Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2013-CJ-002

2013-CJ-002

程序数值误差的扰动检测与优化

汤恩义，Earl Barr，苏振东，李宣东

中国科学: 信息科学 2013 年 第 40 卷 第 1 期

论 文

程序数值误差的扰动检测与优化*

汤恩义[①②†], Earl Barr[③④], 苏振东[④], 李宣东[①‡]

① 南京大学计算机软件新技术国家重点实验室, 南京 210093

② 南京大学软件学院, 南京 210093

③ 英国伦敦大学学院计算机科学系, 伦敦 WC1E 6BT

④ 美国加州大学戴维斯分校计算机科学系, 戴维斯 95616

*通信作者. E-mail: eytang@seg.nju.edu.cn, e.barr@ucl.ac.uk, su@cs.ucdavis.edu, lxd@nju.edu.com

国家自然科学基金 (批准号: 91118007 和61021062)、863 国家高技术研究发展计划 (批准号: 2012AA011205)、973 国家重点基础研究发展计划 (批准号: 2009CB320702) 资助项目

摘要 很多用于关键领域的数值计算程序使用浮点数格式作为数据的内部表示, 但由于浮点数在表示上存在误差, 这类程序的正确性很难得到完全的保障。程序的开发人员很容易在密切关注功能开发的情况下直接使用浮点数值, 从而疏忽了所写程序的数值稳定性问题。本文提出了一种自动的检测途径来帮助应用程序的开发人员获得他们所写代码的稳定性信息。它通过扰动底层的数值量和计算步骤, 并统计在扰动下的运算差异, 来评估数值计算代码的稳定性。在这一思想下, 本文给出了两种具体的扰动技术: 数值扰动与算式扰动。这两种扰动技术能够互为补充的检测程序中的数值计算稳定性缺陷: 数值扰动随机动态地改变程序的输入与中间数值的有效数字尾数, 通过模拟误差的引入来观测程序的计算结果是否稳定; 而算式扰动针对用户程序中算术表达式的计算过程, 通过程序变换方法, 将其转换成 (在实数域里) 数学上等价, 但语法上不同的形式。然后以这些数学等价算式在浮点数表示下的执行结果差异来判断数值计算过程的稳定性。本文进一步使用了并行扰动算法和蒙特卡罗方法来提高扰动技术的处理规模。当用户拥有较丰富的硬件资源时, 扰动技术将利用并行算法来提高运行效率; 当硬件资源不足时, 蒙特卡罗方法也能在较短时间内得到一个可以接受的结果。我们对本文所提出的扰动技术做了实现, 并对一系列文献中采用的数值程序和 GNU 科学计算库 (GSL) 做了评估, 评估结果显示, 本文为数值计算稳定性的自动测试提供了一种实用技术。

关键词

数值计算; 浮点数值; 数值稳定性; 扰动方法; 软件测试

1 引言

随着信息技术的发展和计算机处理能力的增强, 原来只有少数超级计算机可以运行的数值计算程序逐渐被广泛应用到其它普通领域 (诸如商业分析、游戏中的物理引擎计算等)。许多这样的应用程序

*本文的初始工作发表于 The 19th International Symposium on Software Testing and Analysis (ISSTA 2010)

不能直接通过调用数值计算库来实现，而要对相应的算法进行重新编码，这意味着一些没有数值计算分析经验的程序员也需要来编写数值计算程序。计算机内部是采用浮点数来表示数值量的，这种浮点数表示只是数学实数的一种近似，没有经验的程序员往往在思维上是以其数学的精确值来考虑他们的算法，这样的思维方式很容易使所写的程序产生数值上的不稳定，即造成数值程序在某一具体执行状态下的输出与理论的准确值产生很大的误差。另外，即使是经过了一定数值分析训练的程序员，在编写关键性地数值计算程序时，也需要用一种自动化的方式来评估已经写好的数值计算代码的稳定性，本文工作的目标正是为这些程序开发人员提供相应的自动化技术。

近年来，数值计算带来的软件可信问题受到了持续关注。数值分析的领域专家们从各自的领域角度来研究稳定的数值算法，在微分方程计算 [1]、图形学设计 [2]以及几何变换 [3]等领域积累了具体的工作，但从软件代码层面上来说数值计算程序的正确性仍然难以得到完全的保障 [4, 5]，业界需要针对软件代码级的自动化数值程序稳定性分析工具。 本文提出了一种新的，可用的自动分析数值计算稳定性的软件框架。该框架能够对代码中的数值计算部分进行自动扫描，对可能引起数值计算问题的程序位置给予警告，并对整个数值计算程序的稳定程度给出整体评估。这项工作既可以帮助程序员更好地理解他们所写的数值计算程序，也可以帮助使用这些数值程序的用户评估他们所得到的计算结果的稳定性。本文框架的实现技术基于如下实验事实：对数值程序的输入以及计算过程进行系统性地扰动能够有效地获得数值计算过程中有用的误差信息。具体来说，我们提出了两种互为补充的核心扰动技术：数值扰动和算式扰动。数值扰动为待分析程序的运行提供了动态的视角，它在待分析程序的运行过程中动态地改变各个数值量的取值，而算式扰动对应地提供了静态的视图，它静态地改变程序中算术表达式的计算形式。虽然本文的软件框架中用到的程序变换概念在一些相关的研究中已被采用 [6, 7, 8, 9]，但这些工作在一般性和有效性上存在很多不足。本文首次提出的结合动态数值扰动和静态算式扰动的方法在一般性和有效性上有了很大改进，从而为程序员自动地分析数值计算程序的稳定性提供了一套实际可用的解决方案。

数值扰动针对计算机内部数值的浮点数表示法。它的目标是把浮点数表示法因为近似所产生的本质误差 (包括其表示误差和计算误差) 清晰地显示出来。当数值的有效数字不能精确地被计算机内浮点数规定的有限位二进制数完整表示时，所产生的误差叫做表示误差。值得注意的是，很多在实数域中很简单的有限小数，在计算机内部的二进制浮点数表示上也会不可避免的存在表示误差。(例如十进制的 0.2 就无法用有限位的二进制浮点数表示)；计算误差是浮点数值在计算过程中产生的误差，举例来说，假设有两个数值 a 和 b 能完整的用有限位二进制浮点数表示，但它们相乘的计算过程 $a \times b$，在浮点数运算上会产生误差，这样的误差就是计算误差。当数值扰动运行时，扰动过程会对程序中每一个读取和操作的底层浮点数值，在其有效数字的最低位随机地改写一小段二进制比特串。这一扰动过程很好地模拟了由浮点数的近似而带来的本质误差，它既会扰动数值 a 和 b 的表示，也会扰动到它们的相乘过程 $a \times b$。经过扰动后的程序，每一次执行都会产生不同的结果。而这些结果之间差异的多少，可以有效反映出误差对数值程序影响的大小。这样的数值扰动过程类似于蒙特卡罗方法 [9]，对这些扰动后执行结果的差异进行统计分析，能够评估原始程序的数值稳定性。

与数值扰动不同，算式扰动针对表达式中算术运算的语法。对某一算术表达式运用数学上的交换律、结合律和分配律会产生大量不同的语法形式。虽然这些不同形式在实数域上的数学语义是相同的，但在浮点数运算中，它们的语义不一定完全相同。每一种语法形式在本质上都可以看作这个数值计算问题在浮点数运算上的一种实现：对于一个良态的数值计算问题来说，某一些实现是比较稳定的，即较小的程序输入扰动会造成很小的程序输出变化；而另外一些实现则不太稳定。我们做算式扰动的目

标是发现这些良态问题在实现上可能出现的不稳定。已有的相关研究中，Martel 采用了一种基于抽象解释的技术 [7, 8]来提取某一数值计算问题中较为稳定的实现方式，然而他的方法由于本身的复杂性，在可伸缩性，效率和精度问题上都存在很大限制，因而实用性不强。我们的方法并不致力于找到最稳定的表达式实现方式，而是分析已有算术表达式实现形式的稳定性，因此这种方法更简单直观且具有实用性。

在使用算式扰动时，在实数域上数学语义相同但语法不同的算术表达式数量往往很大。这意味着当程序中出现复杂算术表达式时，采用直接的算式扰动算法会耗费很长的计算时间。根据不同的场合，我们采用了两种不同的办法来解决这个问题：并行扰动算法和蒙特卡罗方法。随着硬件技术的发展，多核平台已经逐渐普及 [10]，基于 OpenMP [11] 的并行程序设计方法，已经广泛的用于对称多处理器 (Symmetric Multiprocessor, SMP) 系统以及多核系统的并行程序设计。这一程序设计方法的优势在于很容易自适应不同的多核环境，并能够自动在非并行环境下转化成串行程序。并行扰动算法对扰动中互不相干的部分进行分析，从而在并行平台下有效地提高程序的运行效率。用户可以在硬件资源充足的情况下，利用这一方案改善算式扰动处理复杂表达式的能力；而在硬件资源不够的情况下，用户仍然可以使用蒙特卡罗方法来得到可以接受的结果。蒙特卡罗方法通过对语义相同而语法不同的算术表达式进行随机抽样，得到一个接近均匀分布的样本子集，通过统计样本的输出结果，来估计总体输出的统计结果。需要注意的是，当样本子集过小时，得到的统计结果可能不够精确。用户可以根据具体情况同时采用蒙特卡罗方法和并行扰动算法进行算式扰动的优化，以达到更好的效果。

本文对所提出的软件框架及其优化算法做了软件工具实现，并用它评估了相关文献所使用的多个数值计算程序以及开源的数值计算库 GNU Scientific Library (GSL)。评估结果显示：我们的技术能够有效地评估数值计算程序的稳定性，并能定位数值计算程序中可能存在不稳定因素的算术表达式。我们也针对并行优化做了评估，虽然 OpenMP 相对于 MPI 等其他并行程序设计方式的开销稍大，但是评估结果还是显示出了明显的优化效果。因此，以上综合结果表明，本文的方法为理解科学计算程序的数值计算稳定性评估能力提供了一种有效途径。

本文的贡献主要有如下几个方面：

- 本文针对科学计算代码的数值稳定性，提出了一种新的，可用的自动化分析框架，它采用扰动的方法来帮助没有数值计算编写经验的程序员分析他们所写的程序。
- 本文为分析框架中两种互为补充的核心扰动技术——数值扰动和算式扰动给出了具体的算法，并为其可用性提出了优化的具体方案——并行扰动算法和蒙特卡罗方法。
- 本文对所提出的算法和方案做了具体的软件实现，并对相关文献中提到的多个数值计算程序以及 GSL 进行了实例评估，评估结果显示了我们方法的有效性。

2 解决方案实例

我们的扰动框架结合了静态程序分析方法和动态软件测试方法。用户使用框架时，可以在编译期间自由地选择开启或关闭数值扰动或是算式扰动，这样产生的可执行程序里就会具有相应的扰动效果。只要用户配置好测试环境，框架会按用户指定的次数自动地多次运行具有扰动效果的可执行程序来收集数据，最终输出运行结果的有效统计值：极差和变异系数。

由于数值扰动直接更改了浮点数值有效数字的尾数，相当于减少了浮点数值实际可用的有效数

字，这样的结果是使得数值误差被夸大了，因而数值扰动的结果很可能会产生误报，这一点与 Kahan 的结论相吻合 [12]。而算式扰动由于不影响浮点计算的数值而在稳定性检测上显得较为保守，误报率较低，但在执行速度上不如数值扰动。这是因为数值扰动只需要对代码做单次的简单转换。因此，在整个扰动框架中我们采用两步相结合的方式：首先我们使用数值扰动来得到不稳定的候选程序段，然后再用算式扰动对其进行补充检验。假如一个程序的算式扰动时间过长，我们可以用本文中提出的优化算法进行优化。

