北京大学

博士研究生学位论文

题目：基于搜索的浮点数程序误差测试与分析方法研究

姓　　名：邹达明

学　　号：1701111333

院　　系：信息科学技术学院学院

专　　业：计算机软件与理论

研究方向：软件分析与测试

导　　师：张路 教授

协助指导：熊英飞 副教授

二〇二〇年五月

版权声明

摘要

浮点数是编程语言中必不可少的数据类型，尤其大量应用于数值运算、科学计算类型的程序中，因而被广泛应用于科学、工程、金融等领域。浮点数误差问题是软件工程与编程语言领域的重要问题。在浮点数程序中，结果必然存在误差，而这种误差可能会导致严重的软件错误。为了避免这类严重的软件错误，安全攸关软件对于浮点数误差的上限有着严格的要求。因此，浮点数程序的测试和分析技术是保障软件安全的关键技术，具有十分重要的研究意义。

由于浮点数特殊的数据结构特点与应用场景，传统的程序测试及分析方法并不能直接应用于浮点数程序的测试与分析中。具体而言，浮点数误差问题具有下述特点：

- **未知性**：误差无任何警告或错误信息，精确的测试预言（oracle）获取困难；
- **稀疏性**：显著误差通常仅会被极少数输入触发，不易发现。

已有的浮点数相关研究往往从其中一个角度入手，在一些假设下解决某一个问题，例如针对浮点数异常（exception）的检测工作，针对有警告情形下的浮点数溢出、非法运算等问题，无法解决无警告的浮点数误差问题（即未知性问题）；误差上界分析，通常基于静态分析，尝试分析浮点数程序的误差上界，但无法给出具体会触发大误差的输入（即稀疏性问题）。

为了保证浮点数程序的质量，从程序测试和程序分析的角度，尽可能的对浮点数误差进行检测，本文使用了**基于搜索的方法**对浮点数程序误差进行测试与分析，核心目标是找到会触发显著误差的测试用例。基于搜索的软件工程使用启发式搜索的方法解决软件工程问题，其核心包括问题的定义与转化，搜索目标函数的制定，搜索算法的构建等一系列关键技术。针对浮点数误差问题，基于搜索的方法具有下列特点：（1）是一种动态分析方法；（2）可以给出特定的触发显著误差的测试用例；（3）可以参考及利用最优化问题的搜索算法；（4）可以应用于不同的测试与分析场景下。在具有上述特点的同时，基于搜索的方法也面临未知性与稀疏性问题的挑战：

- 搜索目标函数的制定与获取，受到测试预言*未知性*问题的挑战；
- 搜索算法的效果，受到浮点数误差*稀疏性*问题的挑战。

针对基于搜索的方法面临的上述挑战，本文进行了一系列具有针对性的研究，分别为：（1）高质量的浮点数测试预言获取方法研究，以建立广泛适用的搜索目标函数；（2）黑盒场景下的搜索算法研究，以实现特定于浮点数误差问题的高效搜索；（3）白盒场景下的新搜索目标函数研究，以提升白盒场景下的搜索效率。

本文的主要研究工作及创新点如下：

1. 提出了**面向搜索的浮点数测试预言自动生成技术**，该研究解决了语义解释错误导致的测试预言错误问题。在基于搜索的软件工程中，目标函数的制定是解决搜索问题的关键之一。由于浮点语义运算的存在，基于实数运算语义的测试预言生成工具会产生语义解释错误，进而生成错误的测试预言。该工作提出了兼容语义模型，解决了语义解释错误问题，以及提出了基于该模型的语义检测和精度调整方法，以得到高质量的测试预言，作为搜索目标函数，解决未知性问题。在 GNU C 标准库的实验中，该工作的准确率和召回率分别达到了 77.05% 和 97.92%，生成的测试预言精确可靠。
2. 提出了**基于搜索的浮点数黑盒测试用例生成技术**，该研究提出了一种针对浮点数误差问题的搜索算法，提升了搜索的效果。在基于搜索的软件工程中，搜索算法的有效性是另一个关键难点。由于浮点数误差的稀疏性，直接使用传统的启发式算法搜索效率不高，往往无法完成预期的搜索目标。该工作针对浮点数的数据结构特点以及误差的分布进行分析，提出一种特定于浮点数误差问题的搜索算法，可以高效地生成测试用例，检测浮点数误差问题。该工作深入分析了浮点数中各组成结构对误差产生影响的特性，在搜索过程中使用分层次，分阶段的算法，充分利用了浮点数的组成结构对误差的影响特点，显著提高了搜索算法的有效性。在 GNU 科学计算库的实验中，该工作的检测效率显著优于已有的误差检测技术，并发现了 30 个存在精度问题的 GSL 函数。
3. 提出了**基于搜索的浮点数白盒分析技术**，该研究提出了一种新的搜索目标，提升了误差分析的运行效率与搜索效果。已有工作普遍使用误差作为搜索目标，在测试过程中需要大量计算测试预言，导致运算开销庞大。该工作提出了**原子状态函数**这一概念，用于分析浮点数程序运算过程中误差的引入、传播、及放大过程，构建误差积累模型，降低分析工作对开销昂贵的测试预言的依赖。该工作以原子状态函数作为新的搜索目标函数，开销昂贵的测试预言只用于找到搜索目标后的确认，从而显著降低了分析过程的运算开销；同时，作为白盒分析工具，给搜索算法提供了更多的信息，提高了搜索效率。在 GNU 科学计算库的实验中，相比于已有的技术，该工作的运行效率提高了 1000 倍以上，同时误差检测效果提升了 40%。

本文提出的测试与分析技术高度相关且在不同场景下形成互补：测试预言生成是浮点数程序测试与分析的基石技术；黑盒测试不依赖于源码，可应用于源代码不可见的场景下；白盒分析技术需要分析代码，速度较快，可用于源代码可见的场景下。上述三种方法可以相互配合，最终形成一套基于搜索的浮点数程序误差测试与分析方法。

关键词：浮点数，误差，程序分析，程序测试

Search-Based Methods for Testing and Analyzing Floating-Point Errors

Daming Zou (Computer Software and Theory)
Directed by Prof. Lu Zhang

ABSTRACT

Floating-point is an essential data type in programming languages, and is especially used in numerical analysis and scientific programs. Therefore, floating-point is widely used in the domain of science, engineering, finance, etc. The floating-point error problem is an important issue in the field of software engineering and programming languages. In floating-point programs, there are always errors(also called inaccuracies) within the computation results, and such errors may lead to critical software failures. In safety-critical software, there are strict requirements on the upper bound of floating-point error. Therefore, the testing and analysis techniques of floating-point program are the core techniques to guarantee the safety of software, and have an important research significance.

Due to the special data structure of floating-point numbers and special application scenarios, traditional program testing and analysis method cannot be applied directly to on floating-point programs. Specifically, the floating-point error problem has the following issues:

- **Unawareness and Unknowability**. The floating-point error comes silently, there are no warning/error messages on floating-point errors. It is also difficult to measure the floating-point error. One must know precise calculation result (the *oracle*) before computing the error. However, the oracle result is usually unavailable.
- **Sparseness**. The significant floating-point errors are usually triggered by only a few inputs, and it is not easy to detect such errors.

The existing related research often starts from one the above points, and solves a specific problem under some assumptions. For example, some works focus on detecting floating-point exceptions. They can handle the explicit exceptions with warning messages, but cannot handle the implicit floating-point errors, i.e., it is limited by the unawareness. Another example is error bound analysis. These works are based on static analysis, and try to measure the maximum possible errors. However, error bound analysis cannot report the error trigger input, i.e., it is

limited by the sparseness.

To ensure the quality of floating-point programs, this paper focuses on detecting floating-point errors from program testing and analysis perspective. This paper proposed search-based methods for testing and analyzing floating-point programs. Search-based software engineering uses heuristic search method to solve software engineering problems, the core of which includes a series of key techniques such as problem definition and transformation, search objective function, and search algorithm construction. For floating-point error problem, the search-based approach has the following features: (1) it is a dynamic approach; (2) it can report specific error triggering inputs; (3) it can refer and utilize the optimization algorithms; and (4) it can be applied to different testing and analysis scenarios. Along with the aforementioned features, the search-based approach also faces the challenges of unknowability and sparseness:

- The definition and computation of the search objective function is challenged by the unawareness and unknowability.
- The effectiveness of the search algorithm is challenged by the sparseness.

This paper conducts a series of highly related approaches aiming to solve the aforementioned challenges, including: (1) research on obtaining the oracle of floating-point programs, which can be applied as the objective function; (2) research on a new search algorithm under black-box scenarios; and (3) research on a new objective function under white-box scenarios.

The main contributions of this paper are as following:

- A search-oriented approach to generate test oracles on floating-point programs. This approach tackles the wrong calculation of oracles caused by misinterpreted semantics. In search-based software engineering, objective function is one of the core techniques. In this work, we found that the semantics of floating-point operations could be misinterpreted by real number semantics, and such misinterpretation leads to wrong test oracles. This work proposed an compatible semantics model for both real numbers and floating-point numbers, and also proposed the detection algorithm and tunning algorithm based on this semantics model. The detection algorithm and tunning algorithm are designed to calculate the oracle of the floating-point programs. In the evaluation on the GNU C Library, this approach achieves 77.05% precision and 97.92% recall, and the generated oracles are accurate with the relative error less than 10^{-16}.
- A search-based approach to generate test cases for floating-point error in black-box manner. This approach proposed a novel algorithm which increases the effectiveness of search. In search-based software engineering, search algorithm is another core

techniques. Due to the sparseness of floating-point errors, it is hard to apply ordinary search algorithm to detect floating-point errors. This work conducts a empirical study to analyze how different floating-point structures affect program error differently, and proposed a novel particle swarm algorithm based on the insights from the empirical study. In the evaluation on the GNU Scientific Library(GSL), this approach significantly outperforms than the state-of-the-art techniques, and finds that 23 GSL functions have floating-point error defects.

- A search-based approach to analyze floating-point error in white-box manner. This work proposes a novel objective function, 'atomic condition', for analyzing the error propagation in floating-point programs. Comparing with using 'error' as objective function directly, atomic condition can significantly reduce the runtime cost, and provide more information to guide the search procedure, thus leads to better detection results. In the evaluation on the GSL, this approach significantly outperforms the state-of-the-art techniques, and is 1000x faster.

This approaches proposed in this paper are highly connected and complementary. The test oracle is the basis of floating-point error testing and analysis. The black-box approach applies when source code is unavailable. and the white-box approach applies when source code is available. These approaches are integreted to become a suite of search-based methods for testing and analyzing floating-point programs.

KEYWORDS: Floating-point, Inaccuracy, Program Analysis, Software Testing

目录

第一章 引言 **1**
1.1 问题的提出 1
1.1.1 浮点数误差问题 1
1.1.2 浮点数程序测试与分析技术的挑战 2
1.2 浮点数背景介绍 4
1.2.1 浮点数的结构 4
1.2.2 浮点数的舍入 5
1.2.3 浮点数的精度与误差 5
1.3 相关研究现状 7
1.3.1 针对传统软件工程问题的浮点数测试与分析工作 7
1.3.2 针对误差问题的浮点数测试与分析工作 10
1.3.3 研究现状小结 19
1.4 尚待解决的问题 20
1.5 本文主要工作 21
1.5.1 面向搜索的浮点数测试预言自动生成技术 22
1.5.2 基于搜索的浮点数黑盒测试用例生成技术 24
1.5.3 基于搜索的浮点数白盒分析技术 25
1.6 论文组织 26
第二章 面向搜索的浮点数测试预言自动生成技术 **27**
2.1 引言 27
2.2 实数运算语义与兼容语义模型 29
2.2.1 实数运算语义 29
2.2.2 浮点运算语义导致的解释错误 30
2.2.3 兼容语义模型 32
2.3 基于兼容语义模型的精度调整工具 PSOHUNTER 33
2.3.1 技术概要 33
2.3.2 精度调整 34
2.3.3 兼容语义模型下的语义检测 35
2.3.4 兼容语义模型下的精度调整 37
2.4 工具实现 39

2.5 实验评估 39
2.5.1 实验设计 39
2.5.2 实验结果 41
2.6 讨论与小结 45

第三章 基于搜索的浮点数黑盒测试用例生成技术 47

3.1 引言 47
3.2 黑盒测试场景及特点 49
3.3 浮点数结构及对误差的影响 49
3.3.1 浮点数结构特点 50
3.3.2 样例分析 50
3.3.3 浮点数结构对运算误差的影响 52
3.4 层级混合粒子群算法 HierHybrid 与测试用例生成 52
3.4.1 技术概要 52
3.4.2 粒子群算法概述 53
3.4.3 实证研究 55
3.4.4 针对浮点数误差问题的粒子群算法 HierHybrid 59
3.4.5 应用 HierHybrid 进行测试用例生成 64
3.5 实验评估 65
3.5.1 实验设计 65
3.5.2 实验结果 66
3.6 讨论与小结 69

第四章 基于搜索的浮点数白盒分析技术 71

4.1 引言 71
4.2 白盒分析场景及特点 73
4.3 浮点数原子运算与状态函数 74
4.3.1 浮点数原子运算及其误差 74
4.3.2 状态函数 75
4.4 基于原子状态函数的误差分析 76
4.4.1 技术概要 76
4.4.2 样例分析 77
4.4.3 原子状态函数与误差累积模型 79
4.4.4 基于原子状态函数的搜索过程 82
4.4.5 搜索结果排序 84

4.5 工具实现 85
4.6 实验评估 86
4.6.1 实验设计 86
4.6.2 实验结果 87
4.6.3 实例分析 93
4.7 讨论与小结 94

第五章 结论和展望 **97**
5.1 本文工作总结 97
5.2 未来工作展望 98

参考文献 **101**

个人简历及博士期间研究成果 **107**

致谢 **109**

北京大学学位论文原创性声明和使用授权说明 **111**

第一章　引言

浮点数在计算机系统中无处不在。几乎所有的编程语言中都包含浮点数数据类型；从个人电脑到超级计算机都包含浮点运算单元（加速器）；编译器的一项重要工作就是编译浮点运算至特定的硬件指令；而几乎所有的操作系统都必须处理浮点数的异常信息（例如溢出） [29]。

浮点数是计算机系统中必不可少的数据类型，尤其应用于数值运算、科学计算类型的程序中 [53]，因而被广泛应用于航空航天、医疗设备、交通、核能、金融等安全攸关领域。浮点数是所有数值运算程序的核心构件之一，因而也在上述安全攸关软件中被大量使用。由于浮点数的重要性和基础性，浮点数误差可能导致严重的、甚至灾难性的后果。因此，浮点数误差问题一直是软件工程与编程语言领域的重要问题 [77, 80]。

在保证和提高浮点数程序质量的研究中，浮点数程序测试和分析技术是最主要的方式。本文从基于搜索的角度进行了浮点数程序误差测试与分析的若干技术研究。

本章首先提出本文工作研究的问题，介绍浮点数相关背景，然后讨论该问题的相关研究现状以及尚待解决的问题，介绍本文的主要工作和创新点，最后展示全文的结构。

1.1　问题的提出

1.1.1　浮点数误差问题

在计算机科学中，浮点数（floating-point number）是一个属于有理数的特定子集，用于**近似**表示任意一个实数。浮点数由符号位、指数位（幂数）、和尾数位（有效数字）组成，其表达的值由有效数字乘以幂数得到，类似于科学计数法。由于浮点数具有固定长度的有效数字，浮点数在表达极大或极小的实数时都具有相似的**相对误差**。例如，对于 64 位的双精度浮点数（double precision）而言，在其表达范围内，任意实数与最接近的双精度浮点数之间的相对误差都不大于 2.2×10^{-16}。这一特性使得浮点数被广泛应用于各类数值运算及科学运算程序中，也事实上成为最广泛使用的数值运算类型。

浮点数运算是使用浮点数**近似**表示实数的运算方法。由于浮点数运算使用有限的精度来模拟无限精度的实数运算，其结果必然携带误差 [34]。由于浮点数程序通常由大量的浮点数运算组成，浮点数误差也会在不同的运算中产生并累积，并且在特定条件下产生具有显著误差的运算结果。保证浮点数运算的准确性是一项充满挑战的任务，

尽管经过数十年的努力，IEEE 754 标准 [81] 及 GNU C 基础库 [45] 已经将浮点数基础运算的单次运算误差控制在非常小的范围，但由于运算之间误差的累积及放大现象，浮点数的显著误差问题一直存在。

浮点数程序的显著误差会造成巨大的经济损失甚至灾难性的后果。例如：

- 温哥华证券交易所由于浮点数显著误差，股指出现混乱 [58]；
- 阿丽亚娜-5 运载火箭发射时，由于浮点数显著误差导致处理器运算溢出，火箭在发射 37 秒后引爆自毁 [44]；
- 在海湾战争中，一枚爱国者导弹由于浮点数显著误差偏航，导致了 28 人死亡 [65]。

浮点数误差问题不只存在于传统的数值运算领域。由于其基础性及重要性，现代系统及软件同样受到浮点数误差问题的影响，例如概率编程语言系统 [21] 及深度学习系统 [56]。由于现代系统中软件的复杂性进一步增加，浮点数误差问题变得更加重要，更有挑战性。

1.1.2 浮点数程序测试与分析技术的挑战

在产生巨大危害的同时，浮点数误差问题又是非常难以发现及解决的。本节将从手动和自动化两方面介绍浮点数程序测试与分析技术的挑战。

手动测试与分析的挑战

由于浮点数运算与实数运算的近似性，在日常开发实践中，开发者通常将浮点数作为实数对待，而无法意识到浮点数带来的误差问题。一方面，浮点数误差始终存在，通常情况下，较小的浮点数误差是可以被接受的，并不会引起严重的后果，因此开发者不会尝试——事实上也无法做到——消除这些不显著的浮点数误差；另一方面，由于显著的浮点数误差只在极少的情况下出现，且会触发该误差的输入并不规律，导致开发者无法通过测试少量的边界用例（corner case）来发现误差。因此手动测试浮点数误差问题是非常困难的。

由于现代软件中使用的 32 位或 64 位浮点数的精度早已超出了手动计算的极限，开发者通常只能依赖计算机给出的结果，无法手动判断结果准确与否。准确结果（测试预言）的缺失使得手动分析浮点数误差问题同样极具挑战。

手动测试与分析浮点数误差问题显然无法满足现代软件质量保证的需求，因此自动化的测试与分析工具是解决浮点数误差问题的主流方向。

自动化测试与分析技术的挑战

由于浮点数特殊的数据结构及特殊的运算特性，已有的自动化工具不能很好的解决浮点数程序的测试与分析问题。浮点数使用有限精度模拟无限精度的实数运算。由

于浮点数的精度有限，浮点数运算大量使用舍入操作（rounding），使其不具备一些基础的数学性质。例如，浮点数运算不满足结合律

$$(a+b)+c=a+(b+c)\quad a,b,c\in\mathbb{R}$$

$$(a+b)+c\neq a+(b+c)\quad a,b,c\in\mathbb{F}$$

其中，$\mathbb{R}$ 表示实数，$\mathbb{F}$ 表示浮点数。以下程序样例可以说明这一问题：

```
In [1]: a = 0.1+(0.2+0.3)

In [2]: b = (0.1+0.2)+0.3

In [3]: a == b
Out[3]: False

In [4]: a
Out[4]: 0.6

In [5]: b
Out[5]: 0.6000000000000001
```

由于缺乏基础的数学性质，很多传统的程序分析工具无法直接应用于浮点数问题。例如，符号执行（symbolic execution）是近年来非常流行的一类分析工具，可用于不同领域下的测试用例生成以及缺陷检测问题 [12, 28, 63, 70]。所有符号执行工具的基础与核心都是约束求解器（SMT solver），约束求解器承担了确认路径约束是否可解，以及正确性条件是否可能违背等任务。由于约束求解器通常基于数学运算与数学规则，而无法理解带有误差的浮点数运算，大多数符号执行工具无法直接支持浮点数操作，只能使用近似值替代，甚至拒绝提供支持 [12]。

具体而言，相比于传统程序分析问题，浮点数误差问题具有以下特点，给针对浮点数程序误差的测试及分析工具带来了困难与挑战：

- **未知性**。浮点数误差通常是未知的：即使出现了极其严重的误差使得计算结果无效，浮点数程序仍然会产生结果，并且没有任何（警告或错误）提示 [60]。因此，传统的分析工具严重依赖于测试预言（oracle），以此来判断浮点数程序的误差是否显著。然而，相关研究指出 [73]，即使最先进的测试预言生成工具的质量仍然不高，在一些情况下会生成错误的测试预言，给浮点数误差的分析带来了困难。一些传统工作往往需要通过手动分析来验证分析结果的正确性，或只能应用于测试预言已知的程序，无法扩展应用至其他程序，严重制约了技术的应用范围。近年来，自动化测试预言生成工具的出现 [10] 试图解决未知性问题。但其无法正确的对程序语义进行解释，导致测试预言质量不高，甚至会产生错

误的运算结果。处理上述问题需要大量的专业知识（expertise），而目前尚未有自动化的工具出现，给分析工作带来了困难；

- **稀疏性**。相关工作表明 [8]，只有极少数输入（测试用例）会触发显著的浮点数误差。在上述工作的实验中，只有 0.000004%（亿分之四）的输入触发了显著的浮点数误差。在现代系统中，64 位的浮点数（double）是最常用的浮点数类型，即使待分析的程序只有一个浮点数参数，测试空间也有 2^{64}（$\approx 10^{19}$，千亿亿级别）个输入，而更多的参数会进一步使得测试空间呈指数型增长。在如此庞大的测试空间中，仅有极少数测试用例会触发显著的浮点数误差，给浮点数误差的测试及分析工作带来了很大的困难。

1.2 浮点数背景介绍

1.2.1 浮点数的结构

浮点数是一个属于有理数的特定子集，用于近似表示任意的某个实数。浮点数由符号位、指数位（幂数）、与尾数位（有效数字）三部分组成。其表达的值由有效数字乘以幂数得到，类似于科学记数法。浮点数运算是指使用浮点数进行模拟实数的运算。由于浮点数无法精确表示实数，浮点数运算通常伴随着舍入导致的误差。

IEEE 754 标准 [81] 中定义了浮点数的结构，以及浮点数中各组成部分的位数。表 1.1 中展示了浮点数各个部分在半、单、双精度浮点数中的位数。

表 1.1 IEEE 754 浮点数表示

	符号位	指数位	尾数位
半精度浮点数（16 位）	1	5	10
单精度浮点数（32 位）	1	8	23
双精度浮点数（64 位）	1	11	52

浮点数的值按照如下规则表达：

- 特殊值，包括非数（*NaN*），正无穷（$+\infty$），及负无穷（$-\infty$）。如果浮点数中所有指数位的位数都为 1，则浮点数表示上述三种特殊值中的其中一个：
  - 正无穷对应的符号位为 0，尾数位全部为 0；
  - 负无穷对应的符号位为 1，尾数位全部为 0；
  - 非数对应的尾数位不为 0。非数表示在运算中出现错误，例如除零错误等。

- 数值。如果浮点数的指数位不全为 1，则浮点数的值可以通过下述公式 1.1 计算

$$(-1)^S \times T \times 2^E \tag{1.1}$$

其中：

  - S 表示符号位的值。0 对应正数，1 对应负数；
  - E 表示指数位的值。将指数位的位结构表示为 $b_0 b_1 \ldots b_{m-1}$，则指数位的值可以通过下述公式 1.2 计算：

$$E = \sum_{i=0}^{m-1} 2^i b_i - 2^{(m-1)} + 1 \tag{1.2}$$

  - T 表示尾数位的值。将尾数位的位结构表示为 $d_0 d_1 \ldots d_{n-1}$，则尾数位的值可以通过下述公式 1.3 计算：

$$T = \begin{cases} \sum_{i=0}^{n-1} \frac{d_i}{2^{i+1}}, & \text{如果指数位中所有位数都为 0} \\ 1 + \sum_{i=0}^{n-1} \frac{d_i}{2^{i+1}}, & \text{其他情况} \end{cases} \tag{1.3}$$

1.2.2　浮点数的舍入

由于浮点数的运算结果通常无法被浮点数精确表达，例如 $\sqrt{0.5} \approx 0.707106781\ldots$，在每一次浮点数运算结束后，都需要进行舍入操作。具体而言，浮点数的舍入，是将浮点数运算的结果近似到相近浮点数的操作。浮点数的舍入有多种方法，IEEE 754 标准中列出了 4 种最广泛使用的舍入方法：

- 舍入到最接近的值，偶数优先（round to nearest, ties to even）：将结果舍入到最接近的浮点数的值，当存在两个一样接近的值时，则取其中的偶数，即在二进制中以 0 结尾的浮点数。该方法是 IEEE 754 中的默认舍入方法；
- 向正无穷方向舍入（也称为向上舍入）：将结果朝正无穷的方向舍入；
- 向负无穷方向舍入（也称为向下舍入）：将结果朝负无穷的方向舍入；
- 向零舍入（也称为截断）：将结果朝零的方向舍入，即直接将有效数字以后的部分丢弃。

由于浮点数运算通常无法被浮点数精确表达，在浮点数程序中，舍入操作不可避免。舍入操作会引入浮点数误差，这也是浮点数程序存在误差的最根本原因。

1.2.3　浮点数的精度与误差

在浮点数表达实数时，舍入操作导致的误差可以定义为实数值 x 与浮点数值 $\hat{x}$ 之间的差值：$x = \hat{x} + \eta$，其中 η 即为浮点数表达实数的误差。

对于浮点数程序 $\mathbf{P} : \hat{y} = \hat{f}(\mathbf{x}), \hat{y} \in \mathbb{F}$，在程序的每一次浮点运算中，都会引入误差以

及积累误差。运算过程中误差的不断积累会导致最终结果携带误差。相对误差（relative error，Err_{rel}）和绝对误差（absolute error，Err_{abs}）是两种广泛使用的衡量误差大小的指标。对于理想的精确结果（即实数运算结果）$f(\mathbf{x})$ 和浮点数程序运算结果 $\hat{f}(\mathbf{x})$，相对误差和绝对误差的计算公式如下：

$$Err_{rel}(f(\mathbf{x}), \hat{f}(\mathbf{x})) = \left| \frac{f(\mathbf{x}) - \hat{f}(\mathbf{x})}{f(\mathbf{x})} \right|$$

$$Err_{abs}(f(\mathbf{x}), \hat{f}(\mathbf{x})) = \left| f(\mathbf{x}) - \hat{f}(\mathbf{x}) \right|$$

最小精度单位（unit in the last place，ULP）是一种衡量相对误差的单位 [45, 81]。对于一个浮点数表达及其对应的实数值 $z = (-1)^S \times d_0 d_1 \ldots d_n \times 2^E$，最小精度单位和 ULP 误差（$Err_{ulp}$）可以表示为：

$$ULP(z) = \frac{\left| d_0.d_1 \ldots d_n - (z/2^E) \right|}{2^{n-1}}$$

$$Err_{ulp}(f(\mathbf{x}), \hat{f}(\mathbf{x})) = \left| \frac{f(\mathbf{x}) - \hat{f}(\mathbf{x}))}{ULP(f(\mathbf{x}))} \right|$$

对于单精度浮点数（32 位），其尾数位有 23 位，同时包含始终为 1 的隐藏首位，共有 24 位有效数字。因此单精度浮点数的一个 ULP 误差大约对应 6.0×10^{-8} 至 1.2×10^{-7} 之间的相对误差：

$$\frac{1}{2^{24}} \approx 6.0 \times 10^{-8}$$
$$\frac{1}{2^{23}} \approx 1.2 \times 10^{-7}$$

对于双精度浮点数（64 位），其尾数位有 52 位，同时包含始终为 1 的隐藏首位，共有 53 位有效数字。因此双精度浮点数的一个 ULP 误差大约对应 1.1×10^{-16} 至 2.2×10^{-16} 之间的相对误差：

$$\frac{1}{2^{53}} \approx 1.1 \times 10^{-16}$$
$$\frac{1}{2^{52}} \approx 2.2 \times 10^{-16}$$

以下的 C 程序，概略的展示了单精度与双精度浮点数的精度。

```c
#include <stdio.h>
int main() {
    float  f = 12.34567890123456789;
    double d = 12.34567890123456789;
    printf("float:  %.20e\n", f);
    printf("double: %.20e\n", d);
    return 0;
}
```

其输出结果为：

```
1 float:  1.23456792831420898438e+01
2 double: 1.23456789012345673484e+01
```

1.3　相关研究现状

本节首先对浮点数程序的测试与分析领域的相关研究现状进行介绍，然后对现有相关研究进行总结陈述。

- 本节的第 1.3.1 小节，将介绍针对传统软件工程问题的浮点数程序测试与分析相关工作。本小节将介绍两类工作，分别为代码覆盖的相关工作，其目标是生成测试用例，使其覆盖尽可能多的运行分支；以及异常检测的相关工作，其目标是检测浮点数程序中的错误（error）及异常（exception）运行状态。上述两类目标都是软件工程领域的传统测试与分析目标 [32, 71]；
- 本节的第 1.3.2 小节，将介绍针对误差问题的浮点数程序测试与分析的相关工作。由于浮点数误差的特殊性，传统的软件工程方法无法简单的套用在浮点数误差问题上。本小节将介绍两类工作，分别为误差上界分析的相关工作，其目标是分析浮点数程序可能的误差上界，该方法基于静态分析，不会生成具体的测试用例；以及基于搜索的误差检测相关工作，其目标是使用搜索的方法检测浮点数误差。自 2014 年以来，已有部分工作使用基于搜索的方法对浮点数误差问题进行初步探索 [15, 26]。这部分工作直接使用现有的搜索算法与框架，未针对浮点数误差问题的特性进行深入研究，导致其普遍对误差的检测效率不高。

1.3.1　针对传统软件工程问题的浮点数测试与分析工作

本小节将介绍针对传统软件工程问题的浮点数测试与分析相关工作。包括：

- 代码覆盖。对于浮点数程序，尽可能多的进行分支覆盖 [27]。
- 异常检测。对于浮点数程序，生成可以触发错误及异常的测试用例 [9]。

1.3.1.1　针对代码覆盖的相关工作

代码覆盖（code coverage）是软件测试中用于度量测试用例质量的一个重要指标，最早由 Miller 及 Maloney 于 1963 年提出 [48]。近年来已有大量相关研究在此领域展开 [16, 76]。其中使用符号执行（symbolic execution）的相关技术是目前的主流技术方向。

符号执行是一类用于提高代码覆盖的技术。过去 40 年中，符号执行的研究热度一直很高 [7, 14, 38]。它可以通过分析程序来得到让特定代码区域执行的输入。顾名思义，

```
1  int twice(int v) {
2    return 2*v;
3  }
4  void testme(int x, int y) {
5    int z = twice(y);
6    if (z == y) {
7      if (x > y+10) {
8        ERROR;
9      }
10   }
11 }
12 /* simple driver exercising
      testme() with sym inputs */
13 int main() {
14   x = sym_input();
15   y = sym_input();
16   testme(x, y);
17   return 0;
18 }
```

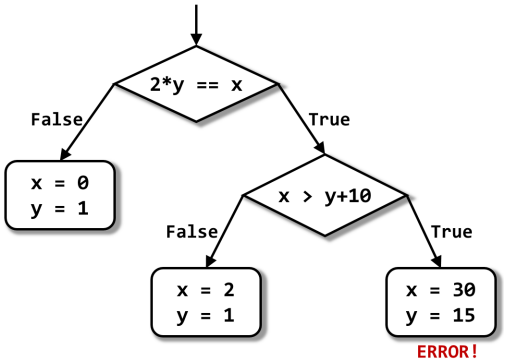


图 1.1　符号执行与执行树表达

使用符号执行分析一个程序时，该程序会使用符号值作为输入，而非一般执行程序时使用的具体值。在达到目标代码时，分析器可以得到相应的路径约束，然后通过约束求解器（constraint solver）来得到可以触发目标代码的具体输入（测试用例）。

软件测试中的符号执行主要目标是：在给定的条件中，探索尽可能多的、不同的程序路径。对于每一条程序路径：

1. 生成一个具体输入的集合（主要目标）；
2. 检查是否存在各种错误（包括错误、异常、违反断言、内存泄漏等）。

从测试生成的角度，符号执行可以生成高覆盖率的测试用例。从缺陷检测的角度，符号执行可以提供一个具体的输入用于触发缺陷，提供给开发者进行调试使用。

符号执行的关键思想就是，把输入变为符号值，那么程序计算的输出值就是一个符号输入值的函数。一个程序执行的路径通常是 `True` 和 `False` 条件的序列，这些条件是在分支语句处产生的。在序列的 i^{th} 位置如果值是 `True`，那么意味着 i^{th} 条件语句进入的是 `then` 分支；反之如果是 `False` 就意味着程序实际执行进入的是 `else` 分支。因此，符号执行通过执行树形式化的表达执行过程。

在图 1.1 的代码中，`testme()` 函数有 3 条执行路径，组成了右图中的执行树。直观上来看，我们只要给出三个输入就可以遍历这三个路径，即图中不同的 x 和 y 取值。符号执行的目标就是能够生成这样的输入集合，在给定的时间内探索所有的路径。

为了形式化地完成这一任务，符号执行会在全局维护两个变量。其一是符号状态

σ，它表示的是一个从变量到符号表达式的映射。其二是符号化路径约束 PC（path constraint），这是一个无量词的一阶公式，用来表示路径条件。在符号执行的开始，符号状态 σ 会先初始化为一个空的映射，而 PC 初始化为 `True` 。σ 和 PC 在符号执行的过程中会不断更新。在符号执行结束时，路径约束就会用约束求解器进行求解，以生成实际的输入值。这个实际的输入值如果用程序执行，就会走符号执行过程中探索的那条路径，即此时路径约束的公式所表示的路径。

以图 1.1 中的例子来阐述这个过程，即当符号执行开始时，符号状态 σ 为空，符号路径约束为 `True` 。当我们遇到一个读语句，形式为 `var=sym_input()` ，即接收程序输入，符号执行就会在符号状态 σ 中加入一个映射 $var \rightarrow s$，这里 s 就是一个新的未约束的符号值。图 1.1 中代码，`main()` 函数的前两行会得到结果 $\sigma = \{x \rightarrow x_0, y \rightarrow y_0\}$，其中 x_0 和 y_0 是两个初始的未约束的符号化值。

当遇到一个赋值语句，形式为 `v=e` ，符号执行就会将符号状态 σ 更新，加入一个 v 到 $\sigma(e)$ 的映射，其中 $\sigma(e)$ 就是在当前符号化状态计算 e 得到的表达式。例如，在图 1.1 中代码执行完第 5 行时，$\sigma = \{x \rightarrow x_0, y \rightarrow y_0, z \rightarrow 2y_0\}$。

当遇到条件语句 `if(e) S1 else S2` ，路径约束 PC 会有两个不同更新。首先是 PC 更新为 $PC \wedge \sigma(e)$，这就表示 `then` 分支；然后是建立一个路径约束 PC'，初始化为 $PC \wedge \neg\sigma(e)$，这就表示 `else` 分支。如果 PC 是可满足的，给定实际值，那么程序执行就会走 `then` 分支，此时的状态为：符号状态 σ 和符号路径约束 PC。反之如果 PC' 是可满足的，那么会建立另一个符号实例，其符号状态为 σ，符号路径约束为 PC'，走 `else` 分支。如果 PC 和 PC' 都不能满足，那么执行就会在对应路径终止。例如，第 7 行建立了两个不同的符号执行实例，路径约束分别是 $x_0 = 2y_0$ 和 $x_0 \neq 2y_0$。在第 8 行，又建立了两个符号执行实例，路径约束分别是 $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$，以及 $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 10)$。

如果符号执行遇到了 `exit` 语句或者错误（指的是程序崩溃、违反断言等），符号执行的当前实例会终止，利用**约束求解器**对当前符号路径约束赋一个可满足的值，而可满足的赋值就构成了测试输入：如果程序执行这些实际输入值，就会在同样的路径结束。例如，在左图例子中，经过符号执行的计算会得到三个测试输入：$\{x = 0, y = 1\}$，$\{x = 2, y = 1\}$，$\{x = 30, y = 15\}$。

