©Copyright 2021
Bill Zorn

Rounding

Bill Zorn

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:
Dan Grossman, Chair
Zach Tatlock, Chair
Daniel Shapero

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Rounding

Bill Zorn

Co-Chairs of the Supervisory Committee:
Professor Dan Grossman
Paul G. Allen School of Computer Science & Engineering

Associate Professor Zach Tatlock
Paul G. Allen School of Computer Science & Engineering

Computer number systems are one of the most fundamental interfaces between software and hardware, but despite recent interest they are rarely studied. We present a suite of tools and techniques to make it easier for both application-level software developers and hardware architects to study number systems and design new ones. Our key theoretical contribution is the use of rounding as an abstraction to describe the behavior of a wide variety of number systems in terms of real arithmetic. By leveraging this abstraction, we can build tools that simulate the behavior of many different number systems, efficiently track error through large computations, and automatically search for number system configurations that are optimized for a particular application.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables v

Chapter 1: Introduction 1
1.1 Computer number systems 2
1.2 Thesis 8
1.3 Outline and relationship with prior work 8
1.4 Why not numerical analysis? 9
1.5 What about error? 10

Chapter 2: Background and related work 12
2.1 Applied math and numerical analysis 12
2.2 General-purpose number systems 14
2.3 Emerging number systems for emerging applications 19
2.4 Mixed precision tuning 22

Chapter 3: FPCore: specifying number systems with real numbers 24
3.1 Concrete bit-level semantics in IEEE 754 24
3.2 An abstract data type for number systems 27
3.3 Computation is rounding 33
3.4 FPCore 1.0: a language for numerical programs 39
3.5 FPCore 1.1: a language for number systems 42
3.6 FPCore 2.0 and beyond 48

Chapter 4: Titanic: implementing rounded reals without infinite precision 50
4.1 Arbitrary precision computation 51

4.2 Representing rounding . . . 53
4.3 Using Titanic . . . 60
4.4 Future work . . . 63

Chapter 5: Sinking-point: a number system with safer rounding . . . 65
5.1 The hidden perils of rounding . . . 65
5.2 Sinking-point . . . 67
5.3 Implementing sinking-point . . . 72
5.4 Case studies . . . 79
5.5 Sinking-point for other number systems . . . 84
5.6 Future work . . . 85

Chapter 6: QuantiFind: exploring application-specific number systems . . . 87
6.1 Representing configurations . . . 89
6.2 Evaluating configurations . . . 89
6.3 Searching the configuration space . . . 92
6.4 Results - RK4 . . . 94
6.5 Results - blur . . . 99
6.6 Conclusion and future work . . . 105

Chapter 7: Conclusion . . . 107

Bibliography . . . 109

LIST OF FIGURES

Figure Number — Page

1.1 Lorenz system from point $(-12, \frac{-17}{2}, 35)$, evolving for $t = \frac{15}{4}$ 4
1.2 Lorenz system solved with Runge-Kutta 4^{th} order method 5
1.3 Lorenz system, RK4, fixed-point number systems 6
1.4 Number system limitations for Lorenz system, RK4 with fixed-point 7

2.1 Floating-point precision selector of IBM Model 44 mainframe 15
2.2 Accuracy of floats and posits over the dynamic range 18

3.1 5-bit floating-point number system, with implementation of square root . . . 25
3.2 Anatomy of a floating-point value 26
3.3 5-bit floating-point number system representation 28
3.4 Ideal floating-point number system with 2 bits of precision 29
3.5 Finite floating-point number system with 2 bits of precision 30
3.6 Ideal fixed-point number system with a largest unrepresentable binary digit $n = -4$ 32
3.7 $\sqrt{2}$ shown on a number line with 5-bit floating-point format 34
3.8 Pseudocode implementation of the rounding monad 38
3.9 Example FPCore program (left), and mathematical notation (right) 39
3.10 The same program, after numerical improvement by Herbie 39
3.11 Lorenz system, FPCore and mathematical notation 47
3.12 4th order Runge-Kutta method, FPCore and mathematical notation 47
3.13 RK4 driver FPCore 47

4.1 Rounding envelopes for example single and double precision values near 1 . . 54

5.1 Binary visualization of $5.25 + 4.015625$ computed with sinking-point 69
5.2 Binary visualization of $5.25 - 4.015625$ computed with sinking-point 70
5.3 Binary visualization of 5.25×5 computed with sinking-point 72

6.1 Selected configurations for RK4 solver, with bitcost and accuracy 91

6.2 Pareto frontier for RK4 experiment 95
6.3 Size of Pareto frontier during search for RK4 96
6.4 Test image used for blur 99
6.5 Box blur kernel in FPCore, configuration E 100
6.6 Pareto frontier for blur experiment 101
6.7 Enlarged segment of Pareto frontier for blur 102
6.8 Size of Pareto frontier during search for blur 104
6.9 Example outputs for blur 105

LIST OF TABLES

Table Number Page

5.1 Summary of rules for computing sinking-point output precision 76
5.2 Results for naive quadratic formula with a close to zero 80
5.3 Results for herbified quadratic formula with a close to zero. 80
5.4 Sinking-point result, precision, and bits of accuracy for adapted 32-bit accuracy challenge. 83
6.1 Selected configurations for RK4 94
6.2 Selected configurations for blur 103

ACKNOWLEDGMENTS

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

DEDICATION

To my dad, who likes to tell me, "it's not the projects you start that count; it's the ones you finish." He might be right about that.

Chapter 1

INTRODUCTION

What's in a number, anyway?

Counting gives us the natural numbers, often written $\mathbb{N}$, from one until you run out of fingers. We might sometimes want to count nothing, so we can add zero to get the whole numbers. We might also want to count the absence of things, so we can add negative numbers to get the integers $\mathbb{Z}$, and close the group under addition.

This is good for counting entire things. However, what happens if we have more than nothing to count, but less than a single something? Putting two integers together in a ratio gives us the rational numbers $\mathbb{Q}$. Ratios are powerful—we can not only count big things, but subdivide in order to measure small things.

Some small things, anyway. What about those things that can't be represented as ratios of integers, say the diameter of a circle compared to its circumference, or the number that, when multiplied by itself, gives exactly two?

These numbers are members of the reals, $\mathbb{R}$. If you imagine an ideal, infinite line, centered at zero, real numbers can represent any distance along that line, without respect for physical concerns like quanta or mathematical ones like rationality. Curiously, there are more reals than naturals [7]. Just by counting more, we can assign a unique natural number to any integer or even any ratio, but not to every real. Not even to every real between zero and one.

We could go further, for example by constructing a perpendicular line of imaginary numbers to deal with the square root of -1 and creating the whole complex plane in the process. In some cases, we might also want to reason about things that are almost but not

quite numbers, such as infinities or infinitesimally small quantities that have a sign but not finite magnitude. But, for most things we would want to represent, real numbers in some quantity are good enough.

Numbers are nice to have, but what we really want to do is perform computations with them. Unfortunately, math is hard. Computers are a big help in this regard: they can do billions of operations each second, and they hardly ever make mistakes unless we specifically tell them to.

But there is a fundamental problem: computers are finite. They cannot directly represent, or perform computations with, continuous real values.

What to do?

1.1 Computer number systems

As the name would imply, a *computer number system* is a system for representing numbers on a computer. A computer number system has two key parts or capabilities: it must have some way to represent numbers, and also some way to perform computations with those numbers. Usually, computation is the more interesting part, but in this presentation we will show that rounding can serve as an abstraction to describe both capabilities, allowing us to turn the usual paradigm on its head and focus on the simpler problem of representation.

The most elemental number representation is binary counting. A sequence of n bits can be in 2^n distinguishable states; if we treat the bits as digits in a base-2 positional notation, we can efficiently map them to whole numbers between 0 and $2^n - 1$. The more bits we have, the larger the numbers we can represent, and priority is given to smaller numbers first so that no gaps arise when counting.

Negative numbers can be represented easily enough with a separate sign bit, or with tricks such as a two's complement representation where the most significant place represents a negative magnitude rather than a positive one.

But what about the reals? Some of them can be represented as rational numbers, using pairs of integers in the natural way. Alternatively, we could decide in advance on a fixed scale

factor for all the numbers in the system. This avoids having to store two separate integers for each number, and is often referred to as a fixed-point representation.

Effectively, the scale factor (fixed or not) lets us control the spacing between real numbers that are included in the representation. If we make it small enough, then for any real number, we should be able to find some represented number that is very close to it.

To compute over the number representations and produce a full number system, we would usually first break computations up into atomic mathematical operations like addition or exponentiation. We could then perform real arithmetic on the represented numbers (which are, after all, real numbers) and find some representable number that's close to the result, for each operation; alternatively, we could build a mapping from representable numbers to representable numbers directly. Either way, the limited number representation means that the results will differ from the behavior of true real arithmetic, both at an operation level and for the entire computation.

Let's see how this works in an example.

1.1.1 Illustrated running example: the Lorenz system

The Lorenz system [30] is a set of 3-dimensional ordinary differential equations that arise as a model of atmospheric convection:

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x)\\ \frac{dy}{dt} &= x(\rho - x) - y\\ \frac{dz}{dt} &= xy - \beta z\end{aligned}$$

with $\sigma = 10$, $\beta = 8/3$, and $\rho = 28$. These equations exhibit chaotic behavior, and are often used as an example in numerical analysis due to their sensitivity to small errors in the calculation.

There is no closed form solution to the system, but if we pick an arbitrary initial point

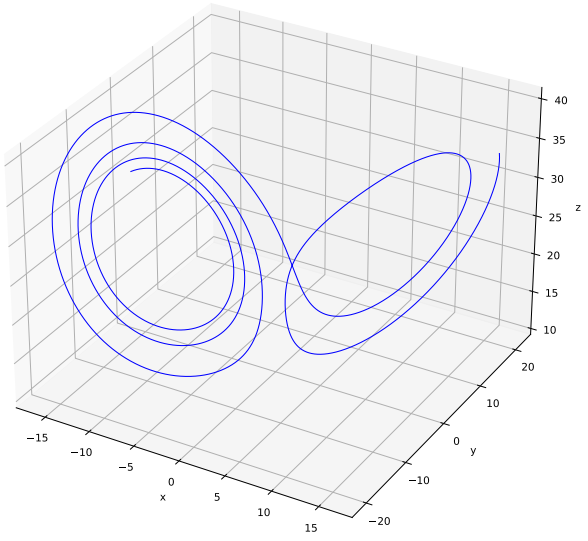
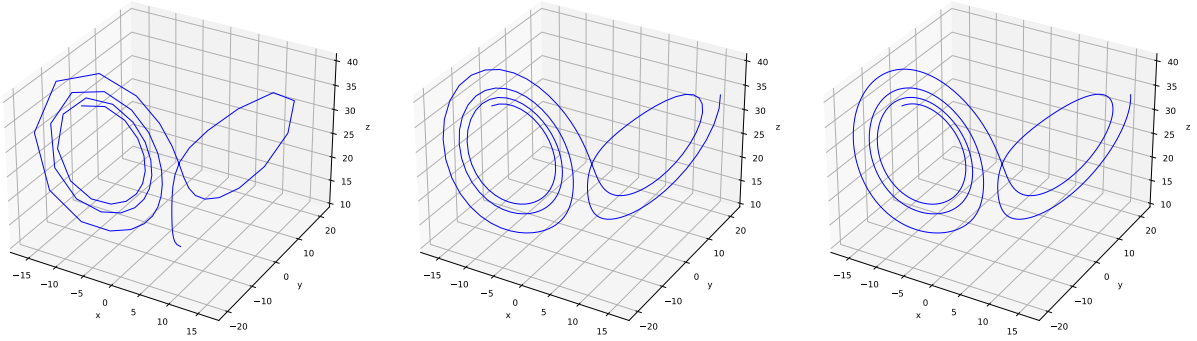


Figure 1.1: Lorenz system from point $(-12, \frac{-17}{2}, 35)$, evolving for $t = \frac{15}{4}$

at $(-12, \frac{-17}{2}, 35)$, we can follow the derivatives in time (for a total evolution over $t = \frac{15}{4}$) to produce the plot in Figure 1.1. In the plot, we can already see the two butterfly wings of the so-called "double-scroll attractor" starting to form. The initial point is at the top of the left scroll; the solution goes around this scroll three times before swinging over to the right and looping around the other scroll about one and a half times.

The plot doesn't actually show real numbers; like everything else in this dissertation, it was rendered using a computer. However, any error in the plotted values is much less than the uncertainty from rendering it on a screen with pixels, or from printing it out, so for all practical purposes (or if we squint at it enough) we can assume it faithfully represents real number behavior.

In practice, particularly for more complex applications, we would approximate the solution using an iterative algorithm such as a Runge-Kutta method [28]. Figure 1.2 shows the approximations obtained with various step sizes. With too large a step size (and thus too coarse an approximation), we can see that the plot is visibly jerky, and the double scroll



Step size $\frac{1}{20}$ Step size $\frac{1}{64}$ Step size $\frac{1}{256}$

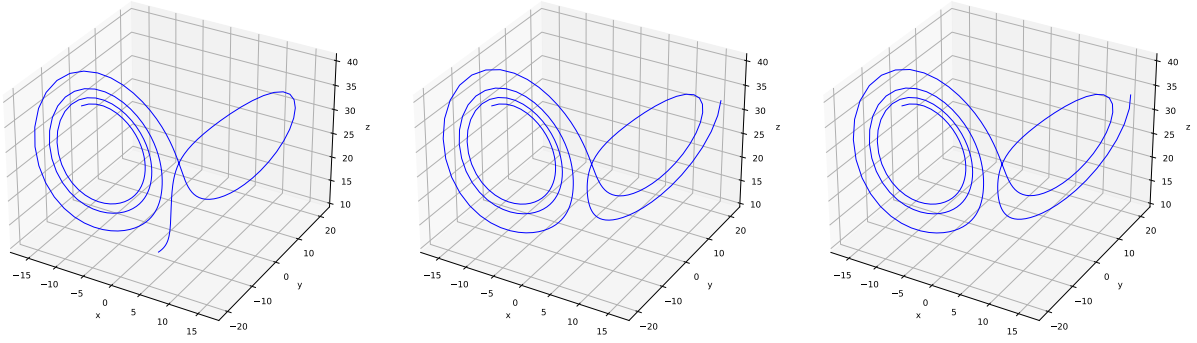
Figure 1.2: Lorenz system solved with Runge-Kutta 4^{th} order method

attractor does not properly form. With a step size of $\frac{1}{64}$, the line is not as smooth as the real solution, but at least on this timescale it faithfully reproduces the scrolls. Of course, this assumes we are working with real numbers.

A practical simulation of the system would use both an approximate, iterative algorithm, and a computer number system with limited precision. Figure 1.3 shows the approximate solutions using the RK4 method with a step size of $\frac{1}{64}$ and a variety of different fixed-point systems, all with 32 integer bits (this is enough to represent magnitudes up to about two billion, more than enough for the equations) and between 10 and 32 fractional bits.

With only 10 fractional bits, the difference between adjacent numbers in the system (the resolution, if you will) is about $\frac{1}{1024}$; we can see that this fails to reproduce the scrolls. 14 fractional bits is just barely enough. Looking closely, the ending point of the plot is a bit lower when computed with 14 fractional bits as opposed to 32, which is completely indistinguishable from using real numbers.