在数值计算领域，精确而有效的计算统计方差是一个基本问题，下面我们以此为例来说明如何使用数值扰动和算式扰动来帮助程序员诊断程序的数值稳定性并且定位疑似不稳定的位置。图 1(a) 是计算方差的 C 语言原始实现，它缓存了当前的样本总和与平方和，并在输出时直接通过统计公式来输出方差。当我们以 1000 个相同的输入 3.1415926×10^{10} 作为样本时，其理论方差值应该为 0。而令人惊讶的是，当我们采用 gcc 4.2.4 在 Ubuntu 8.04 平台上编译并执行此程序，得到的输出为 -1.27446×10^{7}，这是一个明显的由于数值计算程序实现上的不稳定而产生的计算错误。这个错误在书本里已经被明确指出 [13]，其产生原因是第七行中的 `squaresum` 和 `sum * sum / n` 是很接近的数值，当方差较小时，引起了数值分析中应当避免的大数相消。而问题在于，应当怎样采用我们的技术框架自动地检测出这类代码实现的不稳定。

```
1 double naive(double[] samples, int n) {
2     double sum, squaresum;
3     for (int i; i < n; i++) {
4         sum += samples[i];
5         squaresum += samples[i] * samples[i];
6     }
7     return (squaresum - sum * sum / n) / n;
8 }
```

(a) 原始的算法实现

```
1 void knuth(double[] samples, int n) {
2     double delta, mean=0, M2=0;
3     for (int i = 0; i < n; ++i) {
4         delta = samples[i] - mean;
5         mean += delta / i;
6         M2 += delta * (samples[i] - mean);
7     }
8     return M2 / n;
9 }
```

(b) Kunth 改进后的稳定算法

```
1     (1/n) * sum * (1/n) * (-sum) + squaresum * (1/n)
2     (1/n) * sum * (1/n) * (-sum) + (1/n) * squaresum
3     squaresum * (1/n) + (-sum) * sum * (1/n) * (1/n)
4     squaresum * (1/n) + sum * (1/n) * (1/n) * (-sum)
                          ...
142   (-sum * sum * (1/n)) * (1/n) + squaresum * (1/n)
143     (1/n) * (sum * (1/n) * (-sum) + squaresum)
144       (squaresum - sum * sum * (1/n)) * (1/n)
```

(c) 扰动产生的部分数学等价算术表达式

图 1: 方差计算——扰动算法应用示例

数值扰动可以较快地收集到扰动结果的统计数据。而且由于这种扰动趋向于夸大误差，因此漏报率会相对比较低。即如果数值计算程序在数值扰动下仍保持稳定的输出，基本上就不会因实现上的不稳定而产生错误了。而当我们用数值扰动对图 1(a) 中的代码进行评估，仅仅 1 bit 的扰动就产生了 2.228×10^{8} 的极差输出和 2.060 的变异系数输出，这是一个很不稳定的输出结果。

算式扰动比数值扰动在输出结果上更为保守，这可以很好地弥补数值扰动在误报率上的不足。算式扰动将算术表达式转换为一系列在实数域上的数学等价形式。当需要扰动整个程序时，算式扰动算法实际上作用于其组成的各个算术表达式。仍以图 1(a) 为例，我们将算式扰动作用于其第七行的算术表达式时，它总共有 144 个数学等价形式，图 1(c) 列出了其中的一部分。我们仍然以 1000 个 3.1415926×10^{10} 作为输入，并通过这 144 个不同的数学等价形式计算，得到的变异系数为 7.960×10^{-39}

。这意味着将此算术表达式修改成其它的数学等价形式并不能有效地增加数值计算的稳定性，如果这段程序不稳定的话，我们需要采用其它的改进办法。

将这两部分输出结果综合到一起，我们可以结合代码得到这样的结论：当前的数值计算程序并不稳定，而且我们只能采用更高精度的浮点数值或者采用更加稳定的算法来改进这一实现。幸运的是，对于这个例子来说 Knuth 给出了一个更好的算法 [13]，图 1(b) 给出了其实现。我们仍然用 1000 个相同的输入 3.1415926×10^{10} 来测试这个新的算法，得到的结果为 0。这一新算法在数值扰动下的极差为 3.715×10^{3}，变异系数为 0.542，其精度在算式扰动下表现稳定，各个算术表达式输出的变异系数均为 0.000。所有扰动结果都明显优于图 1(a) 中的代码。

再举一个科学实验领域的应用例子。物理学家很少把实验室观测到的数据直接发表在出版物上，而发表的却是另一个由这些数据计算而得的更重要，也更受关注的物理量。测量器具和实验环境的限制会不可避免地在直接观测到的数据上引入误差，因此，直接的观测值就会被写成 $x_0 \pm \Delta x$ 的形式。假如发表的物理量的计算形式为 $z = f(x)$，科学家会通过误差传递公式 $\frac{dz}{dx}$ 来计算 z 的误差 $\Delta z = \Delta x \frac{dz}{dx}|_{x=x_0}$。当这一过程通过数值计算程序完成时，程序还会引入由数值精度造成的误差。程序所引入的误差是否完全满足实验要求，是物理学家密切关注且必须解决的。一项最近发表在物理期刊 Physical Review D 上的研究结果 [14]发布了一个数据 $-37 \pm 88^{\text{stat}} \pm 14^{\text{syst}}\ \text{MeV}/c$，其中 88^{stat} 是实验统计误差，14^{syst} 是实验系统误差，单位 MeV 是兆电子伏特，c 是真空中的光速。假设作者在实验计算时采用了数值计算代码，设 α 代表实验量的统计误差，即这里的 $88^{\text{stat}}\ \text{MeV}/c$，而 β 代表实验量的系统误差，即 $14^{\text{syst}}\ \text{MeV}/c$。作者可以采用我们的工具来检验他们的数值计算程序，我们的工具会报告给他们程序扰动后的变异系数 (CV)，以及扰动后的样本结果中值 (z)。他们可以用下面的公式来检验数值计算的代码是否满足要求：

$$|CV| \leqslant \left| \frac{\sqrt{\alpha^2 + \beta^2}}{z} \right|.$$

这里所采用的误差累积法是误差分析领域里的标准方法 [15]。

3　扰动算法

在这一节里，我们将定义数值扰动和算式扰动的过程，并给出具体的算法和实现。扰动的过程通过自动修改程序的数值计算处理来反映程序浮点数处理稳定程度的倾向性和累积的误差程度。总的来说，这种自动修改使得程序中潜在的误差累积错误更加容易被表现出来。因此，扰动的目标是揭示用户在使用浮点数运算时的疏忽和误用，并能够度量程序在数值误差下的健壮程度。如果浮点数值计算在我们的扰动框架下的结果输出仍然稳定，程序应当基本不存在浮点数运算的稳定性错误。数值扰动算法的核心是如何在数值运算里引入受控制的随机误差，而对于算式扰动来说，我们要描述如何在给定一个表达式 e 的前提下系统的生成一系列在实数域上的数学等价形式。由于数值扰动的关注焦点在浮点数运算的底层数值，故其提供的是一个动态的视图，而算式扰动作为一种替代和补充，关注于算术表达式的静态形式。

扰动算法关注于浮点数值及其相关运算，对于程序中其他的不相关成份，算法会将其看成黑盒，在扰动的过程中不做任何改动。因此，对于扰动算法来说，程序就是一个数值型的函数。并且为了处理上的简单，算法把它分解成单个实数输出的形式 $P : I \to \mathbb{R}$，其中 I 为定义域。即当算法需要处理多个实数输出的程序时，如 $P : I \to \mathbb{R}^n$，我们可以将其分解为的 n 个单输出的函数形式 $P_i : I \to \mathbb{R}, (i = 1, 2, 3 \ldots n)$。当我们不关心程序的逻辑流程时，我们可以把函数 P 的看成由其组成的一系列算术表达式迭代而组

成，这就意味着其返回值 r 可以写成 $r = e_1 \odot e_2 \odot e_3 \odot \cdots \odot e_m$，其中 $\odot$ 为任意的浮点数操作，e_i 为函数 P 中的算术表达式。因此，我们的算法很关注于这些算术表达式的处理。图 2 给出了框架所处理的浮点运算算术表达式的语法 G，其中 v 表示程序中最基本的浮点数值，$\mathcal{E}$ 表示在浮点算术表达式的里存在的与浮点数值计算不直接相关的其他程序设计语言成份，比如说函数调用等等。

$$e ::= -e \mid e + e \mid e - e \mid e \times e \mid e \div e \mid \mathcal{E} \mid v$$

图 2: 浮点算术表达式的语法 G

3.1 数值扰动

浮点数的计算往往会带来舍入误差。普通的软件测试不能穷尽到所有输入，因而可能在某些未测试到的输入上发生误差累积特别严重而造成错误。数值扰动是在夸大舍入误差之后来判断数值计算稳定性的，如果计算程序在这样的条件下仍然很稳定，那么说明它抵抗误差累积的能力很强。这样即使偶尔在某些输入上误差累积增加，也仍能够不发生错误。

图 3: IEEE 754所定义的两种浮点数表示格式

图 3 显示了 IEEE 754 浮点数算术标准 [16]所定义的两种广泛使用的浮点数格式——单精度类型和双精度类型的浮点数。当浮点数的符号位取值为 s，底数为 b，有效数字所占的位数为 p 且其表示的整数值为 f，指数的值为 x 时，此浮点表示的精确值为公式 1 所示。注意，公式 1 所使用的是通用形式，由于目前计算机内数值表示都采用的是二进制，默认情况下 $b = 2$。

$$(-1)^s \times \frac{f}{b^{p-1}} \times b^x \tag{1}$$

当我们用浮点数域的数值 f 来表示精确的实数值 r 时，往往会有误差产生。为了模拟和展现 f 的误差，我们将其有效数字中长度为 k 的尾部随机的替换为二进制 k-比特串。事实上，这相当于我们用 f 有效数字除尾部外的部分连接到 2^k 种随机可能的尾部来表示实数 r，从而使得所模拟的表示误差近似的服从均匀分布。这种基于 k-比特位级的扰动比直接计算 $r(1 \pm \kappa)$ 效率更高，因为这个方法直接基于浮点表示的有效数字，所需要的计算操作少。设当前浮点格式所能表示的数值全集为 F，则此基本扰动过程可以用函数 $p_v : \mathbb{N}_1 \times F \to F$ 来表示，图 4 给出了这个函数在 C 语言下的具体实现。

对于程序中的每一个算术表达式，数值扰动通过一个转换函数来将基本扰动函数传递到每一个浮点数值。公式 2 给出了这个表达式转换函数 $p(e)$。为了书写上的简便，我们在这里把函数 p 和 p_v 中的第二个参数——扰动长度 k 省略了。表达式转换函数递归的将基本数值扰动函数 p_v 作用于表达式

```
1 double p_v (double value, int bits){
2    byte *byte_array = (byte*)(&value);
3    for(i = 0; i < bits/8; ++i){
4       byte_array[i] ^= (byte)rand();
5    }
6    byte_array[i] ^= (0xFF >> (8-bits%8)) & ((byte)rand());
7    return value;
8 }
```

图 4: 基本扰动函数 p_v 的具体实现

输入: $f[e_1, \cdots, e_n]$ {待测试的数值计算函数}
输入: I {f 的测试用例集 (输入集)}
1: $\forall i \in I$ **do**
2:　　输出　$f[p(e_1)/e_1, \cdots, p(e_n)/e_n](p_v(i))$
3: **end for**