符号执行在 2005 年之后的突然重新流行，一大部分原因是因为求解器能力的提升，能够求解复杂的路径约束。但是约束求解在某种程度上依然是符号执行的关键瓶颈，也就是说符号执行所需求的约束求解能力超出了当前约束求解器的能力。

对于浮点数的约束求解一直受限于浮点数的特殊结构及浮点数运算的特殊性质。经典的约束求解器往往将浮点数作为实数理解，这样会导致求解的约束表达与程序的实际语义不符，生成的测试用例实际执行路径与符号执行路径不一致。因此，一些工作

试图增强符号执行及约束求解器对于浮点数的支持。著名的符号执行工具 KLEE [13]，在 2017 年加入了对浮点数的支持 KLEE-Float [43]，Bagnara 等人 [5] 尝试利用浮点数位运算级别的二进制表达来改进符号执行的路径及约束求解能力，著名的约束求解器 Z3 [19] 通过加入随机及启发式算法来增强对浮点数的约束求解能力。

1.3.1.2 针对异常检测的相关技术

错误（errors）和异常（exceptions）是常见的程序机制，用于表示超出程序正常执行流程的特殊条件。浮点数操作同样会导致异常，在 IEEE-754 [81] 浮点数标准中，定义了 5 类异常：

- 无效操作。例如 $\infty - \infty$，$0 \times \infty$，$\sqrt{-1}$ 等；
- 除零错误。例如 $5/0$；
- 向上溢出。运算结果的绝对值超出浮点数能表达的最大数值。
- 向下溢出。运算结果的绝对值小于浮点数能表达的最小数值。
- 不精确。当舍入结果不准确时。

如果上述异常，尤其是前四类异常未经处理，则可能导致程序产生意料之外的结果。2013 年，Barr 等人提出工具 Ariadne [9]，用于生成会导致浮点数异常的测试用例。Ariadne 通过浮点数的特定性质，将浮点数理解为实数运算，给定一系列会触发异常的条件。然后使用约束求解器求解符号执行的路径约束和触发异常的条件约束，找到可能会触发异常的实数值，再检测对应的浮点数值及其临域的浮点数值是否会触发异常。Ariadne 使用 KLEE 作为符号执行工具，使用 Z3 作为约束求解工具。

当向下溢出发生后，浮点数的结果将成为非规格化数，对于 64 位浮点数而言，非规格化数大约位于 $(10^{-324}, 10^{-308})$ 的范围内。由于非规格化数的表达与规格化浮点数具有较大差别，从编译器的指令到硬件的浮点运算单元都无法对非规格化数进行较好的优化，导致涉及到非规格化浮点数的运算效率底下。基于这一特点，Andrysco 等人 [3] 提出一种针对非规格化浮点数的安全攻击，使用模糊测试（fuzz testing）的方法生成大量非规格化浮点数运算的测试用例，用于安全攻击。

1.3.2 针对误差问题的浮点数测试与分析工作

本小节将介绍针对浮点数误差问题的程序测试与分析相关工作。包括：

- 浮点数误差上界分析。总体而言，误差上界分析是一种静态分析方法，可以生成——对于所有输入而言——可能的浮点数程序误差上界。此类工作能产生一个程序误差的上界，无法生成具体的触发误差的测试用例，因而在实践中开发者往往无从下手进行调试。因此，此类工作受到*稀疏性*问题的制约，限制了其在实践中的应用；

- 基于搜索的误差检测。总体而言，基于搜索的误差检测是动态分析方法，目标是搜索会触发显著误差的测试用例。如上文中所述，在浮点数误差问题上，基于搜索的已有工作对误差的检测效果不佳。

1.3.2.1　浮点数误差上界分析

误差上界分析普遍基于静态分析。在相关的静态误差上界分析工作中，最为核心的部分即为误差积累模型。相关研究主要使用区间运算和仿射运算两种误差积累模型，本节将从这两方面入手，介绍相关的研究现状。

基于区间运算的误差上界分析　在区间运算（interval arithmetic），也称为区间分析中，变量（实数）x 被表示为一个浮点数区间 $\bar{x} = [x_{min}, x_{max}]$。这是由于浮点数精度有限，无法精确的表达实数，因而实数的值实际位于两个浮点数值之间。经典的区间分析工作中 [33, 82]，定义了一系列区间运算的基本运算规则，例如对于区间 $\bar{x}$ 和 $\bar{y}$，加法，减法，和乘法分别定义为

$$\bar{x} + \bar{y} = [x_{min} + y_{min}, x_{max} + y_{max}]$$

$$\bar{x} - \bar{y} = [x_{min} - y_{max}, x_{max} - y_{min}]$$

$$\bar{x} \times \bar{y} = [min\{x_{min}y_{min}, x_{min}y_{max}, x_{max}y_{min}, x_{max}y_{max}\}, max\{x_{min}y_{min}, x_{min}y_{max}, x_{max}y_{min}, x_{max}y_{max}\}]$$

具体而言，假设实数 x, y 分别位于区间 $\bar{x} = [2,4], \bar{y} = [-3,2]$ 中，则二者之和 $x + y$ 则位于区间 $\bar{x} + \bar{y} = [2 - 3, 4 + 2] = [-1,6]$ 中，而二者之积 xy 则位于区间 $\bar{x} \times \bar{y} = [4 \times (-3), 4 \times 2] = [-12,8]$ 中。

在大多数情况下，区间除法 $\bar{x}/\bar{y}$ 仅定义在 $\bar{y}$ 不包含 0 的条件下。然而，这样的限制在实践中是不可接受的。一些研究试图使用区间运算的变体来重新定义除法，试图改善该算法在实践中的可行性 [33, 59]。

区间分析最早在 1960 年代由 Ramon E. Moore 正式提出 [49, 50]，在 90 年代，区间分析被认为是一种灵活且有效的范围分析工具。区间分析的流行也得益于浮点数标准 IEEE-754 [81] 被广泛接受。因为在 IEEE-754 标准中定义了确定性的舍入准则，该准则使得有效且不依赖于特定设备的误差边界分析成为可能。

然而，区间分析通常会得到比实际计算值宽松很多的上界。一个极端的例子是，对于 $x \in \bar{x} = [2,5]$，表达式 $x - x$ 的区间被计算为 $[2-5, 5-2] = [-3,3]$，而不是 $[0,0]$，显然后者才是该表达式的精确区间。同时我们注意到，由于区间分析不包含具体的变量信息，因此其并不能将两个相同的 $[2,5]$ 区间假定为表示相同的变量（而抵消），因为这也可能是两个独立的变量恰好具有相同的区间范围。

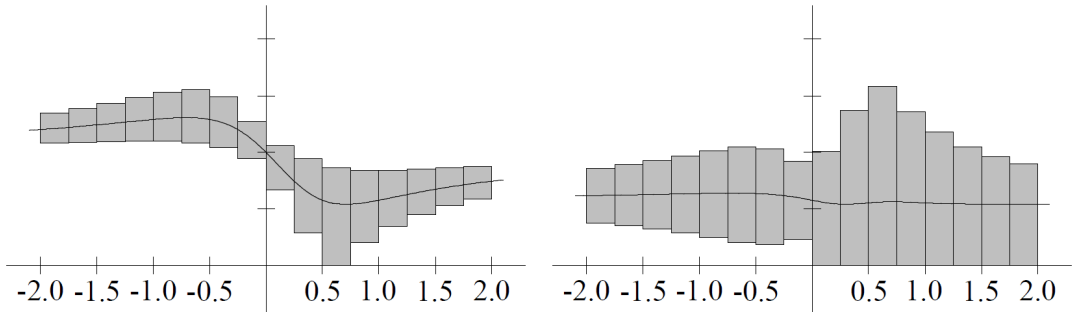


图 1.2　区间分析的边界膨胀问题。$\bar{x} = [-2, 2]$。**左侧**为函数 $g(x) = \sqrt{x^2 - x + 1/2}/\sqrt{x^2 + 1/2}$ 及其区间分析结果；**右侧**为二次迭代后的 $h(x) = g(g(x))$ 及其区间分析结果。

一般而言，如果一个浮点数操作 $z \leftarrow f(x, y)$ 的操作数 x 与 y 具有相关性，则区间分析得到的区间 $\bar{z} \leftarrow \bar{f}(\bar{x}, \bar{y})$ 将比实际的区间要宽松的多。在区间分析中，通常将分析所得到的区间与实际区间宽度的比值定义为高估系数 σ，事实表明，高估系数 σ 仅取决于浮点数操作的类型和数量，而与实际的区间无关。例如，考虑表达式 $x \times (10 - x)$，如果 $x \in [3, 5]$，则区间分析结果 $[15, 35]$ 的宽度为精确区间 $[21, 25]$ 的 5 倍；如果 $x \in [3.9, 4.1]$，区间分析结果 $[23.01, 25.01]$ 的宽度仍为精确区间 $[23.79, 24.19]$ 的 5 倍。

区间分析的边界过于宽松，阻碍了区间分析在实践中的应用。在链式计算下，每一个操作的结果都将作为下一个操作的输入，高估系数 σ 会不断相乘放大，最终导致区间过于宽松而失去实际意义。图 1.2 展示了区间过于宽松的样例。在这种情况下，为了得到有效的区间边界，相关研究将不得不对初始区间划分为数量庞大的、更多更小的子区间，并对每个子区间进行分析计算 [67]。

基于仿射运算的误差上界分析　仿射运算（affine arithmetic）是为了解决区间分析的边界膨胀问题而提出的一种自解释计算模型（self-validated computation methods）[66, 67]。仿射运算不仅记录每个变量的区间，也记录了（误差）数值之间的相关性。由于这类额外的相关性信息，每个仿射运算中的估计的误差与输入区间具有平方关系。因此，如果输入区间足够小，每个仿射运算都可以对变量的区间进行严格的（非宽松的）估计，即高估系数 σ 接近于 1。对于链式计算，该方法同样可以保证不受区间膨胀问题影响。

在仿射运算中，变量被表示为仿射表达 $\hat{x}$，也就是一阶多项式的形式

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n.$$

其中 x_i 为浮点数，ε_i 表示未知的误差量，并假定误差量位于区间 $[-1, 1]$ 中。在仿射表达中，x_0 称为中心值，系数 x_i 称为局部偏差，ε_i 称为误差项（也称为噪音）。

每个误差项 ε_i 代表全体误差中的一个独立分量；相应的系数 x_i 代表了该分量的大

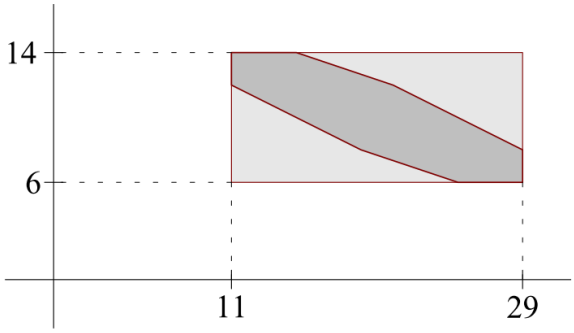


图 1.3　相关变量 $\hat{x}$ 和 $\hat{y}$ 的联合区间 $\langle \hat{x}, \hat{y} \rangle$。其中 $\hat{x} = 20 - 4\varepsilon_1 + 2\varepsilon_3 + 3\varepsilon_4$，$\hat{y} = 10 - 2\varepsilon_1 + 1\varepsilon_2 - 1\varepsilon_4$

小。这种误差的来源既有可能是外部的（已经存在于输入数据中），也有可能是内部的（由于本次运算的舍入导致的）。

仿射运算的一个关键特性在于，同一个误差项 ε_i 可能存在于不同的变量的仿射表达中。由于共享误差项，不同的变量 $\hat{x}$ 和 $\hat{y}$ 之间可能存在相关依赖关系，该依赖关系由相应的系数 x_i 和 y_i 确定。例如，假设 $\hat{x}$ 和 $\hat{y}$ 的仿射表达为

$$\begin{aligned} \hat{x} &= 20 - 4\varepsilon_1 \quad\quad + 2\varepsilon_3 + 3\varepsilon_4 \\ \hat{y} &= 10 - 2\varepsilon_1 + 1\varepsilon_2 \quad\quad - 1\varepsilon_4 \end{aligned}$$

根据上述数据，我们可以知道 x 位于区间 $\bar{x} = [11, 29]$ 中，而 y 位于区间 $\bar{y} = [6, 14]$ 中；即 (x, y) 位于图 1.3 的灰色矩形中。但是，由于 $\hat{x}$ 和 $\hat{y}$ 的仿射表达都包含非零系数的误差项 ε_1 和 ε_4，因此他们并非完全独立。实际上，无论误差项 $\varepsilon_1, \ldots, \varepsilon_4$ 如何取值，(x, y) 一定位于图 1.3 的深灰色区域中。该区域称为 $\hat{x}$ 和 $\hat{y}$ 的联合区间（joint range），表示为 $\langle \hat{x}, \hat{y} \rangle$。

有了上述概念，便可以定义仿射表达之间的运算法则。对于基础仿射运算，$z = f(x, y) \to \hat{z} = \hat{f}(\hat{x}, \hat{y})$，包括加减法，常量加法，和常量乘法，可由如下公式定义

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + (x_1 \pm y_1)\varepsilon_1 + \cdots + (x_n \pm y_n)\varepsilon_n$$

$$\alpha\hat{x} = (\alpha x_0) + (\alpha x_1)\varepsilon_1 + \cdots + (\alpha x_n)\varepsilon_n$$

$$\hat{x} \pm \beta = (x_0 \pm \beta) + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n$$

对于非仿射类运算，$z = f(x, y) \to \hat{z} = f(\hat{x}, \hat{y})$，例如乘法，则定义为

$$\hat{x}\hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i)\varepsilon_i + z_k \varepsilon_k$$

$$z_k = \sum_{i=1}^{n} |x_i| \sum_{i=1}^{n} |y_i|$$

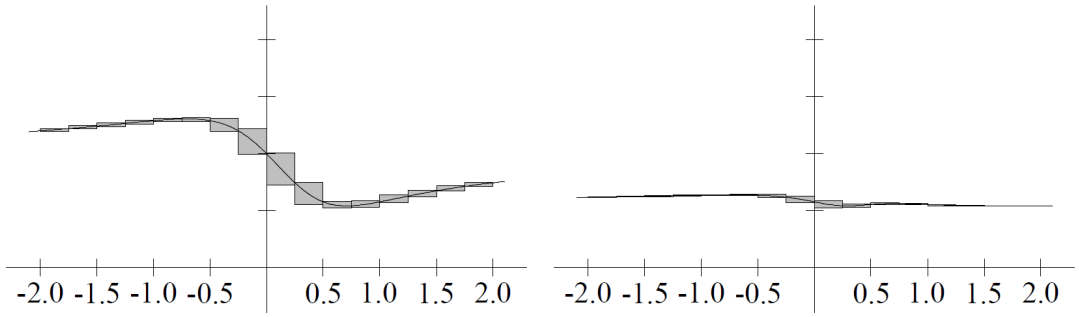


图 1.4　通过仿射运算解决边界膨胀问题。$\bar{x} = [-2,2]$。**左侧**为函数 $g(x) = \sqrt{x^2 - x + 1/2}/\sqrt{x^2 + 1/2}$ 及其仿射运算结果；**右侧**为二次迭代后的 $h(x) = g(g(x))$ 及其仿射运算结果。

在使用区间分析的情况下，可以直接将其中的区间运算替换为仿射运算。由于仿射运算的高估系数普遍比区间运算要小，所以这是一种可替代区间运算的方法。具体来说，如果区间分析使用了 n 个变量并且具有高估系数 σ，则使用仿射运算可以将高估系数降低 $1/\sigma^n$。由于高估系数 σ 通常远大于 1，因此带来的效果提升非常显著 [66]。对于相同的运算，仿射运算（图 1.4）相比于区间运算（图 1.2）有很大的效果提升。

1.3.2.2　基于搜索的误差检测

本小节将从：（1）搜索目标函数及测试预言的计算（2）检测误差的搜索方法，这两方面介绍基于搜索的误差检测相关工作。具体而言，基于搜索的两类相关工作具有如下特点：

- 搜索目标函数及测试预言的计算，即计算误差。此类工作尝试对运算误差进行计算，旨在解决*未知性*问题对误差检测的制约；
- 检测误差的搜索方法，即搜索测试用例。此类工作尝试使用搜索算法，在巨大的搜索空间中找到会触发显著误差的测试用例，旨在解决*稀疏性*问题对误差检测的制约。

搜索目标函数及测试预言的计算　一直以来，获得浮点数程序的测试预言，即计算精确的运算结果被认为是一项困难的任务 [10]。

在已有的基于搜索的方法中，普遍需要计算误差——即精确结果与浮点数程序运算结果的差——来作为搜索的目标函数。因此浮点数程序的测试预言是所有基于搜索的相关技术的前提和保证。

为了获取高质量的浮点数测试预言，一个最简单也最为直接的方法便是提高浮点数的位数以获得更高的精度。由于真正的测试预言 $f(x)$ 为无限精度的实数，不可能在计算机系统中精确的表达，所以使用高精度浮点数程序 $\hat{f}_{high}(x)$ 作为精确结果的替代，即作为测试预言，用于分析浮点数程序。

因此，Fousse 等人提出了高精度的浮点数库 MPFR [25]。GNU MPFR 是一个基于 GNU 高精度库（GNU Multi-Precision Library）的支持任意精度的浮点数计算库。其对于高精度的浮点数运算具有明确定义的语义，即平台无关性。MPFR 遵循了 IEEE 754 标准中对于浮点运算的标准定义，包括其中关于舍入和异常的定义，使得 MPFR 提供的高精度浮点数类型具有与标准浮点数类型相同的运行特征。具体而言，MPFR 具有以下特性：

- 支持特殊数值。包括带符号的零（+0 及-0），无穷大（$+\infty$ 及 $-\infty$）以及非数（not-a-number，NaN）；
- 每个数值变量都可以设定为一个确定的精度，并支持符合 IEEE 754 标准的舍入模式；
- 支持 C99 中的所有数学函数，以及部分常用的数学函数，包括指数函数、对数函数、三角函数及其反函数、双曲函数及其反函数、伽马函数（Γ 函数）、黎曼 ζ 函数等等。所有支持的数学函数都保证进行正确的舍入。
- 不支持非规格化浮点数。但是可以通过 mpfr_submornalize() 函数进行模拟。

通过使用 MPFR，研究人员可以方便的利用 MPFR 重新编写高精度的运算程序 $\hat{f}_{high}$，用于获得待分析的浮点数程序 $\hat{f}$ 的测试预言。

MPFR 是一个提供了高精度类型的浮点数运算库，可以理解为一个基础工具。对于简单的浮点数程序，开发者可以将其中的浮点数类型直接替换为 MPFR 类型，得到其高精度版本。然而，由于数值计算程序往往庞大且复杂，有许多其他库的依赖（此类依赖库往往不支持 MPFR 类型的浮点数计算），因而对于复杂的数值计算程序，构建基于 MPFR 的高精度版本的工程量十分庞大甚至不再可行。

上文提到的 MPFR 只是一个基础的高精度浮点数运算库，无法自动的生成测试预言。因此，Benz 等人提出了自动化的浮点数测试预言生成工具 FpDebug [10]。FpDebug 基于二进制分析工具 Valgrind[51] 和高精度浮点数运算库 MPFR 构建而成。对于一个浮点数程序，FpDebug 通过如下方法获得其高精度运行结果：

1. 对于程序计算出的每个*原始值*（包括写入内存的变量及保存在寄存器中的临时结果），其维护一个*影值*（shadow value）。影值为 MPFR 提供的高精度浮点数类型，具有比原始值更高的精度。
2. 对于原始值的每一次运算，使用 MPFR 提供的高精度运算函数，对影值进行同类型的相同运算。
3. 对于每次运算，计算运算结果中影值与原始值的相对误差。

FpDebug 基于*实数运算语义*，即假设提高浮点数运算可以完全理解为实数运算，提

升类型的精度一定可以得到更准确的结果。相比于 MPFR，FpDebug 可以自动化生成浮点数测试预言，极大的提高了测试预言生成工具的可实践性。给本节接下来将介绍的依赖测试预言的误差检测相关工作提供了基础。

然而，数值计算的语义往往无法通过准确的程序描述，使得基于实数运算语义的上述工具产生错误，给获取高质量的测试预言带来了困难。例如在数值计算领域非常重要的贝塞尔函数（Bessel functions），是下列常微分方程的标准解函数 $y(x)$：

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0$$

这类方程的解无法用初等函数表示。在计算此类函数时，需要大量使用各类数值分析的算法，例如在著名的 GNU 科学计算库 GSL（GNU Scientific Library）中，关于贝塞尔函数的求值就使用了 Thompson-Barnett-Temme 方法 [69]，切比雪夫展开 [47]，Hankel 变换 [74] 等等大量数值方法。此类数值方法中存在的精度特定运算 [84]，会导致语义解释错误，进而导致高精度类型得到错误的运算结果。现有关于精度特定运算的唯一相关工作是本文作者合作完成的工作 [84]，该工作主要使用实证研究的方法探究其特点和检测的可能性。

检测误差的搜索方法 本文在第 1.3.2.1 节中已经介绍了在保证浮点数运算的精确性方面的一些研究成果。这类方法使用基于静态分析方法的区间运算或仿射运算，来判断浮点数程序误差的上界。然而，这类方法具有以下几点局限性：

- 由于静态分析方法自身的局限性，计算出的误差上界只是实际误差的一个估计值，并且该估计值不精确。即使在最新的研究成果中 [17]，计算出的误差上界仍然要比实际的误差大几个数量级，甚至有时上界是无穷大。所以，即使计算出了一个大的误差上界，仍然不能保证是否存在精度相关问题。
- 在调试程序时，开发者需要的是一个会触发显著误差的测试用例，以便于追踪程序的执行过程，找到产生显著误差的原因并修复它。然而，已有的研究成果 [8] 指出，在所有可能的输入中，只有很小的一部分输入会触发显著的误差。所以，对于开发者来说，通过手动查找的方法获取会触发显著误差的输入值是非常困难的。

因此，一些工作开始探索使用基于搜索的方法检测会触发显著误差的测试用例，以便于开发者进行调试。其中比较经典的工作有 2014 年由 Chiang 等人提出的 BGRT [15]，和 2015 年由 Fu 等人提出的 BEA [26]。上述工作都是使用搜索算法的测试用例生成工具，接下来本文将详细阐述。

BGRT（binary guided random testing）是一种基于二分区间搜索的浮点数误差检测算法。BGRT 算法由初始的有效输入区间开始，反复迭代至会产生较大误差的更小的

区间。每次 BGRT 的迭代都从一个初始区间开始，枚举一些更小的区间切分方式，然后选择（局部）误差最大的区间进入下一次迭代。最终迭代至终止条件（区间宽度足够小，或找到产生足够大误差的输入）则停止。

BGRT 是一种启发式的算法，在其论文中并未深入探究该算法的有效性原理。同时，该算法严重依赖于测试预言，导致该论文的实验仅在 32 位浮点数程序上进行——同时其使用完全对应的 64 位浮点数程序的运算结果作为测试预言。然而，由于在实践中单纯提高浮点数的精度并不能得到高质量的测试预言 [73]，因此该设定限制了 BGRT 的可实践性。同时，由于论文的实验部分仅对比了一些基础的随机算法，使得 BGRT 算法的有效性缺乏严谨的支持。

BEA（backward error analysis）是一种基于反向误差的浮点数误差检测方法。在度量浮点数程序的精确性时，大多数程序都使用前向误差——即理论上的精确值（测试预言）与浮点数程序运算结果之间的误差——作为度量标准。本文之前提到的相关技术，包括静态分析、符号执行等方法，都对前向误差进行了深入的研究。而对于数值分析的研究人员而言，反向误差是一种具有更良好的数学性质的分析范式，其已被应用于手动分析许多基本算法 [34]。因此，BEA 技术尝试利用反向误差在数值分析领域的优势，构建一种更为有效的浮点数误差检测方法。

反向误差的定义。假设一个理想的（数学上的）函数 f，其输入为 x，理论上的输出为 $f(x)$，浮点数程序为 $\hat{f}$ 为 f 的数值计算实现，浮点数程序的输出为 $\hat{f}(x)$。那么，对应的前向相对误差 F_{err} 则可以表示为：

$$F_{err} = \left| \frac{\hat{f}(x) - f(x)}{f(x)} \right|$$

而反向误差则将 $\hat{f}(x)$ 理解为 f 在输入 x 存在一个微小扰动的情况下的结果：

$$\hat{f} \simeq f(x + \Delta x)$$

则将上式中的 Δx 定义为反向误差。图 1.5 中展示了反向误差的定义。

使用反向误差，可以很好的判断前向误差是否存在，以及前向误差产生的原因——是由于原函数 f 自身定义了一个病态（ill-conditioned）问题，还是由于浮点数程序 $\hat{f}$ 实现导致了误差问题。

BEA 将搜索前向误差的问题转化为搜索反向误差的问题，之后使用马尔可夫链蒙特卡洛方法（MCMC）[2] 对浮点数输入空间进行搜索采样，试图找到一个会触发显著反向误差的输入。MCMC 是一类用于模拟目标分布的随机采样技术。采用 MCMC 方法的原因有两点。首先，搜索空间是所有可行的浮点数输入。即使在一维情况下，很小的区间也包含大量的浮点数，MCMC 方法是处理大型搜索空间的有效技术。其次，由

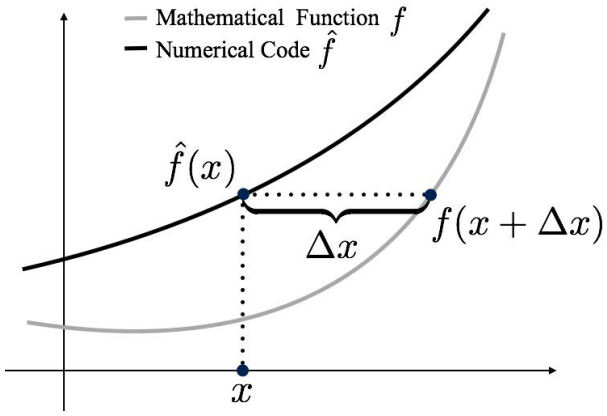


图 1.5　反向误差的定义

于目标函数——即反向误差——不可导也不连续，使得一些基于梯度的搜索算法、优化算法无法工作。因此，考虑到 MCMC 具有良好的数学性质，BEA 使用 MCMC 作为搜索显著反向误差的方法。

在找到大的反向误差后，BEA 对找到的结果进行分析，保留其中会导致大的前向误差的输入，即需要开发者进行修复的误差问题。同时，BEA 也可以作为传统误差检测技术的一个补充：如果一个输入会产生大的前向误差，同时只有较小的反向误差，则证明该误差是由于函数 f 自身而非浮点数程序的实现 $\hat{f}$ 导致的，即为病态问题，开发者无需——事实上也无法——对该误差问题进行修复。表 1.2 中展示了前向误差与反向误差的关系及意义。

表 1.2　前向误差与反向误差的关系

前向误差	反向误差	分析结果
小	大	程序精度对具体实现不敏感
小	小	一个好的浮点数程序
大	小	病态问题
大	大	可以修复的精度问题

反向误差的计算同样依赖于测试预言。因此 BEA 的贡献主要在于对检测浮点数误差的测试用例生成领域的理论推动，以及对于会产生大误差的测试用例的分析。其在搜索模块中使用的 MCMC 方法，虽然在理论上具有良好的收敛性，然而在实践中并未展现出优于其他启发式搜索的效果。

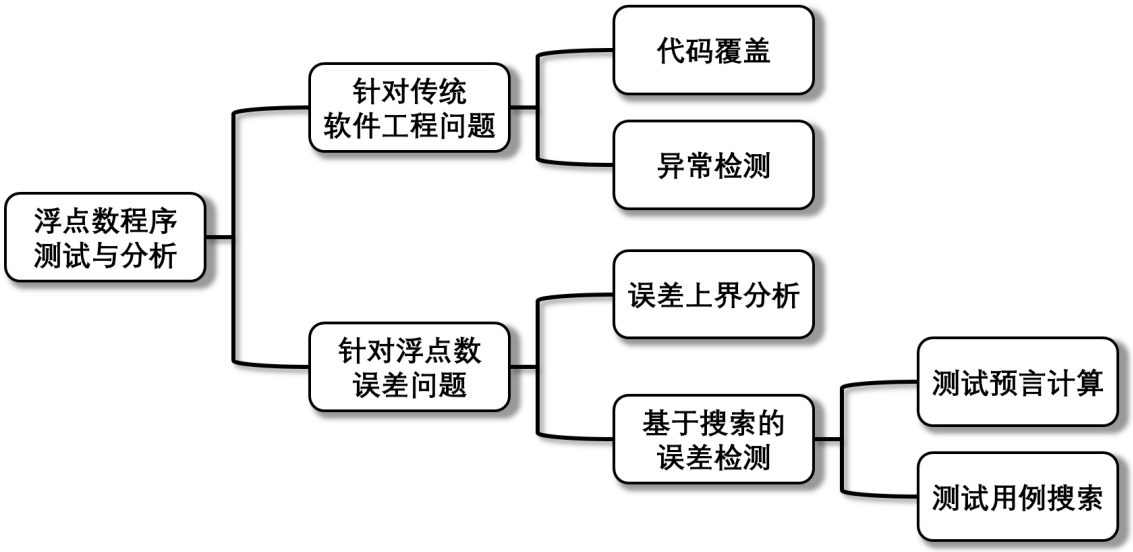


图 1.6　相关工作的分类与总结

1.3.3　研究现状小结

图 1.6 对本节的相关工作进行了分类与总结。总体而言，已有的针对浮点数程序的测试与分析相关技术，可以按照技术目标分为两类：针对传统软件工程问题的相关工作和针对浮点数误差问题的相关工作。

在针对传统软件工程问题的相关工作中，本节介绍了较有代表性的两类工作，即代码覆盖的相关工作和异常检测的相关工作。此类工作的目标与已有的软件工程目标相一致，均使用传统方法——如约束求解器——来达成测试与分析目标，从传统的软件工程角度提升代码质量。

在针对浮点数误差问题的相关工作中，本节较为详尽的介绍了已有的工作。包括静态的误差上界分析相关工作和动态的基于搜索的误差检测相关工作。

误差上界分析的相关工作从静态分析的角度，对浮点数程序在所有输入下的误差上界进行估计。此类工作通过基于区间运算和仿射运算的误差积累模型估计误差。其输出为一个可能的误差上界，并不保证误差上界的触发，也不包含具体的触发误差上界的测试用例。因而，在实践中开发者往往对于这样的分析结果无从下手，限制了其在实践中的应用。

基于搜索的误差检测相关工作从动态分析的角度，尝试在输入空间中搜索会触发显著误差的测试用例，可以给误差的调试带来较大的帮助。此类工作从基于搜索的角度对浮点数误差的检测进行了探索，将误差检测问题转化为搜索问题，目标为会触发显著误差的测试用例。基于搜索的技术必然会涉及到两个关键构件的设定：搜索目标和搜索算法。

- 对于搜索目标，相关工作直接使用误差作为搜索目标。误差为精确结果与浮点

数程序运算结果的差。要计算误差，就必然需要先计算测试预言——即精确的运算结果。已有的相关工作基于实数运算语义对浮点数程序进行解释，直接使用精度调整的方法计算测试预言。然而，由于其未能考虑浮点运算语义，上述假设并不成立，导致其对程序作出错误解释，进而生成错误的测试预言；

- 对于搜索算法，相关工作直接使用已有的搜索算法或最优化算法对误差进行搜索，例如二分搜索算法和蒙特卡洛马尔可夫方法。由于触发显著误差的测试用例在整个搜索空间中极为稀疏 [8]，上述工作未针对浮点数误差问题的特性进行讨论与分析，导致搜索算法效果不佳，并不能有效的对浮点数误差进行检测。

由上述讨论可知，基于搜索的误差检测相关研究开拓了浮点数程序误差测试与分析的新方向。同时，由于相关工作并未对搜索的关键构件进行深入分析，仅直接使用了搜索领域已有的方法与算法，导致误差检测的效果仍不理想。

1.4 尚待解决的问题

在上文的第 1.3.3 节中，本文对已有的相关工作进行了总结。本文发现，对于浮点数误差问题，基于搜索的误差检测方法理论上可以给调试过程带来较大的帮助。然而，相关工作检测显著误差的效果尚不理想。因此，本文总结了基于搜索的浮点数程序误差测试与分析工作的几个尚待解决的问题。通过解决下列问题，可以打通搜索的各个环节，形成一套有效的，适用于不同场景下的，基于搜索的误差测试与分析方法。

首先要解决的是误差的计算问题。由于基于搜索的误差检测技术普遍使用误差作为搜索目标，误差的计算是基于搜索的相关工作的基础和保证。浮点数程序的误差为精确结果与浮点数程序运算结果之差，而精确结果——即测试预言——的获取问题尚未解决。已有工作使用实数运算语义对浮点数程序进行解释，假设提升浮点数类型的精度一定会得到更精确的运算结果。然而，在实际项目中广泛存在的精度特定运算会导致语义解释错误，进而产生错误结果。因此，如何计算精确的运算结果，即测试预言，是一个尚待解决的问题。通过解决该问题，浮点数误差的*未知性*问题也就得到了解决。

然后要解决的是搜索算法的效果问题。相关工作指出，会触发显著误差的浮点数测试用例极为稀少，因此对搜索算法的效果要求很高。已有工作并未对浮点数误差进行深入讨论与分析，直接使用传统的搜索算法对浮点数误差进行搜索，导致搜索效果不好。在一些浮点数程序存在显著误差问题的情况下，仍无法搜索到触发显著误差的输入。因此，如何提升搜索算法的效果，是另一个需要解决的问题。通过解决该问题，浮点数误差的*稀疏性*问题也就得到了解决。

最后值得思考的是，能否在已有的基于搜索的框架上进行拓展，使用更多的信息

以达到更好的误差检测效果。已有工作普遍在黑盒场景下进行误差检测，并不考虑浮点数程序的内部状态与信息。然而在白盒场景下，程序的语义信息是可以利用的重要信息。如何在白盒场景下，利用浮点数程序的语义信息指导搜索算法，以提升搜索的速度和效果，是另一个需要解决的问题。通过解决该问题，可以在不同场景下最大化对误差测试与分析的能力，提升误差检测的效果，解决浮点数误差的稀疏性问题。

1.5　本文主要工作

为了保证浮点数程序的质量，从程序测试和程序分析的角度，尽可能的对浮点数误差进行检测，本文使用了**基于搜索的方法**对浮点数程序误差进行测试与分析，核心目标是找到会触发显著误差的测试用例。基于搜索的软件工程使用启发式搜索的方法解决软件工程问题，其核心包括问题的定义与转化，搜索目标函数的制定，搜索算法的构建等一系列关键技术。针对浮点数误差问题，基于搜索的方法具有下列特点：

- 是一种动态分析方法；
- 可以给出特定的触发显著误差的测试用例；
- 可以参考及利用最优化问题的搜索算法；
- 可以应用于不同的测试与分析场景下。

基于搜索的方法在具有上述特点的同时，也面临未知性与稀疏性问题的挑战：

- 搜索目标函数的制定与获取，受到测试预言未知性的问题挑战；
- 搜索算法的效果，受到浮点数误差稀疏性问题的挑战。

针对基于搜索的方法面临的上述挑战，本文进行了一系列具有针对性的研究，分别为：

- 高质量的浮点数测试预言获取方法研究。该研究提出了兼容语义模型，解决了语义解释错误问题，可以生成正确且准确的测试预言，建立了广泛适用的搜索目标函数；
- 黑盒场景下的搜索算法研究。该研究提出了特定于浮点数误差问题的高效搜索算法；
- 白盒场景下的新搜索目标函数研究。该研究提出了一种新的搜索目标函数“原子状态函数”，在白盒场景下提高了搜索的有效性并降低了运行开销。