There are two different sources of error in our approximation. We need to have a sufficiently accurate algorithm, with a small enough step size to capture the behavior; we also need to have a sufficiently precise number system so that the algorithm behaves as it would with real numbers. These sources of error are interdependent. With real numbers, we would



10 fractional bits, step size $\frac{1}{64}$

14 fractional bits, step size $\frac{1}{64}$

32 fractional bits, step size $\frac{1}{64}$

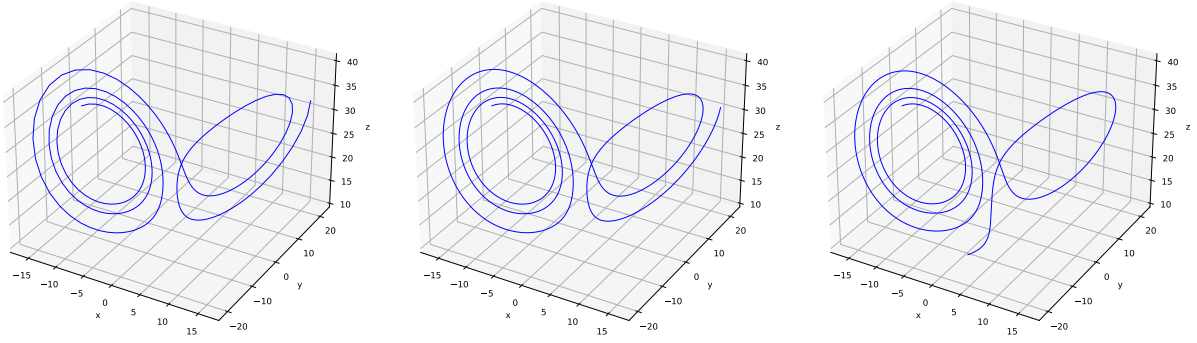
Figure 1.3: Lorenz system, RK4, fixed-point number systems

expect that a more accurate algorithm (here, using a smaller step size for RK) should produce more accurate results. However, if we fix our barely sufficient number system with 14 fractional bits, then we can see in Figure 1.4 that using a smaller step size of $\frac{1}{256}$ actually makes the result less accurate. If we make the step size much smaller, for example $\frac{1}{2048}$ as shown, then the double scroll completely fails to form, even though the plot looks smooth.

1.1.2 Number systems in practice

Dealing with error in numerical computations is not a new challenge. Whenever we model the behavior of a continuous function over real numbers with a finite system like a computer, both algorithmic and numerical error will arise. The interdependence between them is particularly insidious; when something goes wrong, it can be unclear whether it is a problem with the algorithm, a limitation of the number system, or simply a bug in the code.

Historically, the solution has been to standardize on a single, "good enough" number system, and painstakingly modify the algorithms until they work within the constraints of that number system. The chosen number system is floating-point with a particular number



14 fractional bits, step size $\frac{1}{64}$ | 14 fractional bits, step size $\frac{1}{256}$ | 14 fractional bits, step size $\frac{1}{2048}$

Figure 1.4: Number system limitations for Lorenz system, RK4 with fixed-point

of bits, as enshrined in the IEEE 754 standard. Floating-point offers a better dynamic range than the fixed-point systems shown in the example, but it still exhibits exactly the same kinds of problems with numerical error, and the additional complexity of the format and many edge cases can lead to other problems.

This approach has led to huge developments in numerical computing over the past 36 years since the IEEE 754 standard was first adopted. By fixing the behavior of the number system, IEEE 754 makes it possible to write software with reproducible behavior across multiple hardware platforms, effectively separating the development of software and hardware and allowing both to progress independently.

However, standardizing a particular number system also makes that system into a limitation. IEEE 754 floating-point is not suitable for all applications or hardware designs. We propose a more general abstraction for reasoning about the behavior of *many* number systems, which can still preserve the necessary separation between number system implementations and numerical applications, enabling independent development without being limited by a single number system.

1.2 Thesis

This research rests on two theses:

1. Rounding behavior provides a comprehensive, mathematical way to understand the semantics of number systems in software.

2. This perspective enables automated approaches to find application- and algorithm-specific number systems that have significant performance advantages.

1.3 Outline and relationship with prior work

Chapter 2 discusses related work.

Chapter 3 presents our semantics for number systems in FPCore. FPCore [13] is an existing programming language that is part of the FPBench project [18]. Compared to previous publications and the official documentation of the standard on the website, this presentation will delve more deeply into the theoretical underpinnings of the abstract number system representation, and focus less on more typical programming language features.

Chapter 4 presents Titanic, a foundational tool which we use to implement FPCore's semantics for a variety of different number systems. Titanic has been used behind the scenes for several publications, such as [48] and [50], but this is the first detailed description of its design and implementation.

Chapter 5 describes Sinking-point, a number system that uses rounding behavior to detect some forms of numerical error automatically. Sinking-point was previously published in [50], The presentation here omits some of the definitions of rounding behavior, as they are covered by other chapters.

Finally, Chapter 6 presents QuantiFind, a new design space exploration tool that automatically explores the behavior of many different number systems to find configurations with particularly good combinations of performance (or cost) and accuracy for a given application.

We devote the rest of the introduction to a discussion of some expectations and intentions for this work.

1.4 Why not numerical analysis?

Numerical analysis is, broadly speaking, the art and science of getting continuous, real-valued algorithms to work correctly when implemented with finite precision number systems. This is a huge and very important field, with research going back for centuries to develop techniques that are robust to the oddities of floating-point. It is only because of numerical analysis that the IEEE 754 float-point standard has achieved any measure of success; while the standard ensures that results are reproducible, numerical analysis is required to show they are correct.

We think of our work as adjacent to numerical analysis. Our goal is not to produce traditional numerical analysis results, nor to advance the state of the art in numerical analysis. The techniques and results presented here should not be seen as a replacement for numerical analysis, but rather as complementary to it.

Much of traditional numerical analysis concerns two archetypal questions:

1. Given some finite-precision algorithm, what is the worst case error (statically) that it can produce, compared to the expected real-valued result?

2. Given some real-valued algorithm, what is the best finite-precision algorithm to compute the same result as cheaply and accurately as possible?

We will probably see new research that addresses these questions for as long as we have computer science because they are so difficult to answer in a general and scalable way. Our research does not address these questions directly. Instead, we present a new conceptual framework which makes it easier to ask and study other questions, such as:

1. Given some finite-precision computational trace, about how much error (dynamically) has occurred so far?

2. Given some real-valued algorithm, how much numerical error from the number system can it tolerate before it breaks down?

Answering these questions, and phrasing others, does not solve the fundamental problems that numerical analysis addresses, but if we can do it in a practical and scalable way, we can lighten the load and make working with finite precision numbers easier for everyone.

1.5 What about error?

Error is a natural, inevitable effect in any system that seeks to model continuous behavior with something that is finite. That is to say, error is the consequence of rounding.

There are multiple ways of measuring error. Say that x is some ideal real number, and $\hat{x}$ is the concrete, nearest representable number in some number system. The absolute error ϵ is the difference: $\epsilon = |x - \hat{x}|$. In its purest form, absent the context of x and $\hat{x}$, ϵ is just some other real number. Often it is more useful to know how the error compares to x, for example by taking the relative error $\frac{\epsilon}{x}$, than it is to study ϵ directly.

Typically, error is usually thought of in terms of x. The scaling term in the relative error is the true real value x—not $\hat{x}$. This is to say, it doesn't matter what representable number we actually got back; we only care about the real result we wanted, and how close to it we came. The particular properties of the number system are irrelevant, except to determine how big ϵ is.

This is certainly a useful viewpoint, particularly for translating real analysis results into a finite precision setting. If a property holds for all real numbers, then we *shouldn't* care which particular ones are representable in a number system. If we can bound our error, it doesn't matter which or how many representable numbers fall within the bound. That they are all within the bound is enough.

However, there are limitations to this viewpoint. ϵ, like x, might be any real number, so it might be hard to represent or reason about. We also need the real value of x to obtain it. These aren't significant issues for pen and paper proofs, but for automated tools they

can pose a problem. Some bounds that are observably true for particular number systems might not be provable with real arithmetic, because they simply are not true in general: they only hold because of special number system properties, i.e. for the finite number of representable inputs. These kinds of bounds are important for building and verifying optimal, high-performance hardware designs.

We adopt a slightly different view of error. Rather than real number-centric, traditional "$x + \epsilon$" error, we sometimes think in terms of "$\hat{x} + \epsilon$" error. In this viewpoint, it's not the real answer that matters, it's the representable number you actually got, and the distance that the true (possibly unknowable) real answer must be within. We will refer to this as a *rounding envelope*: the interval of real numbers around a representable number that all round to that number.

Obviously rounding envelopes are no good for proving the correctness of an implementation of an atomic square root or logarithm operation. But they can be quite useful when reasoning compositionally about the behavior of individually bounded operations, particularly because we can use their discrete nature to avoid some of the pitfalls of dealing with arbitrary real numbers.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Applied math and numerical analysis

At the broadest possible level, this work falls under the umbrella of applied math. Many algorithms with important real-world or economic applications today (weather modeling, deep learning, etc.) are based on mathematical reasoning in terms of real numbers, and ultimately on the ability of computers to do some approximation of this math quickly. Number systems are the key interfaces between numerical applications and physical silicon that make these applications possible; we can think of applied math as the view of number systems from above, in software.

Roughly speaking, numerical methods is the subset of applied math that deals with building finite approximations of continuous algorithms that are suitable for running on computers, and numerical analysis studies ways to ensure these approximations are correct. Work in this space goes back millennia, if the name of the "Babylonian method" for approximating the square root is any indication.

One frequent thread in numerical analysis is to make the error go away by reproducing the behavior of real number exactly. This can be done in a variety of ways. Constructive reals can track the behavior of a computation symbolically and produce an arbitrary number of correct digits on demand [5]; implementations have been built for various programming languages [6], [31]. Alternatively, systems like the Daisy framework [14] statically bound the numerical error so that the entire computation can be trusted as a reflection of real valued behavior. Both of these approaches face significant challenges with scale—for large computations, it is difficult to compute large numbers of digits or prove a static error bound.

2.1.1 Dynamic precision tracking

Another subfield within numerical analysis with particular relevance for this research is dynamic precision tracking. A standard approach to identify error in a particular computational trace is to keep around higher-precision shadow values for each intermediate result that is computed; comparing these shadow values to the actual results computed with the number system can expose divergences from the behavior expected with real arithmetic.

A variety of tools perform some sort of shadow analysis. For example, Herbgrind [43] uses shadow value computation to localize error and discover small program fragments that are the root causes of numerical problems in a computation.

Shadow value execution is computationally expensive, as it requires not just instrumenting the computation but computing with greatly more precision than the original program, so some techniques have been developed to reduce this cost. The work in [10] makes the shadow execution tasks parallel, so that the instrumented computation can be spread over multiple processors even when the main application is single-threaded. The Shaman library [15] takes a different approach to shadow execution, using another value of the same type as the main computation to store an error term, in a scheme similar to double-double computation.

2.1.2 Automated tools for numerical analysis

Another line of work of particular interest deals with *automating* numerical analysis. The Herbie tool [33] automatically rewrites numerical programs to improve their accuracy; it is assumed that this is done in the presence of, and with respect to, some particular number system, which is always IEEE 754 floating-point. Pareto Herbie [42] further develops the tool to handle multiple number systems, and to do precision tuning at the same time as rewriting, in a manner somewhat reminiscent of QuantiFind.

2.2 General-purpose number systems

We devote the rest of this section to related work on number systems themselves, and how they are implemented in computers; we can think of this as the view of number systems from below, in hardware.

A considerable amount of effort has been invested into designing and using number systems, some of it published as research, other pieces hidden inside standards and applications. We will break our discussion of this work into three rough categories. First, we will discuss some historical work on universal number systems like IEEE 754. Second, we will review more recent developments using specialized number systems to accelerate particular applications. Last, we will cover existing techniques for optimizing applications against number systems, such as mixed precision tuning.

2.2.1 Floating-point as a universal number system

The use of floating-point as a universal computer number system significantly predates the IEEE 754 standard, going back as far as the work of Leonardo Torres y Quevedo, who designed an electro-mechanical version of Charles Babbage's analytical engine in 1914 which used floating-point arithmetic [37]. Konrad Zuse also used floating-point in his Z machines [39], even proposing features like careful round and the use of infinities and NaNs, 40 years before the adoption of the IEEE 754 standard.

Floating-point offers significant advantages over fixed-point in terms of the dynamic range and accuracy that can be provided by a representation with a given number of bits. By separating the representation's bits into two fixed-point values, usually called the significand and the exponent, floating-point expands the dynamic range for n bits from 2^n for a pure fixed-point representation, to $2^{2^{cn}}$ assuming a constant fraction c of the bits are used for the exponent. This is accomplished by representing values as the product $2^{value(exponent)}value(significand)$, where $value(x)$ is the real number represented by a bitvector x in a typical fixed-point system. The precision of this representation, which is approxi-

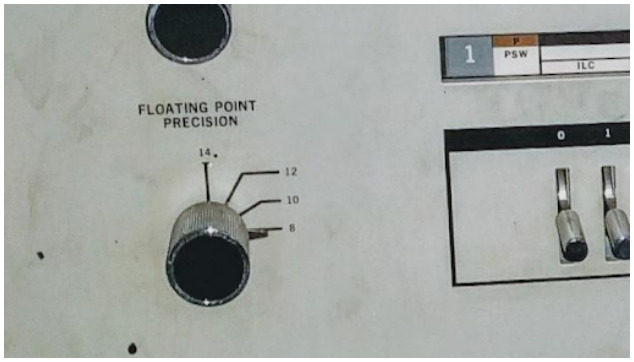


Figure 2.1: Floating-point precision selector of IBM Model 44 mainframe [45]

mately equal to the number of bits in the significand, is mostly constant across the dynamic range.

For a given number of bits, the dynamic range of a floating-point number system can be exponentially larger than that of a fixed-point system. This makes floating-point representations with a relatively small number of bits widely useful for a large variety of scientific calculations, in a way that fixed-point cannot match. Most physical constants and other numbers that appear frequently in calculations have exponents of reasonable size. Floating-point systems thus hit a "sweet spot" in the design space of number systems, where a reasonable, constant number of bits, few enough to fit in a machine register, can represent most values expected to occur in computations. Furthermore, basic arithmetic operations can still be implemented efficiently with binary integer adders and multipliers, though some additional logic is needed to normalize and maintain the exponents.

2.2.2 The IEEE 754 standard

In the decades leading up to the 1980s, a wide variety of floating-point systems were implemented in a diverse set of computers. Figure 2 shows the floating-point precision selector of an IBM System/360 Model 44 mainframe, a physical knob that let the operator select from 4 significand sizes (in multiples of 4 bits) to use during arithmetic operations [45]. While no

doubt useful to an expert with a deep understanding of both the hardware and the software running on the machine, such a knob must have been intimidating for application developers with less hardware expertise.

The contribution of the IEEE 754 floating-point standard was not to introduce floating-point as a number system, but rather to provide unity and standardization in a fragmented environment of existing floating-point systems. Pre-IEEE 754 floating-point systems were essentially designed for application domains, specifically the domains of applications that would run on particular computers. IEEE 754 embodied the idea that such specialization was unnecessary—that a single, universal system would be sufficient for all domains simultaneously.

