Sinking Point: Dynamic Precision Tracking for Floating-Point

Bill Zorn
University of Washington
billzorn@cs.washington.edu

Dan Grossman
University of Washington
djg@cs.washington.edu

Zach Tatlock
University of Washington
ztatlock@cs.washington.edu

ABSTRACT

We present sinking-point, a floating-point-like number system that tracks precision dynamically though computations. With existing floating-point number systems, such as the venerable IEEE 754 standard, numerical results do not inherently contain any information about their precision or accuracy; to determine if a result is numerically accurate, a separate analysis must be performed. By contrast, sinking-point records the precision of each intermediate value and result computed, so highly imprecise results can be identified immediately. Compared to IEEE 754 floating-point, sinking-point's representation requires only a few additional bits of storage, and computations require only a few additional bitwise operations. Sinking-point is fully generalizable, and can be extended to provide dynamic error tracking for nearly any digital number system, including posits.

CCS CONCEPTS

• **Mathematics of computing** → *Arbitrary-precision arithmetic*; • **Computing methodologies** → Representation of exact numbers.

KEYWORDS

Floating point, numerical analysis

ACM Reference Format:
Bill Zorn, Dan Grossman, and Zach Tatlock. 2019. Sinking Point: Dynamic Precision Tracking for Floating-Point. In *Conference for Next Generation Arithmetic 2019 (CoNGA'19), March 13–14, 2019, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3316279.3316283

1 INTRODUCTION

Floating-point error is particularly insidious because it can easily go unnoticed. Consider the following interaction with Python 3, which uses 64-bit IEEE 754 doubles to represent non-integer numbers:

```
>>> import math
>>> math.pi + 1e16 - 1e16
4.0
```

Clearly, something has gone wrong here. With pen and paper arithmetic, we would expect the large terms of 10^{16} to cancel out, leaving π as the result. Instead, we get 4. What happened?

If we look more closely at the computation, we can see that the first addition must have rounded off the low bits of π. This is entirely understandable: 10^{16} is a big number, so out of the 53 bits of precision available to an IEEE 754 double, there are only two bits left to hold the value of π. Subtracting 10^{16} back off again simply exposes this rounding error.

4 is the correct result: given the available precision, IEEE 754 floating-point has done the best it can. However, the way the result is presented is problematic. IEEE 754 floating-point only has one way to represent 4. Like all other IEEE 754 doubles (besides the subnormals), that representation has exactly 53 bits of precision. All the bits which were rounded off have been filled in with zeros; it would be more precise to write down the result as 4.0000000000000000, though Python avoids printing the additional zeros.

While it is unfortunate to round π so imprecisely that the result is equal to 4, it is not just imprecise but also inaccurate to round π to 4.0000000000000000 with 53 bits of precision. And the IEEE 754 standard provides no indication when this happens. In our simple example, it is easy enough to work through the rounding behavior manually, but for more complex computations, low-precision results can easily cause things 'go off the rails' and transform into catastrophic error without any indication that something is wrong.

To address this problem, we introduce sinking-point. Sinking-point can represent the same set of numerical values as IEEE 754 floating-point, but it allows numbers with different precisions to coexist. If we perform the computation from our example with our prototype sinking-point implementation, we will see the following:

```
>>> Sink(math.pi) + Sink(1e16) - Sink(1e16)
[3.5-5.0]
```

To illustrate the uncertainty of inexact numbers, our implementation prints them as ranges of decimal numbers that are indistinguishable at the represented precision; that is, they would all round to the same number. Here, that represented number is still 4, the same as the IEEE 754 result, but with only two bits of precision, sinking-point makes it clear that we would not be able to distinguish it from any other number between about three and a half and five.

Interestingly, the expected correct result of π is not within the range. This serves to highlight two important properties of sinking-point. First, sinking-point is an approximation, not a sound analysis technique like interval arithmetic. Second, it aims to provide a lower bound on the uncertainty: in this case, we know that we can't distinguish results between 3.5 and 5, but the true range of uncertainty might be larger.

If instead we perform a computation that should actually result in 4, such as

```
>>> Sink(4.0) + Sink(math.pi) - Sink(math.pi)
[3.9999999999999998-4.0000000000000004]
```

then we can see that while adding and subtracting π has caused some rounding and made the result inexact, the interval is much smaller, capturing most of the zeros from the precise IEEE 754 representation.

