Automatically Improving Floating Point Accuracy with Casio

Abstract

Many applications from science and engineering depend on floating point hardware to efficiently approximate computations over real numbers. Unfortunately, such approximations introduce rounding error, which can accumulate to produce unacceptable results. When the largest hardware-supported precision does not provide sufficient accuracy, developers may be forced to simulate arbitrary precision floating point arithmetic in software, incurring orders of magnitude slowdown. While the numerical methods literature provides techniques to mitigate rounding error without increasing precision, these techniques often require both understanding the details of floating point arithmetic and also manually rearranging computations.

We introduce Casio, which automatically improves floating point accuracy by searching for error-reducing program transformations. This search is guided by techniques for estimating rounding error, finding its sources, and dividing the input space into regimes with distinct error behavior, allowing Casio to select the best program variant for each regime. We evaluated Casio on examples from a classic numerical methods textbook and found that it improved accuracy in 21 out of 28 tests by up to 27 bits while imposing an average performance overhead of 45%. We also present a series of case studies using a formulas found in the wild to show that Casio has broad applicability.

1. Introduction

Floating point inaccuracy is notoriously difficult to detect and debug [20, 19]. Rounding errors have lead to irreproducibility and even retraction of scientific articles [3, 5, 4], legal regulations in finance [12], distorted stock market indices [22, 24], and skewed election results [27]. Many applications that must produce accurate results, including physical simulators and statistical packages, depend on floating point hardware to efficiently approximate computations over real numbers. While such approximations make these computations feasible, they also introduce rounding error, which can lead to unacceptable *accuracy*, that is, the approximate results may differ from the ideal real number results by an unacceptable margin.

In practice, developers often respond to rounding error by increasing *precision*, the size of the floating point representation. For example, a developer might try to shift error to lower order bits by replacing all 32-bit single precision floats with 64-bit double precision floats. Unfortunately, even the largest hardware-supported precision may still exhibit unacceptable rounding error, and further increases to precision require simulating arbitrary precision floating point in software, incurring orders of magnitude slowdown*.

The numerical methods literature provides techniques to analyze and mitigate rounding error without increasing precision. Forward and backward error analysis [16, 18] can be applied to measure the error of a program, and various program transformations [15] can be applied to improve accuracy. Unfortunately, these techniques often require both understanding the subtle details of floating point arithmetic and also manually rearranging computations.

As a first step toward addressing these challenges, we introduce Casio, a technique for automatically mitigating rounding error in floating point computations. Casio searches for error-reducing program transformations by combining three novel techniques. First, Casio determines which operations are responsible for rounding error by sampling floating point inputs and comparing the results of intermediate operations against accurate results computed using arbitrary precision floating point. Second, Casio applies a database of rewrite rules to the error's source to produce program variants that are evaluated to determine which variants have higher accuracy than the original program. Third, Casio detects when two or more program variants have complementary error behavior on different parts of the input space and combines these programs to improve overall accuracy. Thus Casio improves accuracy automatically, without requiring training or manual effort from the programmer.

We evaluate Casio on examples drawn from a classic numerical methods textbook and consider its broader applicability to floating point expressions extracted from a mathematical library as well as formulas from recent scientific articles. Our results demonstrate that Casio can effectively discover transformations that substantially improve accuracy (by up to 27 bits) while imposing much lower overhead than software floating point (only 45% on average). Furthermore, Casio has already been applied by colleagues in machine learning who were able to significantly reduce rounding error that arose when computing probabilities in a clustering algorithm.

The contributions of this paper include:

- The first automated approach to improving floating accuracy by rearranging computations;

*Even arbitrary precision floating point can exhibit rounding error, and so the developer must still carefully select a precision that provides sufficient accuracy.

- A definition of *real-equivalence* which provides correctness criteria for accuracy-improving transformations;
- A heuristic search for accuracy-improving program transformations whose sampling-based approach relies on real-equivalence to avoid overfitting.

The rest of the paper is organized as follows. Section 2 provides a brief background on floating point arithmetic. Section 3 illustrates Casio on a representative example and describes the high level Casio workflow. Section 4 details Casio's heuristic search and error estimation techniques. Section 5 evaluates Casio's effectiveness at improving accuracy and measures Casio's performance overhead. Section 6 surveys the most closely related work while Section 7 considers future directions for Casio.

2. Floating Point Background

Floating point numbers are a bounded-size approximation to the set of real numbers. Floating point numbers represent real numbers of the form

$$\pm(1+m)2^e,$$

where m, the *mantissa*, is a fixed-width number in $[0,1]$, and e, the *exponent*, is a fixed-width signed integer[†]. Floating point numbers come in a variety of precisions; for example, IEEE 754 double-precision floats are represented by a sign bit, a 52 bit mantissa, and an 11 bit exponent, while single-precision floats are represented by a sign bit, a 23 bit mantissa, and an 8 bit exponent. Note that, to help prevent underflow and overflow, floating point numbers are distributed roughly exponentially; this must taken into account when sampling floating point values.

Floating point operations are executed under a *rounding mode*, a function from real to floating-point numbers. Let $F(r)$ denote the rounded floating point value of real number r and $R(f)$ denote the real number represented by the floating point value f. The rounding mode guarantees both $\mathsf{F}(\mathsf{R}(x)) = x$ and that $\mathsf{R}(\mathsf{F}(x))$ is "close" to x[‡].

2.1. Error of primitive operations

Since a floating point value can only exactly represent a real number of the form $\pm(1+m)\times 2^e$, the conversion F inherent in primitive arithmetic operations must introduce some error. For real numbers neither too large nor too small—that is, whose logarithm in base 2 is within the range $(-2^{N_e-1}, 2^{N_e-1}-1)$—this error is only due to insufficient precision in the mantissa. Thus, the error is approximately 2^{-N_m} times smaller than the output itself. We write $\mathsf{F}(x) = x + x\epsilon$, where ϵ is of absolute value less than 2^{-N_m}, and where applications of F to different inputs will result in different errors ϵ.

Primitive arithmetic operators on floating point numbers such as addition and multiplication are guaranteed to produce accurate results. For example, the floating point sum $x+y$ of floating point values x and y is guaranteed to be equal to the real-number sum of x and y, rounded: $\mathsf{F}(\mathsf{R}(x)+\mathsf{R}(y))$. This also guarantees that floating point addition is commutative and has inverses (since 0 is exactly representable). The addition $x+y$ of two floating point values x and y thus has value $x+y+(x+y)\epsilon$. Since ϵ is a very small quantity, this is normally very accurate. However, because each application of F introduces a new error ϵ, the error in a floating-point computation is non-compositional.

