探索性实验报告

《汇编与接口》Project 1

姓名：黄宇凡

学号：*3220105762*

邮箱：*frankoxer@outlook.com*

指导教师：蔡铭

助教：康致宁、秦嘉俊

日期：*2024* 年 *11* 月 *10* 日

Project 1: 十进制浮点数的测试与原理分析

概述

本次实验以 Python 语言的 decimal 库和 Java 语言的 BigDecimal 类为例，对十进制浮点数的精度和计算性能进行测试并将其与 IEEE 754 标准的二进制浮点数进行比较，而后通过国际标准的 General Decimal Arithmetic Specification（GDA）对十进制浮点数的运算规则进行分析，最后用 C++ 语言实现了一个简单的十进制浮点数计算器。

一、背景说明

在某些行业（例如财务、科学计算等）中，需要对浮点数进行高精度的计算。IEEE 754 标准定义了浮点数的表示方法，包括单精度浮点数和双精度浮点数。

然而，这些浮点数是以二进制存在于计算机中，并以二进制的相关方式来进行运算的。由于二进制浮点数和十进制浮点数之间无法做到完全的一一对应，因此在进行浮点数的运算时，可能会出现精度丢失的情况。

例如，在 Python 语言中，我们直接计算 $0.1 + 0.2$，得到的结果是:

```
>>> 0.1 + 0.2
0.30000000000000004
```

这是因为 0.1 和 0.2 在计算机中以二进制的形式存在，而二进制浮点数无法精确表示 0.1 和 0.2，因此在计算时会出现误差。这些误差可能会在计算中累积，导致最终结果的精度不准确，在某些领域可能会带来严重的后果。

为了解决这些问题，人们希望能在计算机中实现完全符合人类思维的十进制浮点数，**即便会带来一些性能上的损失，但是可以保证计算的精度**。

具体到实现方式上，大多数十进制的浮点数运算都是通过软件来实现，按照 General Decimal Arithmetic Specification（GDA）的规范来进行运算，实际上就是模拟人类进行十进制运算的过程。

Python 语言内置了 decimal 库，可以用来进行十进制浮点数的运算。Java 语言内置了 BigDecimal 类，也可以用来进行十进制浮点数的运算。

二、探索过程

1. Python decimal 库的测试[1]

阅读最新版的 Python 文档关于 decimal 库的说明，可以发现 decimal 库具有以下特征：

- 具有用户可更改的精度（默认为 28 位）

[1]实验环境：Python 3.12.4, 12th Gen Intel® Core™ i5-12500H, 16GB RAM, Windows™ 11

- decimal 模块公开了标准的所有必需部分；在需要时，程序员可以完全控制舍入和信号处理。
- 模块有三个概念：数值、上下文和信号。其中上下文是指定精度、舍入规则等等信息的对象；信号包含计算过程中出现的异常条件组。

精度测试 为了比较 Python 的 decimal 库和 IEEE 754 标准的二进制浮点数的精度，我们可以编写一个简单的测试程序：

```python
from decimal import Decimal, getcontext

def compare_sums():
    # 设置 decimal 模块的精度
    getcontext().prec = 19

    # 使用 Decimal 计算从 0.1 加到 50.0
    decimal_sum = Decimal(0)
    value = Decimal('0.1')
    while value <= Decimal('50.0'):
        decimal_sum += value
        value += Decimal('0.1')

    # 使用二进制浮点数计算从 0.1 加到 50.0
    binary_sum = 0.0
    value = 0.1
    while value <= 50.0:
        binary_sum += value
        value += 0.1

    # 标准结果
    standard_result = 12525.0

    # 输出，精度为 18
    print(f"Decimal  sum: {decimal_sum:.18f}")
    print(f"Binary   sum: {binary_sum:.18f}")
    print(f"Standard sum: {standard_result:.18f}")

if __name__ == "__main__":
    compare_sums()
```

代码很简单，从 0.1 加到 50.0，每次加 0.1，然后分别使用十进制和二进制浮点数计算，最后输出结果并和标准结果，也即 12525.0 进行比较。

输出让人惊讶：

```
Decimal  sum: 12525.000000000000000000
Binary   sum: 12475.000000000063664629
Standard sum: 12525.000000000000000000
```

可以看见，使用十进制浮点数计算的结果和标准结果完全一致，而使用二进制浮点数计算的结果则有误差，并且差了接近 50，误差比例约等于 0.4%。

于是为了进一步的测试，我保持精度在 19 位，然后设置测试为从 0.1 加到 1.5，得到了更加明显的结果：

```
Decimal  sum: 12.000000000000000000
Binary   sum: 10.500000000000000000
Standard sum: 12.000000000000000000
```

