A Simple Calculator

Student ID: 12312710
Name: Yi Xu / Brighton
Course: CS219: Advanced Programming
Date: December 5, 2025

Contents

1 Notice before start

This report is translated by Gemini 3.0 Pro from Chinese to English. I do not have the exact time to recheck, so be carefully reading it.

2 Preface

As we all know, the C++ projects have always been a proving ground for the elites at SUSTech to show off their skills. Browsing through reports from seniors on GitHub, I honestly felt a sense of intimidation. However, a turning point occurred this semester: "The project, which should have been assigned ten days ago, was disrupted by the emergence of DeepSeek." Obviously, the focus of this project has quietly shifted. How to use AI scientifically to assist in coding has become a necessary skill for us to master, and this project inevitably tests our ability in this regard. Therefore, while fulfilling the basic and some advanced requirements, this project incorporates a significant amount of content regarding the scientific use of AI tools.

3 Functional Demonstration

3.1 Runtime Environment

This program is written in C. It was compiled and run on Windows 11 (x64) with an Intel i9-13900HX processor using Powershell. The following code demonstrates compilation and execution (assuming 'calculator.c' is in the current path):

```
PS *path*> gcc -o calculator calculator.c -std=c11
PS *path*> ./calculator
```

3.2 Interactive Input/Output

Direct invocation from the command line:

```
./calculator 1 + 2
1 + 2 = 3
```

Running without arguments enters the continuous input mode:

```
./calculator
Enter expressions (e.g., 2 + 3). Type 'quit' to exit.
> 3 * 4
3 * 4 = 12
> 114514 + 1919810
114514 + 1919810 = 2034324
> help

Expression: (Number) (Operator) (Number) or (function)
Number can be a function like sqrt(2), exp(1)
Supported operators: +, -, x, /
Supported functions: sqrt(x), exp(x)
to set precision for infinite number, use precision(integer), default 10.

> quit
*Program exits*
```

3.3 Support for Various Number Formats

The program supports integers, decimals, and scientific notation. When both operands are in scientific notation, the output maintains scientific notation:

```
> 1.2 - 3.4
1.2 - 3.4 = -2.2
> 1.3e10 - 999999999
1.3e10 - 999999999 = 12000000001
> 1e200 + 1e200
1e200 + 1e200 = 2e200
```

3.4 Lossless Calculation for Finite Precision Numbers

Addition, subtraction, and multiplication support arbitrary precision without loss:

```
> 9999999999999999.0000000000001 + 1.2
9999999999999999.0000000000001 + 1.2 = 10000000000000000.2000000000001
```

```
> 1e100 - 0.00000000001
1e100 - 0.00000000001 = 99999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999.99999999999
> 987654321 * 987654321
987654321 * 987654321 = 975461057789971041
```

3.5 Supported Functions

The calculator features built-in square root and exponential functions, which can be used directly in calculations. The syntax is $sqrt(x)$ and $exp(x)$.

```
> sqrt(10)
sqrt(10) = 3.162277660
> exp(3)
exp(3) = 20.08553692
> exp(3) / exp(2)
exp(3) / exp(2) = 2.718281828
> sqrt(120) + 1e4
sqrt(120) + 1e4 = 10010.95445115
```

3.6 Arbitrary Precision Adjustment for Infinite Numbers

For division, 'sqrt', and 'exp', exact precision is often impossible. Therefore, we retain a specific precision (default is 10 decimal places). The program supports adjusting this via 'precision(x)'. Example:

```
> 1 / 3
1 / 3 = 0.3333333333
> sqrt(2)
sqrt(2) = 1.414213562
> precision(30)
* Precision set to 30.
> 1 / 3
1 / 3 = 0.333333333333333333333333333333
> sqrt(2)
sqrt(2) = 1.41421356237309504880168872421
> precision(150)
* Warning: large precision may cause slow calculation.
Precision set to 150.
> 1 / 7
1 / 7 = 0.142857142857142857142857142857142857142857142857142857
142857142857142857142857142857142857142857142857142857142857142857
571428571428571428571428571428
```

The calculated results above are correct and properly rounded.