IEEE 754 serves two major purposes. First, it standardizes a single binary representation as a common data exchange format, so that numerical values can be communicated between applications and computers without having to go through an intermediate representation, like decimal strings, which could be both costly and error prone. More importantly, the IEEE 754 standard establishes guidelines about the accuracy of arithmetic operations. These guidelines are strong enough for the most basic operations (addition, multiplication, division, and taking square roots) to make the behavior of many programs reproducible across different computers implementing the standard, and enable development of numerical applications and algorithms independently of the hardware that runs them.

The IEEE 754 standard has endured, essentially unchanged, since its inception in 1985, and is still by far the most widely supported (and often only) number system implemented in computer hardware. It is a proof by existence that a sufficiently good universal number system is possible. However, it is not necessarily the only sufficiently good formulation of floating-point, nor is floating-point necessarily the only way to design a number system with the right balance between dynamic range and precision to be universally useful.

2.2.3 Other number systems

The IEEE 754 standard is by no means the only system that has been proposed to standardize floating-point behavior and permit reliable data exchange and reproducible computation. Some systems, such as the "Morris floats" [32] or more recently "unums" [23], offer better numerical properties than IEEE 754 floating point: tapered precision for Morris floats, and a bit efficient representation of rigorous interval bounds for unums.

However, it has proved hard for these systems to catch on, as they do not provide an asymptotically larger dynamic range or greater amount of precision compared to IEEE 754 with a given number of bits. Most new number systems seek to optimize the binary representation as much as possible, or are being developed for high-performance, low-precision applications where all commonly used IEEE 754 formats are too large and slow.

Posits

Posits [22] are a new number system, proposed by John Gustafson as an alternative to IEEE 754 floating-point. Posits are very similar to floating-point, and indeed can be conveniently configured to match the representation sizes and dynamic ranges of the IEEE 754 types.

Rather than representing the exponent as a single fixed-point value that occupies a fraction of the representation's bits, posits break the exponent up into two values: a number of coarse-grained chunks, called the regime r, and a small adjustment, stored like traditional exponent bits. The value represented by a posit is $2^{u\,r+value(exponent)}value(significand)$, where the exponent and significand are interpreted as for typical floating-point, and the constant u is 2^{es} for a representation with es exponent bits.

The difference between posits and traditional floating point is that the regime of a posit is encoded with a unary encoding which allows smaller regimes to be represented with a smaller number of bits. The encoding includes a mechanism to detect where the regime ends dynamically, so that within a single representation, different values will have different regime sizes (based on the magnitude of their exponent), and thus different numbers of bits left over

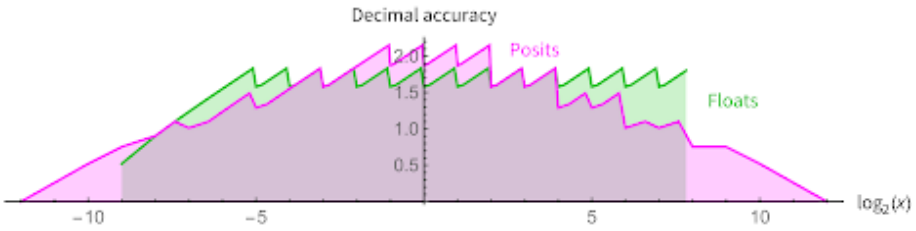


Figure 2.2: Accuracy of floats and posits over the dynamic range [21]

to form the significand. Because of this, posits have tapered precision - the significands of numbers near to 1 in magnitude are longer, allowing more precision, while towards the extreme ends of the dynamic range, the precision falls off as more bits are taken up by the regime.

Figure 2.2, borrowed from [21], shows a comparison of the accuracy of 8-bit posit and floating-point formats across the dynamic range. Floating-point shows a saw-tooth pattern of precision, familiar from fixed-point, that repeats for each exponent; posits show a similar pattern within exponents of a given regime, but also gradual tapering to either side based on the size of the regime.

The properties of floating-point and posit number systems are largely similar. Both can provide dynamic range exponential in the number of elements in the representation, or doubly exponential in the number of bits. Posits can actually be configured to provide much greater dynamic range, by choosing a number of exponent bits greater than the total number of bits in the representation, but this is probably not useful in practice. In cases where many values encountered in a computation can be scaled towards 1 in magnitude, the tapered accuracy of posits can be an advantage; in other cases, the flat precision distribution of floating-point can offer higher minimum precision over a range of values. In practice, low-precision posits have been shown to be a useful number system for many numerical computations, such as weather simulations [8].

Representing real numbers with bisection

Floating-point and posits are examples of different but similar ways to build number systems with sufficient dynamic range and precision to be universal. One natural question is whether there is a more general abstraction that explains both of them and concisely describes their relationship.

Peter Lindstrom outlines such an abstraction in [29]. To explain this abstraction, consider a binary string representing a number as a series of (binary) yes/no questions, each about the real number the string represents. What is the best sequence of questions to ask?

These questions can be broken up into two categories. First, it is necessary to perform an unbounded search to bracket the represented value. This corresponds to the exponent part of a floating-point or posit representation. Second, once the value is bracketed, additional questions can be asked to refine the value via binary search. This corresponds to the significand.

Lindstrom provides a framework for describing any such series of questions in terms of a generator function for bracketing, and a mean function for refinement via binary search. Particular settings of these functions produce number systems that resemble IEEE 754 floating-point (though the bracketing function is extremely contrived, reflecting the uneven distribution of information across the bits of many IEEE 754 floating-point values) and posits. Other configurations are possible, including smooth number systems that avoid the saw-toothed precision behavior seen in Figure 3, but don't admit obvious implementations of basic arithmetic operations.

2.3 Emerging number systems for emerging applications

When designing number systems, it is critical to take the needs of the applications that will run on them into consideration. Most applications have substantially similar needs: given sufficient precision and dynamic range to produce an acceptably accurate result, the hardware implementing the number system should ideally be able to store as much data, and perform

as many calculations, as quickly and efficiently (in terms of system power consumption) as possible.

The great success of universal number systems like IEEE 754 is in finding a representation with a finite, fixed number of bits that has sufficient precision and dynamic range for the vast majority of applications. Because of this, building a more powerful computer can be reduced to building more, faster, and more efficient logic for the same number system, not building a more powerful number system. With Moore's law and Dennard scaling in full effect, this improvement happens more or less automatically with each process generation, giving in effect a free, multiplicative speedup for all applications, without the need for any additional number system design.

It is only now that Dennard scaling has largely tapered off that that significant interest has been brought back to number system design in order to improve application performance and efficiency. The IEEE 754 representation is in one sense quite small, being constant in size - under the reasonable assumption that the cost in terms of area, power, and operation time of a number system is proportional to the bits in its representation, this means that there is no more than a constant improvement to be had by switching from IEEE 754 to more compact representations. However, with few alternatives, even this constant improvement has proven to be worth investigation for certain applications.

2.3.1 Application-specific accelerators

Graphics processing units, or GPUs, are probably the most widely used class of hardware accelerators for numerical computations. Traditionally used for rendering 3D graphics, they have recently seen wide adoption for other embarrassingly parallel, computationally intensive tasks, including AI and deep learning.

Historically, GPUs have often deviated from strict adherence to the IEEE 754 standard. Prior to the Fermi architecture, for example, NVIDIA GPUs used single precision IEEE 754 floating-point, but did not support computing with subnormal values [11]. Properly handling all cases involving subnormals for operations like multiplication requires a significant amount

of overhead, increasing latency. Since these numbers occur rarely in most of the applications that are run on GPUs, it is often an acceptable tradeoff to lose some dynamic range (and compatibility) in order to speed up operations.

Inside the GPU, in parts of the rasterization pipeline that are not directly exposed to the end user as compute operations, computations can be even more specialized and diverge further from the IEEE standard. Over 125 different number formats are used in different parts of the hardware inside Intel GPUs [16]; since they are not exposed as compute functionality available to software, however, only the hardware designers have to reason about their behavior directly.

More recently, GPUs such as NVIDIA's Volta architecture have included support for "half precision" 16-bit floating point [12], which was until recently not an official part of the IEEE 754 standard. To this day, there is disagreement over what 16-bit floating-point should mean. While half-precision representations have significantly reduced dynamic range and precision, too small for most traditional scientific calculations found in high performance computing workloads, they are still sufficient for some emerging workloads in AI and deep learning. With half the representation size, GPUs such as Volta can provide half-precision operations at twice the rate of single-precision operations, trading off the strength of the number system for what amounts to one free generation of process scaling.

Beyond GPUs, even more specific hardware accelerators have been recently proposed for deep learning, specifically Google's TPUs [26], or tensor processing units, and the similar tensor cores introduced in NVIDIA GPUs, including Volta. These accelerators provide coarse-grained primitives, specifically small matrix multiplies, that can be used to implement deep learning workloads with very high performance at the cost of limited generality. For instance, while Volta shows only modest single-precision performance over the previous generation Pascal architecture of about 12.1 to 15.7 TFLOPS, NVIDIA claims that Volta can achieve 125 trillion "tensor operations"" per second using tensor cores, and Google's v1 TPUs achieve 92 trillion operations per second.

This performance is achieved by careful selection of number system parameters inside the

primitives. TPU v1 used a mixed fixed-point representation; later versions have updated this to use low-precision (16-bit) floating-point, making the operations more general. NVIDIA uses 16-bit floating-point inside the multipliers, but accumulates the output of each block with a 32-bit floating-point value.

Some more research-oriented accelerator designs go beyond customized high-performance number systems to implement customizable ones. For example, AdaptiveFloat [47] implements an IEEE 754-like number system with variable precision at the granularity of individual layers that can improve the application performance of some deep learning workloads at very low bitwidths.

2.3.2 Compiler frameworks for numerical computing

Hardware capabilities for exotic number systems are only useful if we can write software that runs on them. Increasingly, this depends on complex compiler frameworks, such as (in the space of machine learning) TensorFlow [1], PyTorch [35], and TVM [8]. In the broader space of numerical computing, compilers for specific application domains have existed for a long time: FFTW [19], OSKI [49], FEniCS [3], and Halide [36], to name a few.

While these frameworks were not inherently designed for computing with new number systems, or for multi-precision, multi-format computation involving many number systems simultaneously, it will become increasingly important to support these kinds of computations as hardware support for them grows, and the performance reasons for using that hardware become more pronounced.

2.4 Mixed precision tuning

Though we have so far treated IEEE 754 as a single, universal number system, in reality it is at least two commonly used number systems: 32-bit single precision, and 64-bit double precision. Since both are implemented in hardware by many processors, it is already possible to optimize representation size or performance for a particular application by choosing which width to use for each stored value or operation.

A considerable amount of research has gone into the area of mixed-precision tuning for IEEE 754 floating-point. Precimonious [41] automatically searches for a configuration of IEEE 754 types, selecting among 80-bit long double, 64-bit double, and 32-bit single precision for each program variable, that gives maximum performance for the desired accuracy. One of the largest challenges is finding a way to systematically explore the search space, as even with three available types, there are an exponential number of configurations to test. Subsequent work [40] improves the performance significantly by using a new search strategy.

Other tools use different techniques to explore the search space and provide guarantees about the discovered mixed-precision programs. FPTuner [9] provides rigorous bounds about the error of a tuned computation on a given input domain based on a formal analysis using symbolic Taylor expansions, similar to that used in the FPTaylor tool [46]. Such guarantees are particularly useful when replacing code in applications with strict accuracy requirements, but the cost of the formal analysis can be prohibitive when scaling to large computations.

Alternatively, Schufza et al. describe a stochastic technique [44] for optimizing small numerical library functions, by extending the STOKE superoptimizer. This approach has the advantage that it can discover optimizations that are format-specific and don't admit easy formal analysis based on the properties of real numbers. A new precision-tuning approach proposed by Khalifa et al. [2] avoids a stochastic or exhaustive search completely by deriving precision constraints from the program being optimized, and then solving them as an integer linear program.

Chapter 3

FPCORE: SPECIFYING NUMBER SYSTEMS WITH REAL NUMBERS

We present a vision of rounding as a point of standardization to think about number systems and describe computations that use them. To turn that vision into something practical, we need to formalize it.

Per our thesis, we want this formalism to be general and mathematical in nature, so that it can be used to reason natively about algorithms designed in terms of real numbers. At the same time, we want it to be specific enough to describe all of the potentially ugly behavior of particular number systems, and elegantly bring those details back to the level of the algorithm.

The IEEE 754 floating-point standard is an existing, widely used example of this kind of formalism. First, we will study it as an example to guide the reasoning behind our design. Then we will work out the mathematical underpinnings of a more general framework for describing the semantics of number systems, which we have deployed on top of the existing FPCore language for specifying numerical benchmarks [18].

3.1 Concrete bit-level semantics in IEEE 754

For purposes of illustration, consider an IEEE 754-like 5-bit floating-point format, with one sign bit, three exponent bits, and one explicit bit for the significand.

Every concrete number system has two parts: some set of representable values, and some method of performing computation with those values. Our example uses bitvectors of length 5 for its representation. Figure 3.1 gives all of the representable values in a table. Computation can be defined in terms of these bitvectors; Figure 3.1 also gives a table with

Bitvector	Value	Bitvector	Value
0b00000	0	0b10000	-0
0b00001	$\frac{1}{8}$	0b10001	$-\frac{1}{8}$
0b00010	$\frac{1}{4}$	0b10010	$-\frac{1}{4}$
0b00011	$\frac{3}{8}$	0b10011	$-\frac{3}{8}$
0b00100	$\frac{1}{2}$	0b10100	$-\frac{1}{2}$
0b00101	$\frac{3}{4}$	0b10101	$-\frac{3}{4}$
0b00110	1	0b10110	-1
0b00111	$\frac{3}{2}$	0b10111	$-\frac{3}{2}$
0b01000	2	0b11000	-2
0b01001	3	0b11001	-3
0b01010	4	0b11010	-4
0b01011	6	0b11011	-6
0b01100	8	0b11100	-8
0b01101	12	0b11101	-12
0b01110	∞	0b11110	$-\infty$
0b01111	NaN	0b11111	NaN

x	$\sqrt{x}$	x	$\sqrt{x}$
0b00000	0b00000	0b10000	0b10000
0b00001	0b00011	0b10001	0b01111
0b00010	0b00100	0b10010	0b01111
0b00011	0b00100	0b10011	0b01111
0b00100	0b00101	0b10100	0b01111
0b00101	0b00101	0b10101	0b01111
0b00110	0b00110	0b10110	0b01111
0b00111	0b00110	0b10111	0b01111
0b01000	0b00111	0b11000	0b01111
0b01001	0b00111	0b11001	0b01111
0b01010	0b01000	0b11010	0b01111
0b01011	0b01000	0b11011	0b01111
0b01100	0b01001	0b11100	0b01111
0b01101	0b01001	0b11101	0b01111
0b01110	0b01110	0b11110	0b01111
0b01111	0b01111	0b11111	0b01111

Figure 3.1: 5-bit floating-point number system, with implementation of square root

the input and output bitvectors for computing the square root.

For finite number systems, tables are the most direct way of defining the parts. While they are not too cumbersome for our small example, building tables will not scale to more realistic number systems like 32-bit IEEE 754 floating point. To avoid this limitation, the IEEE 754 standard gives algorithms to construct the tables, based on the bitvector representation of values, the parameters of the particular number system, and, for mathematical operations, real arithmetic. Implementations of the standard, for example in processor hardware or in software math libraries like libm, would also perform the operations with algorithms (or circuits) that map bitvectors to bitvectors. In this way, represented values can be interpreted, and computation can occur, without ever explicitly constructing the table.

