Project 1: Numerical Precision

Instructor: Dr. Xiaolin Wu

Yasmin Souza – souzay – 400325538 – COMPENG 3SK3 – C01

1. Pseudocode:

1. Divide the series into 4 equal parts with each part assigned to a unique processor.
2. Each processor will calculate the sum of the series for its part using the formula:

$$\sum_{n=1}^{\infty} \left(\frac{(-1)^{n+1}}{n}\right)$$

   where n is the index of the terms in the series. Therefore, the first processor will calculate the first quarter of this series, the second processor will calculate the second quarter, etc.
3. The sum of the first two processors will be added together and stored in processor 1. The sum of processors 3 and 4 will be added and stored in processor 3.
4. Processor 1 will then add and store the sums of processors 1 and 3, achieving the final resulting approximation of ln2 based on n values.

2. Code:

```c
#include <stdio.h>
#include <math.h>


float findSum(int start, int end)
{
    float c = 0; //compensation for lost bits from arithmetic operations
    float tempSum = 0;
    float sum = 0; //holds final sum
    float term = 0;

    for (int n = start; n < end; n++)
    {
        term = (1 / (float)n) * pow(-1.0, n + 1) - c; //taylor series approximation of ln2

        tempSum = sum + term; //add nth term with current sum
        c = tempSum - sum - term; //calculates error to be subtracted from next iteration
        sum = tempSum; //save new sum
    }
    return sum;
}

void main()
{
    unsigned int n = 100000000;

    float proc1 = findSum(1, n / 4); //procesor 1 - calculates 1st quarter of sum
    float proc2 = findSum(n / 4, n / 2); //procesor 2 - calculates 2nd quarter of sum
    float proc3 = findSum(n / 2, 3 * n / 4); //procesor 3 - calculates 3rd quarter of sum
    float proc4 = findSum(3 * n / 4, n + 1);   //procesor 4 - calculates 4th quarter of sum

    proc1+= proc2; //processor 1 adds its sum with the sum of processor 2 to find the 1st half sum
    proc3+= proc4; //processor 3 adds its sum with the sum of processor 4 to find the 2nd half sum

    proc1+=proc3; //processor 1 adds the 2nd half sum to the 1st to find the overall sum approx of ln2

    printf("\n");
    printf("  the calulated value of ln2 with 100000000 terms = %.10f", proc1);
    printf("\n\n");
    printf("  the value of ln2 from the built-in log function = %.10f", log(2));
}
```

3. Output:

```
the calulated value of ln2 with 100000000 terms = 0.6931471825

the value of ln2 from the C built-in log function = 0.6931471806
```

The above code is used to find an approximation of ln2 using the equation $\sum_{n=1}^{\infty}(\frac{(-1)^{n+1}}{n})$. The program works by having each processor variable call the findSum() function for different quarters of the n value. The variable proc1 representing the first processor finds the sum of the first quarter of n, proc2 finds the sum of the second quarter, etc.

The function findSum() takes in 2 integers which become the starting point and endpoint of the sum calculated respectively. A for loop is iterated through from with the variable n growing from the start to end values, finding the sum of the equation above for each given n. A variable c is initialized to represent the value of the rounding error obtained through the arithmetic of floating-point numbers. Each time a term of the sum is calculated, c is subtracted from the ln2 equation. This is done to minimize the loss of significance in the total result, thereby maximizing the precision of the algorithm.

Once the sums of each processor are calculated, the values from proc1 and proc2 are added together and stored in proc1. Likewise, the sum of proc3 and proc4 are added together and stored in proc3. Finally, proc3 and proc 1 are added together and stored in proc1 to give the final approximation of ln2 calculated by the program.

As can be seen from the output, the final value of ln2 calculated with the above code was 0.6931471825, whereas the ln2 approximation from the c built-in function was 0.6931471806. By subtracting these two numbers, the precision of the calculation is found to be 1.9*10^-9.