具体而言，上述三种方法相互关联，解决了各个搜索流程中的关键问题，其关系如图 1.7 中所示。

在黑盒场景下，测试预言生成工作可以直接用于计算误差，作为搜索目标；测试用例生成工作提出了新的搜索算法，以提高搜索的有效性。

在白盒场景下，误差分析方法提出了新的搜索目标，并应用搜索算法对误差进行搜索；同时测试预言生成工作在最终用于对少量分析结果计算误差，起到结果验证的作用。使用新的搜索目标可以提高搜索的有效性；仅在最终验证中计算少量的测试预言，可以降低测试与分析的运行开销。

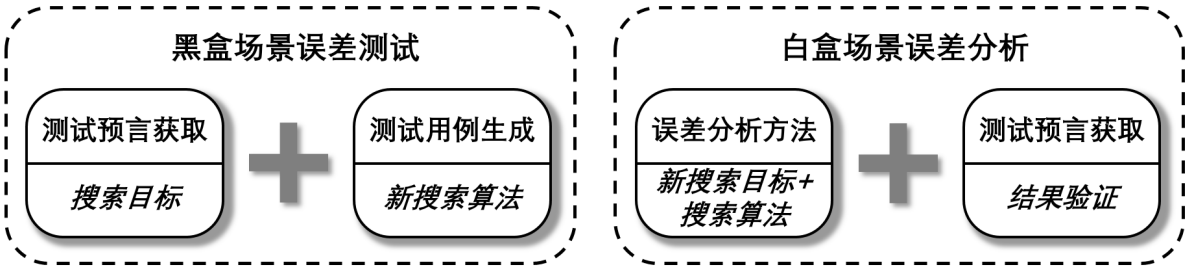


图 1.7　本文提出的各测试与分析技术之间的联系

接下来，本节将分别介绍本文提出的一系列**基于搜索**的浮点数程序误差测试与分析相关研究。

1.5.1　面向搜索的浮点数测试预言自动生成技术

第 1.3.2.2 节中已经介绍，一直以来，浮点数程序的测试预言（oracle）自动生成被认为是一项困难的任务。尽管经过多年的发展，高质量与自动化这两个目标仍然无法同时达成。已有的自动化测试预言生成技术基于实数运算语义，只对于原始程序的每一步操作进行镜像化的高精度模拟，会对精度特定运算做出错误语义解释，导致生成的测试预言不精确甚至错误。高质量的测试预言生成技术严重依赖专家知识（expertise），需要数值分析专家使用各类数值方法手动构建测试预言。例如一个数值分析中的基础函数——贝塞尔函数（Bessel function）——就要手动处理 Thompson-Barnett-Temme 方法 [69]，切比雪夫展开 [47]，Hankel 变换 [74] 等多种复杂的数值分析方法，需要大量时间成本和人力成本。

同时，由于浮点数误差问题的技术链式依赖特点，浮点数误差的测试与分析研究严重依赖于测试预言的质量，因而一个高质量的浮点数测试预言自动生成技术是十分重要且必要的。

通常而言，基于实数运算语义，将浮点数程序中的浮点数类型理解为实数，通过提高浮点数类型的精度便可以提高其运算结果的精度——例如将 32 位的单精度浮点数替换为 64 位的双精度浮点数。然而，对于一类特殊的浮点数运算——精度特定运算 [84]，本文也称为浮点语义运算——使用更高精度的类型反而会违背程序语义，导致错误结果。语义解释错误是制约测试预言生成的核心因素。本工作从浮点数程序的语义解释入手，建立兼容实数运算语义与浮点运算语义的语义模型，对其进行自动检测与修复，

生成高质量的测试预言。

浮点语义运算是一种被设计为特定于浮点数类型的运算，其语义仅在特定的浮点数精度类型下成立。例如，以下代码是从 GNU C Library（glibc）中简化而来：

```
1 double round(double x) {
2     double n = 6755399441055744.0; // n = 3<<51;
3     double y = (x + n) - n;
4     return y;
5 }
```

浮点数不能准确地表示所有的实数，所以浮点数的计算结果和数学表达式的计算结果不一定相同。如果运行上述代码，当输入值 $x = 3.7$ 时，返回值 $y = 4$。这是由于在这个计算中，n 是一个非常大的数字。当比较小的 x 与 n 相加时，x 的部分位数会丢失。事实上，上述代码的目的并不是得到准确的 x，而是为了能够将 x 四舍五入。在设计精妙、融合了各种数值计算技巧的各类数学函数库（例如 glibc）中，此类运算大量存在，旨在高效的完成各类运算操作。

如果使用更高的精度来计算，那么 y 的结果会得到（或接近）3.7。然而，该操作却事实上违背了原始程序的语义，使得运算结果错误。在该代码中，n 的值是一个特定值（magic number），是特意为 64 位浮点数类型设计的用于四舍五入的常数。如果直接将浮点数精度提升为更高精度，该常数就无法再起到四舍五入的作用。上述运算即为一个浮点语义运算——运算语义特定于浮点数类型，而无法使用实数运算语义解释的运算。

在解决语义解释错误带来的困难中，有三个关键性的挑战：

- 建立语义模型。由于实数运算语义不足以支撑对浮点数程序的理解，基于实数运算语义的精度调整会导致语义解释错误，进而导致测试预言错误。因此需要建立一个支持浮点运算的语义模型；
- 语义检测方法。在上述模型下，如何区分浮点数程序运算的语义类型；
- 精度调整方法。如何利用上述语义模型，在提升精度执行的同时，避免语义解释错误带来的影响。

对于语义模型问题，本工作建立了一个轻量级的兼容语义模型，该模型可以对浮点数运算进行分类，对于符合实数运算语义的运算，使用实数语义进行解释；对于不符合实数运算语义的运算，则使用浮点运算语义对其进行解释。

对于语义检测方法，本工作提出了基于统计的方法对运算进行检测与划分。由于浮点数误差的稀疏性，其仅在极少数输入下会产生大的误差，而如果出现语义解释错误，则其表现为始终产生大的误差。因此本文设计了一套基于统计的启发式规则，根据每个运算在实数语义解释下出现误差的概率，检测其真实的语义。

对于精度调整方法，本工作提出了在实数语义运算上提升精度，在浮点语义运算上还原精度的策略。在保证语义解释正确的同时，使用精度调整的方法得到更准确的运算结果。

本工作基于兼容语义模型，对程序进行语义检测与精度调整，生成高质量的浮点数测试预言，作为高质量的搜索目标函数，解决**未知性**问题。

1.5.2 基于搜索的浮点数黑盒测试用例生成技术

在第 1.3.2.2 节中已经介绍，已有针对浮点数误差问题的测试用例生成技术效果不好，在实践中的可行性不高。这是由于已有技术普遍使用经典的启发式搜素算法，并未针对浮点数的数据结构特点以及误差的行为进行分析优化，导致其对浮点数显著误差的检测效果不好。

为了提高在黑盒——即源代码不可见——场景中的测试效果，本文对浮点数数据结构特点与误差行为进行了深入分析。浮点数的数据结构由符号位、指数位和尾数位三部分组成。本文发现，在浮点数运算中，各个结构对浮点数误差的影响方式是完全不同的。浮点数的各个结构通过以下方式影响结果的误差：

- 第一是指数位的值的大小。因为指数位的值是决定整个浮点数大小的主要因素，在指数位相同的情况下，无论怎样修改尾数位，该浮点数的值都不会与原来的值相差一倍以上。所以，只有当指数位为特定的值时，浮点数才可能会进会导致显著误差的范围。
- 第二是尾数位各个位（bit）之间的组合。由于浮点数使用“向偶数舍入”的舍入方式，该方式会将结果舍入为最接近且可以被浮点数表示的值，且当存在两个数一样接近的时候，则取其中的偶数（在二进制中以 0 结尾的）进行舍入。在这种舍入方式下，只有当舍入的误差不能相互抵消时才会产生大的误差。所以，只有当尾数位的各个位之间以特殊的方式组合时，才会使得舍入误差不能相互抵消，进而产生显著的误差。

通过以上发现，本工作提出一种特定于浮点数误差问题的层级混合粒子群算法，用于检测浮点数程序的显著误差问题，生成测试用例，应用于源代码不可见的黑盒测试场景。

粒子群算法是一种启发式算法，其基于模拟自然界动物的运动特点而提出，通过群体的信息共享提供演化信息，进而解决最优化问题 [36]。粒子群算法的本质是一种基于群体的全局搜索方法，它能在搜索过程中自动获取和积累关于搜索空间的信息，并自适应的控制搜索过程，以求得全体最优解。

由于粒子群算法的可定制性，本文设计了一种特定于浮点数误差问题的层级混合粒子群算法。该工作对搜索过程进行层级处理，在第一层级中针对浮点数的指数位，搜

索可能触发显著误差的区域；在第二层级中针对浮点数的尾数位，在子区域内进行局部搜索，找到会触发显著误差的输入。该工作通过在不同层级设定不同的搜索方向，使用特定的距离定义、速度计算、群体更新等操作，充分利用了浮点数数据结构中的不同组成部分对于误差的不同影响方式，显著提高了搜索算法的效果。

通过该方法，可以在巨大的搜索空间中得到一个有效的全局搜索算法，得到一种高效的检测浮点数程序的显著误差问题的测试用例生成技术。该工作在黑盒测试的场景下作为搜索算法解决**稀疏性**问题。

1.5.3　基于搜索的浮点数白盒分析技术

由于浮点数误差的未知性，大量测试和分析工作都依赖于高质量的测试预言来判断误差的大小。而即使在高质量测试预言存在的情况下，其运行效率仍然不高，得到测试预言的开销往往比原始程序的运算开销高数千倍 [55]。因此，本文试图根据浮点数程序内在的数学性质，提出一种不依赖于测试预言的分析技术，对浮点数误差进行白盒分析。该白盒分析技术应用于源代码可见的场景中。

该工作基于对浮点数运算中对误差的引入，传播，及放大过程的深入分析和理解，使用数值分析中的状态函数概念对误差进行分析，从而可以避免在分析过程中使用开销昂贵的测试预言计算误差。状态函数测量了一个数学函数的敏感性，即测量输入参数的微小变化会导致多少输出的变化。并且，该工作通过研究一系列浮点数原子操作，例如 +，−，sin，log 等操作上的状态函数，提出了**原子状态函数**这一概念，用于高效的进行浮点数误差分析。

该工作深入探究了原子状态函数对于浮点数误差的影响，通过原子状态函数构建了误差的生成及传播模型，并进一步通过该模型解释原子状态函数在浮点数误差中起到的作用。

该工作具有以下几点核心优势：

- 运行迅速。分析过程不依赖于测试预言，仅在最终结果验证步骤需要测试预言。由于原子状态函数可以直接在原有精度下得到，因此该分析方法的运行效率很高。
- 分析有效。相比于直接使用误差作为搜索目标，原子状态函数提供了准确的误差生成及积累信息。这些信息可以提高对显著误差的搜索效果。
- 提供调试信息。通过原子状态函数可以清晰的得到误差在哪些浮点数运算上生成及放大，这些信息可以为缺陷定位及缺陷修复等调试工作提供辅助信息。

总体而言，该工作提出了一种新的搜索目标函数，提升了误差分析的运行效率与搜索效果。该工作在白盒分析的场景下解决**稀疏性**问题。

1.6 论文组织

本文章节结构如下：

- **第一章** 引言。主要内容包括问题的提出，浮点数背景的介绍，相关研究现状的探讨，以及本文研究目标和主要创新点的提出。
- **第二章** 面向搜索的浮点数测试预言自动生成技术。对于浮点数测试预言生成问题，本文探究了测试预言生成面临的主要挑战，提出了兼容语义模型，以及对应的语义解释及精度调整方法，用于自动化生成高质量的测试预言，建立了广泛适用的搜索目标函数，解决未知性问题。
- **第三章** 基于搜索的浮点数黑盒测试用例生成技术。在黑盒场景下，本文探究了已有搜索算法在检测浮点数显著误差问题上效果不佳的原因，分析了浮点数类型的数据结构特点及浮点数程序的运算特点，设计了一种特定于浮点数误差问题的层级混合粒子群算法，提高了搜索显著误差的效果。
- **第四章** 基于搜索的浮点数白盒分析技术。在白盒场景下，本文深入探究了浮点数误差在运算间的引入、传播、及放大过程，提出了基于**原子状态函数**的误差累积模型，并以原子状态函数作为新的搜索目标，提升了误差分析的运行效率与搜索效果。
- **第五章** 结论和展望。对本文研究工作进行总结，并展望未来进一步的研究方向。

第二章　面向搜索的浮点数测试预言自动生成技术

2.1　引言

测试预言是浮点数程序测试与分析中的重要一环。由于浮点数误差问题的*未知性*特点，高质量的测试预言是几乎所有浮点数程序测试与分析工作的前提和保证。在误差检测 [8, 10, 15, 80] 和浮点数程序验证 [17, 30, 57] 的工作中，需要测试预言来计算程序运算结果与理想结果之间的误差；在浮点数优化 [18, 53] 的工作中，需要测试预言来使得运算结果尽可能接近于理想结果。通常而言，已有工作中使用的测试预言生成工具普遍使用*实数运算语义*的方式理解浮点数运算。换言之，将浮点数作为实数对待，将浮点数运算作为实数运算对待。实数运算语义将浮点数程序的语义等价于实数运算的语义，并使用近似方法计算的实数运算的近似值，作为测试预言。常用的近似方法即为*精度调整*，即将浮点数的精度提高。例如将 32 位的 `float` 类型替换为 64 位的 `double` 类型，甚至更多位[①]的 `long double` 类型等，并认为结果的精度一定会随着浮点数精度的提高而提高。

然而，浮点数程序中含一类特殊的运算——精度特定运算 [84]，可能会导致高精度的运算结果出现错误。本文发现，出现错误的根本原因是，此类运算特定于浮点数类型执行的语义，而上述工具使用实数运算语义对其的解释出现错误。本章从语义解释的角度进行分析，在下文中将其称为*浮点语义运算*，以便与*实数语义运算*进行区分。具体而言，浮点语义运算是一种被设计为特定于浮点数类型的运算，其语义仅在特定的浮点数精度类型下成立。该类运算广泛存在于各类基础运算库中，用于提高基础运算库的运算效率和运算精度。例如，在 GNU C Library（glibc）运算库的 `exp` 中函数中一个样例：

```
1 double round(double x) {
2     double n = 6755399441055744.0; // n = 3<<51;
3     double y = (x + n) - n;
4     return y;
5 }
```

该函数的语义为：将变量 x 近似舍入为最近的整数，并返回舍入后的结果。在这段代码中，变量 n 的值是一个为 64 位浮点数特殊设计的魔术值（magic number）。在第 3 行语句中，计算 x 与 n 的和时，由于结果的绝对值过大，小数部分将被舍入；当结果减

① C 语言标准中，只规定 `long double` 类型的精度必须高于 64 位的 `double` 类型，并未规定其具体精度。在 x86 架构下，通常编译器将其实现为 80 位精度的浮点数。在不同架构及不同编译器下，`long double` 的精度亦有不同。

去 n 时，便可以得到 x 被舍入后的结果。由于上述操作是浮点语义运算，使用实数运算语义并不能正确的解释该运算的语义。不同语义下的解释如下：

- 实数运算语义：如上文所述，实数运算语义将浮点数作为实数对待，因而该函数的语义被解释为计算 $x+n-n$，在实数上即等价于直接返回 x 的值；
- 浮点运算语义：返回将 x 舍入为最近的整数的值。

如果使用实数运算语义理解上述代码，便会得到错误的结果。例如，在已有的误差检测工作中 [8, 10, 15, 80]，普便使用精度调整的方法来直接计算测试预言。使用精度调整的方法计算上述函数时，例如使用 `long double` 类型替换其中的 `double` 类型，将会使得代码中的魔术值 n 失去舍入的作用，从而导致函数的运算结果返回原始的 x 的值，进而导致测试预言的计算错误，影响误差检测的有效性。

为了解决语义解释错误导致的上述问题，得到正确且准确的测试预言，一个直接的方式是拓展已有的实数运算语义，使其支持解释浮点语义运算。例如，将上文中的例子解释为“将 x 舍入为整数”而不是“计算 $x+n-n$”。然而，直接拓展实数运算语义有以下几点困难：

- 第一，检测困难。找出所有的浮点语义运算很困难。由于浮点语义运算的形式并不固定，上文中所展示的样例只是一种浮点语义运算，其他类型的浮点语义运算可能包括加减特定的常数，对浮点数进行位运算等等。穷举所有的浮点语义运算类型并不现实；
- 第二，解释困难。即使找出了所有的浮点语义运算，将其解释为实数运算语义仍然十分困难。解释浮点语义运算需要理解代码背后的深层次含义，并将其对应到正确的实数语义上。例如，对于上文中的样例，需要将其解释为舍入操作而不是加减一个常数。这一步往往依赖专家知识（expertise），无法自动完成；
- 第三，定义困难。由于浮点语义运算的语义并非由语法直接决定，必须对浮点数程序中的相关变量进行数据流分析才可能得到正确的语义及解释。该方法的复杂性导致了显式的定义及解释其语义变得十分困难。

基于以上几点直接拓展实数语义的困难，本章提出了一种轻量级的兼容语义模型，可以同时支持实数运算语义及浮点运算语义，并基于该兼容语义模型构建自动化的测试预言生成工具 PsoHunter。本文发现，当使用精度调整的方法执行浮点数程序时，如果使用实数运算语义，其高低精度类型的运算结果将产生显著的偏移；而使用兼容语义模型进行精度调整时，高低精度类型的运算结果将会在统计意义上保持较小的偏移。基于上述发现，PsoHunter 可以对浮点数程序进行语义识别，并使用兼容语义模型，保证程序原始语义不被破坏，自动化的计算高质量的测试预言。

本章的主要贡献点如下：

- 提出了兼容语义模型，避免实数语义对浮点数程序的错误解释；

- 提出了基于兼容语义模型的测试预言生成工具 PsoHunter，自动生成高质量的测试预言；
- 实验评估结果表明了本章方法有效性；
- 自动生成的测试预言作为高质量的*搜索目标函数*，成为基于搜索的浮点数程序误差测试与分析的基石技术，解决浮点数误差的*未知性*问题。

本章的组织结构如下：第 2.2 节介绍了实数运算语义及兼容语义模型；第 2.3 节介绍了基于兼容语义模型的测试预言生成方法 PsoHunter；第 2.4 节介绍了 PsoHunter 对应的工具实现；第 2.5 节包含了实验设计以及实验结果分析；第 2.6 节对本章工作进行了总结和讨论。

2.2　实数运算语义与兼容语义模型

本节介绍了实数运算语义，以及基于实数运算语义的浮点数测试预言生成工具 FpDebug [10]。然后讨论浮点运算语义是如何影响此类工具，使其产生错误的语义解释，进而产生错误的测试预言。最后提出可以同时处理实数运算语义和浮点运算语义的兼容语义模型，成为测试预言生成工具的基础模型。

2.2.1　实数运算语义

*实数运算语义*的定义如下：将浮点数程序中的所有浮点数运算都解释为对应的实数运算的语义。大量已有工作都使用实数运算语义解释浮点数程序 [8, 10, 15, 80]。FpDebug [10] 即为其中的一个代表工作。本节将使用 FpDebug 作为样例，展示基于实数运算语义的方法如何对浮点数程序进行分析。

FpDebug 是一种基于*精度调整*的动态分析工具。对于浮点数程序的每一次执行，FpDebug 可以得到其高精度的运算结果，即测试预言。其他基于实数运算语义的工作与 FpDebug 具有类似的原理，因此本文选择 FpDebug 作为介绍样例。

FpDebug 在二进制代码级别对程序进行插桩，从而实现精度调整的功能。FpDebug 的技术细节如下：

浮点数变量

对于堆（heap）和栈（stack）上每一个可被浮点数变量访问的内存地址，FpDebug 自动的生成一个该内存地址对应的影子值（shadow value）。影子值通常是一个高精度的浮点数类型，例如由高精度浮点数库 MPFR [25] 提供。在实数运算语义的定义中，更高精度的浮点数总是更接近准确的结果（即实数运算的结果），因此影子值被用于表示精确的运算结果。

浮点数运算

对于每一个浮点数运算，包括一元运算 $u = \diamond v_1$（例如 $u = \sin v_1$）及二元运算 $u = v_1 \circ v_2$（例如 $u = v_1 + v_2$），FpDebug 首先查找 v_1 及 v_2 对应的影子值是否存在。若影子值存在，则使用对应的高精度影子值进行相同的运算，并将高精度结果储存在 u 对应的影子值中；若 v_1 或 v_2 影子值不存在，则使用对应的 v_1 或 v_2 的值对其影子值进行初始化，然后重复同样的高精度运算与储存。

分支处理

对于分支条件，原始值和影子值有可能导致不同的运算分支。为了保证影子值和原始值的同步运算及比较，FpDebug 总是选择进入原始值导向的运算分支。

通过上述方式处理浮点数变量与浮点数运算，FpDebug 即可获取最终运算结果的影子值，并认为结果的影子值即为整个浮点数程序的精确运算结果。可以看到，FpDebug 使用的是实数运算语义对浮点数运算进行理解。每个浮点数运算都被解释为实数运算，因而可以使用不同精度的浮点数类型进行运算。同理，每个浮点数类型也被解释为实数，因而可以使用不同精度的浮点数进行表示。上述样例即为实数运算语义的一个具体实现。

2.2.2 浮点运算语义导致的解释错误

本节将讨论浮点数程序与实数运算语义的不兼容性，即浮点运算语义是如何导致基于实数运算语义的工具对程序作出错误的解释。图 2.1 概略的展示了浮点运算语义导致的解释错误。

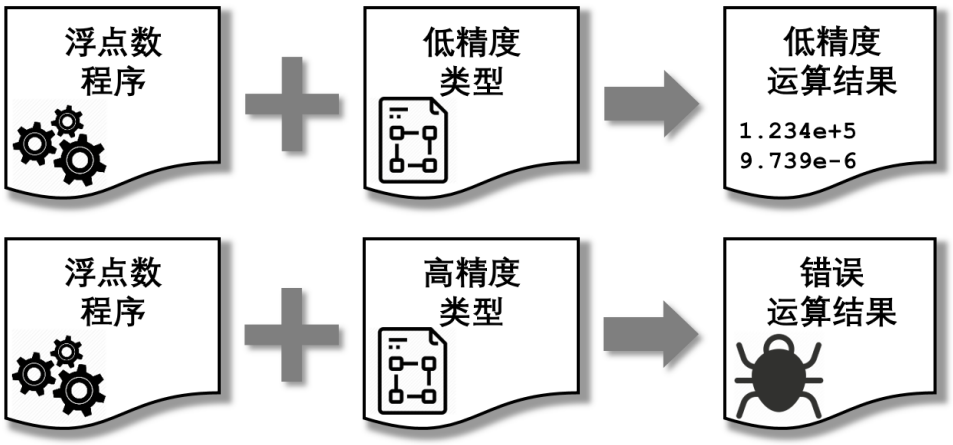


图 2.1 浮点运算语义导致的解释错误

本节将使用两个程序样例进行分析。第一个样例即为在第 2.1 节中讨论的程序。

```
1 double round(double x) {
2     double n = 6755399441055744.0; // n = 3<<51;
3     double y = (x + n) - n;
4     return y;
5 }
```

如第 2.1 节所讨论，该程序的目标是将输入 x 舍入至最接近的整数。然而，如果使用基于实数运算语义的工具对该程序进行解释，例如上文讨论的 FpDebug，将会得到错误的结果。FpDebug 会对变量 x 与变量 n 各创建一个高精度的影子值，并且使用影子值计算 $x+n-n$。由于高精度的影子值可以表示更多的精度，因此在加法结果中，x 的小数部分并不会丢失，也不会造成舍入，其运算结果将接与输入的 x 十分接近。与此同时，我们注意到，在 `double` 精度下执行时，结果将与开发者的期望一致，即在求和时由于精度不足而被舍入，最终得到 x 被舍入后的结果。

第二个样例同样是一种广泛存在于 glibc 中的浮点语义运算，该样例从 `log` 函数中简化而来：

```
1 union Double2Int {
2     int i[2];
3     double d;
4 };
5 double getSignificand(Double2Int x) {
6     x.d = calc();
7     x.i[HIGH_HALF] = (x.i[HIGH_HALF] & 0xFFFFF) | 0x40000000;
8     return x.d;
9 }
```

该函数的目标是获取浮点数 d 在下述科学记数法中的小数部分 μ：

$$\mu \cdot 2^n = |x.d| \qquad \mu \in [2,4), n \in \mathbb{N}$$

例如，当 $d = 1.2$ 时，函数将返回 2.4；当 $d = 37.6$ 时，函数将返回 2.35：

$$2.4 \times 2^{-1} = 1.2$$
$$2.35 \times 2^{4} = 37.6$$

上述代码的工作原理如下所述。`union` 类型的 `Double2Int` 可以获取浮点数在内存中的位表示。`double` 类型在内存中的表示有 64 位，`int` 类型有 32 位，因此用 `int[2]` 来获取 64 位的 `double` 类型所有位。其中，

- `x.i[LOW_HALF]` 表示浮点数 d 中较低的 32 位；
- `x.i[HIGH_HALF]` 表示浮点数 d 中较高的 32 位。

变量 LOW_HALF 与 HIGH_HALF 取 0 或 1，由不同架构中的大小端法决定。在上述代码的第 7 行中，通过和 0xFFFFF 进行“与”操作，可以保留 x.i[HIGH_HALF] 中的尾数位部分，并删除其中保存的指数位及符号位；通过和 0x40000000 进行“或”操作，可以设置符号位为 0（代表正数），指数位为 1（代表 2 的 1 次方）。得到的结果即为 μ 的值。

显然，上述所有操作都是特定于 64 位的 double 类型浮点数的浮点语义运算，将这些操作直接迁移（或映射）到其他精度上是不现实的，因而无法使用实数语义对其进行解释。如果想要获得正确语义下的映射，需要手动构建对应的位运算，建立浮点数至整数的映射，同时手动构建相应的常数，用于替换原始精度下使用的 0xFFFFF 和 0x40000000 。

当使用基于实数运算语义的工具对该程序进行解释，例如上文讨论的 FpDebug，同样会得到错误的结果。FpDebug 会生成一个浮点数 d 的影子值，然而，在上述代码第 7 行执行的并非浮点数运算，而是整数的位运算，因此高精度的影子值并不会同步执行相同的运算。因此，当函数结束时，原始值经过了第 7 行的运算（例如，由输入 37.6 运算成为 2.35），影子值却没有发生任何改变（例如，仍为 37.6）。即，代码第 7 行的语义为浮点运算语义，而当 FpDebug 使用实数运算语义对其进行解释时，会错误的将该运算映射至空运算而导致错误结果。

我们注意到，上述错误是非平凡的（non-trivial），错误并不是由于 FpDebug 自身的缺陷导致的。该错误产生的根本原因是实数运算语义的表达能力有限，无法表达浮点运算语义。而 FpDebug 基于实数运算语义，并不知道应该如何将上述运算映射至高精度影子值的运算，从而导致了该错误。

由本节的两个样例可以发现，实数运算语义在浮点数程序上的解释错误会产生严重的问题，进而导致使用相关的测试预言生成工具产生错误的结果。同时，由于理解浮点运算语义需要大量的专家知识，试图自动化的在实数运算语义模型下表达这些语义是困难且不现实的。

2.2.3 兼容语义模型

在本章第 2.2.1 节中已经讨论，实数运算语义将浮点数程序中的所有浮点数运算都解释为实数运算，并在精度调整时直接使用高精度类型对其进行映射及替换。然而，由于浮点数程序中存在特定的浮点运算语义，此类精度调整往往会导致实数语义作出的解释，进而得到错误的精度调整结果。本小节将提出一种可以同时处理实数运算语义与浮点运算语义的兼容语义模型，使得基于该模型的精度调整可以作出正确解释浮点数程序，成为测试预言生成工具的基础模型。

图 2.2 中概略的展示了兼容语义模型的特点。在兼容语义模型下，可以对浮点数

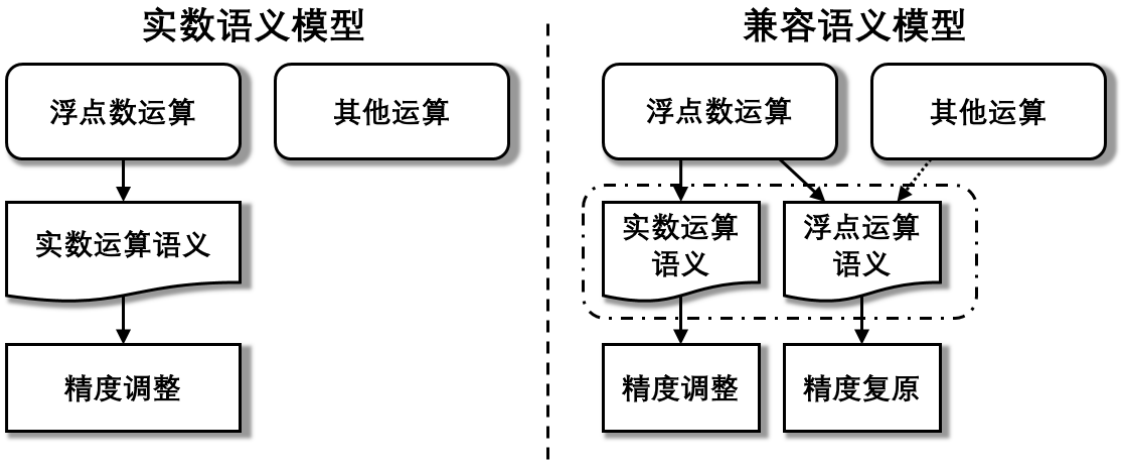


图 2.2　实数语义模型与兼容语义模型

运算进行分类：对于符合实数运算语义的运算，可以使用实数语义对其解释，并进行精度调整操作；对于不符合实数运算语义的运算，则使用浮点运算语义对其进行解释。在兼容语义模型下，运算与语义之间的对应关系如下：

- 实数运算语义用于解释大多数浮点数运算：对于浮点数程序中的一般性运算，都可以使用实数运算语义进行解释，例如 $a+b$，$cos(x)$ 等；
- 浮点运算语义用于解释少数浮点语义运算：对于浮点语义运算，其既可能为浮点数运算，也可能为其他运算（例如第 2.2.2 节中的第二个样例，即为整数的位运算）。

在基于兼容语义模型的精度调整下，需要对不同语义的运算进行分别处理。对于符合实数运算语义的运算，提高其类型精度总是可以得到更精确的运算结果，因此可以使用高精度类型进行执行；而对于符合浮点运算语义的运算，由于其语义在特定的浮点数精度下实现，因此在执行时需要确保其运算精度符合原始语义要求，即需要进行精度还原。

2.3　基于兼容语义模型的精度调整工具 PsoHunter

本节介绍本章的核心工作，首先是精度调整工具 PsoHunter 的技术概要，然后介绍精度调整，兼容语义模型下的语义检测，以及兼容语义模型下的精度调整方法。

2.3.1　技术概要

对于一个待分析的浮点数程序，本章的目标是使用兼容语义模型对其进行解释，使其在精度调整后（即使用高精度浮点数类型）可以得到更准确的运算结果，作为测试与分析中的测试预言。

本章基于兼容语义模型，提出了语义解释的划分及标记方法，形成测试预言生成工具 PsoHunter。图 2.3 展示了本章提出的基于兼容语义模型的语义检测与标记流程。对于给定的浮点数程序，PsoHunter 首先使用精度调整的方法对程序中的运算进行语义检测并划分，得到语义解释划分列表。然后对该列表中的运算进行标记，得到标记后的浮点数程序。在标记后的浮点数程序上，由于相关运算已被标记为实数语义或浮点语义，精度调整的方法将可以对运算作出正确的解释，进而可以直接使用高精度浮点数类型获得高精度的运算结果，即为测试预言。

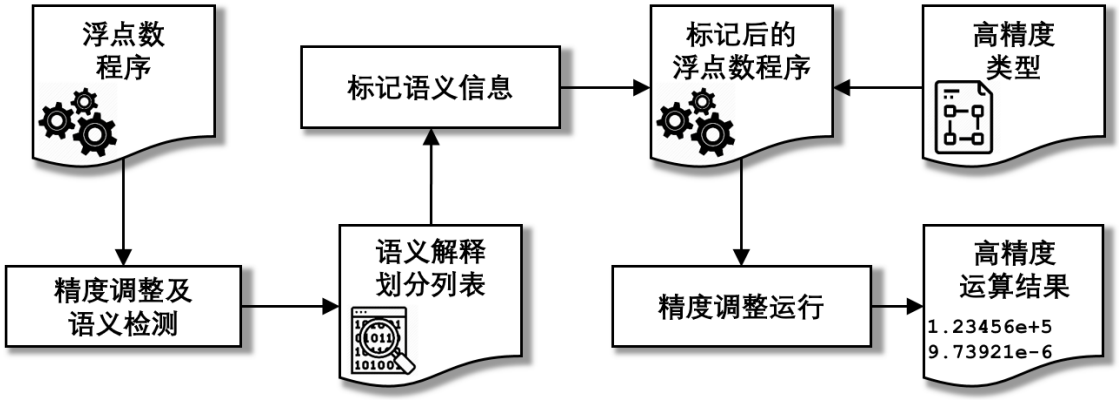


图 2.3　基于兼容语义模型的测试预言生成流程图

2.3.2　精度调整

精度调整的定义

本文提出的精度调整的定义并不依赖于特定的实数运算语义，因而可以用于不同的语义以及不同的工具中。精度调整可由如下规则定义：

- $\mathcal{L}$ 表示一个集合，由浮点数程序中所有可能包含的指令组成的集合；
- l 表示一个程序指令，形如 $l : \gamma = f(\theta_1, \theta_2, \ldots, \theta_n)$；
- γ 为 l 的输出值，$\gamma = out(l)$；
- $\theta_1, \theta_2, \ldots, \theta_n$ 为 l 的输入值，$\theta_1, \theta_2, \ldots, \theta_n = in(l)$；
- f 为一个函数，计算 l 对应的输入到输出的映射；
- $\phi : \mathcal{L} \to \mathcal{L}$ 为*精度调整函数*，对于任意的指令 l，精度调整函数会产生对应的高精度指令 l_h 作为输出，且满足 $\forall l \in \mathcal{L}, in(l) = in(\phi(l)), out(l) = out(\phi(l))$。

例如，对于第 2.2 节中提到的第一个样例，其第 3 行的计算 $(x + n) - n$ 的语句可以被分解为两个二进制级别的指令 `temp=x+n` 以及 `y=temp-n` ，因而精度调整函数会将其映射至高精度下的相同运算。对于同一节中的第二个样例，其第 7 行的语句同样会被分解为两个指令，由于其并非浮点数运算指令，所以精度调整函数会将其映射至空

指令，不进行任何高精度下的运算。

基于精度调整的误差计算

根据上述定义，便可以通过比较原始精度和高精度下的运算结果，对每一个变量所携带的误差进行计算：

- $\gamma_l = out(l)$ 为指令 l 的原始运算结果；
- $\gamma_h = out(\phi(l))$ 为指令 l 的高精度运算结果；
- ϵ 则为该运算结果中所携带的误差：

$$\epsilon_\gamma = \left| \frac{\gamma_h - \gamma_l}{\gamma_h} \right|$$

同理，对于输入的变量 θ，其携带的误差可以被表示为：

$$\epsilon_\theta = \left| \frac{\theta_h - \theta_l}{\theta_h} \right|$$

例如，对于上文提到的指令 `temp=x+n` ，输出变量 `temp` 中携带的误差可以通过比较在不同精度执行后的值得到；输入变量 `x` 携带的误差同样可以通过比较不同执行中的 `x` 得到。