可以看到，使用十进制浮点数计算的结果和标准结果完全一致，而使用二进制浮点数计算的结果则有误差，并且差了接近 1.5，误差比例约等于 12.5%。

还有一点很有趣的发现，如果我在 Python 的交互界面中直接让其运算 $0.1 + \cdots + 1.5$，得到的结果是：

```
>>> 0.1 + 0.2 + 0.3 + 0.4 + 0.5 + 0.6 + 0.7 + 0.8 + 0.9 + 1.0 + 1.1 + 1.2 + 1.3 +
↪  1.4 + 1.5
12.0
```

这里的结果竟然是正确的。我的分析是，在代码中，每次迭代自增 0.1，可能会导致精度的丢失，而在交互界面中，每次直接输入 0.1，可能会有更高的精度。

但是无论如何，decimal 的结果始终是正确的，这也说明 Python 的 decimal 模块确实可以用来进行高精度的十进制浮点数运算。

性能测试　接着进行性能测试，代码如下：

```python
# This file is used to test the performance of the decimal module
from decimal import Decimal, getcontext
import time

def decimal_operations(it):
    getcontext().prec = 50  # 设置精度
    a = Decimal('1.23456789')
    b = Decimal('2.34567890')
    for _ in range(it):
        c = a + b
        c = a - b
        c = a * b
        c = a / b

def binary_operations(it):
    a = 1.23456789
    b = 2.34567890
```

```
        for _ in range(it):
            c = a + b
            c = a - b
            c = a * b
            c = a / b

    if __name__ == "__main__":
        iterations = 100000000

        start = time.time()
        decimal_operations(iterations)
        end = time.time()
        print(f"Decimal operations time: {end - start} seconds")

        start = time.time()
        binary_operations(iterations)
        end = time.time()
        print(f" Binary operations time: {end - start} seconds")
```

让两种计算模型分别进行了两个浮点数的加减乘除运算，重复若干次，统计时间。调整运算重复次数，绘制图表如下：

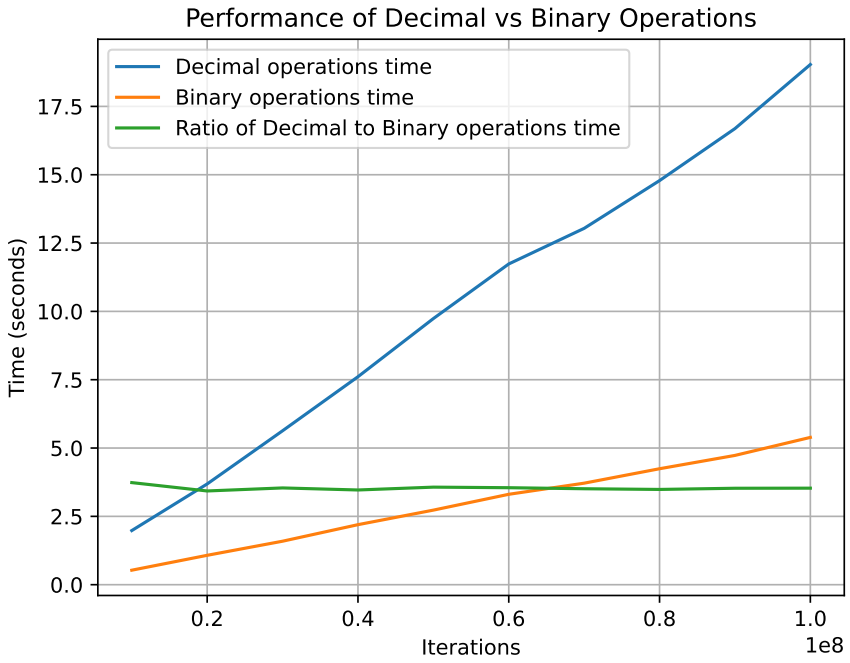

图 1: 十进制浮点数性能测试-对比二进制浮点数

可以看到，随着迭代次数的增加，二者时间均为线性增长，同时十进制浮点数的运算时间明显高于二进制浮点数，这是因为十进制浮点数的运算是通过软件模拟的，而二进制浮点数的运算是通

