A Simple Calculator

Student ID: 12312710
Name: 胥熠/Brighton
Course: CS219: Advanced Programming
Date: 2025 年 3 月 16 日

目录

1 前言 **2**

2 功能展示 **2**
2.1 运行环境 . . . 2
2.2 交互式输入输出 . . . 2
2.3 支持多种数字输入形式 . . . 3
2.4 对有限精度的数字进行无精度损耗的计算 . . . 3
2.5 支持部分函数调用 . . . 4
2.6 对无限精度的数字进行任意精度调整和计算 . . . 4
2.7 错误输入识别 . . . 4

3 功能实现 **5**
3.1 高精度 . . . 5
3.1.1 分析 . . . 5
3.1.2 数据存储方式 . . . 6
3.1.3 代码实现 . . . 6
3.2 加减运算 . . . 6
3.2.1 原理 . . . 6
3.2.2 伪代码实现 . . . 7
3.3 乘法算法 . . . 8
3.3.1 原理 . . . 8
3.3.2 伪代码实现 . . . 9
3.4 除法算法 . . . 10
3.4.1 原理 . . . 10
3.4.2 伪代码实现 . . . 12
3.5 开方算法 . . . 13
3.5.1 原理 . . . 13
3.6 指数计算 . . . 14
3.6.1 原理 . . . 14
3.7 其他函数 . . . 14

4 我是怎么使用 AI 的 **15**
4.1 引言 15
4.2 Copilot 为什么是神 15
4.2.1 自动补全 15
4.2.2 提升代码质量 16
4.2.3 latex/伪代码生成 17
4.3 AI 还做不到的地方 17
4.3.1 超长思考 17
4.3.2 理解复杂结构 18
4.4 如何科学使用 AI 18

5 不足之处 **18**

6 结语 **18**

1 前言

你知道的，于 ++ 的 5 个 Project 一直是南科大大佬们秀技的修罗场。在 GitHub 上看了几位学长的 report，说实话读完之后冥冥中产生了一种畏惧感。但转机恰巧在这个学期出现了，“~~本来十几天前就该布置的 *Project*，硬是被 *DeepSeek* 搞破防了。~~” 显然，这一次 project 的侧重方向悄然的发生了某些改变，我们如何科学的使用 AI 辅助 coding 成为了我们必须掌握的一种技能，而这次 project 会不可避免的考查我们这个部分的能力。因此，本 project 在完成基本要求和部分进阶要求的前提下，融入了大量的科学使用 AI 工具能力的内容。

2 功能展示

2.1 运行环境

本程序是由 C 语言编写。在 Intel i9-13900HX 运行下的 Windows 11(x64) 系统中，使用 Powershell 使用下列代码编译和运行（calculator.c 放置在 path 文件夹内）：

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

2.2 交互式输入输出

直接在命令行中调用：

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

不直接在命令行中传入参数，进入连续输入模式：

```
./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*
```

2.3 支持多种数字输入形式

支持整数，小数，和科学计数法输入。当输入的两个数都为科学计数法形式的时候，输出也为科学计数法形式：

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

2.4 对有限精度的数字进行无精度损耗的计算

加法，减法和乘法可以进行任意精度的无损计算：

```
> 9999999999999999.0000000000001 + 1.2
9999999999999999.0000000000001 + 1.2 = 10000000000000000.2000000000001
> 1e100 - 0.00000000001
1e100 - 0.00000000001 = 99999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999999999.99999999999
> 987654321 * 987654321
987654321 * 987654321 = 975461057789971041
```

2.5 支持部分函数调用

内置了开根，指数函数，并支持直接在计算中使用。调用方式为 $sqrt(x), exp(x)$

```
> 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
```

2.6 对无限精度的数字进行任意精度调整和计算

对于除法，sqrt 和 exp 函数，计算无法获取完整的精度，因此，我们需要保留精度，默认精度为 10 位。同时，程序支持使用 precision(x) 方式调整精度。示例：

```
> 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
571428571428571428571428571428571
```

计算得上述结果均正确且四舍五入。

2.7 错误输入识别

可以正确处理使用者的错误输入

```
> 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!
```

3 功能实现

3.1 高精度

3.1.1 分析

对于正常范围整数的加减法，则直接调用系统 int 或者 long 类型的方法就可以快速解决，但很显然这个 project 的一大目的就是要求我们手搓自己的处理超大数字的方法。我首先想到的是用一个长度为位数的 char 数组，每一位分别存储对应的位数，数组的第一位存储个位数的值。例如 “1919810114514” 可以存储为 $[1, 9, 1, 9, 8, 1, 0, 1, 1, 4, 5, 1, 4]$。

但我很快意识到一个问题，假如输入是形如 “$1234567 \times 1e200$” 的话，如果逐位的保存 1e200 的每一位，将会造成大量的内存浪费以及后续算力的浪费。假如说整数和小数和超级大数用不同逻辑存储的话，计算必将会要多写很多不必要的方法。于是能不能设计一个更加通用的方法储存高精度数字，包括整数，小数和科学计数法呢?

