ARFA: An Agile Regime-Based Floating-Point Optimization Approach for Rounding Errors

Jinchen Xu*
atao728208@126.com
Information Engineering University
Zhengzhou, China

Mengqi Cui*
saki1340@163.com
Information Engineering University
Zhengzhou, China

Fei Li
feili2022@hotmail.com
Information Engineering University
Zhengzhou, China

Zuoyan Zhang
zhangzuoyan523@163.com
Hunan University
Changsha, China

Hongru Yang
hpcyhr@163.com
Information Engineering University
Zhengzhou, China

Bei Zhou†
beibei_0812@126.com
Information Engineering University
Zhengzhou, China

Jie Zhao†
jiezhao@hnu.edu.cn
Hunan University
Changsha, China

Abstract

We introduce a floating-point (FP) error optimization approach called ARFA that partitions the domain $\mathcal{D}$ of an FP expression f_e into *regimes* and rewrites f_e in each regime where f_e shows larger errors. First, ARFA seeks a rewrite substitution f_o with lower errors across $\mathcal{D}$, whose error distribution is plotted for effective regime inference. Next, ARFA generates an incomplete set of ordered rewrite candidates within each regime of interest, so that searching for the best rewrite substitutions is performed efficiently. Finally, ARFA selects the best rewrite substitution by inspecting the errors of top ranked rewrite candidates, with enhancing precision also considered. Experiments on 56 FPbench examples and four real-life programs show that ARFA not only reduces the maximum and average errors of f_e by 4.73 and 2.08 bits on average (and up to 33 and 16 bits), but also exhibits lower errors, sometimes to a significant degree, than Herbie and NumOpt.

CCS Concepts

• **Mathematics of computing → Numerical analysis**; • **Software and its engineering → Error handling and recovery**.

Keywords

dynamic analysis, numerical analysis, floating-point errors, rewrite, e-graph, FPbench

*Both authors contributed equally to this research.
†Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '24, September 16–20, 2024, Vienna, Austria

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

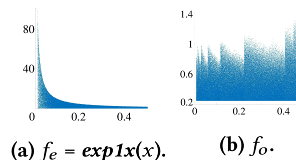
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680378

ACM Reference Format:
Jinchen Xu, Mengqi Cui, Fei Li, Zuoyan Zhang, Hongru Yang, Bei Zhou, and Jie Zhao. 2024. ARFA: An Agile Regime-Based Floating-Point Optimization Approach for Rounding Errors. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680378

1 Introduction

As real numbers are approximated using FP representations with finite precision in computers, numerical rounding errors are inevitable. Errors of individual FP expressions can be significant [23, 37] and may also result in an overall unacceptable result for a program computed using a sequence of such FP expressions. A widely cited example [2, 34, 40] is the American Patriot Missile failure [31] caused by the accumulation of errors of individual FP expressions. Improving the accuracy of individual FP expressions is thus of vital importance.

This issue was partially solved by the state-of-the-art (SOTA) tool–Herbie [23]. Given an f_e with its domain $\mathcal{D}$, Herbie generates a collection of rewrite expressions that are more accurate than f_e on its sampled inputs, and divides $\mathcal{D}$ into sub-domains or *regimes*. Each regime is rewritten using the found rewrite expressions so that the overall accuracy can be improved. By targeting numerically unstable programs where extremely large errors are observed around some inputs, Herbie and its followers [35, 37] infer regimes by sampling a small number of dynamic inputs, but these approaches sometimes loss their effectiveness when targeting specific examples, e.g., Fig. 1a.



(a) f_e = ***exp1x***(x). (b) f_o.

Figure 1: The error plots of *exp1x*(x) before and after preprocessing.

Regina [26] considers numerically stable programs and infers regimes using a search heuristic: derived from an initial partition strategy that splits $\mathcal{D}$ into equally-sized regimes, it merges neighboring regimes with the same expression, but the number of the resulting regimes is often much larger, which degrades the execution performance of its produced expressions. As such, *a generally applicable and effective regime inference algorithm is still missing*.

Once a regime partition is produced, rewriting tools are allowed to replace f_e in each regime of interest. To obtain a better rewrite substitution, many approaches leverage equality saturation [23, 25, 33, 36] to build an equivalence graph or e-graph that contains all rewrite candidates, and the one with the lowest errors is selected. This strategy is also subject to some problems. First, the size of an e-graph can be large due to the massive number of rewrite rules [21], making the search for the optimal expressions non-trivial. Many existing approaches select the optimal expressions via inaccurate cost models and thus obtain sub-optimal results. Second, enhancing precision is rarely considered as a rewriting rule. Hence, *a better heuristic for searching the optimal rewrite substitution is also absent*.

To address these issues, in this paper, we study an **A**gile **R**egime-based **F**loating-point optimization **A**pproach, named Arfa. Given an f_e, Arfa first searches a rewrite substitution f_o that has lower errors across its domain $\mathcal{D}$. Still consider Fig. 1a. A better expression f_o (whose expression is omitted here for the sake of clarity) that has lower errors across $\mathcal{D}$ (depicted in Fig. 1b) is found by Arfa. The purpose of this preprocessing step is to build a better start-up expression for follow-up regime inference and expression rewriting.

Next, Arfa builds a scatter plot of errors for f_o by densely sampling its inputs, better reflecting the error distribution than prior tools [23, 35, 37]. The boundary line of a scatter plot is then used to implement a regime inference algorithm wider applicable than existing tools [7, 23, 26], which can not only split the domain of Fig. 1a (numerically unstable) into $[0.01, 0.33]$ and $(0.33, 0.5]$ but also extract two regimes, $[0.06, 0.13]$ and $[0.22, 0.50]$, for Fig. 1b (numerically stable).

Finally, Arfa generates an incomplete set of ordered rewrite candidates within each regime where f_o should be substituted. Several top ranked rewrite candidates are empirically executed, among which the one with the smallest error is selected to replace f_o. While avoiding the need to build an inaccurate cost model [7, 23, 26], this simple search heuristic also considers increasing precision as a complementary rewriting strategy.

Arfa is made modular and the idea presented can easily be integrated into other tools. Arfa also uses some optimization strategies to combine regimes. In summary, Arfa is a general approach for reducing FP errors by contributing:

- a general regime inference algorithm, which reduces the complexity of Herbie's algorithm [23] (§ 5.5) while applicable to numerically stable and unstable FP expressions;
- a rewriting heuristic that not only allows itself to perform better than prior work [23, 35] but also avoids the need to resort to an inaccurate cost model (§ 6.5);
- a modular architecture (§ 3) that is compatible to other error analysis tools [30], and can be integrated with different rewriting techniques [21, 23] (§ 7.3);
- and experimental results (§ 7) on 60 benchmarks showing that Arfa reduces the maximum and average errors of the original expressions by 4.73 and 2.08 bits on average, and up to 33 and 16 bits, respectively.

2 Background

Rounding Errors and Accuracy. An FP number can be expressed as the combination of a signed mantissa and an exponent, but there always exists some real numbers that cannot be expressed exactly, making some computed results have to be approximated and resulting in the difference between itself and the exact value. This difference is called a *rounding error* or *error* for simplicity. Errors caused by FP representations can be reduced by increasing precision, but specifying an appropriate precision is non-trivial, since these approaches are either restricted by the possible execution performance loss [10] or the precision-specific limitation [34].

Even if a higher precision is available, the errors of an FP expression are also influenced by its arithmetic, whose reordering may offset the improved accuracy brought by the enhanced precision. *Accuracy* expresses the proximity between a computed result and the analytical value it tries to approximate. Consider a scenario that computes the sum $x + y$ of two 32-bit FP numbers x and y. The inexactness of this FP representation can be optimized by upgrading its precision to 64 bits, but the error of the sum may increase when x is very large and y close to 0 or vice versa. We care about the impact of reordering FP arithmetic because such rearrangements nowadays take place frequently in many domains [21, 38], and rewriting an FP expression this way is adopted by the recent regime-based optimization approaches [23, 26].

Regime-based Rewriting. We explain regime-based rewriting approaches using single-variate expressions for the sake of clarity. A rewriting system studies the error distribution of an expression e_x of variable x across its domain $\mathcal{D}$ and tries to substitute e_x using its rewrite candidate e_y. As x is stored discretely in $\mathcal{D}$, it is possible to know the errors of e_x and e_y under each value of x, from which one can observe the distribution of the errors across $\mathcal{D}$, just like Fig. 1 shows. One can determine which one between e_x and e_y is with the lower errors and perform substitutions accordingly.

Regime-based rewriting tools [23, 26, 37] go one step further by performing rewriting within regimes where e_x exhibits larger errors. Traditional rewriting systems are a special case of the regime-based approach when $\mathcal{D}$ is treated as the only found regime, but the regime-based tools can always use the one with lower errors in each regime and thus obtain lower errors than both e_x and e_y. The issue is that it is difficult to know which input ranges cover the values that trigger large errors. Even if one can localize the input values of large errors, there may still be many partition strategies, and the situation becomes more complicated in multi-variate cases.

3 Overview of Arfa

We present Arfa to address the challenges faced by existing regime-based rewriting approaches. Fig. 2 shows the architecture of Arfa. The (violet) thick arrows connect the five stages: error analysis, preprocessing, regime inference, rewrite generation, and code optimization. The internal structure of a stage is related by solid (black)

arrows, and a dashed (black) arrow is used to take in or produce inputs/outputs.

Figure 2: Architecture of ARFA.