By tracking precision dynamically though a computation, sinking-point ensures that if a result with some precision is produced, that precision is meaningful; it contains bits that were actually computed rather than filled in with zeros to fit a particular IEEE 754 format.

2 SINKING-POINT BY EXAMPLE

Sinking-point is based upon the following observation: when a floating-point operation causes a loss of precision, that loss of precision is observable. Rather than viewing an operation as something that takes in only values, and produces another value with some fixed, format-dependent precision as a result, sinking-point operations take as input *both values and precisions*, and output *both values and precisions* to which those values have been computed. The key is that, for arithmetic operations and square root, the basic building blocks of floating-point computation, it is always possible to determine the output precision given the precisions and values of the inputs. To give a high-level explanation of how this works, we will examine examples of a few simple computations, paying particular attention to the way the results are rounded.

2.1 Design philosophy

There are several approaches we could follow to determine the output precision of a floating-point computation. One is to provide a sound underapproximation of the true output precision: for each result, assign it some precision which is always known to be less than the true precision of the result. Such an approach would have similar capabilities to interval arithmetic, though it would be restricted to intervals centered around a particular representable digital number. Like interval arithmetic, it would be hindered by a rapid increase in the interval size over the course of long computations.

Instead, sinking-point uses an unsound approximation, not unlike the approximations inherent to IEEE 754 floating-point. Rather than guaranteeing that the actual precision of the result is greater than the assigned precision, sinking-point seeks to ensure that precision is only reduced for good reason: that is to say, if some bit in the representation is cut off due to reduced precision, then there must not have been enough precision available to precisely compute the value of that bit, and similarly if there is definitely not enough information to compute the value of some bit, then the precision must be reduced enough to cut it off.

As it is unsound, this approximation carries certain risks. In particular, it will not be able to detect or protect against gradual error due to accumulated roundoff in the lowest bits; however, unlike interval arithmetic, it does not suffer from rapidly exploding intervals. In most cases, sinking-point is effective at detecting the catastrophic, floating-point-specific precision problems that make the behavior of the IEEE 754 standard puzzling to users used to working with real numbers. By providing an upper bound on the precision, sinking-point can prevent programmers from mistakenly thinking that the guaranteed 53 bits of precision in an IEEE 754 double is the true precision of a computed result.

$$\begin{array}{ccccccccccc}
 & & 1 & 0 & 1 & . & 0 & 1 & ? & ? & ? & ? \\
+ & & 1 & 0 & 0 & . & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 1 & . & 0 & 1 & 0 & 0 & 0 & 1 \\
 & 1 & 0 & 0 & 1 & . & 0 & 1 & & & &
\end{array}$$

Figure 1: Binary visualization of $5.25 + 4.015625$ **computed with sinking-point. Unknown bit are represented as ?s. Four trailing bits (shown in orange) are rounded off due to sinking-point's dynamic reduction of precision.**

2.2 Addition and subtraction

Consider the addition of 5.25 + 4.015625. For simplicity, assume that both numbers are not known exactly: 5.25 has the binary representation `0b101.01`, with the values of the less significant bits all unknown, and 4.015625, or $4\frac{1}{64}$, has the binary representation `0b100.000001`. Assume that we are not limited by a particular representation: we can compute with arbitrary precision, and produce arbitrary precision results, limited only by the precision to which we know the inputs. What is the most precise answer we can give?

Figure 1 shows a visualization of the computation. The inputs are written out in binary, with unknown bits represented as question marks. If we pretend the unknown bits are all zeros, then the arbitrary precision result should have a binary representation of `0b1001.010001`, or 9.265625. However, in reality the unknown bits might not be zero: since we don't know the value of 5.25 precisely, we don't know what they are. An unknown value plus a known value is not equal to the known value; it would be safer to say that the result is also unknown. The most precise answer we can give for certain is `0b1001.01`, or 9.25; this requires rounding off the bits shown in orange in the figure. We want sinking-point to determine that the precision is not high enough to provide these bits, and round them off automatically.

The picture is similar for subtraction. Figure 2 shows a visualization of the same computation, but with the sign of the second operand reversed. The rounding behavior is exactly the same as before, with the orange bits from the figure rounded off due to insufficient precision. Regardless of the sign, adding or subtracting an unknown bit can never produce a precisely known bit as a result. It is interesting to note that the result of the subtraction, `0b1.01` in binary or 1.25, has significantly less precision than either operand, at only 3 bits. Although the same number of bits after the binary point are known, some of the higher bits have canceled out. While IEEE 754 would immediately fill in more low bits with zeros to maintain constant precision, we want sinking-point to recognize that there is no point in doing so because the values of the low bits are not actually zero. Although we don't know what the bits are, we do know that we don't know what they are, and we can communicate this by lowering the precision.