图 5: 数值扰动测试算法

中的每一个输入数值和中间操作结果数值。函数 $eval(\mathcal{E})$ 对程序设计语言成份 $\mathcal{E}$ 做具体的执行，并返回出 $\mathcal{E}$ 所代表的数值表达式。

$$p(e) = \begin{cases} p_v(v), & e = v \\ p_v(eval(\mathcal{E})), & e = \mathcal{E} \\ -p(e_1), & e = -e_1 \\ p_v(p(e_1) + p(e_2)), & e = e_1 + e_2 \\ p_v(p(e_1) - p(e_2)), & e = e_1 - e_2 \\ p_v(p(e_1) \times p(e_2)), & e = e_1 \times e_2 \\ p_v(p(e_1) \div p(e_2)), & e = e_1 \div e_2 \end{cases} \tag{2}$$

我们通过随机扰动浮点数值有效数字的尾部来对浮点数由于近似而产生的误差进行统计建模，在这个过程里我们不仅仅模拟了表示误差，也同时通过扰动表达式计算的中间结果来模拟计算误差。举例来数，当我们对图 1(a) 中的第七行使用表达式转换函数 p 时，它将其减法中的减数表示转换成 $p_v(p_v(p_v(\mathrm{sum}) \times p_v(\mathrm{sum})) \div p_v(\mathrm{n})))$。当各次扰动引入的误差为 ϵ_i 并且令 $\epsilon_5 = \epsilon_0\epsilon_3 + \epsilon_1\epsilon_3 + \epsilon_0\epsilon_1\epsilon_3$，此数值扰动引入的累积误差如公式 3 所示：

$$\frac{\mathrm{sum}^2(1+\epsilon_5)}{\mathrm{n}(1+\epsilon_2)}(1+\epsilon_4). \tag{3}$$

图 5 描述了数值扰动的整体测试算法，它将公式 2 所定义的转换函数 $p(e)$ 作用于程序中每一个浮点算术表达式，从而将函数 p_v 插装到具体的代码位置。p_v 的具体实现在图 4 中，当编译完成以后，我们把 p_v 的二进制代码直接链接到可执行程序上就可以使用了。在实现工具时，我们采用了 C 语言的转换库 CIL [17]对程序中每个浮点数值输入和计算点插入函数调用自定义函数 p_v 进行扰动。

3.2　算式扰动

在算式扰动里，我们将算术表达式通过变换，形成一系列语法上不同，但在实数域上数学语义相同的表示形式，并观察这些不同的形式对程序数值稳定性的影响。在这一节里，我们主要介绍对于给定的某一个算术表达式 e，穷尽生成其一系列数学等价形式的相关算法，并在后续章节提供相关的优化方法。

首先我们定义一个我们规定的算术表达式的规范形式，我们将每个我们所需要处理的算术表达式转换成这一规范形式然后再做进一步处理。规范形式是一种完全展开的算术表达式形式，即不能再对这样的式子用分配律进行分配而只能对其进行公因式提取。得到规范形式的算术表达式之后，我们

对它进一步做括号化以及交换律、结合律的变换，最后我们对前一步的输出做公因式提取，这正好是一个应用分配律的相反操作。通过将公因式提取算法应用于交换律、结合律的算法的结果，我们可以遍历到原始算术表达式在实数域上数学等价的表达式集。

定义 1 (算术表达式规范形式)**.** *我们称一个满足图 2 所定义的语法 G 的表达式 e 为一个***表达式规范形式***，当且仅当它满足如下性质:*

1. *它是一个最简表达式，即它不能有诸如加零或者乘一之类的冗余操作;*
2. *它没有二元减法操作;*
3. *它只有倒数形式的除法，除此以外没有除法操作;*
4. *它不能被乘法分配律进一步展开。*

当给定我们一个任意表达式时，我们可以丢弃其多余的加零或者乘一之类的冗余操作，然后采用如图 6 所示转换系统 Θ 将其转换为规范形式。这个转换系统采用 R8 和 R12 这两条等价规则来保证转换产生的表达式满足定义 1 中的性质 2 和性质 3。为了满足性质 4，我们循环的应用这个转换系统进行迭代，直到输出的表达式不再发生变化，从而保证表达式中的所有多项式被完全展开，从而在算法施行阶段，我们只需要考虑因式提取就可以了。并且因为转换系统处理了二元的减法和除法，在算法的施行阶段，我们只需要考虑满足交换律的加法和乘法，并且能统一的应用交换律算法。

$$
\begin{aligned}
&\text{(R1)} && \Theta(v) \to v \\
&\text{(R2)} && \Theta(\mathcal{E}) \to \mathcal{E} \\
&\text{(R3)} && \Theta(-(-e_1)) \to \Theta(e_1) \\
&\text{(R4)} && \Theta(-(e_1 + e_2)) \to \Theta(-e_1 - e_2) \\
&\text{(R5)} && \Theta(-(e_1 \times e_2)) \to \Theta(-e_1 \times e_2) \\
&\text{(R6)} && \Theta(-e_1) \to -\Theta(e_1) \\
&\text{(R7)} && \Theta(e_1 + e_2) \to \Theta(e_1) + \Theta(e_2) \\
&\text{(R8)} && \Theta(e_1 - e_2) \to \Theta(e_1) + \Theta(-e_2) \\
&\text{(R9)} && \Theta(e_1 \times (e_2 \pm e_3)) \to \Theta(e_1 \times e_2) + \Theta(\pm e_1 \times e_3) \\
&\text{(R10)} && \Theta((e_1 \pm e_2) \times e_3) \to \Theta(e_1 \times e_3) + \Theta(\pm e_2 \times e_3) \\
&\text{(R11)} && \Theta(e_1 \times e_2) \to \Theta(e_1) \times \Theta(e_2) \\
&\text{(R12)} && \Theta(e_1 \div e_2) \to \Theta(e_1) \times \Theta(1/e_2)
\end{aligned}
$$

图 6: 表达式规范形式转换系统 Θ

我们举个例子来说明转换系统 Θ 的操作。假定我们有如下表达式：

$$e = (x + y) \times (a - b). \tag{4}$$

转换系统 Θ 依次试图用各转换规则作用于这个表达式。第一个起作用的规则是 R9，它将这个表达式转化为 $\Theta((x + y) \times a) + \Theta(-(x + y) \times b)$ 的形式，然后 R10 进一步将其左右分别展开，迭代终止的时

候，产生的规范表达式形式为：

$$\Theta(e) = x \times a + y \times a + (-x \times b + (-y \times b)). \tag{5}$$

下面我们继续介绍在将算术表达式转化为规范形式后的变换算法。算术表达式可以由一颗以操作符为中间节点，以基本的浮点数值 v 或者语言中其他程序设计语言成份 $\mathcal{E}$ 为叶节点的抽象语法树来表示。图 7b - 图 7d 描述了给定某一个输入表达式 e，穷尽产生一系列其数学等价形式的主要算法，在这些算法里，我们定义算式单元如下：

定义 2 (算式单元). 在表示某算术表达式抽象语法树的所有子树中，假如某个子树的根节点操作符不同于父树根节点操作符，且没有更大的满足条件的子树包含它，则称该子树所表示的子表达式为原算术表达式的**算式单元**。

通俗的讲算式单元就是算术表达式的最后一轮加法 (或乘法) 的操作对象，如公式 5 中的表达式最后一轮做的是四部分之和，那么它的算式单元就有四个算式单元，即 $x\times a$、$y\times a$、$-x\times b$ 和 $-y\times b$。图 7c 的第 1 行所调用的函数 $units : E \to 2^E$ 返回某一个算术表达式的所有算式单元。图 7c 的第 4 行所调用的函数 $root : E \to \{+, \times\}$ 返回某一算术表达式的根节点操作符，即最后一步运算的运算符。

图 7b 给出了穷尽产生数学等价形式的主函数 perturb 算法，它通过调用图 7c 定义的交换结合重组算法 $t_{ac} : E \to 2^E$ 和图 7d 定义的公因式提取算法 $f : E \to 2^E$ 来完成其功能。通过对转化为规范形式的算术表达式先进行交换结合律穷举，然后对其每一个结果进行公因式提取穷举，我们可以穷尽产生某个表达式的加法乘法交换律、结合律和分配律范围内所有实数域的数学等价形式。

图 7c 定义了函数 $t_{ac} : E \to 2^E$，它对输入的表达式利用加法和乘法的交换律及结合律，穷尽产生实数域内的数学等价形式，并返回这些等价形式的算术表达式集。它首先在第 1 行通过调用 *units* 提取表达式是所有算式单元，并得到算式单元的全排列。对于每一种排列，在第 4 行通过调用图 7a 所定义的加括号算法 z 而得到所有可能的结合方式。最后在第 7 行，当算式单元不是基本类型 (而是子表达式) 时，我们递归的对这个算式单元应用这一算法，并在第 8 行把算式单元的递归结果代入总结果。这样交换律和结合律所能覆盖到的形式就都能产生了。

图 7a 定义了 t_{ac} 所用到的加括号算法 $z : \mathcal{S} \times \{+, \times\} \to 2^E$，它的输入是一个算式单元排列和一个加法或乘法运算符。通过递归的将原始排列分成不同的两部分 s_1,s_2，我们穷举出所有不同的优先级树，从而实现了所有可能的加括号方式。

设某一个表达式 e 的算式单元个数为 n，我们可以用公式 6 所示的卡塔兰数来计算加括号函数 z 产生的表达式个数。

$$C(x) = \frac{(2x)!}{x!(x+1)!} \tag{6}$$

图 7c 所定义的交换结合重组算法 t_{ac} 所产生的表达式个数由此可得：

$$N_{tac}(e) = \begin{cases} 1, & e \in \{v, \mathcal{E}\} \\ n! \times C(n-1) \times \prod_{i=1}^{n} N_{tac}(u_i), & \text{otherwise.} \end{cases} \tag{7}$$

其中 $u_i \in units(e), n = |units(e)|$。这个数字随着 n 和 u_i 的增长速度很快，诸如公式 5 中的简单表达式，t_{ac} 可以产生出的1920个不同的等价形式。

图 7d 定义了公因式提取算法 $f : E \to 2^E$，它的作用是产生在乘法分配律形式下等价的算术表达式，我们将这个算法作用于交换结合重组算法 t_{ac} 的结果以达到产生所有实数域内数学等价表达式的目的。由于在前文定义的规范形式里我们已经将算术表达式完全展开，并且规范式的性质不会被交换结合重组算法 t_{ac} 破坏，故这个算法只需要考虑公因式提取就可以了。

定义 3 (公因式提取点). 设 $L(G)$ 为符合图 2 所定义的所有有效表达式组成的语言集。对于任意 $a,b,c,d \in L(G)$，我们称一个在形如表达式 $a \times b + c \times d$ 中的运算符 $+$ 为一个**公因式提取点**，当且仅当 $a = c$ 或 $b = d$。

在图 7d 中，第 6 行的函数 $fp : E \to 2^P$ 返回了 e' 中的所有公因式提取点的点集。我们以公式 5 中的算术表达式为例，它首先将第一个与第三个加号标定为公因式提取点，然后通过图 7d 第 8 行的方法，得到对公因式提取点做组合提取而产生的不同的算术表达式。组合提取在有些情况下会产生新的公因式提取点，例如在公式 5 中我们同时提取第一个和第三个加号的时候，表达式就变成形如 $+(x+y)\times a+(-(x+y)\times b)$ 的形式，这样第二个二元加号就成为了新产生的公因式提取点。函数 f 在图 7d 的第 3 行到第 13 行循环中迭代的寻找新的公因式提取点，直到没有新的算术表达式形式产生为止。