众所周知，小数可以看做整数. 小数部分，但我们也可以转换个思路，可以看做是一个整数 $\times 10^n$ 的计算结果。例如 3.1415926，可以写作 31415926×10^{-7}。

同理，对于一个科学计数法，用整数 * 指数表示的方法更是自然不过。如果是存储正常整数，则可以很自然地存储为整数 $\times 10^0$。对于负数，我们单独拿一个布尔变量存储数字的正负。由此，我们优雅的找到了一个通用的存储所有数字，同时不会造成资源浪费的方法。

3.1.2 数据存储方式

一个封装的数字总共有个四个信息：存储每一位的数组 char[]，一个整数变量存储数组的长度 d，一个整数变量存储 10^n 中的 n，一个布尔类型存储数字的正负性 b。为了读取的方便性，我们把 MSB 放在了数组的第一位。

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

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

3.1.3 代码实现

```
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;
```

存储案例：

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

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

3.2 加减运算

3.2.1 原理

假如说我们有两个数，分别为

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

· 假如说非常凑巧的，$n_1 = n_2$，那么我们的计算就非常的方便和快捷:

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

我们可以直接从 char 数组的 LSB 位开始逐位进行加或者减的操作即可。

· 但假如说不太凑巧，$n_1 \neq n_2$，假定 $n_1 < n_2$，那么我们可以根据这个原理:

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

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

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

在数据中进行操作：放大 a_2 中存储数据的 $digit$，放大倍数为 $n_2 - n_1$ 倍。

例如 $1 \times 10^{-1} + 2 \times 10^2$，我们将指数项更大的 2×10^2 改写成 2000×10^{-1}，然后改写后的数指数项和 a_1 相同，调用指数相同项的加减法即可。以上算法时间复杂度全部在 $O(n)$。

3.2.2 伪代码实现

放原代码过于冗长而且不够优雅，因此我直接让 *Copilot* 帮助我生成了伪代码，后续有介绍。同时，因为加法和减法逻辑相似，我这里只放了加法的全过程。

```
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
```

3.3 乘法算法

3.3.1 原理

对于 a_1, a_2，我们有：

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

$$a_1 \times a_2 = (m_1 \times m_2) \times 10^{n_1+n2}$$

因此对于 m 部分，我们只用需要计算两个整数的乘积。而指数部分，则直接相加即可。

如何相乘 m 部分呢？快速傅里叶变换（FFT）是一个好选择，可以在 $O(nlogn)$ 的时间复杂度内计算完两个位数为 n 的乘积，但是由于时间原因，我真的没有足够的时间进行这种方法的移植了，因此这里采用的是列式计算，时间复杂度为 $O(n^2)$。

假设 $m_1 = 456, m_2 = 789$，两个数字的位数均为 3，我们建立一个长度为 $m_1.digit + m_2.digit = 6$ 的 int 数组 (不用 char 防止溢出)。然后根据下列公式计算新数组的每一项：

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

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

用可见的竖式来表示计算过程：(这里 latex 排版太困难，能理解意思就行)

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

我们获得了相乘后的数组 $[0, 28, 67, 118, 93, 54]$

随后从尾项开始将大于 10 的位数移植到前一位（对当前位进行对 10 的取余，同时前一位加上本位/10 的整数部分）：

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

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

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

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

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

至此，我们成功的获取了 $m_1 = 456, m_2 = 789$ 的乘积，并将他们按位保存，符合数字构造结构。

3.3.2 伪代码实现

```
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
```

3.4 除法算法

3.4.1 原理

计算 $a_1 \div a_2$ 非常棘手。我们知道，除法可以理解成 $a_1 \times a_2^{-1}$。由于前面我们已经对乘法进行了移植，所以我们的问题就转换成了如何求 a_2^{-1}。求解 a_2^{-1}，我们可以假设 $a_2^{-1} = x, \quad \frac{1}{x} = a_2$; 令:

