Toward a Standard Benchmark Format and Suite for Floating-Point Analysis

Nasrine Damouche[1] Matthieu Martel[1] Pavel Panchekha[2]
Chen Qiu[2] Alexander Sanchez-Stern[2] Zachary Tatlock[2]

Université de Perpignan Via Domitia[1] University of Washington[2]

Abstract. We introduce FPBench, a standard benchmark format for validation and optimization of numerical accuracy in floating-point computations. FPBench is a first step toward addressing an increasing need in our community for comparisons and combinations of tools from different application domains. To this end, FPBench provides a basic floating-point benchmark format and several quality measures for comparing different tools. Programs expressed in the FPBench format can also serve as a common intermediate representation to enable the combination of different tools. We describe the FPBench format and measures and show that FPBench expresses benchmarks from recent papers in the literature, by building an initial benchmark suite drawn from these papers. We intend for FPBench to grow into a standard benchmark suite for the members of the floating-point tools research community.

1 Introduction

The increasingly urgent demand for reliable software has led to tremendous advances in automatic program analysis and verification [8,4,6,5,18]. However, these techniques have typically focused on integer programs, and do not apply to the floating-point computations we depend on for safety-critical control in avionics or medical devices, nor the analyses carried out by scientific and computer-aided design applications. In these contexts, floating-point accuracy is critical since subtle rounding errors can lead to significant discrepancies between floating-point results and the real results developers expect. Indeed, floating-point arithmetic is notoriously unintuitive and its sensitivity to roundoff errors makes such computations fiendishly difficult to debug. Traditionally, such errors have been addressed by numerical methods experts who manually analyze and rewrite floating-point code to ensure accuracy and stability. However, these manual techniques are difficult to apply and typically do not lead to independently checkable certificates guaranteeing the accuracy.

The research community has responded to these challenges by developing a rich universe of automated techniques that provide guaranteed bounds on the accuracy of floating-point computations or attempt to automatically improve accuracy. For example, Fluctuat [12,13], used in many companies, performs static analysis of `C` programs to prove a bound on the rounding errors introduced by the use of floating-point numbers instead of reals. Fluctuat also helps users debug

floating-point errors by detecting the operations responsible for significant precision loss. Salsa [10] automatically improves the numerical accuracy of programs by using an abstract interpretation to guide transformations that minimize the errors arising during computations. Herbie [20] uses a heuristic search to improve the numerical accuracy of an arithmetic expression by estimating and localizing the roundoff errors of an expression using sampled points, applying a set of rules in order to improve the accuracy of the expression and combining these improvements for different input domains. Rosa [11] combines an exact SMT solver on reals with sound affine arithmetic to verify accuracy post-conditions from assertions about the accuracy of inputs. Rosa can guarantee that the desired precision can be soundly obtained in a finite-precision implementation when propagation error is included. Finally, FPTaylor [22] improves on interval arithmetic by using Taylor series to narrow the computed error bounds.

As the number of tools dedicated to analyzing and improving numerical accuracy grows, it becomes increasingly difficult to make fair comparisons between the techniques. This is because each tool is targeted to slightly different domains, uses slightly different formats for expressing benchmarks, and reports results using related but slightly different measures. Furthermore, without any standard set of floating-point benchmarks, it is difficult to identify opportunities for composing complementary tools.

To address these challenges, the floating-point research community needs a standard benchmark format and common set of measures that enables comparison and cooperation between tools. This goal is motivated by the success of standard benchmark suites like SPEC [17] and SPLASH-2 [23] in the compiler community, the DIMACS [2] format in the SAT-solving community, and the SMT-LIB [3] format in the SMT-solving community. The formats have enabled fair comparisons between tools, crisp characterizations of the tradeoffs between different approaches, and useful cooperation between tools with complementary strengths.

In this article, we propose FPBench, a general floating-point format and benchmark suite. FPBench provides a common language for floating-point tools, allowing users to combine tools that complementary tasks, or compare competing tools to choose the one best for their task. The common scientific methodology FPBench enables is crucial for demonstrating the improvements of each tool on the state of the art.