Based on these two examples, we can begin to formulate a rule to determine the output precision that sinking-point should assign for an addition or subtraction. The output precision is not limited purely by the amount of precision the inputs have, but also by where that precision is in the representation. Specifically, the precision will be limited by whichever number has an unknown bit in a more significant place. Visually, this is whichever number has a question

```
    1 0 1 . 0 1 ? ? ? ?
  - 1 0 0 . 0 0 0 0 0 1
  ---------------------
        1 . 0 0 1 1 1 1
        1 . 0 1
```

Figure 2: Binary visualization of $5.25 - 4.015625$ **computed with sinking-point.**

```
       1 0 1 . 0 1
  *    1 0 0 . ? ?
  ---------------------
     1 0 1 0 1 . ? ? ?
       0 0 0 0 . 0 ? ?
         1 0 1 . 0 1 ?
  +        ? ? . ? ? ?
  ---------------------
     1 0 0 1 0 . 0 1
     1 1 1 ? ? .
```

Figure 3: Binary visualization of 5.25×5 **computed with sinking-point. The shift and add algorithm is used to compute the multiplication as addition.**

mark further to the left in its binary representation: at or to the right of the position of this question mark, we can't possibly know any bits of the output precisely.

2.3 Multiplication

Multiplication requires a different precision-tracking scheme from addition or subtraction, but using a shift and add multiplier, or the "grade school" multiplication algorithm, we can relate it back to the rule we observed previously. Figure 3 visualizes the multiplication 5.25×5, again assuming both values are inexact. We sum from the largest values (which have been shifted farthest left) to the smallest. Note that when the shift becomes small enough, we will effectively be multiplying by an unknown bit: to model this, we add a completely unknown value, shifted by the appropriate amount. We can think of this number as a zero with some specific precision: we don't known exactly what value it is, but we have an upper bound on its magnitude. Above a certain significance all the bits in its representation are known to be zero, but below that we have no idea what the bits are.

Like any other addition, we are limited by the most significant unknown bit. Here, that unknown bit comes from the zero, and it restricts the output precision to only three bits. In contrast to addition and subtraction, the location of the bits we know has moved around significantly in the binary representation; in the inputs, we had known bits down to the 2^{-2}s or 2^0s place, but in the output, the least significant known bit is in the 2^2s place. Conveniently, however, we can see that the output precision is equal to the lesser of the two input precisions.

This is not a coincidence. Assume that the second operand in the multiplication has less precision, as in the example; that is to say, we are shifting and masking by the less precise input. Eventually, we will run out of bits and add an imprecise zero. Relative to this zero, the largest term in the addition can have been shifted left by at most the precision of the second operand. There isn't room for more than that precision's worth of bits. Alternatively, if we assume that the first operand has less precision, then we can see that the largest term in the addition (which has the same precision as the first operand) would limit the output precision in the same way.

As we have seen, the precision of a floating-point operation can be limited in two different ways: by the most significant unknown bit in an addition or subtraction, or by the lesser precision of an input to a multiplication. For sinking-point to work, we need to formalize these rules in a way that lets us compute the output precision efficiently, and store the necessary information for that computation compactly.

3 IMPLEMENTATION

To evaluate the capabilities of sinking-point, we built a prototype implementation using Titanic, a multi-precision, multi-format numerical laboratory that makes it easy to experiment with new, floating-point-like digital number systems. Our prototype extends Titanic's built-in support for IEEE 754 floating-point, and for computations with exactly known inputs, it produces exactly the same output values. In order to support dynamic precision tracking, we need to make two changes: first, we need to change the representation of floating-point numbers to store additional information about their precision, and second, we need to change the arithmetic operations to use that information to compute the output precision as well as the output value.