3.7 Error Handling

The program can correctly handle invalid user input:

```
> 123q + 456
 --Invalid number, try like '2 + 3'
> 123 # 345
 --Invalid operator, try like '2 + 3'
> 123 +
 --Invalid input, try like '2 + 3'
> 123
 --Invalid input, try like 'sqrt(2)'
> 1 + 2 2
 --Invalid input, too many tokens!
> 12 / 0
 --A number cannot be divided by zero.
> sqrt(-1)
 Cannot sqrt a negative number!
 --Invalid input, try like 'sqrt(2)'
> precision(-1)
ERROR: Precision must be a positive integer!
```

4 Implementation

4.1 High Precision

4.1.1 Analysis

For addition and subtraction of integers within a normal range, using the system's 'int' or 'long' types is sufficient. However, a major goal of this project is to implement our own methods for handling super-large numbers. Initially, I considered using a 'char' array of length N, storing each digit separately, with the first index storing the units digit. For example, "1919810114514" could be stored as $[1, 9, 1, 9, 8, 1, 0, 1, 1, 4, 5, 1, 4]$.

However, I quickly realized a problem. If the input is in the form of "$1234567 \times 1e200$", storing every digit of $1e200$ would result in massive memory waste and subsequent computational inefficiency. If integers, decimals, and large numbers were stored using different logic, the calculation methods would become unnecessarily complex. Is there a more generic way to store high-precision numbers, covering integers, decimals, and scientific notation?

As we know, a decimal can be viewed as an integer plus a fractional part, but we can also view it as an integer $\times 10^n$. For example, 3.1415926 can be written as 31415926×10^{-7}.

Similarly, scientific notation is naturally represented as Integer $\times$ Exponent. Storing a normal integer is simply Integer $\times 10^0$. For negative numbers, we use a separate boolean variable to store the sign. Thus, we find an elegant and generic method to store all numbers without wasting resources.

4.1.2 Data Storage Structure

A encapsulated number contains four pieces of information: a 'char[]' array storing the digits, an integer variable 'digit' for the array length, an integer 'exponent' for the power of 10, and a boolean 'sign' for positivity/negativity. For convenience in reading, the MSB (Most Significant Digit) is placed at the first index of the array.

$$\mathrm{m} = \sum_{i=0}^{d} 10^i \times char[d-i-1] \quad (d \in \mathbb{Z}, char[d-i-1] \in \{0, 1, \ldots, 9\})$$

$$\text{General-Number} = (-1)^b m \times 10^n \quad (n \in \mathbb{Z}, b = 0 \text{ or } 1)$$

4.1.3 Code Implementation

```
typedef struct {
    char *digits;       // store number from [0-9] ,MSB in char[0]
    long long digit;    // store count of digits
    long long exponent; // store exponent, 10^exponent, 0 as integer
    bool sign;          // 0: Positive Number 1: Negative Number
} General_Num;
```

Storage Examples:

```
1:{ {1} , 1 , 0 , 0}
-3.1415926:{ {3,1,4,1,5,9,2,6} , 7 , -7 , 1}
1e200:{ {1} , 1 , 200 , 0}
```

$1 = (-1)^0 \times 1 \times 10^0$, $-3.1415926 = (-1)^1 \times 31415926 \times 10^{-7}$, $10^{200} = (-1)^0 \times 1 \times 10^{200}$

4.2 Addition and Subtraction

4.2.1 Principle

Suppose we have two numbers:

$$a_1 = m_1 \times 10^{n_1}, a_2 = m_2 \times 10^{n_2}$$

· If coincidentally $n_1 = n_2$, the calculation is convenient and fast:

$$a_1 \pm a_2 = (m_1 \pm m_2) \times 10^{n_1}$$

We can simply add or subtract bit by bit from the LSB of the char array.

· If $n_1 \neq n_2$, let's assume $n_1 < n_2$. We can use the following principle:

$$a_1 \pm a_2 = m_1 \times 10^{n_1} \pm m_2 \times 10^{n_2}$$

$$= m_1 \times 10^{n_1} \pm (m_2 \cdot 10^{(n_2 - n_1)}) \times 10^{n_1}$$

$$= \left(m_1 \pm (m_2 \cdot 10^{n_2 - n_1})\right) \times 10^{n_1}$$

In the data operation: enlarge the 'digit' count of a_2 by $n_2 - n_1$ times (shifting left).

For example, $1 \times 10^{-1} + 2 \times 10^2$. We rewrite the term with the larger exponent, 2×10^2, as 2000×10^{-1}. Then, having the same exponent as a_1, we invoke the same-exponent addition logic. The time complexity for the above algorithms is $O(n)$.