2.3.3　兼容语义模型下的语义检测

在上一节中，本文已经讨论了浮点数程序的语义复杂性。由上述讨论可知，完全使用实数运算语义解释浮点数程序十分困难，需要依赖于专家知识和手动分析，因而在实践中不具备可行性。因此，本文提出了兼容语义模型，可以对浮点数程序的运算进行语义检测，区分其为实数语义还是浮点语义，进而正确的对程序进行解释和精度调整。本节将讨论如何对浮点数程序的运算进行语义划分。

本节提出的语义检测方法使用启发式规则，基于在精度调整下实数语义运算与浮点语义运算的不同表现，对其进行语义检测及划分。

给定两个阈值 ζ 和 η，使用与第 2.3.2 节中相同的符号表示，基于精度调整的语义识别规则定义如下：

定义 2.1 (一类浮点语义运算) *l 为一个运算，ϵ_θ 为在一次执行中 l 的输入变量 θ 所携带的最大误差，ϵ_γ 为在同一次执行中 l 的输出变量 γ 所携带的最大误差。如果 $\epsilon_\gamma/\epsilon_\theta > \zeta$ 的概率大于 η，则定义 l 为一类浮点语义运算。*

换言之，如果对于多次不同输入下的执行，误差都在 l 处被显著放大，则 l 很可能无法使用实数语义进行解释，是一个基于浮点语义的运算。

该启发式规则基于以下两点发现：

- 第一，浮点语义运算通常符合上述条件。由第 2.2 节中讨论可知，如果一个基于精度调整的工具产生了错误的结果，则其原因一定是实数运算语义对程序进行了错误的解释。同一节的两个样例已经展示了，浮点语义运算在被错误解释的情况下，精度调整的运算结果会与原始结果显著不同，因而会造成运算结果携带显著的误差。

  例如，在第一个样例中，程序的原始语义“将 x 舍入至整数”被错误解释为“返回 x”。在这种情况下，除非运算的输入值 x 本身是一个整数（对于浮点数而言概率极小），其他情况下都会导致高精度运算和原始运算结果之间产生误差。在第二个样例中，程序中第 7 行的位运算会被解释为一个空运算，导致 `temp` 的高精度影子值是一个未初始化的随机值，进而使得在绝大多数情况下，高精度结果与原始结果之间产生大的误差。
- 第二，实数语义运算通常不符合上述条件。相关工作 [8, 10] 指出，在一个普通浮点运算中产生误差只有两种可能：误差积累与近似相减（cancellation）。误差积累是指在一系列浮点运算中，误差在运算间不断积累的情况。即使每个运算可能只产生了较小的误差，积累误差却可能会是一个显著的误差；近似相减指的是当两个相近的数值相减，或两个符号相反而绝对值相近的数值相加时，会导致误差被显著放大。显然，误差积累导致的大误差要求一系列运算进行组合，在单一运算中不会出现。因此，唯一会在单一运算中导致大误差的实数语义运算是近似相减的情况。

  然而，发生近似相减的条件是两个操作数必须足够相近，并且其中至少有一个变量携带误差。在所有可能的输入空间中，满足条件的成对输入十分罕见。因此，若一个运算是实数语义运算，那么其不太可能在大量输入下都导致近似相减，产生显著误差。

在实践中本文发现，上述启发式识别规则的准确率很高，然而对于一类特殊的运算，该规则会产生假阳性的结果，即将实数语义运算识别为浮点语义运算。这类运算即为计算误差的运算。在基础运算库中，大量函数在运算的同时会计算误差，用于误差补偿，返回更准确的结果。给定一个运算 $\gamma = f(\theta_1, \theta_2)$，浮点数运算结果 $\hat{\gamma}$ 与理想的实数运算结果 γ 之间通常存在舍入误差。为了让运算结果更加准确，一个常用的误差补偿方法是增加一个额外运算 g 来计算 $\hat{\gamma}$ 携带的误差 ϵ_γ。

$$\epsilon_\gamma = g(\theta_1, \theta_2, \hat{\gamma})$$

虽然 γ 无法被浮点数准确的表达，但 ϵ_γ 却可以以浮点数的形式储存误差，因而可以用于在计算最终结果时进行误差补偿。例如，Dekker 提出了一种准确的求和算法 [20]，其

中就使用了上述方法储存运算的误差（称之为修正项）。

$$\gamma = \theta_1 + \theta_2$$
$$\epsilon_\gamma = \theta_1 - (\theta_1 + \theta_2) + \theta_2$$

此类计算误差并进行误差补偿的方法广泛应用于各类项目中，例如C语言数学标准库。

为了解决上述假阳性问题，本文进一步提出了一种过滤方法。本文注意到，当一个变量用于存储误差时，其绝对值通常较小，接近于零。利用这个特点，可以过滤假阳性的结果。给定三个阈值 σ_o，σ_h，和 τ，过滤条件的启发式规则定义如下：

定义 2.2 (二类浮点语义运算) *若 l 为一类浮点语义运算，且同时满足其原精度结果的绝对值大于 σ_o 且高精度结果的绝对值大于 σ_h 的概率大于 τ，则定义 l 为二类浮点语义运算。*

由于二类浮点语义运算在一定程度上解决了上述假阳性问题，是比二类浮点语义运算更准确的定义。在本节的后续部分，除特殊说明外，将使用二类浮点语义运算作为检测算法的结果。

基于上文提出的启发式规则，可以构建本文提出的语义检测算法。检测算法由以下几部分组成：

- 采样。首先使用采样算法生成大量的测试输入，采样点在可行的输入空间中遵循均匀分布的规则；
- 执行。使用原始精度和高精度分别执行生成的采样输入点。在每个浮点运算中，按照第 2.3.2 节中描述的精度调整方法计算其误差，记录误差及输入输出值；
- 过滤。当执行到某一个运算 l，发现其产生的误差大于定义 2.1 中的阈值 ζ 时，将其后续语句中所有的记录信息过滤。因为当 l 可能为浮点语义运算的情况下，后续的高精度运算结果可能产生错误，影响检测精度。
- 检测。根据记录的信息，按照定义 2.2 中的规则检测二类浮点语义运算。

在上述语义检测完成后，即可得到基于兼容语义模型的运算与语义类别的映射表。此映射表将记录浮点运算的语义类别，将其划分为需要实数语义解释或需要浮点语义解释。

2.3.4　兼容语义模型下的精度调整

本小节将讨论在对所有运算进行语义识别后，如何针对不同类别的语义进行解释及精度调整，进而得到正确且更精确的运算结果。本文提出的精度调整的核心为，在符合实数语义的运算上使用精度调整执行（即使用高精度变量同步执行）；在符合浮点语义的运算上，将其精度还原为原始精度，保证程序的语义正确。

在此类精度调整及精度还原的策略下，在程序中占绝大多数的实数语义运算将以更高的精度执行，进而得到更为精确的结果；而占少数的浮点语义运算将维持原始精度，避免出现语义解释错误导致的错误结果。因此，基于兼容语义模型的精度调整将比原始结果和基于实数语义模型的精度调整都更准确。

本文注意到，在特定情况下，开发者认为的具有浮点语义的运算可能与语义检测的标注略有不同，这是由于开发者需要更多的程序上下文信息（context）进行理解所导致的。例如，在第 2.2 节中介绍的第一个样例，包含了两个指令 `temp=x+n` 和 `y=temp-n`，通常开发者认为这两个运算都具有浮点运算的语义。然而，本文提出的语义检测算法只将第二个运算标记为浮点语义运算。接下来，本文将就语义检测的边界确定，以及不同边界对精度调整结果的影响进行讨论。

为了减少精度调整和精度还原的次数，本文采用起止点的方式标注连续的浮点语义运算。具体而言，本文引入了两个新的语句 `ps_begin` $(\theta_1,\ldots,\theta_n)$ 以及 `ps_end` $(\theta_1,\ldots,\theta_n)$，分别用于标记浮点语义运算的开始和结束。

手动标记起止点进行精度调整

开发者可以使用上述两个语句手动标记浮点语义运算的开始点和结束点，其中的变量 $\theta_1,\ldots,\theta_n$ 是运算中用到的变量，这些变量的精度会在开始点被还原为原始精度，在结束点重新调整至高精度类型。

自动标记起止点进行精度调整

由于现代程序的复杂性，上述手动标记方法需要大量的人工操作，在实践中并不是一个高效的方法。因此本文提出一种自动化的标记起止点的方法。在实践中本文发现，对于连续的浮点语义运算，在运算的全过程中还原精度与在最后一个运算中还原精度具有相同的效果。例如，对于第 2.2 节的第一个样例，如果在执行 `temp=x+n` 的指令时使用高精度，只在执行 `y=temp-n` 前还原精度，`temp` 的值仍然会被强制舍入，并可以得到正确的运算结果。由于上文提出的语义检测方法可以正确的检测到浮点语义运算的最后一个运算，在被检测到的语句前后加上 `ps_begin` 和 `ps_end` 同样可以得到正确的结果。因此，本章实现了上述的自动标记起止点的精度调整方法。在后续的实验中，我们发现，通过选取适合的阈值 ζ，上述自动标记的方法可以很好的精度调整效果。

2.4　工具实现

本章实现了基于兼容语义模型的精度调整工具 PsoHunter，PsoHunter 基于动态分析工具 FpDebug，可以自动生成高质量的测试预言。

PsoHunter 复用了 FpDebug 中的精度调整部分，并增加了额外的插入 `ps_begin` 和 `ps_end` 语句的功能，在运行时调整变量的精度，使其可以对于不同类别的运算进行精度调整及精度还原。同时，PsoHunter 加入了对于非浮点运算的支持，使其可以追踪浮点数变量的内存使用，即使使用 `union` 或指针对相关内存地址进行非浮点数运算，PsoHunter 仍然可以发现并同步进行相同的精度还原运算。

对于待分析的浮点数程序，PsoHunter 将：

- 首先按照第 2.3.3 节中的语义检测算法对程序中的运算语义进行标注，将运算划分为实数语义与浮点语义运算；
- 然后使用第 2.3.4 节中的精度调整算法对浮点语义运算进行标记，标记其起止点。

在精度调整的执行中，PsoHunter 遇到起止点标记则会自动进行精度的还原和提升，并报告高精度的运算结果，即为测试预言。无需重复进行语义检测及标记。换而言之，对于一个待分析的浮点数程序，PsoHunter 只需执行一次较为复杂的语义检测和标记的过程，后续每次测试预言的生成便可以通过单次执行标记后的程序完成。

2.5　实验评估

本节分为实验设计和实验结果两个部分。

2.5.1　实验设计

实验对象

本章的实验对象为 GNU C Library（glibc）中的函数。大多数 glibc 中的函数包括 `float`，`double`，以及 `long double` 三个不同精度类型的版本。其中，每一个特定精度类型的函数中的输入、运算、及输出三个步骤都使用相同的精度类型完成。对于绝大多数函数而言，其不同精度版本的具体实现都是相似的，因此本章只使用了其中类型为 `double` 的版本，共包含 48 个函数。这 48 个函数涵盖了：

- 三角函数，例如 sin，cos；
- 反三角函数，例如 arcsin，arctan；
- 双曲函数，例如 sinh，cosh；
- 对数函数，例如 log；

- 指数函数，例如 exp；
- 其他各类常用数学函数。

其中每个数学函数的输入为 1 至 3 个 `double` 类型的浮点数或整数输入，并且返回 `double` 类型的浮点数或整数作为输入[②]。

理想结果获取

本章使用高精度浮点运算库 MPFR [25] 以及任意精度计算器 [85] 计算实验对象的理想运算结果，以验证本章提出的工具 PsoHunter 的高精度修复效果。注意到，MPFR 以及任意精度计算器只支持有限的基础数学函数，包括了本章实验对象中的 21 个函数，而 PsoHunter 支持对任意的浮点数程序进行分析。因此，实验验证本章修复方法的有效性部分将在这 21 个支持的函数上进行。表 2.1 中列出了这 21 个函数，其中第一列为函数名，第二列为其数学表达式（用于 MPFR 以及任意精度计算器计算理想结果）。可以看到，这 21 个函数都是常用的数学函数。

表 2.1　glibc 中包含的数学函数

函数名	数学表达式	函数名	数学表达式
acos	$\arccos(x)$	exp10	10^x
acosh	$\ln(x+\sqrt{x^2-1})$	log	$\ln(x)$
asin	$\arcsin(x)$	log2	$\ln(x)/\ln(2)$
asinh	$\ln(x+\sqrt{x^2+1})$	log10	$\log(x)$
atan	$\arctan(x)$	log1p	$\ln(x+1)$
atan2	$\arctan(x/y)$	pow	x^y
atanh	$0.5\times\ln(\frac{1+x}{1-x})$	sin	$\sin(x)$
cos	$\cos(x)$	sinh	$0.5\times(e^x-e^{-x})$
cosh	$0.5\times(e^x+e^{-x})$	tan	$\tan(x)$
exp	e^x	tanh	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$
exp2	2^x		

参数选择

在第 2.3.3 节中介绍的启发式规则中，共包含 ζ，η，σ_o，σ_h，和 τ 五个参数。为了选取适合的参数，本文在 `exp` 函数上进行了前期实验，前期实验的结果表明，不同

② 本章的实验对象包括 `s_fmaf` 函数，其输入和输出都是 `float` 类型，但是其运算过程全部使用 `double` 类型完成，因此本章也将该函数纳入实验对象。

的 ζ 参数对于结果有较为显著的影响，而其他参数则对结果的影响不明显。因此，本章的实验将选取五种不同的 ζ 值进行实验，分别为 10^4，10^6，10^7，10^8，10^{10}，而对于其他的参数则设为固定值，分别为 $\eta = 0.7$，$\sigma_o = 10^{-9}$，$\sigma_h = 10^{-15}$，$\tau = 0.1$。

对于第 2.3.3 节中介绍的采样步骤，本章的实验设定为，根据函数的定义域大小不同，每个函数生成 1000 至 5000 个采样点。

实验步骤

本章实验的由以下几部分组成：

- 首先，在上述实验对象上，通过实验验证第 2.3.3 节中提出的语义检测算法的效果。同时通过实验探究不同的 ζ 参数对于语义检测效果的影响；
- 第二，在上述实验对象上，通过实验验证第 2.3.4 节中提出的自动标记起止点的精度调整算法的效果。通过与手动标记的结果进行比较验证自动修复算法的有效性；
- 第三，在上述实验对象上，通过实验验证本章提出的方法作为测试预言生成工具，与已有工具效果的比较。

2.5.2　实验结果

本小节将介绍实验评估的结果。本节内容与上述实验步骤相对应，分别为：

- 兼容语义模型下的语义检测算法的有效性；
- 兼容语义模型下的精度调整算法的有效性；
- PsoHunter 作为测试预言生成工具的有效性。

兼容语义模型下的语义检测算法的有效性

在 48 个实验对象函数中，本章提出的工具 PsoHunter 可以在 25 个函数中检测到浮点语义运算。表 2.2 中列出了包含浮点语义运算的函数以及其中浮点语义运算的个数，共检测到 153 个浮点语义运算。由于同一个内部函数可能被不同的实验对象重复调用，导致重复检测的情况，在进行去重处理之后，共有 48 个浮点语义运算。

该实验结果表明，浮点语义运算广泛存在于在 C 语言标准库中，其中超过一半的函数运算包含至少一个浮点语义运算。由于 C 语言标准库中的函数几乎是所有数值运算程序的基础，任何调用这些函数的浮点数程序都将受到浮点语义运算的影响。

对于不同的 ζ 参数，PsoHunter 识别为*潜在浮点语义运算*的数量也会有不同。对于识别到的潜在浮点语义运算，其有可能是真实的浮点语义运算，从而需要对其进行处理与修复；也可能是假阳性结果，即实际为实数语义运算。由于实数运算可以使用精度调整的方式执行，对其进行特定的修复可能会导致结果的精度损失（但不会导致

表 2.2　实验对象包含的浮点语义运算

函数名	个数	函数名	个数	函数名	个数
acos	1	acosh	1	asin	3
asinh	1	atan	2	atan2	4
cos	11	cosh	2	erf	2
erfc	2	exp10	2	exp2	1
exp	2	gamma	1	j0	17
j1	17	lgamma	1	log	1
pow	7	sin	15	sincos	22
sinh	2	tan	2	y0	17
y1	17				

错误的结果）。表 2.3 中展示了不同的 ζ 参数对于检测效果的影响，表中包括了不同参数下检测到的潜在浮点语义运算的个数，准确率，以及回调率。

表 2.3　不同阈值 ζ 对检测效果的影响

ζ 参数	10^4	10^6	10^7	10^8	10^{10}
一类浮点语义运算	142	130	129	121	113
准确率	33.80%	36.92%	36.43%	37.19%	36.28%
回调率	100%	100%	97.92%	93.75%	85.42%
二类浮点语义运算	78	64	61	54	48
准确率	61.54%	75.00%	77.05%	83.33%	85.41%
回调率	100%	100%	97.92%	93.75%	85.42%

在表 2.3 中可以看到，对于潜在浮点语义运算的识别结果，二类结果显著优于一类结果。二类浮点语义运算可以将准确率显著提升，同时没有任何回调率的损失。实验结果表明定义 2.2 （二类浮点语义运算的定义）中使用的过滤条件可以显著提升语义检测算法的效果。

在表 2.3 中可以看到，PsoHunter 的语义检测算法具有较高的准确率和回调率。在 ζ 选取合适的情况下，回调率可以达到 100%，意味着该检测算法有能力检测到所有的浮点语义运算。通过手动分析假阳性结果，本文发现，其产生的原因是由于一些分支较难被覆盖，通常只有几个采样点可以覆盖相应的运算。而在只有较少采样点的情况下，判断误差出现的概率是十分困难的，其通常不具备统计学意义上的显著性。同时，本文发现，不同的参数 ζ 对于结果有较为显著的影响，$\zeta = 10^6$ 时的实验结果完全优于

$\zeta = 10^4$ 的结果，二者具有相同的回调率，同时 $\zeta = 10^6$ 具有较高的准确率。

本文发现，对于浮点语义运算，假阳性结果会导致精度损失；而假阴性结果会有更严重的危害，其会因遗漏而产生错误的运算结果。因此，检测算法应保证较高的回调率。在下一节中，会进一步探究精度调整方法的有效性，以及不同语义检测的结果对于精度调整效果的影响。

兼容语义模型下的精度调整算法的有效性

为了探究 PsoHunter 的精度调整算法的有效性，本文在可以获取理想结果的 21 个函数上分别验证了精度调整前后的平均误差。表 2.4 中列出了 PsoHunter 的精度调整结果。表中同时包含不同 ζ 参数的检测结果对于精度调整效果的影响。表 2.4 不包含 $\zeta = 10^4$ 的结果，因为表 2.3 中的结果表明，$\zeta = 10^4$ 时的检测结果全面劣于 $\zeta = 10^6$。

根据上文分析，越低的准确率会导致越多假阳性运算被还原至原始精度执行，因而可能会导致结果的精度下降；越低的回调率会导致越多的假阴性运算（没有被发现的浮点语义运算）在高精度下执行，导致错误的结果。这两点因素都有可能对精度调整的效果产生影响。同时，如第 2.3.4 节中所讨论，浮点语义运算起止点的标记也可能对修复结果产生影响。

表 2.4 中将精度调整前平均误差大于 10^{-3} 的函数高亮显示。在表中可以看到，PsoHunter 精度调整后的平均误差显著低于修复前的平均误差。证明了 PsoHunter 修复算法的有效性。

根据表 2.4 可以发现，随着 ζ 的增大，检测准确率的提升对于精度调整效果的提升不明显。同时，随着 ζ 的增大，检测回调率的下降会对精度调整效果产生影响，例如 pow 函数。表 2.4 的结果基于第 2.3.4 节中的自动标记起止点的算法，实验结果表明，自动标记起止点的精度调整方法能有效的避免语义解释错误产生的影响，提高运算结果的准确性，是一种有效的精度调整方法。

PsoHunter 作为测试预言生成工具的有效性

已有工作 [8, 80] 使用精度调整的方法获取测试预言，对浮点数程序进行测试。然而，错误的测试预言会导致错误的测试结果。本章对上述论文中的结果进行检查，并使用 PsoHunter 作为测试预言生成工具，重新运行相关实验，本文发现了已有工作中的错误实验结果。

表 2.5 中结果显示，PsoHunter 作为测试预言生成工具，发现了上述工作之一 [80] 的三个实验结果为错误结果，并且其错误产生的原因是测试预言生成工具的缺陷。PsoHunter 发现了上述另一篇工作 [8] 包含一个错误结果。注意到，由于第二篇工作未在论文中汇报具体输入，无法直接对该结果进行验证。为了确认这个输入为错误输入，本

表 2.4　精度调整结果（误差越小越好），带有 * 标记的函数为包含浮点语义运算的函数

函数名	修复后平均误差				修复前平均误差
	$\zeta = 10^6$	$\zeta = 10^7$	$\zeta = 10^8$	$\zeta = 10^{10}$	
acos*	1.6106E-19	1.6106E-19	5.8999E-11	9.7265E-20	9.7265E-20
acosh*	1.0333E-17	1.0333E-17	1.0333E-17	1.0333E-17	5.5122E+04
asin*	2.0898E-19	2.0898E-19	2.0898E-19	2.0898E-19	2.0898E-19
asinh*	1.3004E-17	1.3004E-17	1.3004E-17	1.3004E-17	7.5351E+03
atan2*	1.3947E-18	1.3947E-18	1.3947E-18	1.3947E-18	1.3947E-18
atan*	4.0231E-19	4.0231E-19	4.0231E-19	4.0231E-19	4.0231E-19
atanh	3.3868E-17	3.3868E-17	3.3868E-17	3.3868E-17	3.3868E-17
cos*	6.0255E-19	2.3145E-21	2.3145E-21	2.3145E-21	2.5843E-03
cosh*	4.2901E-22	4.2901E-22	4.2901E-22	4.2901E-22	9.5417E-07
exp10*	8.4842E-23	8.4842E-23	8.4842E-23	8.4842E-23	1.2630E-06
exp2*	6.0300E-20	6.0300E-20	6.0300E-20	6.0300E-20	3.2491E-04
exp*	1.9083E-25	1.9083E-25	1.9083E-25	1.9083E-25	9.5402E-07
log10	8.1172E-18	8.1172E-18	8.1172E-18	8.1172E-18	8.1172E-18
log1p	1.2697E-17	1.2697E-17	1.2697E-17	1.2697E-17	1.2697E-17
log2	6.3510E-18	6.3510E-18	6.3510E-18	6.3510E-18	6.3510E-18
log*	8.6195E-21	8.6195E-21	8.6195E-21	8.6195E-21	1.0916E+01
pow*	2.5147E-16	2.5147E-16	1.6663E-08	1.6663E-08	8.8444E-07
sin*	7.7354E-21	7.7354E-21	7.7354E-21	7.7354E-21	4.2733E-02
sinh*	1.7909E-17	1.7909E-17	1.7672E-17	1.7672E-17	5.3316E-07
tan*	1.7135E-17	1.7135E-17	1.7135E-17	1.7135E-17	1.4335E+13
tanh	2.5412E-19	2.5412E-19	1.6890E-19	1.6890E-19	1.6890E-19

文对其进行了手动分析。结果表明，`equake` 函数调用了 `sin` 与 `cos` 函数，这两个函数都包含浮点语义运算，并且该工作使用了精度调整的方法获取测试预言，因而很可能汇报了错误的测试预言结果。进一步的，本文对该函数的进行了大量测试，发现其无法产生显著的误差，确认了上述结果是其使用的错误的测试预言所导致的。

总体而言，本章对已有的两篇误差检测工作进行分析，发现两篇工作中共具有四个由于测试预言错误导致的检测错误。由于测试预言的基础性和重要性，其他工作只要应用了错误测试预言生成工具，也很可能导致错误的结果。

本章实验结果表明，PsoHunter 可以作为高质量的测试预言生成工具，成为浮点数的测试与分析相关工作的前提和保证。

表 2.5　测试预言生成有效性

函数名	基于实数语义得到的测试预言	PsoHunter 生成的测试预言
exprel_2	2.85E+00	5.25E-12
synchrotron_1	5.35E-03	2.24E-13
synchrotron_2	3.67E-03	5.97E-15
equake	around 5.00E-01	n/a

2.6　讨论与小结

测试预言生成是浮点数程序测试与分析的重要一环，也是相关工作的前提和保证。由于浮点数程序语义的复杂性，已有测试预言生成工具无法对程序进行正确的解释，进而会产生错误的结果，影响后续的所有相关工作的质量和有效性。

本章提出了一种轻量级的兼容语义模型，以及基于此模型的高质量测试预言自动生成工具 PsoHunter，该工具可以自动对浮点数程序进行语义检测和精度调整，生成正确且准确的测试预言。实验结果表明，PsoHunter 的语义检测和精度调整算法是有效的，测试预言生成结果准确可靠。同时，PsoHunter 作为测试预言生成工具发现了已有工作中的错误结果。

本章提出的测试预言生成工具在本文中作为高质量的*搜索目标函数*，成为基于搜索的浮点数程序误差测试与分析的基石技术，解决浮点数误差的*未知性*问题。

第三章　基于搜索的浮点数黑盒测试用例生成技术

3.1　引言

浮点数测试用例生成是浮点数程序测试与分析中的关键步骤。由于浮点数误差问题的*稀疏性*特点，找到一个能触发显著误差的测试用例是十分困难的 [8]。然而，无论是开发者手动修复，还是自动化工具修复浮点数误差 [53, 77]，都首先依赖于一个能够触发显著误差的测试用例。因此，一个高质量的浮点数测试用例生成工具是十分重要且必要的。

由于控制浮点数误差的重要性，已有相关工作对浮点数误差进行测试和分析，然而，这些工作并未解决*稀疏性*问题。例如第 1.3.2.1 节中介绍的通过静态分析的方法来预估浮点数程序误差上界的工作。此类方法由于静态分析自身的局限性，计算得到的上界仅仅是实际误差的一个宽松的上界。即使在最新的研究成果中 [17]，计算出的上界仍然比实际的误差大几个数量级，在某些情况下甚至会得到“无穷大”这一没有意义的上界。这样过于宽松的上界限制了相关工作在实践中的应用：即使计算出了一个大的误差上界，该工作仍不确定浮点数误差问题是否真实存在于待测程序中。已有工作表明 [8]，只有极少数测试用例能够触发显著的浮点数误差。因此，手动构建触发误差的测试用例极其困难，进而导致此类误差上界分析的工作在实践中的应用受到很大的限制。

部分工作试图解决*稀疏性*的问题，生成会触发显著误差的测试用例。如第 1.3.2.2 节中介绍的相关工作 [15, 26]。此类工作直接将已有的搜索或采样方法应用于浮点数误差问题，例如二分搜索或马尔可夫蒙特卡洛方法（MCMC），以此来生成测试用例。然而，相关方法并未针对浮点数的数据结构特点以及误差的行为进行调整。实验表明，此类方法对误差的检测效率不高，其搜索方法无法发现有效的测试用例，依然受到*稀疏性*问题的制约。

为了提高在**黑盒**——即源码不可见——场景下的测试效果，本章对浮点数的数据结构特点和误差的表现进行了深入分析。本章发现，浮点数的数据结构的不同组成部分——符号位、指数位、及尾数位——对浮点数运算结果的误差造成的影响是截然不同的。不同组成部分通过如下方式影响误差的分布：

- 指数位通过其*值*的大小影响误差的分布。由于指数位的值是浮点数的值的主要决定因素，任何两个指数位相同的浮点数的值相差都在两倍以内。因此，只有当指数位取特定的值时，浮点数测试用例才可能进入导致显著误差的范围。
- 尾数位通过其*位*（*bit*）之间的组合影响误差的分布。由于浮点数使用“向偶数

舍入”的方法进行舍入。该方法会将任何数值舍入为最接近的可被表达的浮点数；当两个浮点数与待舍入数值一样接近的时候，则舍入至其中的偶数（在二进制中以零结尾浮点数）。这一舍入方式在数学期望上，可以使得产生的误差互相抵消，降低误差不断累积的可能性。在这种舍入模式下，只有当舍入的误差无法相互抵消时才会产生大的误差。因此，只有当尾数位的各个位之间以特殊方式组合时，舍入误差才无法相互抵消，进而产生显著的误差。

通过以上发现，本章提出了一种特定于浮点数误差问题的搜索算法 HierHybrid，并基于该算法提出了黑盒场景下的浮点数测试用例生成技术，用于检测显著的浮点数误差。

在最优化算法及搜索算法中，可以根据其在搜索过程针对个体和群体的不同分为两类算法，即基于个体的搜索算法和基于群体的搜索算法 [22]。基于个体的搜索算法从搜索空间的单一节点出发，尝试寻找最优解 [1, 72]。此类算法具有较强的邻域搜索能力，收敛速度较快，但是对于全局搜索的能力较弱，易陷入局部最优解。基于群体的搜索算法从搜索空间的多个节点出发，反复进行迭代，节点之间可以共享关于搜索空间的信息与知识 [36, 40]。基于群体的搜索算法具有较强的全局搜索能力，相对于个体搜索算法，此类算法收敛速度更慢。同时，由于其在搜索过程中侧重于全局的位置信息，对邻域的搜索能力较差，易收敛于近似最优解或次优解。

针对浮点数误差问题，为了充分利用浮点数类型的数据结构特点，本章提出了一种基于粒子群的层级混合搜索算法（Hier-Hybrid PSO，HierHybrid）。HierHybrid 算法采用两层搜索结构，分别对应处理浮点数结构中的指数位和尾数位的搜索策略。在第一层级的搜索中，目标为对浮点数空间进行全局的群体搜索，找到可能会触发显著误差的可疑指数位的数值；在第二层级的搜索中，目标为在可疑指数位数值下进行局部的个体搜索，找到会触发最大误差的尾数位的位向量，并将指数位与尾数位合并，最终找到会触发显著误差的浮点数测试用例。通过在不同层级中使用不同的搜索策略，HierHybrid 可以充分利用浮点数中不同的数据结构对误差的影响方式，大幅提高搜索算法对浮点数误差的检测能力与检测效果。

综上所述，本章提出的 HierHybrid 算法是一种（1）适应大搜索空间（2）适应稀疏搜索目标（3）搜索效果好的全局搜索算法。基于该算法，本章进一步形成了一种高效的检测浮点数误差问题的测试用例生成技术。

本章的主要贡献点如下：

- 进行了一项实证研究，探究浮点数中不同组成部分对于结果中误差的影响；
- 基于上述实证研究的结果提出了一种特定于浮点数误差问题的层级混合粒子群算法 HierHybrid；
- 用上述算法在黑盒场景下生成高质量的浮点数测试用例；

- 实验评估结果表明了本章方法的有效性；
- HierHybrid 在黑盒场景下，成为基于搜索的浮点数程序误差测试与分析的关键技术，解决浮点数误差的稀疏性问题。

本章的组织结构如下：第 3.2 节介绍了黑盒测试的应用场景和特点；第 3.3 节介绍了浮点数的结构特点以及不同结构对误差的影响方式；第 3.4 节介绍了本章提出的 HierHybrid 算法以及黑盒场景下测试用例生成的方法；第 3.5 节包含了实验设计以及实验结果分析；第 3.6 节对本章工作进行了总结和讨论。

3.2　黑盒测试场景及特点

一般而言，程序测试的方法可以按照其场景被分为黑盒测试（black-box testing）和白盒测试（white-box testing）[52]。黑盒测试又被称为功能测试，其主要关注程序的行为是否与预期或规约（specification）一致。在黑盒测试中，通常将待分析的程序看作一个整体：

- 不考虑程序的内部结构和内部特性；
- 不需要（或不具有）程序的源代码和专业知识（expertise）；
- 只关注程序的输入、输出、和程序功能。

黑盒测试——即源代码不可得——是程序测试中的常见场景。例如，待测试的程序包含或调用了编译后的二进制库（例如 jar 包，.so 类型的库等）。同时，在实践中，开发者或自动化工具也并非支持所有的编程预言，因而对于不被支持的源码，也只能进行黑盒测试。因此黑盒场景在实践中是广泛存在的一类测试场景。

对于本文面对的浮点数误差问题，黑盒测试的场景对应着浮点数程序源代码不可见的场景。在该场景下，本章讨论的测试与分析技术只考虑浮点数的输入、输出、及运算结果所携带的误差。

由于已有技术直接应用了其他黑盒测试的算法，而没有考虑浮点数误差问题的特点和性质，导致已有的工作在黑盒场景下的对浮点数误差的检测效果不好，无法发现潜在的误差问题。

为了提高黑盒场景下的测试效果，本章将对浮点数的数据结构特点和误差表现进行深入讨论，并基于浮点数误差特点，提出了一套基于搜索的浮点数黑盒测试用例生成技术，用于在黑盒场景下对浮点数误差问题进行测试。

3.3　浮点数结构及对误差的影响

本节将介绍浮点数的结构特点，以及讨论对于运算结果中的误差，浮点数中不同的结构可能对其产生不同方式的影响。然后通过一个具体的样例来讨论误差产生的特

点以及与浮点数结构之间的关系。

3.3.1 浮点数结构特点

浮点数（$\mathbb{F}$）表达了一系列不连续的有理数（$\mathbb{Q}$），任何实数（$\mathbb{R}$）都可以被舍入至最接近的浮点数进行表达。浮点数、有理数、与实数的关系如下列公式所述。

$$\mathbb{F} \subset \mathbb{Q} \subset \mathbb{R}$$

本文在第 1.2.1 节中已经对浮点数的结构进行了详细介绍。整体而言，浮点数的结构由符号位，指数位（幂数），与尾数位（有效数字）三部分组成，其表达的值由有效数字乘以幂数得到，类似于科学记数法的形式。对于一般情况，浮点数表达的数值由如下公式定义：

$$(-1)^S \times T \times 2^E$$

其中，S 表示符号位的值，E 表示指数位的值，T 表示尾数位的值。该样例简略的展示了浮点数结构的表达形式，具体的 E、T 的计算方法在第 1.2.1 节已有详细讨论，本节不再赘述。

常用的浮点数类型有 32 位的 `float` 类型和 64 位的 `double` 类型。这两种类型的浮点数结构如表 3.1 所示。

表 3.1 常用的浮点数结构

	符号位	指数位	尾数位
`float` 类型（32 位）	1	8	23
`double` 类型（64 位）	1	11	52

由上述讨论可以得到，浮点数中指数位和尾数位通过不同的方式影响浮点数的表达：

- 指数位中的数位决定了浮点数的值大小，如 `double` 类型的浮点数，只需要 11 位的指数位即可表达 10^{-308} 至 10^{308} 的范围；
- 尾数位中的数位决定了数值的精确表达，同样以 `double` 类型为例，52 位的尾数为保证了浮点数的表达最多具有 10^{-16} 级别的相对误差。

3.3.2 样例分析

为了进一步展示浮点数不同结构对于误差的影响，本章构造了一个始终返回常数的浮点数程序。该程序包括了常用的四则运算以及一个常见的程序范式——在循环中进行多次累加。该程序如下所述：

```
1  float foo(float x) {
2      int n = 8192;
3      float sum = n;
4      float fn = n;
5      float y, z;
6      x = fabs(x); // make sure x is a non-negative number.
7      for (int i = 0; i < n; i++)
8          sum += x;
9      y = sum / fn;
10     z = (x + 0.0625f) / (y - 0.9375f);
11     return z;
12 }
```

程序语义与运算结果

上述程序的输入为一个未知的浮点数 x，其语义如下所述：

- 变量 *sum* 被初始化为 n，其后在程序的第 7、8 行进行了 n 次循环，每次进行一个累加变量 x 的操作，故其最终结果应为 $sum = n + n \cdot |x|$；
- 变量 *fn* 即为浮点数类型的 n，其值为 $fn = n$；
- 变量 y 在第 9 行被计算，其值为 $y = sum/fn = 1 + |x|$；
- 变量 z 在第 10 行被计算，其值为

$$\begin{aligned} z &= (|x| + 0.0625)/(y - 0.9375) \\ &= (|x| + 0.0625)/(1 + |x| - 0.9375) \\ &= (|x| + 0.0625)/(|x| + 0.0625) \\ &= 1 \end{aligned}$$