3.1 Sinking-point representation

Since the IEEE 754 standard does not track precision dynamically, we need to add a few extra bits to the representation to store it. A sinking-point number can be thought of as a tuple

$$(v, inexact, p, n)$$

v is a typical IEEE 754 floating-point value. We say that v is the host value, which comes from some host IEEE 754 format that we are extending with sinking-point. *inexact* is a single bit flag that represents whether this number is inexact. We need to keep track of this because exact values should be given special treatment, as we know them to infinite precision.

p and n together represent the precision of the number. p is just the number of bits of precision in the significand, which can range from 0 to p_{max}, where p_{max} is defined as the maximum precision that can be represented by the host IEEE 754 format; for 64-bit IEEE 754 doubles, $p_{max} = 53$. n represents the position of the most significant unknown bit; going back to our visualizations from section 1, it is the position of the leftmost question mark. More formally, for a number with a typical IEEE 754 floating-point exponent equal to e and precision p, we can define $n = e - p$

3.1.1 Representing p and n efficiently. For our prototype implementation, we do not concern ourselves with how the tuple would be packed into a binary representation. However, we can provide a rough upper bound on the maximum number of bits required. In a packed representation, it would make sense not to represent both p and n explicitly, since one could always be computed from the other given access to the exponent of the host value v. In most situations, it would be better to store p, which could be done in at most $\log(p_{max})$ bits, since the value is an integer and ranges between 0 and p_{max}. Assuming one extra bit for the inexact flag, this means the total number of bits required comes to $\log(p_{max}) + 1$ compared to the size of the host IEEE 754 binary representation. For example, using 64-bit IEEE 754 doubles as the host format, sinking-point would require at most 7 additional bits, 6 for the precision and 1 for the inexact flag.

Of course, those 7 extra bits could also be used to increase the precision of the host format, but this would not have any of the benefits of sinking-point's dynamic tracking. The purpose of sinking-point is to increase confidence in precision, not precision itself, and the benefits are independent of the host precision.

3.1.2 Printing sinking-point values. Printing sinking-point numbers in a human readable format presents some unusual challenges. Unlike IEEE 754 floating-point formats, which can only represent a value with one particular precision, a sinking-point format can represent the same value with many different precisions. To distinguish them, our prototype implementation prints inexact values as ranges of decimal numbers. As we saw in section 1, 4 with two bits of precision is displayed as [3.5-5.0], while with 53 bits of precision it is [3.9999999999999998-4.0000000000000004].

The ends of each range are the largest and smallest decimal numbers that would round to the represented number when using IEEE 754 "round to nearest even" rounding semantics at the represented precision. This gives humans a quick underapproximation of the uncertainty in the represented value, while also encoding precision information that can be read back later. By finding the greatest precision such that both ends round to the same value, we can recover both the value and the precision from a decimal range.

Our tool prints the shortest prefix of digits such that both the value and precision can be recovered.

3.1.3 A note about zero. In terms of precision, zero is a special case: by definition, its precision must be zero. For exactly known zeros, the value truly is zero, and the behavior is the same as we would expect from IEEE 754 floating-point. However, for inexact zeros, the most significant unknown bit n for the zero, which is the same as its exponent, becomes important. As discussed in the multiplication example, an inexact zero provides only an upper bound on the magnitude of some value.

Like other inexact sinking-point values, we can display zeros as ranges of numbers that are considered indistinguishable after rounding. Uniquely, zeros have ranges with ends of different signs, essentially representing the negative and positive magnitude of the most significant unknown bit. For example, a zero with a most significand unknown bit of $n = 0$, or equivalently a least significant known bit in the 2^1s place, would be printed out as [-1.-+1.], while a more "precise" zero with $n = -10$ would be printed as [-.0009-+.0009].

This property of zeros is not quite the same as having precision; it would be more accurate to describe it as an exponent. In any case, tracking n for inexact zeros provides important information about the effective precision of computations that produce them as results or intermediate values.

operation	n	p
+	$\max(n_1, n_2, n_{min})$	p_{max}
-	$\max(n_1, n_2, n_{min})$	p_{max}
*	n_{min}	$\min(p_1, p_2, p_{max})$
/	n_{min}	$\min(p_1, p_2, p_{max})$
sqrt	n_{min}	$\min(p_1 + 1, p_{max})$

Table 1: Summary of rules for computing sinking-point output precision

3.2 Sinking-point operations

Sinking-point operations are substantially similar to IEEE 754 floating-point operations. There are two major differences: first, the output precision must be computed, based on the values and precisions of the inputs, and second, the computed output precision affects the way the results are rounded.