Figure 3.2 shows the calculation, according to the IEEE 754 floating-point standard, of the real number represented by the bitvector `0b11010`. The bitvector representation is

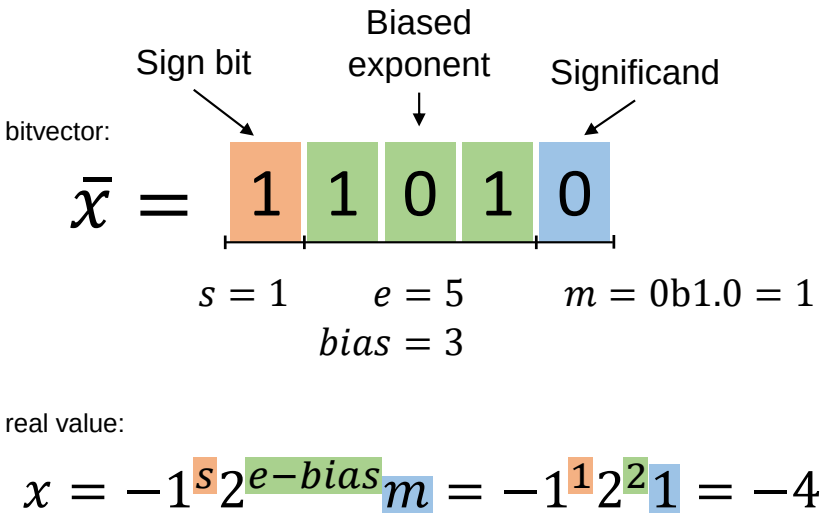


Figure 3.2: Anatomy of a floating-point value. The number -4 is represented with the bitvector `0b11010` in a 5-bit IEEE 754-like format.

broken down into fields, specifically, the sign, exponent, and significand mentioned above when we introduced the example number system, and based on the integer values of the binary numbers in the fields, the represented real value of -4 can be computed.

The IEEE 754 floating-point standard gives a *concrete data type* for representing a variety of number systems. By varying the parameters of the encoding—that is, the total size of the bitvector, or bitwidth, and the relative sizes of the exponent and significand fields—the standard specifies a variety of different floating-point number systems. The representation is concrete because the actual structure and implementation of represented values is transparent. IEEE 754 explicitly specifies not just the set of real numbers that should be represented, but how they should be represented with objects that can exist on a computer.

Operations on this IEEE 754 concrete data type are specified at two different levels. In the specification, it is mandated that certain basic arithmetic operations should be correctly rounded with respect to real arithmetic. That is to say, given two representable inputs, then

the output value should match the value obtained with the following procedure: First, obtain the real values of the inputs; then perform the corresponding operation with real arithmetic; and finally, find the closest representable number to this real output, according to the number system and the specified rounding mode. We call this model *rounded real-valued computation*.

In a given implementation of the IEEE 754 standard, operations will actually be performed as algorithms over the concrete bitvectors of the representation, as real arithmetic is not available for use in a concrete implementation. The concrete nature of the semantics is particularly useful when building implementations of the number system, as it ensures that there is no dependence on real arithmetic in the ultimate hardware specification, and it guarantees that binary data can be transferred between implementations.

However, the standard has limitations. It only describes floating-point number systems, not fixed-point or other more exotic representations such as posits. While the rounded real-valued semantics can be used to relate low-level number system behavior back up to the level of mathematical algorithms, this only works for one very specific class of number systems. As a concrete bit-level standard, IEEE 754 is ill suited as a general mathematical framework for capturing the behavior of all number systems, rather than just a particular one.

3.2 An abstract data type for number systems

To power our semantic framework, we want an *abstract data type* rather than a concrete one. An abstract data type captures behavior while keeping implementation details opaque, or that is to say, abstract. Where IEEE 754 specifies how bitfields are packed into machine words, we would rather have a formalism written in terms of summations. The challenge is keeping the framework general enough while still giving it the power to describe potentially ugly, low-level number system behavior.

Returning to our example number system, we can make the bitfields go away by simply dropping the whole representation onto a numberline, as show in Figure 3.3. The representation is reduced to its most elemental form: a potentially arbitrary set of real numbers. Similarly, we can adopt the idea of rounded real-valued computation to define a simple spec-

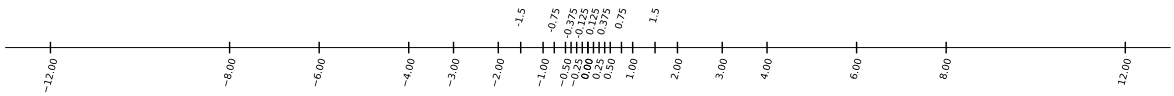


Figure 3.3: A 5-bit floating-point number system representation on the real line. Each tick shows a representable real number. The number system also includes some members that do not represent real numbers: ∞, $-\infty$, -0, and NaN.

ification for operations in terms of this set. Everything is a real number; just take the real number inputs, which may or may not be representable in the number system, perform the operation in real arithmetic, and then select the nearest representable real number in the set to the output under some kind of rounding rules.

The problem with this approach is that, like the naïve table implementation of IEEE 754, it doesn't scale to number systems with a realistic quantity of representable numbers. The challenge is in specifying the representation. If we know what the representation is, then some form of rounded real-valued computation should get us most of the way to a semantics for operations. But, how can we specify a set of useful subsets of the reals for building number systems in terms of a few simple parameters?

3.2.1 p and n: parameters for abstract number system representations

Instead of parameters over bitfields, we define our abstract number system type in terms of two parameters over binary numbers. We will call these parameters p, roughly standing for "precision," and n, or roughly the most significant "unknown" bit. Binary numbers are different from bitvectors in that, while they admit various efficient bitvector implementations, they don't specify any particular one. A binary number is just any number that can be represented with an expression of the form:

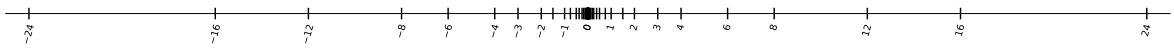


Figure 3.4: Some representable values for an ideal floating-point number system with $p = 2$ bits of precision, between -24 and 24. The pattern extends to cover the whole real line; values close to zero overlap in this graphic and are indistinguishable.

$$\sum_i \bar{x}_i 2^i \tag{3.1}$$

$\bar{x}$ is some (potentially infinte) binary string; its indices must be marked in some way, so that we can determine which values of i to plug into the expression that determines the value of x. Not every real number has a binary representation, but by increasing the number of binary digits included in $\bar{x}$, we can find a binary approximation that is within an arbitrarily small amount of error.

In this general form, the binary number does not admit a particularly convenient representation (as we might have to label each individual bit with its own integer index). To improve on this situation, we can normalize the representation and introduce a scale factor based on a separate integer exponent e. This gives us a general floating-point representation, and introduces our first parameter, p.

$$2^e \sum_{i=0}^{p-1} \bar{x}_i 2^{-i} \tag{3.2}$$

Instead of allowing the digits to be spread all over the numberline, this representation records a starting offset position as a single integer (the exponent) and requires all of the digits to be packed together relative to this index. This permits a convenient concrete implementation using a bitvector, though the representation in no way requires this. The number of digits is bounded by our parameter, p, which as we have suggested gives the

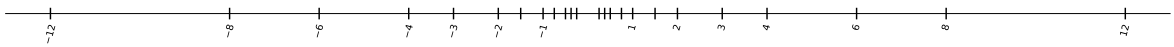


Figure 3.5: Representable values for a finite floating-point number system with $p = 2$ bits of precision, limited to exponents between -2 and 3. Notice the gap near zero.

precision, or the size of the binary fraction that has been used to represent a particular number. By limiting the maximum value of p allowed in the representation (or requiring that all represented values have a particular value of p, though of course parts of the fraction could be filled in with zeros) we can define various ideal, finite-precision, floating-point number systems. For example, if we set $p = 2$, we can represent the real values marked in Figure 3.4.

Ideal floating-point representations of this form are like a superset of an IEEE 754 format: though each representable value has a finite amount of precision, there are no bounds on the values of the exponent, so any such system allows an infinite number of representable values. This is easy enough to fix by also imposing bounds on the value of the exponent, which we can call e_{min} and e_{max}. If we specify these bounds, as well as a particular p, we can define a number system that is almost but not quite the same as what we get with IEEE 754.

In the case of our example, if we want to duplicate the behavior of an IEEE 754 number system with the following properties:

- 1 sign bit (always)
- 3 exponent bits
- 1 explicit significand bit

then we would specify in our new system:

- $p = 2$ (including the implicit bit)

- $e_{max} = 3$
- $e_{min} = -2$

to obtain the set of real numbers in figure 3.5.

Looking closely at this numberline, we can see that it's missing a few values from our original set of real numbers. There is a gap between the smallest representable number and 0, which is somewhat larger in size than the spacing between (for example) the smallest and second smallest representable positive numbers. This gap will always occur in a finite precision floating-point representation with a finite minimum exponent.

IEEE 754 fills in the gap with a special type of number referred to as "subnormal" or "denormal." Much ado is made about these special edge-case values, particularly in high-performance implementations where supporting them may come at a steep cost of additional circuitry, or potential data-dependent variability in performance. However, from the point of view of a specification in terms of real numbers, there is nothing special or unusual about subnormal numbers: they are just real numbers, with a different (but actually simpler) parametric representation scheme.

The infamous IEEE 754 subnormals are just a fixed-point representation. Instead of normalizing the binary representation by aligning the digits on the left and scaling with an exponent, we can also standardize the indexing of digits by insisting on some smallest representable binary place, and shifting the index exponents (or equivalently, scaling by some fixed exponent) to align with it.

$$2^e \sum_{i>n} \bar{x}_i 2^i \tag{3.3}$$

This expression looks a lot like our original definition for the real value of a binary number, but we insist that each index is larger than some parameter n, which tells us the most signficiant index (or binary place) that is not allowed to be present (i.e. non-zero) in a represented number. Number systems of this form are ideal fixed-point formats; we can

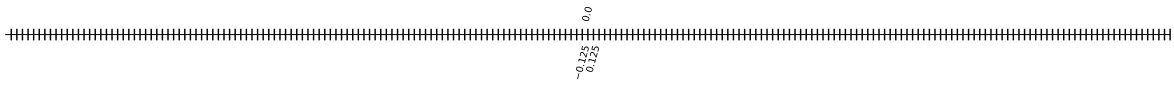


Figure 3.6: An ideal fixed-point number system with a largest unrepresentable binary digit $n = -4$. The pattern extends to cover the whole real line.

think of n as the granularity of representable numbers, as the distance between any two representable numbers is 2^{n+1}. Figure 3.6 shows a fixed-point system with $n = -4$. There is no inherent bound on the largest representable number, but as with floating-point systems specified in terms of p, we could easily limit the maximum exponent to produce a finite system.

Choosing $n = -4$ is convenient, because if we look at the representable values just adjacent to zero, we will see that they include all of the numbers representable in our 5-bit IEEE 754 format but missing from the floating-point system with $p = 2$ and appropriately bounded exponents. To reproduce IEEE 754's representable numbers exactly (at least the representable reals) we can combine a floating-point and a fixed-point number system to get, with specific values of p and n depending on the IEEE 754 parameters we want to emulate:

- $p = 2$ (including the implicit bit)
- $e_{max} = 3 = 2^{ebits-1} - 1$
- $e_{min} = -2 = 1 - e_{max}$
- $n = -4 = e_{min} - p$

Together, these parameters p and n, along with appropriate bounds on allowed exponents, let us define in terms a few integer parameters not just all of the possible sets of real numbers representable by IEEE 754 formats with arbitrary bitfield representations, but also any fixed point number system representation, and ideal number systems that have (for example)

unbounded exponents. We have done this abstractly, without any dependence on concrete representations or bit-level logic.

These parameters solve the hard problem, of specifying which real values a number system can represent. Occasionally we might want to represent things that aren't real numbers, which we will discuss in Section 3.3.3. We still have to address the other component of any number system—a way to perform computations—which as we will see is made easier at the specification level by the the power of rounded real-valued computation.

3.3 Computation is rounding

The premise of rounded real-valued computation is that rounding captures the entire semantic difference between computing with real arithmetic and working in a particular number system. Real arithmetic itself is the most permissive number system, with the least interesting rounding behavior: just never round anywhere. For other number systems where we know the more restricted set of representable values, we need a rounding function to map the true real outputs of computations back to representable numbers from the system.

3.3.1 Rounding $\sqrt{2}$

Let us return to our example 5-bit number system. Say we have the real number $\sqrt{2}$ and would like to represent it. How we got here is not particularly important; maybe we actually computed the square root of the exact representable value 2, or the value is an ideal mathematical constant provided as an input. What matters is that we must now choose a member of our finite set of representable values to represent it. Which one do we choose?

Figure 3.7 shows the representable numbers in the vicinity of $\sqrt{2}$. There are two obvious choices: we can either pick the next number 1.5, or the previous one, 1.0. If we pick any other representable number, than that is arguably the wrong answer, as we could have picked one of the closer ones instead (specifically, the one between that number and $\sqrt{2}$), and have less error, without even changing the sign of the error.

This decision seems obvious, but there are some very important subtleties. First, we

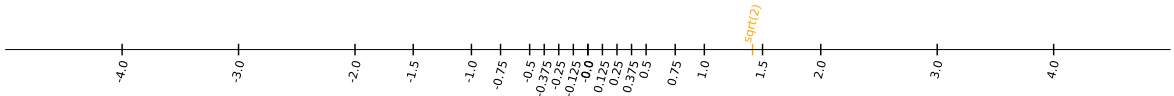


Figure 3.7: $\sqrt{2}$ shown on a number line with representable numbers in the example 5-bit floating-point format between -4 and 4.

are assuming that the goal is to approximate the real numbers as accurately as possible everywhere, particularly locally, at the scale of individual operations. This *does not in any way* guarantee globally optimal accuracy. In some cases, which we will actually find automatically later 6 It might be possible to produce *less accurate* results locally, but have a more accurate result for some larger computation, because the error will cancel out.

However, doing this comes with a huge amount of numerical peril. If we assume that our algorithm works like real numbers everywhere, then we can reason about it like any continuous algorithm, and we can independently improve the number system without breaking it. This makes numerical analysis possible. If we are depending on non-optimal local behavior, our algorithm is specific to the number system; all bets are of when reasoning about it the way would reason about normal real arithmetic, and changing the number system at all, even by making in more accurate, can completely break everything.

Sometimes this kind of specialization is warranted; there are plenty of bit hacks out there in IEEE 754 math libraries, and many custom hardware implementations that are allowed to produce inaccurate results for performance reasons. In almost all of these cases, the hacks incur additional development cost and are done to meet some important goal like application performance. Outside of these special requirements, it is a wiser idea to assume local real behavior to make it possible to do numerical analysis.

It is interesting to think about which way the rounding should go. It might seem like the only obvious, "correct" answer is 1.5, as it is much closer to the real value of $\sqrt{2}$—the absolute error in this case is less than one quarter the error from rounding to 1.0. However,

depending on how (and how consistently) the decision is made, we will see that rounding some values the "wrong" way can provide critical properties about the kinds of error and rounding envelopes produced by the number system.