The main contributions of this article are the following:

(i) A uniform format for expressing floating-point benchmarks, FPCore.
(ii) A set of utilities for converting to and from FPCore programs, and working with FPCore programs.
(iii) A set of measures on which to evaluate various floating-point tools on FPBench benchmarks.
(iv) An initial suite of benchmarks drawing from existing floating-point literature.

The remainder of this article is organized as follows. Section 2, describes the FPBench formats. Section 3 introduces various measures of floating-point er-

ror. Section 4 describes the utilities FPBench provides to support creating and working with benchmarks. Section 5 surveys our existing benchmark suite, highlighting representative case studies from recent tools in the literature. Finally, Section 6 discusses future work and concludes.

2 Benchmark Format

A common floating-point benchmark format must be easy to parse, have simple and clear semantics, support floating-point details, and provide sufficient expressiveness for diverse application domains. To satisfy these requirements, FPBench provides the FPCore format, a minimal expression-based language for floating-point computations with features for iteration, input ranges, error measures, target accuracy, and floating-point precision. A common floating-point benchmark format must also be easy to translate to and from popular industrial languages like C, C++, Matlab, and Fortran. To satisfy these requirements, FPBench also provides the extended FPImp format, a basic imperative language for floating-point computations which can be automatically compiled to FPCore.

2.1 FPCore

FPCore is an S-expression format featuring mathematical functions, `if` statements, and `while` loops. All floating-point functions from C11's `math.h` and all Fortran 2003 intrinsics are supported operators, as well as standard arithmetic operators like addition and comparison; likewise, all constants defined in C11's `math.h` are available as constants. Following IEEE754 and common C and Fortran runtimes, FPCore does not prescribe the accuracy of built-in mathematical functions. However, individual benchmarks can declare the accuracy they assume for built-in operations which analyzers can take into account.

A FPCore benchmark specifies a set of inputs, a floating-point expression, and meta-data flags including a name, citations, preconditions on inputs, and the floating-point precision (`binary32`, `binary64`, ...) used to evaluate that benchmark. The full FPCore syntax is as follows: [1]

⟨*FPCore*⟩ ::= ('`FPCore`' (⟨*identifier*⟩*) ⟨*property*⟩* ⟨*expression*⟩)

⟨*expression*⟩ ::= ⟨*constant*⟩ | ⟨*identifier*⟩ | (⟨*operation*⟩ ⟨*expression*⟩*)
| ('`if`' ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩)
| ('`while`' ⟨*expression*⟩ (⟨*loopvar*⟩*) ⟨*expression*⟩)
| ('`let`' (⟨*binding*⟩*) ⟨*expression*⟩)

⟨*loopvar*⟩ ::= [⟨*identifier*⟩ ⟨*expression*⟩ ⟨*expression*⟩]

⟨*constant*⟩ ::= ⟨*number*⟩ | '`E`' | '`LOG2E`' | '`LOG10E`' | '`LN2`' | '`LN10`' | '`PI`' | ···

⟨*operation*⟩ ::= '`+`' | '`*`' | '`<`' | ··· | '`abs`' | '`acos`' | '`and`' | '`asin`' | ···

[1] Not shown in the grammar: FPBench uses "`;`" the indicate that the remainder of a line is a comment.

⟨*property*⟩ ::= ⟨*propname*⟩ ⟨*propval*⟩

⟨*propname*⟩ ::= '`:name`' | '`:cite`' | '`:pre`' | '`:type`' | ⋯

⟨*propval*⟩ ::= ⟨*expression*⟩ | ⟨*string*⟩ | (⟨*identifier*⟩*) | ⟨*identifier*⟩

⟨*binding*⟩ ::= [⟨*identifier*⟩ ⟨*expression*⟩]