过硬件实现的，因此十进制浮点数的运算性能会受到一定的影响，在这个测试中，二者的耗时比例约为 3.5 : 1。

此外，我还测试了在同样的迭代次数下，调整 decimal 的精度，得到了如下结果：

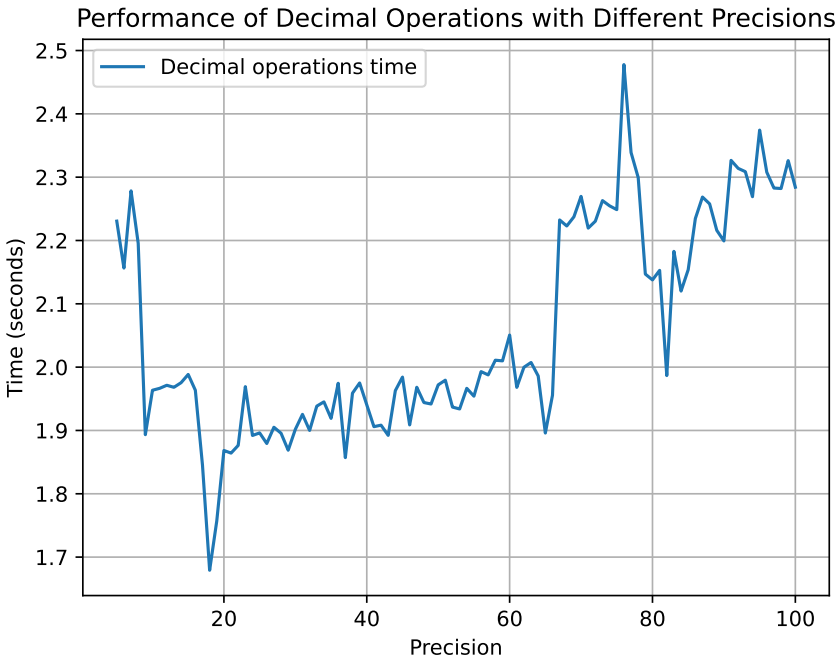

图 2: 十进制浮点数性能测试-对比精度

精度从 5 位到 100 位，时间波动较大，而且在最开始精度比较小的时候耗时较长，从精度约为 20 开始，随着精度增加，耗时逐渐变长。

对于较低精度下（5 到 8）进行计算的性能反常现象，可能是由于较低的精度需要更加频繁的舍入和边界检查。同时 decimal 模块为高精度浮点数而设计，在对精度较高的情况下可能设计了一些计算策略上的优化。

总结：Python 的 decimal 库提供了比较易于使用的十进制浮点数类型，能自由设置精度并实现完全精准的浮点数计算，但是在性能上相比于硬件实现的二进制浮点数会有比较明显的折扣。

2. Java BigDecimal 类的测试[2]

阅读 Java 关于 BigDecimal 类的文档，可以发现 Java 的 BigDecimal 类将浮点数分为两个部分：一个是任意精度的整数 unscaledValue，另一个是指数 scale，从而浮点数被表示为：

$$\text{unscaledValue} \times 10^{-\text{scale}}$$

[2]实验环境：Java 23.0.1, 12th Gen Intel® Core™ i5-12500H, 16GB RAM, Windows™ 11

需要注意的是这里的 scale 是相反的，正数代表小数点在 unscaledValue 最后一位的左边，负数代表在右边。

BigDecimal 类也支持算术运算、比较、舍入等操作。仿照上一节对于 Python decimal 库的测试，我也对 Java 的 BigDecimal 类进行了精度测试与性能测试。

精度测试 按照 BigDecimal 的规范编写测试文件，从 0.1 一直加到 50.0，步长为 0.1：

```java
import java.math.BigDecimal;
import java.math.MathContext;

public class Acc {
    public static void main(String[] args) {
        compareSums();
    }

    public static void compareSums() {
        // 设置 BigDecimal 的精度
        MathContext mc = new MathContext(19);

        // 使用 BigDecimal 计算从 0.1 加到 50.0
        BigDecimal decimalSum = BigDecimal.ZERO;
        BigDecimal value = new BigDecimal("0.1");
        BigDecimal limit = new BigDecimal("50.0");
        BigDecimal increment = new BigDecimal("0.1");
        while (value.compareTo(limit) <= 0) {
            decimalSum = decimalSum.add(value, mc);
            value = value.add(increment, mc);
        }

        // 使用二进制浮点数计算从 0.1 加到 50.0
        double binarySum = 0.0;
        double binaryValue = 0.1;
        while (binaryValue <= 50.0) {
            binarySum += binaryValue;
            binaryValue += 0.1;
        }

        // 标准结果
        double standardResult = 12525.0; // 这个值应该根据精度进行人工计算

        // 输出，精度为 18
        System.out.printf("Decimal  sum: %.18f%n", decimalSum);
        System.out.printf("Binary   sum: %.18f%n", binarySum);
        System.out.printf("Standard sum: %.18f%n", standardResult);
    }
}
```

输出结果为：

```
Decimal  sum: 12525.000000000000000000
Binary   sum: 12475.000000000064000000
Standard sum: 12525.000000000000000000
```

可以看到 BigDecimal 实现的十进制浮点数依然是完全精准的，二进制浮点数产生的结果与 Python 中类似，都是接近于 0.4% 的误差。

然后测试从 0.1 一直加到 1.5，输出结果为：

```
Decimal  sum: 12.000000000000000000
Binary   sum: 10.500000000000000000
Standard sum: 12.000000000000000000
```