由上述分析可知，该程序的返回值 z 应该始终为常数 1。然而，由于浮点数误差的存在，该程序并不保证在所有输入上都返回精确的结果。

导致误差的范围

在实验中本文发现，该程序误差最大时的返回结果为 1.0078125。同时，本文发现，上述样例程序只在非常小的范围会触发显著的误差。这是由如下原因导致的：

- 当 x 的绝对值较大时，该程序不会产生大的误差。例如 $|x| > 0.625$ 时（十倍于样例中的使用的常数），程序的最终运算结果将主要由 $|x|$ 决定，而误差将不再是结果的主要组成因素。实验结果表明，所有导致误差大于 0.001 的输入都集中在-0.4 至 0.4 之间；
- 当 x 的绝对值很小时，该程序同样不会可能会产生大的误差。这是由于程序只进行了 8192 次循环，一个非常小的 $|x|$，不足以使得变量 *sum* 中累积足够的误

差。

基于上述两点原因，发现会触发显著误差的测试用例是十分困难的。在实验中，大的误差仅出现在非常小的测试范围内：只有在测试用例大于 0.0004875 且小于 0.0004883 的范围内，该样例程序才可能产生大于 0.0078 的误差。

注意到，在上述区间 $[4.875 \times 10^{-4}, 4.883 \times 10^{-4}]$ 内的 `float` 类型浮点数只占所有 `float` 类型浮点数的 0.0012%，即约十万分之一左右。这一比例也反映了*稀疏性*问题带来的困难：在巨大的搜索空间中找到极少数会触发显著误差的测试用例是十分困难以及具有挑战的。

3.3.3 浮点数结构对运算误差的影响

通过上述样例本章发现，浮点数的不同结构不仅通过不同的方式影响浮点数的值（见第 3.3.1 节），也通过不同的方式影响浮点数程序的误差。具体而言，浮点数程序的误差受两个结构因素影响：

- 指数位的值的大小决定了误差产生与否。浮点数中，指数位的值是决定浮点数表达数值的主要因素。任何两个指数位相同的浮点数的值相差都在两倍以内。因此，只有当指数位去特定的值时，浮点数测试用例才可能进入导致显著误差的范围；
- 尾数位的位组合决定了误差的大小与分布。浮点数使用“向偶数舍入”这一舍入方式。这一舍入方式在数学期望上，可以使得产生的误差互相抵消，降低误差不断累积的可能性。在这种舍入模式下，只有当舍入的误差无法相互抵消时才会产生大的误差。因此，只有当尾数位的各个位之间以特殊方式组合时，舍入误差才无法相互抵消，进而产生显著的误差。

3.4 层级混合粒子群算法 HierHybrid 与测试用例生成

本节介绍本章的核心工作，包括技术概要，粒子群算法概述，实证研究，针对浮点数误差问题的层级混合粒子群算法 HierHybrid，以及应用 HierHybrid 算法进行浮点数测试用例生成。

3.4.1 技术概要

对于一个待分析的浮点数程序，本章的目标是找到一组测试用例，使得在该测试用例上，浮点数程序的运算结果具有显著的误差。本文将这个问题转化为搜索问题，搜索空间为待分析浮点数程序的所有的可行输入。

在本章讨论的黑盒应用场景下，搜索算法唯一可用的信息即为每个测试用例对应的误差。因此本章将使用误差作为搜索的目标。由于浮点数误差的稀疏性，仅有极少数的测试用例会触发显著的误差，因而搜索算法的效果至关重要。本章提出了一种特定于浮点数误差问题的分层局部搜索算法 HierHybrid。图 3.1 展示了本章的测试用例生成的流程。对于给定的浮点数程序，本章首先通过测试预言生成工具计算其对应的误差，然后基于已有的信息，使用搜索算法生成新的测试用例，目标是找到触发最大误差的测试用例。最终返回找到的最大误差以及对应的测试用例。

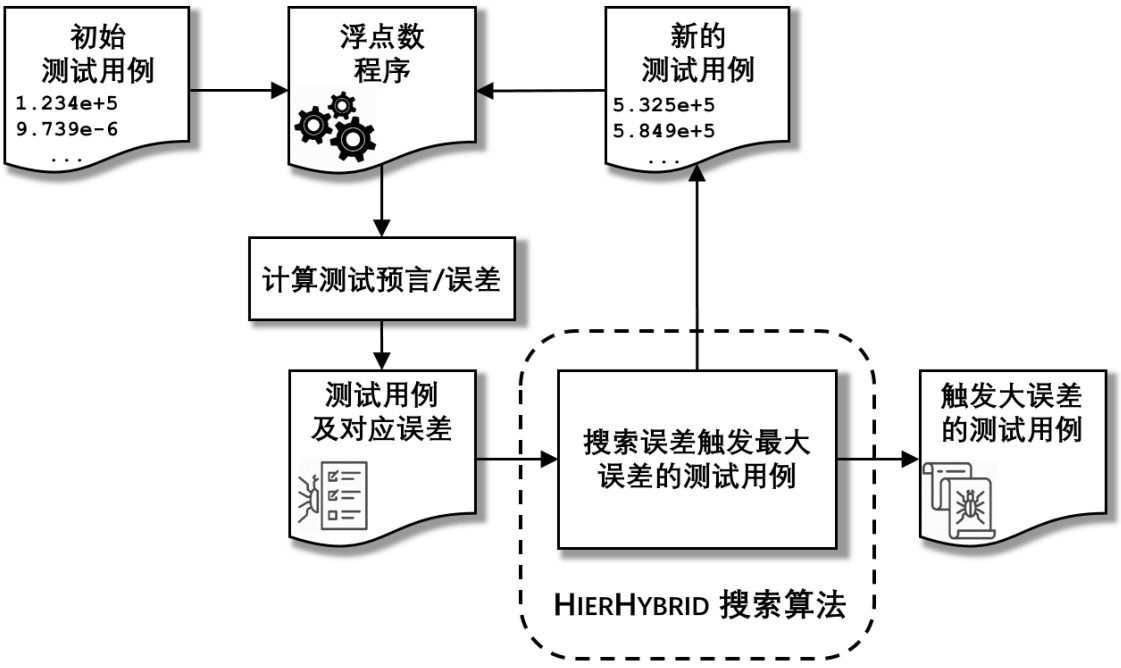


图 3.1　基于搜索的浮点数黑盒测试用例生成流程图

3.4.2　粒子群算法概述

粒子群算法是一种元启发式算法，其基于模拟自然界动物的运动特点而提出，通过群体的信息共享提供演化信息，进而解决最优化问题 [36]。粒子群算法的本质是一种基于群体的全局搜索方法，它能在搜索过程中自动获取和积累关于搜索空间的信息，并自适应的控制搜索过程，以求得全体最优解。

在粒子群算法中，最优化问题的解被称为个体，每一个个体都是搜索空间中的一个节点。个体需要特定的编码以成为计算机可以理解的格式或结构，通常个体以数值的形式进行编码。标准的粒子群算法包括数个核心定义和一系列关键步骤。

粒子群算法的核心定义

粒子群算法包括以下三个核心定义：

- 个体编码，即如何将每个个体进行编码。标准粒子群算法使用数值方法定义个体，其他方式包括向量、字符串等；
- 个体距离，即如何计算不同个体间的距离。粒子群算法中需要计算不同个体间的距离，进而计算个体在迭代中移动的速度。通常距离直接使用个体数值的差值定义，对于向量、字符串等表达，距离的定义也不尽相同；
- 适应度函数，即如何评价每个个体的优劣。通常直接定义为最优化的目标，可能涉及到多目标优化，目标数值化等问题。

粒子群算法的关键步骤

粒子群算法包括初始化、速度计算、群体更新、终止等几个关键步骤。

初始化 在初始化的过程中，会生成搜索空间的一组可行解，构成粒子群。对于一般的粒子群算法而言，粒子群的大小通常在几百到几千个个体之间。粒子群的生成方式可以是随机生成，也可以是基于先验知识、遵循概率分布，在更有可能发现最优解的搜索子空间内生成。

速度计算 在粒子群算法的每一次迭代中，粒子都需要更新自己的移动速度和方向。其运动速度 v_i 通常由如下方式定义：

$$v_i \leftarrow \omega v_i + \varphi_p r_p (P_i - x_i) + \varphi_g r_g (G - x_i)$$

其中，

- x_i 为第 i 个个体的当前位置；
- ω 为惯性权重，用于调节算法的搜索范围；
- P_i 为第 i 个个体的历史最佳位置；
- G 为群体（即所有个体）的历史最佳位置；
- φ_p 为个体移动权重，代表个体历史信息对于个体移动的影响；
- φ_g 为群体移动权重，代表群体整体信息对于个体移动的影响；
- r_p、r_g 为随机变量，增加算法的随机性与探索能力。

在速度计算的过程中，群体通过当前的全局最优点 G 进行信息交换，并且每一个个体都具备记忆信息 P_i。在这两方面信息的共同影响下，粒子群算法对于搜索空间具备较强的全局探索能力。

群体更新 在个体的速度计算完成后，粒子群算法对每一个个体的位置进行更新。

$$x_i \leftarrow x_i + v_i$$

同时，对新的个体计算其适应度函数 $f(x_i)$，并更新个体最佳 P_i 与群体最佳 G。

$$P_i = x_i, \quad \text{if} \quad f(x_i) > f(P_i)$$

$$G = x_i, \quad \text{if} \quad f(x_i) > f(G)$$

终止　在上述一系列过程中，粒子群中的个体经过速度计算和更新过程，在搜索空间中不断移动，并整体向着更好的适应度方向优化。在迭代过程中，更优秀的个体位置（P_i 和 G）始终引导着个体朝着更优方向移动。这种迭代一直进行到满足终止条件为止。一般而言，终止条件有以下几种：

- 迭代次数限制；
- 计算资源限制，如计算时间限制，计算内存限制等；
- 满足条件的最优解或近似最优解已被找到；
- 上述几种条件的组合。

3.4.3　实证研究

为了设计特定于浮点数误差问题的粒子群算法，用于黑盒测试用例生成，本节对浮点数误差在实际程序中的表现进行了实证研究。该实证研究在实际的浮点数程序中，探究在黑盒测试的场景下，浮点数不同结构如何对误差产生不同的影响，以此为基础支持本章后续的算法构建过程。

本文在第 3.3.3 节中，已经从讨论了浮点数的指数位和尾数位会对误差产生不同的影响，尚未讨论更为具体的影响方式和特征。因此，在本节将通过实证研究，更加具体的讨论浮点数指数位和尾数位对误差产生影响的方式，以及其特征与性质。

实证研究设定

在本实证研究中，本文从 GNU 科学计算库（GSL）中随机选取了 4 个函数，作为实验对象。GNU 科学计算库是一个开源的科学计算库，包含了大量常用的数值计算函数，例如贝塞尔函数（Bessel Functions），指数积分（Exponential Integrals），Γ 函数（Gamma Functions）等。GNU 科学计算库被广泛应用于各类浮点数程序中，也是相关浮点数误差分析文献的实验对象 [9, 68]。

为了简化实证分析的复杂度，本章随机选取了输入和输出都为 `double` 类型的浮点数的 4 个函数，分别为 `gsl_sf_bessel_K0` ，`gsl_sf_Ci` ，`gsl_sf_erf` ，和 `gsl_sf_legendre_Q1` 。其中：

- `gsl_sf_bessel_K0` 为第二类修正贝塞尔函数 K_α 的 0 阶形式。K_α 与第一类修

正贝塞尔函数 I_α 共同构成了下列常微分方程的一组线性无关的解系：

$$x^2\frac{d^2y}{dx^2}+x\frac{dy}{dx}-(x^2+\alpha^2)y=0$$

- `gsl_sf_Ci` 为三角积分中的余弦积分，其表达式为：

$$Ci(x)=-\int_x^\infty dt\cos(t)/t$$

- `gsl_sf_erf` 为高斯误差函数，在概率论，统计学，以及偏微分方程中都有广泛的应用，其表达式为：

$$erf(x)=\frac{2}{\sqrt{\pi}}\int_0^x dt\exp(-t^2)$$

- `gsl_sf_legendre_Q1` 为第二类勒让德多项式，其表达式为：

$$Q_1(x)=\frac{x}{2}\log\left(\frac{1+x}{1-x}\right)-1$$

为了探究浮点数中不同结构部分对于运算结果中误差的影响，本节采用控制变量法对指数位及尾数位分开讨论。

- 第一步中，探究指数位对结果中误差的影响。在这一步骤中，对于每个函数，首先随机生成 3 个不同的尾数位，然后遍历 `double` 类型下所有可能的指数位（共 $2^{11}=2048$ 个），组合成为浮点数测试用例，使用第二章中的测试预言生成工具，得到每个输入对应到的运算结果和误差。
- 第二步中，探究尾数位对结果中误差的影响。在这一步骤中，对于每个函数，首先随机生成 3 个不同的指数位。然而，遍历所有可能的尾数位是不现实的（共 $2^{52}\approx 4.5\times 10^{15}$ 个）。因此，随机生成 10000 个尾数位，将其与固定的指数位组合成为浮点数测试用例。同样使用第二章中的测试预言生成工具，得到每个输入对应到的运算结果和误差。

根据上述 4 个函数的定义域，所有测试输入的符号位都被设为 0，即表示非负数。

实证研究结果

本节将首先讨论固定尾数位的情况下，指数位对于运算结果误差的影响；然后讨论固定指数位的情况下，尾数位对于运算结果误差的影响。

指数位对运算结果误差的影响 图 3.2 中，展示了 `Bessel_K0` 函数中指数位对于运算结果误差的影响。在该图中，所有输入的尾数位固定为一个随机的值 0x657065816fd25。图中，x 轴为不同的指数位，y 轴为相对误差，取以 10 为底的对数之后的结果。根据图 3.2 的结果，以及在其他 3 个函数上的实验结果（具有相似特征，未附详图）可以得

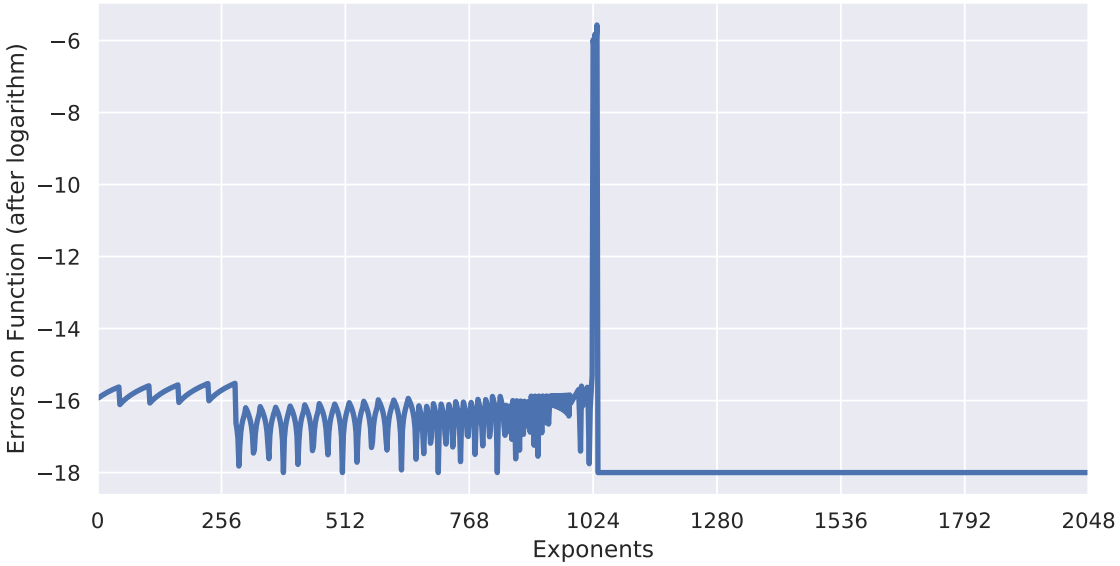


图 3.2　`Bessel_K0` 函数，指数位的影响（尾数位固定为 0x657065816fd25）

到，指数位对于运算结果误差有如下影响：

- 不同的指数位对运算结果的误差有显著影响。在所有 4 个实验对象函数中，不同的指数位对误差造成的影响最小为 3 个数量级，最大为 18 个数量级。即使考虑影响最小的 `gsl_sf_legendre_Q1` 函数，其最小的误差为 1.0×10^{-18}，最大的误差为 3.3×10^{-15}，相差 3000 倍以上。因此，不同的指数位对运算结果的误差影响很大；
- 指数位只在小区间内造成显著误差。该结果与第 3.3.3 节中的发现一致。由于指数位是决定浮点数值的主要因素，只有当其取特定的值时，测试用例才会进入触发显著误差的范围；
- 在所有可行的指数位中，造成显著误差的比例非常小。在图 3.2 中可以观察到，指数位只在取 1024 附近时产生了一个大的峰值。事实上，对于 `legendre_Q1` 函数，只有其指数位位于 1024 至 1030 之间时，造成的运算结果误差才会大于 10^{-10}，而对于 `gsl_sf_erf` 函数，这一区间则更小，仅为 1023 至 1024。上述结果表明，在所有可行的指数位中，只有极少数的指数位值才会导致显著的运算结果误差；
- 导致大误差的指数位位置很可能在所有可行值的中位数附近。对于 `double` 类型的浮点数而言，其指数位的中位数为 1023。通过分析上述 4 个函数，本章发现，该特性在这 4 个函数中都有所体现。其最大误差的区间分别为：
  - `gsl_sf_bessel_K0`：1024 ~1030
  - `gsl_sf_Ci`：1074 ~1152

- `gsl_sf_erf`：1023 ~1024
- `gsl_sf_legendre_Q1`：1023 ~1024

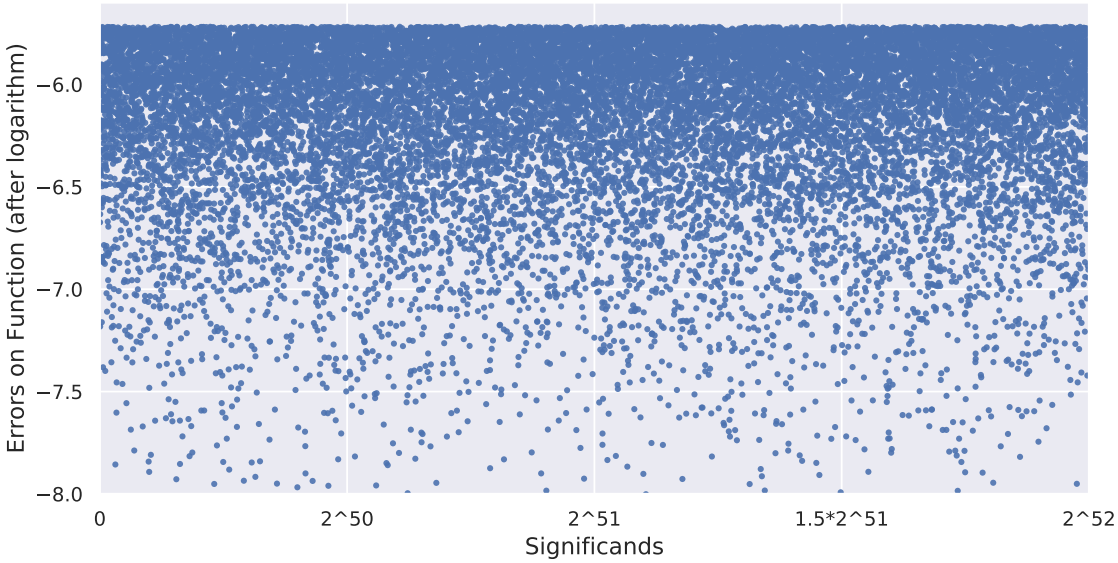


图 3.3　`Bessel_K0` 函数，尾数位的影响（指数位固定为 1030）

尾数位对运算结果误差的影响　图 3.3 中，展示了 `Bessel_K0` 函数中尾数位对于运算结果误差的影响。在该图中，所有输入的指数位固定为 1030。图中，x 轴为不同的尾数位，y 轴为相对误差，取以 10 为底的对数之后的结果。根据图中结果可以得到，尾数位对于运算结果误差有如下影响：

- 不同的尾数位对运算结果的误差有显著影响。在所有的 4 个实验对象函数中，不同的尾数位对误差造成的影响最小为 5 个数量级，最大为 9 个数量级。即使考虑影响最小的 `gsl_sf_bessel_K0` 函数，当指数位固定为 543 时，不同的尾数位导致的运算结果的误差最小为 10^{-19} 级别，最大为 10^{-14} 级别，相差十万倍以上。因此，不同的尾数位对运算结果的误差影响很大。
- 尾数位在整个可行空间上都可能造成显著的误差，并且分布均匀。该结论与第 3.3.3 节中的发现一致。由于浮点数使用的“向偶数舍入”方式，所有位组合中一个微小的变化都会导致舍入的相互抵消情况受到影响，进而对运算结果误差产生影响。这一现象导致尾数位与误差的关系分布是不连续的。
- 在所有可行的尾数位中，造成显著误差的比例很大。在图 3.3 中可以观察到，在整个 x 轴上都可能产生大于 10^{-6} 的误差，并且比较稠密。进一步分析数据可以得到，有 48% 的尾数位都可以产生大于 10^{-6} 的误差。

上述尾数位对运算结果的影响与上文讨论，以及已有的相关研究 [8] 是一致的：虽然在固定指数位时，很大比例的尾数位都可能触发显著误差，但误差必须要求指数位与尾

数位同时触发大的误差，因而随机输入产生显著误差的概率仍然非常小。

综上所述，实证研究的结果表明，在黑盒测试场景下，针对浮点数误差问题的搜索过程，关键在于找到触发大误差的指数位的值。这是由于指数位和尾数位导致显著误差的概率具有明显差异所导致的。在下一节，本章将讨论，利用本节的发现，有针对性的设计一种高效的层级混合粒子群算法 HierHybrid。

3.4.4　针对浮点数误差问题的粒子群算法 HierHybrid

本节基于第 3.4.3 节中的发现，构建适应浮点数黑盒测试场景的粒子群算法 HierHybrid，用于生成会触发显著浮点数误差的测试用例。

根据第 3.4.2 节中对粒子群算法的概述，本节从构建粒子群算法涉及的核心定义和一系列关键步骤的角度，对本文提出的层级混合粒子群算法 HierHybrid 进行描述。

HierHybrid 算法中的核心定义

层级混合粒子群算法 HierHybrid 涉及的核心定义包括个体编码，个体距离，适应度函数三个定义。

个体编码　对于浮点数的结构，在第 3.3 节中已有讨论。浮点数由符号位，指数位，和尾数位三部分组成。浮点数的不同结构组成部分，在黑盒测试的场景下，对于运算结果的误差也会产生不同的影响。因此，HierHybrid 算法针对三个结构部分的特点，使用不同的编码形式进行表达。

- 符号位使用*位*表达。由于符号位只有 1 个位（bit），使用数值类型和位类型没有区别。
- 指数位使用*数值*表达。指数位是决定浮点数大小的核心因素，大的浮点数一定有大的指数位，反之亦然。在第 3.4.3 节的实证研究中表明，在黑盒测试的场景下，运算结果的误差与指数位的大小关系具有连续性。因此使用数值表达，符合实证研究中指数位与结果误差关系的特性。
- 尾数位使用*位数组*表达。首先，尾数位不是决定浮点数大小的核心因素。无论怎样修改尾数位，浮点数的大小相差都在两倍以内。尾数位通过影响舍入误差的积累来影响结果中的误差。根据第 3.4.3 节的实证研究结果，尾数位对误差的影响不具有连续性，因此不适合使用数值类型表达。同时，显著误差可能是由于尾数位中特定的位组合导致的，因此使用位数组表达更符合实证研究中的结论。

具体而言，每个浮点数个体 x 可被表达为如下形式：

$$x = \langle sign^b, expo^n, frac^b \rangle$$

其中 $sign^b$ 为符号位，$expo^n$ 为指数位的数值，$frac^b$ 为尾数位的位数组。

个体距离 在下文的算法步骤中将讨论，HierHybrid 算法分为两个层级，且在不同层级拥有不同的搜索模式和搜索策略，这也导致在不同层级中对于个体距离的定义不同。

在第一层级中，搜索策略侧重于对指数位的搜索，个体间的距离由指数位（主要）和尾数位（次要）共同决定：

$$dist_1(x_1, x_2) = \psi_e \left| (-1)^{sign_1^b} expo_1^n - (-1)^{sign_2^b} expo_2^n \right| + \psi_f \mathcal{L}(frac_1^b, frac_2^b)$$

其中，$\mathcal{L}$ 表示向量间的编辑距离，即 Levenshtein 距离 [78]。ψ_e 与 ψ_f 为距离权重，在第一层级中侧重于指数位的探索，因此应满足 $\psi_e \gg \psi_f$。

在第二层级中，搜索策略侧重于对可疑个体的邻域进行搜索，个体间的距离由浮点数的数值和尾数位共同决定：

$$dist_2(x_1, x_2) = \chi_x \left| \frac{x_1 - x_2}{\max(expo_1^n, expo_2^n)} \right| + \chi_f \mathcal{L}(frac_1^b, frac_2^b)$$

第二层级侧重于个体的邻域搜索。在上述公式中，第一项的目标为得到归一化的浮点数数值距离，第二项的目标为得到浮点数尾数位的编辑距离。浮点数的邻域既由其数值决定，又需要考虑尾数位的位组合关系。因此上述公式中的 χ_x 和 χ_f 参数用于对数值距离和尾数位编辑距离分别进行权重调整。

适应度函数 常用的衡量运算结果误差的适应度函数包括相对误差与绝对误差。由于在科学运算及数值运算中，运算结果的大小可能具有显著的差别，此时相对误差具有更直观的性质。即，在运算结果为 10000 时误差为 0.001 的情况（相对误差 10^{-7}）比运算结果为 0.01 时误差为 0.001 的情况（相对误差 10^{-1}）要精确的多。而上述两种情况的绝对误差都为 0.001，使用绝对误差便无法区分两次运算的精确程度。

本章使用相对误差作为适应度函数，即 HierHybrid 算法的搜索目标是找到会触发最大相对误差的测试用例。HierHybrid 可以使用任何自动化的测试用例生成工具来计算相对误差（即适应度函数），例如本文第二章中提出的工具 PsoHunter。

HierHybrid 算法的关键步骤

粒子群算法 HierHybrid 涉及的一系列关键步骤如图 3.4 所示。本小节将首先对整体流程进行概述，然后对每个关键步骤进行详细描述。

在初始化阶段，HierHybrid 会按照上文所述的个体编码随机生成一系列的符号位、指数位、及尾数位。所有指数位按照均匀分布在指数位的可行范围内（对于 `double` 类型的浮点数即为 0~2046），并且在中位数附近增加生成概率。所有尾数位则为完全均

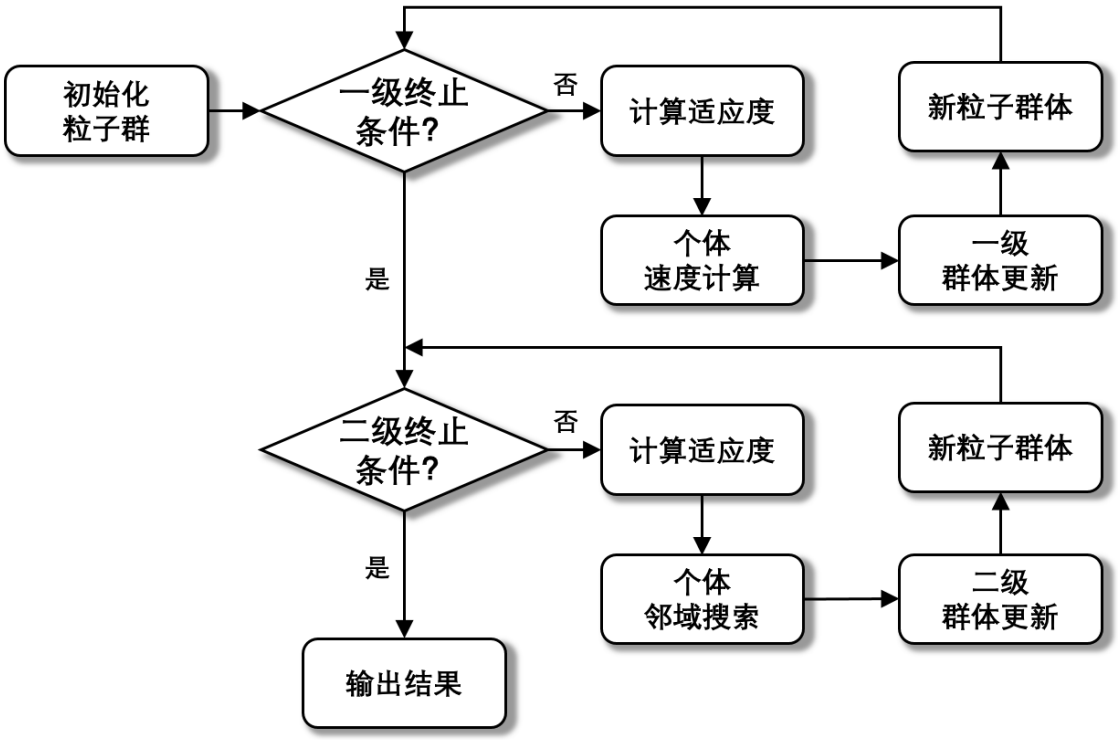


图 3.4　层级混合粒子群算法 HierHybrid 的流程图

匀分布在尾数位可行空间中的随机样本。之后，将按照上述方式生成的三类结构组合成为浮点数测试用例，加上随机生成的移动速度，作为初始化的粒子群。

第一层级　在第一层级中，搜索策略为找到可能触发显著误差的区域。具体而言，搜索算法不断变化个体的指数位，搜索到一批可疑的个体及其指数位。

在第一层级的速度计算步骤中，本文基于上文提出的第一层级距离 $dist_1$，计算个体的移动速度。在传统的粒子群算法中，速度计算方案为惯性权重 ω，个体移动权重 ϕ_p，群体移动权重 ϕ_g 三者共同决定粒子的移动速度 [36]。然而，在浮点数误差问题中，由于在绝大多数的搜索空间中，其误差都为一个非常小的值 [8]，因此个体的历史最优个体 P_i 很可能对应一个无意义的小误差，并不能对搜索带来帮助，反而可能导致个体在小误差区域停留，导致收敛速度降低。为了解决上述问题，本文在第一层级中使用了个体移动权重为零 $\varphi_p = 0$ 的速度计算模型，使得个体更快的向全局最优区域移动，提高收敛速度，增强算法的全局搜索能力。

在第一层级的群体更新步骤中，本文并未遵循传统粒子群算法中的一对一更新——即一个个体总是在更新后严格的对应一个新的个体。对于浮点数误差问题，其指数位所在的位置是误差的决定性因素，而对于大多数指数位而言其误差都非常小。此类特点会导致大多数个体在较劣区域停留过长时间，降低收敛速率。因此本文提出了一种

基于分裂和淘汰的更新算法，应用于 HierHybrid 的群体更新步骤中，进而加速第一层级的收敛速度。

在第一层级的终止条件中，HierHybrid 使用迭代次数或运行时间决定算法的停止时机。当达到预定的迭代次数或运行时间后，HierHybrid 将会保存当前群体信息，进入第二层级的搜索中。

第二层级 在第二层级中，搜索策略为对每个可疑个体的邻域进行局部搜索。具体而言，搜索算法不断变化个体的尾数位（仅在必要情况下变化指数位），搜索会触发最大误差的浮点数测试用例。

在第二层级的邻域搜索步骤中，需要对每个个体的邻域进行局部搜索，找到会触发最大误差的尾数位。因为对于个体而言，不同指数位下的尾数位之间并无关联。因此在探索每个个体的邻域时，无需考虑其他个体的尾数位信息。基于上述讨论，本文在第二层级的邻域搜索中，使用了群体移动权重为零 $\varphi_g = 0$ 的速度模型，增强每个个体对于其邻域的搜索能力。

在第二层级的群体更新步骤中，由于尾数位的表达为位数组的形式，无法与速度直接相加得到新的个体。为了解决上述问题，HierHybrid 推广了粒子群算法中速度的定义。在粒子群算法中，速度为个体移动的距离。而对于不可相加的个体与速度，可以将其速度推广为个体移动的范围，即探索的邻域大小。对于具有较大速度的个体，新的个体位于其较大范围的邻域内，反之亦然。因此，根据上述概念，HierHybrid 算法只需要个体间的距离具有良好的定义，即可基于邻域大小生成新的个体。

在第二层级的终止条件中，HierHybrid 同样使用迭代次数或运行时间决定算法的停止时机。当达到预定的迭代次数或运行时间后，HierHybrid 将会返回找到的最大误差以及对应的测试用例。

下面将具体介绍 HierHybrid 的各个关键步骤，包括初始化，第一层级的各关键步骤，以及第二层级各关键步骤。

初始化粒子群 在初始化的过程中将产生 n 个测试用例作为初始粒子群 X。这 n 个测试用例的符号位和尾数位将通过完全均匀分布的方式随机产生，而指数位则遵循一定的特定条件生成。

在生成 n 个指数位的步骤中，遵循如下定义：

- λ 为所有指数位的中位数，即 `double` 类型的浮点数输入，$\lambda = 1023$；对于 `float` 类型的浮点数输入，$\lambda = 127$；
- k 为指数位的位数，对于 `double` 类型，$k = 11$；对于 `float` 类型，$k = 8$；

- δ 为中心搜索区域的参数，$\delta = 2^{\lceil k/2 \rceil}$。即对于 `double` 类型，$\delta = 64$；对于 `float` 类型，$\delta = 16$。
- σ 为中心搜索区域概率提升的倍数。在 $[\lambda - \delta, \lambda + \delta]$ 的区间内，生成指数位的概率为其他部分的 σ 倍。

同时，对于每个初始粒子，其将被分配一个服从正态分布的随机变量 $v_i \sim \mathcal{N}(0, \upsilon)$ 作为其初始移动速度。

第一层级的速度计算　在第一层级的速度计算中，其原则是根据个体与（当前）全局最优个体间的距离计算其速度，使其向全局最优个体移动，探索更可能出现最优个体的区域。

在上文中已经讨论，HierHybrid 在第一层级中使用了个体移动权重为零 $\varphi_p = 0$ 的速度计算模型：

$$v_i \leftarrow \omega v_i + \varphi_g r_g dist_1(G, x_i)$$

在此模型下，个体的移动只与全局最优个体的距离相关，不同个体间通过全局最优个体交换搜索空间信息，使得个体更快的向全局最优区域移动，提高收敛速度，增强算法的全局搜索能力。

第一层级的群体更新　在第一层级的群体更新中，核心原则为尽可能快的找到会触发显著误差的区域。传统的粒子群算法在群体更新时遵循一对一的原则，即每个个体对应与更新后的一个个体：

$$x_i.expo^n \leftarrow x_i.expo^n + v_i, \quad \text{for} \quad x_i \in X$$

在上文中已经讨论，HierHybrid 提出了一种基于分裂和淘汰的更新算法，用于第一层级的群体更新过程：

首先，对粒子群中的个体 x_i，按照其相对误差 $f(x_i)$ 进行进行排序，保证 $\forall p < q, f(x_p) \geq f(x_q)$。

然后对群体的后 $\alpha\%$ 的个体进行淘汰，同时对群体的前 $\alpha\%$ 的个体进行分裂，保持粒子群的总量不变。在分裂的过程中，待分裂的每个个体都将变为两个个体，其中一个保留当前的位置和速度，另一个将被分配一个随机的速度并立刻进行移动：

$$\begin{cases} x_{i1}.expo^n = x_i.expo^n, & v_{i1} = v_i \\ x_{i2}.expo^n = x_i.expo^n + v_{i2}, & v_{i2} \sim \mathcal{N}(0, \beta) \end{cases}$$