Since the language is S-expression based, parentheses and braces are literals. The semantics of these programs is ordinary function evaluation, with `let` bindings evaluated simultaneously and `while` loops evaluated by simultaneously updating the loop variables until the condition is true, and then evaluating the return value:

$$\frac{H : x_i^0 \Downarrow v_i \qquad H[x_i \mapsto v_i]_i : c \Downarrow \top \qquad H[x_i \mapsto v_i]_i : e_i \Downarrow x_i' \qquad H : (\texttt{while}\ \ c\ \ ([x_i\ \ x_i'\ \ e_i]_i)\ \ y) \Downarrow v}{H : (\texttt{while}\ \ c\ \ ([x_i\ \ x_i^0\ \ e_i]_i)\ \ y) \Downarrow v}$$

$$\frac{H : x_i^0 \Downarrow v_i \qquad H[x_i \mapsto v_i]_i : c \Downarrow \bot \qquad H[x_i \mapsto v_i]_i : y \Downarrow v}{H : (\texttt{while}\ \ c\ \ ([x_i\ \ x_i^0\ \ e_i]_i)\ \ y) \Downarrow v}$$

The list of properties is used to record additional information about each benchmark. Existing benchmarks are annotated with a `:name`, a `:description` of the benchmark and its inputs, the floating-point `:type` (either `binary32` or `binary64`), a precondition `:pre`, and a citation `:cite`. All FPBench tools ignore unknown attributes, so they represent an easy way to record additional benchmark information. We recommend that properties specific to a single tool be prefixed with the name of the tool.

2.2 FPImp

Where FPCore is the format consumed by tools, FPImp is a format for simplifying the translation from imperative languages to FPCore. While FPCore is a functional language with a minimal set of features for representing floating-point computations, FPImp includes common imperative features like variable assignments, multiple return values, and multi-way `if` statements. More complex features, such as arrays, pointers, records, and recursive function calls, are left to future language extensions. In our experience, translating C, Fortran, and Matlab to FPImp is relatively easy. FPImp has expressions similar to those in FPCore, but without `if`, `while`, and `let` expressions; these are instead replaced with statements. The FPImp syntax is as follows:

⟨*FPImp*⟩ ::= ('`FPImp`' (⟨*identifier*⟩*) ⟨*property*⟩* ⟨*statement*⟩* ⟨*output*⟩)

⟨*output*⟩ ::= ('`output`' ⟨*expression*⟩*)

⟨*expression*⟩ ::= ⟨*constant*⟩ | ⟨*identifier*⟩ | (⟨*operation*⟩ ⟨*expression*⟩*)

⟨*statement*⟩ ::= ['`=`' ⟨*identifier*⟩ ⟨*expression*⟩]
  | ('`if`' ⟨*ifbranch*⟩*)
  | ('`while`' ⟨*expression*⟩ ⟨*statement*⟩*)

⟨*ifbranch*⟩ ::= [⟨*expression*⟩ ⟨*statement*⟩*]
  | ['`else`' ⟨*statement*⟩*]

As in FPCore, each FPImp benchmark includes free parameters and properties; as an imperative language, it includes a list of program statements and multiple return values instead of a single body expression. Assignments and while loops behave in the usual way. Expressions in FPImp are evaluated similarly to FPCore expressions. An `if` statement defines a many-way branch; any `else` branch must be the last branch in its `if`. The `output` statement can return several values and appears at the end of a function.

The FPImp to FPCore compiler translates FPImp functions to FPCore benchmarks while retaining all properties and keeping the same set of free parameters. It inlines assignments, converts the imperative bodies of FPImp loops to the simultaneous-assignment loops of FPCore, replaces many-way `if` statements with nested `if` s, and outputs multiple FPCore benchmarks for FPImp programs with multiple outputs.

3 Accuracy Measurements

To begin to compare floating-point benchmark results across tools, tools must have a common measure of error. However, given the diversity of tools and error measurements that exist in the literature, standardizing on a single measure of error would not only be difficult but could harm the development of some classes of tools. Instead, FPBench standardizes the language for discussing error measurements to facilitate using the FPBench suite to compare floating-point tools.

3.1 Measurement Types

Floating-point error is best analyzed along several axes: scaling vs non-scaling difference, forward vs. backward error, maximum vs. average error, sound vs. statistical techniques, and, for tools which transform programs, how to measure improvement.