2.2. Accumulating error

Operators such as exponentiation or the trigonometric functions are not computed by hardware and must be implemented by libraries. Due to the table maker's dilemma [21], these more complex functions cannot provide similar accuracy. Instead, an implementation of a mathematical function $f(x_1, x_2, \dots)$ typically guarantees that its result is among the u closest floating point values to the exact result $\mathsf{F}(f(\mathsf{R}(x_1), \mathsf{R}(x_2), \dots)))$. u is typically less than 8; that is, a library implementation usually guarantees that all but the two or three least-significant bits are correct.

Even though primitive floating-point operations are largely accurate, formulas that combine these operators can nonetheless exhibit arbitrarily large inaccuracies. For example, consider the expression $(x+1)-x$. The addition on the left produces $x+1+(x+1)\epsilon_1$. (1.0 is exactly representable, so $\mathsf{F}(1.0) = 1$.) Then the subtraction then produces $1+(x+1)\epsilon_1+\epsilon_2+(x+1)\epsilon_1\epsilon_2$. The terms ϵ_2 and $(x+1)\epsilon_1\epsilon_2$ are small compared to the true value 1, but $(x+1)\epsilon_1$ may not be if x is very large. Thus, for large values of x, the expression $(x+1)-x$ may have very large error. Depending on the rounding mode, it may equal 0, or it might represent some relatively large quantity. The same phenomenon occurs with real-world formulas, such as the quadratic formula (see Section 3).

2.3. Rearrangement

To compute formulas which combine primitive operators, programmers must their rearrange computations to avoid accumulating error. These rearrangements rely on identities which are true for real-number formulas, yet false on floating-point numbers. For example, the identity $(x+y)+z = x+(y+z)$ is true of real numbers and false

[†] IEEE 754 floating point also represents a few special values: positive and negative *infinity*, positive and negative *zero*, *not-a-number* error values, and *denormal* numbers of form $\pm m2^{e_{\min}}$.

[‡] Applications can choose between either rounding up, down, or toward zero; or rounding to the mathematically closest value, with ties breaking either toward the value with a zero least significant bit, or away from zero

for floats; applying it allows converting $(1+x)-x$ into 1, which is exactly accurate.

The necessary rewrites can be unintuitive. William Kahan provides the example [19] of computing $(e^x-1)/x$. The unintuitive trick is to change the denominator from x to $\log e^x$. Such changes can stymie backward error analysis and similar numerical tools [19], making them difficult to find even for experts.

3. Overview

We overview Casio's heuristic search for accuracy-improving transformations using a familiar and representative example, the quadratic formula. Consider this case to compute the smallest real root of a quadratic equation:

$$\frac{-b-\sqrt{b^2-4ac}}{2a} \tag{1}$$

This formula is inaccurate for negative b, especially when a and c are close to zero; reducing the rounding error involves most of Casio's major systems.

Casio begins the process of improving (1) by *focusing*, identifying which subexpressions contribute the most rounding error. In this case, error arises from the subtraction $-b-\sqrt{b^2-4ac}$: for small values of a and c, $\sqrt{b^2-4ac}$ approximately equals $\sqrt{b^2}$, which for negative b equals $-b$; therefore, for negative b and small values of a and c the two operands are approximately equal, leading to so-called "catastrophic cancellation". Casio detects this error by sampling inputs and analyzing the rounding error for each input. For each expression in the program, Casio computes the *exact* value of its children to 64 bits of accuracy using arbitrary-precision arithmetic. A *locally approximate* result is then computed by applying a floating-point operator to exact arguments. By comparing this locally approximate result to the exact result, Casio obtains a measure of how much each operator contributed to the expression's overall error. For the quadratic example, Casio notices that the previously described subtraction in the numerator produces the most local error for negative values of b This information is passed to Casio's recursive rewrite step, directing rewrites toward improving this particular expression.

Once focusing has identified problematic subexpressions, Casio applies local transformations to their operators. A recursive rewrite algorithm is responsible for finding possible transformations: if a rewrite rule does not immediately pattern match on the expression, but destructs the correct operator, the children of the expression are recursively rewritten in an attempt to make the rule match. This rewrite step produces several candidate programs for each misbehaving subexpression. For example, rewriting at the subtraction discussed above produces ten candidate programs; two other expressions with nonzero local error produce twenty-six more. Of the candidates produced from rewriting at the subtraction, one applies the identity $x-y=(x^2-y^2)/(x+y)$ to the problematic subtraction. This transforms (1) into

$$\frac{\frac{(-b)^2-\sqrt{b^2-4ac}^2}{(-b)+\sqrt{b^2-4ac}}}{2a} \tag{2}$$

This candidate does not have improved precision, but does introduce the possibility of cancelling the two b^2 terms.

To discover this cancellation, Casio cannot use focusing and recursive rewrite because there are too many intermediate steps: removing the unnecessary square root squared, removing the canceled negation of $(-b)^2$, and reassociating the subtraction $b^2-(b^2-4*a*c)$. Discovering such a long sequence of undirected steps by enumerating all sequences of rewrites is impractical; instead, Casio has a simplification pass specialized toward cancelling like terms. Simplification automatically performs all the steps necessary to transform the candidate (2) into

$$\frac{\frac{4ac}{(-b)+\sqrt{b^2-4ac}}}{2a} \tag{3}$$

Further simplification does not impact the error behavior of the function, so it is not performed. This new program has no imprecision for negative b, but it is inaccurate for positive b. Regime inference will now be used to combine both programs into one that is inaccurate for all b.

Notice that, as in many examples, no single expression is accurate for all input values. To further improve accuracy, Casio must produce a program specialized to certain ranges of input values. Casio's regime inference system computes the branch condition that selects between several program variants depending on the input data. Casio's regime inference subsystem infers the best branch condition by considering all generated programs that are best on at least one sample point, and then using the sample points to divide each input variable into several regions, with one program per region, although some programs may cover multiple non-adjacent regions. In the case of programs (1) and (3), regime inference correctly decides to use (1) for positive values of b and (3) for negative values. The final program produced by Casio is

$$\begin{cases} \dfrac{\frac{4ac}{-b+\sqrt{b^2-4ac}}}{2a} & \text{if } x \geq 0 \\ \dfrac{-b-\sqrt{b^2-4ac}}{2a} & \text{otherwise} \end{cases}$$

In summary, at each step of Casio's goal directed search, it focuses on expressions with high error, uses its recursive-rewriter to destruct the operators of those expressions,

and them simplifies the results. To pursue different programs for different inputs, Casio saves the best program variants it has seen at each input point and combines them using regime inference. By applying high-level strategies, Casio is able to explore deeply into the search space without running into exponential slowdown.

