Towards Numerical Assistants

Trust, Measurement, Community, and Generality for the Numerical Workbench

Pavel Panchekha, Zachary Tatlock

May 2020

Abstract

The last few years have seen an explosion of work on tools that address numerical error in scientific, mathematical, and engineering software. The resulting tools can provide essential guidance to expert non-experts: scientists, mathematicians, and engineers for whom mathematical computation is essential but who may have little formal training in numerical methods. It is now time for these tools to move into practice.

Practitioners need a "numerical workbench" that not only succeeds as a research artifact but as a daily tool. We describe our experience adapting Herbie, a tool for numerical error repair, from a research prototype to a reliable workhorse for daily use. In particular, we focus on how we worked to increase user trust and use internal measurement to polish the tool. Looking more broadly, we show that community development and an investment in the generality of our tools, such as through the FPBench project, will better support users and strengthen our research community.

Floating-point rounding error is a problem for scientific, mathematical, and engineering software. Rounding error can destroy the utility of such software by rendering results meaningless; at the same time, few programmers—let alone programmers with expertise in the scientific, mathematical, or engineering domain in question—have the specialized numerical training necessary to account for and avoid rounding error.

The last few years have seen renewed interest from the broader programming languages community addressing this issue, with important progress in input generation [Kne17], error evaluation [SJRG15, TFMM18, TZPT19a], verification [ID17], tuning [DV19, DHS18, RGt13, CBB+17], and debugging [BHH12, SSPLT18, BZ13]. Herbie [PSSWT15] and FPBench [DMP+16] are our contribution to this effort. A unifying theme is the marriage of traditional programming language techniques with numerical notions of error, stability, and correctness. Tools cannot replace human expertise, but that expertise is rarely taught and thinly spread; tools can provide non-experts some recourse when numerical problems crop up.

In this presentation, we briefly reflect on our past efforts and future plans towards building a "numerical workbench" of tools to help empower more programmers to confidently and effectively develop numerical software.

1 Trust and Measurement in Herbie

Herbie is a tool for rearranging and rephrasing mathematical expressions to minimize floating-point error. For example, a developer may be interested in evaluating the expression $\sqrt{x+1} - \sqrt{x}$, but concerned about error. They can pass the expression to Herbie (via Herbie's graphical user interface), and Herbie will output the rearrangement $1/(\sqrt{x+1} + \sqrt{x})$, which has much less rounding error [Ham87]. Herbie is aimed at expert non-experts: people with extensive training in some domain, and a need for mathematical computation to support their work, but without specific training in mathematical computation and numerical methods. Herbie is freely available online:

`https://herbie.uwplse.org`

Since we first published Herbie in 2015 [PSSWT15], we have been maintaining and improving Herbie to help these expert non-experts; we will soon release Herbie 1.4, and have had yearly releases since publication. We've found that maintenance has required few changes to the core algorithms, but extensive work on *trust* and *measurement*, work that often seemed orthogonal to Herbie's goal but was necessary to support our users.

Trust Our users know that floating-point arithmetic is tricky, but they are rightly worried that replacing hand-written mathematical expressions with Herbie's suggestions will introduce bugs or make the code less maintainable; one textbook [Kne17] recommends keeping the original expression in a comment in case the code needs edits. We have thus built infrastructure for generating HTML reports with error charts, an interactive tester, and a derivation of the output expression from the input expression.[1] These reports reassure users, and also help us fix bugs and understand regressions.

Trust also stems from predictability. For example, Herbie can introduce `if` statements to select between alternative programs, and the branch conditions compare a variable to a constant:

$$\exp(x) - 1 \mapsto \left| \begin{array}{ll} \textbf{if} & -10^{-4} < x < 10^{-4} \\ \textbf{then} & x(1 + \frac{x}{2}(1 + \frac{x}{3})) \\ \textbf{else} & \exp(x) - 1 \end{array} \right.$$

Prior to Herbie 1.2, that constant was always selected from among the 256 randomly-sampled input points Herbie was using to evaluate rounding error. As a result, each Herbie run would yield a different `if` statement, with the constants chosen often differing by factors of two. Users found this suspicious: if the branch is important, why is the branch condition seemingly-unimportant? We added a binary search step to refine the constant by sampling more points, ensuring that constants differed by less than 3% across runs. This change didn't

[1] We recommend the reader try out the Herbie web demo, at `https://herbie.uwplse.org/demo/`, to see a report for themselves.

affect our metrics–the more-accurate constant was no different on the evaluation points—but gained trust with users.

We have likewise made changes to simplify Herbie's output expressions, even at the cost of a slight accuracy penalty. If users do not *trust* that Herbie is accurate, they will not use it, at much larger cost to accuracy.