Scaling vs non-scaling difference (ε) There are several ways to measure the error incurred by producing the inaccurate value $\hat{x}$ instead of the correct value x. Two common mathematical notions are the absolute and relative error:

$$\varepsilon_{abs}(x, x') = |x - \hat{x}| \quad \text{and} \quad \varepsilon_{rel}(x, x') = \left| \frac{x - \hat{x}}{x} \right|$$

Relative error scales with the quantity being measured, and thus makes sense for measuring both large and small numbers, much like the floating-point format

itself. A notion of error more closely tied to the floating-point format is the Units in the Last Place (ULPs) of error.[2]

$$\varepsilon_{ulps}(x, x') = |\{y \in \mathbb{F} : \min(x, \hat{x}) \leq y \leq \max(x, \hat{x})\}|$$

The floating-point numbers are distributed roughly exponentially, so this measure of error scales in a similar manner to relative error; however, it is better-behaved in the presence of denormal numbers and infinities. Some tools use the logarithm of the ULPs to roughly approximate the number of incorrect low-order bits in $\hat{x}$.

Forward vs. backward error (ϵ) Forward error and backward error are two common measures for the error of a mathematical *computation*. For a correct function f and its approximation $\hat{f}$, forward error measures the difference between outputs for a fixed input, while backward error measures the difference between inputs for a fixed output. Formally,[3]

$$\epsilon_{fwd}(x) = \varepsilon(f(x), \hat{f}(x)) \quad \text{and} \quad \epsilon_{bwd}(x) = \min\left\{\varepsilon(x, x') : x' \in \mathbb{F}^n \wedge \hat{f}(x') = f(x)\right\}.$$

Backward error is important for evaluating the stability of an algorithm, and in scientific applications where multiple sources of error (algorithmic error vs. sensor error) must be compared. However, backward error can be tricky to compute, especially for floating-point computations where there may not be an input that produces the correct output. To our knowledge, all existing tools measure forward error.

Average vs. maximum error (E) Describing the error of a floating-point computation means summarizing its behavior across multiple inputs. Existing tools use either maximum or average error for this task. Formally,[4]

$$E_{\max} = \max\{\epsilon(x) : x \in \mathbb{F}^n\} \quad \text{and} \quad E_{\text{avg}} = \frac{1}{N}\sum_{x \in \mathbb{F}^n} \epsilon(x) \ .$$

Worst-case error tends to be easier to measure soundly, while average error tends to be easier to measure statistically.

Sound vs. statistical techniques It is intractable to evaluate the error of the program on all valid inputs. Existing tools use either static analyses to soundly overapproximate the error or statistical sampling to tightly approximate the error.

Most static techniques are based on interval or affine arithmetic to overapproximate floating-point arithmetic, often using abstract interpretation. Abstract interpretation may be either non-relational [19] or relational abstract

[2] We are using $|S|$ to denote the number of elements in a set S

[3] Where n is the number of arguments.

[4] Where N is the number of valid inputs, and n is the number of arguments.

domains [7,14,1], and may use acceleration techniques (widenings [9]) to over-approximate loops without unrolling them. While such techniques tend to provide loose over-approximations of the floating-point error of programs, they are fast and provide sound error bounds. In some embedded applications, correctness is critical and unsound techniques will not do.

In domains where correctness is not absolutely critical, dynamic sampling techniques can provide tighter approximations of error as it appears in practice. Dynamic error evaluation techniques tend to be slower than static techniques, since they involve running a program multiple times, sometimes with varying semantics.

Measuring improvement (ι) Tools that transform floating point programs, such as those that attempt to improve the accuracy of programs, often compare the accuracy of two floating-point programs instead of measuring the accuracy of a single program. Such comparison uses the improvement in worst-case or average error between the original $\hat{f}$ and improved $\hat{f}'$ implementation of the mathematical function f:

$$\iota_{\text{imp}} = E(\hat{f}) - E(\hat{f}')$$

One cannot, in general, improve the accuracy of a computation simultaneously on all inputs. It is thus often necessary to make a computation less accurate on some points to make it more accurate overall. In this case, it may be useful to report the largest unimprovement, which measures the cost of improving accuracy:

$$\iota_{\text{wrs}}(\hat{f}, \hat{f}') = \max \left\{ \epsilon(f(x), \hat{f}'(x)) - \epsilon(f(x), \hat{f}(x)) \ : \ x \in \mathbb{F}^n \right\}$$

Other measures, such as those describing the trade-off between accuracy and speed, are also interesting, but are less commonly used in the literature and thus not standardized. Similarly, improvement tools could also estimate their effect on numerical stability using automatic differentiation [15] or Lyapunov exponents [21], but we do not know of any such tools.