ARFA first performs error analysis (§ 4) across $\mathcal{D}$ for f_e, which is extracted from a program's basic blocks, i.e., without control constructs like loops and **if** conditionals. f_e can be a (piecewise) function over multiple variables computed by any FP arithmetic operators, possibly including transcendental functions as its subexpressions. For an n-variate FP expression, we assume that $\mathcal{D}$ is a hyper-rectangular interval denoted as $[\![[lb_1, ub_1], \cdots, [lb_n, ub_n]]\!]$, where lb_i and ub_i ($1 \leq i \leq n$) are lower and upper bounds. This simplification is not always a limitation, since a non-rectangular domain is often the intersection of such lower/upper bounds and complex constraints like $x_i + x_j \leq 10$. ARFA still works when these complex constraints are relaxed. We found that this simplification seems also required by prior work [7, 23] when we experimented with them, although it was not explicitly declared.

ARFA's error analysis uses the *ulp* error as its error metric. An *ulp* error is computed as

$$error(x) = \frac{|p(x) - f(x)|}{\mathtt{ulp}(f(x))}, \tag{1}$$

where $p(x)$ is the approximated result and $f(x)$ is the correctly rounded, double-precision value. $\mathtt{ulp}(f(x))$ is the *ulp* function of $f(x)$, used to measure the closeness between two accurately representable double-precision numbers a and b nearest to $f(x)$, where $a \leq f(x) < b$ if $f(x) \geq 0$ or $a < f(x) \leq b$ otherwise. The *ulp* function is defined as

$$\mathtt{ulp}(f(x)) = \begin{cases} 2^{k-52} & \text{if } 2^k \leq f(x) < 2^{k+1} \wedge -1022 \leq k < 1023 \\ 2^{-1074} & \text{if } 0 \leq f(x) < 2^{-1022} \end{cases}, \tag{2}$$

where k is an integer. This definition is the same as the one used in recent work [16, 37], which in turn is derived from [12]. ARFA defines its error metric this way because accurate errors do not exactly reflect how significant they are over the value of $f(x)$ while relative errors are not defined at $f(x) = 0$. In contrast, *ulp* errors do not have such limitations.

Next, the preprocessing (§ 4) rewrites f_e using f_o across $\mathcal{D}$. We consider f_o is better than f_e if the maximal *ulp* error, which is referred to as *mae* and computed as

$$mae = \max_{\forall x \in \mathcal{D}} (error(x)) = \max_{\forall x \in \mathcal{D}} \left(\frac{|p(x) - f(x)|}{\mathtt{ulp}(f(x))} \right), \tag{3}$$

of f_o is smaller than that of f_e. *mae* can be replaced by other metrics, e.g., L_∞-norm [3], which can be computed by modifying the denominator of *mae*, or average absolute error (*aae*), which computes the average mean of all absolute errors measured a multiple of *ulp*. In principle, *mae* and *aae* are nothing but only used to compute the maximal and average ones of ARFA's error metric.

ARFA's regime inference (§ 5) partitions $\mathcal{D}$ using three steps. While configurable using a default setting, this engine can also take as input a partition grain, offering the flexibility to control the generation of regimes. Each regime that needs rewriting passes through the search stage (§ 6), which first converts fractions into a normalized form, next simplifies the input and transforms it into an expanded form, possibly with a follow-up reordering step, and finally selects the best equivalent expressions for individual regimes.

As producing regimes introduces additional **if** conditionals in the generated code, the found rewrite substitutions are delivered to the code optimizer, which aggregates **if** conditionals by making use of the hardware branch predication techniques. Due to the page limitation, we will not dig into the details of this stage in this paper. Finally, the optimized expression $\widehat{f_e}$ is generated.

4 Error Analysis and Preprocessing

Since the preprocessing step also relies on error analysis, we introduce these two steps together in this section.

4.1 Dynamic Error Analysis using MPFR

We use dynamic error analysis [30, 40] in ARFA because its results are often considered as more accurate than those of static analysis [5, 9]. The difficulty is that dynamic analysis only inspects the errors of f_e under a limited number of values sampled from $\mathcal{D}$. The value(s) that trigger(s) the *mae* may be missed if the number of dynamic samples is small, i.e., the risk faced by Herbie [23] and AutoRNP [37]. To mitigate this, we set the total number of dynamic samples as $N = \prod_{i=1}^{n} d_i$, where d_i is the number of samples along the i-th variable dimension. We instantiate N as 500,000, a value also used by an error optimizer S^{3}FP [4] if $n = 1$; otherwise, each d_i is instantiated using 1024 when $n = 2$ or 256 when $n \geq 3$. The bounds of N are inferred by well compromising between the degree to which a selected number of samples can avoid the risks faced by prior work and error analysis overhead. One can redefine them according to his/her experimental environments. Sampling is executed in parallel when dealing with a large n.

The next step is to determine how these samples are chosen. FP numbers do not uniformly distribute along the real number line: the distribution is dense if the range bounded by $\mathcal{D}$ is around zero, or loose if the range is close to infinity. A sophisticated selection strategy should consider the closeness of $\mathcal{D}$ to zero. However, we just let the N samples uniformly distribute across $\mathcal{D}$ for simplification purposes, since N is large and $\mathcal{D}$ is often small enough.

The errors are measured using MPFR [10], which can work with arbitrary FP precision. Some FP expressions require MPFR to use up to 8000 bits for correctly rounding results, which is heavyweight and the reason why prior work [23] use a small number of samples. Fortunately, many benchmarks do not require such a high precision; even for those that do have such a requirement, they do not need such a high precision for every input. Furthermore, our sampling approach remains embarrassing parallel even at such a high precision, making the overhead of our sampling process controllable. Note that it is free to substitute MPFR with any other error analysis libraries [14, 30, 40].

4.2 Preprocessing

ARFA can now partition $\mathcal{D}$. However, a start-up FP expression is important for an FP rewriting system. For instance, if ARFA tries to optimize Fig. 1a, its result may still have larger errors than Fig. 1b, although the expressions of these examples are equivalent across $\mathcal{D}$. This is because the errors of Fig. 1b are much smaller than Fig. 1b, but the optimization of ARFA is performed on the basis of Fig. 1a.

By considering this, we first try to rewrite f_e without regime partitioning so as to build a better start-up expression. The search for f_o is performed by first querying Herbie (version 1.6) and Daisy (version 0.1) [7]. Next, ARFA selects the better one between the found candidates, which is again compared with f_e. The final winner is f_o. For example, the expression of Fig. 1b is delivered to ARFA instead of the original one.

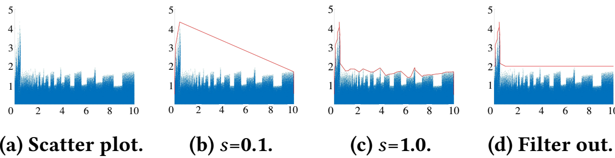
Note that ARFA can work independently of these tools. Its rewriting heuristics can be used in preprocessing at the expense of slightly increased overhead, since we empirically execute each rewrite candidate to analyze their errors. These rewriting heuristics can also be easily adapted to construct a complete e-graph in advance, which can be queried without additional overhead if a well-defined cost model is established.

5 Effective Regime Inference

With error analysis, each input sample $(x_1, \cdots, x_n)$ and its error constitute an n-dimensional tuple $(x_1, \cdots, x_n, error)$, which we use to perform regime inference for f_o.

5.1 Plotting Error Distribution

We first assume f_o is a single-variate expression. Given a set of 2D tuples, one can use them to outline an error scatter plot of errors for f_o. ARFA employs MATLAB-*v2020a* [20] for this purpose, which obtains error scatter plots like Fig. 1 or Fig. 3a, whose expression is *Bspline0*(x).



(a) Scatter plot. **(b) s=0.1.** **(c) s=1.0.** **(d) Filter out.**

Figure 3: The scatter plot of *Bspline0*(x) and its boundary lines in (b) and (c) under different shrink factors. $\mathcal{D} = [\![0, 10]\!]$ and $N = 500,000$.

5.2 Sketching Boundary Lines

ARFA evaluates every $\left(x^{(i)}, error\right)$ to determine whether its error should be optimized. $x^{(i)}$ is a sample of the x variable. A group of such 2D tuples whose errors need to be reduced constitute a regime. However, ARFA has to evaluate all of the N samples, which may be time-consuming: a large N is good to build a scatter plot, but ARFA has to evaluate the same number of 2D tuples to infer regimes. To resolve this, we let ARFA sketch the boundary line of an obtained scatter plot, which is doable using the *boundary* function of MATLAB.

Specifically, this function takes as input a scatter plot like Fig. 3a and sketches a boundary line by managing a shrink factor s $(0 \le s \le 1)$. A smaller s yields a rough (red) boundary line in Fig. 3b, and a larger one produces an exact one in Fig. 3c. In practice, we use the greatest possible value of s, i.e., 1, which generates Fig. 3c for the scatter plot of *Bspline0*(x).

A boundary line is connected by b points that lie in the boundary of the scatter plot. One can obtain these b points for further use. $b \ll N$ always holds because these points constitute a strict subset of the N points used to build the scatter plot. As such, sketching the boundary line of a scatter plot can significantly reduce the number of samples from N to b. We collect the values of b for *Bspline0*(x) under different shrink factors in Table 1. The bottom row reports the reduced rate by sketching boundary lines. The reduced rate gets decreased with the increase of s, but ARFA only needs to use 1364 samples instead of 500,000 to generate regimes for *Bspline0*(x) even when $s = 1$, which consumes 7.79 seconds, making efficient regime inference possible.