3.3.2 Three kinds of rounding

We distinguish three flavors of rounding behavior, in order of increasing numerical guarantees. At the most basic level, committing to give one of the two nearest representable values is what we will call "faithful rounding," and is a necessary prerequisite for correct rounding in any system that locally approximates the reals.

This alone does not provide particularly strong properties for numerical analysis. We can strengthen the guarantees of the number system by further committing to round not just any particular value, but consecutive ranges of values ("consistent rounding"), or maximally to always round not just to a nearest representable value, but *the* nearest ("nearest rounding").

Faithful rounding

Formally, if the exact answer to some computation is the real number x, and the (exactly representable) result in some number system is $\hat{x}$, then that result is faithfully rounded if there is no real number y between x and $\hat{x}$ such that y is exactly representable.

In terms of error bounds, this guarantees a bound of one unit in the last place, or "ulp". Equivalently, the rounding envelope extends all the way to the next and previous representable values. This means that adjacent rounding envelopes will overlap.

This means that for any real value, it could round two different ways. The rounding rules are underspecified and do not define a single function that tells us what must always happen. This can be troublesome for analysis because it means that operations will lose properties like monotonicity that they might be expected to have over the reals. For this reason, faithful rounding is rare as a software-level abstraction, but the extra degrees of freedom it allows in low-level implementations can be worth the trouble for high-performance hardware.

Consistent rounding

If some real number x rounds to $\hat{x}$, then that rounding is consistent if there is no real number x' such that x' is between x and $\hat{x}$ and x' rounds to a different representable $\hat{x'} \neq \hat{x}$.

Essentially, a consistently rounded system does not allow any overlap in the rounding envelopes. This is extremely important for analysis because it means the rounding function is indeed a function. Among other properties, it now preserves monotonicity: if $x \leq y$ in the reals, then $\hat{x} \leq \hat{y}$ in the number system.

Most number systems exposed to software have this property: note that all IEEE rounding modes do. In terms of error bounds, the maximum error is actually the same as faithful rounding, at 1 ulp.

Nearest rounding

The strongest property we can imagine for rounding is to always round to the nearest value. This implies faithful and consistent rounding, necessarily.

Formally, if some real number x rounds to $\hat{x}$, then that rounding is nearest if there is no representable $\hat{y}$ such that $|x - \hat{y}| < |x - \hat{x}|$.

The only remaining decision is which way to break ties. As long as this is done consistently (so that rounding remains a function) all of the other properties will be preserved. For most implementations of IEEE 754, the round to nearest even mode (RNE) is used by default.

While nearest rounding does reduce the maximum error to half an ulp, it does not reduce the rounding envelope size compared to consistent rounding. As long as there are the same number of envelopes, they cover the whole real line, and they don't overlap, the average size must be the same.

For a given number system, capturing the exact flavor of rounding used is important to fully define its behavior, but this is much easier for practical systems than specifying the set of representable numbers. Most useful rounding behaviors are simple, as demonstrated by the IEEE 754 rounding modes: round down to the smaller magnitude value, for example, or

RNE.

3.3.3 Beyond real numbers

So far we have glossed over the parts of the IEEE 754 floating-point standard that deal with representable "numbers" that do not have a distinct real values. These are "negative zero", positive and negative infinity, and all representations of NaN, or "not a number."

These constructs are useful for a variety of reasons, mostly related to capturing error conditions that can occur when computations are attempted that are either ill-specified in the reals (e.g. $\sqrt{-2}$) or exceed the capabilities of the number system (e.g. computing $2^{2^{100}}$, for most number systems with a finite exponent).

Really, rather than the reals, we want to reason about arithmetic on a slightly more general set, which we can descriptively call the "affinely extended reals with signed zero and NaN." Arithmetic in this space is mostly the same as normal real arithmetic, but we have to also check the error conditions in certain cases, and any operation that would be ill-defined instead is fully defined and returns an error condition.

What the conditions are is possibly a good topic for further standardization. The IEEE 754 standard specifies certain rules for what error conditions should be returned by mathematical functions for inputs outside their domain; we will borrow these rules, but make no claim to have vetted their utility or completeness. Ensuring the arithmetic is fully specified with respect to all error conditions and other edge cases is a worthy goal in the long term, but is not particularly useful for most computations, which should be able to operate correctly in terms of real numbers even if the exception cases are undefined, as they are for the reals.

3.3.4 The rounding monad

We can describe rounded real-valued computation in its most general form as the action of a monad, which we call the rounding monad. Figure 3.8 gives an implementation in pseudocode.

```
return(x: R): -> M R
  wrap x with rounding context r;

bind (x: M R, f: R -> M R): -> M R
  unwrap x to determine rounding context r;
  find x' that represents x in context r;
  compute f(x');
```

```
f(x: R): -> M R
  return f(x);
```

Figure 3.8: Pseudocode implementation of the rounding monad

In the pseudocode, `R` is the type of real numbers $\mathbb{R}$, and `M R` is the monadic type for some number system.

The `return` operation takes any real number `x` and wraps it in the monadic type. This is different from rounding, as it does not change the value, it only annotates it.

Rounding is performed inside the `bind` operation, which takes a wrapped real value `x` and a function `f` over real values, unwraps `x` to produce a normal real value `x'`, and then applies `f` to `x'`. This unwrapping is rounding, as it changes the original wrapped real value `x` into some other real value `x'`.

The effect of the monad is to make rounding lazy, as it only happens on demand when a rounded value is consumed, rather than on production when the rounding specification is attached by `bind`. Any real function `f` can be converted to produce values in the monad by simply wrapping its output with `return`.

In this formulation, the concrete rounding context `r` is left arbitrary. It is global to the type `M`, so bind can modify it in arbitrary ways to implement stateful rounding. This makes the abstraction very general: we can, for example, describe number systems that alternately round up and down for consecutive operations, or that look at the inputs to `f` directly and bypass its real valued implementation.

In practice, the rounding context is usually simple and constant. For IEEE 754 number systems, the only things we need to keep in `r` are the exponent and mantissa sizes and the rounding mode.

```
(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)))
```

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

Figure 3.9: Example FPCore program (left), and mathematical notation (right)

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

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

Figure 3.10: The same program, after numerical improvement by Herbie

3.4 FPCore 1.0: a language for numerical programs

FPCore began life as part of Herbie [33], and was later split off as part of FPBench [13]. In its original form, FPCore 1.0 is a purely functional programming language that can describe numerical kernels. FPCore is ideal for working with number systems because it is designed to describe programs that depend on them, and interact with them, in exquisite detail.

3.4.1 FPCore by example

A small example program is provided in Figure 3.9, borrowed from [13]. This program finds one of the roots of a parabola according to the quadratic formula. It is interesting for numerical analysis because this particular formulation suffers from numerical instability. Mathematical notation for the expression the kernel computes is provided on the right side of the figure, while the FPCore code which implements this math is on the left.

Every FPCore program begins with the identifier `FPCore` and a list of arguments to

the program, which correspond to free variables in the mathematical notation. Here those arguments are the parameters of the parabola, a, b, and c. The syntax is based on S-expressions, reminiscent of Lisp or the SMTLIB input format for SMT solvers [4], making it easy to parse and process with automated tools while still being human readable.

Lines 2-4 provide some metadata properties, such as the `:name` and a source to `:cite` for the origin of the benchmark. These two properties have no effect on evaluation of the program, but other properties might specify details like the IEEE 754 datatype or the particular math library the computation is intended to use. Following the precondition, the main program body is given on lines 5-6. Both of these expressions describe math and logic operations in prefix notation. Supported operations include all typical math functions from the C11 standard. The non-standard `sqr` operation, for computing the square, was provided as a special case for Herbie, but was later removed from the standard (since it could be implemented as a user-defined function).

FPCore is a middle ground between mathematical notation and more traditional programming languages. Most algorithms that can be written down in mathematical notation can also be described by FPCore, as long as they use operations available in C11. FPCore makes the whole computation tree explicit, so there can be no confusion about things like order of operations or associativity (does $1+2+3$ compute $1+2$ first, or $2+3$?) which might not matter in real arithmetic, but do matter when using a finite precision number system.

Compared to more general-purpose programming languages like C or Python, FPCore only allows for description of the numerical part of the application, plus a bit of metadata—no string processing, complex data structures, I/O, etc. This means that whole applications cannot be directly ported into FPCore, but the critical numerical kernels they depend on can. Performing analysis is generally easier once the math is isolated.

This situation is ideal for tools like Herbie. Figure 3.9 is a typical input to Herbie, and Figure 3.10 shows the tool's output, after automatically rewriting the program to have better numerical behavior. This particular example again is from [13]; Herbie produces the FPCore program on the left, and the mathematical notation on the right is created manually to

explain its behavior.

Both programs compute the same thing, but they use different algorithms at the mathematical level. FPCore makes it easy for Herbie to explain these differences in an automatically generated output. This more complex program uses FPCore's `if` statements to introduce control flow. Since FPCore is just a standard, and does not have a standardized execution environment outside of Herbie, a user would typically translate these numerically improved output programs into another language like C in order to actually run them. FPBench provides a suite of compilers to partially automate this process.

3.4.2 FPCore and number systems

So far we haven't said anything about FPCore's relationship with number systems. That is because while it deals with numerical programs, which by definition must operate in the presence of some number system, FPCore 1.0 is deliberately vague about how programs should be executed and formally leaves number system behavior up to tool implementers [18]:

> FPCore expressions can describe concrete floating-point computations, abstract specifications of those computations, or intermediates between the two. The semantics of FPCore are correspondingly flexible.
>
> Following IEEE-754 and common C and Fortran implementations, FPCore does not prescribe an accuracy to any mathematical functions except the arithmetic operators, sqrt, and fma. If the exact accuracy is important, we recommend that benchmark users declare the implementation used with the :math-library property.

This is perfectly fine in a world where everything uses IEEE 754. Herbie's numerical improvements are number system specific, and simply assume that the programs will be run with IEEE 754 double precision; otherwise the bounds for the conditionals won't make a lot of sense.

At the same time, FPCore is ideally positioned to do more than just describe numerical programs that use IEEE 754 floating-point. Most programming languages already have enough on their plate, so to speak, when trying to formally define their semantics, and would rather stay as far away from real arithmetic as possible, with a concession to use IEEE 754 being a sort of effective compromise. FPCore, however, is not so burdened. The entire purpose of the language is to describe the behavior of numerical kernels, and the particularities of running them in a finite precision environment.

3.5 FPCore 1.1: a language for number systems

Our contribution to FPCore is a specification layer for number systems themselves, not just programs that run on top of them. Semantically, the specification layer resembles the rounding monad. FPCore 1.1 formally clarifies what this means:

> FPCore expressions can describe concrete floating-point computations, abstract specifications of those computations, or intermediates between the two. The semantics of FPCore are correspondingly flexible, **and are made explicit by the rounding context**.
>
> Function applications round their results using the rounding context. More precisely, a function application `(`f e_1 `...)` in a rounding context ρ must evaluate to the same value as if f were evaluated in exact real arithmetic, and then rounded according to the rounding context.

The rounding context is just some additional state that determines the behavior of rounding when operations are performed; in the definition of the rounding monad we called it `r`. Syntactically, the rounding context is controlled by lexically scoped precision annotations which can be added at the top level, or wrapped around individual expressions with the `!` operation.

3.5.1 Precision annotations

Let us consider a few examples of FPCore programs to illustrate. The following program computes $x + 1$ and does not contain any precision annotations:

```
(FPCore (x)
  (+ x 1))
```

While there are no explicit annotations in this program, there is still a rounding context. For backwards compatibility with tools like Herbie that assume FPCore programs will use IEEE floating-point, the default context if none is provided explicitly is 64-bit IEEE 754 double precision. The following FPCores all use the same rounding context, either implicitly as the default, explicitly as a top-level annotation, or explicitly as an annotation around the expression `(+ x 1)`.

```
(FPCore (x)     (FPCore (x)             (FPCore (x)
  (+ x 1))        :precision double       (! :precision double (+ x 1)))
                  (+ x 1))
```

Precision annotations can contain any number of properties for the rounding context. A property is a pair of a key, which must be an FPCore symbol starting with a colon, and a value, which can be an arbitrary s-expression. This makes the language very flexible, as any relevant information for rounding behavior can be encoded as s-expression data and tagged with some appropriate key. The FPCore standard leaves the interpretation of this data up to individual tools that use the standard. Tools may also read other data from the environment (such as the inputs to operations), and keep persistent state in the rounding context between operations.

For convenience, FPCore recommends common notations for common number systems such as IEEE 754 floating-point and general fixed-point computation. Double precision can be requested in a few different officially documented ways:

```
(FPCore (x)             (FPCore (x)               (FPCore (x)
  :precision double       :precision binary64       :precision (float 11 64)
  (+ x 1))                (+ x 1))                  (+ x 1))
```

Alternatively, we could specify a mathematical version of double precision in terms of p and n directly, which would reproduce the behavior of subnormals but not be limited by a

maximum exponent:

```
(FPCore (x)
  :p 53 :n -1075 :round nearestEven
  (+ x 1))
```

Note that the properties `p` and `n` are not officially part of the FPCore standard, but this notation might be useful to particular tools or analyses. Other properties such a `:round` are officially documented to control aspects of the rounding behavior, such as specifying an IEEE 754 rounding mode. Properties can be specified independently and mixed together according to the needs of individual users.

One special case is the `:precision real` annotation, which indicates that a subcomputation should not computed with real values and never rounded.

```
(FPCore (x)
  :precision real
  (+ x 1))
```

This is very different from not specifying a rounding context, which as discussed above reverts to a reasonable finite-precision number system as a default behavior. Describing true real-valued components of computations is valuable for a variety of reasons, particularly for writing mathematical specifications. We can, for example, write down a simple description of the C11 `expm1` operation:

```
(FPCore (x)
  :spec (! :precision real (- (exp x) 1))
  (expm1 x))
```

By using real precision in the body expression, we can also define new fused operations. Of course, reasoning about such constructs is hard, and is not a task for the FPCore standard. That is left to tools and mathematicians that use it.

3.5.2 Lexical scoping of precision annotations

Precision annotations are lexically scoped. An annotation modifies the rounding context of the expression contained lexically within it. For example, in the FPCore

```
(FPCore (x)
  (! :precision double
    (- (! :precision single (+ x 1)
       1))))
```

`(+ x 1)` is computed with (IEEE 754) single precision, but the final subtraction of 1 from this intermediate value is computed with double precision.

Annotations apply to entire subexpressions, so if we apply a precision annotation to the quadratic formula example

```
(FPCore (a b c)
  :precision double
  (/ (+ (- b) (sqrt (- (sqr b) (* 4 (* a c)))))
     (* 2 a)))
```

It will determine the rounding behavior of every mathematical operation in the computation. Variable bindings can be used to restrict annotations to the relevant parts of a computation:

```
(FPCore (a b c)
  :precision single :round toZero
  (let ([x (! :precision double
              (+ (- b) (sqrt (- (sqr b) (* 4 (* a c))))))])
  (/ x (* 2 a))))
```

In this version of the quadratic formula, the intermediate value `x` is computed with higher precision, but the denominator `(* 2 a)` and the final division will use single precision, as specified at the top level. All operations will inherit the `toZero` rounding mode.