3.2 Existing Tools

The error measures described can be applied to categorize the error measurements used by existing tools. Table 1 compares Fluctuat [12], FPTaylor [22], Herbie [20], Rosa [11], and Salsa [10].

Fluctuat, FPTaylor, and Rosa all verify error bounds on the accuracy of floating-point computations. Given their need for soundness, it is natural that they use sound error analyses and estimate maximum error. Their use of absolute forward error derives from the difficulty of approximating the other forms of error statically. Herbie and Salsa are tools for improving the accuracy of programs, but differ dramatically in their approach. Salsa uses abstract interpretation to bound maximum absolute error, producing a sound overapproximation of the

Fluctuat	Absolute	Forward	Max	Sound	
FPTaylor	Absolute	Forward	Max	Sound	
Herbie	ULPs	Forward	Average	Statistical	Improvement
Rosa	Absolute	Forward	Max	Sound	
Salsa	Absolute	Forward	Max	Sound	Improvement

Table 1. A comparison of how five floating-point measure error across the axes identified in this section.

maximum error. Herbie, on the other hand, uses random sampling to achieve a tight statistical approximation of ULP error. The tight estimates enabled by statistical techniques provide additional opportunities for Herbie to improve the accuracy of an expression, but prevent it from providing sound error bounds. Unsound tools such as Herbie can be supplemented by using a sound verification tool on its output, a composition enabled by the common FPCore format. Meanwhile, since Fluctuat, FPTaylor, Rosa, and Salsa all soundly measure maximum forward absolute error, their measurements can be compared to determine which technique is best.

4 Tools

FPBench features a collection of compilers and measurement tools that operates in its common format, FPCore. These tools can be a community resource, increasing interoperability as well as code reuse. They also make it easier to write new floating-point analysis and transformation tools by automating what are currently common but tedious tasks.

FPImp to FPCore The FPCore format faithfully preserves important program constructs, such as variable binding and operation ordering, while abstracting away details not relevant to floating-point semantics. However, it is syntactically very different from some of the languages from which benchmarks might originate. To make translation to FPCore from source languages like C, Fortran, and Matlab easier, FPBench provides the FPImp format and a compiler from FPImp to FPCore. FPImp is syntactically similar to imperative languages, in order to make translation of benchmarks as easy as possible.

FPCore to C Since C is a common implementation language for mathematical computations, FPBench provides a FPCore to C compiler. We expect this to be especially useful for tools that improve the accuracy of floating-point expressions and want to return the more accurate expressions to users. What's more, the FPCore to C compiler can also be used to make FPCore benchmarks run in the many available C analysis tools.

Average error estimation Average error is an important metric for floating-point programs, and FPBench provides a tool to statistically approximate it. The

statistical approach is necessary to produce accurate estimates of average error given the current state of the art. The tool allows choosing between absolute, relative, ULP error, and bits of error (computed as the logarithm of ULPs error, before averaging).

We plan to continue developing community tools around the FPBench formats, especially tools for estimating the other measures of error described in Section 3.

5 Benchmark Suite and Examples

The FPBench suite currently includes 44 benchmarks sourced from recent papers on automatic floating-point verification and accuracy improvement. This section first summarizes these benchmarks and then details how representative examples were translated to FPBench from the input formats of various tools in the literature.

The current FPBench suite contains examples from a variety of domains, including 28 from the Herbie test suite [20], 9 from the Salsa test suite [10], 7 from the Rosa test suite [11], and one example from the FPTaylor test suite [22]. These examples range from simple test programs for early tool development up to large examples from industrial control applications for evaluating more mature tools. The larger examples are more challenging, including loops that mutate anywhere from 2 to 13 variables in the loop body. As shown in Table 2 and Table 3, these programs exercise the full range of functionality available in FPBench, and span a variety application areas, from control software to mathematical libraries.