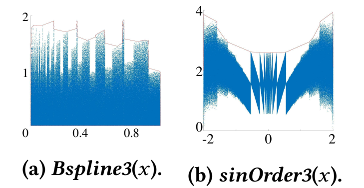
Table 1: The values of b for *Bspline0*(x) under different s.

variable	values					
s	0.1	0.3	0.5	0.6	0.8	1.0
b	182	445	708	839	1102	1364
reduced by	99.96%	99.91%	99.86%	99.83%	99.78%	99.73%

A large s, however, makes a boundary line fluctuate significantly (Fig. 3c). The boundary line can be partitioned in many ways, and it is not straightforward to decide which one is the best. In addition, some partition strategies [26] may split $\mathcal{D}$ into too many regimes, which complicates the execution of ARFA and its generated expression. We thus define an upper bound, u so that each $(x^{(i)}, error)$ $(0 \le i < b)$ on the boundary line with $error \le u$ can be filtered out by defining

$$filter(x) = \begin{cases} u & \text{if} \quad error \le u \\ error & \text{if} \quad error > u \end{cases} \tag{4}$$

According to the GNU C library reference manual [19], we set u to 2 *ulp*s by default for the x86_64 architecture we target and it can be dynamically altered as will be explained later. We apply Expr. (4) and generate a new boundary line, as shown in Fig. 3d. Each 2D tuple can be denoted as $\left(x^{(i)}, filter\left(x^{(i)}\right)\right)$. Compared to Fig. 3c, Expr. (4) can conceal the fluctuations below u and thus generates fewer regimes.



(a) *Bspline3*(x). **(b) *sinOrder3*(x).**

Figure 4: u dose not cut the boundary line. $\mathcal{D}$ is $[\![0, 1]\!]$ for (a) and $[\![0, 3]\!]$ for (b). $N = 500,000$.

It might happen that u does not cut the boundary line. For example, Fig. 4a, which means the boundary line is always below the horizontal line of $u = 2$ *ulp*s. In such cases, we introduce a lower bound l and first let $l = 0$ *ulp*s. We repeatedly update u using $u = \frac{u+l}{2}$ until (1) the boundary line intersects with or becomes higher than the

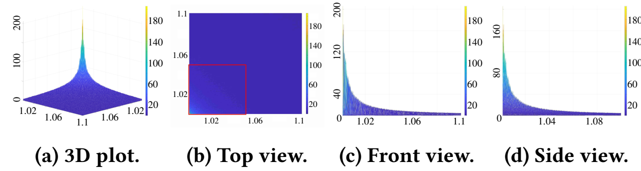
horizontal line of u, or (2) $u = 0.5$ *ulp*s. For the first condition, we apply our regime inference algorithm if the horizontal line of u cuts the boundary line; otherwise, we let $l = 2$ *ulp*s and repeatedly update u by letting $u = \frac{u+l}{2}$ to make the horizontal line of u go upwards until it cuts the boundary. If the second condition is reached, we will not optimize the expression because all errors are already below $u = 0.5$ *ulp*s.

On the contrary, the boundary line of Fig. 4b is also always above the horizontal line of $u = 2ulps$. When such undesired cases take place, we allow ARFA to dynamically change u by repeatedly updating its value using $u = \frac{u+mae}{2}$ until the boundary line can be cut by the upper bound.

5.3 Handling Multi-variate Expressions

When given an n-dimensional FP expression, one can obtain an $(n+1)$-dimensional scatter plot. By following the approach used for single-variate expressions, one can compute a hyperface of the error distribution and a hyperplane for the error upper bound u. This hyperface is cut by the hyperplane, and the resulting objects of the $(n+1)$-dimensional scatter plot are the sets of errors that need rewriting. One can project each of these $(n+1)$-dimensional objects onto the space of n-dimensional variables, with the cast projections forming the regimes ARFA would like to obtain.

We take Fig. 5a, the 3D error scatter plot of a dual-variate function *sec4-example*(x, y), as an example. The projection onto the space of 2D variables yields the top view shown in Fig. 5b. In principle, only the left bottom corner composed of all light blue points should be considered as a regime requiring rewriting, but it is too complicated to exactly express it.



(a) 3D plot. (b) Top view. (c) Front view. (d) Side view.

Figure 5: Error distribution of *sec4-example*(x, y) and its three-view drawings. $\mathcal{D} = [\![[1.001, 1.1], [1.001, 1.1]]\!]$.

We do not have to exactly compute such a range but instead seek a more regular, possibly over-approximated shape, e.g., a convex hull, that covers this range. This allows us to compute multiple 2D error distributions along one of its variables each time, resulting in n 2D error scatter plots. This process can be considered as projecting multiple $(n+1)$-dimensional points to a 2D point

$$\left(x_1^{(k_1)}, \cdots, x_i^{(k_i)}, \cdots, x_n^{(k_n)}, error\right) \rightarrow \left(x_i^{(k_i)}, error\right) \quad \forall 1 \le k_j \le N_j \wedge j \ne i \tag{5}$$

where $x_i^{(k_i)}$ $(1 \le i \le n)$ denotes the k_i-th sample of the i-th variable. The resulting 2D points constitute the i-th variant of the these n 2D scatter plots. For each of these 2D scatter plots, one can follow the approach to single-variate expressions to obtain the 1D ranges of each x_i. The final regimes should be the combination of these 1D ranges.

Still consider the *sec4-example*(x, y) function. We obtain two 2D error distribution plots, Fig. 5c and Fig. 5d, for Fig. 5a. Following the way in single-variate cases, one can obtain two 1D ranges, $x \in [1.001, 1.04906]$ and $y \in [1.001, 1.04906]$, for this dual-variate expression. The final sub-domain of our interest is the intersection of these two independent 1D ranges, which can be written as $[\![[1.001, 1.04906], [1.001, 1.04906]]\!]$ and is represented by the red rectangular shown in Fig. 5b.

5.4 Partitioning $\mathcal{D}$

Now we introduce the regime inference algorithm. For single-variate functions, the key insight to partition is to determine the bounds $r_j = [lb_j, ub_j]$ such that $filter\left(x^{(i)}\right) > u$ holds for each $x^{(i)} \in [lb_j, ub_j]$. We assume the total number of such ranges is M. Meanwhile, each point $\left(x^{(i)}, filter\left(x^{(i)}\right)\right)$ along this boundary line can use f_o if $x^{(i)}$ falls outside these M ranges, which are collected in one range. The total number of the ranges is thus $M+1$.

With $Set = \left\{\left(x_i^{(k)}, filter\left(x_i\right)\right) : 0 \le k < b_i \wedge 0 \le i < n\right\}$ and a partition grain g taken as inputs, Algo. 1 invokes the handling of a single-variate case for n times to deal with a multi-variate expression, i.e., the outer **foreach**. The first inner **foreach** determines the bounds of r_j by scanning the b points along a boundary line. lb_j is set to $x^{(i+1)}$ each time an increase from u of $filter\left(x^{(i)}\right)$ is captured, and ub_j is set to $x^{(i-1)}$ each time a decrease to u of $filter\left(x^{(i)}\right)$ is observed. Three corner cases are handled as follows: (1) the starting point of the scanning process is used to instantiate lb_0 if the first change of $filter\left(x^{(i)}\right)$ is a decrease; (2) the ending point is used to set ub_{M-1} when the last change of $filter\left(x^{(i)}\right)$ is an increase; (3) u is changed when it does not intersect with the boundary. The second inner **foreach** combines tight ranges into $R+1$ looser ones, with each treated as a regime. g is determined through many times of empirical execution, which also makes Algo. 1 flexible to be used in diverse scenarios.

5.5 Intellectual Advances of Algo. 1 over SOTA

We summarize the intellectual advances of Algo. 1 over Herbie [23] since it represents the state of the art. First, Algo. 1 reduces the time complexity of regime inference. We discuss time complexity of single-variate expressions for the sake of simplicity. The time complexity is $O(b)$ for Algo. 1 since it evaluates b points when determining the regime bounds; however, it is $O(h^2)$ for Herbie that uses dynamic programming to evaluate its samples, whose number is denoted using h. Herbie (version 1.6) sets h to 8000, making h^2 much larger than b.

Second, Algo. 1 can also handle numerically stable functions. Herbie uses an **if** conditional in its dynamic programming to capture a 1-bit error difference between two samples and decides a regime should be added when this predicate is true. However, errors of two samples of a numerically stable function often violate this predicate. In contrast, Algo. 1 can capture the slight fluctuation of a numerically stable program.

Algorithm 1: Regime inference

```
   Input: Set and g
   foreach i ∈ [1, n] do
1      project onto to i-th variable dimenion;
2      M_i ← 0;
       foreach k ∈ [0, b_i) do
           if filter(x_i^(k)) = u then
3              continue;
4          lb_{M_i} ← x_i^(k);
           while filter(x_i^(k)) > u and k < b_i do
5              k ← k + 1;
6          ub_{M_i} ← x_i^(k−1); M_i ← M_i + 1;
       if M_i = 0 then
7          update u and goto line 1;
8      R_i ← 0, reset ←true;
       foreach j ∈ [0, M_i) do
           if reset = true then
9              sub-domain_{R_i} ← r_j;
10             reset ←false;
           if sizeof(sub-domain_{R_i})< g × D_i then
11             ub_{R_i} ← ub_{j+1}; j ← j + 1;
           else
12             R_i ← R_i + 1; reset ←true;
13 intersecting sub-domains from n dimensions;
   Output: (∏_{i=1}^n R_i + 1) sub-domains
```

6 Customized Rewrite Generation

$\prod_{i=1}^{n} R_i$ out of the Algo. 1 output regimes need rewriting. ARFA decides to empirically execute the generated rewrite candidates to return exact results. This requires ARFA to produce fewer rewrite candidates. To achieve this, we introduce several preparation steps before rewrite generation and search.

6.1 Normalization

First, f_o may also include fractions as its sub-expressions that complicate rewrite generation and search. We thus first convert f_o involving fractions into a normalized form. We find the lowest common denominator of all terms in f_o, converting f_o into the form of $\frac{num_o}{den_o}$. Without loss of generality, we assume that f_o is of the form $\frac{1}{x+y} + \frac{1}{x-y} + 1$ and it can be rewritten as $\frac{1}{x+y} + \frac{1}{x-y} + 1 \rightsquigarrow \frac{(x-y)+(x+y)+(x+y)(x-y)}{(x+y)(x-y)}$.

