# Numerical Program Analysis and Testing

Zheng Gao
University College London
Gower Street, WC1E 6BT
London, United Kingdom
z.gao.12@ucl.ac.uk

## ABSTRACT

Numerical software is playing an increasingly critical role in modern society, but composing correct numerical programs is difficult. This paper describes a doctoral research program that aims to alleviate this issue. It tackles real world problems and is guided by features learned from empirically studying these programs. By assisting developers in the production of numerical software, it improves the quality and productivity of software development. The research depends on numerical analysis and lies in the intersection of software engineering and program analysis.

## Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]

## Keywords

Numerical program, real encoding, static analysis, testing

## 1. INTRODUCTION

Numerical software, whose behavior is determined by the computational representation of real arithmetic, is increasingly important in many subjects [17]. In physics, numerical software provides the capability to enter disciplines that are either inaccessible to traditional experimentation or where carrying out conventional empirical inquiries is prohibitively expensive.

Some real values, like Pi, have an infinite number of digits. Computers have limited storage and therefore must encode such reals. A real number can be written in a numeric form, like scientific notation, or as an algebraic expression. Computers encode numeric forms using fixed width bit vectors. For instance, they capture scientific notation into three fields, one bit for the sign and two fixed-width bit vectors for the exponent and the significand, since the base, which is always 2, does not need to be encoded [26]. Computer algebra systems represent real numbers symbolically as a syntax tree.

Although finite precision approximation is accurate enough for many applications, numerical programs can output incorrect results or throw arithmetic exceptions. For example, the accumulation of rounding errors in floating-point computations can lead to arbitrarily inaccurate results [8]. The fact that such inaccuracy can cause massive financial and human losses has been well-established since 1982, when the Vancouver Stock Exchange experienced a huge loss. A new index was instituted and initialised to 1000.000. The index was updated after each transaction and 22 months later it had fallen to 520 because of the propagated rounding error [14].

Writing numerical software that does not violate its specification is difficult [13], especially for scientists who have not received formal training in programming [17]. Therefore, it is critical to automatically detect and eliminate such errors. The underlying cause of this problem is that the semantics of floating-point arithmetic is complex and unintuitive. A programming model is a set of abstractions and operations that developers can use to define new abstractions and write algorithms. Another way to put the problem, then, is that the programming model of floating-point lacks *affordance*, a concept we borrow from psychology and here refers to the property of a computer system's easy discoverability of possible actions [19].

My research aims to decide whether floating-point is the well-suited real encoding of a problem, and when it is, improve the affordance of floating-point programming model. I will employ an empirically driven approach; that is, I first identify a real world problem and then extract features from its instances to guide the formulation of analysis and testing techniques. I intend to assist programmers in the production of correct numerical software thereby improving the productivity and quality of software development. The research starts with a thorough survey in the field of numerical program analysis and testing, detailed in Section 2. Section 3 describes an empirical study of different real encodings. I continue in Section 4 to Section 5 introducing two areas of potential concentration. As I progress into my PhD, I will refine and focus my investigation of these areas into a cohesive topic. In Section 6, I conclude by presenting the expected contributions to the area.

## 2. SURVEY

As modern society increasingly relies on numerical software, numerous researchers have contributed to the field and established various theories, such as abstract interpretation, metamorphic testing and satisfiability modulo floating-point arithmetic. However, there has never been a systematic and self-contained survey. The reason for me to begin the doctoral journey with a survey is that it can benefit researchers, including me, by presenting a detailed and objective description of the state of the art. The survey covers three topics: static analysis, floating-point testing and symbolic execution.

### 2.1 Static Analysis

Static analysis has a long tradition of being applied to numerical programs and the general problem is imprecision. Among all

*FSE'14*, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2666603

the techniques in static analysis, abstract interpretation and model checking are the most prominent.

Abstract interpretation is a theory of the approximation of program semantics with properties of interest preserved, established by Cousot and Cousot in 1977 [6]. It rests on two key concepts: the relation between concrete and abstract semantics through Galois connections, and the computation of a fixpoint of the abstract semantics, through the combination of widening and narrowing operators. Since its birth, abstract interpretation has been well-developed and considerable research has been undertaken.

I briefly introduce some research in abstract interpretation to illustrate how the survey is conducted. Abstract domain, the computer representation of program invariants and operators to manipulate them, is the key concept of abstract interpretation. The interval and affine domains are often used to analyse numerical programs. The interval domain is based on interval arithmetic, a classical concept developed by Moore [18], and has been widely used in scientific computing. Interval arithmetic tracks three sources of errors — rounding, truncation and input errors — by setting lower and upper bounds, thereby yielding reliable intervals that the true results lie within. Cousot and Cousot adapted interval arithmetic to the needs of abstract interpretation in 1977 [6]. In this domain, each program variable is bounded by an interval $[a,b]$, where $a$ and $b$ are computer-representable values and $a \leq b$ holds. The interval domain is efficient and easy to implement, but the lost track of relations among variables leads to a severe imprecision. For instance, given a variable $x$ represented by $[5,10]$, we now estimate a function $f(x) = x - x$. Apparently, $f(x) = 0$ holds. However, according to the interval subtraction, $f(x) = [5,10] - [5,10] = [5-10, 10-5] = [-5,5]$, which is considerably inaccurate.