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

则我们的问题转换成了求解 $f(x)=0$，有牛顿迭代公式：

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}=(2-a_2x_n)x_n$$

牛顿迭代法必须要有一个初值 x_0，我们注意到：a_2^{-1} 已经有一个现成的，比较接近的解了：

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

因此，我们直接令

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

代码上可以这样实现：

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

然后接踵而至的第二个问题：要迭代多少次？我们可以观察求解 3^{-1} 收敛速度：

```
0.17
0.2533
0.31411733
0.3322255689810133
0.33332965190775252026451231566933
0.33333333329267465041213880999538087124790039306156862O5...
0.33333333333333333332837394784267462768029897529088892l4...
0.33333333333333333333333333333333333333335468999506148499...
```

注意到，精确位数以 2^n 的速度增加，所以说我们只需要计算

$$log_2(precision)+C次$$

这里的 C 是一个常数，由于我们的 a_2^{-1} 的初始近似解 x_0 可能距离 a_2 非常接近，也有可能相差接近十倍，所以达到对应精度所需要的迭代次数不同。假设极限情况相差十倍，为了达到对应进度则需要多迭代 $log_210\approx 3.3$ 次，因此此处 $C=4$。

同时，注意到最后四次迭代后，数字位数会达到 $16\times precision$ 位，当 precision 很大的时候会有很多冗杂的计算。因此最后四次迭代中：每次迭代后将存储的位数缩减为前 $precision+1$ 位，以为了更快的计算。多预留的一位是为了四舍五入获取更加精确的结果。因此，本除法时间复杂度为 $O(n^2log_n)$，如果乘法使用 FFT 则可以降低为 $O(n(logn)^2)$

3.4.2 伪代码实现

```
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
```

3.5 开方算法

3.5.1 原理

开方运算也可以用牛顿迭代法计算。由于直接用牛顿迭代法计算 $\sqrt{a}$ 会涉及到本身就是用牛顿迭代计算的除法，如果再在其基础上运行新的一轮牛顿迭代法运行时间会变得无法接受。优化后，问题变成原理为求解 $\frac{1}{\sqrt{a}}$，同理有：

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

更好注意的是；$\sqrt{a} \approx (a.exponent + a.digit)/2$，所以说对于初解 $x_0 \approx 1 \times 10^{-(a.exponent+a.digit)/2}$。但是我的存储结构中，指数不支持存储类似 3.5 的值，怎么办？这简单：

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

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

同样的，这次也要观察需要迭代多少次。举例计算 $\sqrt{0.25}$ 的乘法逆元：

```
0.149875000
0.22439167880883789062500000000
0.33517520749064639876190935465311517109330452512 94...
0.49805601200616398069052912515824628040871522223 76...
0.73164055922273359761787309817021460984900726732 88...
1.04850513124828344436008392366502769961686088415 25...
1.42867172720294299476762036662127379523210219738 29...
```

```
1.7784993393778637880314403614741943254206368111014...
1.9645615162092323285611845070444241497250887941323...
1.9990636487375062650527662787526566185022955008028...
1.9999993425373535974798188645696231203958687855931...
1.9999999999996758071869631607311179755129679964573...
1.9999999999999999999999999211742649814499930601877...
```

额，我们可以观察到精确位数以 2^n 的速度增加，但是算法的一开始显著得收敛的有点慢了，所以说这次的 C 需要取的比较大。根据多次尝试，发现 $C = 10$ 能够用最少的迭代次数完成几乎所有情况的收敛到指定位数。因此我们需要迭代：

$$log_2(precision) + 10次$$

运行完牛顿迭代法后，我们获得了 $\frac{1}{\sqrt{a}}$ 的精确值。最后调用除法 $1 \div (\frac{1}{\sqrt{a}})$，就可以快速而且精确的计算出 $\sqrt{a}$。由于伪代码和除法极其相似，原理都是牛顿迭代法，这里就不再放开方的伪代码，详情可以直接查看 *calculator.c* 中的 $sqrtofnumber$ 函数。

3.6 指数计算

3.6.1 原理

利用泰勒公式：

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

移植泰勒公式的过程相对简单，问题是要迭代多少回？我设计的思路是：首先估计出 e^{num} 总共有多少位，然后根据当前 precision，我们可以计算出能够容忍的最小误差。计算 e^{num} 有多少位，可以用：

$$10^{num.exponent+num.digit-1} * log_{10}e$$

为了计算方便，直接取 $log_{10}e = 0.4342944819$。当计算的当前项小于 $10^{e^{num}-precision-1}$ 时候，就结束泰勒公式的计算，因为余项肯定是小于误差范围内的。但是这样遇到了一个问题：例如计算 $e^{50}, precision = 10$ 的时候：第一项是 50，而误差允许是 $10^{21-10-1} = 10^{10}$。因此，第一项就退出了，这显然是不可接受的，因此我们必须人为框定最少循环次数。