4.2.2 Pseudocode Implementation

Since the original code is too long and not elegant enough, I used Copilot to generate pseudocode. Also, because addition and subtraction logic is similar, I only show the full process for addition.

```
FUNCTION sum_of_two_number(num1, num2, result):
    IF result is NULL OR num1.digit is 0 OR num2.digit is 0 OR
       num1.digits is NULL OR num2.digits is NULL THEN
        RETURN failure

    IF num1 is positive && num2 is negative THEN
        RETURN sub_of_two_number(num1, -num2, result)
    ELSE IF num1 is negative && num2 is positive THEN
        RETURN sub_of_two_number(num2, -num1, result)

    IF num1.exponent < num2.exponent THEN
        SWAP num1 and num2

    Create enlarged_num1.digits = num1.digit+num1.exponent-num2.exponent
    enlarged_num1.exponent = num2.exponent
    enlarged_num1.sign = num1.sign

    Enlarge num1.digits to enlarged_num1.digits
    Fill remaining positions with 0

    sum_of_two_integer(enlarged_num1, num2, result)
    result.exponent = num2.exponent

    Deal with trailing zeros
    RETURN success
END FUNCTION
```

```
FUNCTION sum_of_two_integer(num1, num2, result)
    IF result is NULL OR either number has no digits/memory THEN
        RETURN failure

    both_negative = (num1.sign == negative AND num2.sign == negative)

    IF (both numbers are positive) OR (both_negative) THEN
        IF num1.digit < num2.digit THEN
            SWAP(num1, num2)
```

```
        result = {{0,0,...,0},num1.digit+1,0,both_negative}

        FOR i = 0 to num1.digit-1 DO
            result.digits[i+1] = num1.digits[i]

        FOR i = num2.digit-1 downto 0 DO
            position = num1.digit - num2.digit + i + 1
            result.digits[position] += num2.digits[i]

            IF result.digits[position] >= 10 THEN
                result.digits[position] -= 10
                result.digits[position-1]++

        FOR i = num1.digit-1 downto 1 DO
            IF result.digits[i] >= 10 THEN
                result.digits[i] -= 10
                result.digits[i-1]++

        IF result.digits[0] == 0 THEN
            SHIFT all elements left by 1
            result.digit--
            RESIZE result.digits to new size

        RETURN success
    ELSE
        IF num1.sign == negative THEN
            // -a + b = b - a
            num1.sign = positive
            RETURN sub_of_two_integer(num2, num1, result)
        ELSE
            // a + (-b) = a - b
            num2.sign = positive
            RETURN sub_of_two_integer(num1, num2, result)
END FUNCTION
```

4.3 Multiplication Algorithm

4.3.1 Principle

For a_1, a_2, we have:

$$a_1 = m_1 \times 10^{n_1}, a_2 = m_2 \times 10^{n_2}$$

$$a_1 \times a_2 = (m_1 \times m_2) \times 10^{n_1+n_2}$$

Therefore, for the mantissa part m, we only need to calculate the product of two integers. The exponents are simply added.

How to multiply part m? Fast Fourier Transform (FFT) is a good choice, capable of calculating the product of two n-digit numbers in $O(n \log n)$. However, due to time constraints, I did not have enough time to port this method. Thus, I used long multiplication, with a time

complexity of $O(n^2)$.

Assume $m_1 = 456, m_2 = 789$, both having 3 digits. We create an 'int' array of length $m_1.digit + m_2.digit = 6$ (using 'int' to prevent overflow). Then we calculate each item of the new array using the following formula:

$$\text{cal}[k] = \sum_{i+j=k} \text{num1.digits}[\text{num1.digit} - i - 1] \times \text{num2.digits}[\text{num2.digit} - j - 1]$$

$$(\text{where} \quad k \in \{0, 1, \ldots, m_1.digit + m_2.digit - 2\})$$

Visualizing the calculation process:

						4	5	6
					×	7	8	9
			36	45	54			
		32	40	48	0000			
	28	35	42	000	000			
0	28	67	118	93	54			

We get the array $[0, 28, 67, 118, 93, 54]$.

Then, starting from the last item, we carry values greater than 10 to the previous digit (taking the current digit modulo 10, and adding the integer part of the division to the previous digit):

$$Step1 : [0, 28, 67, 118, 98, 4]$$

$$Step2 : [0, 28, 67, 127, 8, 4]$$

$$Step3 : [0, 28, 79, 7, 8, 4]$$

$$Step4 : [0, 35, 9, 7, 8, 4]$$

$$Step5 : [3, 5, 9, 7, 8, 4]$$

Thus, we successfully obtain the product of $m_1 = 456, m_2 = 789$, saved digit by digit.