For simplicity, we assume the ability to compute all arithmetic operations and square roots to arbitrary precision. In our prototype implementation using Titanic, arbitrary precision arithmetic operations are provided by GNU MPFR. We also assume the existence of a rounding function with the following signature:

$$round(v_{in}, p, n) \rightarrow (v_{out}, inexact_{out}, p_{out}, n_{out})$$

v_{in} is the input value to round, according to some target precision p and least significant bit n. The result is both a rounded value v_{out}, and the corresponding exactness *inexact*, precision p, and most significant unknown bit n. The rounding function assumes its inputs are exact, so it is the case that $(\neg inexact_{out}) \iff v_{in} = v_{out}$.

It is useful to define some precision-related quantities relative to sinking-point's IEEE 754 host format. Specifically, we define p_{max} to be the maximum precision supported by the host format, and n_{min} to be one less than the least significant bit representable in any number in the host format. For IEEE 754 doubles, $p_{max} = 53$, and $n_{min} = -1075$, which in general can be computed as $e_{min} - p_{max}$, where e_{min} is the minimum exponent.

Table 1 gives an overview of the rules for computing sinking-point output precisions. The following sections provide pseudocode for each operation, as well as some additional details.

3.2.1 Addition and subtraction. Sinking-point addition and subtraction effectively share an implementation, described in Python-like pseudocode as:

```
def add((v1, ie1, p1, n1), (v2, ie2, p2, n2)):
  limiting_n = nmin
  if ie1:
    limiting_n = max(limiting_n, n1)
  if ie2:
    limiting_n = max(limiting_n, n2)
  v_out, ie_out, p_out, n_out = \
      round(v1 + v2, pmax, limiting_n)
  return (v_out, ie_out or ie1 or ie2, p_out, n_out)
```

Subtraction is exactly the same, other than flipping the sign of the second argument by passing $v_1 - v_2$ to the rounding function.

Addition and subtraction are limited by n, not p; the limiting value cannot be less than n_{min}, and might be limited further if either of the inputs is not exact. The limiting value of n is determined by taking the maximum. Since n is the most significant unknown bit, larger values of n indicate results that are less precise. Most of the work is done by the addition itself, which our prototype computes to arbitrary precision but in principle could be implemented in much the same way as IEEE 754, and by the rounding function. The final result is inexact either if it became inexact after rounding, or if either of the inputs was inexact.

3.2.2 Multiplication and division. Like addition and subtraction, multiplication and division share what is effectively the same implementation, shown below:

```
def mul((v1, ie1, p1, n1), (v2, ie2, p2, n2)):
  limiting_p = pmax
  if ie1:
    limiting_p = min(limiting_p, p1)
  if ie2:
    limiting_p = min(limiting_p, p2)
  v_out, ie_out, p_out, n_out = \
      round(v1 * v2, limiting_p, nmin)
  return (v_out, ie_out or ie1 or ie2, p_out, n_out)
```

Again, division is the same, other than using arbitrary precision division v_1/v_2 instead of multiplication. Here, the output precision is limited by the precision p of the inputs. The final precision cannot exceed p_{max}, and might be further limited by the precision of either input if it is inexact. The limiting value is computed by taking the minimum. Rounding is exactly the same as for addition and subtraction, with the final result being inexact if either input or the rounded value exhibits inexactness.

3.2.3 Square root. Taking the square root is similar to multiplication in that the output precision is limited by p. However, there are some differences.

```
def sqrt((v1, ie1, p1, n1))
  limiting_p = pmax
  if ie1:
    limiting_p = min(limiting_p, p1 + 1)
  v_out, ie_out, p_out, n_out = \
      round(real_sqrt(v1), limiting_p, nmin)
  return (v_out, ie_out or ie1, p_out, n_out)
```

Since it only takes one argument, there is only one value to limit the precision of a square root operation. The way of computing the limiting precision is also slightly different. The square root is relatively insensitive to errors in the last few bits: multiple nearby floating-point numbers tend to share the same square root at a given precision, even if the last bit is different. Because of this, we can relax the limiting precision slightly by adding one to it, as long as we are not also limited by p_{max}.

3.2.4 Special values. As noted, sinking-point tracks n for inexact zeros. This does not require any modifications to the underlying arithmetic or the host IEEE 754 representation; it will happen automatically as long as the rounding function produces the correct value of n.