第一层级的终止条件 第一层级的终止方式可以根据需求定义为迭代次数或运行时间。

在第一层级的终止条件中，并不要求所有粒子收敛于同一个区域。只需要粒子在停留在数个较优区域即可进入第二层级的局部搜索。为了最大化搜索的效率，在较短时间内对搜索空间进行探索，HierHybrid 在第一层级中采用了大群体，少迭代的模式。

第二层级的邻域搜索 在第二层级的邻域搜索步骤中，其原则是对已有个体的邻域进行探索，试图找到会触发更大误差的尾数位。根据上文讨论，HierHybrid 在第二层级中使用了群体权重为零 $\varphi_g = 0$ 的速度计算模型：

$$v_i \leftarrow \omega v_i + \varphi_p r_p dist_2(P_i, x_i)$$

在此模型下，个体的速度只与个体的历史信息相关，不同个体间不再进行信息交换，而专注于局部的邻域搜索，增强了邻域搜索能力。

第二层级的群体更新 在第二层级的群体更新步骤中，由于尾数位的表达限制，其速度无法直接与个体相加得到更新后的个体。根据上文讨论，本文提出了一种广义的速度概念，将速度 v_i 定义为搜索邻域的大小。而对于更新后的个体，只需满足其与原个体的距离与速度近似即可：

$$dist_2(x_i^{new}, x_i^{now}) \simeq v_i$$

为了得到满足条件的 x_i^{new}，本文使用了随机梯度下降法 [11] 对 x_i^{new} 进行搜索。注意到，由于此子步骤中对于 x_i^{new} 的搜索只涉及到两个浮点数之间的距离计算，而不需要计算测试预言 $f(x_i^{new})$ 和 $f(x_i^{now})$，因而其运算开销很小，不会对 HierHybrid 的整体运行开销造成显著影响。

第二层级的终止条件 第二层级的终止方式同样可以根据需求定义为迭代次数或运行时间。

第二层级的终止条件也是 HierHybrid 算法整体的终止条件。本文将第二层级的迭代次数或运行时间设置为与第一阶段相同，即运算资源将被平均分配在 HierHybrid 的两个层级中。

3.4.5 应用 HierHybrid 进行测试用例生成

本章将黑盒场景下的浮点数误差检测问题转化为基于搜索的测试用例生成问题。在黑盒场景下，给定待测试的浮点数程序 $\mathbb{F} : \hat{f}(x)$ 和测试预言生成工具 $\mathbb{O} : f(x)$，本章

的测试目标为生成测试用例 t，最大化程序的误差 $\varepsilon(t)$：

$$\arg\max_{t \in T} \left(\varepsilon(t) = \left| \frac{\hat{f}(t) - f(t)}{f(t)} \right| \right)$$

为了完成上述目标，可以使用任意的搜索算法对输入空间 T 进行搜索。然而，传统的优化算法以及启发式算法对于浮点数误差的搜索能力不强，搜索结果较差。因此，本章在第 3.4.4 节中提出了针对浮点数误差问题的层级混合粒子群算法 HierHybrid。

HierHybrid 算法可以在黑盒场景下对浮点数输入空间进行高效的全局搜索，返回搜索到的最大误差 ε_{max} 以及对应的测试用例 t_{max}，可直接用于后续的调试及修复过程。

3.5　实验评估

本节分为实验设计和实验结果两个部分。

3.5.1　实验设计

实验对象

本章的实验对象包括两部分：

- 6 个经典浮点数程序，已有工作详细讨论这些函数是否会导致显著误差 [68]；
- 一系列来自于 GNU 科学计算库（GNU Scientific Library）中的函数。GNU 科学计算库是一个开源的科学计算库，包含了大量常用的数值计算函数，例如贝塞尔函数（Bessel Functions），指数积分（Exponential Integrals），Γ 函数（Gamma Functions）等。GNU 科学计算库被广泛应用于各类浮点数程序中，也是相关浮点数误差分析文献的实验对象 [9, 68]。

在 GNU 科学计算库包含的所有函数中，本章选取了其中输入和输出都为浮点数类型的共 88 个函数作为本章的实验对象。

理想结果获取

本章使用了第二章中提出的测试预言生成工具 PsoHunter 计算实验对象的精确运算结果，将 PsoHunter 得到的精确结果和实验对象的原始结果进行对比，即可得到运算结果中携带的误差。

同时，本章对实验结果同步使用 mpmath [87] 计算精确结果，进一步降低实验的外部风险（由于潜在的测试预言错误导致实验结果错误的风险），保证实验结果的可靠性。mpmath 是基于任意精度的浮点数计算库 MPFR [25] 之上开发的一个任意精度的运算库，其对 MPFR 进行了进一步包装，支持了更多的科学运算函数。

注意到，测试预言生成工具 PsoHunter 支持任意的浮点数程序，而 mpmath 只支持特定的程序（包括了本章的实验对象）。若直接使用 mpmath 作为测试预言生成工具，实验结果将不具备可推广性（因为实践中存在大量程序不被 mpmath 支持）。本章提出的工具 HierHybrid 并不限于特定的程序，因而将采用 PsoHunter 作为测试预言生成工具，同时只使用 mpmath 作为有效性参考。

参数选择

在第 3.4.4 节中，定义了 HierHybrid 算法的一系列参数。

在针对浮点数的个体距离中，本文设定为 $\psi_e = 1$，$\psi_f = 0.01$，$\chi_x = 1$，$\chi_f = 0.01$。

粒子群算法中参数的设置参考了已有工作中的设定 [35, 79]。具体而言，相关设定为粒子群大小 $n = 100$，中心搜索区域权重 $\sigma = 5$。在第一层级与第二层级的速度计算中，惯性权重 $\omega = 1$，个体权重 $\varphi_p = 2$，群体权重 $\varphi_g = 2$。

其他参数在第 3.4.4 节中已有设定和讨论，此处不再赘述。

实验步骤

本章实验由以下几部分组成：

- 第一部分为在已知误差特性的 6 个实验对象上，通过实验验证 HierHybrid 作为测试用例生成工具，检测误差的效果。同时，已有工作 BGRT [15] 是一种基于二分区间搜索的浮点数误差检测算法，该部分将对比 HierHybrid 与 BGRT 在上述实验对象上的误差检测效果；
- 第二部分为在 GNU 科学计算库的实验对象上，通过实验验证 HierHybrid 与传统的粒子群算法（PSO）在检测浮点数误差上的效果对比，验证本文提出的层级混合粒子群算法的有效性；
- 第三部分为在 GNU 科学计算库的实验对象上，通过实验验证 HierHybrid 与另一已有的针对浮点数误差的遗传算法 [80] 工作进行比较，进一步验证本文算法的有效性。

3.5.2 实验结果

本小节将介绍实验评估的结果。本节内容与上述实验步骤相对应，分别为：

- 在已知特性的对象上，HierHybrid 检测误差的效果，与已有工作的对比；
- 在实际项目上，HierHybrid 与传统粒子群算法的对比；
- 在实际项目上，HierHybrid 与已有工作的对比。

在已知特性函数上，HierHybrid 检测误差的效果，以及与已有工作的对比

表 3.2 中的 6 个函数，已有工作 [68, 83] 已经对其进行了分析，将其划分为稳定函数和不稳定函数，其中稳定函数的运算结果要显著优于不稳定函数。这 6 个实验对象包括 2 个稳定函数和 4 个不稳定函数，如表所示。

表 3.2　HierHybrid 与 BGRT 的对比实验结果

函数名	是否为稳定函数	HierHybrid 最大误差	BGRT 最大误差
Newton	是	6.4e-16	1.7e-16
Inv	是	4.2e-16	2.5e-16
Root	否	1.1e-01	1.3e-14
Poly	否	3.1e+03	4.7e-14
Exp	否	7.9e-04	2.1e-15
Cos	否	2.7e-02	1.2e-16

表 3.2 中，可以看到，本章提出的 HierHybrid 方法可以区分稳定函数和不稳定函数，具有良好的检测误差效果：

- HierHybrid 可以确认稳定函数的稳定性。对于 Newton 函数和 Inv 函数，HierHybrid 检测到的最大误差都在 10^{-16} 这一数量级，对于 `double` 类型的浮点数，这一数量级的误差接近于 `double` 类型的表达极限，所以可以确认这两个函数的稳定性；
- HierHybrid 可以发现不稳定函数的显著误差。对于表中的 4 个不稳定函数，HierHybrid 都检测到了至少大于 10^{-4} 这一数量级的误差。这个数量级的误差与 `double` 类型的表达极限相差了万亿倍（10^{12}）以上，可认为对于每一个不稳定函数，HierHybrid 都找到了显著的误差。

同时，可以在表 3.2 中看到，HierHybrid 相比于已有工作 BGRT 具有明显的优势：

- 在表中的所有函数上，HierHybrid 检测到的最大误差都大于 BGRT；
- BGRT 无法区分稳定函数与不稳定函数。

综上所述，本章提出的 HierHybrid 作为黑盒场景下的测试用例生成工具，具有良好的检测误差的能力，且显著优于相同测试场景下的其他已有工作。

在实际项目上，HierHybrid 与标准粒子群算法 PSO 的对比

根据已有工作 [68]，本实验将大于 10^{-3} 的误差定义为显著误差。为了保证实验的公平性，本实验中，标准粒子群算法 PSO 采用了与 HierHybrid 实验设定中相同的迭代次数和运行时间限制，标准粒子群算法的相关参数参考了已有工作进行设定 [36, 79]。

表 3.3 HierHybrid 与标准粒子群算法的比较

HierHybrid	PSO	HierHybrid 带来的提升
30（34%）	3（3.4%）	900%

表 3.3 显示了在 GNU 科学计算库的 88 个实验实验对象中，HierHybrid 检测到的显著误差数量，和标准粒子群算法 PSO 检测到的显著误差数量的对比。从表中数据可以看到，HierHybrid 共发现了 30 个函数具有显著的误差，占总实验对象的 34%（30/88 = 34%）。相比之下，标准粒子群算法 PSO 搜索浮点数误差的能力较弱，这是由于浮点数的搜索空间过大所导致的。对于浮点数而言，其能表达的范围为 $(-10^{308}, 10^{308})$，而 PSO 算法在这一巨大的搜索空间内，收敛速度过慢，进而导致其无法在相应的迭代次数内找到显著的浮点数误差。

表 3.3 中的结果表明，本文提出的针对浮点数误差问题的层级混合粒子群算法 HierHybrid，可以显著提高粒子群算法对于浮点数误差的检测能力，是一种高效的误差搜索算法。

在实际项目上，HierHybrid 与已有工作的对比

LSGA [80, 86] 是一个已有的浮点数误差检测工作，其使用一种特定于浮点数结构的遗传算法对误差进行搜索。LSGA 同样在 GNU 科学计算库上进行了实验，因此 HierHybrid 可以直接在 GNU 科学计算库上与 LSGA 算法进行对比。在本实验中，HierHybrid 采用了与 LSGA 相同的实验设定，保证实验的公平性。

表 3.4 HierHybrid 与标准粒子群算法的比较

HierHybrid	LSGA	HierHybrid 带来的提升
30（34%）	23（26%）	30.4%

表 3.4 显示了在 GNU 科学计算库的 88 个实验对象中，HierHybrid 检测到的显著误差数量，和 LSGA 检测到的显著误差的数量的对比。从表中数据可以看到，HierHybrid 的误差检测效果显著优于 LSGA，效果提升了 30.4%。

HierHybrid 不仅针对浮点数的数据结构进行了优化，还充分利用了浮点数结构中不同部分对误差的影响，将搜索算法进行分层处理。表 3.4 中的结果表明，本文提出的层级混合粒子群算法 HierHybrid 可以充分利用浮点数的特性，显著优于已有的针对浮点数误差检测工作。

综上所述，本章实验结果表明，HierHybrid 作为黑盒场景下的测试用例生成工具，

可以有效的生成触发显著误差的测试用例，提高浮点数程序的测试效果。

3.6　讨论与小结

浮点数测试用例生成是浮点数程序测试与分析中的关键步骤。由于浮点数的特殊结构和误差的特殊性质，已有的测试用例生成技术不能很好的检测浮点数误差，使得针对浮点数误差的测试质量不高。

本章通过实证研究探究了浮点数不同结构对于运算结果误差的影响，并基于实证研究的发现，提出了针对浮点数误差问题的测试用例生成技术 HierHybrid。该技术可以充分利用浮点数的特性，在黑盒场景下自动生成触发显著误差的浮点数测试用例。实验结果表明，HierHybrid 具有良好的检测误差能力，其可以在实际项目中发现大量的误差缺陷，并显著优于已有的误差检测工作。

本章提出的测试用例生成工具在本文中作为黑盒场景下的搜索方法，成为浮点数程序误差测试与分析的关键技术，解决浮点数误差的稀疏性问题。

第四章　基于搜索的浮点数白盒分析技术

4.1　引言

浮点数误差分析是浮点数程序测试与分析中的另一个关键步骤。由于由于浮点数误差的未知性特点，大量测试和分析工作都依赖于测试预言来判断误差的存在与大小。此前，高质量的测试预言获取十分困难。利用本文第二章提出的 PsoHunter 技术，可以自动化的获取高质量的测试预言，解决未知性问题。然而其运行开销很大，如果需要大量获取测试预言，将显著降低相关测试与分析技术在实践中的可用性。因此，一个不依赖于测试预言的浮点数误差分析工具是十分重要且必要的。

在浮点数误差分析中，关注的首要问题即为运算误差的大小这一核心测量标准。因此，几乎所有的已有工作都直接使用“误差”这一测量标准对浮点数程序进行测试和分析 [8, 24, 53, 61, 62, 80]，进而上述工作都需要在测试和分析的过程中大量计算程序的误差，即浮点数程序 $\hat{f}(x)$ 与测试预言 $f(x)$ 之间的误差：

$$Err_{rel}(\hat{f}(x)) = \left| \frac{f(x) - \hat{f}(x)}{f(x)} \right|$$

其中，测试预言 $f(x)$ 必须通过高精度的测试预言生成工具得到，例如使用本文第二章提出的 PsoHunter 技术，或此前的自动化工具如 FpDebug [10]，甚至手动基于高精度浮点运算库 MPFR [25] 构建。然而，即使在本文第二章提出的 PsoHunter 已经解决未知性问题的前提下，获取测试预言的运算开销仍然十分巨大。由于现有的浮点运算体系——从编译器到浮点计算单元（FPU）——都是针对常用类型如 `float` 和 `double` 类型高度优化的，同时高精度浮点数类型不仅运算量更大，并且无法使用现有体系中的优化机制，因此运算开销较常用类型大了数个数量级：即使 128 位的四精度浮点数类型运算也比 64 位的 `double` 类型慢一百倍以上 [55]，而其他任意精度的运算库如 MPFR 的开销将进一步膨胀。由于测试预言的计算一定涉及到高精度浮点数类型的使用，因而，测试预言的获取开销十分庞大。

为了在分析中减少测试预言的使用，提高误差分析的运行效率与效果，本章对浮点数误差在运算间的产生、积累以及放大过程进行了深入分析，不同于上述所有已有技术直接使用“误差”这一衡量标准，本文提出了“原子状态函数”这一概念用于误差分析，从而可以在分析的过程中避免使用测试预言直接计算误差。在数值分析领域，状态函数 [34] 衡量了函数的稳定性：在函数输入中添加一个小的扰动，会导致函数输出中出现扰动的大小。在本章中，将关注所有原子浮点运算——例如 $+, -, \sin(x), \log(x)$

等基础运算——上的状态函数，本章定义为*原子状态函数*。

本章发现，原子状态函数是浮点数原子运算产生误差的核心因素，进而可以在复杂的浮点数程序中用于误差分析。对于每一个浮点数原子运算，其误差可被表示为：$\varepsilon_{out} = \varepsilon_{in}\Gamma + \mu$，其中：

- μ 为每个浮点运算的*引入误差*。在 IEEE 754 标准 [81] 和 GNU C 标准库 [45] 中都明确定义，引入误差被保证为一个非常小的值；
- Γ 为即为原子状态函数。由于引入误差极小，所以原子状态函数是决定运算误差的核心因素。

基于上述发现，本章提出了一种在**白盒**——即源代码可见——场景下的误差分析方法 Atomu。Atomu 可以自动化的检测浮点数误差，并且分析误差产生的原因。具体而言，相比于已有工作，本章提出的 Atomu 具有如下优点：

- 运算开销低。在白盒场景下，可以通过计算原子状态函数来判断运算误差的存在。计算原子状态函数不涉及高精度浮点数类型的使用，因而运算开销显著降低；
- 分析效果好。原子状态函数提供了浮点数程序运行状态的详细信息，可以在基于搜索的分析方法中，给搜索算法提供更多的信息，有助于提高检测误差的效果；
- 不依赖测试预言。由于在分析过程中使用原子状态函数，不再依赖测试预言进行分析，提高了实践中的可用性；
- 提供误差原因。对于检测到的触发误差的测试用例，可以通过分析其运行时的原子状态函数，判断误差产生的原因，有助于开发者和自动化工具对浮点数误差进行修复。

整体而言，Atomu 的分析过程可以概括为：在所有可行的输入空间中搜索会导致显著的原子状态函数的输入。Atomu 的返回值为一个有序列表，其中包括一系列潜在的导致浮点误差问题测试用例。该列表所包含的测试用例通常较少，开发者可以手动检查这些测试用例是否会触发误差，或使用如第二章提出的测试预言生成工具检查其误差。即使使用测试预言生成工具，由于数量较少，其运行开销仍然很低。

本章的主要贡献点如下：

- 提出了*原子状态函数*这一概念，并基于这一概念解释浮点数误差在运算间的产生、累积、和放大过程；
- 建立了基于原子状态函数的误差积累模型，基于该模型证明了原子状态函数为浮点数运算误差的核心因素；
- 设计并实现了基于原子状态函数的白盒误差分析技术 Atomu。Atomu 基于搜索算法，在所有可行空间中搜索会触发显著原子状态函数的测试用例，将其标记

为潜在的触发误差问题的测试用例。Atomu 不依赖测试预言，运行速度快，检测效果好；

- 实验评估结果表明了本章提出的误差分析技术 Atomu 的有效性——其运行速度比已有方法快 1000 倍以上，同时误差检测效果提高 40%。

本章的组织结构如下：第 4.2 节介绍了白盒分析的应用场景和特点；第 4.3 节介绍了浮点数原子运算的概念和特点，以及数值分析中的状态函数概念；第 4.4 节介绍了本章提出的基于原子状态函数的误差分析方法 Atomu；第 4.5 节介绍了本章提出的方法对应的工具实现；第 4.6 节包含了实验设计以及实验结果分析；第 4.7 节对本章工作进行了总结和讨论。

4.2　白盒分析场景及特点

本文在第 3.2 节中已经介绍过黑盒测试场景及特点。白盒测试与黑盒测试相对，又被称为结构测试，其主要关注程序的内部结构或运作机理 [37]，而不是程序的功能（即黑盒测试的目标）。在白盒测试中，通常将程序看作一系列子结构的组合：

- 需要关注程序的内部结构和状态；
- 需要程序的源代码和对程序的专业知识及深入理解；
- 考虑在不同测试下，不同节点的内部状态是否正确。

白盒测试——即源代码可见——是程序测试中的常见场景。相比于黑盒测试，白盒测试需要提供更多的信息（源代码、理解代码的专业知识），因此通常而言，白盒测试比黑盒测试具有更好的效果。

对于本文面对的浮点数误差问题，白盒测试的场景对应着浮点数程序源代码可见的场景。在该场景下，本章讨论的测试与分析技术将从基础的浮点运算入手（例如四则运算，三角函数等等），分析这些原子运算对于误差的影响。

由于已有技术大量依赖于测试预言，而测试预言的获取开销较大，影响了相关测试与分析的速度，降低了其在实践中的可行性。在白盒场景下，有更多的代码信息及运行时信息可以利用，使得不依赖测试预言的分析技术成为可能。

为了提高白盒场景下的测试与分析效果，本章将对浮点数原子运算的特性进行深入讨论，提出原子状态函数这一概念，减少了对于测试预言的依赖，并基于原子状态函数构建量浮点数程序的误差积累模型。本章使用上述模型提出一套基于搜索的浮点数白盒分析技术，用于在白盒场景下对浮点数误差问题进行测试和分析。

4.3 浮点数原子运算与状态函数

本节介绍了浮点数原子运算的概念，并讨论了这些原子运算所携带的误差。然后介绍了数值分析领域的状态函数这一概念，并将其引申，讨论浮点数误差与状态函数之间的关联性。

4.3.1 浮点数原子运算及其误差

浮点数原子运算

浮点数程序由一系列浮点数原子运算组成，原子运算包含下列四则运算和基础函数：

- 四则运算：包括 +，−，×，÷；
- 基础函数：
  - 三角函数：`sin`，`cos`，`tan`，`asin`，`acos`，`atan`，`atan2`；
  - 双曲函数：`sinh`，`cosh`，`tanh`；
  - 指数函数与对数函数：`exp`，`log`，`log10`；
  - 幂函数：`sqrt`，`pow`。

原子运算的误差

在第 1.2.3 节中，介绍了最小精度单位（ULP）以及 ULP 误差的概念。简单而言，对于 `double` 类型的浮点数，一个 ULP 误差大约对应于 1.1×10^{-16} 至 2.2×10^{-16} 之间的相对误差。由于 ULP 误差实际为浮点数的表达极限，其常用于精确定义浮点数运算所携带的误差。

对于原子运算中的四则运算，IEEE 754 标准 [81] 保证了所有四则运算的运算都被正确的舍入，并且运算结果的误差不会超过 1+1/1000 个 ULP 误差。

对于原子运算中的基础函数，GNU C 标准库 [45] 进行了如下表述：

> 理论上，在舍入到最接近的值的模式下，所有的基础函数的运算误差总是小于 *0.5* 个 *ULP* 误差。

舍入到最接近的值（见第 1.2.2 节）即为浮点数的默认舍入模式。同时，GNU C 标准库包含了一份已知最大误差的列表。通过查询该列表，本章发现，上述所有基础函数的已知最大误差为 2 个 ULP 误差。

综上所述，IEEE 754 标准和 GNU C 标准库共同规定了：原子运算的结果保证准确，并且一般而言运算结果的误差小于 0.5 个 ULP 误差，最多不超过 2 个 ULP 误差。

同时，本文注意到，即使所有的原子运算都是准确的，浮点数程序的运算结果却可能包含显著的误差：对于本文第三章中所讨论的 GNU 科学计算库中的函数，在某些

情况下其运算结果具有 10% 的相对误差，对应为 7×10^{14} 个 ULP 误差。本章发现，这种显著误差产生的原因是：在浮点数的原子运算之间，误差不仅仅会产生和积累，还会被某些原子运算显著放大。

4.3.2　状态函数

状态函数是数值分析领域的一个重要概念。状态函数衡量了一个数学函数 f 的内在稳定性 [34]。

假设对于函数 f，其输入 x 携带了一个小的误差 Δx，下列公式展示了函数 f 会在输出中将误差 Δx 放大的系数：

$$
\begin{aligned}
Err_{rel}(f(x),f(x+\Delta x)) &= \left|\frac{f(x+\Delta x)-f(x)}{f(x)}\right| \\
&= \left|\frac{f(x+\Delta x)-f(x)}{\Delta x}\cdot\frac{\Delta x}{f(x)}\right| \\
&= \left|(f'(x)+\frac{f''(x+\theta\Delta x)}{2!}\Delta x)\cdot\frac{\Delta x}{f(x)}\right|, \theta\in(0,1) \\
&= \left|\frac{\Delta x}{x}\right|\cdot\left|\frac{xf'(x)}{f(x)}\right| + O\left((\Delta x)^2\right) \\
&= Err_{rel}(x,x+\Delta x)\cdot\left|\frac{xf'(x)}{f(x)}\right| + O\left((\Delta x)^2\right)
\end{aligned}
$$

其中 θ 是一个 $(0,1)$ 之间的值，含有 θ 的项为泰勒展开中的拉格朗日余项 [39]。上述公式中展示了状态函数的定义 [34]：

$$
\Gamma_f(x) = \left|\frac{xf'(x)}{f(x)}\right|
$$

从误差分析的角度，状态函数测量了：对于函数 f 而言，输入携带的相对误差 $|\Delta x/x|$ 在输出中被函数 f 放大的系数。

对于一个浮点数程序，其表达的函数 $f(x)$ 的导数 $f'(x)$ 通常是未知的。因而，除非在某些导函数 $f'(x)$ 已知的情况下，计算一个函数的状态函数 $\Gamma_f(x)$ 通常比计算原函数 $f(x)$ 更加困难 [26]。

本章提出的原子状态函数是原子运算上的状态函数。因为原子运算的导函数的表达式都可以提前计算得到，所以计算原子状态函数是十分简单的（将在第 4.4.3 节中详细讨论）。

4.4 基于原子状态函数的误差分析

本节介绍本章的核心工作，包括本章的误差分析技术概要，一个基于原子状态函数的误差样例分析，然后将详细讨论原子状态函数与浮点数程序的误差之间的关系，并建立误差积累模型。进而讨论基于原子状态函数的误差搜索过程，最后讨论对搜索结果中的可疑测试样例进行排序。

4.4.1 技术概要

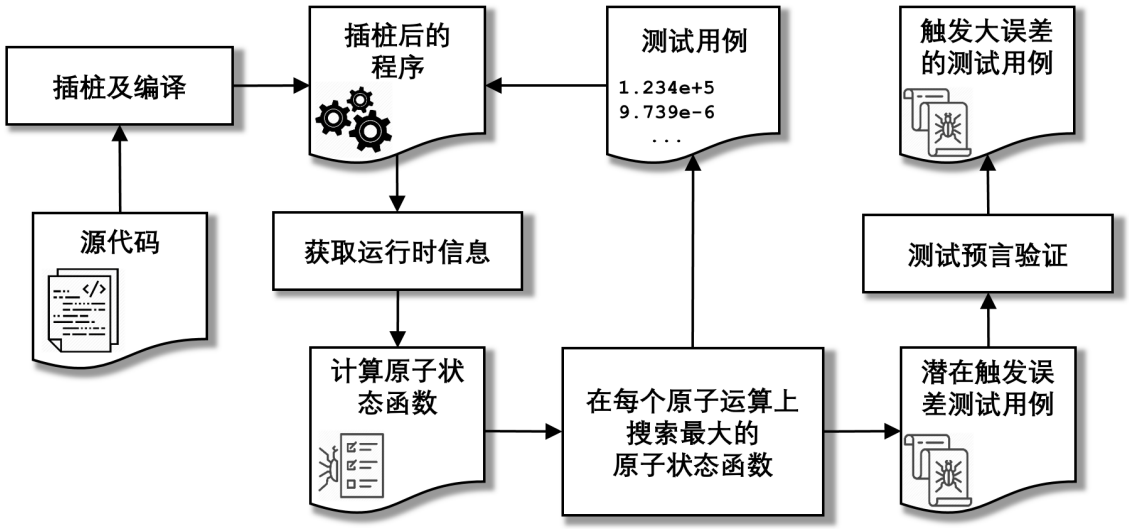


图 4.1 基于原子状态函数的误差分析流程图

对于一个待分析的浮点数程序 $\hat{y} = \hat{f}(\mathbf{x})$，本章的分析目标是找到一组测试用例 $\mathbf{x}$，使得在这组测试用例上浮点数程序具有显著的误差。本文将这个问题可以转化为搜索问题。搜索空间即为待分析浮点数程序的所有的可行输入。该搜索空间既可以为显式的，例如在文档中明确定义了函数的输入范围，也可以为隐式，例如只能在运行时通过观察函数是否抛出异常来决定输入的可行性。

对于搜索问题，最关键的一个核心定义即为搜索目标。在本章的定义下，搜索目标应为一个测试用例可能导致显著误差的可能性。在本章的第 4.1 节中已有讨论，一种常用做法是直接使用误差作为搜索的目标。然而根据上述讨论，直接使用误差作为搜索目标并没有充分利用白盒场景下的程序结构信息和运行时信息，使得分析过程大量依赖测试预言计算误差，运行效率不高。

因此，本章提出了使用原子状态函数作为搜索目标的分析方法 Atomu，深入分析了原子状态函数与浮点数误差的强关联性，通过搜索显著的原子状态函数间接的搜索触发显著误差的测试用例，完成浮点数误差分析工作。图 4.1 中展示了本章基于原子函数的搜索算法的流程。对于一个待分析的浮点数程序，Atomu 首先对程序的源代码进

行插桩，获得程序的运行时信息。然后利用上述运行时信息计算每个浮点数原子运算上的原子状态函数。通过搜索算法不断生成测试用例，搜索每个原子运算上最大的原子状态函数对应的测试用例。最终对于搜索结果，即潜在的触发误差的测试用例，使用测试预言对其进行验证，确认最终会触发误差的测试用例。

4.4.2　样例分析

为了详细展示原子状态函数和浮点数程序误差之间的关联性，本小节将通过一个实际的程序样例进行介绍和分析，展示原子状态函数的概念，以及其对浮点数程序误差的影响。

考虑表 4.1 中的样例程序 $\hat{f}$: foo(x)，其计算目标为数学表达式 $f(x) = (1-\cos x)/(x*x)$。浮点数样例程序在表 4.1 中的第一列。已知数学表达式 $f(x)$ 在 $x \to 0$ 时的极限为 $1/2$。

基于上述条件，开发者一般会认为样例程序 $\hat{f}$ 可以精确的计算数学表达式 f 的值。但是，实际上当输入为一个非常接近于 0 的值，例如为 10^{-7} 时，浮点数程序的计算结果将具有显著的误差：

- $\hat{f}(10^{-7}) \approx 0.4996$；
- $f(10^{-7}) = 0.4999999999999999583$。

本小节接下来将对于该浮点数程序中的原子运算进行分析，讨论误差是如何在运算间被引入和放大的。在表 4.1 中，不同的列分别展示了每次运算的输入，其对应的原子状态函数，以及运算结果和运算结果的误差。不精确的数位使用加粗显示。如本文第 4.3 节中所讨论，每次原子运算将：

1. 增加一个约为 1 个 ULP 误差大小的引入误差；
2. 将输入携带的误差放大，放大倍数为原子状态函数的值。

每次运算的具体信息如下：

- **原子运算 1：`v1 = cos(x)`**
  - 原子状态函数：$\Gamma_{\cos}(x) = |x \cdot \tan(x)|$；
  - 放大误差：由于函数的输入 x 被认为是精确值 [45]，即其携带误差为零，因此本次原子运算没有放大误差；
  - 引入误差：3.9964×10^{-18}，引入误差小于 1 个 ULP 误差。
- **原子运算 2：`v2 = 1.0 - v1`**
  - 原子状态函数：$\Gamma_{-}(v1) = \left|-\frac{v1}{1.0-v1}\right| = 2.0016 \times 10^{14}$；
  - 放大误差：运算输入 v1 包含一个非常小的误差 3.9964×10^{-18}。然而，由于该运算的原子状态函数非常大，运算结果中的误差将被放大为 7.9928×10^{-4}；

表 4.1　样例函数 f 以及浮点数程序 $\hat{f}$: foo(x) 的误差积累过程

样例函数：$f(x) = \dfrac{1-\cos(x)}{x^2}$　　$\lim\limits_{x\to 0} f(x) = \dfrac{1}{2}$

浮点数程序 $\hat{f}$: foo(x)	运算输入值	原子状态函数 Γ_{op}	浮点数运算结果 $\hat{f}$	运算结果包含的相对误差
foo(double x):				
v1=cos(x)	1.0e-7	1.0000e-14	9.99999999999995004e-01	3.9964e-18
v2=1.0-v1	1.0, 9.99999999999995004e-01	2.0016e+14, 2.0016e+14	4.996003610813204 43e-15	7.9928e-04
v3=x*x	1.0e-7, 1.0e-7	1, 1	9.9999999999999841e-15	6.8449e-17
v4=v2/v3	4.996003610813204 43e-15, 9.9999999999999841e-15	1, 1	4.996003610813204 99e-01	7.9928e-04
return v4				

  - 引入误差：约为 1 个 ULP 误差，由于放大误差非常大，引入误差可以忽略不计。

- **原子运算 3：v3 = x * x**
  - 原子状态函数：$\Gamma_{\times}(x) = 1$。乘法的原子状态函数始终为 1，即对于乘法运算，输入中包含的误差将直接传到输出中，既不放大也不缩小。
  - 放大误差：由于变量 x 是精确值，因此本次原子运算没有放大误差；
  - 引入误差：6.8449×10^{-17}，引入误差小于 1 个 ULP 误差。
- **原子运算 4：v4 = v2 / v3**
  - 原子状态函数：$\Gamma_{\div}(v2) = \Gamma_{\div}(v3) = 1$；
  - 放大误差：输入值 v2 所包含的相对误差 7.9928×10^{-4} 将被传到输出中（放大系数，即原子状态函数为 1），输入值 v3 所包含的相对误差非常小，可以忽略不计；
  - 引入误差：约为 1 个 ULP 误差，由于放大误差非常大，引入误差可以忽略不计。

该样例表明：

- 浮点数运算结果的显著误差可能是由于一个或多个显著的误差放大过程所导致的；
- 原子状态函数可以精确的计量每一个原子运算对误差的放大倍数；
- 原子状态函数的计算不会受到导函数 $f'(x)$ 未知的影响。因为所有原子运算的导函数 $f'(x)$ 的表达式都可以事先计算好，直接用于在分析过程中计算原子状态函数 $\Gamma_f(x)$。

4.4.3　原子状态函数与误差累积模型

本章的第 4.4.2 节使用了一个简单的样例展示原子状态函数与误差的联系。本小节中，将使用形式化的规则对原子状态函数与浮点数误差的关联进行更严谨的描述与讨论。

原子状态函数的定义

本章在第 4.3 节中已经讨论了组成浮点数程序的原子运算与状态函数的概念。本章将原子运算上的状态函数定义为**原子状态函数**。为了具体定义原子状态函数与原子运算中误差的关联，本节首先讨论一元运算，形如 $z = op(x)$：

- x 为一元运算的输入，并且携带了一个小的误差 ε_x；
- $\Gamma_{op}(x)$ 为该运算 op 的原子状态函数；
- z 为一元运算的输出，携带一个未知的误差 ε_z；
- $\mu_{op}(x)$ 为一元运算的引入误差（见第 4.3.1 节中讨论）。

根据第 4.3 节中讨论，原子运算 op 会把输入携带的误差 ε_x 放大原子状态函数 $\Gamma_{op}(x)$ 的倍数，即放大误差为 $\varepsilon_x\Gamma_{op}(x)$；同时，原子运算 op 还会引入一个约为 1 个 ULP 误差的引入误差，即 $\mu_{op}(x)$。所以，输出 z 中携带的误差 ε_z 可以表示为：

$$\varepsilon_z = \varepsilon_x\Gamma_{op}(x) + \mu_{op}(x) \tag{4.1}$$

同理，公式 4.1 可以使用相同的规则扩展至二元运算 $z = op(x, y)$，例如 $z = x + y$：

$$\varepsilon_z = \varepsilon_x\Gamma_{op,x}(x, y) + \varepsilon_y\Gamma_{op,y}(x, y) + \mu_{op}(x, y) \tag{4.2}$$

其中 $\Gamma_{op,x}(x, y)$ 和 $\Gamma_{op,y}(x, y)$ 是基于偏导数的原子状态函数。

基于原子状态函数的误差累积模型

接下来，本节将基于公式 4.1 与公式 4.2，提出基于原子状态函数的误差累积模型。首先考虑一个简单的样例：

```
1 double bar(double x) {
2   double v1,v2,v3; // 中间变量
3   double y;        // 运算结果
4   v1 = f1(x);
5   v2 = f2(v1);
6   v3 = f3(v1,v2);
7   y = f4(v3);
8   return y; }
```