4. Improving Accuracy with Casio

Tools to improve program behavior are typically required to preserve the semantics of the input program. For Casio this constraint is not satisfiable: in general, given a program over real numbers, there is no completely equivalent program over floating numbers due to rounding error. Even finding floating point programs with minimal rounding error is intractable [11]. Thus, our first obligation is to develop a notion of *correctness* for accuracy improving transformations.

4.1. Equivalent input and output

Casio's search is guided by sampling many input points and evaluating candidate programs on these points. Since Casio's search is guided by sampled points, there is a danger that Casio might overfit, resulting in a final program that is very accurate on the sampled points but not on general floating point numbers. To prevent overfitting, Casio is required to return programs which, interpreted as real number formulas, are equivalent to the original. This restricts the set of programs Casio can produce, and largely eliminates overfitting, since the programs Casio can produce must be valid implementations of the same formula as the input.

Formally, consider two semantics for programs: a real-number semantics, where each constant and variable is a real number and each operator is a function from reals to reals; and a floating-point semantics, where each constant and variable is truncated to a floating-point number and where each function is replaced by its floating-point version. Casio's transformations must preserve the real-number semantics of programs, but may change the floating-point semantics. Some important transformations can preserve the real-number semantics of a program, except at isolated points. For example, the transformation $x + y \rightsquigarrow (x^2 - y^2)/(x - y)$ does not preserve the real-number semantics when $x = y$. This class of transformations is useful for improving the accuracy of programs, so instead of requiring that the input and output program are equivalent in the real semantics, we only require that they are equivalent where both are defined. We call such programs *real-equivalent*.

Since Casio must not change the real-number semantics of programs (except by changing where the program is defined), it is natural to structure Casio as applying a sequence of rewrite rules, each of which preserve real equivalence. These rules are specified as input and output patterns; for example, the transformation above is a rule, with x and y matching arbitrary subexpressions. Casio contains 122 rules, including those for the commutativity, associativity, distributivity, and identity of basic arithmetic operators; fraction arithmetic; laws of squares, square roots, exponents, and logarithms; and some basic facts of trigonometry.

Each of these rules is a basic fact of algebra; it incorporates no understanding of numerical methods. In preparing the rules, care was taken to ensure that rules do not change the real-number semantics of programs. This was usually easy, but required some care to avoid false "identities" such as $\sqrt{x^2} = x$ (this identity being false for negative x).

4.2. The main loop

The simplest approach to using this database of rewrites is a hill-climbing brute force search, where one applies rewrite rules at random, keeping the transformations where error behavior is improved. Unfortunately, this method is not effective: too much time is spent on rewrites that have no effect on program accuracy, and sequences of rewrites that would improve accuracy are often not found, since intermediary steps do not neccessarily improve error. Often, multiple rewrites without any effect are necessary to enable a rewrite that significantly improves accuracy. Sometimes, even after the correct rewrite is found, terms need to be canceled before accuracy is gained. Brute force search struggles with these cases. To improve accuracy, important subexpressions must be found and focused on; sequences of rewrites need to be considered as a unit; and cancellation needs to occur automatically.

Casio's core is a simple loop which threads together several rewrite techniques, addressing each of these problems. The loop tracks a set of candidate programs, each real-equivalent to the input. Only programs which have the best-known accuracy on at least one input point are tracked. In each iteration, Casio generates new candidates by choosing one alternative that has not been chosen before; finding expressions where a rewrite may lead to improvement; selecting the rewrite-rules which match the error-causing operator and applying a sequence of rewrites to enable the application of that rule; and simplifying the result, canceling like terms. Then the best candidate is chosen for the next iteration. Before entering the main loop, Casio analyzes the program for periodic subexpressions and attempts to optimize them separately. After the main loop completes, Casio aggregates all candidates generated throughout the course of the loop, and infers regimes of input values for which to execute different candidates.

```
def casio(e) :=
  pts, exs := pickPoints(e)
  e' := simplify(e)
  t := makeTable(e'')
  while(not t.done)
    e, t' := pick t
    es := map simplify (map rr (focus e))
    t := t' ∪ es
  return regimes (∪{correlate e | e ∈ t})
```

Figure 1: The Casio main loop. Casio initializes its sample points, exact outputs, and candidate program table t, and enters the main loop. At each step through the loop, a candidate e is chosen from t, and explored by focusing on expressions with local errors, rewriting those expressions, and simplifying the results. When all candidates have been explored, regime inference is used to combine the found candidates. Extra candidates are generated at the end by attempting to correlated error.

4.3. Sampling points

Casio uses sampled input points to guide its search. 500 input points are used for estimating the error of candidate programs; as discussed in Section 5.1, this is sufficient for accurately evaluating the accuracy of candidate programs. These input points are sampled uniformly from the set of floating point *values*. That is, each sampled point is a combination of a random mantissa, exponent, and sign bit.

This unusual distribution of samples serves two purposes. First, by sampling exponents uniformly, Casio generates both very small and very large input points. A sampling strategy that simply samples uniformly between the maximum and minimum finite floating point values would almost never sample values very close to zero, and would thus be unlikely to mitigate rounding error for small values. If Casio is run with this strawman sampling strategy, it is unable to solve any but the most trivial examples. Second, the uniform sampling over exponents allows accurately evaluating the correct output for each input point.

Casio must produce a program which approximates the ideal, real-number semantics of the original program. This requires knowing the output generated by the real-number semantics of the original program on the sampled input points. However, the real-number semantics is not executable, since exact real numbers cannot be efficiently represented on a computer. Instead, Casio uses arbitrary precision floating point as provided by GNU MPFR [13]. Arbitrary precision floating point requires selecting a precision in bits, which MPFR will simulate in software.