Feature	Benchmarks
Basic Arithmetic	44
Exponentials	13
Trigonometric	10
Comparison	12
Loops	12
Conditionals	3

Table 2. Functions and language features used in the FPBench benchmarks. Benchmarks contain a variety of features, and many benchmarks incorporate several. Exponential functions include logarithms, the exponential function, and the power function.

Domain	Benchmarks
General expressions	31
Math algorithms	6
Embedded systems	4
Scientific computing	3

Table 3. Domains which the FPBench benchmarks are drawn from. Most are general mathematical operations, useful in a variety of domains. The general expressions are the smallest, and are drawn from *Numerical Methods for Scientists and Engineers* [16] and Rosa [11].

5.1 FPTaylor

FPTaylor [22] uses series expansions and interval arithmetic to compute sound error bounds. The authors gave the following simple program as an example input for their tool:

```
1: Variables
2:   float64 x in [1.001, 2.0],
3:   float64 y in [1.001, 2.0];
4: Definitions
5:   t rnd64= x * y;
6: Expressions
7:   r rnd64= (t-1)/(t*t-1);
```

This program is representative of the code necessary to correct sensor data in control software, where the output of the sensor is known to be between 1.001 and 2.0. We manually translated this program to FPCore, yielding:

```
(FPCore (x y)
  :name "FPTaylor example"
  :cite (solovyev-et-al-2015)
  :type binary64
  :pre (and (and (< 1.001 x) (< x 2.0)) (and (< 1.001 y) (< y 2.0)))
  (let ([t (* x y)])
    (/ (- t 1.0)  (- (* t t) 1.0))))
```

The benchmark takes inputs `x` and `y` and uses a `let` statement to represent the intermediate variables from the FPTaylor example. FPCore faithfully preserves important program constructs, such as variable binding and operation ordering, making the translation a simple matter. The benchmark additionally specifies a name and cites its source using a key to a standard BibTeX file. Since the original program uses 64-bit floating-point numbers, the type `binary64` is specified in the benchmark. The constraints on input variables are translated to a single predicate under the `:pre` property.

5.2 Rosa

Rosa [11] soundly verifies error bounds of floating-point programs, including looping control flow through recursive function calls. Several benchmarks in its repository demonstrate this capability, including one that uses Newton's method on a series representation of the sine function. In Rosa's input language the original benchmark is represented as:

```
def newton(x: Real, k: LoopCounter): Real = {
  require(-1.0 < x && x < 1.0)
  if (k < 10) {
    newton(x - (x - (x**3)/6.0 + (x**5)/120.0 + (x**7)/5040.0) /
      (1.0 - (x*x)/2.0 + (x**4)/24.0 + (x**6)/720.0), k++)
  } else {
    x
  }
} ensuring(res => -1.0 < res && res < 1.0)
```

The `require` clause denotes input preconditions, and the `ensures` clause provides the error bound to be verified. We manually translated this program to FPCore, yielding:

```
(FPCore (x0)
  :name "Rosa Example"
  :cite (darulova-kuncak-2014)
  :pre (< (abs x0) 1)
  :rosa-post (< (abs res) 1)
  (while (< i 10)
    ([i 0 (+ i 1)]
     [x x0
      (let ([f (+ (+ (- x (/ (pow x 3) 6))
                     (/ (pow x 5) 120)) (/ (pow x 7) 5040))]
            [df (+ (+ (- 1.0 (/ (* x x) 2))
                      (/ (pow x 4) 24)) (/ (pow x 6) 720))])
        (- x (/ f df)))])
    x))
```

Like the FPTaylor benchmark, this benchmark includes a name, citation for its source, and precondition on inputs. For completeness, we've also included the `requires` annotation, denoted `:rosa-post`, demonstrating the ability to add tool specific annotations by prefixing with the tool name. Note how the `while` construct from FPCore can be used to represent the tail-recursive loop from the Rosa original benchmark.