其中 f1 至 f4 为 4 个原子运算，例如 sin，log，+，sqrt 等。根据公式 4.1 与公式 4.2，每个运算的原子状态函数如下：

$$\begin{aligned}
\varepsilon_x &= \mu_{init} \\
\varepsilon_{v1} &= \varepsilon_x \Gamma_{f_1} + \mu_{f_1} \\
\varepsilon_{v2} &= \varepsilon_{v1} \Gamma_{f_2} + \mu_{f_2} \\
\varepsilon_{v3} &= \varepsilon_{v1} \Gamma_{f_3,v1} + \varepsilon_{v2} \Gamma_{f_3,v2} + \mu_{f_3} \\
\varepsilon_y &= \varepsilon_{v3} \Gamma_{f_4} + \mu_{f_4}
\end{aligned}$$

将上述公式联合展开，则可以得到：

$$\begin{aligned}
\varepsilon_y = & \overbrace{\mu_{init} \Gamma_{f_1} \Gamma_{f_3,v1} \Gamma_{f_4}}^{x \to v1 \to v3 \to y} + \overbrace{\mu_{init} \Gamma_{f_1} \Gamma_{f_2} \Gamma_{f_3,v2} \Gamma_{f_4}}^{x \to v1 \to v2 \to v3 \to y} \\
& + \overbrace{\mu_{f_1} \Gamma_{f_3,v1} \Gamma_{f_4}}^{v1 \to v3 \to y} + \overbrace{\mu_{f_1} \Gamma_{f_2} \Gamma_{f_3,v2} \Gamma_{f_4}}^{v1 \to v2 \to v3 \to y} \\
& + \overbrace{\mu_{f_2} \Gamma_{f_3,v2} \Gamma_{f_4}}^{v2 \to v3 \to y} + \overbrace{\mu_{f_3} \Gamma_{f_4}}^{v3 \to y} + \overbrace{\mu_{f_4}}^{y}
\end{aligned}$$

在该公式中可以看到，每次原子运算的引入误差 μ 会被后续的数据流路径中的原子状态函数 Γ_{op} 放大。例如对于变量 x 而言，其有两条数据流路径到运算结果 y：$x \to v1 \to v3 \to y$ 对应于公式中的第一项；$x \to v1 \to v2 \to v3 \to y$ 对应于公式中的第二项。根据上述讨论，可以形式化的对误差累积模型进行表述和定义：

- P 为一个浮点数程序；
- E 为程序 P 中原子运算的集合；
- V 为程序中浮点数变量的集合；
- $G : \langle V, E \rangle$ 为程序的动态数据流图，y 为运算结果。图中 V 为图中点的集合，E 为图中边的集合。对于每个执行的一元原子运算 $\gamma = op(\alpha)$，有一条边 $op : \alpha \to \gamma$；对于每个执行的二元原子运算 $\gamma = op(\alpha, \beta)$，有两条边 $op_\alpha : \alpha \to \gamma$ 和 $op_\beta : \beta \to \gamma$；
- $m = [\alpha, \beta, \ldots, y]$ 为数据流图 G 中的一条由 α 至 y 的数据流路径；
- $M(\alpha)$ 为所有由节点 α 至运算结果 y 的数据流路径的集合；
- $res(op) : op \to \gamma$ 为原子运算 op 到其运算结果 γ 的映射关系。

根据上述定义，误差累积模型可以表示为：

$$\varepsilon_y = \sum_{e \in E} \left(\mu_e \cdot \sum_{m \in M(res(e))} \prod_{op \in m} \Gamma_{op} \right) \tag{4.3}$$

公式 4.3 可以证明本章的核心发现，即浮点数运算结果的误差 ε_y 只由原子状态函数 Γ_{op} 和引入误差 μ 所决定，由于引入误差非常小（不超过 2 个 ULP 误差，见第 4.3.1 节

中讨论），原子状态函数是影响浮点数误差的核心因素。

原子状态函数的计算

根据第 4.3.2 节中讨论，通常而言，对于一个任意的函数 $f(x)$，由于其导函数 $f'(x)$ 的表达式未知，计算其状态函数 $\Gamma_f(x)$ 比计算原函数 $f(x)$ 要困难的多。不同于传统的状态函数，由于本章考虑的是原子运算上的状态函数，而所有的原子运算的导函数都具有初等表达式的形式，可以事先计算好。因此，计算原子状态函数是十分简单的。

表 4.2　原子状态函数计算公式

原子运算 op	原子状态函数 Γ_{op}	触发误差区域
$op(x,y)=x+y$	$\Gamma_{+,x}(x,y)=\left\lvert\frac{x}{x+y}\right\rvert,\ \Gamma_{+,y}(x,y)=\left\lvert\frac{y}{x+y}\right\rvert$	$x\approx -y$
$op(x,y)=x-y$	$\Gamma_{-,x}(x,y)=\left\lvert\frac{x}{x-y}\right\rvert,\ \Gamma_{-,y}(x,y)=\left\lvert-\frac{y}{x-y}\right\rvert$	$x\approx y$
$op(x,y)=x\times y$	$\Gamma_{\times,x}(x,y)=\Gamma_{\times,y}(x,y)=1$	-
$op(x,y)=x\div y$	$\Gamma_{\div,x}(x,y)=\Gamma_{\div,y}(x,y)=1$	-
$op(x)=\sin(x)$	$\Gamma_{\sin}(x)=\lvert x\cdot\cot(x)\rvert$	$x\to n\pi, n\in\mathbb{Z}$
$op(x)=\cos(x)$	$\Gamma_{\cos}(x)=\lvert x\cdot\tan(x)\rvert$	$x\to n\pi+\frac{\pi}{2}, n\in\mathbb{Z}$
$op(x)=\tan(x)$	$\Gamma_{\tan}(x)=\left\lvert\frac{x}{\sin(x)\cos(x)}\right\rvert$	$x\to \frac{n\pi}{2}, n\in\mathbb{Z}$
$op(x)=\arcsin(x)$	$\Gamma_{\arcsin}(x)=\left\lvert\frac{x}{\sqrt{1-x^2}\cdot\arcsin(x)}\right\rvert$	$x\to -1^+, x\to 1^-$
$op(x)=\arccos(x)$	$\Gamma_{\arccos}(x)=\left\lvert-\frac{x}{\sqrt{1-x^2}\cdot\arccos(x)}\right\rvert$	$x\to -1^+, x\to 1^-$
$op(x)=\arctan(x)$	$\Gamma_{\arctan}(x)=\left\lvert\frac{x}{(x^2+1)\cdot\arctan(x)}\right\rvert$	-
$op(x,y)=\arctan(\frac{y}{x})$	$\Gamma_{atan2,x}(x,y)=\Gamma_{atan2,y}(x,y)=\left\lvert\frac{xy}{(x^2+y^2)\arctan\left(\frac{y}{x}\right)}\right\rvert$	-
$op(x)=\sinh(x)$	$\Gamma_{\sinh}(x)=\lvert x\cdot\coth(x)\rvert$	$x\to\pm\infty$
$op(x)=\cosh(x)$	$\Gamma_{\cosh}(x)=\lvert x\cdot\tanh(x)\rvert$	$x\to\pm\infty$
$op(x)=\tanh(x)$	$\Gamma_{\tanh}(x)=\left\lvert\frac{x}{\sinh(x)\cosh(x)}\right\rvert$	-
$op(x)=\exp(x)$	$\Gamma_{\exp}(x)=\lvert x\rvert$	$x\to\pm\infty$
$op(x)=\log(x)$	$\Gamma_{\log}(x)=\left\lvert\frac{1}{\log x}\right\rvert$	$x\to 1$
$op(x)=\log_{10}(x)$	$\Gamma_{log10}(x)=\left\lvert\frac{1}{\log x}\right\rvert$	$x\to 1$
$op(x)=\sqrt{x}$	$\Gamma_{sqrt}(x)=0.5$	-
$op(x,y)=x^y$	$\Gamma_{pow,x}(x,y)=\lvert y\rvert,\ \Gamma_{pow,y}(x,y)=\lvert y\log(x)\rvert$	$x\to 0^+, y\to\pm\infty$

表 4.2 中展示了第 4.3.1 节中讨论的所有原子运算的原子状态函数的计算公式。其中，触发误差的区域是指，当运算的输入在该区域时，原子状态函数将会显著增大，进而使得该运算可能会将误差显著放大。

通过分析表 4.2 中的原子状态函数的极值，可以将原子运算分为两类：

- 潜在不稳定运算：包括 +，-，sin，cos，tan，arcsin，arccos，cosh，exp，pow，log，log10。对于这些潜在不稳定运算，如果其输入在触发误差区域内，则这些运算的原子状态函数 Γ_{op} 将趋近于正无穷。此时，输入中携带的任何误差都将被显著放大，使得运算结果携带显著的误差，变得非常不准确；
- 稳定运算：包括 ×，÷，arctan，arctan2，sqrt。对于这些稳定运算，由其原子状态函数表达式的值域可知，其原子状态函数 Γ_{op} 始终不大于 1。因此，在任何情况下，稳定运算都不会造成显著的误差放大效应。

4.4.4 基于原子状态函数的搜索过程

经过上述讨论，本章已经发现并证明了原子状态函数与浮点数运算误差之间的关联，同时提出了基于原子状态函数的误差累积模型。为了分析浮点数误差，本章提出了基于搜索的分析方法 Atomu。对于浮点数程序，Atomu 尝试在每个潜在不稳定运算上搜索会触发最大原子状态函数的测试用例。其搜索步骤过程可以表述为如下形式：

1. 生成一组初始的测试用例；
2. 使用上述测试用例执行待分析的浮点数程序；
3. 收集运行时信息，计算每个（潜在不稳定）原子运算的原子状态函数；
4. 根据已有的信息继续生成新的测试用例，目标为触发更大的原子状态函数；
5. 重复上述第 2 步至第 4 步中的步骤，直到满足搜索终止条件；
6. 对于每一个（潜在不稳定）原子运算，返回找到的最大的原子状态函数以及其对应的测试用例。

本章使用一种进化搜索算法作为上述搜索过程的具体实现。进化搜索算法 [4] 是一种解决最优化问题的全局搜索算法。在本章的搜索过程中，搜索个体为测试用例，搜索空间为所有可行的浮点数程序输入，搜索目标为最大化原子状态函数。

算法 4.1 描述了本章提出的进化算法框架。该算法中有 3 个核心步骤，分别为初始化种群、选择、和变异。接下来本小节将具体解释算法中的各个步骤和细节。

初始化种群

首先，该搜索算法将均匀的在搜索空间中生成个体——即浮点数测试用例（第 2 行），然后在待分析的浮点数程序上运行所有的测试用例，收集运行时信息，计算所有潜在不稳定运算上的原子状态函数（第 3-4 行），供后续步骤使用。

在第 4.4.3 节中已有讨论，所有的原子运算可以按照其原子状态函数的数学性质分为稳定运算和潜在不稳定运算。由于稳定运算的原子状态函数始终不大于 1，该算法只

算法 4.1: 进化搜索算法

输入: 插桩后的浮点数程序 $\mathbb{P}$
输入: 初始化种群大小 *initSize*
输入: 迭代次数 *iterSize*
输出: 一组测试用例 $X = \{x_1, x_2, \ldots, x_n\}$，对应每个原子运算 $\{op_1, op_2, \ldots, op_n\}$

```
1  X ← ∅
2  initTests ← 生成初始种群 (initSize)
3  for test in initTests do
4      计算所有原子状态函数 (test)
5  for op_i in ℙ do
6      if op_i 是潜在不稳定运算 then
7          T_i ← 初始种群
8          for j ← 0 to iterSize do
9              x ← 选择测试用例 (T_i, op_i)
10             x' ← 变异测试用例 (x, j)
11             ac' ← 计算原子状态函数 (op_i, x')
12             T_i. 添加 ({x', ac'})
13         {x_i, ac_i} ← 最优个体 (T_i)
14         if ac_i > 不稳定阈值 then
15             X. 添加 (x_i)
16 return X
```

关注潜在不稳定运算（第 5-6 行），并且在每个潜在不稳定运算上，不断进行迭代搜索，尝试找到最大的原子状态函数（第 8-12 行）。

在每次迭代搜索的过程中，进化算法首先选择一个测试用例（第 9 行），对其进行变异（第 10 行），然后执行变异后的测试用例并计算新的原子状态函数（第 11 行），并将其放入种群中（第 12 行）。

在迭代搜索结束后，算法检查在运算 op_i 上找到的最大的原子状态函数是否大于不稳定阈值（第 13-14 行），如果大于阈值的话，则将对应的测试用例加入输出集合中（第 15 行）。

最后，在所有运算上的搜索都结束后，输出集合 X 将包含一系列会触发显著原子状态函数的测试用例 $\{x_1, x_2, \ldots, x_n\}$，每一个测试用例 x_i 都对应了一个原子运算 op_i，且其原子状态函数 $\Gamma_{op_i}(x_i)$ 大于不稳定阈值。

选择

选择步骤是进化算法的一个核心步骤。在选择步骤中（第 9 行），选择目标是以较大的概率选择较优的个体，使其进入后续进化步骤。在本章中，较优的个体即为触发较大原子状态函数的个体。本章提出的进化算法使用了基于排名的选择算法 [6]，即根

据个体的原子状态函数大小进行排名，然后按照其排名进行选择操作。

具体而言，选择步骤中需要给每一个个体分配一个被选择的概率。本章使用了基于几何分布的概率分配算法 [75] 计算概率。假设测试用例 t_r 对应的原子状态函数在所有的 M 个测试用例中排名为 r，则其分配的概率为：

$$P(t_r) = \frac{\alpha(1-\alpha)^{r-1}}{\sum_{j=1}^{M}\alpha(1-\alpha)^{j-1}}$$

其中 α 为几何分布的参数。该几何分布在排序后的测试用例序列上模拟了独立伯努利过程 [54]。例如，假设共有 10 个测试用例 $M = 10$，几何分布的参数为 $\alpha = 0.2$，则测试用例被选取的概率从大到小为 $20\%, 16\%, 12.8\%, \ldots, 2.7\%$。为了满足所有个体被选取的概率之和为 1，对上述几何分布的概率经过归一化后，其实际被选取概率为 $22.4\%, 17.9\%, 14.3\%, \ldots, 3\%$。

变异

变异步骤是进化算法的另一个核心步骤。在变异步骤中（第 10 行），其目标是对于被选择的个体，在其邻域内生成一个新的个体。变异步骤的核心思路是，在较优个体的邻域内进行搜索，有更大的可能性找到更优的个体。本章的进化算法使用了一种常用的变异方法对测试用例进行变异，即在被选择的个体加入一个均值为 0 的正态分布随机变量进行变异 [4]：

$$t' = t + t \cdot \mathcal{N}(0, \sigma_j)$$

其中，j 为迭代次数的计数器，而 σ_j 为正态分布的方差参数。σ_j 参数控制了搜索邻域的大小，例如，若被选取的测试用例 $t = 1.2$，对于一个较大的 σ_j 参数，变异后的个体可能为 1.3，而对于一个较小的 σ_j 参数，变异后的个体可能为 1.2003。本章使用的变异策略为在迭代过程中不断缩小搜索的邻域，即随着迭代计数器 j 的增加，σ_j 参数随之减小：

$$\sigma_j = \sigma_{st}^{(N-j)/N} \cdot \sigma_{end}^{j/N}$$

其中 N 为总的迭代次数，σ_{st} 和 σ_{end} 是迭代开始时的参数和迭代结束时的参数。例如，假设邻域搜索参数为 $\sigma_{st} = 10^{-1}$ 和 $\sigma_{end} = 10^{-7}$，且迭代次数为 $N = 100$。则在第 1 次迭代时，邻域搜索参数为 $\sigma_0 = 10^{-1}$ 在第 50 次迭代时，邻域搜索参数为 $\sigma_{50} = 10^{-4}$ 在最后一次迭代时，邻域搜索参数为 $\sigma_{100} = 10^{-7}$。

4.4.5 搜索结果排序

本章第 4.4.4 节中提出了基于原子状态函数的搜索过程。在搜索结束后，将返回一组会触发显著原子状态函数的测试用例集合 $X = \{x_1, x_2, \ldots, x_n\}$。由于显著的原子状态

函数并不保证总是会触发显著的误差，因此，仍然需要对相关测试用例进行进一步验证，检查其是否会产生实际的误差。

无论是手动对测试用例进行检查，还是使用自动化的测试预言生成工具进行检查，都可以对测试用例进行排序 [23]，以提高测试和分析的效率。本小节基于第 4.4.3 章中提出的误差累积模型，提出了一种简单快速的搜索结果排序算法。

考虑第 4.4.3 章中的样例程序，其在误差积累模型下的运算结果的误差公式为：

$$\begin{aligned}\varepsilon_y =& \mu_{init}\Gamma_{f_1}\Gamma_{f_3,v1}\Gamma_{f_4} + \mu_{init}\Gamma_{f_1}\Gamma_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4} \\ &+ \mu_{f_1}\Gamma_{f_3,v1}\Gamma_{f_4} + \mu_{f_1}\Gamma_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4} \\ &+ \mu_{f_2}\Gamma_{f_3,v2}\Gamma_{f_4} + \mu_{f_3}\Gamma_{f_4} + \mu_{f_4}\end{aligned}$$

在上述公式中可以发现，距返回值运算越近的运算，在误差公式中具有越高的重要性。例如 Γ_{f4} 在除了最后一项的每一项中出现，意味着大的 Γ_{f4} 更有可能导致结果中大的误差。从形式化的公式中同样可以的相同的结论：

$$\varepsilon_y = \sum_{e\in E}\left(\mu_e \cdot \sum_{m\in M(res(e))}\prod_{op\in m}\Gamma_{op}\right)$$

越靠后的运算出现在越多的数据流路径中，因而具有更大的权重及重要性。

综上所述，本章根据上述发现提出了一种简单的搜索结果排序算法：对于搜索结果中的测试用例 x_i 和其对应的原子运算 op_i，本章使用 $step_i$ 表示运算 op_i 至返回值的最少运算数。然后根据 $step_i$ 对搜索结果进行排序，$step_i$ 越小，则其对应的测试用例 x_i 排序越靠前。

4.5　工具实现

本章实现了基于搜索的白盒分析工具 Atomu，Atomu 可以对给定的浮点数程序进行插桩，对浮点数误差进行检测和分析，生成会触发显著误差的浮点数测试用例，以及对应的导致显著误差的原子运算。

程序插桩

程序插桩的目标是获取程序的运行时状态，用于计算每个原子运算的原子状态函数。插桩由如下步骤组成：

1. *源代码 → LLVM IR 中间码*：该步骤将源代码文件——例如 `sample.c` ——编译至 LLVM IR 中间码（LLVM Intermediate Representation） [42]。Atomu 使用 Clang[①]对 C/C++ 程序进行编译；

① https://clang.llvm.org/

2. 对 *LLVM IR* 中间码进行插桩：该步骤将对 LLVM IR 中间码中的指令进行扫描。对于每个浮点数原子运算，该步骤会插入一个外部的函数调用，将运算的类型，输入值，以及指令 id 作为参数传给外部函数调用；

3. *LLVM IR* 中间码 → 插桩后的运算库：该步骤编译对插桩后的 LLVM IR 中间码，生成一个插桩后的运算库。对于任何外部程序，例如本章提出的搜索模块，都可以通过调用该运算库的方式获取 `sample.c` 文件的运行时信息。

搜索算法

Atomu 使用 C++ 语言实现了第 4.4.4 节提出的进化算法。对于其中的随机变量，Atomu 使用了 C++ 中的随机数生成库（`<random>` 运算库）中的函数进行实现，例如使用 `uniform_real_distribution` 生成均匀分布的随机变量，使用 `geometric_distribution` 生成几何分布的随机变量等。

对于搜索算法中的参数，初始化的种群大小为 100000，迭代次数为 10000，变异步骤中的邻域搜索参数为 $\sigma_{st} = 10^{-2}$ 以及 $\sigma_{end} = 10^{-13}$。上述参数可以根据实际需求进行调整。

4.6 实验评估

本节分为实验设计和实验结果两个部分。

4.6.1 实验设计

实验对象

本章的实验对象为一系列来自于 GNU 科学计算库（GNU Scientific Library）中的函数。GNU 科学计算库是一个开源的科学计算库，包含大量常用的数值计算函数，例如贝塞尔函数（Bessel Functions），指数积分（Exponential Integrals），Γ 函数（Gamma Functions）等。GNU 科学计算库被广泛应用于各类浮点数程序中，也是相关浮点数误差分析文献的实验对象 [9, 68, 77]。

在 GNU 科学计算库包含的所有函数中，本章选取了其中输入和输出都为浮点数类型的共 88 个函数作为本章的实验对象。

理想结果获取

本章提出的白盒分析方法 Atomu 在完整的分析过程中都不需要使用测试预言。Atomu 的分析结果为一组高度可疑的会触发显著误差的测试用例。由于分析结果通常较小（平均每个函数 11 个测试用例，见第 4.6.2 节），开发者可以手动对结果进行后

续的调试，或使用自动化的测试预言生成工具辅助调试。为了验证 Atomu 的效果，本章使用第二章提出的测试预言生成工具 PsoHunter，以及 mpmath [87] 获取测试预言，验证 Atomu 检测和分析浮点数误差的效果。

实验步骤

本章实验由以下几部分组成：

- 第一部分通过实验验证 Atomu 检测不稳定运算的效果。在该实验中，将验证 Atomu 在实验对象中能找到多少触发显著原子状态函数的测试用例，即发现其检测不稳定运算的效果；
- 第二部分通过实验验证 Atomu 检测与分析浮点数误差的效果。在该实验中，将验证 Atomu 在实验对象中能找到多少触发显著误差的测试用例，并与已有工作进行比较，验证 Atomu 检测与分析误差的效果；
- 第三部分通过实验验证 Atomu 的运行开销。在该实验中，将比较 Atomu 与已有工作的运行开销，验证 Atomu 的运行效率。

4.6.2　实验结果

本小节将介绍实验评估的结果。本节内容与上述实验步骤相对应，分别为：

- Atomu 检测不稳定运算的效果；
- Atomu 检测与分析浮点数误差的效果；
- Atomu 的运行开销。

Atomu 检测不稳定运算的效果

表 4.3　Atomu 检测不稳定运算的实验结果

GSL 函数包含的运算	浮点运算	潜在 不稳定运算	不稳定运算	Atomu 检测结果
每个函数平均值	90.4	39.8	11.8	11.8
所有函数总值	7957	3948	1037	1037

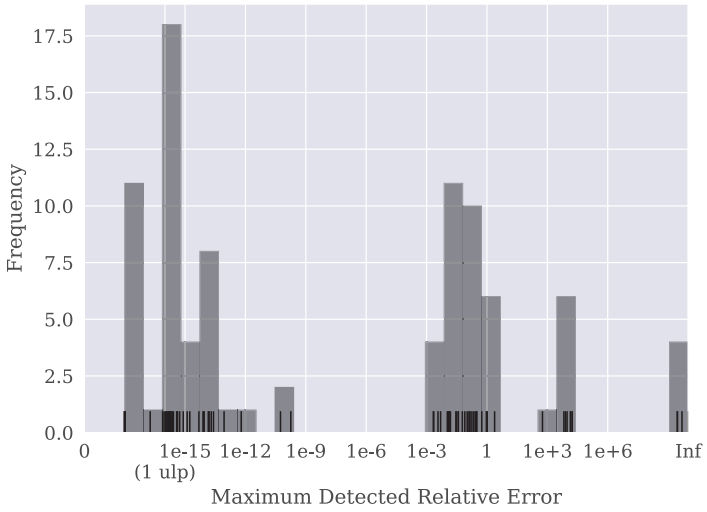
表 4.3 中展示了 Atomu 检测不稳定运算的实验结果。在表中，所有数量的单位为浮点数运算的个数。其中：

- “浮点运算”列展示了实验对象——即每个 GSL 函数——平均包含的浮点运算数量以及总计浮点运算数量；
- “潜在不稳定运算”列显示了在 GSL 函数中，约 44%（$39.8/90.4 \approx 0.44$）的运算为潜在不稳定运算，例如 sin，log 等运算（见第 4.4.3 节中讨论）；

- “不稳定运算”列显示了实际包含的不稳定运算的个数。不稳定运算的定义为，至少存在一个测试用例，使得该运算的原子状态函数大于 10；
- “Atomu 检测结果”列与不稳定运算列具有相同的数值，因为对于每个不稳定运算，Atomu 只保留触发最大原子状态函数的测试用例。

表 4.3 中 Atomu 的检测结果显示，约为 13%（11.8/90.4 ≈ 0.13）的浮点数运算为实际的不稳定运算，Atomu 至少可以找到一个测试用例，使得不稳定运算在该测试用例的执行过程中将误差显著放大。

Atomu 检测与分析浮点数误差的效果



最大发现误差	函数个数
$Err_{rel} \in [0, 10^{-15})$	31
$Err_{rel} \in [10^{-15}, 10^{-12})$	13
$Err_{rel} \in [10^{-12}, 10^{-9})$	2
$Err_{rel} \in [10^{-9}, 10^{-6})$	0
$Err_{rel} \in [10^{-6}, 10^{-3})$	0
不显著误差	46
$Err_{rel} \in [10^{-3}, 1)$	29
$Err_{rel} \in [1, 10^{3})$	3
$Err_{rel} \in [10^{3}, +\infty)$	10
显著误差	42
Total	88

图 4.2　Atomu 检测浮点数误差的实验结果

图 4.2 中展示了 Atomu 在 88 个实验对象函数上发现的最大误差。左侧的直方图展示了最大误差在 GSL 函数中的分布。图中结果显示，GSL 函数按照最大误差可以被分为两类，对应为图中的两个峰值区域：直方图左侧的函数最大误差小于 10^{-9}，即这些函数不涉及误差问题，其运算结果始终准确；直方图右侧的函数最大误差大于 10^{-3}，即这些函数受到浮点数误差问题的影响，在某些测试用例下，其计算结果包含显著的误差。

图 4.2 中右侧的表格显示了具体的函数分布。由于上述两类函数的最大误差具有显著的区别：一部分小于 10^{-9}，另一部分大于 10^{-3}，两类函数的最大误差相差十万倍以上。同时，相对误差大于 10^{-3} 对于 `double` 类型的浮点数而言已经是非常大的误差。因此，本章将运算结果大于 10^{-3} 的函数定义为具有显著浮点数误差问题的函数。

实验结果表明，本章提出的白盒分析技术 Atomu 发现约 47.7% 的 GSL 函数具有显著的浮点数误差问题。

DEMC [77] 是已有的浮点数误差检测工作。DEMC 的实验对象为 49 个 GSL 函数，为本章实验对象的子集。

表 4.4 与表 4.5 中显示了 Atomu 与 DEMC 的实验结果对比。在 DEMC 的 49 个实验对象函数上，DEMC 可以发现 20 个有显著误差的函数，而 Atomu 可以发现 28 个。同时，Atomu 发现的 28 个函数中包含了所有 DEMC 发现的 20 个函数，显示了 Atomu 检测与分析浮点数误差的效果显著优于 DEMC。Atomu 找到而 DEMC 未找到的具有显著误差的函数在表表 4.4 中高亮显示。

为了验证 Atomu 的分析结果是否存在假阳性或假阴性问题，本章随机抽取了一部分一部分函数进行进一步验证。

对于假阳性问题，本章通过使用两种不同的测试预言生成工具进行验证，即第二章提出的 PsoHunter 以及 mpmath [87] 分别计算测试预言。实验结果表明，PsoHunter 与 mpmath 生成的测试预言一致，Atomu 生成的测试用例确实可以在 42 个具有显著误差的函数上触发显著误差，不涉及假阳性问题。

对于假阴性问题，通过实验“完全”证明不存在假阴性问题是不现实的：必须对所有可能的测试用例进行测试，才能证明显著误差不存在，而所有可能的测试用例至少有 2^{64} 个，且随着函数参数的增加而指数增长。因此，本章从实践角度进行验证：将搜索中的初始种群数量和迭代次数同时提高 1000 倍，极大增加了搜索过程的搜索次数和时间。实验结果表明，在密集测试的实验结果中，上述函数的最大发现误差与之前保持了相同的数量级，没有发生显著变化——即没有发现新的具有显著误差的函数。因此，该结果在实践角度表明 Atomu 不涉及假阴性问题。

为了尽早发现搜索结果中更可疑的测试用例，本章第 4.4.5 节中提出的对搜索结果排序的方法。本章通过实验验证对搜索结果排序和未排序（每次随机选取）的效果。图 4.3 展示了，对于 Atomu 的搜索结果，排序选取和随机选取的检测误差效果。图 4.3 结果表明，对于所有的 42 个有显著误差问题的函数：

- 使用第 4.4.5 节中的排序算法，可以在只检测 top-1 的情况下发现 74% 的误差问题；只检测 top-4 的情况下发现 95% 的误差问题；
- 未对结果进行排序时，相对应的比例仅为 21% 和 57%。

上述结果表明，第 4.4.5 节提出的排序算法非常有效，同时也证明了 Atomu 的分析结果可靠，生成的测试用例很可能触发显著的浮点数误差问题。

综上所述，在 Atomu 检测与分析浮点数误差的效果上，Atomu 能检测到 47.7% 的实验对象具有显著的浮点数误差问题。相比于已有工作，检测效果提高了 40%。同时，Atomu 的分析结果可靠，不涉及假阳性和假阴性问题。最后，Atomu 使用的排序算法可以有效对分析结果中的测试用例进行排序。

表 4.4　42 个具有显著误差的函数的实验数据

函数名	最大发现误差	Atomu 运行时间（秒）	DEMC 是否发现显著误差	DEMC 运行时间（秒）
gsl_sf_airy_Ai	6.64E+03	0.94	✓	112.35
gsl_sf_airy_Bi	2.89E+09	0.88	✓	91.37
gsl_sf_airy_Ai_scaled	1.40E+04	0.95	-	
gsl_sf_airy_Bi_scaled	4.91E+09	0.93	-	
gsl_sf_airy_Ai_deriv	3.70E-03	0.37	✓	203.47
gsl_sf_airy_Bi_deriv	2.20E-01	0.37	✓	188.82
gsl_sf_airy_Ai_deriv_scaled	1.09E-02	0.36	-	
gsl_sf_airy_Bi_deriv_scaled	1.26E-02	0.39	-	
gsl_sf_bessel_J0	5.97E-02	0.68	✓	2079.15
gsl_sf_bessel_J1	1.79E-01	0.72	✓	1777.88
gsl_sf_bessel_Y0	7.93E-02	0.64	✓	325.22
gsl_sf_bessel_Y1	1.04E-01	0.69	✓	753.16
gsl_sf_bessel_j1	2.17E-03	0.07	-	
gsl_sf_bessel_j2	4.99E-03	0.07	-	
gsl_sf_bessel_y0	1.72E+04	0.22	-	
gsl_sf_bessel_y1	9.58E+03	0.56	-	
gsl_sf_bessel_y2	1.46E+04	0.60	-	
gsl_sf_clausen	9.36E-01	0.25	✓	471.55
gsl_sf_dilog	5.52E-01	0.27	✗	459.64
gsl_sf_expint_E1	2.92E-02	0.43	✗	96.18
gsl_sf_expint_E2	2.40E+00	0.49	✗	165.38
gsl_sf_expint_E1_scaled	2.92E-02	0.63	-	
gsl_sf_expint_E2_scaled	3.01E+212	0.62	-	
gsl_sf_expint_Ei	1.11E-02	0.44	✓	112.66
gsl_sf_expint_Ei_scaled	1.41E-01	0.63	-	
gsl_sf_Chi	1.28E-01	0.80	✓	199.98
gsl_sf_Ci	5.74E+02	1.46	✓	84.80
gsl_sf_lngamma	3.06E-01	0.32	✗	106.87
gsl_sf_lambert_W0	3.11E-01	0.11	✗	309.05
gsl_sf_lambert_Wm1	1.00E+00	0.12	-	
gsl_sf_legendre_P2	3.81E-02	0.02	✓	1168.49
gsl_sf_legendre_P3	3.72E-02	0.02	✓	908.69
gsl_sf_legendre_Q1	1.28E-02	0.04	✓	995.65
gsl_sf_psi	9.89E-01	0.60	✓	187.66
gsl_sf_psi_1	1.40E-01	0.33	✓	165.71
gsl_sf_sin	2.90E+09	0.26	✗	135.14
gsl_sf_cos	7.96E+03	0.26	✗	130.22
gsl_sf_sinc	1.00E+00	0.37	✗	149.43
gsl_sf_lnsinh	2.64E-01	0.03	✓	236.93
gsl_sf_zeta	1.29E-02	0.98	✓	584.12
gsl_sf_zetam1	2.26E-03	1.11	-	
gsl_sf_eta	1.53E-02	1.05	✓	668.39
显著误差组函数平均运行时间		0.5		459.6
所有函数平均运行时间		0.34		585.8

表 4.5　46 个无显著误差的函数的实验数据

函数名	最大发现误差	Atomu 运行时间（秒）	DEMC 是否发现显著误差	DEMC 运行时间（秒）
gsl_sf_bessel_I0	2.37E-16	0.16	✗	191.23
gsl_sf_bessel_I1	2.21E-16	0.17	✗	189.81
gsl_sf_bessel_I0_scaled	1.92E-16	0.29	-	
gsl_sf_bessel_I1_scaled	0.00E+00	0.29	-	
gsl_sf_bessel_K0	2.64E-16	0.19	✗	3336.70
gsl_sf_bessel_K1	1.70E-16	0.20	✗	7905.00
gsl_sf_bessel_K0_scaled	1.44E-16	0.18	-	
gsl_sf_bessel_K1_scaled	1.69E-16	0.19	-	
gsl_sf_bessel_j0	1.12E-16	0.04	-	
gsl_sf_bessel_i0_scaled	0.00E+00	0.02	-	
gsl_sf_bessel_i1_scaled	4.87E-15	0.03	-	
gsl_sf_bessel_i2_scaled	0.00E+00	0.03	-	
gsl_sf_bessel_k0_scaled	0.00E+00	0.01	-	
gsl_sf_bessel_k1_scaled	0.00E+00	0.01	-	
gsl_sf_bessel_k2_scaled	0.00E+00	0.01	-	
gsl_sf_ellint_Kcomp	1.79E-10	0.37	✗	48.44
gsl_sf_ellint_Ecomp	1.27E-15	0.81	-	
gsl_sf_erfc	8.13E-16	0.28	✗	205.25
gsl_sf_log_erfc	5.37E-16	0.20	✗	295.88
gsl_sf_erf	9.10E-17	0.27	✗	71.04
gsl_sf_erf_Z	0.00E+00	0.02	-	
gsl_sf_erf_Q	8.64E-15	0.29	-	
gsl_sf_hazard	7.78E-15	0.23	-	
gsl_sf_exp	0.00E+00	0.01	✗	46.14
gsl_sf_expm1	2.58E-14	0.02	✗	39.11
gsl_sf_exprel	1.52E-14	0.02	-	
gsl_sf_exprel_2	5.51E-11	0.03	-	
gsl_sf_Shi	4.21E-16	0.63	✗	151.78
gsl_sf_Si	1.88E-16	0.56	✗	657.17
gsl_sf_fermi_dirac_m1	0.00E+00	0.02	-	
gsl_sf_fermi_dirac_0	1.51E-14	0.02	-	
gsl_sf_fermi_dirac_1	3.97E-16	0.30	-	
gsl_sf_fermi_dirac_2	3.82E-16	0.30	-	
gsl_sf_fermi_dirac_mhalf	1.72E-15	0.42	-	
gsl_sf_fermi_dirac_half	1.42E-14	0.44	-	
gsl_sf_fermi_dirac_3half	8.74E-15	0.41	-	
gsl_sf_gamma	1.99E-14	0.24	✗	197.16
gsl_sf_gammainv	8.67E-14	0.51	✗	207.41
gsl_sf_legendre_P1	0.00E+00	0.01	✗	416.00
gsl_sf_legendre_Q0	7.66E-17	0.04	✗	659.26
gsl_sf_log	1.11E-16	0.02	✗	154.74
gsl_sf_log_abs	1.85E-17	0.02	✗	245.43
gsl_sf_log_1plusx	1.30E-16	0.12	✗	120.50
gsl_sf_log_1plusx_mx	1.53E-16	0.11	✗	347.54
gsl_sf_synchrotron_2	6.16E-13	0.17	-	
gsl_sf_lncosh	0.00E+00	0.04	✗	441.09
无误差组函数平均运行时间		0.19		758.4
所有函数平均运行时间		0.34		585.8