Casio must choose a sufficiently high level of precision, lest the ground truth against which it evaluates candidate programs is incorrect. However, accuracy does not improve smoothly with precision. Consider the program $((1 + x^k) - 1)/x^k$ at $x = \frac{1}{2}$. Until k bits of precision are available, the computed answer is 0, even though the correct result is 1. Once k bits are available, the correct answer is computed exactly. A similar pattern occurs with many real-world programs. To mitigate it, we increase working precision until further increases in precision do not change the computed answer for any input point, to 64 bits. We have found this method to select a sufficiently large working precision for arbitrary precision evaluation; the precision required can be as large as 560 bits (see Figure 6 in Section 5). We have found all candidate programs evaluate to the same values in arbitrary precision, suggesting that further increases in working precision are unnecessary.

Once sample points are chosen and exact and floating-point answers are computed, some measure of error is necessary to compare the two. Mathematically-natural measures include absolute and relative error. However, both of these measures are ill-suited to measuring error between floating-point values [26]. We follow STOKE [26] in defining the error between x and y by the number of floating-point values in the range $[x, y]$:

$$\textbf{def } \mathcal{E}(x, y) := \#\{z : \mathsf{FP}_{64} \mid x \leq z \leq y\}$$

This measure of error is uniform over the input space and avoids special handling for infinite and denormal values.

4.4. Focusing with local error

If Casio detects that the input program has significant floating-point error, Casio applies errors to reduce it. For large programs, many rewrites are possible; to reduce this search space, Casio must identify which expressions are the source of rounding error. The error in computing each subexpression is a rough measure for which expressions are the source of error; however, it this measure is biased toward larger expressions, because functions whose arguments are computed high error usually compute results with high error[§]. Casio eliminates this bias by focusing on expressions with high *local error*. The local error of an expression is the error in its output if it were called with exactly-computed inputs:

$$\begin{aligned}
&\textbf{def } \mathsf{localerror}(ctx, f(e_1, \ldots, e_k)) := \\
&\quad e'_i := \mathsf{eval}_\infty(ctx, e_i) \\
&\quad r_{64} := f_{64}(\mathsf{F}_{64}(e'_1), \ldots, \mathsf{F}_{64}(e'_k)) \\
&\quad r_\infty := \mathsf{F}_{64}(f_\infty(e'_1, \ldots, e'_k)) \\
&\quad \textbf{return } \mathcal{E}(r_{64}, r_\infty)
\end{aligned}$$

[§]Garbage in, garbage out

By exactly evaluating inputs, Casio avoids penalizing the output of operators for errors in computing their inputs. Figure **??** gives pseudo-code for computing local error. Casio computes the local error of each sub-expression at every sample point, and selects the three expressions with the highest local error for rewriting.

4.5. Recursive rewrite pattern matching

After focusing on an expression using local error, Casio must apply rewrite rules to that expression. A common problem is that an expression may require multiple rewrites to enable a rewrite that actually improves accuracy. For example, consider the expression

$$\left(\frac{1}{x-1} - \frac{2}{x}\right) + \frac{1}{x+1}.$$

Casio correctly identifies the $(+)$ operator to have the most local error (it adds terms with similar values and opposite signs). To improve the accuracy of this program, all of the fractions must be placed under a common denominator, and then the numerator must be simplified. Casio has rules for fraction addition and subtraction; however, doing a single fraction addition or subtraction does not significantly change the accuracy of the program, since accuracy loss is caused by a cancellation that occurs when *all* values are added together. In order to improve the accuracy of this program, Casio must use the fraction addition/subtraction rules twice: once on the parenthesized subtraction, then again for the remaining addition. The first rewrite must occur not at the focused-upon expression, but on a child expression; this rewrite is necessary for enabling a later rewrite at the focused-upon expression.

Casio automatically finds sequences of rewrites like this example, where subexpressions are rewritten to allow a rewrite of the parent expression. In order to rewrite an expression to match a rule, Casio rewrites each subexpression, recursively, to match its associated pattern in the rule (if the subexpression already matches, the recursion terminates). This recursive algorithm produces dozens of rewrite sequences for each focused location; they vary from one to eight sequences in length.

4.6. Simplification

In order to improve the accuracy of a formula, it is often necessary to first apply a rewrite rule at an expression of high error, and then cancel like terms and function inverses. For example, Casio will rewrite $\sqrt{x+1} - \sqrt{x}$, into $(\sqrt{x+1}^2 - \sqrt{x}^2)/(\sqrt{x+1} + \sqrt{x})$. On it's own, this transformation does not reduce error. However, the numerator of this fraction simplifies to just 1, and $1/(\sqrt{x+1} + \sqrt{x})$ is a much more accurate program than the original $\sqrt{x+1} - \sqrt{x}$. To find such beneficial transformations, Casio automatically simplifies the expressions generated by the recursive matcher. This simplification pass automatically removes function inverses (as in $\sqrt{x}^2$), cancels like terms (as in $x - x$), and combines terms (as in $x + x$). It is implemented as a tree walk over programs, which first removes function inverses, then reduces each expression into a polynomial canonical form, and cancels like terms. Such transformations are difficult to find heuristically, as they involve long chains of reassociation and commutation rewrites. A separate simplification pass allows these transformations to be discovered easily.

4.7. Regime inference

Often two real-equivalent programs have incomparable accuracy—one performs better on some inputs, but worse on others. The quadratic formula, discussed in Section 3, is striking example of this phenomenon. To accurately evaluate the quadratic formula, both expressions

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad \frac{2c}{-b - \sqrt{b^2 - 4ac}}$$

are necessary, the first for $b < 0$ and the second for $b \geq 0$. A similar pattern occurs in many real-world programs, with different programs being most accurate for different input regions. The process that automatically detects that two programs are best on different inputs, and automatically infer the regimes is called *regime inference*.

Casio's regime inference must balances the cost of adding a branch against the potential benefit of doing so: branches are computationally expensive. Casio must also avoid overfitting the sampled points; very small regimes likely represent some peculiar behavior of the sampled points. To avoid overfitting and generating slow programs, Casio observes two constraints on the regimes it infers: it infers no more than 5 per input variable; and, it infers the existence of a regime only if it improves average accuracy by at least one bit.

