-bit floating number system, any high precision you desire by summing up the first N terms of the series and by running the summation program for a sufficiently large N? Explain your answer.

Regardless of how high of an N we specify, the highest degree of precision that can be achieved with the IEEE 32-bit floating number system is to 7 decimal digits. This is due to the fact that this standard uses a finite number of bits, which dictates that a finite number of real values can be represented exactly with no error.

1. With respect to the IEEE 32-bit floating number system, what is the minimum N value such that 1/N becomes too small to be representable?

Chart

Description automatically generated with medium confidence

The lower bound *L* on the positive representable range using the IEEE 32-bit floating number system is defined as the value when each of are at their lowest possible value. Therefore, when ,

The minimum N such that 1/N is too small to be representable can be found by setting . This defines the limit of representable numbers. To get the minimum unrepresentable number, we simply add 1 => .

1. What is the minimum N value such that 1/(N-1)-1/N becomes too small to be representable?

Similar to the above, we set . This simplifies to the maximum representable . The minimum unrepresentable *N* is found to be

1. What is the minimum N value such that 1/N is smaller than machine precision?

In the IEEE 32-bit standard, machine precision , or the maximum relative error, is equal to .

Similar to above, the minimum N such that 1/N is smaller than machine precision is .

1. What is the minimum N value such that 1/(N-1)-1/N is smaller than machine precision.

Similar to the above, the minimum N such that 1/(N-1)-1/N is smaller than machine precision is .

1. For your algorithm to sum the first N terms of the alternating harmonic series, plot the curve of numerical error against N. Explain the behavior of the error curve. To compute the numerical error, the reference (ground truth) is the Matlab double precision value ln2 = Matlab function log(2) = 0.693147180559945....

|  |  |  |
| --- | --- | --- |
| **N** | **Algorithm Approximation** | **Error** |
| 10 | 0.688172 | 0.047512 |
| 100 | 0.692647 | 0.004975 |
| 1,000 | 0.693097 | 0.0005 |
| 10,000 | 0.693142 | 0.00005 |
| 100,000 | 0.69314671 | 0.000005 |
| 1,000,000 | 0.69314706 | 4.749E-07 |
| 10,000,000 | 0.69314706 | 1.173E-07 |
| 100,000,000 | 0.69314718 | 1.9E-09 |
| 1,000,000,000 | 0.69314718 | 1.9E-09 |

Table 1: Numerical error vs N

Figure 1: Numerical Error vs N

The results of running my parallel summation simulation algorithm for a broad range of *N* values can be summarized in Table 1 and Figure 1 above. For each *N*, the error can be defined as the absolute value of the difference between the experimental value of *ln2*, obtained through running the parallel summation algorithm and the built-in C math library version of *ln2*, stored in double floating-point precision. As can be seen in the Table and Figure above, the error decreases as *N* increases in a decaying exponential fashion. This inverse relation between the numerical error and N appears linear for very low values of N, but the slope flattens out as N increases beyond . Beyond that and until about , increasing *N* reduces the error by several orders of magnitude (due to the already small value of the error). For very high values of *N*, in the or range, the value of the error approaches the , indicating that precision is very high. Increasing *N* beyond that yields minimal improvements in precision for this design. As the final precision achieved is close to , this indicates that the bonus component of this project has been achieved.

1. Does your algorithm achieve the highest possible precision? If not, suggest a possible way to improve the precision.

The algorithm achieves the highest possible precision given project constraints. This is achieved by using a sufficiently high N (), and by implementing the Kahan Summation Algorithm to perform the partial summation of the alternating harmonic series terms. However, there are more complex ways to improve the precision of my implementation:

* Using double-precision floating number representations (*double* in C) will result in values being stored in my computer’s memory using 64 bits instead of 32. With double the number of bits, this will allow precision up to 16 decimal places, instead of the 7 allowed with the single-precision floating number representation currently used.
* Using an open-source library to implement summation, such as the Kahan library or AccSumK.
* Manually applying the IEEE representation in code by saving each of the sign, exponent and mantissa to their own variable.
* Representing each term in the series as an array in C and performing the summation manually. This idea is based on a lab done in an earlier course, where we used arrays to represent and perform operations on Huge Integers (integers that go beyond the natural *int* range of numbers in memory).