这个输出与 Python 完全一致：二进制浮点数仍然产生了较大的误差（12.5%），而 BigDecimal 实现的十进制浮点数依然是完全精准的。

性能测试　仿照前面 Python 的测试，编写测试文件：

```java
import java.math.BigDecimal;
import java.math.MathContext;
import java.io.FileWriter;
import java.io.IOException;

public class Perf {
    public static void main(String[] args) {
        int[] iterationsList = {10000000, 20000000, 30000000, 40000000, 50000000, 60000000, 70000000, 80000000, 90000000, 100000000};
        try (FileWriter writer = new FileWriter("perf-bin.txt")) {
            for (int iterations : iterationsList) {
                long start = System.nanoTime();
                decimalOperations(iterations);
                long end = System.nanoTime();
                double decimalTime = (end - start) / 1e9;

                long begin = System.nanoTime();
                binaryOperations(iterations);
                long stop = System.nanoTime();
                double binaryTime = (stop - begin) / 1e9;

                writer.write(String.format("Iterations: %d, Decimal operations time: %.6f seconds, Binary operations time: %.6f seconds%n", iterations, decimalTime, binaryTime));
                System.out.printf("Iterations: %d, Decimal operations time: %.6f seconds, Binary operations time: %.6f seconds%n", iterations, decimalTime, binaryTime);
```

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void decimalOperations(int it) {
        MathContext mc = new MathContext(50);  // 设置精度
        BigDecimal a = new BigDecimal("1.23456789");
        BigDecimal b = new BigDecimal("2.34567890");
        for (int i = 0; i < it; i++) {
            BigDecimal c = a.add(b, mc);
            c = a.subtract(b, mc);
            c = a.multiply(b, mc);
            c = a.divide(b, mc);
        }
    }