Next, normalization is applied to num_o and den_o respectively until all arithmetic operations between fractions are removed. In case num_o is still a fraction, den_o is substituted by the product of num_o's denominator and den_o; num_o is replaced by its numerator. If den_o is a fraction, its numerator is used to substitute it and the product of its denominator and num_o is used to substitute num_o. This process is recursively performed until neither of num_o and den_o involves fractions.

6.2 Simplification

We use *quasi-polynomials* to refer to the expression forms acceptable by f_o's numerator/denominator after normalization. A quasi-polynomial only involves addition, subtraction, multiplication, and transcendental functions of its variables. We introduce a simplification step to reduce the number of FP operations in quasi-polynomials. Still consider the example used in § 6.1. Analytically, it can be simplified to $2x + x^2 - y^2$. We first multiply each term with another one in every other multiplied sub-expression $(x-y)+(x+y)+(x+y)(x-y) \rightsquigarrow (x-y)+(x+y)+x(x-y)+y(x-y) \rightsquigarrow (x-y)+(x+y)+x^2-xy+xy-y^2$, making use of the distributivity identity, and next combine like terms $(x-y)+(x+y)+x^2-xy+xy-y^2 \rightsquigarrow 2x+x^2-y^2$, enabling term rearrangement and cancellation. This simplification is performed recursively until a simplest form is obtained.

We also consider the removal of function inverses, e.g., $(\sqrt[3]{x^3}) \rightsquigarrow x$, and, when given an expression $x^2 + \sin(x)x^2$, ARFA transforms it through $x^2 + \sin(x)x^2 \rightsquigarrow (1 + \sin(x))\,x^2$, which is treated as a quasi-polynomial of x when transcendental functions are viewed as coefficients. They are combined with constants when not multiplied with a positive power of x, since they are viewed the coefficients of x^0. Such combinations were not considered before [7, 23] but generate more equivalent expressions. Note that Unlike Herbie [23], ARFA performs a simplification rewrite search and does not need to locate the most significant operator.

6.3 Reordering

A quasi-polynomial $1+3x+3x^2+x^3$ tends to be more accurate within a regime where x is close to infinity, while $1 + (3x + (3x^2 + x^3))$ is more likely to exhibit lower errors when given a regime around $x = 0$. This is because the powers of x increase/decrease much faster than x when $x \geq 1.0/x < 1.0$. Indeed, $1 + (3x + (3x^2 + x^3))$ can be defined using the associativity identity, but it is also achievable through reordering. The addition operators are with varying errors, and a good reordering of these terms can lead to lower errors.

To benefit from such a reordering, we categorize a regime using the FP number 1.0. A regime is considered close to 0 when the absolute values of each x falling in this regime is less than 1.0, or it can be treated as close to infinity. A regime is still considered as close to 0 when the FP number 1.0 is covered by this regime, since a regime is usually a small range, whose upper bound would not be too far from 1.0.

For multi-variate cases, we treat other variables as constants when reordering one variable once a time. Consider $x^2+2xy+y^2$ as an example. It is a good start when the current variable x is close to 0 and y is considered as a constant. However, $y^2+2xy+x^2$ is preferred when y is very small variable and x is treated as a constant. Such analysis still obtains a consistent reordered expression, $x^2+2xy+y^2$, in some cases, e.g., x is close to 0 and y is near infinity, but they may conflict with each when x and y are both close to 0. In such cases, we randomly select one from the reordered expressions.

6.4 Rewrite Generation and Search

By taking as a simplified expression f'_o and a flag *is_fraction* as inputs, Algo. 2 describes how rewrite candidates are generated, which we also leverage e-graph [22] to represent.

The outermost **foreach** iterates n variables and updates the e-graph by considering f'_o as a single-variate form

$$f'_o(x_v) = c_0(x_v) + c_1(x_v)x_v + c_2(x_v)x_v^2 + \cdots + c_t(x_v)x_v^t \quad (6)$$

where x_v is one of the n variables and its coefficient $c_k(x_v)$ can contain transcendental functions and/or other variables. The constant term $c_0(x_v)$ is removed from f'_o (line 3) and the number of remaining terms is denoted as t (line 4). The degree k of a term $c_k(x_v)x_v^k$ can also be a positive fraction. We extract the common factor x_v

Algorithm 2: Build an incomplete e-graph

```
  Input: f'_o, e-graph, is_fraction
1 e-graph ← { f_e, f_o, f'_o };
  foreach v ∈ [1, n] do
2   f'_o ← write f'_o like Expr. (6) w.r.t the v-th variable;
3   f'_o ← remove constant terms from f'_o;
4   t ← number of terms in f'_o;
    foreach i ∈ [1, t − 1] do
      foreach j ∈ [i + 1, t] do
5       d ← j − i + 1;
6       c ← the common coefficient of x_v after extracting x_v from the first
          d terms from f'_o;
7       set ← rewrite transcendental functions;
8       g ← Algo. 2 (c, set, false);
        foreach e in g do
9         e_tmp ← substitute c using e in f'_o;
        if is_fraction then
10        den_o ← the denominator of f'_o;
11        set ← rewrite transcendental functions;
12        g' ← Algo. 2 (den_o, set, false);
          foreach f in g' do
13          e-graph ← e-graph ∪ e_tmp / f;
        else
14        e-graph ← e-graph ∪ e_tmp;
  Output: e-graph
```

Algorithm 3: Select the optimum

```
  Input: f'_o, ∏_{i=1}^n R_i sub-domains
1 r ← 1;
  while r ≤ ∏_{i=1}^n R_i do
2   f'_o ← handle f_o as described in § 6.1 to § 6.3;
    if f'_o is a numerator of f_o then
3     e-graph ← Algo. 2 (f'_o, ∅, true);
    else
      if f'_o is the denominator of f_o then
4       do nothing;
5     e-graph ← Algo. 2 (f'_o, ∅, false);
6   e-graph ← sort nodes in an ascending order w.r.t. the number FP operators
      and preserve the top 30 nodes;
7   s ← 1, f_r = f'_o;
    while s ≤ num and s ≤ sizeof(e-graph) do
8     e_tmp ← select one node from e-graph;
      if e_tmp = f'_o then
9       continue;
      if e_tmp exhibits lower mae than f_r then
10      f_r ← e_tmp;
11    s ← s + 1;
    if f_r = f'_o then
12    f_r ← rewrite f_r by enhancing precision;
  Output: {f_r : 1 ≤ r ≤ ∏_{i=1}^n R_i}
```

from each sub-expression composed of $d = j - i + 1$ terms (line 5) like $c_i(x_v) + c_{i+1}(x_v)x_v + \cdots + c_j(x_v)x_v^j$ where i iterates from the leftmost term $c_1(x_v)x_v$ to the rightmost but one, i.e., $c_{t-1}(x_v)x_v^{t-1}$, and j iterates from $c_i(x_v)x_v^i$ to the rightmost term $c_t(x_v)x_v^t$.

Algo. 2 instantiates c (line 6) as $c_1(x) + c_2(x)x$ for Expr. (6) in the dual-nested loop's first iteration, to which Algo. 2 is applied recursively (line 8). Algo. 2 also tries to rewrite a sub-expression involving transcendental functions via pattern matching. The e-graph, g, of c is then visited, and each equivalent expression of c is used to substitute c in f'_o. The resulting expression e_{tmp} is a rewrite candidate of f'_o and is added to *e-graph* (line 14) when f'_o is not a fraction. Otherwise, the denominator, den_o, of f'_o is obtained (line 10), to which Algo. 2 is recursively applied again (line 12), and the fraction $\frac{e_{tmp}}{f}$ is taken as a node into *e-graph* (line 13).

Unlike prior work, Algo. 2 does not try to add all equivalent of f_e into the e-graph but only considers limited equivalent expressions. Hence, it only builds an incomplete e-graph. By convention, one can now navigate this e-graph using a cost model to find the best, but we decide to empirically execute the e-graph nodes to obtain more accurate error distributions. The problem of empirical execution is the heavy execution overhead, and we thus cannot visit all nodes of the e-graph. Although ARFA does not saturate its e-graph, this representation is still used because a future saturation can be introduced easily without much effort, which can then implement an exhaustive search when necessary.