Affine domain which was built to overcome the drawbacks of interval domain also originates from a mathematical theory. Affine arithmetic, introduced by Comba and Stolfi in 1993 [5], is a model for numerical computation in which an unknown numeric quantity $x$ is represented by an affine form $\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n$, where $\varepsilon_i \in [-1,1]$. Elementary operations are defined based on this representation. Goubault first applied affine arithmetic to abstract interpretation in 2001 [12], intending to statically analysing the precision of numerical values with acceptable accuracy. Since then, affine domain has been continuously extended. It now mainly serves to identify computations which introduce significant precision loss and to describe the propagation of rounding errors.

Model checking is an automated technique to exhaustively check whether a given model of a computer system has the specified properties [9]. Though it is extensively applied to hardware designs, according to a case study under the Certification Technologies for Advanced Flight Critical Systems program [25], model checking can effectively find errors at the design stage of software development, even for complex numerical programs. In 2006 Siegel *et al.* proposed an idea of combining model checking and symbolic execution to verify the correctness of parallel numerical programs [22].

Other techniques are emerging. In 2013, Bao and Zhang developed an efficient technique to address instability problems in floating-point programs [1]. By comparing the exponents of the operands, the technique identifies computations where canceled bits occur and marks the result as inaccurate. It tracks inaccurate values and reports instability when the error reaches a critical execution point, such as a predicate. In 2014, Schkufza *et al.* used stochastic search to optimise floating-point programs [21]. This approach repeatedly applies random transformations, including opcode replacement, code swap and instruction replacement. To validate an optimisation, they check whether the rewriting introduces imprecision greater than a user-defined value.

## 2.2 Testing

For decades, testing has dominated software validation in industry. Broadly, testing depends on solving two problems: generating test cases and defining a test oracle, a means of determining whether a program's output on a specific input is correct. Floating-point programs exacerbate both problems. For brevity, I only discuss the test oracle problem in the numerical context. As described in Section 1, floating-point is a finite precision approximation to the real number system, thus testing for equality is problematic. Two computations that are equivalent in real arithmetic may produce different floating-point results. To check the equality in floating-point, a maximum error must be specified. Simply put, if two values are closed enough, we regard them as equivalent. Hence, the definition of test oracles in floating-point depends on a detailed specification, which is often neglected in practice because it is complicated and time-consuming. Regarding floating-point testing, the survey will start with research including Berkeley elementary function test suite [16] which was later extended at Sun Microsystems. The survey also considers metamorphic testing [4] which attempts to alleviate the problem of test oracles by exploiting metamorphic relations.

## 2.3 Symbolic Execution

Symbolic execution is a program analysis technique born in the mid 1970s [15]. It executes programs by supplying symbolic instead of concrete inputs and generates test cases by solving constraints that is collected along the path and updated whenever a branch statement is executed. Though numerous applications have utilised the power of symbolic execution, in addition to the existing issues such as path explosion and external functions, additional problem arises when it is applied to numerical programs. The fact that the Associative and Distributive laws do not hold over floating-point makes constraint rewriting extremely difficult. Thus general purpose solvers that support floating-point have not been implemented. In 2006 Botella *et al.* defined efficient projection functions over floating-point intervals to solve normalised symbolic expressions [3]. Moreover, Lakhotia *et al.* provided a unified framework to solve constraints over floating-point variables by combining symbolic execution and search-based software testing in 2010. The described research will be the preliminary target of the survey.

## 3. REAL ENCODING COMPARISON

Though floating-point has been adopted by a majority of the computer systems, other real encodings, such as symbolic algebra representations [10], Q format [20] and logarithmic number system [24], have different trade-offs and can replace or even outperform floating-point under particular circumstances. Symbolic algebra representation is extensively used in computer algebra systems like Mathematica and Maple, while Q format is often adopted when the hardware does not have a floating-point unit or constant resolution is required. For logarithmic number system, it is suitable for applications in which most arithmetic operations are multiplication or division [11]. However, no empirical study regarding the comparison of popular real encodings has ever been conducted. Can we analyse an arbitrary numerical program and determine an acceptable encoding based on the comparison?