由于伪代码和除法较为相似，原理都是不断的累加数字，这里就不再放指数函数的伪代码，详情可以直接查看 *calculator.c* 中的 $expofnumber$ 函数。

3.7 其他函数

如上述，我使用牛顿迭代法和泰勒展开实现了两种函数的移植，对于绝大多数函数如对数、三角、反三角函数，均可用套用现有方法进行移植。但由于时间有限，我没来得及完成对其他函数的移植，望理解。

4 我是怎么使用 AI 的

4.1 引言

我认为承认使用 AI 并不是什么羞耻的事情。事实上，如果没有 copilot 的辅助，我这次 project 的完成难度将会提升不只一倍。在完成这个 project 的过程中，我甚至有想法说提议学校开设一门 AI 工具使用的通识课，因为它太强大了，而且会可见的未来更加强大。不会正确高效的使用 AI，生产力就是显著低一档。

4.2 Copilot 为什么是神

在进行 GitHub 学生认证后，我可以免费的使用 Copilot Pro。因为 Copilot 高度嵌入 VS-Code，而且生成回答的时候会参考整个 path 下的所有文件，极其高效且准确。而使用其他 AI 则有很多复制粘贴源码的过程，而且精确程度肯定不及读了我放在 path 下 *project*1.*pdf* 的 Copilot。本次 Project 我选用的 Copilot 内置模型是最新的 *Claude*3.7。

4.2.1 自动补全

场景：在计算过程中，可能 *num.digits* 数组后端会混入很多 0。由于我们加减乘除的效率和数组长度直接挂钩，为了计算效率，可以消除 *num.digits* 的尾端连续的'0'。同时为了保证数字不变，要对 *num.exponent* 进行加的操作。在最后，我忘记了如何调用挪数组方法，但 Copilot 能够理解我在干什么并自动补齐：

```
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
```

马上 Copilot 补齐了以下：

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

看着一般，而且没有我期望的 $num.exponent+ = count$。我选中这行代码，叫它检查是否有内存风险：

```
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;
```

嗯，我们就极其快速的调用了 Copilot 生成了我忘记的调库代码。由于 Copilot 高度嵌入 VSCode，整个使用过程不会超过 20s，我们只需要自己添上 $num.exponent+ = count$ 即可。

4.2.2 提升代码质量

我是 C 语言初学者，之前依靠 Java 度过了相对温柔的编程环境，即写代码不需要回收内存、变量都有一个默认的初始值和数组边界检查。一下子进入 C 语言的编程，确实很多时候会忘记这些事。但是我能借助 Copilot 的快速响应，动态的检查代码问题。例如：

```
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]
```

嗯，敲完这个 project 的时候，~~虽然已经很小心的回收内存子~~，但是 Copilot 给我扫出来了 30 多个 Warning。在 VSCode 中，直接右键这个 Warning，Copilot 就可以自动帮我修复。VSCode 似乎使用枚举的方式 debug，右键后会详细的给出在什么条件下会出现这个问题，这的确是单靠我个人 debug 很难做到的事情：

```
1.To fix the potential memory leak, you need to free the memory allocated to
inverse_num2.digits before returning.
2.To fix the issue, ensure that the function is_valid_sqrt returns a value in
all control paths.
3.The problem is that the variable operator is not initialized before it is
used in the comparison. To fix this, initialize operator to a default value.
-- char operator = ' ';
```

4.2.3 latex/伪代码生成

在写项目报告之前，我可能总共接触 latex 时长不到半个小时，只感觉它是一个长得很像 HTML 的东西。但是想要追求精致的项目文档，我需要学习 latex 并做出一个像模像样的报告。这个时候，AI 终于到了他最厉害的领域——超快速上手。

首先，我选择了一个我比较喜欢的项目文档报告，将其下载并截取了其中关键的几页，喂给 DeepSeek R1 和 OpenAI o3-mini，使用 prompt：

```
试图逆向一下这个pdf的latex:（文件）
```