As such, we first sort the nodes of the e-graph in an ascending order with respect to the number of FP operators. We consider a rewrite expression with fewer FP operators is tend to exhibit smaller error. In case the e-graph is of a large size, we only preserve the top 30 ones and empirically execute *num* out of them, which is set to 10 by default. An exhaustive search of the e-graph is triggered when $num = -1$. Algo. 3 summarizes the search algorithm, with f'_o and $\prod_{i=1}^n R_i$ regimes taken as inputs.

f'_o requires no further processing when it is the denominator of f_o, since its rewrite candidates have been combined with the nominator. Line 6 performs the sort of the e-graph's nodes. The number of FP operators is the sum of FMA instructions, addition, subtraction, and multiplication operators in the expression. To demonstrate this sorting method is reasonable, we apply Algo. 2 to three examples, i.e., b25, b27, and b34 of Table 2, which obtains 134, 61, and 169 rewrite substitutions, respectively, for these examples. For each of them, we empirically measure the errors of all rewrite substitutions and also sort them in an ascending order according to their errors. When compare the top 30 of this order with the top 30 obtained by line 6 of Algo. 3, there are 11, 27, and 6 rewrite candidates that are included in both top 30 lists.

f_r is the rewrite candidate of the r-th regime, randomly selected from the e-graph (line 8), and its errors are compared with those of f'_o. Line 9 is introduced to select a node other than f'_o. f_r will be substituted if the randomly selected rewrite exhibits a lower *mae* (line 10). Finally, Algo. 3 tries to rewrite the expression by increasing the precision if no better equivalent expression is found (line 12). Note that we only increase precision for a regime, whose impacts on execution performance are much slighter than tuning precision across $\mathcal{D}$. Finally, the output of ARFA can be expressed as

$$\widehat{f_e} = \begin{cases} f_r & \text{if } 1 \le r \le \prod_{i=1}^n R_i \\ f_o & \text{else} \end{cases} \quad (7)$$

6.5 Superiority of Algo. 3 over SOTA

We still compare with Herbie [23] to clarify the superiority of Algo. 3. First, Algo. 3 is more likely to generate results whose *mae* and *aae* are smaller than or competitive to the best Herbie can find. We take b25 of Table 2 as an example. The *mae* of the rewrite substitution found by Herbie is shown using orange circle in the left plot of Fig. 6, and the 30 blue ones are the *mae* of rewrite candidates generated by ARFA, all of which have smaller *mae* than that of Herbie. This phenomenon also happens when comparing *aae*. The right plot of Fig. 6 depicts the results of b34 of Table 2. The *mae* of

the 30 (blue) rewrite candidates generated by Arfa is often smaller than or comparable to that of Herbie, and the *aae* is very close. Such results can also be observed by other examples.

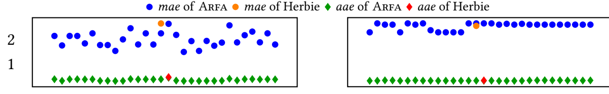


Figure 6: Comparing rewrite candidates generated by Arfa and Herbie. Left: b25; right: b34. The y axis represents errors.

Second, Algo. 3 avoids the need to define a cost model to select the optimal rewrite candidate. This allows Arfa to select some of the candidates and empirically execute them, whose precision can be increased without changing the order of arithmetic operators. Note that Herbie rewrites each sub-expression when it derives the final result. The number of its rewrite candidates is often larger than ours (Fig. 6), where we only show the errors of its final result.

7 Experiments

Arfa is implemented in about 7k LoC of C/C++, 0.2k of Python, 0.2k of MATLAB, and 0.3k of miscellaneous, which has been made publicly accessible at https://gitee.com/mathlib/expr-auto. We conduct experiments on an Intel Xeon E5-2630 v4 CPU. The generated C programs are compiled using GCC v9.4.0 with "*-O3 -lm*" flags. FPBench [6] includes 131 benchmarks, but some of them share the same expression. We consider only one of such repeated benchmarks. We also exclude an expression whose *mae* is smaller than 0.5 *ulps*, since they do not need rewriting. Four benchmarks containing control constructs are handled by unrolling the **if** conditional statements, just like what Regina [26] did. In total, we have 56 FPbench benchmarks. To cover as diverse scenarios as possible, we also collect four benchmarks extracted from real-life numerical programs, one from POV-Ray [24] that uses ray-tracing to render a 3D image and the other three from a polynomial support vector classifier [18]. The 60 benchmarks are summarized in Table 2. The fourth column indicates whether a benchmark is stable or not, which is listed here to show that Arfa is applicable to both kinds of FP expressions. The sixth column collects the values used to instantiate u on the fly, demonstrating the agility of Arfa. The fifth and rightmost columns are reported for reproducibility purpose.

We compare the results with Herbie-v1.6 [23] and NumOpt [35] that rewrites programs via prioritized stochastic algebraic transformations. We also tried to experiment using AutoRNP [37], but its publicly accessible code is tightly bound to the GSL library used in its publication. We thus did not consider this tool. We use the default $\mathcal{D}$ provided by FPBench when available. If some of the considered tools cannot work on a given $\mathcal{D}$, we either set it as ⟦[0.01, 100]⟧, a widely-used domain used by FPBench, or a reasonable range that can let all considered tools work.

7.1 Comparison of Errors

We first report the *mae* in Fig. 7. As the *mae* values vary among the 60 benchmarks, we did not report its real values but normalize each data with respect to the one of Arfa. That is to say, the normalized

Table 2: Summary of the benchmarks. Braces enclose ordered values of u and the numerical stability, which is determined after error scatter plots. Source of f_o is reported in the rightmost: 26 from Herbie, 29 from f_e and 5 from Daisy.

	no.	benchmark	stable	$\mathcal{D}$	u (in $ulps$)	f_o
single-variate	b1	Bspline3	✓	⟦[0, 1]⟧	1.0	Herbie
	b2	exp1x	×	⟦[0.01, 0.5]⟧	1.0	Herbie
	b3	exp1x_log	✓	⟦[0.01, 0.5]⟧	1.0	original
	b4	intro_example	✓	⟦[0, 999]⟧	1.0	original
	b5	logexp	×	⟦[−8, 8]⟧	1.0	Herbie
	b6	NMSEexample31	✓	⟦[0.01, 100]⟧	2.0	Herbie
	b7	NMSEexample310	×	⟦[−1, 1]⟧	2.0	Herbie
	b8	NMSEexample34	×	⟦[0.01, 3]⟧	2.0	Herbie
	b9	NMSEexample35	✓	⟦[0.01, 100]⟧	1.0	Herbie
	b10	NMSEexample36	✓	⟦[0.01, 100]⟧	2.0	Herbie
	b11	NMSEexample37	×	⟦[0.01, 100]⟧	0.5	Herbie
	b12	NMSEexample38	×	⟦[0.01, 100]⟧	6.05	Daisy
	b13	NMSEexample39	×	⟦[0.01, 1]⟧	6.02	original
	b14	NMSEproblem331	✓	⟦[0.01, 100]⟧	1.0	Herbie
	b15	NMSEproblem333	✓	⟦[0.01, 0.99]⟧	1.5	Herbie
	b16	NMSEproblem334	×	⟦[0.01, 100]⟧	2.0	Daisy
	b17	NMSEproblem336	✓	⟦[0.01, 100]⟧	1.0	Herbie
	b18	NMSEproblem337	×	⟦[0.01, 100]⟧	1.0	original
	b19	NMSEproblem341	×	⟦[0.01, 100]⟧	2.0	Herbie
	b20	NMSEproblem343	×	⟦[−1, 1]⟧	1.0	Herbie
	b21	NMSEproblem344	×	⟦[0.01, 100]⟧	0.875	Herbie
	b22	NMSEproblem345	×	⟦[0.01, 100]⟧	6.02	original
	b23	NMSEsection311	×	⟦[0.01, 100]⟧	1.0	Herbie
	b24	predatorPrey	✓	⟦[0.1, 0.3]⟧	2.0	original
	b25	sine	✓	⟦[$-\frac{\pi}{2}$, $\frac{\pi}{2}$]⟧	1.0	Herbie
	b26	sineorder3	✓	⟦[−2, 2]⟧	3.13	Herbie
	b27	sqroot	✓	⟦[0, 1]⟧	0.625	Herbie
	b28	sqrt_add	✓	⟦[1, 1000]⟧	2.0	original
	b29	test05_nonlin1,r4	×	⟦[1.00001, 2]⟧	1.25	Herbie
	b30	test05_nonlin1_test2	✓	⟦[1.00001, 2]⟧	1.25	Herbie
	b31	verhulst	✓	⟦[0.1, 0.3]⟧	1.375	original
multi-variate	b32	ComplexSinCos	{✓, ×}	⟦[0, 1], [0, 1]⟧	{286.11, 6.02}	original
	b33	ComplexSquareRoot	{✓, ✓}	⟦[0, 1], [0, 1]⟧	{1.0, 1.0}	original
	b34	doppler1	{✓, ✓, ✓}	⟦[−100, 100], [20, 20000], [−30, 50]⟧	{2.59,2.0, 2.59}	original
	b35	doppler2	{✓, ✓, ✓}	⟦[−125, 125], [15, 25000], [−40, 60]⟧	{2.0,2.0, 3.11}	original
	b36	doppler3	{✓, ✓, ✓}	⟦[−30, 120], [320, 20300], [−50, 30]⟧	{2.51,2.0, 2.51}	original
	b37	hypot32	{✓, ✓}	⟦[1, 100], [1, 100]⟧	{2.0, 2.0}	Herbie
	b38	i4	{✓, ✓}	⟦[0.1, 10], [−5, 5]⟧	{2.0, 2.0}	Daisy
	b39	i6	{×, ×}	⟦[0.1, 10], [−5, 5]⟧	{7.87E+06, 8.40E+06}	Daisy
	b40	NMSEexample33	{×, ×}	⟦[0, 100], [0, 100]⟧	{1.68E+06, 1.54E+06}	original
	b41	NMSEexample332	{✓, ×}	⟦[0, 100], [0, 100]⟧	{495.77, 1740.88}	original
	b42	NMSEexample335	{×, ×}	⟦[0, 100], [0, 100]⟧	{1.0, 1.0}	original
	b43	NMSEexample346	{✓, ✓}	⟦[0, 1], [1, 2]⟧	{4.07, 4.04}	original
	b44	NMSEsection35	{×, ×}	⟦[0, 1], [0, 1]⟧	{0.75, 0.75}	Herbie
	b45	polarToCarthesianX	{✓, ×}	⟦[1, 10], [0, 360]⟧	{8.23E+12, 2}	original
	b46	polarToCarthesianY	{✓, ×}	⟦[1, 10], [0, 360]⟧	{2.16E+12, 1.34E+12}	original
	b47	sec4example	{×, ×}	⟦[1.001, 2], [1.001, 2]⟧	{1.25, 1.25}	Herbie
	b48	test03_nonlin2	{✓, ✓}	⟦[0, 1], [−1, −0.1]⟧	{1.375, 1.25}	original
	b49	theta	{✓, ✓}	⟦[1, 100], [1, 100]⟧	{2.0, 2.0}	original
	b50	turbine1	{✓, ✓}	⟦[−4.5, −0.3], [0.4, 0.9], [3.8, 7.8]⟧	{3.00, 2.83, 2.85}	original
control flow	b51	squareRoot3	✓	⟦[0, 10]⟧	0.5	original
	b52	squareRoot3Invalid	✓	⟦[0, 10]⟧	0.5	original
	b53	cav10	✓	⟦[0, 10]⟧	0.5	original
	b54	gustafsonExample	✓	⟦[0, 100]⟧	0.5	original
	b55	smartRoot	×	⟦[−2, 2]⟧	2	original
	b56	triangleSorted	✓	⟦[1, 9]⟧	0.5	original
real-life	b57	pov_ray	{×, ×, ×, ×, ×}	⟦[$-\pi$, π], [$-\pi$, π], [−1, 1], [−1, 1], [−1, 1]⟧	{6.06E+15, 1.05E+16, 2.0, 1.11E+04, 9.86E+03}	Herbie
	b58	polyIDX0	{×, ×, ×, ×}	⟦[3.0, 9.0], [1.5, 5.0], [−2, 10] , [−1.0, 3.5]⟧	{3.88E+05, 9.34E+05, 2.25E+05, 6.0}	Herbie
	b59	polyIDX1	{×, ×, ×, ×}	⟦[3.0, 9.0], [1.5, 5.0], [−2, 10] , [−1.0, 3.5]⟧	{7.59E+05, 9.71E+05, 6.34E+05, 6.0}	Daisy
	b60	polyIDX2	{×, ×, ×, ×}	⟦[3.0, 9.0], [1.5, 5.0], [−2, 10] , [−1.0, 3.5]⟧	{6.0, 9.88E+05, 5.61E+05, 5.61E+05}	original

