COMPLETE OPERATIONS

PITH PEISHU XIE

Abstract. The Operator axioms have produced complete operations with real operators. Numerical computations have been constructed for complete operations. The classic calculator could only execute 7 operator operations: + operator operation(addition), - operator operation(subtraction), × operator operation(multiplication), ÷ operator operation(division), ˆ operator operation(exponentiation), $\sqrt{}$ operator operation(root extraction), log operator operation(logarithm). In this paper, we invent a complete calculator as a software calculator to execute complete operations. The experiments on the complete calculator could directly prove such a corollary: Operator axioms are consistent.

1. Introduction

In [1], we distinguished the limit from the infinite sequence. In [2], we defined the Operator axioms to extend the traditional real number system. In [3], We improved on the Operator axioms. The Operator axioms have produced complete operations with real operators. In [4], we constructed numerical computations for complete operations.

The classic calculator could only execute 7 operator operations: + operator operation(addition), - operator operation(subtraction), × operator operation(multiplication), ÷ operator operation(division), ˆ operator operation(exponentiation), $\sqrt{}$ operator operation(root extraction), log operator operation(logarithm). In this paper, we invent a complete calculator as a software calculator to execute complete operations.

The paper is organized as follows. In Section 2, we design the architecture for the complete calculator. In Section 3, we construct the project for the complete calculator. In Section 4, we design complete-operation instructions for the processor. In Section 5, the experiments on the complete calculator could directly prove such a corollary: Operator axioms are consistent.

2. Architecture

2.1. Hardware.

The hardware of complete calculator contain the following components: CPU, memory, input device, output device, power supply.

The hardware of complete calculator is a laptop with the configurations as Table 1.

2.2. Software.

The software of complete calculator is configured as Table 2.

2.3. Arbitrary-Precision Arithmetic.

The complete operation requires running multiple algorithms. The root-finding algorithm in the complete operation uses the bisection algorithm[4].

The complete operation uses arbitrary-precision arithmetic algorithms to output the arbitrary-precision operation result. The arbitrary-precision arithmetic algorithms are

2020 *Mathematics Subject Classification.* Primary 11-04; Secondary 68-04, 11Y16, 65H05, 03-04.

Key words and phrases. numerical computation, calculator, operator axioms.

Table 1. The Hardware Configuration Of Complete Calculator

Category	Configuration
CPU	Intel i7 x64
RAM	8G
Hard Disk	1T
Input Device	mouse, keyboard
Output Device	LCD
Power Supply	lithium battery

Table 2. The Software Configuration Of Complete Calculator

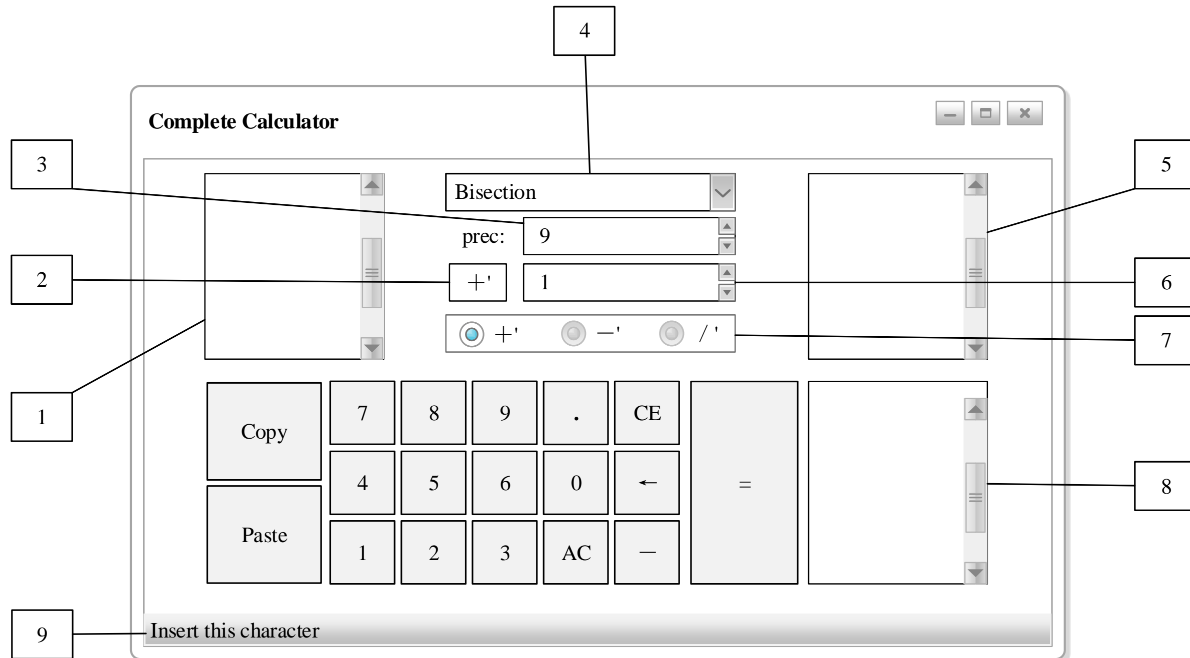
Category	Configuration
Operating System	Windows 10 x64
Programming Language	C++
Compiler	Visual C++ 2015 x64
IDE	Qt Createor
Arbitrary-Precision Arithmetic Libraries	MPIR, MPFR

implemented by the open-source software libraries MPIR and MPFR. MPIR and MPFR compile and output the following files on the Windows platform: gmp.h, mpfr.h, mpir.lib, mpir.dll, mpfr.lib, mpfr.dll.