Subnormal numbers also do not require any special treatment. They will be handled naturally by the rounding function, as the limit on n_{min} will restrict the precision of values with extremely small magnitudes, even if there seem to be sufficient bits in p_{max}. In a sense, subnormals and sinking-point are closely related; both are floating-point values with decreased precision, but while subnormals occur due to a peculiarity of the format, sinking-point values can only have reduced precision because of suspicious behavior within a computation.

The other special floating-point values, namely infinities and NaN, or not a number, are retained, and their behavior is exactly the same as for the host IEEE 754 format. Any precision information about them is disregarded. Once a computation has gone so off the rails it no longer produces a real value, dynamic precision tracking is not going to help.

4 CASE STUDIES

To illustrate the capabilities of sinking-point, we present two case studies of interesting computations. The first is based on the quadratic formula, and the second is based on a modified version of John Gustafson's "accuracy on a 32-bit budget" challenge.

4.1 The quadratic formula

Beloved of high-school algebra teachers and numerical analysts alike, the quadratic formula gives the solution to the general quadratic equation and computes the roots of a parabola.

Given the general quadratic equation

$$ax^2 + bx + c = 0$$

the formula for the positive root is

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Though simple, the naive form of the computation can be very inaccurate for some inputs when implemented with IEEE 754 floating-point. We can catch these inaccuracies by performing the same computation with sinking-point and checking the output precision.

For purposes of the case study, assume we have some parabola defined with $b = 2$ and $c = 3$. Additionally, we know that a is positive but very small; it is greater than zero, but the magnitude is significantly less than that of b or c. a is the x^2 term of our parabola; near the origin, the smaller a is the more we expect the parabola to look like a line. For the line $bx + c$, there is one zero at $-c/b$, which in our case works out to $-\frac{3}{2}$. Therefore, the smaller a is, the closer we expect the positive zero to be to $-\frac{3}{2}$.

We can plug in various values of a to see what the IEEE 754 standard gives us, and how much precision sinking-point thinks is left. The results are show in table 2, compared to the true answer to 16 decimal places.

At first, down to about $a = 10^{-9}$, IEEE 754 confirms our mathematical intuition. But for smaller values, floating-point inaccuracies start to creep into the computation, making the result increasingly inaccurate until finally collapsing to a catastrophically inaccurate result of zero around $a = 10^{-17}$.

Meanwhile, sinking-point reports ever decreasing precision; while the first result with $a = 0.1$ retains 51 bits of precision, reporting the values of bits down to the 2^{-50}s place, the result at

a	b	c	x (IEEE 754 double)	x (real value)	x (sinking-point)
0.1	2	3	-1.6333997346592444	-1.6333997346592446	-1.633399734659244[0-8]
0.001	2	3	-1.5011266906707066	-1.5011266906707219	-1.501126690670[68-78]
1e-9	2	3	-1.5000000130882540	-1.5000000011250001	-1.[49999995-50000005]
1e-15	2	3	-1.5543122344752189	-1.5000000000000011	-1.[44-56]
1e-16	2	3	-2.2204460492503131	-1.5000000000000002	-[1.8-2.5]
1e-17	2	3	0	-1.5000000000000000	[-1.-+1.]

Table 2: Results for naive quadratic formula with a close to zero. Inaccurate digits in the IEEE 754 result are colored orange. Sinking-point values are represented as a range of values which are indistinguishable at the resulting precision.

a	b	c	x (IEEE 754 double)	x (real value)	x (sinking-point)
0.1	2	3	-1.6333997346592446	-1.6333997346592446	-1.633399734659244[5-7]
0.001	2	3	-1.5011266906707219	-1.5011266906707219	-1.50112669067072[18-20]
1e-9	2	3	-1.5000000011250001	-1.5000000011250001	-1.500000001125000[0-2]
1e-15	2	3	-1.5000000000000013	-1.5000000000000011	-1.500000000000001[3-4]
1e-16	2	3	-1.5000000000000004	-1.5000000000000002	-1.500000000000000[4-5]
1e-17	2	3	-1.5000000000000000	-1.5000000000000000	-1.[4999999999999999-5000000000000001]

Table 3: Results for herbified quadratic formula with a close to zero.

$a = 10^{-16}$ only has two bits of precision. We can also see the utility of tracking the most significant unknown bit of zeros; though it also returns zero for $a = 10^{-17}$, the sinking-point zero has a least significant (known to be zero) bit in the 2^1s place, so it could conceivable be any number between about -1 and 1.