图 7e 给出了算式扰动的测试算法，它通过在第 2 行调用图 7b 所定义的扰动函数 perturb，逐个按次序的变换输入函数 f 中的算术表达式。函数 perturb 产生了表达式的所有实数域等价形式，从而在第 3 行每次用其中的一个等价形式 e'_j 对相应的算术表达式 e_j 做替换。从本质上说，这个算法相当于把待测函数 f 分解成了 n 个函数族 $f_j : I \to \mathbb{R}$，且 $f_j = e_1 \odot \cdots \odot e_{j-1} \odot e'_j \odot e_{j+1} \odot \cdots \odot e_n$，其中 $1 \leqslant j \leqslant n$，$\odot$ 表示一种任意的复合算子。这意味着这个算法一次仅对一个算术表达式进行扰动，而不是针对一个表达式子集进行扰动。每个函数族 f_j 代表着一个针对 e_j 进行扰动的函数集合，集合的大小为 e_j 的实数域等价算术表达式的个数。值得注意的是，无论函数 f 的具体语义如何，所有的函数族 f_j 在实数域上的输出应该完全相同，因此我们能够通过对比在浮点数域上函数族 f_j 的输出值的差异，判断 f 在浮点数值上的稳定性。在算式扰动的实现上，我们采用了 C 和 C++ 语言的静态分析框架 ROSE[18]。ROSE 拥有较丰富的处理抽象语法树 (AST) 的机制，我们使用了其中的 visitor 设计模式来遍历输入函数算术表达式的抽象语法树。

4 算法优化

虽然算式扰动算法针对程序在数值计算上的稳定性具有误报率低，定位准确的优点，是数值扰动的一个很好的补充，但是由于在实数域上数学语义相同但语法不同的算术表达式空间常常很大，直接使用基本的算式扰动算法的时间复杂度会很高。例如对于算术表达式 $e = (x+y)\times(y+z)\times(z+x)$，由公式 7 可以算出 $N_{tac}(e) = 7,437,513,790,586,880$，这意味着单单交换结合律等价的算术表达式形式的数目就有这么多。在本章里，我们采用两种不同的优化算法来解决这个问题，当用户拥有包括多核在内的并行计算硬件资源时，可以利用多核并行优化算法提高算式扰动的效率；否则，用户可以使用蒙特卡罗方法来抽样求解。用户可以结合使用这两种方法进行优化，以达到更好的效果。

4.1 多核并行优化算法

随着多核处理器的日益普及，如何高效的利用多核处理器的计算性能来提高工具软件的实用性也被提上了议事日程。我们的算式扰动由于存在效率问题，利用并行资源成倍的提高算法的效果正是一个很好的解决方法。我们采用了 OpenMP 针对共享内存多处理器 (SMP) 结构进行优化。算法的本质

输入: $s \in \mathcal{S}, \odot \in \{+, \times\}$
1: $T \leftarrow \emptyset$
2: **if** $s \neq \epsilon$ **then**
3: $\quad \forall s_1, s_2 . s_1 s_2 = s$ **do**
4: $\quad\quad T \leftarrow \{(t_1 \odot t_2) \mid t_1 \in z(s_1, \odot), t_2 \in z(s_2, \odot)\} \cup T$
5: $\quad$ **end for**
6: **end if**
7: **返回** T

(a) 加括号算法　$z : \mathcal{S} \times \{+, \times\} \to 2^E$

输入: $e \in E$
输入满足条件: $e \neq v \wedge e \neq \mathcal{E}$
1: $U \leftarrow units(e)$ {e的所有算式单元全集}
2: $T \leftarrow \emptyset$
3: $\forall s \in \{U$的全排列集$\}$ **do**
4: $\quad T \leftarrow T \cup z(s, root(e))$
5: **end for**
6: $\forall u_i \in U \wedge u_i \neq v \wedge u_i \neq \mathcal{E}$ **do**
7: $\quad S \leftarrow t_{ac}(u_i)$
8: $\quad T \leftarrow \bigcup_{u_i' \in S} T[u_i / u_i']$
9: **end for**
10: **返回** T

(c) 交换结合重组算法　$t_{ac} : E \to 2^E$

输入: $f[e_1, \cdots, e_n]$ {待测试的数值计算函数}
输入: I {f 的测试用例集 (输入集)}
1: **for** $j \leftarrow 1; j \leqslant n; j \leftarrow j + 1$ **do**
2: $\quad \forall i \in I, \forall e_j' \in \text{perturb}(e_j)$ **do**
3: $\quad\quad$ 输出$f[e_j / e_j'](i)$
4: $\quad$ **end for**
5: **end for**

(e) 算式扰动测试算法

输入: $e \in E$
1: **if** $e = v \vee e = \mathcal{E}$ **then**
2: $\quad$ **返回** $\{e\}$
3: **end if**
4: $R \leftarrow \emptyset, T \leftarrow t_{ac}(e)$
5: $\forall e' \in T$ **do**
6: $\quad R \leftarrow R \cup f(e')$
7: **end for**
8: **返回** R

(b) 主函数算法　perturb:$E \to 2^E$

输入: $e \in E$
输入满足条件: $e \neq v \wedge e \neq \mathcal{E}$
1: $E_l \leftarrow \{e\}, R \leftarrow \emptyset$
2: $P_o \leftarrow \emptyset$ {已经访问到的节点}
3: **repeat**
4: $\quad E_n \leftarrow \emptyset$
5: $\quad \forall e' \in E_l$ **do**
6: $\quad\quad P_n \leftarrow fp(e') \setminus P_o$ {新的公因式提取点}
7: $\quad\quad \forall P_a \in 2^{P_n}$ **do**
8: $\quad\quad\quad E_n \leftarrow E_n \cup \{factor(e', P_a)\}$
9: $\quad\quad$ **end for**
10: $\quad\quad P_o \leftarrow P_o \cup P_n$
11: $\quad$ **end for**
12: $\quad R \leftarrow R \cup E_l, E_l \leftarrow E_n$
13: **until** $E_l = \emptyset$
14: **返回** R

(d) 公因式提取算法　$f : E \to 2^E$

图 7: 穷尽产生某一算术表达式实数域数学等价形式的主要算法

是按照所需要的粒度，提取出转换程序中相对独立的运行部分，并利用 OpenMP 将这些部分以不同的线程的形式分配到不同的处理器或处理器核心上运行。

由公式 8 可以知道，算式扰动产生算术表达式的总个数是各步骤的乘积。算式扰动复杂度问题的根本原因在于 N_{tac} 太大，当交换结合律穷举产生出很多结果之后，我们可以并行的对各个结果独立的做公因式提取而互不干扰，另外，在交换结合律和公因式提取内部也有很多互相独立的计算。因此，我们的并行算法优化主要用于改进交换结合重组算法和公因式提取算法。

输入: $e \in E$
输入满足条件: $e \neq v \wedge e \neq \mathcal{E}$
1: $U \leftarrow units(e)$ {e 的所有算式单元全集}
2: $T \leftarrow \emptyset$
3: **parallel** $\forall s \in$ {U的全排列集} **do**
4: $T \leftarrow T \cup z(s, root(e))$
5: **end parallel for**
6: **parallel** $\forall u_i \in U \wedge u_i \neq v \wedge u_i \neq \mathcal{E}$ **do**
7: $S \leftarrow t'_{ac}(u_i)$
8: $T \leftarrow \bigcup_{u'_i \in S} T[u_i/u'_i]$
9: **end parallel for**
10: **返回** T

(a) 交换结合重组并行优化 $t'_{ac} : E \rightarrow 2^E$

输入: $e \in E$
输入满足条件: $e \neq v \wedge e \neq \mathcal{E}$
1: $E_l \leftarrow \{e\}, R \leftarrow \emptyset$
2: $P_o \leftarrow \emptyset$
3: **repeat**
4: $E_n \leftarrow \emptyset$
5: **parallel** $\forall e' \in E_l$ **do**
6: $P_{ne'} \leftarrow fp(e') \setminus P_o$ {对于不同的 e' 并行的寻找新的公因式提取点}
7: **parallel** $\forall P_a \in 2^{P_{ne'}}$ **do**
8: $E_n \leftarrow E_n \cup \{factor(e', P_a)\}$ {同一个 e' 并行做组合公因式提取}
9: **end parallel for**
10: **end parallel for**
11: $P_o \leftarrow \bigcup_{e' \in E_l}(P_o \cup P_{ne'})$ {将结果整合.提取点加入到 P_o}
12: $R \leftarrow R \cup E_l$ {提取到的算式加入 R，用于返回}
13: $E_l \leftarrow E_n$ {新提取的算式式加入 E_l，用于下一轮提取}
14: **until** $E_l = \emptyset$
15: **返回** R

(b) 公因式提取并行优化 $f' : E \rightarrow 2^E$

图 8: 并行优化算法

图 8a - 图 8b 描述了多核优化后的交换结合重组算法 t'_{ac} 和公因式提取算法 f'。这几个算法利用 OpenMP 的 parallel 结构把并行区域分配到不同线程上执行。在图 8a 中，当在第 3 行得到了完整的 U 的全排列集合后，可以并行的对各个排列做加括号处理，并在 6-9 行的算式单元计算与代入过程中也可以动用并行算法同步进行。需要注意的是，采用并行算法以后在第 3 行和第 6 行的并行分配点以及第 5 行和第 9 行的结果集中点都需要额外的计算开销。在图 8b 中，第 5 行和第 7 行构成了嵌套的并行结构，第 5 行的 parallel 结构针对目前需要做公因式提取的多个不同的表达式，来并行的做公因式提取。第 7 行的 parallel 结构针对的是同一个表达式 e' 的不同公因式提取点组合，来并行的做公因式提取。需要注意的是，在这个算法里除了有并行分配和集中的同步开销，还会出现由于同步而产生的并行不足，不能有效利用硬件并行资源的问题。比如，在刚刚进入算法 f' 的时候，$|E_l| = 1$，从而在图 8b 的第 5 行并行分配的时候只能有一个线程被分配运行，而在经过第 13 行新提取的算式加入 E_l 以后，再次经过循环来到第 5 行才能得到比较充分的并行分配。

为了对复杂的算术表达式做进一步的优化，在实现并行算法的时候，我们利用了 OpenMP 3.0 的 task 特性[19]。在实现 t'_{ac} 的时候，我们在做第 7 行递归对算式单元调用交换结合律算法的同时，并行的计算第 4 行的整体算式的交换结合重组算法。这需要在第 1 行获得算式单元全集的时候就进行 OpenMP 3.0 的 task 分配，每个 task 独立运行。一直到结果返回之前再进行结果的整合。在这一过程中，经常要用到抽象语法树 (AST) 片段的复制和替换，我们使用了 Map 数据结构对复制后的抽象语法树片段保存节点复制对应关系，便于回溯，但这样就会带来共享数据的同步访问操作。由于 OpenMP 本身不具有读写锁的功能，我们使用 OpenMP 提供的互斥锁实现了一个经典的读者优先的读写锁来完成同步[20]。经过了这样的设计与实现，算式扰动可以充分的利用计算机的硬件并行资源来有效的达到我们需要的扰动效果。在 5.3 节我们将会讨论并行优化的实验评估效果。

4.2　蒙特卡罗优化算法

在硬件资源不足，或者算术表达式的复杂程度过高，传统方法与多核方法都无法胜任算式扰动算法的执行时，我们可以做这样的可行性折衷，即不去穷尽所有在实数域上数学等价的算术表达式形式，而通过对其做一个随机抽样，并以抽样的统计性质来作为对整体统计性质的估计。这一算法我们称为蒙特卡罗算式扰动优化算法。

定义 4 (抽样数限制). 对于算术表达式 e，使用交换结合律产生的在实数域等价的表达式总数为 $N_{tac}(e)$，而我们所期望在抽样后的样本个数为 L，我们称之为**抽样数限制**。对于一次蒙特卡罗抽样来说，抽样数限制是由用户给出的。

定义 5 (算式长度). 我们将一个算术表达式的抽象语法树 *(Abstract Syntax Tree, AST)* 的叶节点个数称为**算式长度**，即算术表达式中基本浮点数值与数值无关成份 $\mathcal{E}$ 的总数。