3. The Project

3.1. **Interactive Interface.** The interactive interface of complete calculator is designed as Figure 1.

Figure 1. Interactive Interface



The notes for interactive interface are listed in Table 3.

The functions of the buttons in interactive interface are listed in Table 4.

The precision n indicates that the error of the complete operation is less than 10^{-n}. The input device can modify the precision n through the input. When the complete calculator executes "input the left operand, select the real operator, input the operator level, input the right operand, click the equal sign" in sequence, it will perform the complete operation and output an approximate result with a given precision. The complete operation requires running multiple algorithms. The root-finding algorithm in the complete operation uses the bisection algorithm. The complete operation uses arbitrary-precision arithmetic algorithms to output the arbitrary-precision operation result.

3.2. **Main File.** The name of the complete calculator project is CompleteCalculator. It includes the following files: CompleteCalculator.pro, gmp.h, mpfr.h, mainwindow.h, operation.h, main.cpp, mainwindow.cpp, operation.cpp.

CompleteCalculator.pro includes the following configuration statements:

LIBS += mpfr.lib LIBS += mpir.lib

HEADERS += gmp.h mpfr.h operation.h

SOURCES += operation.cpp

TABLE 3. The Notes For Interactive Interface

Label	Note
1	QTextEdit. Enter the left operand here.
2	QLabel. The selected real operator.
3	QSpinBox. Number of digits after the radix point (Precision).
4	QComboBox. Select a root-finding method.
5	QTextEdit. Enter the right operand here.
6	QSpinBox. Enter the operator level here.
7	QRadioButton. Select a real operator.
8	QTextEdit. Show the operation result.
9	QStatusBar. Show notes.

TABLE 4. The Functions Of The Buttons

Button	Function
Copy	Copy the selected number
Paste	Paste the selected number
7	Insert the character '7'
4	Insert the character '4'
1	Insert the character '1'
8	Insert the character '8'
5	Insert the character '5'
2	Insert the character '2'
9	Insert the character '9'

Button	Function
6	Insert the character '6'
3	Insert the character '3'
.	Insert the character '.'
0	Insert the character '0'
AC	All clear
CE	Clear entry
←	Backspace
—	Insert the character '—'
=	Perform the operation

operation.h defines the class Operation. It includes the following statements:

```
#ifndef OPERATION_H
#define OPERATION_H

#include <QThread>
#include <QMainWindow>
#include <QTime>
#include <math.h>
#include "mpfr.h"
#include "qmutex.h"

class Operation;

typedef void (Operation::*fp)(mpfr_t, const mpfr_t, const int,
   mpfr_t, mpfr_t);

class Operation : public QThread
{
    Q_OBJECT

public:
    explicit Operation();
    ~Operation();

    // Prefix Function
    void prefix(mpfr_t y, const mpfr_t x, const int level,
       mpfr_t r, mpfr_t l);

    // Suffix Function
    void suffix(mpfr_t y, const mpfr_t x, const int level,
       mpfr_t r, mpfr_t l);

    // Bisection Algorithm
    void bisection(fp f, mpfr_t x1, mpfr_t x2, mpfr_t ERR,
       mpfr_t x, const int level, mpfr_t g, mpfr_t h);

    // Brent Algorithm
    void brent(fp f, mpfr_t lower_bound, mpfr_t upper_bound,
       mpfr_t ERR, mpfr_t x, const int level, mpfr_t g, mpfr_t h
       );

    /*
     output: output of the addote operation +'
     lOperand: left operand
     level: level of the addote operation +'
```

```
     rOperand: right operand
    */
    void addote(mpfr_t output, mpfr_t lOperand, int level,
       mpfr_t rOperand);

    /*
     output: output of the subote operation -'
     lOperand: left operand
     level: level of the subote operation -'
     rOperand: right operand
     ERR: error
    */
    void subote(mpfr_t output, mpfr_t lOperand, int level,
       mpfr_t rOperand, mpfr_t ERR, int Root_Finding);

    /*
     output: output of the logote operation /'
     lOperand: left operand
     level: level of the logote operation /'
     rOperand: right operand
     ERR: error
    */
    void logote(mpfr_t output, mpfr_t lOperand, int level,
       mpfr_t rOperand, mpfr_t ERR, int Root_Finding);

protected:
    void run();

signals:
    void sendMsg(QString msg);

public:
    int Root_Finding;
    mpfr_t ERR;
    mpfr_exp_t et;
    int vec;
    mpfr_t* x;
    mpfr_t* y;
    int level;
    mpfr_t g;
    mpfr_t h;
    QMainWindow* mainPtr;
};

#endif // OPERATION_H
```