mae should always be 1 for ARFA, and a larger normalized number is worse. To better reflect the difference between the used approaches, we set the maximal value for the normalized *mae* to 5, and those greater than this value are not exactly depicted.

We also report how close a system can get to the maximal effective number of bits (ENOB). As the maximal ENOB ENOB_{max} depends on the precision of the variables, we also normalize these results. Specifically, an FP expression using 64-bit variables can have at most 53 ENOB, which is 24 for one with 32-bit variables. We have 3 benchmarks (b37-b39) using 32-bit variables; all of the remaining use 64-bit variables. When a system obtains an *mae* for an expression, we first compute the ENOB number for its *mae* using

$$\mathrm{ENOB} = \mathrm{ENOB}_{max} - \log_2(mae) = \mathrm{ENOB}_{max} - \log_2\left(\max_{\forall x \in \mathcal{D}}\left(\frac{|p(x) - f(x)|}{\mathtt{ulp}(f(x))}\right)\right). \tag{8}$$

Next, we normalize Expr. (8) with respect to ENOB_{max}. ENOB is the metric used by Herbie and NumOpt to report their experimental results. The downwards bars of Fig. 7 show these normalized numbers.

ARFA always exhibits lower *mae* than Herbie, because it always finds effective regime strategies while Herbie fails to achieve this goal for many benchmarks. Table 3 reports the number of regimes found by our work, with the effect of code optimization also shown. Besides, ARFA's rewrite search algorithm is better than those of others. Herbie builds an e-graph and tries to simplify expressions when possible, with rewriting and series expansion both considered, but it often rewrites f_e without regime partition. Take *predator-Prey*(x) as an example. Its original expression obtains the $mae = 2.79$ *ulps* at $x = 0.13017$, but Herbie increases it to $3.67E + 15$ *ulps* at $x = 0.9999996$. Our prepossessing can take as input these optimized expressions and enhance the accuracy.

For b4, the original expression is already with ENOB_{max}, the considered tools thus did not enhance its accuracy. The closeness to ENOB_{max} is use to demonstrate that (1) ARFA can optimize the *mae* when its original ENOB number is not so close to the maximal one (b7, b8, b19, b45, and b56) and (2) the true value of *mae* can still be lowered when the original ENOB number is already very close to ENOB_{max} (b25-b28, b30, b34 and b35).

None of the above-mentioned tools optimizes b39, whose expression is $sinf(x * y)$, since $x * y$ has no equivalent expressions. By enhancing precision, ARFA replaces the 32-bit *sinf* with its 64-bit equivalence *sin*. The reason why the *mae* is still not optimized by ARFA resides in the regime inference algorithm. Its *mae* is outside the regimes that need rewriting. However, its *aae* across $\mathcal{D}$ can be improved slightly, as will be discussed in § 7.2.

ARFA obtains larger *mae* than NumOpt for 8 benchmarks. For b7, b13, b33, b39, b53, and b57, ARFA obtains finer-grained regimes than NumOpt, and its number of regimes is large than that found by NumOpt. Both approaches rewrite an expression within each regime and only consider precision enhancing as a complementary optimization when no better rewrite substitution can be found. Due to the larger number of regimes, ARFA is more likely to rewrite an expression than enhancing its precision, but the accuracy improvement effect of precision enhancing is usually better than a rewrite substitution. For b40 and b41, ARFA performs less well than NumOpt because it only rewrites f_o in its $\prod_{i=1}^{n} R_i$ regimes that need rewriting, which has been explicitly described by Expr. (7). For the remaining one sub-domain, ARFA still uses f_o. Note that NumOpt always rewrites an expression across $\mathcal{D}$ regardless whether it is split or not. This is closely related to the upper bound u defined by our approach. Further optimizing the single sub-domain can alleviate this issue.

Table 3: Comparison of regimes (fused by the code optimization in the 3rd column) and the overhead of ARFA.

no.	Regimes R_i		Overhead of ARFA (in seconds)					
	Original	Fused	Preprocessing	Regime inference: Sketching	Regime inference: Splitting	Rewrite search	Code generation	Overall
b1	34	19	1.69	20.91	4.52E-03	70.74	0.86	94.20
b2	2	2	2.01	20.83	1.69E-03	4.02	0.95	27.81
b3	1	1	2.11	35.02	1.86E-03	6.71	0.97	44.82
b4	9	6	1.67	21.89	3.66E-02	11.10	0.83	35.53
b5	3	1	1.95	21.03	7.88E-04	2.17	1.09	26.24
b6	16	5	1.84	21.03	2.20E-03	25.90	0.79	49.55
b7	3	3	2.46	20.63	1.06E-03	6.40	1.19	30.67
b8	0	0	2.30	0.00	0.00E+00	0.00	0.00	2.30
b9	15	3	2.45	20.99	2.97E-03	70.10	1.11	94.65
b10	13	6	1.75	21.06	1.61E-03	105.59	0.81	129.21
b11	75	1	1.77	21.23	1.62E-02	2.12	0.94	26.08
b12	1	1	3.11	31.51	8.55E-04	12.52	1.01	48.15
b13	1	1	1.13	30.89	8.41E-04	3.65	0.90	36.58
b14	18	10	1.77	20.95	1.64E-03	44.41	0.87	68.00
b15	2	2	1.76	20.74	1.56E-03	7.49	0.79	30.77
b16	3	3	2.36	35.83	5.61E-04	31.45	1.05	70.69
b17	12	6	2.11	21.65	5.63E-03	11.70	1.10	36.56
b18	2	2	1.21	30.90	5.17E-03	4.46	0.94	37.52
b19	5	2	1.87	20.65	1.16E-03	11.35	0.83	34.70
b20	3	3	2.01	20.72	9.30E-04	6.68	1.12	30.52
b21	5	4	2.01	37.18	7.85E-03	1.88	0.98	42.06
b22	2	2	1.22	31.63	3.76E-03	10.97	1.20	45.01
b23	1	1	2.05	19.25	2.38E-04	2.04	1.05	24.39
b24	1	1	1.83	20.73	1.37E-03	4.73	0.90	28.19
b25	2	2	2.06	20.22	1.35E-03	358.16	0.89	381.32
b26	2	2	2.27	20.75	8.06E-04	13.09	0.87	36.98
b27	1	1	1.83	20.68	3.43E-04	94.18	0.88	117.58
b28	40	3	1.74	22.49	1.45E-02	5.63	0.86	30.74
b29	1	1	2.11	21.53	1.32E-03	6.07	0.84	30.54
b30	1	1	1.67	21.57	1.28E-03	6.10	0.83	30.17
b31	1	1	1.68	20.42	1.92E-03	4.61	0.85	27.56
b32	{1, 1}	{1, 1}	2.31	19.80	2.21E-03	6.25	1.14	29.51
b33	{1, 1}	{1, 1}	1.81	14.14	6.23E-04	7.09	0.89	23.92
b34	{3,8, 101}	{1,4,1}	6.28	14.81	4.58E-03	439.85	3.09	464.03
b35	{2,9,6}	{1,5,2}	6.37	14.49	4.59E-03	322.72	3.09	346.67
b36	{13, 7,49}	{8, 5,11}	6.21	15.71	4.09E-03	229.06	3.11	254.10
b37	{0,0}	{0,0}	1.75	0.00	0.00E+00	0.00	0.00	1.75
b38	{0,0}	{0,0}	1.68	13.18	2.41E-02	3.29	1.23	19.41
b39	{1,1}	{1,1}	2.10	16.04	5.81E-03	2.14	1.13	21.42
b40	{1,1}	{1,1}	2.15	19.79	4.27E-04	4.37	1.22	27.53
b41	{1,1}	{1,1}	2.63	21.84	9.26E-04	5.74	1.34	31.54
b42	{1,1}	{1,1}	2.08	17.73	5.23E-03	8.99	1.12	29.92
b43	{1,2}	{1,2}	2.86	27.98	6.91E-04	3.83	2.03	36.69
b44	{1,1}	{1,1}	1.90	15.59	5.19E-04	2.40	1.16	21.05
b45	{1,7}	{1,2}	3.07	15.85	3.68E-02	5.82	1.11	25.88
b46	{1,1}	{1,1}	2.91	17.03	1.58E-02	11.24	1.08	32.27
b47	{2,2}	{2,2}	1.92	13.15	6.66E-04	0.00	0.93	16.00
b48	{2,2}	{2,2}	1.67	12.55	1.49E-03	2.17	0.96	17.36
b49	{41,26}	{6,11}	3.41	20.32	5.22E-03	12.51	1.46	37.71
b50	{2,1,3}	{2,1,3}	5.32	14.03	1.05E-03	1840.57	2.63	1862.54
b51	3	3	0.85	117.49	5.97E-03	6.62	0.84	125.80
b52	3	3	0.92	21.62	5.77E-03	5.72	0.92	29.18
b53	7	3	0.83	19.73	4.28E-03	10.24	0.84	31.64
b54	0	0	0.95	0.00	0.00E+00	0.00	0.00	0.95
b55	2	2	0.87	16.52	1.13E-03	5.41	0.94	23.73
b56	0	0	2.23	0.00	0.00E+00	0.00	0.00	2.23
b57	{2,2,2,2,1}	{1,1,1 2,1}	2.84	14.57	1.42E-03	677.49	1.48	696.39
b58	{1,1,1,1}	{1,1, 1,1}	4.87	15.39	1.11E-03	66.78	1.72	88.77
b59	{1,1,1,1}	{1,1, 1,1}	1.47	14.76	1.14E-03	56.31	1.49	74.03
b60	{1,1,1,1}	{1,1, 1,1}	2.76	18.54	2.87E-03	13.80	1.37	36.47