4.3.2 Pseudocode Implementation

```
FUNCTION multiply_of_two_number(num1, num2, result)
IF result is NULL OR num1.digit is 0 OR num2.digit is 0 OR
   num1.digits is NULL OR num2.digits is NULL THEN
    RETURN failure

result = {{0,0,...,0},num1.digit + num2.digit,0,0}
result.sign = num1.sign XOR num2.sign
result.exponent = num1.exponent + num2.exponent

IF memory allocation failed THEN
    RETURN failure
```

```
    cal = ALLOCATE_MEMORY(result.digit * sizeof(int))
    IF memory allocation failed THEN
        FREE(result.digits)
        RETURN failure

    FOR i = 0 to result.digit - 1 DO
        cal[i] = 0

    FOR i = 0 to num1.digit - 1 DO
        FOR j = 0 to num2.digit - 1 DO
            cal[i + j] += num1.digits[num1.digit - i - 1] *
                          num2.digits[num2.digit - j - 1]

    FOR i = 0 to result.digit - 2 DO
        IF cal[i] >= 10 THEN
            cal[i+1] += cal[i] / 10
            cal[i] = cal[i] MOD 10

    FOR i = 0 to result.digit - 1 DO
        result.digits[i] = cal[result.digit - i - 1]

    IF result.digits[0] = 0 THEN
        SHIFT all digits one position left
        result.digit -= 1
        RESIZE result.digits to new size

    FREE(cal)
    RETURN success
END FUNCTION
```

4.4 Division Algorithm

4.4.1 Principle

Calculating $a_1 \div a_2$ is tricky. We know that division can be understood as $a_1 \times a_2^{-1}$. Since we have already implemented multiplication, the problem transforms into finding a_2^{-1}. To solve for a_2^{-1}, let $a_2^{-1} = x$, so $\frac{1}{x} = a_2$. Let:

$$f(x) = \frac{1}{x} - a_2$$

The problem becomes solving $f(x) = 0$. Using the Newton-Raphson method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = (2 - a_2 x_n)x_n$$

Newton's method requires an initial value x_0. We notice that a_2^{-1} has a readily available, close approximation:

$$a_2^{-1} \approx 10^{-a_2.exponent - a_2.digit}$$

So we set:

$$x_0 = 1 \times 10^{-a_2.exponent - a_2.digit}$$

In code:

$$x_0.digits[0] = 1, x_0.digit = 1, x_0.exponent = (-a_2.exponent - a_2.digit), x_0.sign = 0;$$

The second problem is: how many iterations? We can observe the convergence speed for 3^{-1}:

```
0.17
0.2533
0.31411733
0.3322255689810133
0.33332965190775252026451231566933
0.33333333329267465041213880999538087124790039306156862 05...
0.33333333333333333332837394784267462768029897529088892 14...
0.33333333333333333333333333333333333333335468999506148499...
```

Note that the number of accurate digits increases by 2^n. Thus, we need to calculate:

$$log_2(precision) + C \text{ iterations}$$

Here C is a constant. Since the initial approximation x_0 might be very close or up to a factor of 10 away, we need extra iterations. In the worst case (factor of 10), we need $log_2 10 \approx 3.3$ extra iterations. Thus, $C = 4$.

Also, note that in the final 4 iterations, the number of digits reaches $16 \times precision$, causing redundant calculations. Therefore, in the last 4 iterations, we truncate the digits to $precision+1$ after each step to speed up calculation. The extra digit is for rounding. The time complexity is $O(n^2 \log n)$.