operation.cpp implements the methods in the class Operation. It includes the following statements:

```
#include "operation.h"
#include <QFile>
#include <QTextStream>

Operation::Operation()
{
    mpfr_set_emax(mpfr_get_emax_max());
    mpfr_set_emin(mpfr_get_emin_min());
    mpfr_init(ERR);
    mpfr_set_d(ERR, 1e-9, MPFR_RNDN);
    vec = 2;
    x = new mpfr_t[vec];
    y = new mpfr_t[vec];

    for(int i=0; i<vec; i++)
    {
        mpfr_init(x[i]);
        mpfr_init(y[i]);
    }
}

Operation::~Operation()
{
    if(x != NULL)
    {
        for(int i=0; i<vec; i++)
        {
            mpfr_clear(x[i]);
        }
        x=NULL;
    }
    if(y != NULL)
    {
        for(int i=0; i<vec; i++)
        {
            mpfr_clear(y[i]);
        }
        y=NULL;
    }
    this->quit();
    this->wait();
}
```

```
void Operation::run()
{
}

// Prefix Function
void Operation::prefix(mpfr_t y, const mpfr_t x, const int level
   , mpfr_t r, mpfr_t l)
{
    mpfr_t v[3];
    for(int i=0; i<3; i++)
        mpfr_init(v[i]);

    mpfr_set(v[0], x, MPFR_RNDN);
    mpfr_set(v[1], r, MPFR_RNDN);
    mpfr_set(v[2], l, MPFR_RNDN);

    addote(y, v[0], level, v[1]);
    if(mpfr_inf_p(y)!=0)
    {
        mpfr_set_inf(y, MPFR_RNDN);
    }
    else
        mpfr_sub(y, y, v[2], MPFR_RNDN);

    for(int i=0; i<3; i++)
        mpfr_clear(v[i]);
}

// Suffix Function
void Operation::suffix(mpfr_t y, const mpfr_t x, const int level
   , mpfr_t r, mpfr_t l)
{
    mpfr_t v[3];
    for(int i=0; i<3; i++)
        mpfr_init(v[i]);

    mpfr_set(v[0], x, MPFR_RNDN);
    mpfr_set(v[1], r, MPFR_RNDN);
    mpfr_set(v[2], l, MPFR_RNDN);

    addote(y, v[1], level, v[0]);
    if(mpfr_inf_p(y)!=0)
    {
        mpfr_set_inf(y, MPFR_RNDN);
    }
```

```
    else
        mpfr_sub(y, y, v[2], MPFR_RNDN);

    for(int i=0; i<3; i++)
        mpfr_clear(v[i]);
}

/*
Using Bisection, find the root of a function f known to lie
   between x1 and x2. The
root, returned as bisection, will be refined until its accuracy
   is ERR.
f: funtion
x1: lower bound
x2: upper bound
ERR: error
x: root
level: level of the complete operation
g: operand
h: operand
*/
void Operation::bisection(fp f, mpfr_t x1, mpfr_t x2, mpfr_t ERR
   , mpfr_t x, const int level, mpfr_t g, mpfr_t h)
{
    mpfr_t v[4];
    for(int i=0; i<4; i++)
        mpfr_init(v[i]);

    mpfr_set(v[0], x1, MPFR_RNDN);
    mpfr_set(v[1], x2, MPFR_RNDN);
    mpfr_set(v[2], g, MPFR_RNDN);
    mpfr_set(v[3], h, MPFR_RNDN);

    mpfr_t a, b, c, d;
    mpfr_init(a);
    mpfr_init(b);
    mpfr_init(c);
    mpfr_init(d);
    mpfr_set(a, v[0], MPFR_RNDN);
    mpfr_set(b, v[1], MPFR_RNDN);
    mpfr_set(c, v[1], MPFR_RNDN);

    mpfr_t fa, fb, fc;
    mpfr_init(fa);
    mpfr_init(fb);
```

```
    mpfr_init(fc);
    (this->*f)(fa, a, level, v[2], v[3]);
    (this->*f)(fb, b, level, v[2], v[3]);

    if((mpfr_cmp_si(fa, 0)>0 && mpfr_cmp_si(fb, 0)>0) || (
       mpfr_cmp_si(fa, 0)<0 && mpfr_cmp_si(fb, 0)<0))
    {
        emit sendMsg("Root must be bracketed.");

        mpfr_clear(a);
        mpfr_clear(b);
        mpfr_clear(c);
        mpfr_clear(d);
        mpfr_clear(fa);
        mpfr_clear(fb);
        mpfr_clear(fc);
        for(int i=0; i<4; i++)
            mpfr_clear(v[i]);

        return;
    }

    if(mpfr_cmp(fa, fb)>0)
    {
        mpfr_swap (a, b);
        mpfr_swap (fa, fb);
    }

    while(true)
    {
        mpfr_sub(d, b, a, MPFR_RNDN);
        if(mpfr_cmp(d, ERR)<0)
        {
            mpfr_set(x, a, MPFR_RNDN);
            break;
        }

        mpfr_sub(c, b, a, MPFR_RNDN);
        mpfr_div_si(c, c, 2, MPFR_RNDN);
        mpfr_add(c, a, c, MPFR_RNDN);
        (this->*f)(fc, c, level, v[2], v[3]);

        if(mpfr_cmp_si(fc, 0)==0)
        {
            mpfr_set(x, c, MPFR_RNDN);
```

```
            mpfr_clear(a);
            mpfr_clear(b);
            mpfr_clear(c);
            mpfr_clear(d);
            mpfr_clear(fa);
            mpfr_clear(fb);
            mpfr_clear(fc);
            for(int i=0; i<4; i++)
                mpfr_clear(v[i]);

            return;
        }
        else if(mpfr_cmp_si(fc, 0)<0)
        {
            mpfr_set(a, c, MPFR_RNDN);
            mpfr_set(fa, fc, MPFR_RNDN);
        }
        else
        {
            mpfr_set(b, c, MPFR_RNDN);
            mpfr_set(fb, fc, MPFR_RNDN);
        }
    }

    mpfr_clear(a);
    mpfr_clear(b);