As we can see, the naive form of the quadratic formula is not an accurate way to look for zeros, given what we know about our parabola. Instead, we should be using an alternative formulation, such as the following expression produced by the Herbie tool [8]:

$$x = \frac{1}{\left(\sqrt{b^2 - 4ac} + b\right)\left(\frac{-1}{2c}\right)}$$

Results for this computation are shown in table 3. By restructuring the computation, Herbie has completely avoided the floating-point inaccuracies that plagued the naive version of the formula, producing results that are mostly accurate down to the last few bits. Sinking-point confirms this. Since none of the operations result in loss of precision, sinking-point produces the same values as standard IEEE 754 floating-point, though as they are not exact values, they are shown as very tight decimal ranges.

4.2 Accuracy on a 32-bit budget, adapted

In [4], John Gustafson proposes the following expression for evaluating the accuracy of number systems on a 32-bit budget for precision:

$$\left(\frac{\frac{27}{10} - e}{\pi - (\sqrt{2} + \sqrt{3})}\right)^{67/16}$$

Since sinking-point does not have support for the power function, we cannot use it to evaluate this expression directly. However, we can perform a similar computation:

$$\left(\frac{\frac{27}{10} - e}{\pi - (\sqrt{2} + \sqrt{3})}\right)^{3/2}$$

exponent bits	result	p	bits of accuracy
3	NaN	—	—
4	7.7412[84-91]	20	17.6
5	7.7413[11-25]	19	20.9
6	7.7414[1-3]	18	15.6
7	7.7414[3-8]	17	15.2
8	7.741[64-76]	16	13.8
9	7.740[7-8]	15	13.1
10	7.740[5-9]	14	13.1
11	7.74[37-46]	13	10.9
12	7.74[4-5]	12	10.9
13	7.73[3-6]	11	9.6
14	7.69[2-9]	10	6.9
15	7.7[6-7]	9	7.8
16	7.8[0-2]	8	6.2
17	7.[79-84]	7	6.2
18	[7.94-8.12]	6	4.4
19	7.[13-37]	5	3.4
20	[7.8-8.5]	4	4.4
21	[4.5-5.5]	3	0.7
22	[2.8-3.2]	3	-0.5
23	NaN	—	—

Table 4: Sinking-point result, precision, and bits of accuracy for adapted 32-bit accuracy challenge.

Instead of taking the power directly, we compute the inner expression, multiply it by itself three times, and then take the square root.

The idea of this 32-bit accuracy challenge is to get as close as possible to the true answer while computing with some number system that is only allowed to use 32 bits. For our modified version, the true answer (to 10 decimal places) is 7.7413150952. In order to come up with a "winning" IEEE 754 format, we might want to investigate different ways of partitioning the 32 available bits between the exponent and the significand. To do this, we can sweep across all of the different configurations using sinking-point augmented versions of the corresponding IEEE 754 formats, and compare the precision left in the results.

This might seem like cheating, since a sinking-point augmented format will use more than 32 bits, but we aren't really interested in the accuracy of the sinking-point results. What we want to see is the comparison between sinking-point's assessment of the precision, and the accuracy compared to the true result. We can obtain this by computing the "bits of accuracy" for each of the sinking-point answers. Bits of accuracy, defined for two numbers a and b as

$$-\log_2\left(\left|\log_2\left(\frac{a}{b}\right)\right|\right)$$

is a measure inspired by John Gustafson's similar "decimals of accuracy." [4] For finite a and b with the same sign, the bits of accuracy tells us approximately how many bits in their binary representations are the same. Ideally, we would want every sinking-point result to have p bits of accuracy when compared to its ideal, true value.

The sinking-point results, their precisions, and the corresponding true bits of accuracy for a range of exponent bits are shown in table 4. Sinking-point has a very consistent view of the loss of precision that occurs during the computation: for almost all of the results, the output precision is 8 bits less than the maximum that the format can represent. For the most part, these precisions agree with the true bits of accuracy. However, we are starting to see the limits of sinking-point's capabilities. If we picked the result with the largest sinking-point output precision, with 4 exponent bits and 28 bits of precision, we would actually end up with a worse answer than the winning format with 5 exponent bits and 27 bits of precision. This is likely a fluke due to the peculiarities of rounding for this specific computation, but we can also see that sinking-point systematically overestimates the output precision by about 2-3 bits. Sinking-point is not designed to provide a sound analysis: its goal is to quickly and cheaply detect catastrophic floating-point issues like we saw with the quadratic formula.