The simplest way to infer regimes would be to try to place a regime boundary between every set of 5 pairs of consecutive sample points. Unfortunately, with 500 sample points, this brute-force approach would be incredibly expensive (there are nine trillion such sets of five pairs). Instead, Casio uses dynamic programming to find the optimal set of up to five consecutive sample points. The dynamic program computes the optimal set of at most k regimes in $(-\infty, x_i)$, where x_i is a sampled point; this problem has the optimal substructure property required for dynamic programming. Figure **??** contains the algorithm in detail. This algorithm is quadratic in the number of sample points, and linear in the number of regimes; we have found this algorithm to be sufficiently fast for Casio. Once Casio has determined that branch should be placed between two sampled points, it uses a binary search on the chosen variable to find the exact value location of the regime boundary.

$$
\begin{aligned}
&\textbf{def}\ \mathsf{regimes}(\{p_1, \ldots, p_c\}) \coloneqq \\
&\quad N \coloneqq \text{Sampled points} \\
&\quad r_{t,0} \coloneqq p_{i_1},\ \text{where } p_{i_1} \text{ best on } [0, x_t] \\
&\quad \varepsilon_{t,0} \coloneqq \mathsf{avgerror}(r_{t,0}, [0, t]) \\
&\quad \textbf{for } k \in \{1, \ldots, 5\} \\
&\qquad r_{t,k} \coloneqq \min_s \{(r_{s,k}, p_k) | \\
&\qquad\quad p_k \text{ best on } [x_s, x_t], \\
&\qquad\quad r_{s,k} = [r_{s',k-1}, p_{k-1}], \\
&\qquad\quad \left(\sum_{i=s}^{t} \mathsf{error}(p_k, i)\right) > N + \left(\sum_{i=s}^{t} \mathsf{error}(p_{k-1}, i)\right)\} \\
&\quad r_{t,\star}\{(r_{t,5}, p_k) | p_k \text{ best on } [x_t, \infty]\} \\
&\quad \textbf{return}\ r_{t,\star},\ \text{for } r_{t,\star} \text{ best of all } t
\end{aligned}
$$

Figure 2: A dynamic programming algorithm for regime inference. For $k \in \{0, \ldots, 5\}$ and $t \in \{1, \ldots, N\}$, the best way to split $(-\infty, x_t]$ into up to k regimes is computed by considering, for all s, the best way of splitting $(-\infty, x_s]$ into regimes along with the best program on $[x_s, x_t]$. The best way of splitting $(-\infty, \infty)$ into regimes is the best way, over all t, of splitting $(-\infty, x_t)$ into regimes plus the best possible program on (x_t, ∞). After the regimes are found, each boundary is improved by a binary-search between the two sampled points that it lies between.

4.8. Candidate programs table

Between iterations of the core loop, Casio stores only those candidate programs which are more accurate (of all generated candidates) on at least one sample point. This allows programs that are accurate on only some input values to be explored without having to explore exponentially-many candidates. When new candidate programs are generated, this set must be updated.

To update this set efficiently, Casio stores two maps: a map from points to a set of alternatives that do best at that point, and a map from alternatives to the points they do best at. A candidate is added to this data structure only if it is better at some point than the alternatives currently stored for that point. The alternatives that are no longer best at this point are removed from the data structure if there are no more points that they are best at. This ensures that a minimal number of alternatives are stored. This update rule is also biased toward keeping alternatives that were already contained in the table: this minimizes the amount of work that Casio does.

4.9. Correlating Error

In rewriting expressions where local error is high, Casio is using the assumption that the accuracy of a program can only be improved by improving the accuracy of each expression in it. This assumption is often true in practice; however, it is sometimes best not to reduce the error in a subexpression, but instead to introduce error into a different expression, so that the errors cancel. A simple example of this approach, presented by William Kahan [19], is the formula $(e^x - 1)/x$. This formula is inaccurate for x near 0, because of rounding error when computing e^x. However, if the denominator is changed from x into $\log e^x$, the same rounding occurs in the denominator; when the two values are divided, the answer is 1, which is correct.

Casio use this trick. After the main rewrite loop is complete, Casio finds all remaining expressions with error, and tries rewrite any siblings of those expressions or their ancestors. This produces many candidates, but since it is done only once and after the main rewrite loop is complete, the associated cost is small. We've found this technique to be useful in only a small number of cases, but in those cases it is the necessary for reducing error.

5. Evaluation

We evaluate Casio on benchmarks drawn from Hamming's *Numerical Methods for Scientists and Engineers* [15], a standard numerical analysis textbook. Our evaluation includes all twenty-eight worked examples and problems from Chapter 3, which discusses manually rearranging formulas to improve accuracy, the same task that Casio automates.

All experiments were carried out on an Intel Core i5-3317U with 4GB RAM running Fedora Linux 19, PLT Racket 5.3.6, and GCC 4.9.1. When compiling programs to C for performance measurements, we used GCC flags `-march=native`, `-mtune=native`, `-mfpmath=both`, `-O3`, and `-flto`.

Overall, our results demonstrate that Casio can improve accuracy for 21 out of our 29 benchmark, often without significantly slowing execution (average 45% overhead). In all cases, the programs produced by Casio run orders of magnitude faster than software floating point (a slowdown of two or more orders of magnitude). In some cases, the accuracy of the programs output by Casio even exceeds that solutions provided by experts in numerical methods textbooks. We also separately evaluate our error estimation technique and our regime inference algorithm, and describe several case studies using Casio.

Accuracy. Our results for are shown in Figure 3. Casio improves the program's accuracy for 21 out of 28 programs; for 20 programs, this is an improvement of at least one bit. Of the seven benchmarks that Casio does not improve, NMSE suggests solutions which are not real-equivalent to the input because they use Taylor expansions to recover precision near 0. While Casio cannot currently improve these benchmarks, adding support for unsound rewrites such as Taylor expansions is an interesting direction for future work.