3.5.3 Subtleties of precision annotations

Precision annotations are very expressive, because they allow for the description of any conceivable rounding behavior, and they allow the behavior to be changed at a fine granularity, on the level of individual operations. Lexical scoping also makes it convenient to add multi-precision rounding information to existing computations without having to annotate each operation individually, but also without sacrificing any generality where individual operations need to be rounded in a particular way.

There are a few important subtleties of working with the rounding context, and with rounded real-valued computation in general. Rounding only applies to the output of operations. Variables in FPCore are never rounded: the following FPCore expressions are all the same thing, which is just `x`:

```
x    (! :precision double x)    (! :precision double (! :precision single x))
```

The cast operation can be used to explicitly round a value in some rounding context, for example to convert a value to single precision and then back to double precision:

```
(FPCore (x)
  (! :precision double
    (cast (! :precision single (cast x )))))
```

Having mathematical variable bindings simplifies the semantics of FPCore, as no special considerations need to be made for data storage. This also prevents any multiple rounding. While it is possible to write expressions that return real values that might be hard to store, most practical computations will use a finite-precision rounding context. This means that while values are stored exactly, they will mostly be rounded (finite-precision) values to begin with, so storing them is not an insurmountable challenge.

Special rules are provided to deal with constants and inputs to FPCores. Constants are rounded in the context where they are defined; the expression `(! :precision double PI)` represents the nearest representable value to the mathematical constant π in double precision. Changing the context changes the nearest representable value. Inputs are also rounded, using either top-level annotations from the FPCore or per-input annotations if they are provided. Of course, any of these rounding contexts can be given `real` precision to handle true real values, at least at the specification level.

FPCore's rounding behavior, and therefore its semantics for number systems, is tied to mathematical operations. In contrast to languages like C, which treat precision as a property of the way values are stored, FPCore only rounds at the output of operations. This is the ideal point to mediate between real arithmetic and the behavior of particular number systems. As long as we can define any arbitrary rounding behavior, we can describe any number system; at the same time, real arithmetic is conveniently available to provide the unrounded results, so there is no need to redefine common notions like addition or exponentiation for every single number system.

```
(FPCore lorenz-3d ((xyz 3))
 :precision (float 5 16)
 (let ([sigma 10]
       [beta 8/3]
       [rho 28]
       [x (ref xyz (# 0))]
       [y (ref xyz (# 1))]
       [z (ref xyz (# 2))])
  (array
      (* sigma (- y x))
      (- (* x (- rho z)) y)
      (- (* x y) (* beta z))
  )))
```

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - x) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

Figure 3.11: Lorenz system, FPCore and mathematical notation

```
(FPCore vec-scale ((A n) x)
 (tensor ([i n])
  (* (ref A i) x)))
(FPCore vec-add ((A n) (B m))
 :pre (== n m)
 (tensor ([i n])
  (+ (ref A i) (ref B i))))
(FPCore rk4-3d ((xyz 3) h)
 :precision (float 5 14)
 (let* ([k1 (! :precision (float 5 13)
          (vec-scale (lorenz-3d xyz) h))]
        [k2 (! :precision (float 5 10)
          (vec-scale
           (lorenz-3d (vec-add xyz (vec-scale k1 1/2)))
           h))]
        [k3 (! :precision (float 5 12)
          (vec-scale
           (lorenz-3d (vec-add xyz (vec-scale k2 1/2)))
           h))]
        [k4 (! :precision (float 5 9)
          (vec-scale (lorenz-3d (vec-add xyz k3)) h))])
  (tensor ([i (# 3)])
   (+ (ref xyz i)
      (* 1/6
         (+ (+ (+ (ref k1 i) (* (ref k2 i) 2))
                  (* (ref k3 i) 2))
            (ref k4 i)))))))
```

$$k_1 = f(x_n)$$
$$k_2 = f(x_n + h\tfrac{k_1}{2})$$
$$k_3 = f(x_n + h\tfrac{k_2}{2})$$
$$k_4 = f(x_n + hk_3)$$
$$x_{n+1} = x_n + \tfrac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$$

Figure 3.12: 4th order Runge-Kutta method, FPCore and mathematical notation

```
(FPCore main ((initial-conditions 3) h steps)
 (tensor* ([step steps])
  ([xyz initial-conditions (rk4-3d xyz h)])
  xyz))
```

Figure 3.13: RK4 driver FPCore

3.6 FPCore 2.0 and beyond

Besides providing a semantics for number system behavior, FPCore is also a general purpose language for writing numerical kernels. This aspect of it isn't a major contribution of this work (there are, after all, many general purpose programming languages out there) but it is interesting to reflect on some of the experiences we have had, and look towards the future.

The latest revision to the standard is FPCore 2.0, which adds some quality of life features to make writing realistic programs easier. The most important features are function abstraction, by allowing FPCore programs to be referred to and used as operations in other FPCore programs, and structured data in N-dimensional arrays. To illustrate these features in action, Figures 3.11, 3.12, and 3.13 show a complete implementation of the Lorenz System example with RK4, using mixed low-precision floating-point.

Functions are a critical abstraction both for implementing larger computations and for decomposing them in ways that are good for analysis. In its original form, FPCore programs were intended to be independent, stand-alone benchmarks, which is why the language has no formal description of functions or a top-level scope above individual programs, but this is not practical for implementing kernels from real-world applications.

FPCore handles name resolution as a dictionary; if an operation is performed that is not one of the usual primitives, then implementations should look in the dictionary to see if there is another program with this identifier that can be invoked. The dictionary can be filled in any way that is convenient for the implementation, for example by parsing all of the programs together at the top level and adding them before any are executed. Calling functions is different from executing a standalone benchmark in that inputs to a function are *never rounded* (they are already FPCore variables). In contrast, inputs at the top level are coming in from outside of FPCore, and are therefore rounded with the top-level rounding context as if they were constants.

Structured data is also important for implementing real benchmarks. Without some notion of structured data, a dot product of 1000-element vectors takes 500 times more in-

puts than a dot-product of 2-element vectors, even though both of these algorithms can be expressed with the same code in most languages. FPCore standardizes on N-dimensional arrays as its only data structure.

This allows for most of the benefits of having data structures, without adding analysis complexity to handle more complex structures like sets or maps. In practice, most numerical algorithms use arrays of varying dimensionallity, so it is natural to make this the data structure of choice. Arrays are implemented in a pure way, in keeping with the design of FPCore as a pure functional language, and operations are kept in terms of scalar values, to minimize the impact on the rest of the semantics. Arrays can be read element-wise, and they can be created by the array-creating construct `tensor`, but due to purity the can't be modified in place. To simulate mutation, programs can create a new (updated) array, as is standard in functional programming.

The FPCore 2.0 standard also provides new iteration constructs to make working with arrays more natural, specifically for loops. With the addition of these constructs, it is possible to represent most linear algebra kernels in about the same code size as other general-purpose languages. FPCore does not provide an implementation of BLAS or similar libraries, but such a thing would be possible to implement on top of it, and that is how we intend it would be provided (i.e. not as language primitives).

FPCore is still an evolving standard; we do not anticipate FPCore 2.0 to be the final revision. With that said, we do expect that the rounded real-value semantics of number system operations will be preserved in all future versions without change. This is the key core to the semantics that makes working with different number systems practical.

Chapter 4

TITANIC: IMPLEMENTING ROUNDED REALS WITHOUT INFINITE PRECISION

FPCore is only a *specification* for the semantics of number systems. It does not provide a concrete implementation to actually run numerical programs, except by working out their behavior by hand, with pen and paper, in terms of real arithmetic.

Because of the close relationship with the reals, implementing the semantics is tricky. Real numbers are difficult to represent because they might require infinite precision; in a binary system, there might be an infinite number of digits in the exact real representation of a particular number. Additionally, comparing two arbitrary real numbers is undecidable [38]. Taking the view of numbers as streams of digits, even if there are programs that produce the streams, comparing two of them requires determining arbitrary program equivalence since the streams themselves are infinitely long.

FPCore can therefore encode programs whose output cannot be computed. However, in a sense this is true of many languages, including FPCore, due to the possibility of infinite loops or recursion. We can still build a reference implementation for a large subset of FPCore's semantics, including support for the wide variety of number systems that we can describe in terms of p and n, that is total and does not require an implementation of real arithmetic. We do this in a new tool called Titanic, which serves as a laboratory for exploring the behavior of number systems. Instead of rounding true real values, Titanic rounds arbitrary precision values, which it can compute using the MPFR library with enough precision to ensure that the final, rounded representable value matches the behavior that would be expected from rounding a true real-valued output.

We say Titanic is a number system "laboratory" because it is has built-in support for

the most common binary number systems (fixed- and floating-point, as well as posits, with arbitrary representation parameters or values of p and n), it is easily extensible to implement other number systems in terms of their rounding behavior, and it has good enough performance to run interesting FPCore programs to empirically observe their behavior. Critically, Titanic allows completely different number systems to interact seamlessly in the same computation. It also provides a pathway to integrate reasoning about ideal, real-valued specification with concrete implementations of number systems.

4.1 Arbitrary precision computation

While we cannot finitely represent an arbitrary real number, it is possible to represent any real number to an arbitrary amount of precision. An arbitrary precision number system is one that does not place an upper bound on p for represented numbers. The precision p used to represent any given number is not bounded by some global limit, but each number must have some finite p, and numbers with a large amount of precision will require correspondingly more storage space in a concrete implementation.

Similarly, while comparing real numbers is undecidable in general, it is possible to finitely compare arbitrary precision numbers, and more generally to compute real functions that would occur as mathematical operations in a numerical program to arbitrary precision.

Arbitrary precision computation works the same way as rounded real-valued computation in a finite-precision number system, but the output precision p is given as a parameter to the computation. This way, an arbitrary (though again, still finite) amount of precision can be requested, to represent the true real value with an arbitrarily small amount of error.

4.1.1 The GNU MPFR library

The GNU Multiple Precision Floating-Point Reliable Library [17], or MPFR, is a concrete implementation of correctly-rounded, arbitrary precision floating point. We use its capabilities, particularly the guarantees it offers about correct rounding, as the core computation engine of Titanic.

MPFR represents numbers with an arbitrary precision format which greatly resembles our ideal binary floating-point number system. Representable numbers have a normalized significand of arbitrary size, limited by the amount of available memory, and floating-point style exponent. MPFR implements a variety of math functions, including basic arithmetic and all of the transcendental functions from the C11 standard and FPCore.

When performing a computation with MPFR, the precision of the output must be specified as a parameter, independently of the precisions of any inputs. The output is limited to this amount of precision. MPFR guarantees that the output will be correctly rounded, according the the specified rounding mode. All standard IEEE 754 rounding modes are supported, but we exclusively use the round towards zero (RTZ) mode in our implementation of Titanic.

Computation assumes that the values of the inputs are exact. MPFR is not an interval arithmetic library; if the inputs are not the true real values intended for the computation (i.e. if they are arbitrary precision values, rounded at some other precision) then error may accumulate across multiple operations. However, because the underlying number system allows values with arbitrary precision, MPFR will never round its inputs, as long as they can be specified as binary numbers.

Compared to other number system implementations, particularly hardware implementations of IEEE 754 with fixed precision, MPFR is relatively slow. As with any arbitrary precision library for computing for transcendental functions, it suffers from the "tablemaker's dilemma" [27]: it is not possible to bound in advance the amount of computation required for a given amount of output precision. However, given what it has to compute, MPFR is highly efficient for a software library.

Arbitrary precision computation with MPFR is about as close as it is possible to get to a concrete reference implementation of the real arithmetic used in rounded real-valued computation. The difference compared to real arithmetic is that MPFR cannot always produce exact real results, only rounded results with some finite amount of error, even if that error is arbitrarily small. To ensure that Titanic provides a correct implementation of

rounded reals, we will need a more sophisticated model of rounding to track that error and ensure that it never causes a wrong answer.

4.2 Representing rounding

Rounding numbers that have already been rounded is more difficult than rounding true reals because rounding multiple times can be dangerous. Let's construct an example to demonstrate this. Our example will address the following question: Does rounding a real number first to a higher precision number system, then to a lower precision one, always produce the same answer?

Let's use the common IEEE 754 32-bit single precision and 64-bit double precision floating-point formats for our example, with the strongest round to nearest even rounding mode. First, consider the number that is halfway between 1 and the next larger representable number in single precision. This number is exactly 1.00000005960464477539062 5, or $\frac{16777217}{16777216}$. This number is not exactly representable in single precision; if we round it, we should get one, since the significand of one is even. The number is exactly representable in double precision.

Now, consider some other number that is not more than halfway between this exactly representable double precision value and the next larger representable double precision value. For our example, we can use the number that is one quarter of the way between them. This new number is 1.000000059604644830901776231257827021181583404541015625, or $\frac{18014399583223809}{18014398509481984}$. This number is representable in neither double nor single precision. In single precision, it is larger than the halfway point between 1 and the next representable number, so it should round up. In double precision, however, it is close to the exact halfway point, and so it rounds down. If we round that halfway number again in single precision, we will break the tie down to 1 as before—a different answer than if we had rounded directly from the real value.

This answers the question: no, rounding to a higher precision number system first and then to a lower precision one does not always produce the same result as direct rounded

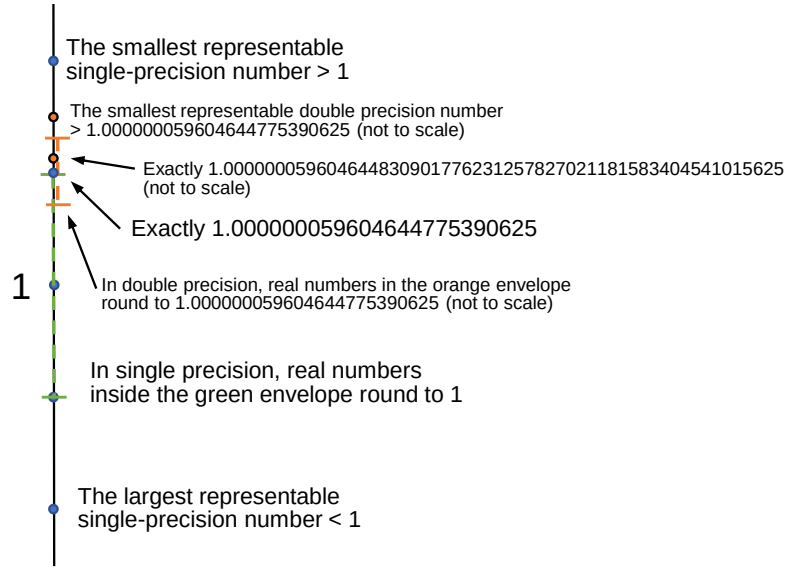


Figure 4.1: Rounding envelopes for example single and double precision values near 1. Note that the green and orange envelopes do not completely overlap.

real-value computation in the lower precision system. If we are to use (rounded) arbitrary precision values to implement number systems with Titanic, we will need some way to avoid these differences.

4.2.1 Rounding envelopes

To illustrate the example, we can draw it on a number line in Figure 4.1. The ranges of real values that round to 1 in single precision, and to our awkward halfway point value in double precision are shown on the number line, in green and orange respectively. We refer to these ranges of real numbers that all round to some common representable number as the *rounding envelope* of that number.