Measurement After its initial publication, Herbie was, in the parlance of John D. Cook, a "software exoskeleton" [Coo]: a research prototype optimized to support quick development and experimentation, but featuring little coherence and several hacky kludges. That made it a poor fit for users: many bugs, confusing behavior, and poor explainability. Improving this state of affairs meant focusing our efforts on measurement and internal controls.

Doing so required first constructing a "theory of Herbie", giving each component a clear role and specification instead of evaluating Herbie end-to-end. For example, Herbie's regime inference component generates `if` statements that select between different floating-point expressions based on input ranges. That provides a clear measure of success: the resulting `if` statement is more accurate than any individual (branch-free) candidate program. Investigating the cases where it was not lead us to improve regime inference (by, for example, specially handling input points with equal outputs), even when end-to-end Herbie was performing well. Likewise, we developed an "oracle" for regime inference (evaluating each candidate program and selecting the one closest to a high-precision evaluation) to quantify how much accuracy regime inference was leaving on the table. Separating components according to their roles discovered new bugs, focused our efforts where they mattered, and improved Herbie's internal organization by enforcing a separation of concerns.

Measurement's other role was forcing us to consider alternative ways to evaluate Herbie's results. For example, comparison against Daisy [BPDT18] meant rectifying Herbie's average accuracy with Daisy's worst-case error analysis. The differences between the two lead to a better understanding of what "average accuracy" was really measuring: the size of the input space where the expression produced a meaningful answer. This realization led to a better understanding of Herbie's individual components. For example, regime inference could now be phrased as the union of input spaces, making its importance more clear.

Finally, measurement and trust build on one another. For example, we performed experiments to better establish Herbie's defaults. Our experiments showed that Herbie's default of 2048 randomly sampled inputs wasn't enough to find subtle errors while at the same time was overkill for finding simple errors, where even a hundred samples sufficed. We lowered the default, but not to the minimum possible: generality meant adding a generous safety margin, and Herbie now uses 256 sample points during search. But to ensure that subtle errors weren't ignored, we added a re-sampling stage with 8000 sample points for post-search validation. Here, better measuring the defaults made Herbie faster while increasing its trustworthiness.

2 Community and Generality in FPBench

FPBench [DMP+16] is a set of standards, benchmarks, and tools for studying the behavior of numerical kernels. Initially, we developed FPBench to enable interoperability between tools like Herbie [PSSWT15], Daisy [ID17], and Salsa [DM17]. These efforts proved successful [BPDT18] and several groups began adopting FPBench's interchange format and using its benchmarks for evaluating new tools [CR19, STF+19, JPV18, YCMJ17, ZMRM18]. As adoption has grown, we have discovered new challenges in supporting more diverse tools, especially with respect to multi-format, multi-precision (MPMF) computation and data structure support; newer versions of the FPBench standards and tools are adding features to address these challenges based on user feedback, discussion across the community, and early prototypes from the core FPBench development team.

Community Support Herbie is but one tool—practitioners ultimately need a complete workbench of tools, from design-time tools like Herbie to additional tools for input generation, debugging, repair, and verification to support their work. As an academic community, we have neither the coordination nor the budget to develop the complete workbench as a single, integrated tool. Thus, better supporting our users has meant developing standards and formats for combining tools, and ensuring that other researchers can make use of our work.

This need was the catalyst for the FPBench project [DMP+16]. Originally developed so that Herbie and Salsa [Mar09, DMC15, DM17] could share benchmarks, FPBench has grown into a standard format for expressing multi-precision, multi-format numerical computations. FPBench collects benchmarks from multiple research groups, adds standard metadata, and provides tools for compilation to other languages (including the input formats to other community tools) and for standard program transformations (such as loop unrolling). FPBench binds the community more tightly making it easier for users to try out multiple research tools.

Supporting users also means making it easier for users to discover new research tools and try them out with minimal investment. The FPBench community page (`https://fpbench.org/community.html`), maintained by Mike Lam, attempts a comprehensive listing of floating-point research tools, along with brief summaries about what they do, making it easy for non-experts to explore and understand the space. And some of these tools, including Herbie, offer a web-based demo tool, reducing the cost for users to try them out. We make it easy to launch these tools from the FPBench benchmarks page (`https://fpbench.org/benchmarks.html`) so users can immediately try a tool without even having to come up with example inputs.

These investments in a cohesive community of floating-point researchers helps users by surfacing useful tools and allowing those tools to interoperate, and has also helped us as researchers compare and contrast multiple tools [BPDT18], uncover bugs in Herbie, and bring in a continuous stream of users that file bugs, make improvements, and suggest features that make Herbie more practical.