To work on this topic, I will mathematically compare these real encodings. For example, I can establish a function which maps every floating-point representation to a mathematically equivalent one in Q format. Regarding these two encodings, I can undertake a theoretical comparison over the supporting range and intervals between adjacent values. Afterwards, I will choose a set of sample programs in which at least one encoding is likely to outperform others. For example, I will select programs from FPGA-based applications where multiplication and division are frequently used for logarithmic number system. With respect to Q format, the source will possibly be applications for DSP fixed-point operation where floating-point tends to perform worse. Based on these sample codes, I will actually run them using different real number encodings and record the statistics such as computation time, accuracy and memory consumption. From the set of programs that solve some numerical problems, it is undecidable to choose the best encoding in terms of accuracy and compactness. Thus I aim to identify a nontrivial subset of actually occurring programs that people actually care about. Within the subset which is amenable to the proposed analysis, a tool that is able to choose the optimal encoding can be built. Eventually, I will validate the tool by inputting programs outside of the corpus and comparing the result with manual selection.

A key difficulty of the proposed study will be generating real-world inputs for the programs. Generally falling back on a uniform distribution over a program's input domain can, in principle, arbitrarily distort, and therefore invalidate, my findings. To capture these inputs, there are several possible methods. First, I observe that some programs document the subset of their domain that their authors designed them to consume. I will seek out such documentation, as it appears in comments or conditionals in the source code. Second, I will favour programs that have associated test cases: clearly, the program's developers believed these inputs to be likely, or at least possible, in the program's operating environment. Third, similar to metamorphic testing introduced in Section 2, I will, where possible, exploit mathematical identities, like $\sin^2 x + \cos^2 x = 1$, as test oracles to generate inputs that a program that implements such an identity must represent. I can also utilise reverse symbolic execution or Dijkstra's weakest precondition [7] to solve the input set with the premise of knowing the exit point of the program.

The following examples briefly demonstrate how such a tool might work. The tool can calculate the percentage of multiplication and division in all arithmetic operations to decide whether to adopt logarithmic number system. As particular embedded systems do not have floating-point unit, the tool can also analyse the program including variable names, comments and even documents along with the code to decide whether it is targeting at these embedded systems, thereby making decisions upon whether to apply Q format.

## 4. FLOATING-POINT EXCEPTIONS

According to the IEEE 754 standard, floating-point exceptions are divided into five categories: overflow, underflow, finite division by zero, inexact and invalid [26]. The meaning of the first three exceptions is self-explanatory. Inexact is thrown when a real number cannot be represented accurately; that is, it lies between two floating-point values and therefore needs to be rounded. Invalid covers situations where undefined operations occur, such as $\infty + (-\infty)$, $0 * \infty$, $\infty/0$, and $\sqrt{x}$ where $x < 0$.

As discussed in Section 1, floating-point exceptions can be catastrophic. In 2013 Barr *et al.* presented Ariadne, a symbolic execution engine for automatically identifying inputs which trigger floating-point exceptions in a given C/C++ program [2]. The fundamental idea is to convert the task of exception detection into a reachability problem. Specifically, Ariadne adds a conditional guard before each floating-point computation, symbolically executes the transformed program over arbitrary precision rationals and checks whether the exception is reachable. Ariadne rests on a hypothesis that dense neighborhoods of exception-triggering inputs exist because it approximates floating-point using arbitrary precision rationals. However, this hypothesis has not been validated. Additionally, Ariadne only analyses C/C++ programs and supports scalars. This restriction undermines its programming model.

I plan to refactor, extend, and release Ariadne, because it will support and validate my future research. Scientists will also benefit from the extension by using Ariadne to analyse their numerical programs. First, I will undertake an experiment to compare Ariadne against fuzz testing in terms of the number of exceptions found in a given period of time. Next, I intend to enhance the language support of Ariadne. Among the rest of active languages, Fortran and Python have the highest priorities, because the former has been used in a majority of the legacy numerical systems and the latter is currently regarded as a popular scripting language for scientific computing. Finally, I will add support for non-scalars, such as pointers and arrays, to Ariadne.

## 5. EXPRESSION REWRITING

Mathematical expressions that are encoded by floating-point can trigger arithmetic exceptions. Because of the poor affordance of floating-point programming model, inexperienced programmers tend to have an illusion that they are coding in real arithmetic, thereby neglecting cases where intermediate computation underflows or overflows. The research on automatic detection of floating-point exception by Barr *et al.* [2] shows such cases are nontrivial. Rewritings of a single expression using the laws of algebra — the Commutative, Associative and Distributive laws — are equivalent over real arithmetic by definition, but may be nonequivalent over floating-point arithmetic. If the result of an expression that throws exceptions can be represented by floating-point, can we always rewrite it to an equivalent form that is exception-free?