The problem, in short, is that the two rounding envelopes overlap partly but not completely. Specifically, the halfway point number is itself in the green rounding envelope because

the tie is broken down; but, some of the numbers in the orange envelope that round to it are outside of the green envelope. The orange envelope is ambiguous with respect to the green envelope. If we only know that some number was rounded to the halfway point, and therefore must be in the orange envelope, then we cannot say for sure if it should be in the green envelope or not.

Titanic tracks enough information about rounded numbers to accurately reconstruct the rounding envelopes. This means that for any two rounded numbers, it can trivially detect if one can be safely rounded to the other by checking if the second rounding envelope completely contains the first. Therefore, for any sequence of rounding operations, Titanic can ensure that the final result is still correctly rounded, as if the original real value had been rounded directly to the final number system. This capability is exactly what we need to ensure that rounded arbitrary precision remains correctly rounded.

4.2.2 Conditions for computing

To guarantee that computation is possible, i.e. that there is a sufficiently precise "compute" number system we can use as a stand-in for the real valued result, it must be the case that every rounding envelope in the compute number system is completely contained within a single envelope in the "target" or output number system. Fortunately, this is true for binary number systems that use all of the standard IEEE 754 rounding rules.

Formally, if we want to correctly round an output in some finite-precision number system with maximum precision p_0 and rounding mode rm_0, then we must find some p_1 and rm_1 such that if we compute in arbitrary precision p_1 and rm_1, then round to p_0 and rm_0, we will produce the same result as if we had computed in real precision and rounded to p_0 and rm_0 directly. This is true if each rounding envelope at p_1 and rm_1 is completely contained by an envelope at p_0 and rm_0, because rounding from the contained envelope to the containing one is always correct.

First let's consider a few cases where computation fails. If $p_1 < p_0$, then the rounding envelopes at p_1 will be larger than those at p_0, so there must be some envelopes that are not

completely contained, and computation will fail.

A more subtle problem occurs if rm_1 is any flavor of nearest rounding. Consider any boundary between b two rounding envelopes in p_0. For binary systems and IEEE 754 rounding modes, this b is exactly representable at $p_e = p_0 + 1$. If $p_1 < p_0$, then the compute envelopes are too large as before, and computation fails. If $p_1 > p_0$, then $p_1 \geq p_e$, and b is exactly representable at p_1. This means that there are some values greater than b that will round to b, and some values less than b that will round to b, since rm_1 is nearest rounding. But since b is a boundary at p_0, these values must be split between envelopes in p_0, and computation also fails.

Computation is only possible in this case when $p_1 = p_0$ and $rm_1 = rm_0$, which is to say, when the number systems are exactly the same and arbitrary precision computation is implementing rounded real valued computation directly, without any intermediate rounding steps. This completely defeats the purpose of Titanic, which is to use an existing arbitrary precision library that is correct for a limited set of compute number systems to provide answer for a wide variety of target number systems, which do not exactly match the arbitrary precision rounding semantics of the library.

Fortunately, it is easy to find p_1 to reproduce the rounding behavior of any IEEE 754 number system if we use the round towards zero mode (or simple truncation) instead of nearest rounding for rm_1.

Consider the case where rm_0 is not nearest rounding: it must be IEEE 754 round towards zero, towards positive, or towards negative. In all of these cases, the rounding envelope boundaries are all representable numbers at p_0, and each envelope only contains numbers rounded from one side. This is even true for zero with rm_0 as round to zero, because positive and negative values will be distinguishable by the sign of the rounded zero.

This means that for any $p_1 = p_0$, we will have exactly the same set of rounding envelopes as long as rm_1 is round towards zero, or more generally, any mode that is not nearest rounding. Switching between the rounding modes is just shifting the envelopes up or down by one. The only situation we have to be careful about is values that are exactly represented,

and the only extra work we need to do is identify them and not shift them up or down when changing rounding modes.

Additionally, we can use not just $p_1 = p_0$, but any $p_1 \geq p_0$. If rm_1 is round towards zero, then we know that all envelopes at p_1 and rm_1 are one-sided. If $p_1 \geq p_0$, then all envelope boundaries at p_0 are still representable at p_1. So, any envelope at p_1 must be completely enclosed by an envelope at p_0, as the endpoints of the enclosing p_0 envelope are representable at p_1, and because the p_1 envelopes are one-sided, no p_1 envelope spans across a representable number at p_1. Intuitively, smaller envelopes at larger p_1 will tile larger envelopes without inconveniently spanning across them.

Finally, consider a target number system p_0 with rm_0 as nearest rounding. Because rm_0 is nearest rounding, envelope boundaries in this system are representable at $p_e = p_0 + 1$.

Let $p_1 = p_e = p_0 + 1$, and rm_1 is round towards zero. Envelopes in the target number system span numbers representable at p_0, and also some numbers representable at p_1, since $p_1 > p_0$. However, while this would be a problem for the compute number system, it doesn't matter in the target system. Because rm_1 is round towards zero, we know that envelopes at p_1 are one sided, so they do not span across values at p_1. Consider an envelope at p_1; there must be some enclosing envelope at p_0 since the endpoints of all envelopes at p_0 are representable at $p_1 = p_e$, and the p_1 envelope cannot span across them. Intuitively, while the fact that they are not one-sided means that rounding envelopes in a round to nearest system do not tile nicely, they are tiled without overlap by any number system with greater precision that *does not* use nearest rounding.

4.2.3 Implementing number systems with rounding

Titanic provides the following facilities to implement number systems:

- An arbitrary precision representation for binary numbers, which tracks rounding envelopes

- Computation of elementary math operations with MPFR, to any precision p, including a safe rounding envelope for the output

- Safe rounding of any represented number to a number system with specified p, n, and IEEE 754 style rounding mode

With these operations, it is easy to implement arbitrary-precision versions of common number system such as IEEE 754 floating-point and general fixed-point. This is different from computing directly with MPFR. While MPFR does provide a direct implementation of many IEEE 754 floating-point number systems, it does not allow for mantissa sizes below 3 bits, and it does not explicitly track any of the guarantees about correct rounding. Titanic's explicit rounding envelopes allow values computed with MPFR to be used to safely implement other number systems as well, beyond just IEEE 754 floating-point.

Tracking envelopes can be done with only a few bits. In addition to the arbitrary precision value, stored in typical floating-point format with a separate sign bit, exponent, and mantissa, Titanic records the following properties about each represented number:

1. Whether or not the value was rounded

2. Which direction the value was rounded from

3. The size of the envelope

4. Whether or not the value is exactly on the endpoint of the interval

(1) is a single bit. An unrounded value must have been computed exactly, and can be rounded again to any precision. It will remain unrounded until some rounding operation changes its value.

(2) is another single bit, to record if the the true value was larger or smaller in magnitude than the represented, rounded value. For one-sided rounding modes (i.e. not nearest rounding) it will have a consistent sign, given the sign of the value.

(3) records the integer size of the rounding envelope, in -log(ulps). This becomes important for rounded numbers on exponent boundaries. As the exponent changes, the size of ulps changes as well; for values with the larger exponent, it is twice as large as it is for values with the smaller one. Keeping this information around explicitly allows for consistent safe rounding of values without knowing the number system in which they were initially rounded.

(4) is another bit. If it is true, then the rounded value is actually exact, and fell exactly on the boundary between this rounding envelope and another adjacent one. This information is necessary to ensure that nearest rounding can correctly break ties.

Titanic's safe rounding capability is provided as a function that takes any represented value and rounds it according to some precision p and some greatest unrepresentable binary place n. While these measures of precision might seem to be independent, they are actually related. Rounding can provide only p, which resembles floating-point rounding; only n, which resembles fixed-point rounding; or both, which resembles IEEE 754 rounding with subnormals.

The limiting binary place for any number with exponent e can be determined based on optional p and n. This is the absolute place beyond which less significant digits will be rounded off. Floating-point rounding for some fixed exponent is, after all, just a special case of fixed-point rounding. If n is supplied, then it gives a lower bound on the limiting place. If p is also provided, then the limiting place is the larger of n or $e - p$.

Reducing to fixed-point rounding in terms of this limiting position allows Titanic to provide safe, general fixed-point and floating-point rounding behavior with full support to model subnormals, all with a single unified implementation. We say that Titanic's rounding is safe because attempting to round to a large amount of precision, or a smaller value of n (such that the output rounding envelope would not fully enclose the input envelope) will raise an exception. This gives number system implementers a good starting point to build correct rounding models for various number systems.

Because of its dependence on MPFR, Titanic is limited to binary number systems, where each exactly representable number has some finite representation as a binary fraction. Binary

number systems tend to be very convenient, due to easy mapping to hardware and parameterization with p and n in addition to computation with MPFR, but there is no obstacle in theory to building safely rounded number systems in other bases in terms of rounding envelopes, or (say) number systems that could exactly represent rational numbers such as $\frac{1}{3}$.

4.3 Using Titanic

In addition to rounding behavior, Titanic also provides a full parser and interpreter for the control layer of the FPCore language. As this is a typical functional programming language implementation, it represents a significant engineering effort but not a theoretical research contribution of this work.

By making the semantics of FPCore executable, Titanic enables a wealth of new research projects. The implementation is built entirely in Python 3 (with the stated dependence on MPFR, which is accessed from Python through the gmpy2 binding library [24]), which makes it easy to extend and instrument.

The full implementation of Titanic is open source an is available on GitHub at `https://github.com/billzorn/titanic`.

4.3.1 An extensible framework for number systems

Providing the core rounding operations as a library, as well as the integrated computational capabilities with MPFR and the general design of the system in Python, means that actual number system implementations are a very small part of the overall codebase. The rounding logic required to emulate arbitrary precision IEEE 754 number systems fits in 15 lines of code, and arbitrary precision fixed-point only requires 22 lines. These number systems are provided with Titanic as additional libraries, and allow it to run all FPCore programs annotated with standard fixed-point and floating-point number systems.

Titanic's provided rounding library only implements ideal rounding, which does not impose an upper bound on the exponent, or in the case of fixed-point, the maximum number of bits in the representation. These limits must be implemented manually where appropriate,

which is why IEEE 754 takes 15 lines to implement rather than a single function call to the general rounding function. Other number systems, specifically posits, are not as easy to implement in terms of general rounding with p and n. Titanic also provides an implementation of rounding for posits, which is about 150 lines long.

Comparison between values is built in to the arbitrary precision representation itself, so values that have been rounded in different number systems can be compared to each other with no additional effort. In the worst case, rounding can even be implemented for any ordered number system via binary search. To round some value, pick some member of the target representation, find the endpoints of the envelope as arbitrary precision values, and simply check if the value to be rounded fits in the envelope. If not, binary search in the appropriate direction.

Titanic does not provide support for number systems that do not use a binary representation. This means that some error will be introduced, even at arbitrarily large precision, for computations that can be computed exactly in a different base (for example, decimal calculations over dollars). In practice, most large-scale numerical computations use binary IEEE 754 floating-point or other binary number systems. Binary integers and fixed-point, IEEE 754 floating-point, and posits, including quires, are all fully binary number systems, and Titanic makes it easy to implement all of them and even have them interoperate.

4.3.2 Instrumenting computations

As it is not a research contribution, we have largely glossed over the way Titanic implements an FPCore interpreter. At a high level, the tool uses an ANTLR4 [34] to parse FPCore syntax into an AST, and then it executes the AST directly with a recursive virtual machine, implemented in Python.

The simple structure of the virtual machine makes it very easy to instrument FPCore computations, in the number system or elsewhere. Each executed AST node reports its inputs and output values after executing, and this information can be captured by analysis hooks, without changing the code of the interpreter itself.

The shared arbitrary precision representation is also fully available as a library, to perform computations or store values outside the FPCore interpreter. The usual way to implement a number system is to define a new number type, which inherits all of its math operations from the shared type, and then override the rounding behavior with a custom implementation for the new system. Computing with the number types works like any other shallowly embedded language in Python; numbers are Python objects that provides methods like `add`, `sub` and `sqrt` to perform computations with them.

4.3.3 Performance considerations

Titanic is not intended for *production* workloads, but it is fast enough for research. To give a rough sense of scale, creating the Lorenz system plot for Figure 1.1 runs about 105 million FPCore AST nodes through the Titanic interpreter, using MPFR with 4096 bits of precision, and takes about 15 minutes on one thread of a Ryzen R9 5950X. A visually identical plot can be produced in Python in a fraction of a second, and a C implementation could reduce that by another factor of 100; of course, neither of these implementations would guarantee correctly-rounded 4096-bit intermediate values. Waiting 15 minutes for the plot is perfectly acceptable for the purposes of this research, where we are more concerned with doing it once, correctly, than with being able to rerun or modify it in real time.

In cases where performance is a key consideration, which is to say almost all numerical computing workloads in the real world, whether they are high-precision scientific computations or low precision machine learning, rounding is not the right way to think about performing computation. In those cases, traditional hardware and software libraries, made to be as efficient as possible, while not being too inaccurate to produce the necessary results, are absolutely the way to go.

In performing research, however, we often want to answer questions about systems that don't exist yet, so that we can make informed decision and invest in their future existence. Titanic is performant enough to reproduce the behavior of interesting and representative workloads, like our Lorenz system example, in order to support this kind of research. In

exchange for more expensive computation, the framework greatly reduces the upfront cost of prototype implementation.

Titanic can also serve as a reference interpreter to check more optimized implementations of number systems. Because its correctness (and the correctness of any number system implemented on top of it) is entirely determined by rounding behavior, the amount of code that needs to be correct will always be small.

4.4 Future work

Titanic is still actively in development. Many improvements could be made to broaden the capabilities of the system and enable further research.

In its current form, Titanic (and also the FPCore language) does not have a formal interface to translate between binary representations and real values. It is understood that the rounded real values stored as intermediates in FPCore programs will have to be stored in bits somewere, but is never made explicit how that is done.

One avenue of future work is to formalize this interface. This could be as simple as defining a binary encoding function, which maps bitvectors to real values. Between rounding and encoding, algorithms can be carried out as rounded real-valued computations, pure bitvector algorithms, or a mixture of the two. This would be especially useful to model the behavior of custom hardware that defies a more abstract mathematical description of its numerical behavior.

Encoding can also be used to implement rounding via binary search, which would make it easy to work with number systems, particularly the bisection based formulations described by Lindstrom [29], that admit simple encoding functions but not obvious implementations of rounding.

Another avenue of future work is to enhance Titanic's representation of rounding envelopes to interact with proper interval arithmetic. As described here, Titanic implements interval arithmetic, but only for the round operation, not any other mathematical function. However there is nothing in principle stopping Titanic from using an interval arithmetic

library to do computation instead of MPFR.

This would allow for Titanic to correctly implement some FPCore programs that specify `real` precision. Specifically, as long as the region of the program done in `real` precision had bounded size (i.e. no loops), then Titanic could, with some amount of computation and an interval arithmetic library, provide an output that satisfied any rounding specification for the whole real expression. This would allow Titanic to model custom fused operations, such as `expm1` (but for other real functions), among other things.

Chapter 5

SINKING-POINT: A NUMBER SYSTEM WITH SAFER ROUNDING

5.1 The hidden perils of rounding

So far, we have viewed rounding as a good thing. It provides an interface to model many different number systems in terms of real numbers, and even lets us implement them easily on top of existing arbitrary precision libraries.

But rounding error in finite precision number systems is often a problem. It 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 real arithmetic, we would expect the large terms of 10^{16} to cancel out, leaving π as the result. Instead, we get 4. This is not a problem with Python's math library: if we just type in `math.pi`, the interpreter prints 3.141592653589793, or about 16 decimal digits of precision, as one would expect from a 64-bit double precision value. 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 in this case: 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. 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.