    mpfr_clear(c);
    mpfr_clear(d);

    mpfr_clear(fa);
    mpfr_clear(fb);
    mpfr_clear(fc);

    for(int i=0; i<4; i++)
        mpfr_clear(v[i]);
}

/*
Using brent, find the root of a function f known to lie between
   x1 and x2. The
root, returned as brent, will be refined until its accuracy is
   ERR.
f: funtion
lower_bound: lower bound
```

```
upper_bound: upper bound
ERR: error
x: root
level: level of the complete operation
g: operand
h: operand
*/
void Operation::brent(fp f, mpfr_t lower_bound, mpfr_t
   upper_bound, mpfr_t ERR, mpfr_t x, const int level, mpfr_t g,
    mpfr_t h)
{
    mpfr_t v[8];
    for(int i=0; i<8; i++)
        mpfr_init(v[i]);

    mpfr_set(v[0], lower_bound, MPFR_RNDN);
    mpfr_set(v[1], upper_bound, MPFR_RNDN);
    mpfr_set(v[2], g, MPFR_RNDN);
    mpfr_set(v[3], h, MPFR_RNDN);

    mpfr_t a, b, c, d, s;
    mpfr_init(a);
    mpfr_init(b);
    mpfr_init(c);
    mpfr_init(d);
    mpfr_init(s);
    mpfr_set(a, v[0], MPFR_RNDN);
    mpfr_set(b, v[1], MPFR_RNDN);

    mpfr_t fa, fb, fc, fs;
    mpfr_init(fa);
    mpfr_init(fb);
    mpfr_init(fc);
    mpfr_init(fs);
    (this->*f)(fa, a, level, v[2], v[3]); // calculated now to
       save function calls
    (this->*f)(fb, b, level, v[2], v[3]); // calculated now to
       save function calls
    mpfr_set_si(fs, 0, MPFR_RNDN); // initialize

    // if magnitude of f(lower_bound) is less than magnitude of
       f(upper_bound)
    if((mpfr_cmp_si(fa, 0)>0 && mpfr_cmp_si(fb, 0)>0) || (
       mpfr_cmp_si(fa, 0)<0 && mpfr_cmp_si(fb, 0)<0))
    {
```

```
        emit sendMsg("Root must be bracketed.");

        mpfr_clear(a);
        mpfr_clear(b);
        mpfr_clear(c);
        mpfr_clear(d);
        mpfr_clear(s);
        mpfr_clear(fa);
        mpfr_clear(fb);
        mpfr_clear(fc);
        mpfr_clear(fs);
        for(int i=0; i<8; i++)
            mpfr_clear(v[i]);

        return;
    }

    if(mpfr_cmp(fa, fb)>0)
    {
        mpfr_swap (a, b);
        mpfr_swap (fa, fb);
    }

    mpfr_set(c, a, MPFR_RNDN); // c now equals the largest
       magnitude of the lower and upper bounds
    mpfr_set(fc, fa, MPFR_RNDN); // precompute function
       evalutation for point c by assigning it the same value as
        fa
    bool mflag = true; // boolean flag used to evaluate if
       statement later on
    mpfr_set_si(s, 0, MPFR_RNDN); // Our Root that will be
       returned
    mpfr_set_si(d, 0, MPFR_RNDN); // Only used if mflag is unset
        (mflag == false)
    bool e[5];

    while(true)
    {
        mpfr_sub(v[6], b, a, MPFR_RNDN);
        mpfr_abs(v[6], v[6], MPFR_RNDN);

        // stop if converged on root or error is less than ERR
        if(mpfr_cmp(v[6], ERR)<0)
        {
            mpfr_set(x, s, MPFR_RNDN);
```

```
            break;
        }

        if(mpfr_cmp(fa, fc)!=0 && mpfr_cmp(fb, fc)!=0)
        {
            // use inverse quadratic interopolation
            mpfr_sub(v[6], fa, fb, MPFR_RNDN);
            mpfr_sub(v[7], fa, fc, MPFR_RNDN);
            mpfr_mul(v[6], v[6], v[7], MPFR_RNDN);
            mpfr_mul(v[7], a, fb, MPFR_RNDN);
            mpfr_mul(v[7], v[7], fc, MPFR_RNDN);
            mpfr_div(s, v[7], v[6], MPFR_RNDN);

            mpfr_sub(v[6], fb, fa, MPFR_RNDN);
            mpfr_sub(v[7], fb, fc, MPFR_RNDN);
            mpfr_mul(v[6], v[6], v[7], MPFR_RNDN);
            mpfr_mul(v[7], b, fa, MPFR_RNDN);
            mpfr_mul(v[7], v[7], fc, MPFR_RNDN);
            mpfr_div(v[7], v[7], v[6], MPFR_RNDN);
            mpfr_add(s, s, v[7], MPFR_RNDN);

            mpfr_sub(v[6], fc, fa, MPFR_RNDN);
            mpfr_sub(v[7], fc, fb, MPFR_RNDN);
            mpfr_mul(v[6], v[6], v[7], MPFR_RNDN);
            mpfr_mul(v[7], c, fa, MPFR_RNDN);
            mpfr_mul(v[7], v[7], fb, MPFR_RNDN);
            mpfr_div(v[7], v[7], v[6], MPFR_RNDN);
            mpfr_add(s, s, v[7], MPFR_RNDN);
        }
        else
        {
            // secant method
            mpfr_sub(v[6], b, a, MPFR_RNDN);
            mpfr_sub(v[7], fb, fa, MPFR_RNDN);
            mpfr_div(v[6], v[6], v[7], MPFR_RNDN);
            mpfr_mul(v[7], v[6], fb, MPFR_RNDN);
            mpfr_sub(s, b, v[7], MPFR_RNDN);
        }

        /*
            (condition 1) s is not between  (3a+b)/4  and b or
            (condition 2) (mflag is true and |sb| >= |bc|/2) or
            (condition 3) (mflag is false and |sb| >= |cd|/2) or