4.4.2 Pseudocode Implementation

```
FUNCTION divide_of_two_number(num1, num2, result):
IF result is NULL OR num1.digit is 0 OR num2.digit is 0 OR
   num1.digits is NULL OR num2.digits is NULL THEN
    RETURN failure

sign = (num1.sign XOR num2.sign)
num1.sign,num2.sign = 0
result.exponent = num1.exponent - num2.exponent

FOR i = 0 TO num2.digit - 1 DO
    IF num2.digits[i] != 0 THEN
        BREAK
    IF i == num2.digit - 1 THEN
```

```
            RETURN division_by_zero (-1)
    END FOR

    inverse_num2 = {{1}, 0, -num2.exponent - num2.digit,0}
    number_2 = {{2},0,0,0}

    loops = CEILING(log(precision)/log(2) + 4)

    FOR i = 0 TO loops - 1 DO
        multiply_of_two_number(inverse_num2, num2, &temp) IF Fail THEN
            CLEANUP_AND_RETURN failure

        sub_of_two_number(number_2, temp, &temp2) IF Fail THEN
            CLEANUP_AND_RETURN failure

         multiply_of_two_number(temp2, inverse_num2, &next_solution)IF Fail
            CLEANUP_AND_RETURN failure

        // For final iterations, reduce digits to prevent excessive growth
        IF i > loops - 4 THEN
            new_digit = (next_solution.digit + 1) / 2
            IF new_digit > 0 THEN
                Truncate next_solution to new_digit digits
                next_solution.exponent += next_solution.digit - new_digit
                next_solution.digit = new_digit
            END IF
        END IF

        DESTROY(temp)
        DESTROY(temp2)
        DESTROY(inverse_num2)
        inverse_num2 = next_solution
        next_solution.digits = NULL
    END FOR

    multiply_of_two_number(num1, inverse_num2, result)  IF Fail THEN
        CLEANUP_AND_RETURN failure

    count = result.digit
    precision_install(result) //round off here
    result.exponent += count - result.digit
    result.sign = sign

    DESTROY(inverse_num2)
    DESTROY(number_2)
    RETURN success
END FUNCTION
```

4.5 Square Root Algorithm

4.5.1 Principle

Square roots can also be calculated using Newton's method. Calculating $\sqrt{a}$ directly involves division (which itself uses Newton's method), making it too slow. We optimize by solving for $\frac{1}{\sqrt{a}}$ instead.

$$f(x) = \frac{1}{x^2} - a, \quad x_{n+1} = (1.5 - 0.5 \times a \times x_n) \times x_n$$

Note that $\sqrt{a} \approx 10^{(a.exponent+a.digit)/2}$. So for the initial guess $x_0 \approx 1 \times 10^{-(a.exponent+a.digit)/2}$. However, my storage structure does not support fractional exponents like 3.5. Solution:

$$x_0 = \begin{cases} 1 \times 10^{-(a.exp+a.dig)/2}, & (a.exp + a.dig) \pmod 2 = 0 \\ 3 \times 10^{-(a.exp+a.dig+1)/2}, & (a.exp + a.dig) \pmod 2 = 1 \end{cases}$$

$$\text{Since:} \quad 10^{-0.5} * 3 \approx 1$$

Checking iterations for $\sqrt{0.25}$ inverse:

```
0.149875000
0.22439167880883789062500000000
0.33517520749064639876190935465311517109330452512 94...
0.49805601200616398069052912515824628040871522223 76...
0.73164055922273359761787309817021460984900726732 88...
1.04850513124828344436008392366502769961686088415 25...
1.42867172720294299476762036662127379523210219738 29...
1.77849933937786378803144036147419432542063681110 14...
1.96456151620923232856118450704442414972508879413 23...
1.99906364873750626505276627875265661850229550080 28...
1.99999934253735359747981886456962312039586878559 31...
1.99999999999967580718696316073111797551296799645 73...
1.99999999999999999999999992117426498144999306018 77...
```

Initial convergence is slow, so C must be larger. Testing shows $C = 10$ is sufficient. Thus, we iterate:

$$log_2(precision) + 10 \text{ times}$$

After obtaining the precise value of $\frac{1}{\sqrt{a}}$, we calculate $1 \div (\frac{1}{\sqrt{a}})$ to get $\sqrt{a}$.

4.6 Exponential Calculation

4.6.1 Principle

Using Taylor Series:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

The challenge is determining when to stop. First, estimate the digits of e^{num} using $10^{num.exponent+num.digit-1} * log_{10}e$. Using $log_{10}e \approx 0.43429$, we calculate the minimum tolerable error based on precision. When the current term is less than $10^{e^{num}-precision-1}$, we stop. However, for e^{50} with precision 10, the first term is 50, while the error tolerance is $10^{21-10-1} = 10^{10}$. The first term is smaller than the tolerance, which causes premature exit. Thus, a minimum loop count is enforced.

4.7 Other Functions

Most functions (log, trig, etc.) can be ported using similar methods. Due to time constraints, I focused on these core functions.