Overhead. We also the performance of both the

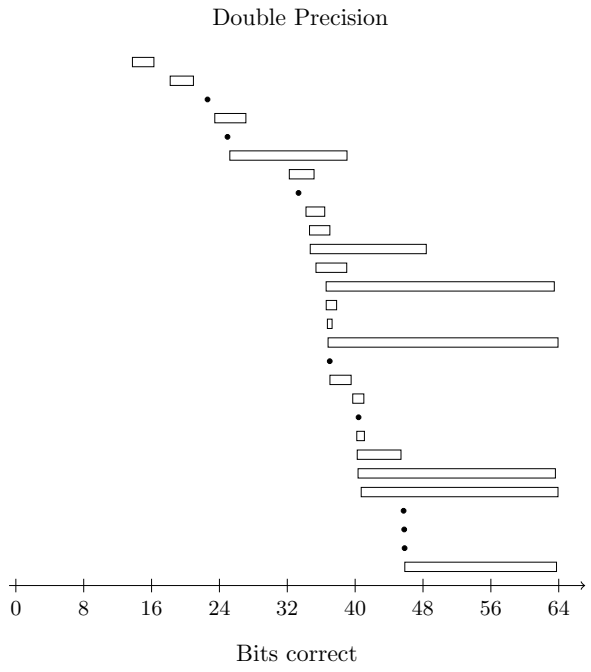

Figure 3: Accuracy Results. Each row of the figure represents a single benchmark. For each benchmark, the accuracy of the input program is drawn as the left edge of a rectangle, and the accuracy of Casio's output is drawn as the right edge. The benchmarks are sorted from top to bottom in increasing order of the accuracy of Casio's output. The horizontal axis measures accuracy in terms of the number of bits of output that agree with the exact result, averaged across one million input points. Target programs were evaluated using double precision 64-bit floats. For those inputs which Casio is not able to improve by at least half a bit, a dot is drawn at the input accuracy level.

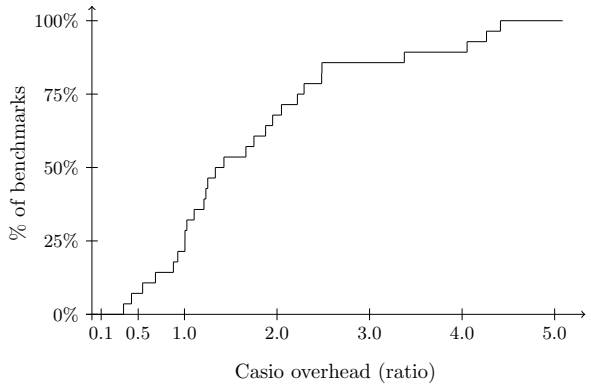

Figure 4: Cumulative distribution of overhead induced by Casio. The horizontal axis shows the ratio between the performance of the output and input programs. Programs run using double precision floats and compute on one million randomly chosen points.

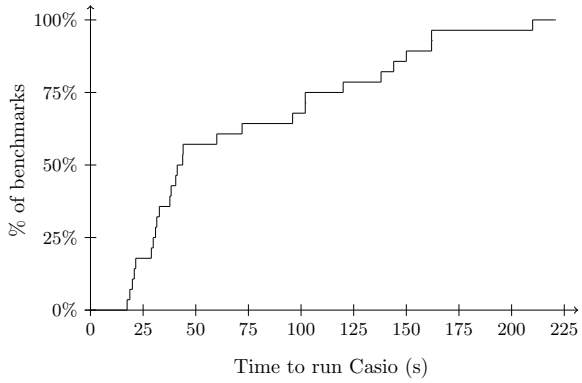

Figure 5: Cumulative distribution of running time of the Casio tool on each benchmark. The horizontal axis shows time in seconds. Most benchmarks complete in less than two minutes.

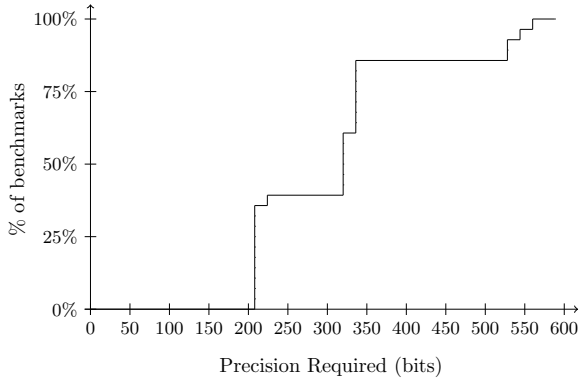

Figure 6: Cumulative distribution of precision required to evaluate each benchmark to 64 bits of accuracy. The horizontal axis shows bits of precision. Results were obtained by iteratively increasing the precision until the high-order 64 bits converged.

original program and the program produced by Casio. On average, the program produced by Casio is 45% slower, much smaller than a slowdown of two orders of magnitude for using software arbitrary precision.

Runtime. Casio was run on each benchmark in a standard configuration[¶]. The main loop was capped at 2 iterations. Figure 5 is a cumulative distribution function for Casio's runtime on these benchmarks. For 15 of 29 tests, Casio runs in under one minute.

Error Estimation. The accuracy and speed was measured, both for the original program in each benchmark, and for Casio's output. Examples were compiled to C, which was then compiled by the GNU Compiler Collection with flags optimizing for performance without loss of accuracy[‖]. Both the input and output program were run

[¶]Including regime inference and the normal number of sample points.

[‖]Version 4.9.1 of `gcc` was used, with flags `-march=native`, `-mtune=native`, `-mfpmath=both`, `-O3`, and `-flto`. Experiments were run on an Intel Core i5-3317U processor.

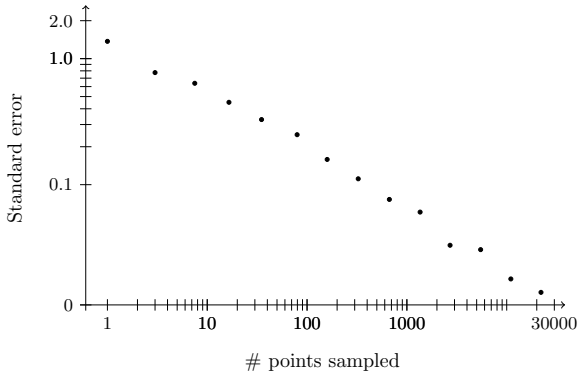

Figure 7: Standard error vs. sample size. Each point represents one hundred runs of Casio on the quadratic formula benchmark with varying sample size used to estimate error of program variants. The horizontal axis shows the sample size. The vertical axis shows standard error of the mean of the error estimates produced by the one hundred runs. By default, Casio runs the search with a sample size of 1024, which has a standard error of approximately 0.1.