Generality Trust and measurement make a tool better; users then apply that tool to new problems and expanded domains. One clear example has been interest in applying Herbie to new number systems. Herbie was initially developed to support rearrangement of double-precision formulas; while it has always supported single-precision as well, that support has been fragile. Meanwhile, new number systems such as posits [GY17] have become increasingly prominent. Researchers thus reached out to see if Herbie could better support posits, bfloat16, and other novel number systems being developed. Herbie 1.3 thus introduced a split between types (such as real numbers) and representations (such as double-precision), and added a plugin system to define new representations via their bit representation and real values.

However, supporting new number requires strengthening Herbie's other components. For example, Herbie uses random sampling to evaluate the error of expressions; this requires establishing a ground-truth, correct result for an expression at a point. Herbie uses arbitrary-precision arithemtic for this, and that requires choosing a sufficient precision. Herbie originally used a hand-tuned algorithm that worked well in double-precision but turned out to fail disastrously for lower precision, returning incorrect ground-truth values and corrupting Herbie's results. This leads us to switch to a slower but provably-sound interval-analysis technique [LB90]. As a benefit, the technique is not only sound but also more easily measurable and generalizes to arbitrary number representations.

Users would also like to apply Herbie to expressions besides pure floating-point expressions, most importantly matrix operations. This too requires flexibility, most importantly allowing users to define new rewrite rules for Herbie to use. While Herbie always operated from a library of rewrite rules, its internal algorithms were quite brittle, and tweaking the set of rules used was frequently necessary, making user-extensibility difficult. Our work on Egg [WWF+20] was motivated by the need to build easy, efficient, and extensible tools for operating with large libraries of rewrite rules, and involved a breakthrough in congruence-closure data structures that made Herbie's internal algorithms 60× faster, with gains continuing as we improve this library.

3 The Future

The growth of machine learning, robotics, and high-performance computing are making numerical errors a growing concern. In machine learning, for example, a growing interest in custom hardware and new number representations [Int18, Joh18] means numerical techniques need to be ported or reimagined for new number representations. Tools must expand in scope from short expressions to matrix and tensor computations. Trust, measurement, and generality remain key themes but must be extended to new domains.

Trust requires that the rise of new precisions and formats, and the rise of multi-precision, multi-format computation is accompanied by clear semantics and standard formats. Yet existing languages usually do not offer this, relegating the precise meaning of a floating-point computation to ad-hoc interpretations of

IEEE-754. In FPBench, we are attempting to standardize on a format general enough to incorporate many proposed number systems [TZPT19b], and we are working on making Herbie's core support multiple precisions. The work is, however, rough going: a careful split between the real-number and floating-point semantics has to be brought in, and mixed-precision computations raise complex issues even at the level of type-checking programs.

Measurement also requires that FPBench expand to support new applications. Beyond the mixed-precision support added in FPBench 1.2, matrix and tensor computations must be given expression, given their clear centrality in modern numerical work. In FPBench 2.0 we hope to add the `tensor` and `for` constructs for regular loop structures and tabular data structures. Supporting these in analysis tools, however, will be its own challenge, since tensors introduce new exceptional cases (array out of bounds), new data types (indices), and new scaling challenges (thousands of variables). It is unclear how tools will adapt.

Finally, the rapidly evolving numerical environment also raises the burden of *generality*. Herbie, at its publication and still, relies on a core of hand-crafted rules combined together with a general-purpose search algorithm. As programs get more complex, crafting these rules is a challenge, and more and more special cases (triangular matrices, tridiagonal matrices, ...) mean that user extensibility becomes essential. We have been simplifying Herbie's core to rely more on components which support such generalization, transitioning more of Herbie to equivalence graphs and trading our custom evaluation algorithm for a traditional, interval-analysis based approach. We hope generality and simplicity can insulate us from change, but that remains to be seen.

Conclusion Future programmers will face a vastly more complex numerical space, with novel number representations, new applications, and less support from the numerical methods literature. They will thus rely on a constellation of tools that help them evaluate error, improve their programs, and debug issues. Our experience building and maintaining Herbie suggests that many of these tools are on the cusp of user-friendliness, and that investment in trust, measurement, and generality will pay off for both researchers and the users of these tools. Furthermore, investing in community infrastructure will enable better research and greater impact for the whole community.

Acknowledgements Herbie and FPBench are the result of many years of work from stellar contributors including Alex Sanchez-Stern, Bill Zorn, David Thien, Oliver Flatt, Brett Saiki, Jason Qiu, Ian Briggs, Heiko Becker, Eva Darulova, Max Willsey, James Wilcox, and many others. This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[BHH12] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. PLDI '12, pages 453–462, New York, NY, USA, 2012. ACM.