我们知道，需要对算式扰动进行优化的根本原因是 N_{tac} 太大，而由于公因式提取算法作用在交换结合律穷举的每一个结果上，故而我们有

$$N_{all} = N_{tac} \times N_f \tag{8}$$

这样最终生成的算术表达式总数就更大了，只要我们对 N_{tac} 进行充分的抽样，就可以达到使 N_{all} 整体减小的目的。因而蒙特卡罗优化只要针对交换结合律穷尽算法就可以了。

让我们重新对照图 7c 来分析一下公式 7，N_{tac} 的数值在算术表达式不是基本单元 (算式长度为 1) 时由三部分决定 (设当前表达式的算式单元个数为 n)：图 7c 第 3 行 U 的全排列个数 $N_p = n!$、第 4 行 z 产生的加括号总数 $N_z = C(n-1) = \frac{(2n-2)!}{(n-1)!n!}$ 以及由算式单元递归而得到的算式总数$N_r = \prod_{i=1}^{n} N_{tac}(u_i)$。因此我们有：

$$N_{tac} = N_p \times N_z \times N_r. \tag{9}$$

我们希望在对数目为 N_{tac} 的总体做出数目为 L 的抽样后尽可能的对统计性质产生无偏估计，即尽量保证每个算术表达式被抽样到的概率为 L/N_{tac} 。而算式扰动过程是分步进行的，即我们需要使这个抽样数限制合理的分配到每一步并且不能破坏其一致性，即要求公式能够保证：

$$\frac{L_p}{N_p} \times \frac{L_z}{N_z} \times \frac{L_r}{N_r} = \frac{L}{N_{tac}} \tag{10}$$

其中，L_p/N_p 是全排列的选择概率，L_z/N_z 是加括号步骤的选择概率而 L_r/N_r 是递归步骤的选择概率。故而我们把各步骤的抽样数限制定义成如下形式：

$$L_p = L^{\frac{\ln N_p}{\ln N_{tac}}}, L_z = L^{\frac{\ln N_z}{\ln N_{tac}}}, L_r = L^{\frac{\ln N_r}{\ln N_{tac}}}, \tag{11}$$

在实际的优化过程中，我们很难在算式扰动之前计算出 N_r 的值 (公式 7 对 N_r 的计算是一个递归式，而不是一般式)，由公式 7 可以看出，各子层递归的算式单元总数决定了 N_r 值的大小，我们可以用算式长度 k 来作为各子层递归算式单元总数的一个估计，并以 $\mu = \ln N_{tac}$ (可由公式16 解得具体值) 作为递归层数的估计。我们有：

$$\frac{N_r}{N_p \times N_z} \approx (\frac{k}{n})^{\mu} \tag{12}$$

其中当前算术表达式的算式长度为 k，算式单元个数为 n。由此我们可以推出：

$$\frac{L_r}{L_p \times L_z} = L^{\frac{\ln N_r - \ln N_p - \ln N_z}{\ln N}} = L^{\frac{\ln \frac{N_r}{N_p N_z}}{\mu}} \approx L^{\ln(\frac{k}{n})^{\mu/\mu}} = L^{\ln(\frac{k}{n})} \tag{13}$$

由公式 11 我们可以得到 $L = L_p \times L_z \times L_r$，且由于 L_p、L_z 和 L_r 都是正数。和公式 13 联立，我们可以得到：

$$\begin{aligned} L_r &= \sqrt{\frac{L_r}{L_p \times L_z} \times L} \approx \sqrt{L^{\ln(\frac{k}{n})} \times L} = L^{\ln\sqrt{\frac{ke}{n}}} \\ L_p \times L_z &= L / L_r \approx L^{\ln\sqrt{\frac{ne}{k}}} \end{aligned} \tag{14}$$

注意，这里的 e 指的是欧拉数，而不是算式表达式。对公式 11 到公式 14 联立求解，可以解得如下重要结论：

$$\mu = \ln N \approx \frac{\ln N_p N_z}{\ln\sqrt{\frac{ne}{k}}} \tag{15}$$

$$L_p \approx L^{\frac{\ln\sqrt{\frac{ne}{k}} \ln N_p}{\ln N_p N_z}}, L_z \approx L^{\frac{\ln\sqrt{\frac{ne}{k}} \ln N_z}{\ln N_p N_z}} \tag{16}$$

虽然公式 14 给出了 L_r，即递归步骤的总的抽样数限制，在具体优化时我们还需要知道对于某一个具体的算式单元 u_i 在递归时所用的抽样数限制 $L_{r,i}$。同样的，我们使用 u_i 的算式长度 k_i 来作为估计依据，并且由于

$$L_r = \prod_{i=1}^{n} L_{r,i}, k = \sum_{i=1}^{n} k_i. \tag{17}$$

我们推出：

$$L_{r,i} \approx L_r^{\frac{k_i}{k}} \approx L^{\frac{k_i}{k}\ln\sqrt{\frac{ke}{n}}}. \tag{18}$$

图 9a 描述了经过蒙特卡罗抽样的交换结合重组算法 $t_{ac}^{\circ} : E \times \mathbb{N} \to 2^E$。算法新增加的一个自然数参数用于指定整体算术表达式的抽样数限制，它通过计算公式 16 和公式 18 得到具体实施抽样时的每一步的抽样数限制。在具体的抽样位置，算法做出的选择完全是非确定性的，因而能保证在递归树中的每一条递归路径能得到相同的选择概率，从而使得抽样结果仍然满足整体的统计规律。在算法第 3 行由公式 16 计算出 L_p，L_z 的值并在第 5 行和第 8 行得到使用。在第 11 行通过求解公式 18 计算出 $L_{r,i}$ 并在第 12 行得到递归使用。从而达到了对算式元素递归进行蒙特卡罗抽样的目的。

4.3 综合优化

图 9b 描述了蒙特卡罗抽样的并行算法 $t_{ac}^{\circ\prime} : E \times \mathbb{N} \to 2^E$，此算法综合了抽样优化和并行优化的特点。与图 9a 一样，在图 9b 的第 3 行由公式 16 计算出 L_p，L_z 的值并在第 5 行和第 10 行得到使用。在第 13 行通过求解公式 18 计算出 $L_{r,i}$ 并在第 14 行得到递归使用。从而实现了并行的蒙特卡罗抽样综合优化。

图 9b 第 10 行所使用到的函数 z' 是图 7a 加括号函数的蒙特卡罗优化版本。它通过针对对应步骤的抽样数限制 L_z 对加了括号的算术表达式进行抽样。此函数在图 9a 第 8 行也得到了使用。

输入: $e \in E, L \in \mathbb{N}$

1: $U \leftarrow units(e)$
2: $T \leftarrow \emptyset$
3: 已知 $L, k = |e|, n = |units(e)|$ 求解公式 16 得 L_p,L_z
4: $U_p \leftarrow U$ 的全排列
5: **for** $i := 0; i < L_p; i := i + 1$ **do**
6: 　随机选取 $s \in U_p$
7: 　$U_p \leftarrow U_p - s$
8: 　$T \leftarrow T \cup z'(s, root(e), L_z)$
9: **end for**
10: $\forall u_i \in U \wedge u_i \neq v \wedge u_i \neq \mathcal{E}$ **do**
11: 　已知 $L, k_i = |u_i|$ 求解公式 18 得 $L_{r,i}$
12: 　$S \leftarrow t^{\circ}_{ac}(u_i, L_{r,i})$
13: 　$T \leftarrow \bigcup_{u'_i \in S} T[u_i/u'_i]$
14: **end for**
15: **返回** T

(a) 经过蒙特卡罗优化的交换结合重组算法

$t^{\circ}_{ac} : E \times \mathbb{N} \rightarrow 2^E$

输入: $e \in E, L \in \mathbb{N}$

1: $U \leftarrow units(e)$ {e 的所有算式单元全集}
2: $T \leftarrow \emptyset$
3: 已知 $L, k = |e|, n = |units(e)|$ 求解公式 16 得 L_p,L_z
4: $U_p \leftarrow U$ 的全排列
5: **parallel for** $i := 0; i < L_p; i := i + 1$ **do**
6: 　**begin critical**
7: 　　随机选取 $s \in U_p$
8: 　　$U_p \leftarrow U_p - s$
9: 　**end critical**
10: 　$T \leftarrow T \cup z'(s, root(e), L_z)$
11: **end parallel for**
12: **parallel** $\forall u_i \in U \wedge u_i \neq v \wedge u_i \neq \mathcal{E}$ **do**
13: 　已知 $L, k_i = |u_i|$，求解公式 18 得 $L_{r,i}$
14: 　$S \leftarrow t^{\circ\prime}_{ac}(u_i, L_{r,i})$
15: 　$T \leftarrow \bigcup_{u'_i \in S} T[u_i/u'_i]$
16: **end parallel for**
17: **返回** T

(b) 同时使用蒙特卡罗优化与并行优化的交换结合重组算法

$t^{\circ\prime}_{ac} : E \times \mathbb{N} \rightarrow 2^E$

图 9: 优化后的交换结合重组算法

定义 6 (括号化形态). 一个算术表达式按运算优先顺序，构成的括号对的索引集，我们称为一个**括号化形态**。

值得注意的是，括号化形态只和算术表达式中算式单元的个数有关，而和具体算式单元的内容无关。因此无论是图 7c 第 4 行和图 8a 第 4 行所使用的加括号算法 z 还是图 9a 第 8 行和图 9b 第 10 行所使用优化加括号算法 z' 都可以通过产生满足抽样数限制 L_z 的对应括号化形态集，然后将具体的算式单元代入产生所需要的加括号的算术表达式。由于括号化形态的产生、修改操作、抽样等操作都远快于直接重写算术表达式，因此这一步优化的效果十分明显。

同时采用蒙特卡罗优化和并行优化的时候，需要特别注意同步问题，比如在图 9b 的第 6 行到第 9 行，我们使用了一个OpenMP自带的临界区来处理排列抽样的选择问题，以保证算法的正确性。在实现中，我们增加了互斥锁[20]的使用，这些同步机制都会在一定程度上增加算法的运算开销。但不管怎样，从 5.3 节的评估结果来看，综合使用各种优化方法还是可以进一步起到优化的作用。

5　实验与评估

我们在本章描述扰动算法的实验评估结果，以及扰动优化算法的效果。这些实验运行于 Hewlett-Packard®xw4600 工作站。这个工作站配置了 Intel Core 2 Quad Q9550 的 4 核 CPU 和 4G 内存，并且

运行了 64 位版本的 Ubuntu 10.04 操作系统以及 gcc 4.4.3 版本的编译器。我们采用了 gcc 4.4.3 内置的 OpenMP 3.0 来运行我们的多核算法。

在扰动效果上，我们采用统计后的极差和变异系数作为评估结果来输出。极差反映了在扰动输出变化的最大振幅，是一种比较保守的估计值，而变异系数反映了在扰动下软件稳定性的平均状况。我们使用变异系数作为输出的主要原因有：1) 它是一个归一化的度量值，可以用于不同软件和不同环境下的统计比较。2) 比起极差来说，它能对整个样本集做出累积的统计度量。

变异系数在数值扰动方面能比较成功的把不稳定的部分从稳定部分中分离出来，但是正如 Kahan 所描述的，数值扰动会夸大数值误差[12]。这个问题在图 13 上表现的十分明显，某些函数的最大变异系数的阶数达到了 120。数据表明，算式扰动相对更加保守，因而能作为数值扰动的补充来对数值扰动的结果做进一步的确认。

5.1 评估用例

我们的评估主要针对两类用例，一类是相关的数值计算学术工作中所用到的代码。我们将它们组成了一个学术用例测试集。这些用例能够比较好的反映出我们的工具在学术所聚焦的特定问题上的效果，也能够比较好的追踪和分析工具不稳定输出的原因。另一个用例是开源数值计算库 GNU Scientific Library (GSL)[21]，它能够体现出我们工具在实际广泛使用的大型计算库上的可伸缩性。

```
1 double xi, xsi, A;
2 xi = 1;
3 A=random(20.0,30.0);
4 xsi = 2*xi-A*xi*xi;
```