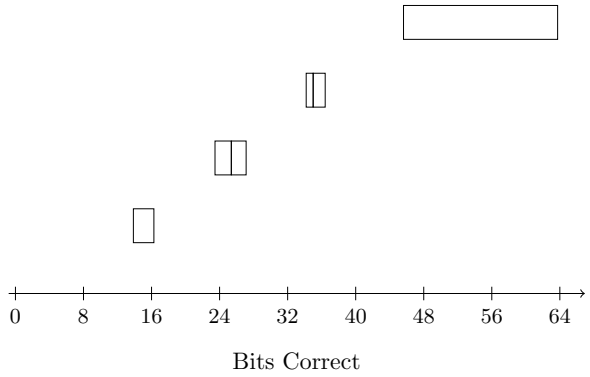

Figure 8: Evaluation of regimes inference. For each of the four programs for which regimes improve accuracy significantly, a rectangle is drawn. The left edge of the rectangle is the unimproved program. The middle line, if any, is Casio's output without regimes. The right line is Casio's output with regimes. If no middle line is visible, Casio without regimes was not able to improve the program.

with double precision intermediates. Each program was run on the same set of 1,000,000 points drawn randomly from the set of single-precision floating point numbers. Results were compared with a ground truth computed via the MPFR arbitrary-precision library [13]. Figure 6 shows the cumulative distribution of the number of bits necessary to exactly evaluate the benchmarks. The error of an output computed in floating-point, was taken to be the base-2 logarithm of the number of 64-bit floating point values**, between the exact and inexact answer: the bits in error for the approximate output, compared to the exact answer. This error was averaged over all points for which the exact answer was a finite floating point value, resulting in a simple number which can be used for comparing implementations. Sampling one million points is a very accurate measure of the average error, as demonstrated in Section 5.1.

5.1. Error Evaluation

The tests above evaluated Casio by comparing the average error of its input and output. Average error was computed by sampling one million points and computing the average error across the sample. To demonstrate that this accurately estimates average error, we computed average error with the number of points N varying between 1 and 10,000. For each N, the experiment was repeated 100 times with different sampled points. The standard error of the mean of these 100 repetitions, for each value of N, measures the accuracy of our computation of average error. Figure 7 is a representative result; it was generated from the the quadratic formula example described in Section 3, but every test case generated a qualitatively similar result. In each case, sampling a thousand points gave a result with standard error of approximately 0.1 bits. This demonstrates that the numbers given above are accurate.

To ensure that our evaluation in arbitrary precision is accurate (that is, to ensure that we use a sufficiently large number of bits to evaluate the results accurately), we computed the exact answer for both Casio's input and output in arbitrary precision. In every case, the answers were identical, showing that arbitrary precision evaluation is accurate.

5.2. Regime Inference

Two experiments evaluate Casio's regime inference. The first is an end-to-end test, running Casio on its standard benchmarks but with regime inference turned off. Comparing the results of Casio with and without regime inference measures the effectiveness of regime inference. The same sample points were used to make the results directly comparable. Regime inference adds branches to four of the 29 benchmarks; Figure 8 shows Casio's accuracy and speed on these four benchmarks, with and without regime inference. The results show that for these test cases, regime inference makes a significant improvements to the program's accuracy. However, the added branches also make the resulting programs slower. Regime inference infers branches of the form $a_< < x_i < a_>$, with x_i a program variable. Since points are selected randomly, these branches are impossible for the processor to predict; in practical use, where points are not randomly chosen, the overhead may be lower. Users may opt to disable

**Inputs are single precision values, while outputs are double precision. Single precision for inputs avoids handicapping implementations that use single precision for intermediate values. Double precision for outputs allows measuring error more accurately than single precision would.

regime inference if this performance cost is unacceptable.

A second experiment measured regime inference in isolation. An input point p is chosen and regimes is asked to combine the simple programs $sqrt(x)$, $sin(x)$, $cos(x)$, $log(x)$, $sqr(x)$, and $abs(x)+10$, to most accurately compute ($\mathsf{if}\, x < p \,\mathsf{then}\, a \,\mathsf{else}\, b$), for a and b chosen from the same candidate programs. Regime inference must infer a branch of the form $x < p'$, with the correct candidate programs on each side of this branch, to succeed. The difference between the computed value p' and the ground truth p measures regime inference's ability to accurately infer branch conditions. This experiment was repeated 40 times. In each case, regime inference inferred a branch of the correct form; the difference between p and p' varied between 0 and 1 bits of error. This suggests that regime inference infers branch conditions very accurately.

5.3. Wider applicability

To test Casio's applicability beyond its standard benchmark suite, we gathered mathematical formulas from a variety of sources and tested both their numerical accuracy and Casio's ability to improve it. These sources included several papers from Volume 89 of Physical Review; standard definitions of mathematical functions, such as hyperbolic trigonometric functions, arithmetic on complex numbers; and approximations to special functions like erf and ζ. Of the 68 formulas gathered, we found that 20 exhibited significant floating point inaccuracies. For these 20 examples, Casio was able to improve 6 with no modifications and without enlarging its database of rules. This is yet another confirmation that rounding error can arise in the daily practice of scientists and engineers. However, it is important to note that for these examples we have not determined if inaccuracies arise for valid inputs; and, for formulas Casio was unable to improve, whether a real-equivalent formula with lower error exists.

5.4. Case Study: Computing Probabilities

Many numerical programs are not library functions or textbooks examples, but are instead simulations or data analysis scripts used by scientists, engineers, and mathematicians in the course of their work. The expressions encountered in these programs are more complex and less structured than examples like (1). Casio has demonstrated success in improving precision in these more-difficult cases.

A colleague of ours had difficulties with the numerical precision of a MCMC update rule in a clustering algorithm. He needed to compute the expression

$$\begin{aligned} \mathsf{ans} &= \frac{p(e_-, e_+|s)}{p(e_-, e_+|t)} \\ p(e_-, e_+|x) &= (\operatorname{sig} x)^{e_+}(1 - \operatorname{sig} x)^{e_-} \\ \operatorname{sig} x &= 1/(1 - e^{-x}) \end{aligned}$$

and found that the simple encoding of this formula as a program lead to spurious results and violated invariants in later code. Our estimates suggest that this simple encoding produces seventeen bits of error, averaged over floating point values. To avoid these problems, our colleague manually manipulated the expression into a form that seemed not to cause similar problems; our estimates suggest that this improved variant had ten bits of average error.

Upon hearing of his troubles, we fed the original, naive implementation to Casio. Casio produced a program with four bits average error:

$$\exp\left(e_+ \ln \frac{1 + e^{-t}}{1 + e^{-s}} + e_- \ln \frac{1 - \frac{1}{1+e^{-s}}}{1 - \frac{1}{1+e^{-t}}}\right)$$

Casio obviated the need for manual algebraic manipulation, and produced superior results with no manual steps. The low-error program that Casio produced was so complex that human analysts would be unlikely to reproduce the result. Furthermore, our attempts to simplify the program into a more understandable form have only been able to do so by greatly increasing its error. By properly targetting its search, Casio is able to consider a greater number of viable alternatives than human analysts are able to, and is capable of producing better results.