There exists some examples for which other tools did not have data in Fig. 7 due to execution failures or out-of-time execution

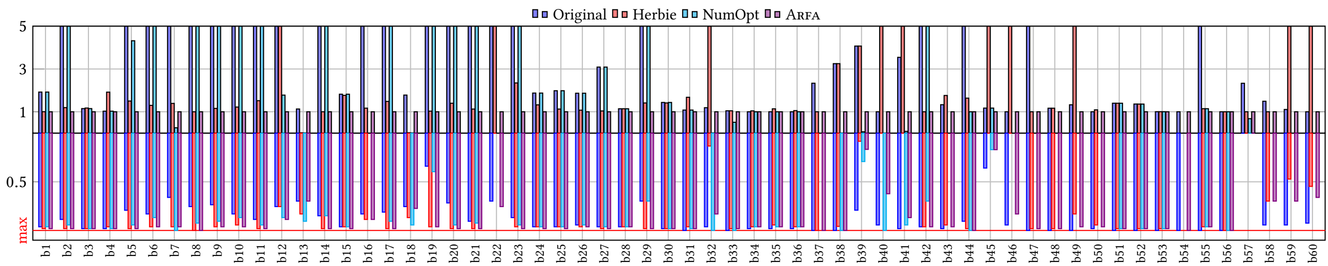


Figure 7: The normalized *mae* (the upwards y axis; lower is better.) and closeness to the maximal ENOB (the downwards y axis; longer is better.). x axis is benchmark labels. An *mae* has the largest ENOB when its downwards bar touches the red line.

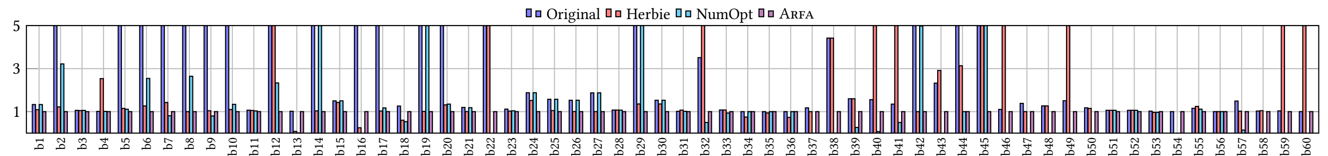


Figure 8: The normalized *aae*, represented by the y axis; lower is better. x is the label number of each benchmark.

issue. For b57, the original *mae* is 2.83E+16, which is reduced by Arfa to 1.21E+16. Both are 0 ENOB, but Arfa can still reduce the errors.

As for ENOB, Arfa and NumOpt improve the ENOB numbers of f_e by 4.73 bits and 4.42 bits, respectively. Although the average means of these two tools are close, Arfa achieves higher ENOB improvements for some of the benchmarks: Arfa achieves a 33-bit accuracy for b20, but the largest ENOB improvement of NumOpt is 20 for b40. Arfa usually leads to positive improvements or does not degrade the ENOB number (b3, b33, and b48), because the original ENOB numbers are already very close to the maximum (Fig. 7). In contrast, Herbie some reduces the ENOB numbers significantly (b32, b39, b50, b58, b59, and b60) though it also achieves a high ENOB number for b19 and b20. This makes it average mean improvement of ENOB number is close to 0.

7.2 Comparison of *aae*

To demonstrate Arfa can also optimize other metrics, we reported the data of *aae* in this section, with the normalized results with respect to Arfa's results shown in Fig. 8.

Arfa achieves similar results to that of *mae*: its *aae* is the best one for 46 out of the 60 benchmarks. This demonstrates that Arfa exhibits a steady performance when considering different metrics although it uses *mae* to perform regime inference and rewrite generation/search. Herbie obtains lower *aae* for b13, b16, b18, b34, b35, b36, b37, etc., but the different is insignificant. This is because Arfa found a rewrite substitution with lower *mae* than other tools for these examples, but the *aae* is larger than those of Herbie. Using *aae* to infer regime inference can alleviate or even avoid this issue. The result of NumOpt is similar to its performance in § 7.1. Generally speaking, Herbie performs better when using *mae* as the metric.

We also collected the ENOB numbers for *aae*, which is similar to the case of *mae*. Due to the page limitation, we do not show this comparison here.

7.3 Isolating the Effects of Regime Inference and Rewrite Search of Arfa

A major difference between Algo. 1 and Herbie's regime inference is the increased number of dynamic samples. This section reports the result of Herbie$^+$ which samples the same number of inputs for regime inference. Unfortunately, Herbie$^+$ throws out an out-of-time exception when handling multi-variate functions, for which we thus set N to 1024^2, a maximal value that can guarantee the execution of Herbie$^+$. The *mae* and *aae* of Herbie$^+$ normalized with respect to their counterparts of Arfa are depicted in Fig. 9.

While achieving different results from Herbie, Herbie$^+$ still performs less well than Arfa concerning both *mae* and *aae*. The exceptional cases Herbie$^+$ obtains a smaller *mae* than Arfa are b13 and b16, for which the search heuristic randomly filters out the optimal rewrite candidate. In summary, this experiment further demonstrates the necessity to implement Algo. 1 and its superiority to the state of the art.

Arfa also allows its regime inference to work with other rewrite search heuristics. To validate this effect, we feed the regimes produced by Algo. 1 to Herbie, which we refer to as Herbie$^\alpha$, whose *mae* and *aae* normalized with respect to their counterparts of Arfa are also reported in Fig. 9.

Similar to Herbie$^+$, Herbie$^\alpha$ under-performs in most cases than Arfa concerning both *mae* and *aae* even when given the same partition strategy, validating that the rewrite search algorithm of Herbie often finds an inferior substitution than ours (Algo. 3). Some benchmarks observe smaller *mae* and/or *aae* than Arfa, for which the reason is two folded: one is the same as that in the case of Herbie$^+$, and the other is the same as explained in § 7.2.

7.4 Overhead

We report the overhead of Arfa in the rightmost three columns of Table 3, which shows that the code generation overhead is often

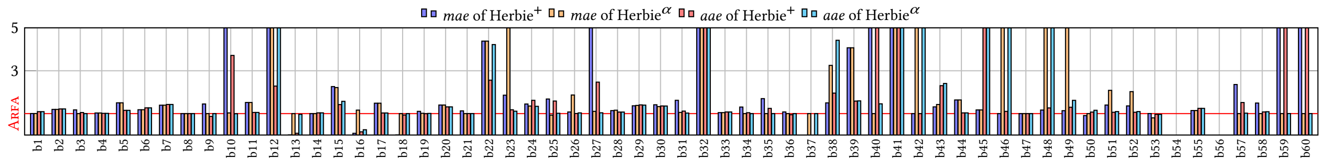


Figure 9: The normalized *mae* and *aae* of Herbie$^+$ and Herbie$^\alpha$ over those of Arfa. Lower is better.

less than 1 second due to parallel execution. In contrast, empirically inspecting errors consumes most of Arfa's execution time, we alleviate this issue using a tunable number of search trails in Algo. 3. To illustrate the effect of *num*, we report *mae* and the search overhead of b27. One can obtain the minimal *mae*, 0.66 *ulp*s, when *num* is set to 30, the total number of the top ranked rewrite candidates generated by Arfa. We perform five searches, each empirically inspecting the errors of *num* e-graph nodes, and compute the average mean for both *mae* and the search overhead. Table 4 shows that the *mae* gets smaller with the increase of *num*, but the execution time also becomes longer. As such, setting a reasonable *num* is important, and we set *num* to 10 as a well trade-off between this possibility and the search overhead. One can set *num* to its greatest possible value if he/she does not care about the search time.