5 How I Used AI

5.1 Introduction

I don't think admitting to using AI is shameful. Without Copilot, the difficulty of this project would have doubled. I even suggest the university offer a general course on AI tool usage, as it massively boosts productivity. Not knowing how to use AI efficiently is a handicap.

5.2 Why Copilot is a Game Changer

With GitHub Student Verification, I use Copilot Pro for free. Embedded in VSCode, it references the entire path, making it efficient. I used the Claude 3.7 model.

5.2.1 Auto-completion

Scenario: removing trailing zeros from 'num.digits'. I forgot how to reallocate memory, but Copilot understood the context:

```
WHILE True DO
    IF result.digits[pointer] == 0 THEN
        count = count + 1
        pointer = pointer - 1
    ELSE
        BREAK
    END IF

    IF pointer == -1 THEN
        result = {0}
        RETURN success
    END IF
END WHILE

IF count == 0 THEN
    RETURN success
END IF
```

Copilot immediately suggested:

```
result->digit -= count;
result->digits = realloc(result->digits, result->digit * sizeof(char));
return 1;
```

It missed 'num.exponent += count'. I asked it to check for memory risks:

```
char* temp = realloc(result->digits, result->digit * sizeof(char));
if (temp == NULL) {
  // Keep the original memory and report failure
    return 0; // Memory allocation failed
}
result->digits = temp;
return 1;
```

This whole process took less than 20 seconds.

5.2.2 Improving Code Quality

As a C beginner coming from Java (where memory and bounds are managed), I make mistakes. Copilot helps spot them dynamically:

```
1.Potential leak of memory pointed to by 'number 0p5.digits'[Ln 540, Col 48]
2.A non-void function does not return a value in all control paths [Ln
339,Col 1]
3.The left operand of'==' is a garbage value C/C+ +[Ln 935, Col 34]
```

Copilot found over 30 warnings and helped fix them by explaining the specific conditions causing the issue, something hard to do manually.

5.2.3 LaTeX/Pseudocode Generation

I had less than 30 minutes of LaTeX experience before this report. To create a professional report, I used DeepSeek R1 to reverse-engineer a PDF I liked into a template. I solved specific formatting issues by asking the AI:

```
How to add a divider and info in the header in LaTeX?
How to adjust line spacing?
How to display text in a code block?
How to wrap code in a grey rounded box to simulate a command line?
How to type summation symbols?
...
```

The pseudocode in this report was also generated by Claude 3.7 based on my source code, saving significant time.

5.3 Limitations of AI

5.3.1 Long Context Reasoning

I tried feeding the project requirements to DeepSeek R1-671B to generate the whole code. It failed three times, using 'long long' and 'double' instead of implementing high precision, ignoring the core requirement.

5.3.2 Understanding Complex Structures

I tried to get AI to port FFT for multiplication. However, 'DeepSeek', 'GPT-4o', and 'Claude 3.7' all failed to provide working code that matched my custom data structure. This shows AI's limitation in understanding complex, custom architectures without extensive guidance.

5.4 Scientific Usage of AI

AI is best for replacing simple labor: generating snippets, finding libraries, or writing pseudocode. It lowers the learning curve. However, it cannot yet replace human programmers for complex logic or architecture. Currently, it's best suited for low-intelligence tasks.

6 Limitations

First, efficiency. I didn't use FFT, making high-precision multiplication (and dependent functions) slow. Second, style. Though improved, I still rely too much on 'if-else' chains. Third, memory management. I relied on Copilot for checks, so I haven't fully developed my own memory management skills.

7 Conclusion

As a beginner in C and LaTeX, this project took 50+ hours. It involved learning syntax, debugging, and writing this report with AI. Overall, the project is a success, implementing high-precision calculation, functions, and precision adjustment. I thank Monad for ideas and Maystern for report phrasing references. Unlike previous years, I explicitly discussed AI usage. I will upload the source code to https://github.com/BrightonXX/SUSTech-CPP-Project.

Thank you for reading.

References

References

[1] Yan W.Q. *CS205 Project2: A Better Calculator*. Southern University of Science and Technology, 2022. [Source code]. Available: github.com/YanWQ-monad/CS205_Project2

[2] Maystern *CS205 Project2: A Better Calculator*. Southern University of Science and Technology, 2022. [Source code]. Available: github.com/Maystern/CS205_Project2

[3] GitHub. *Copilot Chat Cookbook*. Documentation, 2025. Available: docs.github.com/copilot-cookbook