6. Related Work

Casio builds upon decades of work in numerical analysis and recent advances in the analysis of floating point computations.

Numerical analysis Numerical analysis analyzes mathematical computations performed by computers. It includes a vast literature on how to evaluate mathematical functions. The technique of rearranging formulas appears in surveys [14, 20], and in common textbooks [15]. Casio uses the techniques invented in this literature, but rearranges formulas automatically, avoiding the need for an expert.

It is difficult to know the working precision necessary to accurately evaluate a function, a problem known as the table maker's dilemma [21]. Recent work on this problem allowed the creation of MPFR, an arbitrary-precision floating point library which correctly rounds exponential, trigonometric, and other basic functions [13]. However, composing these functions can still lead to error. Casio uses MPFR internally to determine the exact output

of a computation, to compare candidate programs for accuracy.

Verification Floating point computation is defined in the IEEE 754 standard [1, 2]. This standard specifies the behavior for floating point numbers and their primitive operations. However, programming languages often do not require adherence to this standard. For example, C [17] allows intermediate results to be computed in extended precision, and only truncated when stored into memory [23]. This makes verification and analysis difficult for floating point computations.

Recent work in the program verification community has allowed formal verification of numerical programs. The triangle area computation discussed by Kahan [20], for example, was proven accurate by S. Boldo in the interactive proof assistant Coq [8]. Larger computations—programs for computing discriminants [7] or solving simple partial differential equations [9]—have also been proven correct. Automatic proofs have been also explored: Rosa [10] provides a language for specifying accuracy requirements and uses an SMT solver to automatically prove them. Ariadne [6] applies SMT to detect inputs which cause floating point overflow. Casio focuses on the problem of floating point accuracy, not overflow; if verification is required, tools like Rosa could be used to prove that the output of Casio is sufficiently accurate.

Optimization of floating point programs The Stoke super-optimizer supports optimizing floating point programs while guaranteeing that the resulting accuracy is acceptable [26]. Stoke uses a stochastic search over assembly programs to find faster implementations of a given function, without introducing unacceptable error relative to the original program. A similar tool is Precimonious [25], which attempts to decrease the precision of intermediate results to improve performance. Both tools assume that the input program they are given accurately represents the computation the program intended to perform. If the input program is inaccurate, Stoke and Precimonious can at best guarantee that the accuracy of the optimized result is not much worse. On the other hand, Casio attempts to improve the input program, so that it better represents the real number formula that it implements. We suspect that Stoke and Casio may complement each other, with Casio improving the accuracy of a program, so that its error (relative to a real number computation) is acceptable, and Stoke then optimizing the program within the given error bounds.

7. Future Work

We intend to improve Casio's search for accuracy-improving transformations and also work toward automating additional numerical analysis techniques, e.g. supporting complex number and matrix computations, reducing error accumulation in loops, and analyzing numerical stability. We believe that Casio's insight of using real-equivalence to enable sampling techniques that avoid overfitting can help in all these areas.

Eventually, we hope to integrate Casio into a compiler pass, allowing it to run transparently as part of a compilation pipeline. This would make Casio significantly easier to use, as it would obviate the need to isolate and input floating point computations. Since compilers must optimize for speed as well as numerical accuracy, we would like to allow the user to specify a tradeoff between speed and accuracy.

We would also like to explore allowing Casio to output programs that are not real-equivalent to the original program. This would enable using Taylor expansions and interpolation to improve the accuracy of input programs, but will require error analysis techniques less dependent on sampling. While real-equivalent guarantees that Casio preserves the programmer's intent, some programs cannot be improved at all, or can be improved more efficiently, by making unsound transformations. This may also lead to further explorations of precision vs. performance tradeoffs for floating point.

References

[1] IEEE standard for binary floating-point arithmetic. *IEEE Std. 754-1985*, 1985.

[2] IEEE standard for binary floating-point arithmetic. *IEEE Std. 754-2008*, 2008.

[3] Micah Altman, Jeff Gill, and Michael P. McDonald. *Numerical Issues in Statistical Computing for the Social Scientist*. Springer-Verlag, 2003.

[4] Micah Altman and Michael McDonald. The robustness of statistical abstractions. *Political Methodology*, 1999.

[5] Micah Altman and Michael P. McDonald. Replication with attention to numerical accuracy. *Political Analysis*, 11(3):302–307, 2003.

[6] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. POPL '13.

[7] Sylvie Boldo. Kahan's algorithm for a correct discriminant computation at last formally proven. *IEEE Transactions on Computers*, 58(2):220–225, February 2009.

[8] Sylvie Boldo. How to compute the area of a triangle: A formal revisit. *Computer Arithmetic, IEEE Symposium on*, 0:91–98, 2013.

[9] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave Equation Numerical Resolution: a Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, April 2013.

[10] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 235–248, New York, NY, USA, 2014. ACM.

[11] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of fixed-point programs. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 22:1–22:10, Piscataway, NJ, USA, 2013. IEEE Press.

[12] European Commission. Directorate-General for Economic and Financial Affairs. *The introduction of the euro and the rounding of currency amounts*. Euro papers. European Commission, Directorate General II Economic and Financial Affairs, 1998.

[13] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.

[14] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[15] Richard Hamming. *Numerical Methods for Scientists and Engineers*. Dover Publications, 2 edition, 1987.

[16] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002.

[17] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[18] W. Kahan. *A Survey of Error Analysis*. Defense Technical Information Center, 1971.

[19] W. Kahan and Joseph D. Darcy. How Java's floating-point hurts everyone everywhere. Technical report, June 1998.

[20] William Kahan. Miscalculating area and angles of a needle-like triangle. March 2000.

[21] Vincent Lefèvre and Jean-Michel Muller. The table maker's dilemma: our search for worst cases. October 2003.

[22] B. D. McCullough and H. D. Vinod. The numerical reliability of econometric software. *Journal of Economic Literature*, 37(2):633–665, 1999.

[23] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, May 2008.

[24] Kevin Quinn. Ever had problems rounding off figures? this stock exchange has. *The Wall Street Journal*, page 37, November 8, 1983.

[25] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. SC '13.

[26] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating point programs using tunable precision. PLDI 14.

[27] Debora Weber-Wulff. Rounding error changes parliament makeup, 1992.