    public static void binaryOperations(int it) {
        double a = 1.23456789;
        double b = 2.34567890;
        for (int i = 0; i < it; i++) {
            double c = a + b;
            c = a - b;
            c = a * b;
            c = a / b;
        }
    }
}
```

性能测试数据写入文件，然后用 Python 绘制图表：

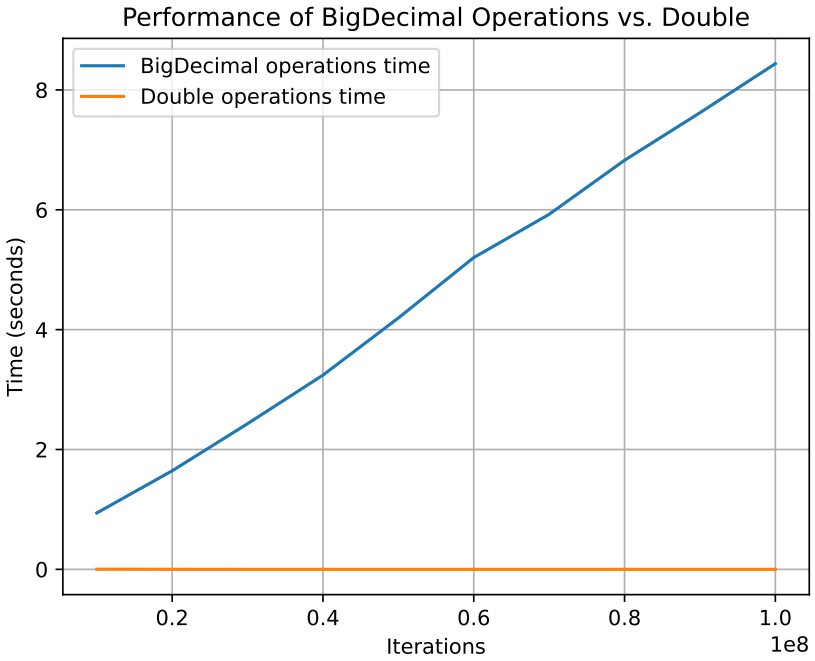

图 3: 十进制浮点数性能测试-对比二进制浮点数（Java）

同样地，十进制浮点数的运算时间明显高于二进制浮点数，且随着迭代次数的增加，时间也是线性增长的。

对于二进制浮点数来说，速度飞快，而且迭代次数增加时，用时反而减少，这可能是因为 JIT 编译器的优化，使得运行时间更短，到最后迭代次数接近 100000000 时，用时仅为 0.000001 秒左右，几乎可以忽略不计，由此可见 Java 的二进制浮点数运算性能非常高。

然后固定迭代次数为 50000000，调整 BigDecimal 的精度，得到了如下结果：

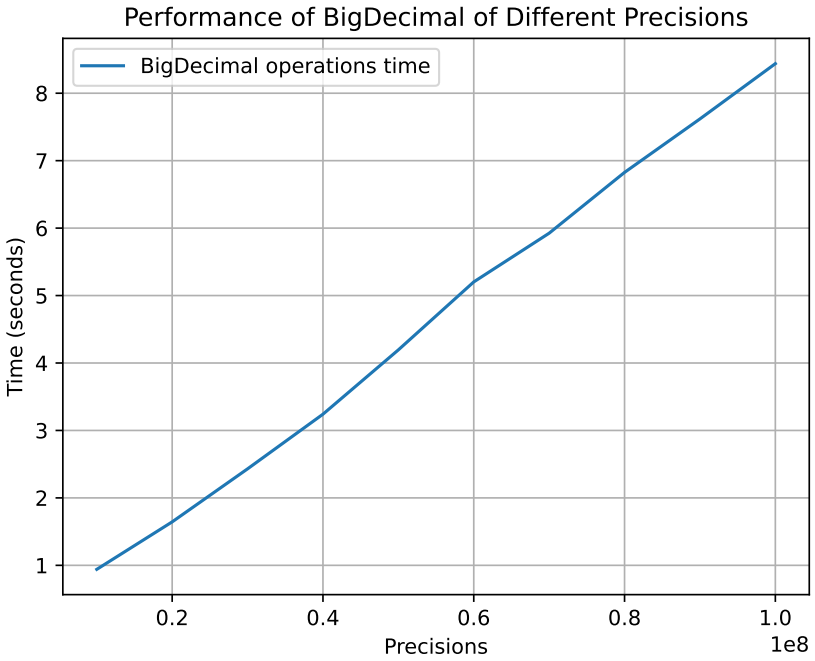

图 4: 十进制浮点数性能测试-对比精度（Java）

不同于 Python 中的结果，在 Java 中 BigDecimal 类的操作时间随着精度上升而增加，这可能是因为 Java 的 BigDecimal 类在高精度下需要更多的计算，导致时间增加。

总结：Java 的 BigDecimal 类同样提供了比较易于使用的十进制浮点数类型，能自由设置精度并实现完全精准的浮点数计算，但是在性能上相比于硬件实现的二进制浮点数会有比较明显的折扣。

3. 原理分析

Python 的 decimal 库和 Java 的 BigDecimal 类都是基于 General Decimal Arithmetic Specification（GDA）的规范实现的。GDA 是一个国际标准，定义了十进制浮点数的运算规则，包括加减乘除、舍入、比较等等。

阅读 GDA 的文档，可以发现 GDA 的运算规则如下：

- 数字分为三部分：符号、系数和指数。系数是一个整数，指数是一个整数，符号是正或负。
- 系数和指数的基数是 10，也就是说，十进制浮点数的系数和指数都是以 10 为基数的。
- 有限数据的值可以表示为：

$$\text{value} = (-1)^{\text{sign}} \times \text{coefficient} \times 10^{\text{exponent}}$$

- 存在三种特殊值：
  1. 无穷大
  2. quiet NaN：“does not cause an Invalid operation condition”的 NaN，也就是说，不会引起异常
  3. signaling NaN：“will usually cause an Invalid operation condition if used in any operation defined in this specification”的 NaN，会引起异常
- 关于 subnormal 数的定义类似于 IEEE 754 标准，都是指数比最小指数小，但是系数不为 0 的数。
- 在格式转换上，定义了两种 number-to-strings 的转换（科学计数法和工程字符串）和一种 string-to-number 的转换，**并强烈建议实现与 BCD 编码的转换**。对于字符串的转换，GDA 定义的规则类似于人们对于十进制数的理解。
- 大部分的计算都按照数学规则进行，但是 special value 上面有一些特殊的规定：
  - 包含 NaN 的运算结果是 NaN
  - 0 区分正负。例如 $-1 \times 0 = -0$
  - 0 作为除数是无穷大，且区分正负
- 加减法对阶的时候，是指数小的对阶到指数大的，然后进行加减法。乘法和除法的时候，是指数相加和相减，然后进行乘法和除法。
- 舍入有一些内置选项：round-down，round-half-up，round-half-even，round-ceiling，round-floor 等等。
- 基本上下文信息包含精度、舍入、flags 和 trap-enablers。

相关的规则还有很多，核心的思想就是通过软件的方式来实现对于十进制浮点数的运算，模拟人类进行十进制运算的过程，从而保证运算的精度。

4. 实现简单的十进制浮点数模块[3]

这里使用 C++ 语言实现一个简单的十进制浮点数模块，实现加减乘除、比较、格式转换等功能，类声明如下：

```
1  class Decimal {
2  private:
3      bool sign = false;
```

[3]实验环境：g++.exe (tdm64-1) 10.3.0, 12th Gen Intel® Core™ i5-12500H, 16GB RAM, Windows™ 11

```
    int coefficient = 0;
    int exponent = 0;
public:
    /* construct from string */
    Decimal(const char* str);

    /* output */
    std::string dec2str() const;
    void print() const;

    /* assign */
    Decimal& operator=(const Decimal& rhs);