(a) Inv.c

```
1 double x = 1.0;
2 for (i=1; i<=6; i++) {
3     x = 0.5 * (x + 2.0/x);
4 }
```

(b) Newton.c

```
1 float x,z;
2 x = random(0,1);
3 z=x*x*x*x-4*x*x*x+6*x*x-4*x+1;
```

(c) Poly.c

```
1 float a,b,c,d;
2 a=98765.0; b=1.0;
3 c=5.0e-8; d=5.0e-8;
4 float A=a*((b+c)+d);
```

(d) exp.c

```
1 float a, b, c, r1;
2 a=7; b=8686; c=2;
3 r1=(-b+sqrt(b*b-4*a*c))/(2*a);
```

(e) root.c

```
1 double alpha, NA, re, Z, L, Lp, A;
2 Z = 4; A = 26; re = 11.3;
3 alpha = 0.7; NA = 12.5;
4 L = log( 184.15 / pow(Z,1.0/3) );
5 Lp = log( 1194.0 / pow(Z,2.0/3) );
6 double X0=(4.0*alpha*re*re)
```

(f) sample_run.c

```
1 int a=0; int b=0;
2 E = MatrixSolve(input);
3 for(j=0; j<n-1; j++){
4     a += Y[j] * E[0][j];
5     b += Y[j] * E[1][j]; }
6 for(i=1; i<n; i++)
7     M[i] = ( input[0] - b/a ) *
8         exp( -a * i ) * ( 1 - exp(a) );
```

(g) GM.c

图 10: 学术相关用例测试集的关键代码段

图 10 给出了相关学术用例测试集的关键代码段，其中 Inv.c 来自于 Goubault 的工作[22]，主要用于迭代计算输入数值的倒数。图 10a 给出了其中关键的数值计算部分，限于篇幅，我们省略了其循环迭代的控制部分；同样来自于 Goubault 的文献的还有图 10c 的 Poly.c，它是一个简单的多项式计算测试；图 10b 描述的 Newton.c 是一个牛顿法逼近 $\sqrt{2}$ 的处理代码，它来自于 Eggert 的工作[6]；图 10d 描

述的 exp.c 来自于 Martel[7]，他用这个用例来测试他自己的数值分析优化；图 10e 描述的 root.c 是一个二次方程求根的用例，它来自于 Parker 的文献[9]；sample_run.c 来自于一个机械工程模拟器[23]，运算逻辑相对来说比较复杂，图 10f 给出了其中关键的数值计算代码；图 10g 描述的 GM.c 来自于文献[24]，主要描述了一个预测灰色模型的数值计算。

我们使用 GSL 版本 1.11 进行评估，为了方便且不失一般性，我们按照 GSL 文档中描述的次序，从前往后选择了 48 个单值输入且单值输出的函数进行评估。这些函数的输入输出类型都是浮点数。很多函数会有发散点和非连续点 (比如在 π 点发散)，而完全使用随机测试用例生成很难测试到这些发散点附近，故而我们使用了随机测试用例生成和启发式导向用例生成相结合的方式。具体用例生成方式如下：我们在 $(-0.1, 0.1)$ 区间随机生成 30 个用例，在 $(-\pi, \pi)$ 区间随机生成 30 个用例，在 $(-100, 100)$ 区间随机生成 30 个用例，并毫无限制的随机生成 30 个用例。以上 120 个用例我们都是用均匀分布的方式生成的。另外，我们令 $\epsilon = 5 \times 10^{-10}$ 以及 $I = \{0, 1, \pi/2, 2, \pi, e\}$，则对于 $\forall i \in I$，我们在区间 $(i - \epsilon, i + \epsilon)$ 以及 $(-i - \epsilon, -i + \epsilon)$ 上各取 10 个用例 (当 $i = 0$ 时我们只取一次)。这样我们对每个待测试的 GSL 函数总共生成 230 个用例输入来进行评估。

5.2　扰动评估

图 11 - 图 12 描述了学术相关用例测试集的数值扰动效果。在评估里，我们将扰动后缀的长度 k 设置为从 1 到 16 比特。由于 $k = 0$ 时不会有扰动，输出的统计值也会完全相同，故而在横轴上我们把 0 点省略了。对于每一个 k 值，我们对程序执行 1000 次来获得有效的统计结果。图 11 给出了输出结果的极差，图 12 给出了输出结果的变异系数。从图 11 中我们可以看出，在对数纵坐标刻度下，极差和扰动后缀的长度基本上呈线性关系。从图 11 中的变异系数来看，测试用例集被清楚而又准确的划分成了两部分：Inv.c、Newton.c 和 sample_run.c 表现稳定而 exp.c、Poly.c、GM.c 和 root.c 相对不稳定。在图 11 中，不稳定的用例在 1 比特数值扰动下的极差大于 7×10^{-7}，而稳定的用例在此扰动下的极差小于 5×10^{-14}。在图 12 里，不稳定用例的变异系数大于 8×10^{-8}，举例来说，当我们使用 5 比特的数值扰动时，root.c 输出结果的变异系数为 19.8。而稳定用例的变异系数小于 9×10^{-9}。

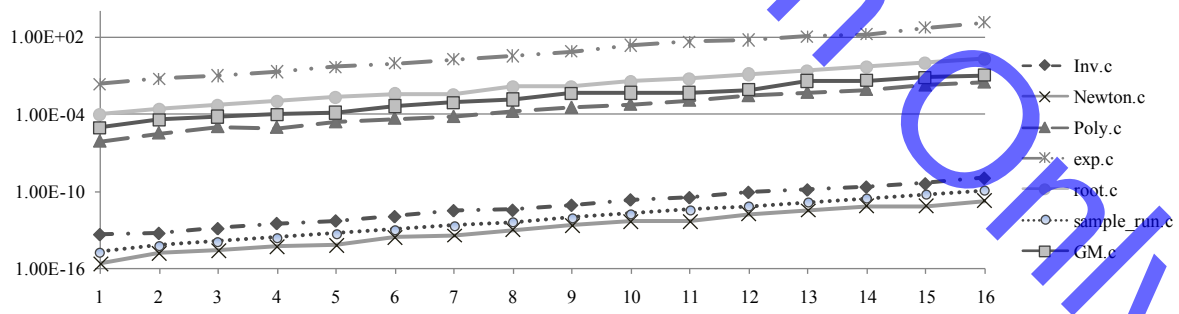

图 11: 学术相关用例测试集数值扰动结果的极差

由图 10 中的源代码可以追踪分析出这些用例在数值计算下不稳定的原因，exp.c 第四行括号里的加数完全不在一个数量级上，并且将其结果乘以了一个大数而把误差进一步放大了。对于这个算式有一种更稳定的形式：`a*b+a*(c+d)`。Poly.c 不稳定的原因是 `x` 的随机数取值可能会很接近 1，这样会使第

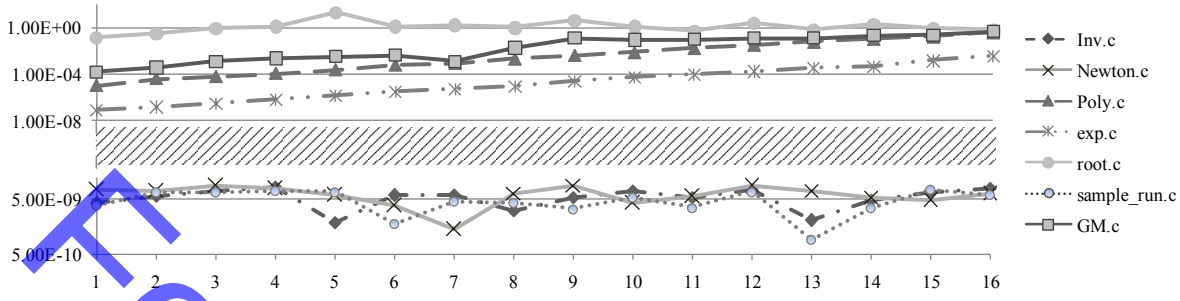

图 12: 学术相关用例测试集数值扰动结果的变异系数

三行的表达式 `z=(x-1)^4` 不稳定。root.c 的情形和图 1 类似，`sqrt(b*b-4*a*c)` 在 root.c 的取值下很接近 b 的值，而进一步的计算产生的误差会很大。GM.c 不稳定的原因是其模型的条件数较大，使得误差在求解过程中迅速积累。值得注意的是，虽然 sample_run.c 的式子很长，在实数范围内的数学等价的算术表达式很多，但它却是稳定的。这说明数值计算的数值稳定程度和算式长短并没有必然联系。

表 1 给出了学术相关用例测试集的算式扰动评估结果，我们采用了图 7e 所描述的测试函数来完成这一评估。对于测试用例集中的每一段代码，我们选取其最复杂的算术表达式作为疑似不稳定算式，并分别以 5, 50, 100, 300, 500 的抽样数限制进行蒙特卡罗优化下的算式扰动。在对应算术表达式比较简单的情况下，我们还会对该算式进行不优化的原始扰动，以便得到对应的对比结果。当抽样数限制小于能在实数域产生的数学等价算术表达式总数时，抽样的选择是均匀分布的。在得到的各个等价算术表达式的结果后，我们对这些结果分别计算极差和变异系数。用例 Inv.c 和 Poly.c 代码中有随机数生成语句 (见图 10a 第三行以及图 10c 第二行)，因而这两个用例在算式扰动过程中被进一步重复执行了 1000 次。

在表 1 中，每一个用例有两行数据，其中一行记录的是算术表达式扰动结果的统计极差，另一行是扰动结果的变异系数。在第一列的用例名字后面，我们加了一个括号来记录我们扰动的算术表达式在实数域内的数学等价形式个数。由于 Poly.c、sample_run.c 和 GM.c 的个数过大，超出了我们能使用原始扰动的能力范围，故而其用例名后的括号里和最后一列都用 (-) 来表示。

在算式扰动过程中，Inv.c 和 Newton.c 是完全稳定的，所有实数域等价的算术表达式在浮点数域完全等价，输出结果完全没有变化。sample_run.c 和 root.c 在算式扰动下表现的还是基本稳定的，它们结果的变异系数小于 9.00×10^{-9}，极差小于 8.00×10^{-12}。这意味着单单通过改变算术表达式的形式很难在这两个用例中对数值计算的稳定性起到改进作用。Poly.c、exp.c 和 GM.c 不稳定，exp.c 代码里最复杂的算术表达式只有 168 个实数域内的数学等价形式，然而它的极差却超过 5.00×10^{-3}。Poly.c 和 GM.c 在抽样较多时，结果的极差也分别达到了 1.19×10^{-7}和 3.22×10^{-7}。

图 13 给出了在 5 比特扰动下的对于 48 个 GSL 函数采用不同参数输入情况下的输出结果的最大变异系数，其输入生成过程已经在 5.1 节的开头详细描述。对于每一个测试函数和每一种扰动后缀长度，我们的将用例重复运行了 1000 次来获得较有效的统计结果。

限于篇幅，完整的结果数据集可以从 `http://seg.nju.edu.cn/~eytang/numerical_perturbation/GSLResult.tar.gz` 处下载。对于同一个函数来说，使用不同的扰动后缀长度，结果的变异系数差距不是很大，例如，对于函数 gsl_sf_bessel_I1_scaled 其从 1 到 16 的扰动后缀下最大变异系数都在 1.25×10^{-17}