            (condition 4) (mflag is set and |bc| < |TOL|) or
            (condition 5) (mflag is false and |cd| < |TOL|)
```

```
    */
    mpfr_mul_si(v[6], a, 3, MPFR_RNDN);
    mpfr_add(v[6], v[6], b, MPFR_RNDN);
    mpfr_div_si(v[6], v[6], 4, MPFR_RNDN);
    e[0] = (mpfr_cmp(s, v[6])<0 || mpfr_cmp(s, b)>0);

    mpfr_sub(v[6], s, b, MPFR_RNDN);
    mpfr_abs(v[6], v[6], MPFR_RNDN);
    mpfr_sub(v[7], b, c, MPFR_RNDN);
    mpfr_abs(v[7], v[7], MPFR_RNDN);
    mpfr_div_si(v[7], v[7], 2, MPFR_RNDN);
    e[1] = (mflag && mpfr_cmp(v[6], v[7])>=0);

    mpfr_sub(v[6], s, b, MPFR_RNDN);
    mpfr_abs(v[6], v[6], MPFR_RNDN);
    mpfr_sub(v[7], c, d, MPFR_RNDN);
    mpfr_abs(v[7], v[7], MPFR_RNDN);
    mpfr_div_si(v[7], v[7], 2, MPFR_RNDN);
    e[2] = (!mflag && mpfr_cmp(v[6], v[7])>=0);

    mpfr_sub(v[6], b, c, MPFR_RNDN);
    mpfr_abs(v[6], v[6], MPFR_RNDN);
    e[3] = (mflag && mpfr_cmp(v[6], ERR)<0);

    mpfr_sub(v[6], c, d, MPFR_RNDN);
    mpfr_abs(v[6], v[6], MPFR_RNDN);
    e[4] = (!mflag && mpfr_cmp(v[6], ERR)<0);

    if(e[0] || e[1] || e[2] || e[3] || e[4])
    {
        mpfr_add(v[6], a, b, MPFR_RNDN);
        mpfr_div_si(s, v[6], 2, MPFR_RNDN);
        mflag = true;
    }
    else
    {
        mflag = false;
    }

    (this->*f)(fs, s, level, v[2], v[3]); // calculate fs
    mpfr_set(d, c, MPFR_RNDN); // first time d is being used
        (wasnt used on first iteration because mflag was set
       )
    mpfr_set(c, b, MPFR_RNDN);  // set c equal to upper
       bound
```

```
        mpfr_set(fc, fb, MPFR_RNDN); // set f(c) = f(b)

        mpfr_mul(v[6], fa, fs, MPFR_RNDN);
        // fa and fs have opposite signs
        if(mpfr_cmp_si(v[6], 0)<0)
        {
            mpfr_set(b, s, MPFR_RNDN);
            mpfr_set(fb, fs, MPFR_RNDN); // set f(b) = f(s)
        }
        else
        {
            mpfr_set(a, s, MPFR_RNDN);
            mpfr_set(fa, fs, MPFR_RNDN); // set f(a) = f(s)
        }

        if(mpfr_cmp(fa, fb)>0)
        {
            mpfr_swap (a, b);
            mpfr_swap (fa, fb);
        }
    }

    mpfr_clear(a);
    mpfr_clear(b);
    mpfr_clear(c);
    mpfr_clear(d);
    mpfr_clear(s);
    mpfr_clear(fa);
    mpfr_clear(fb);
    mpfr_clear(fc);
    mpfr_clear(fs);
    for(int i=0; i<8; i++)
        mpfr_clear(v[i]);
}

/*
 output: output of the addote operation +'
 lOperand: left operand
 level: level of the addote operation +'
 rOperand: right operand
*/
void Operation::addote(mpfr_t output, mpfr_t lOperand, int level
   , mpfr_t rOperand)
{
    if(mpfr_nan_p(lOperand)!=0 || mpfr_nan_p(rOperand)!=0)
```

```
{
    mpfr_set_nan(output);
    return;
}
if(mpfr_inf_p(lOperand)!=0 || mpfr_inf_p(rOperand)!=0)
{
    mpfr_set_inf(output, MPFR_RNDN);
    return;
}
mpfr_t v[10];
for(int i=0; i<10; i++)
    mpfr_init(v[i]);

mpfr_set(v[0], output, MPFR_RNDN);
mpfr_set(v[1], lOperand, MPFR_RNDN); // v[1]=a
mpfr_set(v[2], rOperand, MPFR_RNDN);

switch(level)
{
    case 0:
        mpfr_set_nan(v[0]);
        break;
    case 1:
        mpfr_add(v[0], v[1], v[2], MPFR_RNDN);
        break;
    case 2:
        mpfr_mul(v[0], v[1], v[2], MPFR_RNDN);
        break;
    case 3:
        if(mpfr_cmp_si(v[1], 0)==0 && mpfr_cmp_si(v[2], 0)
           ==0)
            mpfr_set_nan(v[0]);
        else
            mpfr_pow(v[0], v[1], v[2], MPFR_RNDN);
        break;
    default:
        if(mpfr_cmp_si(v[1], 1)<0)
        {
            mpfr_set_nan(v[0]);
        }
        else if(mpfr_cmp_si(v[1], 1)==0)
        {
            mpfr_set_si(v[0], 1, MPFR_RNDN);
        }
        else
```

```
            {
                if(mpfr_cmp_si(v[2], 0) == 0)
                {
                    mpfr_set_si(v[0], 1, MPFR_RNDN);
                }
                else if(mpfr_cmp_si(v[2], 1) == 0)
                {
                    mpfr_set(v[0], v[1], MPFR_RNDN);
                }
                else if(mpfr_cmp_si(v[2], 0)>0)
                {
                     // v[4]=b
                    mpfr_modf(v[3], v[4], v[2], MPFR_RNDN);

                    // v[5]=k=n-3
                    mpfr_set_si(v[5], level-3, MPFR_RNDN);

                    // b=b^k
                    mpfr_pow(v[4], v[4], v[5], MPFR_RNDN);

                    // v[6]=r=a-1
                    mpfr_sub_si(v[6], v[1], 1, MPFR_RNDN);

                    // v[7]=1
                    mpfr_set_si(v[7], 1, MPFR_RNDN);