    /* compare */
    friend bool operator==(const Decimal& lhs, const Decimal& rhs);
    friend bool operator<(const Decimal& lhs, const Decimal& rhs);
    friend bool operator>(const Decimal& lhs, const Decimal& rhs);
    friend bool operator<=(const Decimal& lhs, const Decimal& rhs);
    friend bool operator>=(const Decimal& lhs, const Decimal& rhs);

    /* abs */
    friend Decimal abs(const Decimal& dec);

    /* arithmic */
    friend Decimal operator+(const Decimal& lhs, const Decimal& rhs);
    friend Decimal operator-(const Decimal& lhs, const Decimal& rhs);
    friend Decimal operator*(const Decimal& lhs, const Decimal& rhs);
    friend Decimal operator/(const Decimal& lhs, const Decimal& rhs);
};
```

我的 Decimal 类包含三个成员变量：符号、系数和指数。构造函数可以从字符串中构造一个 Decimal 对象，输出函数可以将 Decimal 对象转换为字符串，也可以直接输出这个类的成员变量。类重载了赋值运算符、比较运算符、绝对值函数、加减乘除运算符。

对于加减法来说：

1. 首先确定结果的符号，如果同号则直接赋值即可，如果异号，则调用绝对值函数确定最后的符号。
2. 然后对阶，使得两个数的指数相同，这里**被对阶的数的系数需要乘以 10 的差值次方**。
3. 然后进行加法运算，最后得到结果。
4. 减法直接调用加法的结果，然后将第二个数的符号取反即可。

对于乘除法来说：

1. 首先确定结果的符号，然后将两个数的系数相乘，指数相加，最后得到结果。

2. 除法类似，只是将两个数的系数相除，指数相减，最后得到结果。

由于这是一个简单的实现，因此没有考虑精度、舍入等问题，只是实现了基本的运算功能，具体实现可以参考附件中的代码。

接着我仿照之前的方式编写了对于精度和性能的测试，得到了如下结果：

精度测试 从 0.1 加到 1.5，输出结果为：