表 1: 学术相关用例测试集的算式扰动统计结果

抽样数限制		5	50	100	300	500	all
Inv.c(48)	(极差)	0	0	0	0	0	0
	(变异系数)	0	0	0	0	0	0
Newton.c(56)	(极差)	0	0	0	0	0	0
	(变异系数)	0	0	0	0	0	0
Poly.c(-)	(极差)	5.96E-08	1.19E-07	1.19E-07	1.19E-07	1.19E-07	-
	(变异系数)	5.455E-08	4.056E-08	3.599E-08	4.818E-08	4.815E-08	-
exp.c(168)	(极差)	7.81E-03	1.56E-02	1.56E-02	1.56E-02	1.56E-02	1.56E-02
	(变异系数)	3.64E-08	4.798E-08	5.227E-08	5.048E-08	5.048E-08	5.05E-08
root.c(1920)	(极差)	2.27E-13	2.27E-13	2.27E-13	2.27E-13	2.27E-13	2.27E-13
	(变异系数)	6.782E-09	6.093E-09	6.086E-09	5.987E-09	5.903E-09	7.08E-09
sample_run.c(-)	(极差)	3.64E-12	7.28E-12	7.28E-12	7.28E-12	7.28E-12	-
	(变异系数)	7.184E-09	8.452E-09	8.783E-09	8.055E-09	8.179E-09	-
GM.c(-)	(极差)	1.36E-07	3.21E-07	3.22E-07	3.22E-07	3.22E-07	-
	(变异系数)	6.782E-08	6.926E-08	7.015E-08	7.092E-08	7.093E-08	-

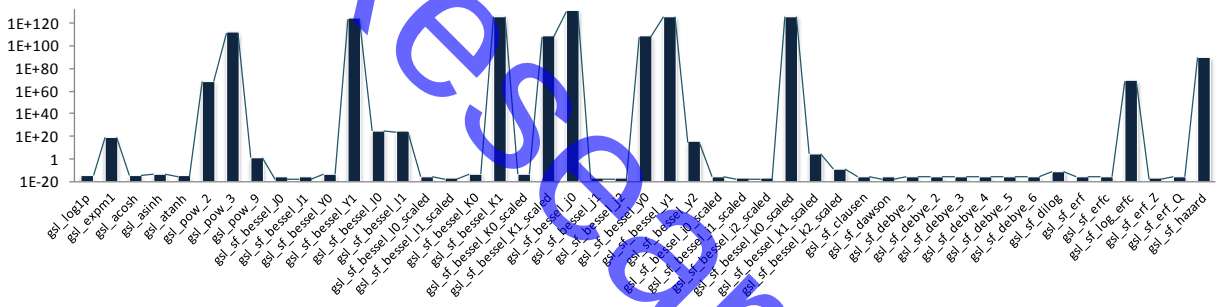

图 13: GSL 在 5-bit 数值扰动下对于测试输入的最大变异系数

到 1.42×10^{-17} 之间，gsl_sf_erf 在 1 比特扰动后缀下的最大变异系数为 1.11×10^{-16}，此时的函数输入为 15.2，而在 16 比特扰动后缀下的最大变异系数为 1.07×10^{-16}，此时的函数输入为 78.1；而不同测试函数之间的结果变异系数差距很大，整体 48 个函数在 6 比特扰动后缀下的变异系数的方差达到 1.12×10^{266}。

在图 13 里，横坐标给出了被扰动测试的函数名，纵坐标是对数刻度下输出结果的相对于生成的测试输入的最大变异系数。有些函数对数值函数相当敏感，比如 gsl_sf_bessel_j0 和 gsl_sf_bessel_K1 的最大变异系数分别达到了 1.62×10^{131} 和 1.64×10^{126}。而很多函数的变异系数比较小，其中有 30 个函数的变异系数小于 1。这些变异系数很大的函数有可能是因为数值扰动的误报而引起的，因此我们需要进一步用算式扰动来检验它们。

表 2 记录了对图 13 中变异系数超过 1.00×10^{60} 的函数做进一步算式扰动的确认结果。它对 GSL 函数采用了和数值扰动相同的输入测试用例。我们采用了图 7e 所描述的测试过程来完成这一评估，并对每一个在函数执行路径上的算术表达式进行扰动。通过运行这些函数得到这些路径上算术表达式在实数域上数学等价形式的计算值，并输出这些值的统计极差和变异系数。表 2 记录了这其中最大的极差和变异系数，以及产生这最大极差和变异系数对应的函数输入。从结果来看，这其中 gsl_pow_2、gsl_pow_3 和 gsl_sf_bessel_K1 这三个函数比较稳定，有两个函数 gsl_sf_bessel_Y1 和 gsl_sf_bessel_j0 显得非常不稳

表 2: GSL 不稳定函数的进一步算式扰动统计结果

GSL 函数名称	最大极差	最大变异系数	对应函数输入
gsl_pow_2	0	0	any
gsl_pow_3	0	0	any
gsl_sf_bessel_j0	1.26E+117	2.07E+233	-1.48E+26
gsl_sf_bessel_k0_scaled	0	0	any
gsl_sf_bessel_K1	1.42E-14	4.90E-29	8.15E-02
gsl_sf_bessel_K1_scaled	1.42E-14	4.90E-29	9.98E-03
gsl_sf_bessel_y0	1.11E-16	2.75E-33	9.98E-03
gsl_sf_bessel_y1	2.22E-16	8.24E-33	3.67E+01
gsl_sf_hazard	1.11E-16	2.75E-33	3.67E+01
gsl_sf_log_erfc	9.86E-32	8.06E-64	-1.00E+00
gsl_sf_bessel_Y1	4.90E+55	5.83E+110	1.54E-72

定，扰动效果夸张。其它的一些函数虽然有一些扰动效果，但从输出来看还是基本是稳定的，这类对数值扰动不稳定而对算式扰动稳定的用例一般来说需要进一步的手工确认。

针对函数 gsl_sf_bessel_Y1 和 gsl_sf_bessel_j0，我们进一步手动进行了详细的追踪。结果显示，GSL 1.11 版的函数 gsl_sf_bessel_j0 会执行到其源代码 gslroot/specfunc/trig.c 第 205 行的算术表达式 `z * (1.0 + z*z * sin_cs_result.val)`，在某些输入下，例如 -1.48×10^{26}，算术表达式的不同形式会导致结果差异很大。gsl_sf_bessel_Y1 的情形与之类似，在执行路径上会经过 gslroot/specfunc/bessel_Y1.c 第 94 行的算术表达式 `two_over_pi*lnterm*J1.val + (0.5+c.val)/x`，在输入为 4.90×10^{55} 时会不稳定。追踪细节数据见附录A。总体来说，数值扰动和算式扰动的评估结果显示出两种扰动方式能很好的进行相互的补充和确认，来完成数值计算的稳定性检测。

5.3 优化评估

蒙特卡罗优化的评估效果在表 1 中已经得到了体现。在表 1 里我们可以看到，Poly.c 和 sample_run.c 这样算术表达式较复杂的数值计算代码，由于在实数域内的数学等价形式个数过多而根本无法采用原始的算式扰动算法。然而在抽样数限制从 50 到 500 的情况下，统计结果无论是极差还是变异系数的差距都不是很大。这说明蒙特卡罗抽样方法能够有效的估计出总体的统计结果。

我们对学术相关用例测试集采用了 OpenMP 并行算法运行，并对执行时间进行了评估。通过比较串行算法和并行算法的时间差距来检验并行算法的优化效果。图 14 给出了并行算法和串行算法的时间对比，由于各个实验用例的执行时间不在同一个数量级上，为了显示的清晰，我们将纵坐标轴的度量在不同数量级上进行了三个划分，各个用例的所用时间的具体数值也写在了表 3 的总时间里列出。从图 14 来看，Poly.c 和 exp.c 的优化效果十分明显，而 Inv.c 和 sample_run.c 并没有很明显的优化效果。表 3 给出了算法各部分的时间占用状况，正如在 4.1 节所指出的，由于并行算法在线程分配和数据与结果同步方面需要额外的开销，使得各算法部分的运行时间并不总是并行优于串行。算法各部分时间占用的差异和开销的状况最后决定了图 14 里各个用例并行算法的综合效果。

从表 3 可以看出，每个用例在公因式提取阶段的并行优化效果都比较好，因为这一阶段在算法的最后阶段，各个算术表达式等价形式的相关程度较低，同步开销较小。而在交换结合率部分有些用例如 Poly.c 和 exp.c 效果比较明显，这是算术表达式的结构造成的，这两个程序里的算术表达式能较多

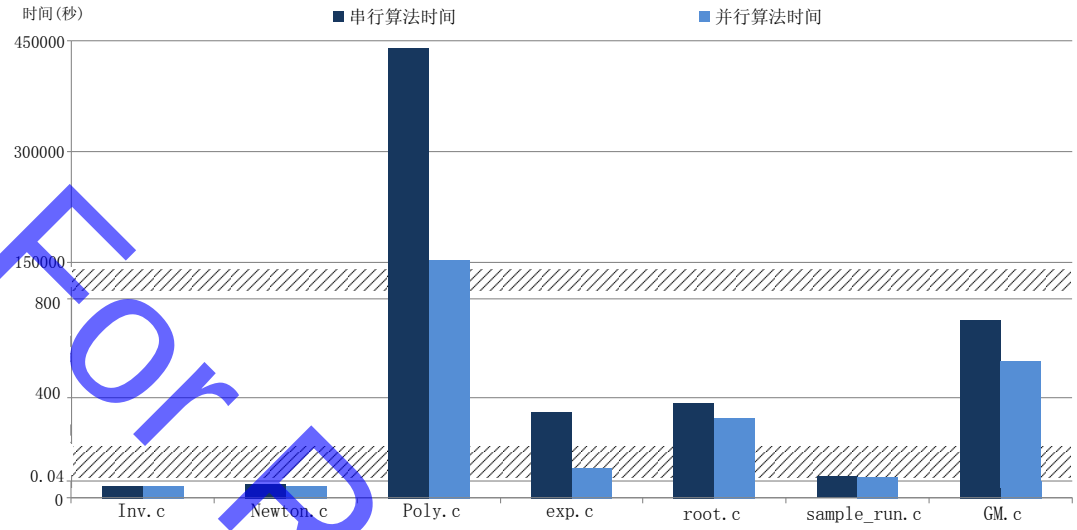

图 14: 串行算法和并行算法时间对比

表 3: 串行算法和并行算法时间分布

	总时间	公因式提取	交换结合律	递归计算
Inv.c	0.027437/0.027142	0.005813/0.007231	0.016757/0.016062	0.004498/0.003433
Newton.c	0.032869/0.029208	0.014582/0.012243	0.013956/0.013746	0.003929/0.002766
Poly.c	440247/153767	439844/153413	377.56/178.431	0.01/0.118719
exp.c	343.534/117.576	217.974/68.4206	125.477/49.0653	0.082257/0.089394
root.c	375.198/317.71	84.0325/25.8341	267.7/268.351	23.4645/23.5243
sample_run.c	81.0119/77.7316	3.80812/0.297882	77.0721/77.2448	0.130891/0.18807
GM.c	712.369/544.906	235.017/82.4336	422.183/410.345	55.1692/52.1274

的分解出相互独立的算式单元来，使并行计算不相互干扰。而另外几个用例算术表达式在转化成规范形式之后，单层的算式单元个数较少，不易充分并行。另外，从表 3 可以看出，递归计算在算式扰动的时间占用中比重较小。

虽然由于并行的开销，使得部分算术表达式的优化效果并不是十分明显，不过，由于较复杂的算术表达式往往会比较容易得到相互独立的算式单元，因而优化效果在执行时间较长的算式扰动用例上会相对稳定。因而在硬件条件允许的条件下，并行优化还是能够较好的达到优化的效果。