5.3 Herbie

Herbie [20] heuristically improves the accuracy of straight-line floating-point expressions. The authors demonstrate the improvements Herbie can produce using the quadratic formula for computing the roots of a second degree polynomial. It has uses from calculating trajectories to solving matrix equations. In mathematical notation, the quadratic formula is given by:[5]

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

Herbie produces the following more-accurate variant:

$$\begin{cases} \frac{4ac}{-b+\sqrt{b^2-4ac}}/2a & \textbf{if } b < 0 \\ \left(-b - \sqrt{b^2 - 4ac}\right)\frac{1}{2a} & \textbf{if } 0 \leq b \leq 10^{127} \\ -\frac{b}{a} + \frac{c}{b} & \textbf{if } 10^{127} < b \end{cases}$$

In FPCore format, the original formula is represented as:

```
(FPCore (a b c)
  :name "NMSE p42, positive"
  :cite (hamming-1987)
  :pre (and (>= (sqr b) (* 4 (* a c))) (!= a 0))
  (/ (+ (- b) (sqrt (- (sqr b) (* 4 (* a c))))) (* 2 a)))
```

and the improved version is represented as:

```
(FPCore (a b c)
  :name "NMSE p42, positive"
  :cite (hamming-1987)
```

[5] We use the negative variant here, as in the Herbie paper; the positive variant is analogous.

```
  :pre (and (>= (sqr b) (* 4 (* a c))) (!= a 0))
  (if (< b 0)
      (/ (/ (* 4 (* a c)) (+ (- b) (sqrt (- (sqr b) (* 4 (* a c))))))
         (* 2 a))
      (if (< b 10e127)
          (* (- (- b) (- (sqr b) (* 4 (* a c)))) (/ 1 (* 2 a)))
          (+ (- (/ b a)) (/ c b)))))
```

Like the Rosa and FPTaylor examples, this benchmark gives a name, citation, and precondition. This benchmark uses FPCore's ability to write arbitrary boolean expressions as preconditions, restricting the value under the square root to be non-negative and requiring the denominator be non-zero. Unlike the FPTaylor and Rosa examples, this precondition arises from mathematical considerations, not domain knowledge. The FPCore version of Herbie's output furthermore uses `if` constructs to evaluate different expressions for different inputs, which improves accuracy. Herbie could add additional metadata to the output, such as its internal estimate of accuracy or the number of expressions considered during search, by using prefixed keys like `:herbie-accuracy-estimate`.

5.4 Salsa

Salsa [10] is a tool for soundly improving the worst-case accuracy of programs. The authors evaluate Salsa on a suite of control and numerical algorithms, including the widely used PID controller algorithm. This algorithm is used in aeronautic and avionic systems for which correctness is critical. In C, the benchmark is written as:

```
volatile double p, i, t, d, dt, invdt, m, e, eold, r;
int pid(double m0, double kp, double ki, double kd, double c){
  t     = 0.0;
  invdt = 5.0;
  dt    = 0.2;
  m     = m0 ;
  eold  = 0.0;
  i     = 0.0;
  while (t < 100.0) {
       e = c - m;
       p = kp * e;
       i = i + ki * dt * e;
       d = kd * invdt * (e - eold);
       r = p + i + d;
       m = m + 0.01 * r; /* computing measure: the plant */
       eold = e;
       t = t + dt;
  }
  return m;
}
```

To ease the conversion of this code from C to FPCore, this program was first manually translated to the following FPImp program:

```
(FPImp (m kp ki kd)
 :name "PID"
 :description "Keep a measure at its setpoint using a PID controller."
 :cite (damouche-martel-chapoutot-nsv14)
 :type binary64
 :pre (and (and (< -10.0 m) (< m 10.0)) (and (< -10.0 c) (< c 10.0)))
 [= t 0.0]
 [= dt 0.2]
```

```
[= invdt (/ 1 dt)]
[= c 0.0]
[= eold 0.0]
[= i 0.0]
(while (< t 100.0)
  [= e (- c m)]
  [= p (* kp e)]
  [= i (+ i (* (* ki dt) e))]
  [= d (* (* kd invdt) (- e eold))]
  [= r (+ (+ p i) d)]
  [= m (+ m (* 0.01 r))]
  [= eold e]
  [= t (+ t dt)])
(output m))
```