```
***** Precision comparison *****
Decimal result = 12.0
Double result  = 10.5
```

可以发现，double 实现的二进制浮点数产生了较大的误差（12.5%），而 Decimal 类实现的十进制浮点数依然是完全精准的。

性能测试 分别用 Decimal 类和 double 类型实现不同迭代次数的简单加法运算，得到了如下结果：

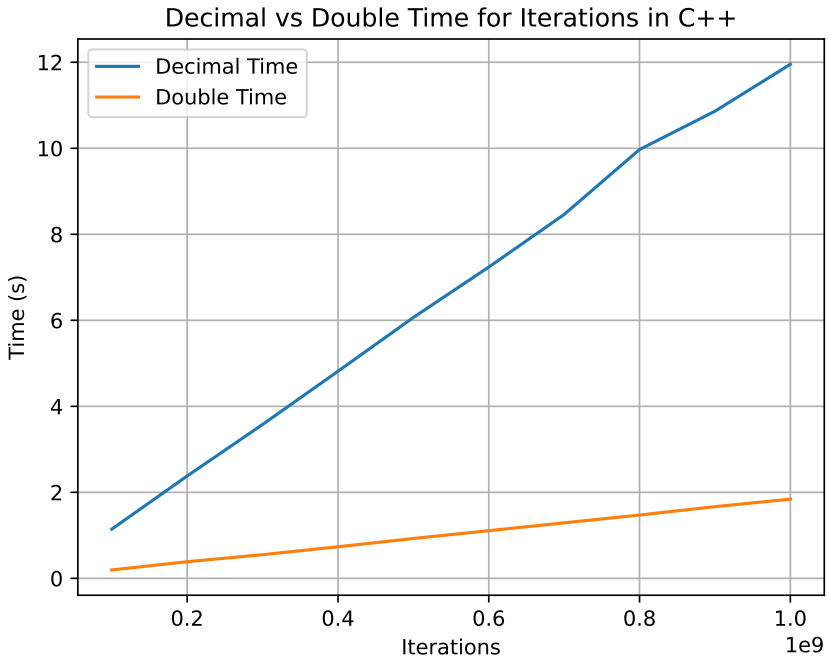

图 5: 十进制浮点数性能测试-对比二进制浮点数（C++）

可以发现，Decimal 类的运算时间明显高于 double 类型，且随着迭代次数的增加，时间也是线性增长的。

三、效果分析

通过对 Python 的 decimal 库、Java 的 BigDecimal 类和自己实现的 Decimal 类进行测试，可以得到如下结论：

- Python 的 decimal 库和 Java 的 BigDecimal 类都是基于 General Decimal Arithmetic Specification（GDA）的规范实现的，能够实现完全精准的十进制浮点数运算。
- Python 的 decimal 库和 Java 的 BigDecimal 类在性能上相比于硬件实现的二进制浮点数会有比较明显的折扣，但是能够保证运算的精度。
- 自己实现的 Decimal 类虽然没有考虑精度、舍入等问题，但是能够实现基本的十进制浮点数运算，且能够保证运算的精度。
- 通过测试可以发现，十进制浮点数的运算时间明显高于二进制浮点数，且随着迭代次数的增加，时间也是线性增长的。

在科学计算和金融领域，对于精度要求较高的场景，可以使用 Python 的 decimal 库或 Java 的 BigDecimal 类来进行十进制浮点数运算，以保证运算的精度。而对于一般的场景，可以使用硬件实现的二进制浮点数来进行运算，以提高运算的性能。

某些硬件也实现了十进制浮点数运算器，但是由于其成本较高，因此在实际应用中并不常见。

四、实验体会

本次实验以二进制浮点数的精度损失为出发点，探讨了十进制浮点数的实现和应用。通过对 Python 的 decimal 库、Java 的 BigDecimal 类和自己实现的 Decimal 类进行测试，我对十进制浮点数的运算规则和性能有了更深入的了解。

在探索的过程中，我强制自己阅读 Python、Java 和 GDA 的文档，虽然全英文的文档阅读起来比较吃力，但是通过这种方式我对十进制浮点数的运算规则和实现有了更深入的了解，也获得的是一手的信息。

在精度测试和性能测试的过程中，我对 Python 的画图库有了比较熟练的掌握，能够快速地绘制出图表，从而更好地展示实验结果。

在考虑设计 Testbench 的时候，我使用了大语言模型的生成能力，生成了一些测试用例，从而更好地测试了十进制浮点数的运算。这些测试用例还远不够覆盖各种应用场景，但是已经能够展示出十进制浮点数的优势和劣势。

在自己实现 Decimal 类的时候，我大量运用了《面向对象程序设计》课程的知识，能够迅速地设计出一个基本类，实现了基本的运算功能。这也让我对 C++ 语言的类设计有了更深入的了解。

总的来说，这次实验让我对十进制浮点数的实现和应用有了更深入的了解，也让我对 Python、Java 和 C++ 语言有了更深入的了解，收获颇丰。

五、经验教训

在实验中，我发现了一些问题和经验教训：

- 在进行精度测试的时候，需要根据精度手动计算标准结果，这样才能更好地对比不同的实现，通过计算得到的数字不一定是正确的，需要仔细检查。
- 在进行性能测试的时候，需要考虑到硬件的影响，不同的硬件可能会有不同的性能表现，同时要做到控制变量，避免其他因素的影响。
- 对于编程语言中常见的类型，在进行复杂操作的时候，编译器等工具会对其进行优化，因此在进行性能测试的时候，需要考虑到这一点，避免出现不符合预期的结果或是遇到无法解释的问题。

六、参考文献

1. Python 3.13.0 Documentation: decimal —Decimal fixed-point and floating-point arithmetic. https://docs.python.org/3.13/library/decimal.html
2. Java SE 23 Documentation: Class BigDecimal. https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/math/BigDecimal.html
3. General Decimal Arithmetic Specification. https://www.speleotrove.com/decimal/decarith.html

七、附件

Decimal 类的实现

```
Decimal::Decimal(const char* str) {
    // check positive or negative
    int i = 0;
    if (str[i] == '-') {
        sign = true;
        i++;
    }
    while (str[i] ≠ '.' && str[i] ≠ '\0') {
        coefficient = coefficient * 10 + (str[i] - '0');
        i++;
    }
    if (str[i] == '.') {
        i++;
    }
    while (str[i] ≠ '\0') {
        coefficient = coefficient * 10 + (str[i] - '0');
```

```
17          exponent--;
18          i++;
19      }
20  }
21
22  std::string Decimal::dec2str() const {
23      std::string str;
24      if (sign) {
25          str += '-';
26      }
27      str += std::to_string(coefficient);
28      if (exponent < 0) {
29          // if extra 0s are needed to be added at the left
30          if (abs(exponent) >= str.size()) {
31              str.insert(0, abs(exponent) - str.size(), '0');
32              str.insert(0, "0.");
33          } else {
34              str.insert(str.size() + exponent, ".");
35          }
36      } else if (exponent > 0) {
37          for (int i = 0; i < exponent; i++) {
38              str += '0';
39          }
40      }
41
42      return str;
43  }
44
45  void Decimal::print() const {
46      std::cout << "sign: " << sign << std::endl;
47      std::cout << "coefficient: " << coefficient << std::endl;
48      std::cout << "exponent: " << exponent << std::endl;
49  }
50
51  Decimal& Decimal::operator=(const Decimal& rhs) {
52      sign = rhs.sign;
53      coefficient = rhs.coefficient;
54      exponent = rhs.exponent;
55      return *this;
56  }
57
58  bool operator==(const Decimal& lhs, const Decimal& rhs) {
59      return lhs.coefficient == rhs.coefficient && lhs.exponent == rhs.exponent &&
        ↪  lhs.sign == rhs.sign;
60  }
61
62  bool operator<(const Decimal& lhs, const Decimal& rhs) {
63      if (lhs.sign && !rhs.sign) {
64          return true;
65      } else if (!lhs.sign && rhs.sign) {
```