                    // v[8]=b^2
                    mpfr_pow_si(v[8], v[4], 2, MPFR_RNDN);
                    mpfr_mul(v[8], v[8], v[6], MPFR_RNDN);
                    mpfr_mul_si(v[8], v[8], 3, MPFR_RNDN);
                    mpfr_add(v[7], v[7], v[8], MPFR_RNDN);

                    // v[9]=b^3
                    mpfr_pow_si(v[9], v[4], 3, MPFR_RNDN);
                    mpfr_mul(v[9], v[9], v[6], MPFR_RNDN);
                    mpfr_mul_si(v[9], v[9], 2, MPFR_RNDN);

                    // f(b)
                    mpfr_sub(v[0], v[7], v[9], MPFR_RNDN);

                    mpfr_t ite;
                    for(mpfr_init_set_si(ite, 0, MPFR_RNDN);
                       mpfr_cmp(ite, v[3]) < 0; mpfr_add_si(ite,
                        ite, 1, MPFR_RNDN))
                    {
```

```
                        addote(v[0], v[1], level-1, v[0]);
                        if(mpfr_nan_p(v[0])!=0 || mpfr_inf_p(v
                            [0])!=0)
                            break;
                    }
                    mpfr_clear(ite);
                }
                else
                {
                    // v[3]=-b
                    mpfr_si_sub(v[3], 0, v[2], MPFR_RNDN);
                    addote(v[4], v[1], level, v[3]);
                    mpfr_si_div(v[0], 1, v[4], MPFR_RNDN);
                }
            }
            break;
    }
    mpfr_set(output, v[0], MPFR_RNDN);
    for(int i=0; i<10; i++)
        mpfr_clear(v[i]);
}

/*
 output: output of the subote operation -'
 lOperand: left operand
 level: level of the subote operation -'
 rOperand: right operand
 ERR: error
*/
void Operation::subote(mpfr_t output, mpfr_t lOperand, int level
    , mpfr_t rOperand, mpfr_t ERR, int Root_Finding)
{
    if(mpfr_nan_p(lOperand)!=0 || mpfr_nan_p(rOperand)!=0)
    {
        mpfr_set_nan(output);
        return;
    }
    if(mpfr_inf_p(lOperand)!=0 || mpfr_inf_p(rOperand)!=0)
    {
        mpfr_set_inf(output, MPFR_RNDN);
        return;
    }

    mpfr_t v[7];
    for(int i=0; i<7; i++)
```

```
        mpfr_init(v[i]);

    mpfr_set(v[0], output, MPFR_RNDN);
    mpfr_set(v[1], lOperand, MPFR_RNDN);
    mpfr_set(v[2], rOperand, MPFR_RNDN);

    switch(level)
    {
        case 0:
            mpfr_set_nan(v[0]);
            break;
        case 1:
            mpfr_sub(v[0], v[1], v[2], MPFR_RNDN);
            break;
        case 2:
            if(mpfr_cmp_si(v[2], 0) == 0)
                 mpfr_set_nan(v[0]);
            else
                mpfr_div(v[0], v[1], v[2], MPFR_RNDN);
            break;
        case 3:
            if(mpfr_cmp_si(v[1], 0)<=0 || mpfr_cmp_si(v[2], 0)
                ==0)
            {
                mpfr_set_nan(v[0]);
            }
            else
            {
                mpfr_ui_div(v[2], 1, v[2], MPFR_RNDN);
                mpfr_pow(v[0], v[1], v[2], MPFR_RNDN);
            }
            break;
        default:
            if(mpfr_cmp_si(v[2], 0)==0)
            {
                mpfr_set_nan(v[0]);
            }
            else if(mpfr_cmp_si(v[1], 1)<0 && mpfr_cmp_si(v[2],
                0)>0)
            {
                mpfr_set_nan(v[0]);
            }
            else if((mpfr_cmp_si(v[1], 0)<=0 && mpfr_cmp_si(v
                [2], 0)<0) || (mpfr_cmp_si(v[1], 1)>0 &&
                mpfr_cmp_si(v[2], 0)<0))
```

```
            {
                mpfr_set_nan(v[0]);
            }
            else
            {
                if(mpfr_cmp_si(v[2], 1) == 0)
                {
                    mpfr_set(v[0], v[1], MPFR_RNDN);
                }
                else if(mpfr_cmp_si(v[2], 0) > 0)
                {
                    mpfr_set_si(v[3], 1, MPFR_RNDN);
                    mpfr_set_si(v[4], 2, MPFR_RNDN);
                    mpfr_set(v[5], ERR, MPFR_RNDN);

                    addote(v[6], v[4], level, v[2]);
                    while(mpfr_cmp(v[6], v[1]) < 0)
                    {
                        mpfr_set(v[3], v[4], MPFR_RNDN);
                        mpfr_mul_si(v[4], v[4], 2, MPFR_RNDN);
                        addote(v[6], v[4], level, v[2]);
                    }
                    if(Root_Finding == 0)
                        bisection(&Operation::prefix, v[3], v
                            [4], v[5], v[0], level, v[2], v[1]);
                    else
                        brent(&Operation::prefix, v[3], v[4], v