5.4　有效性影响因素

我们所采用的分析测试方法并不是完备的证明方法，因而测试用例上可能会遗漏关键输入，也有可能在数据上会存在误报和漏报。虽然漏报问题是软件测试本身所存在的，但软件测试仍然被广泛地认为是有效提高软件质量的实用方法。我们相信我们的技术在分析软件数值计算的稳定性方面是有用的，我们的学术相关用例测试集虽然不大，但具有很好的代表性和典型性。对于测试输入用例，我们在对应函数的定义域内采用手工和随机方法相结合的方式生成，在算式扰动优化方面，我们对算术表达式的实数域等价形式也做了有效地随机选取。不过我们的算式扰动扰动的是独立的算术表达式，

而并未考虑算式之间的组合问题。我们的算式扰动假设程序里的算术表达式并未被慎重考虑后写成最稳定的形式，这在非数值计算的专家所写的程序中是一种很普遍的现象。当这个假设不成立时，一些已经被慎重考虑的算术表达式，可能会因为其实数域的等价形式中有不稳定的形式而被误报。假如发生这种情况，我们认为多一些参考的数据总是好的，这些数据可以更好地帮助开发人员加强对数值程序稳定程度的理解。

6 相关工作

到目前为止，在舍入误差研究和数值计算算法的稳定性分析上已积累了大量相关工作 [5, 25, 26, 27, 28]。这些工作从各自的领域应用出发，探讨保持该领域数值算法稳定性的相关技术。例如，高小山等学者在微分方程和几何约束求解问题上，采用了符号计算的手段来避免误差传播和扩散 [1, 29]；陈发来在图形学求解算法设计中，以区间计算为主要手段来提高算法的稳定性 [2]；而胡事民等学者更进一步探讨了在几何变换领域里的多种误差传播保障途径 [3]。同这些已有的工作不同，本文的方法直接针对数值计算的软件实现，因而不需要使用者掌握具体的应用领域算法知识，更具有通用性。我们进一步调研了和我们的工作最相关的数值分析软件的研究，主要有三个方面工作和我们比较接近：1) 数值软件的程序分析；2) 基于扰动方法的随机算术；3) 数值分析相关的并行计算。

程序分析方法近年来被应用于数值领域，在这里我们将提及一些具有代表性的工作。Goubault[30]提出了一种基于抽象解释的程序分析方法[31]来逼近引入浮点数运算的误差。这项工作后来被 Goubault 和 Putot 使用了一个更精细的抽象域来进一步精化[22]。Martel[32]为浮点数操作提出了一套通用的具体语义来解释计算中的误差前向积累。在更进一步的工作中，Martel 将其具体语义用于通过静态程序分析来检查循环的稳定性[33]，他对一般的程序[8]开发了静态分析方法来优化数值精度[7, 5]，在这个方法里蕴含了和我们相同的观点，即在实数域数学等价的算术表达式在浮点数域不一定等价。我们的方法和他不同之处在于我们对各个不同的替换形式采用了基于统计的分析方法，这种方法更为简单且容易理解。Benz在文献[34]中以动态字长来表示浮点数值，并以该角度来探讨程序的数值稳定性，相对于我们的方法，该项技术需要在软件发布之后进行动态字长的数值计算，因而需要在运行时占用更多的计算资源。本文的作者Barr等进一步给出了一种通用数值稳定性检测的符号计算方法 [35]，该方法对于寻找不稳定的数值测试用例较为敏感，而本文的方法对于定位引起不稳定的算术表达式位置较为敏感。

在随机算术领域，比较有代表性的工作有 CESTAC[36]，它以另外的一个目的对程序做类似于我们数值扰动类型的程序变换。CESTAC 使用多次扰动执行的均值来表示有效数字的数值，而我们的目的在于评估数值程序的健壮性。这项技术目前已经被集成在 CANDA 工具集里[37]。同随机算术比较相似的方法，Parker 等研究人员[9, 38]形式化的引入了蒙特卡罗算术 (Monte Carlo Arithmetic，MCA) 框架，并基于这一框架开发了 wonglediff 工具[6]。这一框架允许随机舍入来模拟计算误差，这与我们的数值扰动类似，与我们工作不同的是，这一方法它不改变数值程序的内容，但它只能改变舍入的模式，因而不支持通用的数值扰动，也不支持算式扰动。我们方法的扰动过程更为通用，从而应用范围更广。

由于高性能计算上的需求，并行计算是一个长期被广泛研究的领域。常用的并行计算模型除了OpenMP[11]外，还有 MPI[39]等其他模型。虽然 MPI 等并行模型的效率较高，但由于其开发花费较大，比较适合分布内存并行程序。OpenMP 不仅开发比较容易，而且能较容易自适应不同的多核环境，甚至能自动的在非并行环境下转化成串行程序，今年来被广泛用于包括多核环境在内的对称多处理器环境。另外，在并行数值计算算法方面[40]，也已经积累了大量的研究。这些工作与我们的不同在于，我们的贡献是分析和检测数值稳定性的并行算法，而不是数值分析本身的并行算法。

7 结论

数值计算程序的正确性很难得到保障，尤其是当未受过深入训练的应用程序开发人员将思维停留在实数的精确计算上时，所写的程序很容易因为不稳定的误差积累而产生错误。在本文中，我们提出一种新的、可用的技术框架来帮助开发人员获得科学计算程序关于数值稳定性和精度方面的高层信息，并对潜在的可能引起的数值计算的错误情况提出警告。这一框架采用了数值扰动和算式扰动两种互补的核心扰动技术。数值扰动通过随机改变有效数字的尾数来发现浮点数的内在误差积累；而算式扰动通过统计性的比较算术表达式在实数域的不同数学等价形式来发现表达式的不稳定。我们在一系列文献中采用的程序和科学计算库上评估了这一框架的有效性。相信这一框架为程序开发人员在开发科学计算应用程序上提供了一项实用的自动化辅助技术。

参考文献

1 Gao X S, Chou S C, Solving geometric constraint systems. II. A symbolic approach and decision of Rc-constructibility, Computer-Aided Design, 1998, 30(2):115–122

2 Chen F, Lou W, Degree reduction of interval Bézier curves, Computer-Aided Design, 2000, 32(10):571–582

3 Hu S, Wallner J, Error propagation through geometric transformations, Journal for Geometry and Graphics, 2004, 8(2):171–183

4 Goldberg D, What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Computing Surveys, 1991, 23(1):5–48

5 Higham N, Accuracy and stability of numerical algorithms, Society for Industrial and Applied Mathematics, 2nd ed., 2002

6 Eggert P R, Parker D S, Perturbing and evaluating numerical programs without recompilation: the wonglediff way, Software Practice and Experience, 2005, 35(4):313–322

7 Martel M, Semantics-Based Transformation of Arithmetic Expressions, in: Static Analysis, 2007, 298–314

8 Martel M, Program transformation for numerical precision, in: Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, 2009, 101–110

9 Parker D S, Eggert P R, Pierce B, Monte Carlo Arithmetic: a framework for the statistical analysis of roundoff error, 1997

10 Sutter H, The Free Lunch is Over : A Fundamental Turn Toward Concurrency in Software, Dr Dobb's Journal, 2005, 30(3)

11 Dagum L, Menon R, OpenMP: an industry standard API for shared-memory programming, Computational Science Engineering, IEEE, 1998, 5(1):46–55

12 Kahan W, The improbability of probabilistic error analyses for numerical computations, 1996, First presented in 1995 in Hamburg at the third ICIAM Congress. `http://www.cs.berkeley.edu/~wkahan/improber.pdf`

13 Knuth D E, Seminumerical Algorithms, vol. 2 of *The Art of Computer Programming*, Reading, Massachusetts: Addison-Wesley, 3rd ed., 1998

14 Adare A, Afanasiev S, Aidala C *et al.*, Double-helicity dependence of jet properties from dihadrons in longitudinally polarized $p + p$ collisions at $\sqrt{s} = 200\ GeV$, Physical Review D, 2010, 81(1):012002

15 Frodesen A G, Skjeggestad O, Tøfte H, Probability and Statistics in Particle Physics, Universitetsforlaget, 1979

16 American National Standards Institute, IEEE standard for binary floating-point arithmetic, 1985

17 Necula G C, McPeak S, Rahul S P *et al.*, CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, in: Proceedings of the 11th International Conference on Compiler Construction, 2002, 213–228

18 Quinlan D, ROSE: Compiler Support for Object-Oriented Frameworks, in: Proceedings of Conference on Parallel Compilers (CPC2000), 2000, 215–226

19 Ayguade E, Copty N, Duran A *et al.*, The Design of OpenMP Tasks, IEEE Transactions on Parallel and Distributed Systems, 2009, 20(3):404

20 Courtois P J, Heymans F, Parnas D L, Concurrent control with "readers" and "writers", Communications of the ACM, 1971, 14:667–668

21 Galassi M, Davies J, Theiler J *et al.*, Gnu Scientific Library Reference Manual, Network Theory Ltd., 1st ed., 2002

22 Goubault E, Putot S, Static Analysis of Numerical Algorithms, in: Proceedings of the 13th International Static Analysis Symposium, 2006, 18–34

23 Jiang L, Su Z, Osprey: a practical type system for validating dimensional unit correctness of C programs, in: ICSE '06: Proceedings of

the 28th international conference on Software engineering, 2006, 262–271
24 Lin K H, Liu B D, A gray system modeling approach to the prediction of calibration intervals, IEEE Transactions on Instrumentation and Measurement, 2005, 54(1):297–304
25 Miller W, Toward mechanical verification of properties of roundoff error propagation, in: Proceedings of the ACM Symposium on Theory of Computing, 1973, 50–58
26 Miller W, Software for Roundoff Analysis, ACM Transactions on Mathematical Software, 1975, 1(2):108–128
27 Miller W, Spooner D, Software for roundoff analysis, II, ACM Transactions on Mathematical Software, 1978, 4(4):369–387
28 Wilkinson J H, Rounding Errors in Algebraic Processes, Dover Publications, Incorporated, 1994
29 Gao X S, Luo Y, Yuan C, A characteristic set method for ordinary difference polynomial systems, Journal of Symbolic Computation, 2009, 44(3):242–260
30 Goubault E, Static Analyses of the Precision of Floating-Point Operations, in: Proceedings of the 8th International Static Analysis Symposium, 2001, 234–259
31 Cousot P, Cousot R, Abstract Interpretation: A unified Lattice model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, 1977, 238–252
32 Martel M, Propagation of roundoff errors in finite precision computations: a semantics approach, in: 11th European Symposium on Programming, Le Métayer D, ed., Springer Berlin / Heidelberg, 2002, 194–208
33 Martel M, Static Analysis of the Numerical Stability of Loops, in: Proceedings of the 9th International Static Analysis Symposium, Hermenegildo M, Puebla G, eds., Springer Berlin / Heidelberg, 2002, 133–150
34 Benz F, Hildebrandt A, Hack S, A dynamic program analysis to find floating-point accuracy problems, SIGPLAN Not, 2012, 47(6):453 – 462
35 Barr E T, Vo T, Le V *et al.*, Automatic detection of floating-point exceptions, in: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA: ACM, 2013, POPL '13, 549 – 560
36 Brunet M C, Chatelin F, CESTAC, a tool for a stochastic round-off error analysis in scientific computing, Numerical Mathematics and Applications, 1986, 11–20
37 Vignes J, A stochastic arithmetic for reliable scientific computation, Mathematics and Computers in Simulation, 1993, 35(3):233–261
38 Parker D, Pierce B, Eggert P, Monte Carlo arithmetic: how to gamble with floating point and win, Computing in Science & Engineering, 2000, 2(4):58–68
39 Gropp W, Lusk E, Doss N *et al.*, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 1996, 22(6):789–828
40 Freeman T, Phillips C, Parallel numerical algorithms, Prentice-Hall, Inc., 1992