其中 DeepSeek R1 的尝试最成功，至少模仿了原报告的 8 成相貌，于是我直接将其作为本 report 的最基础模版部分。随后，写 report 的过程中想修改页面格式、插入数学公式和放置文本框等操作，我都不会！但是这种问题可以通过询问 AI 极其快速的解决。所以说我就借助 AI 的辅助，顺利的在几乎不会 latex 语法的背景下完成了这个 report。下面是我的几次经典询问：

```
latex中如何在每个页头添加一个分割线并放一些信息？
如何调整latex行间距
latex如何用一个代码框展示文字
listing中，我希望代码块被灰底圆角矩形所包裹，用来提示这是命令行环境，该怎么写
latex中求和符号怎么打
latex中怎么敲出分类讨论的那个条
latex如何划掉文字
......
```

同时，上述文档的伪代码部份，则是我在 VSCode 内置环境中，直接选定某一些行代码，直接叫 Claude 3.7 帮助我生成伪代码实现。最终效果还不错，我只是人工进行了一些微调，这无疑极大的节省了我的时间。

4.3 AI 还做不到的地方

4.3.1 超长思考

相信大家拿到 project 的第一时间就是直接喂给 AI 工具，虽然不寄予厚望，但是也希望生成一个大概的框架。我也尝试了使用 DeepSeek R1-671B（采用了火山引擎提供的服务，最大支持 15K 输出），采用下列 prompt 进行了三次生成：

```
Now, Suppose that you are a C programmer, you are required to read
the following documents, then try all you best to provide the whole
code with every function. Here' s the Document:
*Reduced Document Context, deleted some unrelated things *
```

但遗憾的是，三次生成中，DeepSeek 都是直接使用 $long\quad long$ 存储整数类型，$(long)double$ 存储小数类型，然后直接调用了系统方法进行计算，并没有满足高精度的需求。

4.3.2 理解复杂结构

很多时候，我们并不需要成为制造轮子的人，而是在一个现有的体系下添砖加瓦。在移植乘法的时候。我想投机取巧的使用 AI 工具帮助我直接移植快速傅里叶变换。由于我的数据封装方式比较不同，所以说要写对 FFT 算法，必须理解我的封装结构。$DeepSeek, GPT-4o, Claude3.7$ 都没有在第一轮回答中都提供了看似正确的答案，但没有任何代码通过了案例测试。由于我时间不足，而且我对傅里叶变换也只是了解基本原理水平，没有对 AI 工具进行更深入的问答 debug 环节，但这依旧说明了 AI 工具的对理解复杂结构的能力不足。

4.4 如何科学使用 AI

目前来看，AI 的最大作用就是能够替代部分简单劳动力，例如帮我生成一段简单的但容易忘记的代码、调库，帮我直接生成伪代码等。也能将几乎所有新入手事务的上手成本几乎降至零。根据 Github Copilot Cookbook 的指导，调用 prompt 的语句是一般都比较简短，例如 "Simplify this code. Avoid using if/else chains but retain all function return values."，~~并不需要让 AI 扮演成写不出好代码就被炒鱿鱼的程序员~~。同时，目前我能接触到的 LLM 能力距离替代人类程序员尚有一段距离，至少说直接让 AI 生成某一个模块的代码，最后耗在 Debug 上的时间不见得会比我自己写短。因此，此时此刻，还是让 AI 做一些低智力附值的工作吧。

5 不足之处

首当其冲的效率方面，我没有用 FFT 算大数乘法，导致了后续一系列依附于乘法的功能速度都比较缓慢。

其次，我的代码风格已经经过多次改善，而且附带了大量注释，但仍然有不小的提升空间，例如我还是习惯于写一串的 if else 条件，还需要更正。

对于内存管理方面，我直接利用 Copilot 进行了内存管理的筛查，但本质上没有锻炼到我个人对内存管理的能力，日后还需自行提升。

6 结语

我是 C 语言和 latex 的初学者，这个看着略显简陋的 Project 和 Report 花费了我至少 50+ 小时，包括前期自行学习 C 的语法，中期对 C 语言特性带来的问题进行漫长的调试，以及后期和 AI 对话写 latex 文档等。

总的来说，这次 Project 实现的还算成功，确实实现了高精度的计算。在此之上还加入了精度调整功能和指数函数，也感谢 Monad 为我的工作提供了很多思路，还在项目报告的语言上参考了 Maystern 的一些表述。而且与前几年报告不同的是，我主动的在报告中阐明了我是如何使用 AI 参与工作的。在 DDL 结束后，我也会将 *calculator.c* 和 *report.tex* 源码一并上传到 https://github.com/BrightonXX/SUSTech-CPP-Project。

感谢各位能读到这里。

References

参考文献

[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