```
66          return false;
67      } else if (lhs.sign && rhs.sign) {
68          if (lhs.exponent < rhs.exponent) {
69              return true;
70          } else if (lhs.exponent > rhs.exponent) {
71              return false;
72          } else {
73              return lhs.coefficient > rhs.coefficient;
74          }
75      } else {
76          if (lhs.exponent < rhs.exponent) {
77              return false;
78          } else if (lhs.exponent > rhs.exponent) {
79              return true;
80          } else {
81              return lhs.coefficient < rhs.coefficient;
82          }
83      }
84  }
85
86  bool operator>(const Decimal& lhs, const Decimal& rhs) {
87      return !(lhs < rhs) && !(lhs == rhs);
88  }
89
90  bool operator<=(const Decimal& lhs, const Decimal& rhs) {
91      return lhs < rhs || lhs == rhs;
92  }
93
94  bool operator>=(const Decimal& lhs, const Decimal& rhs) {
95      return lhs > rhs || lhs == rhs;
96  }
97
98  Decimal abs(const Decimal& dec) {
99      Decimal result = dec;
100     result.sign = false;
101     return result;
102 }
103
104 Decimal operator+(const Decimal& lhs, const Decimal& rhs) {
105     Decimal result("0");
106
107     if(lhs.sign ≠ rhs.sign)
108     {
109         if(abs(lhs) == abs(rhs))
110             return result;
111         else if(abs(lhs) < abs(rhs))
112             result.sign = rhs.sign;
113         else
114             result.sign = lhs.sign;
115     }
```

```
    else result.sign = lhs.sign;

    if(lhs.exponent < rhs.exponent) {
        result.exponent = lhs.exponent;
        result.coefficient = lhs.coefficient + rhs.coefficient * pow(10,
        ↪  rhs.exponent - lhs.exponent);
    } else {
        result.exponent = rhs.exponent;
        result.coefficient = rhs.coefficient + lhs.coefficient * pow(10,
        ↪  lhs.exponent - rhs.exponent);
    }

    return result;
}

Decimal operator-(const Decimal& lhs, const Decimal& rhs) {
    Decimal neg("0");
    neg.sign = !rhs.sign;
    neg.coefficient = rhs.coefficient;
    neg.exponent = rhs.exponent;

    return lhs + neg;
}

Decimal operator*(const Decimal& lhs, const Decimal& rhs) {
    Decimal result("0");
    result.sign = lhs.sign ≠ rhs.sign;

    result.coefficient = lhs.coefficient * rhs.coefficient;
    result.exponent = lhs.exponent + rhs.exponent;

    return result;
}

Decimal operator/(const Decimal& lhs, const Decimal& rhs) {
    Decimal result("0");
    result.sign = lhs.sign ≠ rhs.sign;

    result.coefficient = lhs.coefficient / rhs.coefficient;
    result.exponent = lhs.exponent - rhs.exponent;

    return result;
}
```

C++ 精度与性能测试代码

```
void preDecBin() {
    std::cout << "***** Precision comparison *****" << std::endl;
```

```
    Decimal a("0.1");
    Decimal res("0");
    while (a <= Decimal("1.5")) {
        res = res + a;
        a = a + Decimal("0.1");
    }
    std::cout << "Decimal result = " << res.dec2str() << std::endl;

    double b = 0.1;
    double res2 = 0;
    while (b <= 1.5) {
        res2 += b;
        b += 0.1;
    }
    std::cout << "Double result  = " << res2 << std::endl;

    return ;
}

void perfDecBin() {
    std::cout << "***** Performance comparison *****" << std::endl;
    Decimal a("0.1");
    Decimal res("0");
    double b = 0.1;
    double res2 = 0;

    int iterations[] = {100000000, 200000000, 300000000, 400000000, 500000000,
    ↪  600000000, 700000000, 800000000, 900000000, 1000000000};

    for(auto it : iterations) {
        std::cout << "Iterations = " << it;
        std::clock_t start = std::clock();
        for (int i = 0; i < it; i++) {
            res = res + a;
        }
        std::clock_t end = std::clock();
        std::cout << ", Decimal time = " << (double)(end - start) /
        ↪  CLOCKS_PER_SEC << "s";
        start = std::clock();
        for (int i = 0; i < it; i++) {
            res2 += b;
        }
        std::cout << ", Double time  = " << (double)(std::clock() - start) /
        ↪  CLOCKS_PER_SEC << "s" << std::endl;
    }

    return ;
}
```