5 SINKING-POINT FOR OTHER NUMBER SYSTEMS

As we have described it so far, sinking-point is an extension of a host IEEE 754 format. However, this choice is not due to any particular limitations of sinking-point itself. The precision quantities n and p can be determined for almost any host number system that uses a digital representation of numbers, so in principle, sinking-point could be used to extend any such system. This includes not just IEEE 754 floating-point, but also other similar formats such as posits [5], fixed-point representations, or other application-specific floating-point designs [7]. Similarly, sinking-point is not restricted to host number systems that use constant or finite amounts of precision; our prototype is based on Titanic's arbitrary precision arithmetic libraries, so it can already provide a sinking-point implementation for an IEEE 754 format with arbitrary precision, or mix inputs from formats with different precision.

Sinking-point's ability to track precision would be particularly valuable for multi-precision, multi-format computations, since the meaning of precision remains constant across different formats, even if they use completely different representations under the hood. In a multi-precision, multi-format computation, the precision information sinking-point provides could be used both to search for precision and format parameters that produce high output precision, as we showed with the 32-bit accuracy challenge, or to dynamically adapt when precision becomes too low, for example by redoing part of a computation with a different format.

6 RELATED WORK

Floating-point analysis is a rich area, spread somewhere between Programming Languages and Formal Methods. Many tools have been developed to analyze floating-point programs statically, such as Fluctuat [3], Rosa [2], and FPTaylor [10]. These tools can provide tight and sound worst-case error bounds for small computations, but they usually do not scale well to larger programs. The Herbie [8] tool takes a slightly different approach, using a stochastic static analysis to rewrite floating-point programs into forms that suffer less from pathological floating-point error, but it does not provide formal guarantees about its output. Other tools, such as Herbgrind [9] and FPDebug [1] perform dynamic analyses similar to sinking-point's precision tracking, though they depend on storing higher-precision "shadow values" and thus are much more computationally expensive. In contrast, sinking-point is a combination of a dynamic analysis and a floating-point standard, such as IEEE 754 [6] or posits [5].

7 CONCLUSION

We have presented sinking-point, a floating-point arithmetic that dynamically tracks precision through computations in order to provide better transparency about the precision of results. While typical IEEE 754 floating-point hides the fact that some results may have been computed with less precision than others, sinking-point allows numbers of many different precisions to coexist within the same number system. Though it is an approximation like IEEE 754 floating-point and cannot provide sound worst-case guarantees about precision like an interval analysis, sinking-point has a compact representation, requiring at most $\log p_{max} + 1$ extra bits compared to a host IEEE 754 floating-point format, and its precision analysis is inexpensive to compute. With the recent explosion of interest in new number systems, sinking-point provides a way to tie them all together with a shared precision analysis, encouraging rapid exploration in spaces where traditional floating-point analysis techniques have not been fully developed.

ACKNOWLEDGMENTS

This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A Dynamic Program Analysis to Find Floating-point Accuracy Problems *(PLDI '12)*. ACM, New York, NY, USA, 453–462. http://doi.acm.org/10.1145/2254064.2254118

[2] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals *(POPL '14)*. ACM, New York, NY, USA, 235–248. http://doi.acm.org/10.1145/2535838.2535874

[3] Eric Goubault and Sylvie Putot. 2011. Static Analysis of Finite Precision Computations *(VMCAI'11)*. Springer-Verlag, Berlin, Heidelberg, 232–247. http://dl.acm.org/citation.cfm?id=1946284.1946301

[4] John Gustafson. 2018. Notebook on Posits. https://posithub.org/docs/Posits4.pdf Accessed 14 December 2018.

[5] John Gustafson and Isaac Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations* 4, 2 (2017). http://superfri.org/superfri/article/view/137

[6] IEEE. 2008. IEEE Standard for Binary Floating-Point Arithmetic. *IEEE Std. 754-2008* (2008).

[7] Jeff Johnson. 2018. Rethinking floating point for deep learning. *CoRR* abs/1811.01721 (2018). arXiv:1811.01721 http://arxiv.org/abs/1811.01721

[8] 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 (PLDI'15)*. ACM.

[9] 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 (PLDI 2018)*. 256–269. https://doi.org/10.1145/3192366.3192411

[10] Alexey Solovyev, Charlie Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2015. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions *(FM'15)*. Springer.