I plan to investigate rewriting these expressions to eliminate their intermediate under- or over- flows. Since the number of possible rewritings resulted from applying the Associative and Commutative law is the Catalan Number [23] and permutation, respectively, brute force testing every possible rewriting is unfeasible. However, the set of mathematical expressions in programs that developers actually compose to tackle actual problems may be small enough to be solvable. I propose to validate the feasibility of the rewriting by undertaking an empirical study of the expressions extracted from a corpus of real world numerical software.

Assuming that such a rewriting is viable, I will build a tool to automatically identify and extract mathematical expressions from the control flow graph (CFG) of a given numerical program. This plan is based on an assumption that accurate CFGs have been constructed, which is, in fact, challenging because of two reasons. First, building a precise CFG is undecidable because the conditional expression itself may be undecidable. The second problem is related to indirect jumps, where the address of the instruction to be executed next, instead of being a constant, is located in registers or memory.

To efficiently search for the acceptable rewriting in such a vast space, I propose to utilise a genetic algorithm, because it only needs a fitness function to guide the search and currently the order of the algebra laws applied to transform the expression is not clear. How to appropriately encode the candidate solution and accurately define a fitness function remains challenging. Considering the specific working procedure of this tool, it will automatically create a new program which only contains necessary definitions and one possible rewriting from the sample space, simulate a running process and measure properties such as the occurrence of arithmetic exception, computation time and data accuracy. These properties act as parameters of the fitness function. The entire process will be repeated until a solution has been selected to replace the original one.

I will conduct a thorough evaluation by running a set of sample programs which are composed of different kinds of complex arithmetic computations with and without the involvement of this tool and compare the statistics.

## 6. CONCLUSION

By the end of my PhD study, I hope to make contributions to the field of numerical program analysis and testing as following:

- Provide a systematic and self-contained survey of the existing research in the area.
- Release the extended Ariadne to benefit researchers who need to check the correctness of their numerical programs.
- Present validated instructions to programmers of which real encoding is optimal under certain circumstances.
- Propose an approach to automatically rewrite mathematical expressions to eliminate floating-point exceptions caused by some intermediate computations.

This research depends on numerical analysis and lies in the intersection of software engineering and programming language. I welcome feedback on how to best present it for publication in a software engineering venue.

## 7. ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Earl T. Barr, for his valuable feedback on early drafts of this paper. I would also like to thank Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su for their hard work on Ariadne.

## 8. REFERENCES

[1] T. Bao and X. Zhang. On-the-fly detection of instability problems in floating-point program execution. In *OOPSLA 2013*, pages 817–832. ACM, 2013.

[2] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *POPL 2013*, pages 549,560. ACM, 2013.

[3] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97,121, June 2006.

[4] T. Y. Chen, S. C. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. *Department of Computer Science, Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01*, 1998.

[5] J. L. D. Comba and J. Stol. Affine arithmetic and its applications to computer graphics. In *Proceedings of VI SIBGRAPI (Brazilian Symposium on Computer Graphics and Image Processing)*, pages 9–18. Citeseer, 1993.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, pages 238–252. ACM Press, 1977.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[8] A. Edalat and P. John Potts. A new representation for exact real numbers. *Electronic Notes in Theoretical Computer Science*, 6:119–132, 1997.

[9] E. A. Emerson and E. M. Clarke. *Characterizing correctness properties of parallel programs using fixpoints*. Springer, 1980.

[10] G. E. Forsythe, C. B. Moler, and M. A. Malcolm. Computer methods for mathematical computations. *Printice-Hall, Englewood Cliffs, NJ*, 1977.

[11] H. Fu, O. Mencer, and W. Luk. Comparing floating-point and logarithmic number representations for reconfigurable acceleration. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 337–340. IEEE, 2006.

[12] E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis*, pages 234–259. Springer Berlin Heidelberg, 2001.

[13] J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):139–174, 1996.

[14] D. B. Henriques. High risks don't deter U.S. funds from Vancouver deals. The New York Times, 08 1994.

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

[16] Z. A. Liu and W. Kahan. Berkeley elementary function test suite, 1987.

[17] Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467(7317):775–777, 2010.

[18] R. E. Moore. *Interval analysis*, volume 4. Prentice-Hall Englewood Cliffs, 1966.

[19] D. A. Norman. *The design of everyday things*. Basic books, 2002.

[20] E. L. Oberstar. Fixed-point representation and fractional math. *Report Oberstar Consulting*, 2007.

[21] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI 2014*, page 9. ACM, 2014.

[22] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 157,168. ACM, 2006.

[23] R. P. Stanley. Catalan addendum, 2008.

[24] E. E. Swartzlander and A. G. Alexopoulos. The sign/logarithm number system. *IEEE Transactions on Computers*, 24(12):1238–1242, 1975.

[25] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *Formal Methods for Industrial Critical Systems*, pages 68–84. Springer, 2008.

[26] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, et al. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.