4. a) You cannot achieve any high precision desired by simply increasing the value of N. This is due to the fact that certain values of N are too large to be properly represented by the IEEE 32-bit floating number system and would thereby result in an overflow which will give a result of infinity of NaN.

   b) Minimum value of N such that $\frac{1}{N}$ becomes too small to be representable:

$$\frac{1}{N} < L$$
$$\frac{1}{N} < 1.1755 * 10^{-38}$$
$$N = 8.507 * 10^{37}$$

   Minimum value of N such that 1/(N-1)-1/N becomes too small to be representable:

$$\frac{1}{N-1} - \frac{1}{N} < L$$
$$\frac{1}{N^2 - N} < 1.1755 * 10^{-38}$$
$$N^2 - N - 8.507 * 10^{37} < 0$$
$$N = \frac{1 \pm \sqrt{1^2 - 4(-8.507 * 10^{37})}}{2}$$
$$N = 9.223 * 10^{18}$$

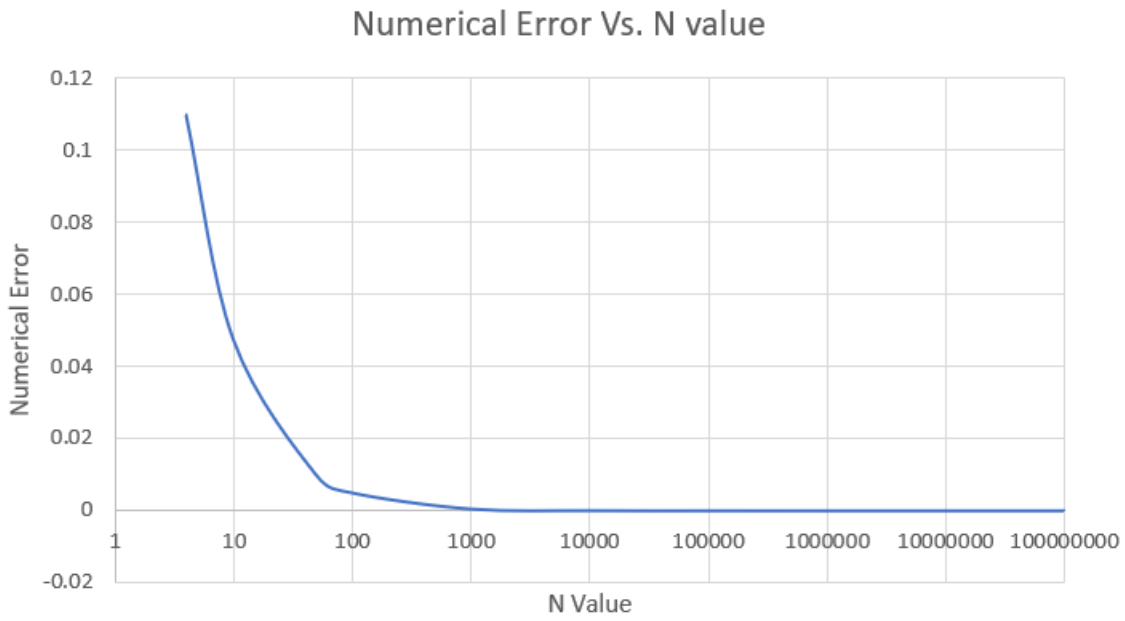
Minimum value of N such that $\frac{1}{N}$ is smaller than machine precision:

$$\frac{1}{N} < E_{mach}$$
$$\frac{1}{N} < 2^{-24}$$
$$N = 2^{24}$$

Minimum N value such that 1/(N-1)-1/N is smaller than machine precision:

$$\frac{1}{N-1} - \frac{1}{N} < 2^{-24}$$
$$\frac{1}{N^2 - N} < 2^{-24}$$
$$N^2 - N - 2^{24} < 0$$
$$N = \frac{1 \pm \sqrt{1^2 - 4(-2^{24})}}{2}$$
$$N = 4096.5$$

c)



The above figure was graphed on excel based on the numerical error calculated for the following N values: 4, 10, 50, 100, 1000,10000, 100000, 1000000, 10000000. As can be seen, As the value of N increases, the numerical error decreases, rapidly approaching 0.

d) My algorithm achieves a precision of 1.9*10^-9, which is not the highest precision possible. To improve, as can be seen in the graph above, higher N values correspond to higher precision. My current N value is 10^8. A value of 10^9 could be used to achieve a higher precision as it is still within the acceptable range of values to be represented. The downside of this change is it would increase the time required to execute the code.