                            [5], v[0], level, v[2], v[1]);
                }
                else
                {
                    mpfr_si_div(v[1], 1, v[1], MPFR_RNDN);
                    mpfr_si_sub(v[2], 0, v[2], MPFR_RNDN);
                    subote(v[0], v[1], level, v[2], ERR,
                        Root_Finding);
                }
            }
            break;
    }
    mpfr_set(output, v[0], MPFR_RNDN);
    for(int i=0; i<7; i++)
        mpfr_clear(v[i]);
}

/*
```

```
 output: output of the logote operation /'
 lOperand: left operand
 level: level of the logote operation /'
 rOperand: right operand
 ERR: error
*/
void Operation::logote(mpfr_t output, mpfr_t lOperand, int level
   , mpfr_t rOperand, mpfr_t ERR, int Root_Finding)
{
    if(mpfr_nan_p(lOperand)!=0 || mpfr_nan_p(rOperand)!=0)
    {
        mpfr_set_nan(output);
        return;
    }
    if(mpfr_inf_p(lOperand)!=0 || mpfr_inf_p(rOperand)!=0)
    {
        mpfr_set_inf(output, MPFR_RNDN);
        return;
    }

    mpfr_t v[7];
    for(int i=0; i<7; i++)
        mpfr_init(v[i]);

    mpfr_set(v[0], output, MPFR_RNDN);
    mpfr_set(v[1], lOperand, MPFR_RNDN);
    mpfr_set(v[2], rOperand, MPFR_RNDN);

    switch(level)
    {
        case 0:
            mpfr_set_nan(v[0]);
            break;
        case 1:
            mpfr_sub(v[0], v[1], v[2], MPFR_RNDN);
            break;
        case 2:
            if(mpfr_cmp_si(v[2], 0) == 0)
                 mpfr_set_nan(v[0]);
            else
                mpfr_div(v[0], v[1], v[2], MPFR_RNDN);
            break;
        case 3:
            if(mpfr_cmp_si(v[1], 0)<0 || mpfr_cmp_si(v[1], 0)==0
                ||
```

```
                    mpfr_cmp_si(v[2], 0)<0 || mpfr_cmp_si(v[2],
                       0)==0)
            {
                mpfr_set_nan(v[0]);
            }
            else
            {
                mpfr_log(v[1], v[1], MPFR_RNDN);
                mpfr_log(v[2], v[2], MPFR_RNDN);
                mpfr_div(v[0], v[1], v[2], MPFR_RNDN);
            }
            break;
        default:
            if(mpfr_cmp_si(v[1], 0)<=0 || mpfr_cmp_si(v[2], 1)
               <=0)
            {
                mpfr_set_nan(v[0]);
            }
            else if(mpfr_cmp_si(v[1], 1) == 0)
            {
                mpfr_set_si(v[0], 0, MPFR_RNDN);
            }
            else if(mpfr_cmp(v[1], v[2]) == 0)
            {
                mpfr_set_si(v[0], 1, MPFR_RNDN);
            }
            else
            {
                if(mpfr_cmp_si(v[1], 1)>0)
                {
                    mpfr_set_si(v[3], 0, MPFR_RNDN);
                    mpfr_set_si(v[4], 1, MPFR_RNDN);
                    mpfr_set(v[5], ERR, MPFR_RNDN);

                    addote(v[6], v[2], level, v[4]);
                    while(mpfr_cmp(v[6], v[1]) < 0)
                    {
                        mpfr_set(v[3], v[4], MPFR_RNDN);
                        mpfr_mul_si(v[4], v[4], 2, MPFR_RNDN);
                        addote(v[6], v[2], level, v[4]);
                    }
                    if(Root_Finding == 0)
                        bisection(&Operation::suffix, v[3], v
                           [4], v[5], v[0], level, v[2], v[1]);
                    else
```

```
                        brent(&Operation::suffix, v[3], v[4], v
                            [5], v[0], level, v[2], v[1]);
                }
                else
                {
                    mpfr_set_si(v[3], 0, MPFR_RNDN);
                    mpfr_set_si(v[4], -1, MPFR_RNDN);
                    mpfr_set(v[5], ERR, MPFR_RNDN);

                    addote(v[6], v[2], level, v[4]);
                    while(mpfr_cmp(v[6], v[1]) > 0)
                    {
                        mpfr_set(v[3], v[4], MPFR_RNDN);
                        mpfr_mul_si(v[4], v[4], 2, MPFR_RNDN);
                        addote(v[6], v[2], level, v[4]);
                    }
                    if(Root_Finding == 0)
                        bisection(&Operation::suffix, v[4], v
                            [5], v[6], v[0], level, v[2], v[1]);
                    else
                        brent(&Operation::suffix, v[4], v[5], v
                            [6], v[0], level, v[2], v[1]);
                }
            }
            break;
    }
    mpfr_set(output, v[0], MPFR_RNDN);
    for(int i=0; i<7; i++)
        mpfr_clear(v[i]);
}
```