[BPDT18] Heiko Becker, Pavel Panchekha, Eva Darulova, and Zachary Tatlock. Combining tools for optimization and analysis of floating-point computations. FM, pages 355–363, 2018.

[BZ13] Tao Bao and Xiangyu Zhang. On-the-fly detection of instability problems in floating-point program execution. *SIGPLAN Not.*, 48(10):817–832, October 2013.

[CBB+17] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous floating-point mixed-precision tuning. POPL, pages 300–315, 2017.

[Coo] John D. Cook. Software exoskeletons. `https://www.johndcook.com/blog/2011/07/21/software-exoskeletons/`. Accessed: 2020-06-05.

[CR19] M. Claude and M. Rueher. Dedicated search strategies for finding critical counterexamples in programs with floating point computations. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 138–139, 2019.

[DHS18] Eva Darulova, Einar Horn, and Saksham Sharma. Sound mixed-precision optimization with rewriting. ICCPS, pages 208–219, 2018.

[DM17] Nasrine Damouche and Matthieu Martel. Salsa: An automatic tool to improve the numerical accuracy of programs. AFM, 2017.

[DMC15] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Formal methods for industrial critical systems: 20th international workshop, fmics 2015 oslo, norway, june 22-23, 2015 proceedings. pages 31–46, 2015.

[DMP+16] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. July 2016.

[DV19] Eva Darulova and Anastasia Volkova. Sound approximation of programs with elementary functions. CAV, pages 174–183, 2019.

[GY17] John Gustafson and Isaac Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 2017.

[Ham87] Richard Hamming. *Numerical Methods for Scientists and Engineers*. Dover Publications, 2nd edition, 1987.

[ID17] Anastasiia Izycheva and Eva Darulova. On sound relative error bounds for floating-point arithmetic. FMCAD, pages 15–22, 2017.

[Int18] Intel. BFLOAT16 – Hardware Numerics, 2018. White Paper, Document Number: 338302-001US, Revision 1.0.

[Joh18] Jeff Johnson. Rethinking floating point for deep learning. *CoRR*, abs/1811.01721, 2018.

[JPV18] Maxime Jacquemin, Sylvie Putot, and Franck Védrine. A reduced product of absolute and relative error bounds for floating-point analysis. In Andreas Podelski, editor, *Static Analysis*, pages 223–242, Cham, 2018. Springer International Publishing.

[Kne17] R.T. Kneusel. *Numbers and Computers*. Springer International Publishing, 2017.

[LB90] Vernon A. Lee and Hans-J. Boehm. Optimizing programs over the constructive reals. PLDI '90, New York, NY, USA, 1990.

[Mar09] Matthieu Martel. Program transformation for numerical precision. PEPM '09, pages 101–110, New York, NY, USA, 2009. ACM.

[PSSWT15] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. PLDI, 2015.

[RGt13] Cindy Rubio-González *et al.* Precimonious: Tuning assistant for floating-point precision. SC, pages 1–12. IEEE, 2013.

[SJRG15] Alexey Solovyev, Charlie Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. FM, 2015.

[SSPLT18] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. Finding root causes of floating point error. PLDI, pages 256–269, 2018.

[STF+19] Rocco Salvia, Laura Titolo, Marco A. Feliú, Mariano M. Moscato, César A. Muñoz, and Zvonimir Rakamarić. A mixed real and floating-point solver. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, pages 363–370, Cham, 2019. Springer International Publishing.

[TFMM18] Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Munoz. An abstract interpretation framework for the round-off error analysis of floating-point programs. VMCAI, pages 516–537, 2018.

[TZPT19a] D. Thien, B. Zorn, P. Panchekha, and Z. Tatlock. Toward multi-precision, multi-format numerics. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 19–26, 2019.

[TZPT19b] David Thien, Bill Zorn, Pavel Panchekha, and Zachary Tatlock. Toward multi-precision, multi-format numerics. In Ignacio Laguna and Cindy Rubio-González, editors, *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), Denver, CO, USA, November 18, 2019*, pages 19–26. IEEE, 2019.

[WWF+20] Max Willsey, Yisu Remy Wang, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. egg: Easy, efficient, and extensible e-graphs, 2020.

[YCMJ17] X. Yi, L. Chen, X. Mao, and T. Ji. Automated repair of high inaccuracies in numerical programs. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 514–518, 2017.

[ZMRM18] heytem Zitoun, Claude Michel, Michel Rueher, and Laurent Michel. Sub-domain Selection Strategies For Floating Point Constraint Systems, July 2018. 24th International Conference on Principles and Practice of Constraint Programming Doctoral Program CP 2018.