The FPImp program was then automatically compiled, using the compiler tool in FPBench, to the following FPCore benchmark:

```
(FPCore (m0 kp ki kd c)
  :name "PID"
  :description "Keep a measure at its setpoint using a PID controller."
  :cite (damouche-martel-chapoutot-nsv14)
  :type binary64
  :pre (and (and (< -10.0 m0) (< m0 10.0)) (and (< -10.0 c) (< c 10.0)))
  (while (< t 100.0)
   ([i 0.0 (+ i (* (* ki 0.2) (- c m)))]
    [m m0
     (let ([p (* kp (- c m))]
           [d (* (* kd (/ 1 0.2)) (- (- c m) eold))])
     (+ m (* 0.01 (+ (+ p (+ i (* (* ki 0.2) (- c m)))) d)))))]
    [eold 0.0 (- c m)]
    [t 0.0 (+ t 0.2)])
   m))
```

Beyond the metadata used in the FPTaylor, Rosa, and Herbie examples, this benchmark includes a `:description` tag to describe for readers what the benchmark program computes. These descriptions contain information about the distribution of inputs, the situation in which the benchmark is used, or any other information which might be useful to tool writers. The FPBench suite features descriptions for its most complex benchmarks.

6 Conclusion

The initial work on FPBench provides a foundation for evaluating and comparing floating-point analysis and optimization tools. Already the FPBench format can serve as a common language, allowing these tools to cooperatively analyze and improve floating-point code.

Though FPBench supports the composition of the floating-point tools that exist today, there is still much work to do to support the floating-point research community as it grows. FPBench must be sufficiently expressive for the broad range of applications that represent the future of the community. In the near term, we will add additional metrics for accuracy and performance to the set of evaluators provided by the FPBench tooling and begin developing a standard set of benchmarks around the various measures. We will also expand the set of languages with direct support for compilation to and from the FPBench

format. Longer term, we intend to investigate support for mixed-precision benchmarks, fixed-point computations, and additional data structures such as vectors, records, and matrices.

We hope that FPBench encourages the already strong sense of community and collaboration around floating-point research. Toward that end, we encourage any interested readers (and tool writers) to get involved with development of FPBench by signing up for the mailing list and checking out the project website:

http://fpbench.org/

References

1.
2. DIMACS challenge. Satisfiability. Suggested format., 1993. `http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps`.
3. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. `www.SMT-LIB.org`.
4. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. *Found. Trends Program. Lang.*, 2(2-3):71–190, Dec. 2015.
5. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.
6. C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.
7. A. Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. In *Static Analysis Symposium*, volume 6337 of *LNCS*, pages 184–200. Springer, 2010.
8. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
9. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.
10. N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In M. Núñez and M. Güdemann, editors, *FMICS'15*, volume 9128 of *LNCS*, pages 31–46. Springer, 2015.
11. E. Darulova and V. Kuncak. Sound compilation of reals. In S. Jagannathan and P. Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.
12. E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In F. Logozzo and M. Fähndrich, editors, *SAS'13*, volume 7935 of *LNCS*, pages 1–3. Springer, 2013.

13. E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *European Congress on Embedded Real Time Software, ERTS 2006, Proceedings*, 2006.
14. E. Goubault and S. Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*, pages 232–247. Springer, 2011.
15. A. Griewank and A. Walther. *Evaluating derivatives - principles and techniques of algorithmic differentiation (2. ed.)*. SIAM, 2008.
16. R. Hamming. *Numerical Methods for Scientists and Engineers*. Dover Publications, 2nd edition, 1987.
17. J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.
18. G. J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, Feb. 2014.
19. M. Martel. An overview of semantics for the validation of numerical programs. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2005.
20. J.-R. W. P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In D. Grove and S. Blackburn, editors, *PLDI'15*, pages 1–11. ACM, 2015.
21. S. Sankaranarayanan. Static analysis in the continuously changing world. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 4–5. Springer, 2013.
22. A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. *FM 2015: Formal Methods: 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, chapter Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions, pages 532–550. Springer International Publishing, Cham, 2015.
23. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.