3.3. **Interaction.** When the complete calculator executes "input the left operand, select the $+'$ operator, input the operator level, input the right operand, click the equal sign" in sequence, it will run the addote function and output an approximate result with a given precision.

When the complete calculator executes "input the left operand, select the $-'$ operator, input the operator level, input the right operand, click the equal sign" in sequence, it will run the subote function and output an approximate result with a given precision.

When the complete calculator executes "input the left operand, select the $/'$ operator, input the operator level, input the right operand, click the equal sign" in sequence, it will run the logote function and output an approximate result with a given precision.

4. Instruction Set For Complete Operations

The complete-operation algorithm can also be implemented using integrated circuits to speed up operations. For example, the integrated circuit can run instructions such

as addote, subote, logote, and so on. The complete-operation algorithm can also be integrated into the processor. The processor can provide a set of complete-operation instructions and these instructions can run complete operations faster.

5. Experiments And Conclusion

The experimental results of the complete calculator are shown in Table 5. Among them, the operator level is located at the subscript of the real operator, and the precision of the complete operation is 9.

Table 5. Experiments

Complete Operation	**Result**
$2+'_4 2.3$	5.003776891
$3+'_4 2.3$	199.863711347
$2+'_5 2.3$	4.000052635
$3+'_5 2.3$	21590520079800.546875000
$2-'_4 3$	1.476684337
$3-'_4 3$	1.635078474
$2-'_5 3$	1.571959510
$3-'_5 3$	1.731080394
$2/'_4 5$	0.326351821
$3/'_4 5$	0.500000000
$2/'_5 5$	0.571272109
$3/'_5 5$	0.707106780

These experiments on the complete calculator could directly prove such a corollary as follows:

Corollary 5.1. *Operator axioms are consistent.*

References

[1] P. Xie, A logical calculus to intuitively and logically denote number systems, Progress in Applied Mathematics, Vol.1, No.2, (2011), 43-70. arXiv preprint arXiv:0805.3266, 2008.
[2] P. Xie, Number systems based on logical calculus, International Mathematical Forum, Vol.8, No.34, (2013), 1663-1689. viXra preprint viXra:1301.0021, 2013.
[3] P. Xie, Improvement on operator axioms and fundamental operator functions, International Journal of Mathematics And its Applications, Volume 5, Issue 2-A (2017), 125-151. viXra preprint viXra:1412.0001, 2014.
[4] P. P. Xie. Numerical Computations For Operator Axioms. AIMS Mathematics. 2021, 6(4): 4011-4024 [J]. arXiv preprint arXiv:1208.0701, 2012.
[5] C.A. Shaffer, Data Structures and Algorithm Analysis in C++, Third Edition, Dover Publications, 2011.
[6] B. Parhami, Computer Arithmetic : Algorithms and Hardware Designs, Oxford University Press, 2000.
[7] S.C. Chapra, R.P. Canale, Numerical Methods for Engineers, Sixth Edition, McGraw-Hill Companies, Inc., 2009.

Axiom Studio, PO Box #3620, Jiangdongmen Postoffice, Gulou District, Nanjing, 210036, P.R. China
Email address: pith.xie@outlook.com