Table 4: Effect of *num* on *mae* and search overhead in seconds.

num	*mae*	average mean of five searches: sequential (current result)	parallel (estimated result)
5	0.79	3.55	0.71
10	0.72	6.53	0.65
15	0.70	10.14	0.68
20	0.68	13.71	0.69
25	0.66	17.29	0.69
30	0.66	20.87	0.70

Sequentially executing the search algorithm can be parallelized to reduce its overhead. The rightmost column of Table 4 shows that the estimated execution time, each value of which is obtained by dividing its preceding number using *num*. One can expect for a constant overhead (about 0.70s) for this search process if parallelization is introduced in future. Besides, breaking a large, complex FP experssions into smaller sub-expressions using stable FP operators is another solution to alleviate its long overhead issue, because such operators do not introduce significant FP errors [41].

8 Related Work

Error detection is the first step to develop a tool for reducing errors. Arfa leverages MPFR [10] for error detection but can also be specialized to integrate with other methods. Arfa does not intend to prove tight bounds for errors like what static error analysis [5, 8, 9, 11, 15, 16, 32] did.

Many existing approaches leverages regimes to locate inputs of large errors for numerically unstable [23, 35, 37] or stable expressions [26]. Arfa also uses a dynamic sampling approach for regime inference, but the sampling number N is much larger than these tools. Increasing the number of samples Herbie cannot obtain the same effect of Arfa. An error plot is used for generating regimes by analyzing the fluctuation of its boundary line, without creating many equally-sized sub-domains like Regina [26]. Besides densely sampling, the workflow of Arfa also differs from that of Herbie. As depicted in Fig. 2, Arfa first performs regime inference and next produces rewrite expressions. On the contrary, Herbie carries out the regime inference as a post pass of its generation of rewrite expressions. In particular, Arfa leverages its preprocessing step to find a good start-up expression, which further distinguishes itself from Herbie.

Due to the great number of rewrite rules [21], a rewriting system usually has to evaluate many rewrite candidates. This promotes the research on equality saturation [25, 33, 36], a technique using a cost model to select rewrite candidates. Arfa also uses an e-graph to find a partial set of rewrite candidates with the given rules, but we empirically inspect the errors of selected rewrite candidates, which has the potential to scale with new FP representations [17, 38].

Another progress Arfa makes is that it usually allows for the generation of different rewrites for individual regimes, but other systems [7, 23, 26] often produce a uniform rewrite substitution for the whole $\mathcal{D}$. Arfa substitutes an FP expression using rewrite candidates inferred by combining the real-valued identities before building its e-graph, while some existing methods [23, 32, 37] also leverage Taylor-based approximations. Recent RLIBM-based techniques [1, 17] try to round math libraries with different FP variants, and oracle-free program synthesis is also used to optimize FP errors [39]. Integrating these approaches into Arfa is an interesting direction to follow. Arfa also increases precision as a complementary strategy. However, our rewrite search heuristic does not intend to automatically tune precision like what existing autotuners [13, 27, 28] did. Recently, precision tuning is also integrated into Herbie [29].

9 Conclusion

We introduce Arfa to reduce the errors of FP expressions. It first leverages a preprocessing step and uses an algorithm to partition the input domain of an FP expression, based on which the heuristic to generate and search customized rewrites is developed. In addition to searching equivalent expressions, Arfa also considers increasing precision as a complementary optimization. The results demonstrate that Arfa exhibits lower *mae* and *aae* than Herbie and NumOpt. For future tasks, we plan to generalize Arfa to handle more complex scenarios like control constructs and parallelize its execution to reduce the overhead. We also plan to add more rewrite rules and integrate the RLIBM-based approaches [1, 17].

Acknowledgments

The authors acknowledge the anonymous reviewers for their valuable comments, as well as Jiangwei Hao and Yuanyuan Xia for their feedback on the early versions of the paper. This work is supported by the National Natural Science Foundation of China under Grant No. U20A20226.

References

[1] Mridul Aanjaneya, Jay P. Lim, and Santosh Nagarakatte. 2022. Progressive Polynomial Approximations for Fast Correctly Rounded Math Libraries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 552–565. https://doi.org/10.1145/3519939.3523447

[2] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-Point Accuracy Problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI'12)*. Association for Computing Machinery, New York, NY, USA, 453–462. https://doi.org/10.1145/2254064.2254118

[3] S. Chevillard, M. Joldeş, and C. Lauter. 2010. Sollya: An Environment for the Development of Numerical Codes. In *Mathematical Software - ICMS 2010 (Lecture Notes in Computer Science, Vol. 6327)*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama (Eds.). Springer, Heidelberg, Germany, 28–31.

[4] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient Search for Inputs Causing High Floating-Point Errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/2555243.2555265

[5] George Constantinides, Fredrik Dahlqvist, Zvonimir Rakamarić, and Rocco Salvia. 2021. Rigorous Roundoff Error Analysis of Probabilistic Floating-Point Computations. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 626–650.

[6] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. 2017. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification*, Sergiy Bogomolov, Matthieu Martel, and Pavithra Prabhakar (Eds.). Springer International Publishing, Cham, 63–77.

[7] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 270–287.

[8] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL'14)*. Association for Computing Machinery, New York, NY, USA, 235–248. https://doi.org/10.1145/2535838.2535874

[9] Marc Daumas and Guillaume Melquiond. 2010. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Trans. Math. Softw.* 37, 1, Article 2 (jan 2010), 20 pages. https://doi.org/10.1145/1644001.1644003

[10] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2 (jun 2007), 13–es. https://doi.org/10.1145/1236463.1236468

[11] Zhoulai Fu and Zhendong Su. 2019. Effective Floating-Point Analysis via Weak-Distance Minimization. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 439–452. https://doi.org/10.1145/3314221.3314632

[12] David Goldberg. 1991. What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Comput. Surv.* 23, 1 (mar 1991), 5–48. https://doi.org/10.1145/103162.103163

[13] Hui Guo and Cindy Rubio-González. 2018. Exploiting Community Structure for Floating-Point Precision Tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/3213846.3213862

[14] Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE'20)*. Association for Computing Machinery, New York, NY, USA, 1261–1272. https://doi.org/10.1145/3377811.3380359

[15] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying Bit-Manipulations of Floating-Point. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI'16)*. Association for Computing Machinery, New York, NY, USA, 70–84. https://doi.org/10.1145/2908080.2908107

[16] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2017. On Automatically Proving the Correctness of Math.h Implementations. *Proc. ACM Program. Lang.* 2, POPL, Article 47 (dec 2017), 32 pages. https://doi.org/10.1145/3158135

[17] Jay P. Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. 2021. An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants. *Proc. ACM Program. Lang.* 5, POPL, Article 29 (jan 2021), 30 pages. https://doi.org/10.1145/3434310

[18] Debasmita Lohar, Milos Prokop, and Eva Darulova. 2019. Sound Probabilistic Numerical Error Analysis. In *Integrated Formal Methods*, Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa (Eds.). Springer International Publishing, Cham, 322–340.

[19] Sandra Loosemore, Richard M Stallman, Rolandand McGrath, Andrew Oram, and Ulrich Drepper. 2019. The GNU C Library Reference Manua. https://www.gnu.org/software/libc/manual/2.35/html_node/Errors-in-Math-Functions.html

[20] MathWorks. 2019. MATLAB & Simulink. https://www.matlab.com

[21] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

[22] Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford University.

[23] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI'15)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

[24] Tomas Plachetka. 1998. POV Ray: persistence of vision parallel raytracer. In *Proc. of Spring Conf. on Computer Graphics, Budmerice, Slovakia*. 123–129.

[25] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

[26] Robert Rabe, Anastasiia Izycheva, and Eva Darulova. 2021. Regime Inference for Sound Floating-Point Optimizations. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 81 (sep 2021), 23 pages. https://doi.org/10.1145/3477012

[27] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-Point Precision Tuning Using Blame Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1074–1085. https://doi.org/10.1145/2884781.2884850

[28] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-Point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC'13)*. Association for Computing Machinery, New York, NY, USA, Article 27, 12 pages. https://doi.org/10.1145/2503210.2503296

[29] Brett Saiki, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. 2021. Combining Precision Tuning and Rewriting. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*. 1–8. https://doi.org/10.1109/ARITH51176.2021.00013

[30] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 256–269. https://doi.org/10.1145/3192366.3192411

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

[32] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 2 (dec 2018), 39 pages. https://doi.org/10.1145/3230733

[33] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL'09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[34] Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. 2016. Detecting and Fixing Precision-Specific Operations for Measuring Floating-Point Errors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2950290.2950355

[35] Xie Wang, Huaijin Wang, Zhendong Su, Enyi Tang, Xin Chen, Weijun Shen, Zhenyu Chen, Linzhang Wang, Xianpei Zhang, and Xuandong Li. 2019. Global Optimization of Numerical Programs via Prioritized Stochastic Algebraic Transformations. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE'19)*. IEEE Press, 1131–1141. https://doi.org/10.1109/ICSE.2019.00116

[36] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[37] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient Automated Repair of High Floating-Point Errors in Numerical Libraries. *Proc. ACM Program. Lang.* 3, POPL, Article 56 (jan 2019), 29 pages. https://doi.org/10.1145/3290369

[38] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting Numerical Bugs in Neural Network Architectures. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 826–837. https://doi.org/10.1145/3368089.3409720

[39] Daming Zou, Yuchen Gu, Yuanfeng Shi, Mingzhe Wang, Yingfei Xiong, and Zhendong Su. 2022. Oracle-Free Repair Synthesis for Floating-Point Programs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 190 (oct 2022), 28 pages. https://doi.org/10.1145/3563322

[40] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE'15)*. IEEE Press, 529–539.

[41] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, Lu Zhang, and Zhendong Su. 2019. Detecting Floating-Point Errors via Atomic Conditions. *Proc. ACM Program. Lang.* 4, POPL, Article 60 (dec 2019), 27 pages. https://doi.org/10.1145/3371128