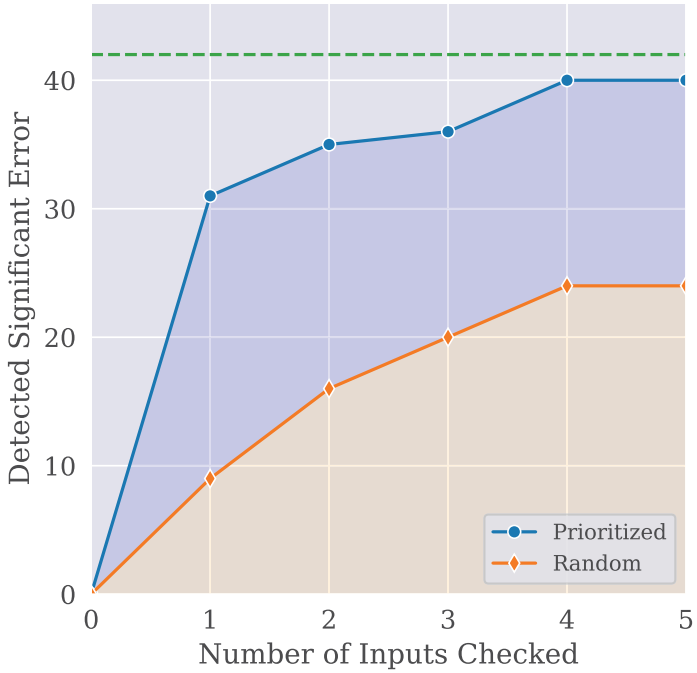


图 4.3　搜索结果排序效果

Atomu 的运行开销

由于 Atomu 的分析过程不涉及测试预言的调用，只在分析结束后需要测试预言对结果进行最终验证。因此相比于已有工作，Atomu 理论上的运行开销要小的多。本章通过实验验证了上述讨论。

已有工作中，DEMC [77] 使用了全局的 MCMC 搜索算法，并且也在 GSL 实验对象上进行了实验。由于实验对象都是 GSL 函数，本章将 Atomu 的运行效率与 DEMC 进行对比，验证 Atomu 的运行效率。

实验结果表明，Atomu 在每个 GSL 函数上，平均需要 0.34 秒进行分析，需要 0.09 秒调用测试预言对结果进行验证。因此，分析一个函数的总耗时为 0.43 秒。相比之下，DEMC 分析一个 GSL 函数的平均耗时为 585.8 秒。Atomu 比 DEMC 的运行速度快了 1362 倍。同时，由于 Atomu 并不指定搜索的区间，对于任何函数都假定其初始搜索空间为 $(-\infty, +\infty)$，而 DEMC 需要对每个函数指定搜索空间，减小其搜索范围。例如，对于 `gsl_sf_eta` 函数，DEMC 只在 $(-168, 100)$ 的范围内进行搜索。因此，该项实验并非公平的比较，其显著偏向于 DEMC。即使如此，Atomu 仍然比 DEMC 的运行效率高了 1000 倍以上。

4.6.3　实例分析

本小节通过具体分析一个实验对象函数 `gsl_sf_lngamma(x)`，来展示 Atomu 是如何对实践中的浮点数程序进行分析的。

按照 GSL 的文档[②]，上述函数在 $x > 0$ 时计算了对数 Gamma 函数，即 $\log(\Gamma(x))$；对于 $x \leq 0$ 的情况，上述函数将返回 $\log(\Gamma(x))$ 的实数部分，等价与 $\log(|\Gamma(x)|)$。该函数使用了数值分析中的 Lanczos 算法计算 Gamma 函数 [41]。

Atomu 的分析结果显示，对于上述函数，其输入为 –2.457024738220797 时会使得运算结果具有 30.6% 的误差。为了进一步分析误差产生的原因以及解释 Atomu 分析的原理，本节手动对上述函数的代码进行了分析。该函数的简化代码如下所述：

```
1  // inaccurate at x = -2.457024738220797
2  double gsl_sf_lngamma(const double x) {
3    // x < 0 while x is not near an integer.
4    if ( ... ) {
5      double z = 1.0 - x;
6      double lngamma_z = lngamma_lanczos(z);
7      double val = LOG_PI - (log(fabs(sin(PI*z))) + lngamma_z);
8      return val;
9    }
10   else { ... }
11 }
```

首先，本节讲解释上述代码的逻辑。Lanczos 算法 [41] 是一种用于计算 Gamma 函数以及对数 Gamma 函数的迭代算法，该算法只在 $x > 0$ 时有效。为了计算 x 为负数时的对数 Gamma 函数，GSL 的开发者使用了欧拉反射方程（Euler's reflection formula）[64] 来避免上述问题：

$$\Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin(\pi z)}, z \notin \mathbb{Z} \tag{4.4}$$

因此，使用 z 的对数 Gamma 函数的运算结果可以间接的得到 x 的对数 Gamma 函数的结果，$z = 1 - x$。下述公式描述了这一推导过程：

$$\begin{aligned}
&\log\left(\left|\Gamma(x)\right|\right) \\
&= \log\left(\left|\Gamma(1-z)\right|\right), x = 1 - z, x < 0, x \notin \mathbb{Z} \\
&= \log\left(\left|\frac{\pi}{\Gamma(z)\cdot\sin(\pi z)}\right|\right) \\
&= \log(\pi) - \log\left(\left|\Gamma(z)\cdot\sin(\pi z)\right|\right) \\
&= \log(\pi) - \left(\log(|\Gamma(z)|) + \log(|\sin(\pi z)|)\right)
\end{aligned} \tag{4.5}$$

② https://www.gnu.org/software/gsl/doc/html/specfunc.html#gamma-functions

可以看到，公式 4.4 与上述 `gsl_sf_lngamma(x)` 的代码逻辑相一致：其在第 6 行使用 Lanczos 算法计算 $\log(|\Gamma(z)|)$，然后在第 7 行使用与公式 4.4 中的推导结果完全一致的方式计算 $\log(|\Gamma(x)|)$。

其次，本节分析 Atomu 是如何生成 −2.457024738220797 这一会触发显著误差的测试用例的。在上述代码中，运行时信息如下：

1. `LOG_PI` = 1.1447298858494002 是一个常数；
2. `lngamma_z` = 1.153871662795107**8** 携带了一个 1.58×10^{-15} 级别的误差；
3. `log(fabs(sin(PI*z)))` = -0.009141776945711**324** 携带了一个 4.66×10^{-15} 级别的误差；
4. `tmp = lngamma_z + log(fabs(sin(PI*z)))` = 1.144729885849**364** 携带了一个 1.50×10^{-15} 级别的误差；
5. `val = LOG_PI - tmp` = **3.774758283725532**e-15 携带了一个 3.06×10^{-1} 级别的显著误差。对于最后的减法操作，其原子状态函数为 6.06×10^{14}。任何 `tmp` 携带的微小误差都会被该运算显著放大。

当 Atomu 在最后的减法运算上搜索时，会不断生成测试用例，搜索最大的原子状态函数，并且最终发现 −2.457024738220797 这一输入会触发最大的原子状态函数。

最后，本节注意到 `gsl_sf_lngamma(x)` 这一函数也在 DEMC 的实验对象中，但是 DEMC 并未在该函数上发现显著的误差。该结果表明本章提出的白盒分析方法具有更强的误差检测能力。

本实例分析表明，一些实践中广泛应用的科学计算库，即使其长期由专业开发者进行开发与维护，并使用了各种数值分析方法进行优化——例如 Lanczos 算法以及欧拉反射方程等，其仍然受到浮点数误差问题的挑战。

4.7 讨论与小结

浮点数误差分析是浮点数程序测试与分析中的关键步骤。由于已有的工作严重依赖于测试预言，而获取测试预言的运算获取开较大，降低了已有的工作在实践中的可用性。

本章通过深入分析浮点数误差的特点，提出了“原子状态函数”这一概念，用于分析浮点数误差的引入，积累，及放大过程。构建了基于原子状态函数的误差积累模型，并基于该模型提出了针对浮点数误差问题的分析技术 Atomu。该技术可以在白盒场景下对浮点数程序进行分析，生成会触发显著误差的测试用例，并找到导致该误差

的浮点数运算。Atomu 运算开销低，分析效果好，不依赖于测试预言。实验结果表明，Atomu 具有良好的误差分析能力，其可以在实际项目中发现大量的误差缺陷，比已有的工作提高了 40%，同时运行效率提高了 1000 倍以上。

本章提出的误差分析工具在本文中作为白盒场景下的新搜索目标函数和搜索方法，成为浮点数程序误差测试与分析的关键技术，解决浮点数误差的稀疏性问题。

第五章　结论和展望

5.1　本文工作总结

浮点数是编程语言中必不可少的数据类型，尤其大量应用于数值运算、科学计算类型的程序中，因而被广泛应用于科学、工程、金融等领域。浮点数误差问题是软件工程与编程语言领域的重要问题。在浮点数程序中，结果必然存在误差，而这种误差会导致严重的软件错误。在安全攸关软件中，对于浮点数误差的上限有着严格的要求。因此，浮点数程序的测试和分析技术是保障软件安全的关键技术，具有十分重要的研究意义。

由于浮点数特殊的数据结构特点，已有的浮点数程序误差测试与分析技术中主要受到了**未知性**与**稀疏性**这两个问题的挑战。

为了保证浮点数程序的质量，从程序测试和程序分析的角度，尽可能的对浮点数误差进行检测，本文使用了**基于搜索的方法**对浮点数程序误差进行测试与分析，核心目标是找到会触发显著误差的测试用例。基于搜索的软件工程使用启发式搜索的方法解决软件工程问题，其核心包括问题的定义与转化，搜索目标函数的制定，搜索算法的构建等一系列关键技术。针对浮点数误差问题，基于搜索的方法具有下列特点：

- 是一种动态分析方法；
- 可以给出特定的触发显著误差的测试用例；
- 可以参考及利用最优化问题的搜索算法；
- 可以应用于不同的测试与分析场景下。

基于搜索的方法在具有上述特点的同时，也面临未知性与稀疏性问题的挑战：

- 搜索目标函数的制定与获取，受到测试预言*未知性*的问题挑战；
- 搜索算法的效果，受到浮点数误差*稀疏性*问题的挑战。

针对基于搜索的方法面临的上述挑战，本文进行了一系列具有针对性的研究，分别为：

- 高质量的浮点数测试预言获取方法研究。该研究提出了兼容语义模型，解决了语义解释错误问题，可以生成正确且准确的测试预言，建立了广泛适用的搜索目标函数；
- 黑盒场景下的搜索算法研究。该研究提出了特定于浮点数误差问题的高效搜索算法；
- 白盒场景下的新搜索目标函数研究。该研究提出了一种新的搜索目标函数“原

子状态函数”，在白盒场景下提高了搜索的有效性并降低了运行开销。

具体而言，上述三种方法相互关联，解决了各个搜索流程中的关键问题，其关系如图 5.1 中所示。

在黑盒场景下，测试预言生成工作可以直接用于计算误差，作为搜索目标；测试用例生成工作提出了新的搜索算法，以提高搜索的有效性。

在白盒场景下，误差分析方法提出了新的搜索目标，并应用搜索算法对误差进行搜索；同时测试预言生成工作在最终用于对少量分析结果计算误差，起到结果验证的作用。使用新的搜索目标可以提高搜索的有效性；仅在最终验证中计算少量的测试预言，可以降低测试与分析的运行开销。

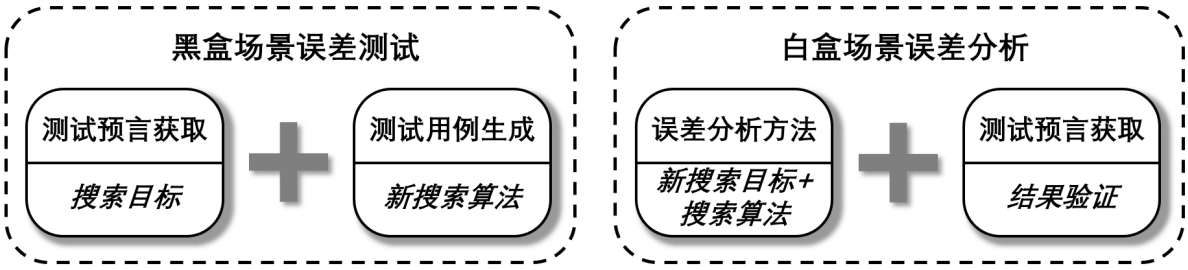


图 5.1　本文提出的各测试与分析技术之间的联系

上述三种方法从不同角度解决了浮点数误差的未知性与稀疏性问题。上述方法可以相互配合，在不同的应用场景下形成互补，最终形成一套基于搜索的浮点数程序误差测试与分析方法。

5.2　未来工作展望

在软件工程中，一个完整的调试过程包括了测试与分析，定位与修复等步骤。由于浮点数误差问题的复杂性，现阶段浮点数误差的定位与修复工作仍大量依赖于开发者手动进行，自动化的定位与修复工作尚只能处理表达式级别的程序 [53]。

因此，针对浮点数误差问题的自动化定位与修复技术是未来的一个重要研究方向。相关技术的发展可以与本文提出的测试与分析技术相结合，使得浮点数误差的调试形成闭环，进一步保证浮点数程序的质量。

本文工作还可以从下述几方面进行深入研究：

- *误差本质原因分析*：本文提出的分析工具 Atomu 可以定位误差被显著放大的运算。然而，在对误差进行修复前，需要首先分析误差产生的根本原因。根本原因包括误差产生的运算，运算最低的精度要求，误差放大的运算等等。通过分析上述因素，可以进一步理解浮点数误差的本质，对浮点数误差的修复以及提高浮点数程序的质量有所帮助。

- 误差的自动修复：在数值分析领域，对于经典的数值运算函数（例如贝塞尔函数），已有大量的数值算法对其进行精确计算，尽可能避免了浮点数的运算误差。然而，相关算法都建立在专业开发者手动构建的基础上。对于任意的浮点数程序，在发现其运算误差后，如何对其进行自动化的修复，以提高其运算精度，是一个尚未解决的挑战。
- 运行时精度保证：本文提出的基于原子状态函数的误差积累模型，可以在不使用高精度类型的条件下，对误差的累积放大过程进行精确的描述。上述模型进而使得不依赖于高精度类型的误差估计成为可能。由于高精度类型的运算开销较大，使用上述模型对误差估计的运行开销将显著低于使用高精度类型的技术，进而使得运行时（时延要求严格的场景）对精度进行保证成为可能。
- 低误差程序生成：程序生成（program synthesis） [31, 46] 的目标是生成符合规约（specification）的程序。如果将规约定义为运算误差不超过一定的阈值，程序生成技术可用于生成保证计算精度的浮点数程序。在此过程中，将涉及到对生成的浮点数程序进行误差分析，等价变化，规约优化等一系列问题。

在将来，将进一步对本文所提出的技术进行拓展，尝试解决上述挑战，用于保证与提高浮点数程序的质量。

参考文献

[1] Emile Aarts, Emile HL Aarts and Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, **2003**.

[2] Christophe Andrieu, Nando De Freitas, Arnaud Doucet *et al.* "*An introduction to MCMC for machine learning*". *Machine learning*, **2003**, *50*(1-2): 5–43.

[3] Marc Andrysco, David Kohlbrenner, Keaton Mowery *et al.* "*On subnormal floating point and abnormal timing*". In: *2015 IEEE Symposium on Security and Privacy*. **2015**: 623–639.

[4] Thomas Bäck, David B Fogel and Zbigniew Michalewicz. *Evolutionary computation 1: Basic algorithms and operators*. CRC press, **2018**.

[5] Roberto Bagnara, Matthieu Carlier, Roberta Gori *et al.* "*Exploiting Binary Floating-Point Representations for Constraint Propagation*". *INFORMS Journal on Computing*, **2016**, *28*(1): 31–46.

[6] James Edward Baker. "*Adaptive selection methods for genetic algorithms*". In: *Proceedings of an International Conference on Genetic Algorithms and their applications*. **1985**: 101–111.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D' elia *et al.* "*A survey of symbolic execution techniques*". *ACM Computing Surveys (CSUR)*, **2018**, *51*(3): 50.

[8] Tao Bao and Xiangyu Zhang. "*On-the-fly detection of instability problems in floating-point program execution*". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, **2013**: 817–832.

[9] Earl T Barr, Thanh Vo, Vu Le *et al.* "*Automatic detection of floating-point exceptions*". In: *ACM Sigplan Notices*. **2013**: 549–560.

[10] Florian Benz, Andreas Hildebrandt and Sebastian Hack. "*A dynamic program analysis to find floating-point accuracy problems*". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM, **2012**: 453–462.

[11] Léon Bottou. "*Stochastic gradient descent tricks*". In: *Neural networks: Tricks of the trade*. Springer, **2012**: 421–436.

[12] Cristian Cadar, Daniel Dunbar and Dawson R. Engler. "*KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, **2008**: 209–224.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler *et al.* "*KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.*" In: *OSDI*. **2008**: 209–224.

[14] Cristian Cadar and Koushik Sen. "*Symbolic execution for software testing: three decades later.*" *Commun. ACM*, **2013**, *56*(2): 82–90.

[15] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric *et al.* "*Efficient search for inputs causing high floating-point errors*". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. ACM, **2014**: 43–52.

[16] John Joseph Chilenski and Steven P Miller. "*Applicability of modified condition/decision coverage to software testing*". *Software Engineering Journal*, **1994**, *9*(5): 193–200.

[17] Eva Darulova and Viktor Kuncak. "*Trustworthy Numerical Computation in Scala*". In: *Proc. OOPSLA*. **2011**: 325–344.

[18] Eva Darulova and Viktor Kuncak. "*Sound compilation of reals*". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, **2014**: 235–248.

[19] Leonardo De Moura and Nikolaj Bjørner. "*Z3: An efficient SMT solver*". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. **2008**: 337–340.

[20] Theodorus Jozef Dekker. "*A floating-point technique for extending the available precision*". *Numerische Mathematik*, **1971**, *18*(3): 224–242.

[21] Saikat Dutta, Owolabi Legunsen, Zixin Huang *et al.* "*Testing probabilistic programming systems*". In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. **2018**: 574–586.

[22] Mohamed El Yafrani and Belad Ahiod. "*Population-based vs. single-solution heuristics for the travelling thief problem*". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. **2016**: 317–324.

[23] Sebastian Elbaum, Alexey G Malishevsky and Gregg Rothermel. "*Test case prioritization: A family of empirical studies*". *IEEE transactions on software engineering*, **2002**, *28*(2): 159–182.

[24] François Févotte and Bruno Lathuiliere. "*Verrou: Assessing floating-point accuracy without recompiling*". **2016**.

[25] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre *et al.* "*MPFR: A multiple-precision binary floating-point library with correct rounding*". *ACM Transactions on Mathematical Software (TOMS)*, **2007**, *33*(2): 13.

[26] Zhoulai Fu, Zhaojun Bai and Zhendong Su. "*Automated backward error analysis for numerical code*". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, **2015**: 639–654.

[27] Zhoulai Fu and Zhendong Su. "*Achieving high coverage for floating-point code via unconstrained programming*". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, **2017**: 306–319.

[28] Patrice Godefroid, Michael Y. Levin and David A. Molnar. "*Automated Whitebox Fuzz Testing*". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, **2008**.

[29] David Goldberg. "*What Every Computer Scientist Should Know About Floating-Point Arithmetic*". *ACM Comput. Surv.* **1991**, *23*(1): 5–48.

[30] Eric Goubault and Sylvie Putot. "*Static Analysis of Numerical Algorithms*". In: *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*. Springer, **2006**: 18–34.

[31] Sumit Gulwani. "*Dimensions in program synthesis*". In: *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. **2010**: 13–24.

[32] Neelam Gupta, Aditya P Mathur and Mary Lou Soffa. "*Generating test data for branch coverage*". In: *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. **2000**: 219–227.

[33] Timothy J. Hickey, Qun Ju and Maarten H. van Emden. "*Interval arithmetic: From principles to implementation*". *J. ACM*, **2001**, *48*(5): 1038–1068.

[34] Nicholas J. Higham. *Accuracy and stability of numerical algorithms, Second Edition*. SIAM, **2002**.

[35] James Kennedy. "*The particle swarm: social adaptation of knowledge*". In: *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*. **1997**: 303–308.

[36] James Kennedy and Russell Eberhart. "*Particle swarm optimization*". In: *Proceedings of ICNN'95-International Conference on Neural Networks*. **1995**: 1942–1948.

[37] Mohd Ehmer Khan. "*Different forms of software testing techniques for finding errors*". *International Journal of Computer Science Issues (IJCSI)*, **2010**, *7*(3): 24.

[38] James C King. "*Symbolic execution and program testing*". *Communications of the ACM*, **1976**, *19*(7): 385–394.

[39] Morris Kline. *Calculus: an intuitive and physical approach*. Courier Corporation, **1998**.

[40] John R Koza. "*Survey of genetic algorithms and genetic programming*". In: *Wescon conference record*. **1995**: 589–594.

[41] Cornelius Lanczos. "*A precision approximation of the gamma function*". *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, **1964**, *1*(1): 86–96.

[42] Chris Lattner and Vikram Adve. "*LLVM: A compilation framework for lifelong program analysis & transformation*". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* **2004**: 75–86.

[43] Daniel Liew, Daniel Schemmel, Cristian Cadar *et al.* "*Floating-point symbolic execution: A case study in N-version programming*". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. **2017**: 601–612.

[44] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue *et al. Ariane 5 flight 501 failure report by the inquiry board*. European space agency Paris, **1996**.

[45] Sandra Loosemore, Richard M Stallman, Rolandand McGrath *et al.* "*The GNU C Library Reference Manual*". **2019**. https://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html.

[46] Zohar Manna and Richard J Waldinger. "*Toward automatic program synthesis*". *Communications of the ACM*, **1971**, *14*(3): 151–165.

[47] John C Mason and David C Handscomb. *Chebyshev polynomials*. Chapman and Hall/CRC, **2002**.

[48] Joan C. Miller and Clifford J. Maloney. "*Systematic mistake analysis of digital computer programs*". *Commun. ACM*, **1963**, *6*(2): 58–63.

[49] Ramon E Moore. *Interval analysis*. Prentice-Hall Englewood Cliffs, NJ, **1966**.

[50] Ramon E. Moore. *Methods and applications of interval analysis*. SIAM, **1979**.

[51] Nicholas Nethercote and Julian Seward. "*Valgrind: a framework for heavyweight dynamic binary instrumentation*". In: *ACM Sigplan notices*. **2007**: 89–100.

[52] Srinivas Nidhra and Jagruthi Dondeti. "*Black box and white box testing techniques-a literature review*". *International Journal of Embedded Systems and Applications (IJESA)*, **2012**, *2*(2): 29–50.

[53] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox *et al.* "*Automatically improving accuracy for floating point expressions*". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, **2015**: 1–11.

[54] Athanasios Papoulis and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Tata McGraw-Hill Education, **2002**.

[55] Peter Larsson. *Exploring Quadruple Precision Floating Point Numbers in GCC and ICC*. **2013**, [Online; accessed 24-June-2019].

[56] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi *et al.* "*CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries*". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE*. **2019**: 1027–1038.

[57] Sylvie Putot, Eric Goubault and Matthieu Martel. "*Static Analysis-Based Validation of Floating-Point Computations*". In: *Numerical Software with Result Verification, International Dagstuhl Seminar, Dagstuhl Castle, Germany, January 19-24, 2003, Revised Papers*. Springer, **2003**: 306–313.

[58] Kevin Quinn. "*Ever had problems rounding off figures*". *This stock exchange has. The Wall Street Journal*, **1983**: 37.

[59] Dietmar Ratz. "*On extended interval arithmetic and inclusion isotonicity*". *Submitted for publication in SIAM Journal on Numerical Analysis*, **1996**.

[60] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner *et al.* "*Finding root causes of floating point error*". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, **2018**: 256–269.

[61] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner *et al.* "*Finding root causes of floating point error*". In: *ACM SIGPLAN Notices*. **2018**: 256–269.

[62] Eric Schkufza, Rahul Sharma and Alex Aiken. "*Stochastic optimization of floating-point programs with tunable precision*". *ACM SIGPLAN Notices*, **2014**, *49*(6): 53–64.

[63] Koushik Sen, Darko Marinov and Gul Agha. "*CUTE: a concolic unit testing engine for C*". In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. ACM, **2005**: 263–272.

[64] Richard A Silverman *et al. Special functions and their applications*. Courier Corporation, **1972**.

[65] Robert Skeel. "*Roundoff error and the Patriot missile*". *SIAM News*, **1992**, *25*(4): 11.

[66] Jorge Stol and Luiz Henrique De Figueiredo. "*Self-validated numerical methods and applications*". In: *Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro. Citeseer*. **1997**.

[67] Jorge Stolfi and Luiz Henrique de Figueiredo. "*An introduction to affine arithmetic*". *Trends in Applied and Computational Mathematics*, **2003**, *4*(3): 297–312.

[68] Enyi Tang, Earl Barr, Xuandong Li *et al.* "*Perturbing numerical calculations for statistical analysis of floating-point program (in) stability*". In: *Proceedings of the 19th international symposium on Software testing and analysis*. **2010**: 131–142.

[69] IJ Thompson and AR Barnett. "*Coulomb and Bessel functions of complex arguments and order*". *Journal of Computational Physics*, **1986**, *64*(2): 490–509.

[70] Nikolai Tillmann and Jonathan de Halleux. "*Pex-White Box Test Generation for .NET*". In: *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*. Springer, **2008**: 134–153.

[71] Nigel Tracey, John Clark, Keith Mander *et al.* "*Automated test-data generation for exception conditions*". *Software: Practice and Experience*, **2000**, *30*(1): 61–79.

[72] Peter JM Van Laarhoven and Emile HL Aarts. "*Simulated annealing*". In: *Simulated annealing: Theory and applications*. Springer, **1987**: 7–15.

[73] Ran Wang, Daming Zou, Xinrui He *et al.* "*Detecting and fixing precision-specific operations for measuring floating-point errors*". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. ACM, **2016**: 619–630.

[74] George Neville Watson. *A treatise on the theory of Bessel functions*. Cambridge university press, **1995**.

[75] L Darrell Whitley *et al.* "*The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best.*" In: *Icga*. **1989**: 116–123.

[76] Qian Yang, J Jenny Li and David M Weiss. "*A survey of coverage-based testing tools*". *The Computer Journal*, **2009**, *52*(5): 589–597.

[77] Xin Yi, Liqian Chen, Xiaoguang Mao *et al.* "*Efficient automated repair of high floating-point errors in numerical libraries*". *PACMPL*, **2019**, *3*(POPL): 56:1–56:29.

[78] Li Yujian and Liu Bo. "*A normalized Levenshtein distance metric*". *IEEE transactions on pattern analysis and machine intelligence*, **2007**, *29*(6): 1091–1095.

[79] Zhi-Hui Zhan, Jun Zhang, Yun Li *et al.* "*Adaptive particle swarm optimization*". *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, **2009**, *39*(6): 1362–1381.

[80] Daming Zou, Ran Wang, Yingfei Xiong *et al.* "*A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies*". In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, **2015**: 529–539.

[81] Dan Zuras, Mike Cowlishaw *et al.* "*IEEE Standard for Floating-Point Arithmetic*". *IEEE Std 754-2008*, **2008**: 1–70.

[82] 姜加红, 陈立前 and 王戟. "基于浮点区间幂集抽象域的浮点程序分析". 计算机科学与探索, **2013**, *7*(3): 209–217.

[83] 汤恩义, 苏振东 and 李宣东. “程序数值误差的扰动检测与优化”. 中国科学: 信息科学, **2014**, *44*: 1445–1466.

[84] 王然. 精度特定运算的实证研究和处理方法 [B.S. Thesis]. **2015**.

[85] 赵世忠. “算术表达式的一种可信计算算法及其软件 *ISReal*”. 中国科学: 信息科学, **2016**(006): 698–713.

[86] 邹达明. 一种检测显著浮点数误差的遗传算法 [B.S. Thesis]. **2015**.

[87] Fredrik Johansson *et al. mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.1.0)*. 2018-12, `http://mpmath.org/`.

个人简历及博士期间研究成果

个人简历

邹达明，1993 年 8 月出生于江西省南昌市；2011 年 9 月考入北京大学信息科学技术学院，专业为计算机科学与技术，2015 年 7 月本科毕业并获得理学学士学位与经济学双学位；2015 年 9 月保送进入北京大学信息科学技术学院计算机软件与理论专业攻读硕士学位，2017 年 9 月经过硕转博手续攻读博士学位至今。

发表论文

作为第一作者发表的论文

[1] **Daming Zou**, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. "*Detecting floating-point errors via atomic conditions*," In: *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages, **POPL 2020**, New Orleans, LA, USA, January 19-25, 2020*. 1-27. （**CCF A 类**）

[2] **Daming Zou**, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. "*An Empirical Study of Fault Localization Families and Their Combinations*," In: *IEEE Transactions on Software Engineering, **TSE 2019***. Online First. （**CCF A 类**）

其他发表的论文

[3] Ran Wang, **Daming Zou**, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. "*Detecting and fixing precision-specific operations for measuring floating-point errors*," In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, **FSE 2016**, Seattle, WA, USA, November 13-18, 2016*. 619–630. （**CCF A 类**）

参与课题

1. 基于情境的安全攸关软件的构造方法与运行机理研究，973 计划青年科学家专题项目，No. 2014CB347701。
2. 软件分析与测试，国家自然科学基金杰出青年基金项目，No. 61225007。
3. 数据驱动的软件自动构造与演进方法研究，国家重点研发计划课题，No. 2017YFB1001803。

致谢

时光荏苒，韶华易逝，距离初次踏入燕园的那个秋天，已经过去了九年。最初的记忆在脑海中逐渐模糊，发生的故事却在我的成长中留下了不可磨灭的印记。

首先想感谢的，便是北京大学。这里既是无数前辈留下的北大精神的凝聚，也是现时师生传道授业解惑之所在。在这匆匆又漫长的九年中，在这学术的殿堂中，我得以聆听名家大师的教诲，和优秀的同侪切磋进步，丰富了学识，增进了思想。本想用一句校训概括北大的精神，却意识到北大并无明文的校训，也许就像“未名湖”一样，这种“未名”却又切实存在的北大精神，在千千万万的北大人之间薪火相传，也不断指引着我，让我成长为今天的自己。

衷心感谢杨芙清院士和梅宏院士。两位院士是北京大学乃至全国软件工程研究领域的开拓者，为我国的软件工程事业发展作出了巨大的贡献。我有幸进入他们领导的北京大学软件工程研究所进行学术科研工作，让我在这个最优秀的平台之上，对最前沿的科研问题进行研究，与最优秀的学术同行进行交流合作，在优越与自由的科研环境下，专注于自己感兴趣且有影响力的课题研究。

由衷感谢我的导师张路教授。张老师是我带领我走入科研殿堂的引路人。还记得在本科刚进入实验室的时候，张老师在科研上事无巨细的悉心指导，引导我发挥自己的优势，在科研道路上迅速成长，发表了自己的第一篇学术成果，体会到了科研的乐趣。在我的整个博士阶段，张老师不仅在学术科研上帮助我成长和进步，也在生活上给予我关怀与照顾，让我得以从容自信的面对和解决种种问题。张老师对待科研的严谨态度和深入见解，为人处事的真诚谦虚，我都将铭记于心，并以之为榜样。

由衷感谢我的指导老师熊英飞副教授。在我进入实验室后，一直在熊老师的悉心指导下展开科研工作。在博士生涯的初期，熊老师带我体验学术的全过程，学习了从想法提出到最终成文的每一个步骤与要点；在博士生涯的中期，熊老师联系了海外合作的机会，让我体验了学术合作的乐趣以及提高了自己的英文交流能力；在博士生涯的末期，熊老师鼓励我注重学术的影响力与研究的质量。感谢熊老师在我整个学习生涯中的因材施教，一步步将我培养为一个具有独立性的学者。在生活中，熊老师的善良正直与风趣幽默也时刻感染着我，让我能够自信和坚定的渡过充满挑战的博士生涯。

感谢苏黎世联邦理工学院的苏振东教授。苏老师在本文第三章和第四章的内容形成过程中提出了许多宝贵的指导意见。感谢苏老师给予我出国交换访问的机会，在瑞士交流培养的一年是短暂而快乐的。苏老师对于科研问题的敏锐才思以及高屋建瓴的指导，实验室内自由活跃的学术氛围，都使我受益匪浅，让我对于学术有了更多的兴

趣以及更大的信心。

感谢小组的指导老师郝丹副教授。郝老师是组内每一位同学的榜样，她的自律精神和学术热情让组内的同学受益良多。郝老师对同学们全心全意的指导以及严谨的学术态度，都让我十分敬佩。

感谢金芝老师，胡振江老师，黄罡老师，谢冰老师，陈向群老师，邹艳珍老师，陈淞婕老师，李戈老师，周明辉老师，曹东刚老师，赵俊峰老师，王亚沙老师。他们在我的博士培养过程中，给予了我很多宝贵的指导意见，让我不断进步。

感谢王然同学和曾沐焓同学和我一起进行了浮点数的相关研究。王然同学参与了本文第二章内容的讨论，协助完成了第二章中工具的相关实现；曾沐焓同学协助完成了本文第四章中工具的相关实现。

感谢软工所这个大家庭，感谢软件测试与分析小组的高庆、孟思辰、万馨忆、兰天、朱沐尧、张洁、叶挺、唐浩、臧磊、王彦博、陈俊洁、臧琳飞、胡文翔、娄一翎、冯致远、周建祎、孙泽宇、王冠成、郭翊庆、李丰、孙培艺、朱琪豪、张文杰。感谢程序语言与开发环境小组的悦茹茹、王博、姜佳君、梁晶晶、陈逸凡、章嘉晨、吉如一、吴宜谦、石元峰、叶振涛。谢谢各位同学在科研和生活上对我的帮助和支持。

感谢我的朋友们，骆宇冲，朱纪乐，贾灏，魏亮晨，唐子豪。同窗几年来，我们有太多的欢声笑语，我最为珍惜的，便是与你们的这份真挚的友谊。感谢在这里我们相遇相识，愿友谊地久天长。

感谢我的家人，谢谢爸爸、妈妈、姑姑、叔叔、爷爷奶奶、外公外婆对我无微不至的关心与照顾。感谢他们给了我一个温暖的家庭，一个坚强的后盾。他们在我迷茫时给我努力的方向，在我低落时给我前进的勇气，用他们的智慧，指引我走出更精彩的人生。

我深深地感到自己是幸运的。这一路走来，我受到了太多太多的帮助与支持。把这份温暖而无私的善意传递给更多的人，将是我一生所追求的目标。

时值 2020 年春天，全国乃至全世界人民都在对抗新冠肺炎的疫情。感谢奋战在抗疫一线的医护工作者，感谢他们用自己的血与汗守护着全国人民的安全，他们是值得铭记的伟大英雄。感谢在此次抗疫过程中付出努力的千千万万的人民，众志成城，我们终将取得胜利。

最后，愿祖国繁荣昌盛，人民安居乐业。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因须要延迟发布学位论文电子版，授权学校在 □ 一年 / □ 两年 / □ 三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日