5.1.1 Can a number system do better?

To address this problem, we introduce a new number system which we call 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. We can think of these ranges as a concrete manifestation of the rounding envelope, translated into decimal notation for human consumption. Here, the 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. This is entirely expected as rounding envelopes only capture the uncertainty of the most recently performed operation, not the total error that has accumulated through the computation.

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 rounding envelope 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.

5.2 Sinking-point

Sinking-point is based upon the following observation: when a floating-point operation causes a loss of precision, that loss of precision is often immediately obvious. 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.

5.2.1 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.

5.2.2 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 sig-

		1	0	1	.	0	1	?	?	?	?
+		1	0	0	.	0	0	0	0	0	1
	1	0	0	1	.	0	1	0	0	0	1
	1	0	0	1	.	0	1				

Figure 5.1: 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.

nificant 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 5.1 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 5.2 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

	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 5.2: Binary visualization of $5.25 - 4.015625$ computed with sinking-point

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 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.

5.2.3 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 5.3 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^{0}s place, but in the output, the least significant known bit is in the 2^{2}s 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

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

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.

5.3 Implementing sinking-point

To evaluate the capabilities of sinking-point, we built a prototype implementation using Titanic. The source is distributed as part of the Titanic project on GitHub. 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.

5.3.1 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 are familiar from our discussion of rounding. p is 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 5.2.1, it is the position of the leftmost question mark. For a number with a typical IEEE 754 floating-point exponent equal to e and precision p, we can define $n = e - p$. With sinking-point, p and n can change for individual values based on how they have been rounded, rather than being fixed for the entire number system as they are for IEEE 754.

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. In Titanic, we can simply add it to the metadata recording the rounding envelope. 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.

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 5.1.1, 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 nearest even rounding 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 decimal digits such that both the value and precision can be recovered.

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.

5.3.2 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, as Titanic provides this capability for us via MPFR. We also assume the existence of a rounding function with the following signature:

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

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 5.1: Summary of rules for computing sinking-point output precision

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 5.1 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.

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.

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.

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}.

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.

5.4 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.

5.4.1 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

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

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 5.2: 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 5.3: Results for herbified quadratic formula with a close to zero.

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 5.2, 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 $a = 10^{-16}$ 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 [33]:

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

Results for this computation are shown in table 5.3. 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.

5.4.2 Accuracy on a 32-bit budget, adapted

In [21], 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:

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

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.

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

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." [21] 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 5.4. 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.

5.5 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 [22], fixed-point representations, or other application-specific floating-point designs [25]. 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.

5.6 Future work

Sinking-point exposes some problematic rounding behaviors that commonly occur in finite-precision number systems but that are often hidden by implementations like the IEEE 754 standard. While the analysis is not a sound guarantee, it is computationally cheap to compute and it scales to nontrivial computations that are too large to provide sound error bounds for with naïve interval arithmetic.

By exposing more information about the precision of values, not just the values themselves, sinking-point allows for an entirely new programming model. Precision aware programs could be implemented by extracting this information directly from the values, rather than requiring the programmer to track a separate precision or error estimate. In the simplest form, this could be used to check comparisons, and perhaps raise an exception if the outcome of the comparison would be unclear due to a lack of precision.

There are several ways to tighten the analysis. For repeated additions, such as would occur in a dot product, sinking-point could analyze all of the additions together and provide sound p and n values for the entire sum, rather than individually for each operation. This can be done dynamically, without identifying the sum in advance, by recording a bit more metadata (the number of consecutive additions).

The analysis can also be extended to other math operations. Though we have not presented a formal analysis here, the sinking-point rules are related to the condition numbers of

the functions in question. This explains why the square root allows for additional precision in the output; the condition number is $\frac{1}{2}$.

We also speculate that sinking-point could be implemented efficiently in hardware. While this would not have any performance advantages over the host number system, it would enable new programming paradigms for precision-aware algorithms, particularly because sinking-point style precision information can be transferred between entirely different number systems, as long as they have some notion of binary precision.

Sinking-point demonstrates that there is more to a number system than representing as many numbers with as few bits as possible, and making math operations fast. It is also a good showcase for the capabilities of Titanic, and the ease with which it can be extended to model number systems beyond IEEE 754.

Chapter 6

QUANTIFIND: EXPLORING APPLICATION-SPECIFIC NUMBER SYSTEMS

Titanic gives us the ability to run FPCore programs with a wide variety of number system. With sinking-point, we have used that flexibility to create a new number system that helps improve numerical correctness.

But what about performance? Most computations today are implemented using a single universal number system, or at best a few related number systems, such as single and double precision floating point. What if instead, we could choose the right amount of precision for each operation to build a fully customized number system for a specific application? How many bits can we save, without sacrificing the correctness of the application?

To explore this question, we developed QuantiFind, a prototype tool for tuning *application-specific number systems*. Unlike a universal number system, an application-specific number system is co-designed with a particular application or algorithm in mind and embedded into its implementation.

QuantiFind takes as input a mathematical description of an application, annotated with optimization sites where the behavior of the number system can be controlled. By experimentally observing the behavior of the application, QuantiFind searches for application-specific number system configurations, filling the annotation sites with concrete number system candidates. The output is a Pareto frontier of these configurations, each leading to an implementation with some Pareto-optimal combination of observable metrics configured per application to represent cost or output quality.

To evaluate QuantiFind, we detail the behavior of two representative applications, and explore application-specific number system configurations comprising either IEEE 754 floating-

point with varying numbers of bits, or various configurations of posits [21]. The search procedure can generalize to other number systems as well; our metrics are general enough to compare floating-point and posit configurations directly.

QuantiFind provides a Pareto frontier of the optimal tradeoffs between accuracy and cost discovered during its search, rather than a single "best" implementation. The gaps between our experimental frontiers and a mixed-precision baseline are considerable, amounting to approximately a 2× quality improvement for a given bitcost, or the same quality at roughly half the bitcost, depending on which part of the frontier is most important for the given application.

Components of QuantiFind

The prototype QuantFind tool is built from the following major components:

- A representation for application-specific number systems in terms of the existing FPCore language and Titanic interpreter (Section 6.1).

- An abstract *bitcost metric* (Section 6.2).

- A hill climbing search procedure (Section 6.3).

In Sections 6.4 and 6.5, we discuss the results of using QuantiFind for two applications.

The purpose of QuantiFind is not to prove that any of the configurations we explore can meet accuracy guarantees, nor is it to directly design efficient hardware for them. These are well-known, difficult problems, and other approaches can help address them. The purpose of QuantiFind is to *find* these configurations within a massive search space; the purpose of the prototype tool is to show that exploration can be done at a high level, purely in software "doing the simplest thing possible," and thus opening many avenues for future work.

6.1 Representing configurations

In order to search for application-specific number system configurations of any kind, we first need a way to describe what an application-specific number system is. Fortunately, FPCore makes this easy. We can specify the number system for each individual operation in an FPCore program, so an application-specific number system is just an FPCore program that makes use of this capability.

In fact, we have already presented such a program, in Figures 3.11, 3.12, and 3.13. As a running example, we will return to our FPCore implementation of the Lorenz system with RK4. The orange highlighted precision annotations define an application-specific number system found by QuantiFind works particularly well for our initial conditions.

To create a template for optimization, we replace the concrete precision annotations with optimization sites that are filled in automatically by the tool. As this is not a feature of FPCore, we fill in the optimization sites for each configuration to explore with a simple Python string formatting function.

6.2 Evaluating configurations

To find interesting configurations, we also need some way to quantify their impact on application quality, performance, and cost, relative to each other and to more traditional baselines. A simple way to achieve this is to execute the FPCores directly with representative inputs and measure the results.

6.2.1 Quantifying application quality

To determine the behavior of a particular configuration, we can simply run it with Titanic. This will tell us what the application-specific number system does, but not how good it is. To measure that, we need to develop some quality metric for the output of the application.

In most cases, this can be done by looking at the accuracy of the result, or using a domain-specific quality metric. For our RK4 example, we obtain a correct reference solution

by running the algorithm with a significantly smaller step size (16 times more steps) and a very precise number system (double precision floating-point is more than precise enough in this case). We then compare the final values of the x, y, and z coordinates of the simulated result from each application-specific configuration to this reference, and calculate the bits of accuracy. "Bits of accuracy" (similar to decimals of accuracy in [21]) measure the number of bits that are in agreement between two values a and b, and can be calculated with the formula:

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

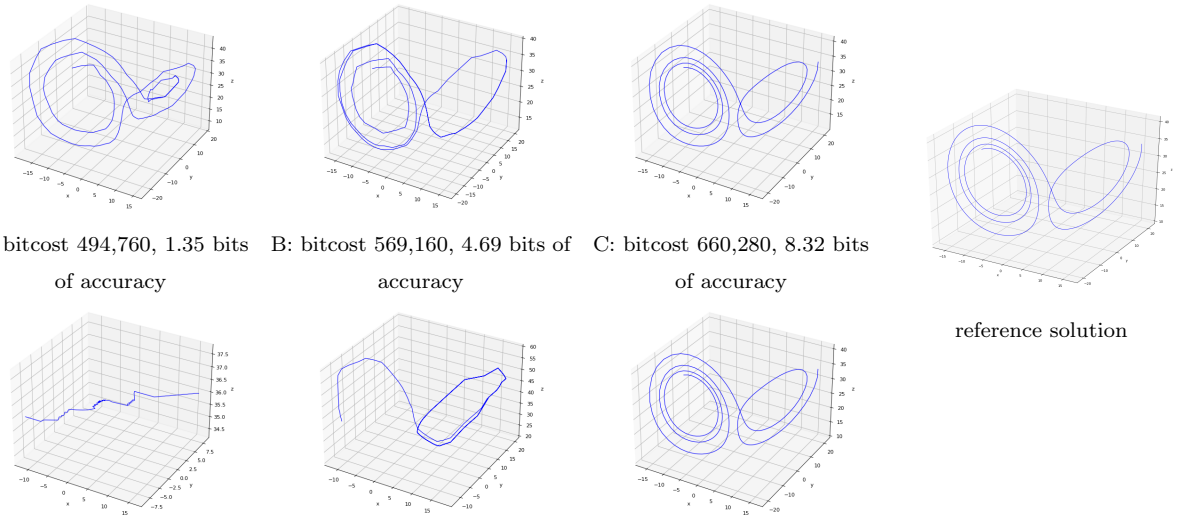
Numerical algorithms are approximate by definition, so in most domains comparing the quality of different outputs is well understood and a suitable quality metric can be found or created. Titanic handles the more difficult task of simulating the application-specific number systems, though of course it has limited performance compared to, say, IEEE 754 implementations in hardware.

Like sinking-point, the implementation of QuantiFind is available on GitHub as part of Titanic.

6.2.2 Quantifying performance: Bitcost

Titanic can simulate the functional impact different number systems have on application quality, but it is a very poor proxy for real application performance. Ideally, we would like to use the performance and power consumption of the most efficient hardware design we can imagine for our cost metric, but this is intractable. We have too many configurations to explore, and in any case hardware design depends on details that might not yet be known while conducting the search.

To break the potential bootstrapping problem, we use a high-level cost metric which we refer to as "bitcost." The bitcost of a computation is the sum of the sizes of the representations of all numbers which are used as inputs to mathematical operations. To give an



A: bitcost 494,760, 1.35 bits of accuracy

B: bitcost 569,160, 4.69 bits of accuracy

C: bitcost 660,280, 8.32 bits of accuracy

reference solution

D: bitcost 379,800, 3.26 bits of accuracy

E: bitcost 464,520, 3.27 bits of accuracy

F: bitcost 680,860, 7.96 bits of accuracy

Figure 6.1: Selected configurations for RK4 solver, with bitcost and accuracy. A, B, and C use custom IEEE 754 floating-point number system configurations; D, E, and F use posits. Full configuration parameters in Table 6.1

example, computing the x coordinate of the Lorenz equation `(* sigma (- y x))` in some `(float exp? fn?)` rounding context will depend on the value chosen for the parameter `fn?` in this configuration, as well as the value chosen for `rk?`.

The bitcost of the multiply is 2`fn?`, since the constant σ and the result of the subtraction are both rounded under the enclosing context with `fn?` total bits. The bitcost of the subtraction is 2`rk?`, as the values of x and y were last rounded in the `rk?`-bit context. The bitcost for the whole subexpression is then 2`fn?` + 2`fn?`; this might vary for the first iteration, depending on the representation used for the `initial-conditions` argument. Bitcost is tracked dynamically by the simulator across all operations as the FPCore program executes.

Bitcost is by no means a complete description of hardware performance characteristics. It ignores the widely varying costs of different mathematical operations, as well as other microarchitectural nuances like caches and the memory hierarchy. For the purposes of the

QuantiFind prototype, however, bitcost is useful for determining the relative potential of different configurations. Sending fewer bits to operations is almost never bad, all other things being equal, and the dynamic analysis can easily be extended, for example to measure operation costs, if that information is available. Most importantly, bitcost in practice is able to differentiate different configurations enough that a Pareto frontier can form.

Figure 6.1 shows plots of the output of several sample configurations of the RK4 solver run from the point ($x = -12$, $y = -17/2$, $z = 35$), using a step size of 1/64 for 240 steps. Configurations A-C use floating-point, while D-F are various configurations of posits. Taken together they illustrate the different tradeoffs between bitcost and accuracy we expect to find. To explore more fully, we need an algorithm to conduct a systematic search.

6.3 Searching the configuration space

The space of configurations for an application-specific number system tends to be large. For our RK4 solver example, we have decided on 6 annotations sites to control the rounding behavior. At each site we let the precision of the floating-point number system vary from 3 to 24 bits; combined with a global exponent size which we allow to range between 2 and 8 bits, this gives us 7 parameters to tune and almost 800 million configurations to explore.

In theory, one could explore this space exhaustively, but only with immense patience, a supercomputer, or a much faster number system simulator. Fortunately, the configuration space has two useful properties that we can leverage in a search procedure that can be scaled to larger applications.

6.3.1 The space of application-specific number systems

The first useful property of the space is its discreteness. Each configuration is a list of 7 small integers; in general, we expect that most application specific number systems could be parameterized in a similar way. This suggests some classic techniques from AI that might be applicable, in particular genetic algorithms [20].

To state our search problem as a genetic algorithm, we simply make the list of 7 integer

parameters the genome. It is straightforward to generate random candidates (simply by taking random parameters) and to create mutations (randomize a subset of the parameters). Crossover can be done with any standard algorithm (for example, random parameter exchange between pairs of configurations). To limit the population after each generation, we can measure properties of each configuration such as cost and output quality, and only keep configurations that are on the Pareto frontier.

The second useful property is that the search space is ordered, at least in a global sense. In general, we expect configurations with more bits to be more accurate and also more costly, while configurations with fewer bits are less accurate but cheaper. Locally, these expectations will tend to break down: sometimes, rounding in just the right place might increase the accuracy of a configuration, by counteracting some of the algorithmic error with quantization error in the other direction.

The search space is not completely smooth or convex, but it is "smooth enough" in practice that exhaustively exploring points within a small radius of a given configuration avoids many local minima and efficiently maps out the Pareto frontier.

6.3.2 Search algorithm

Our prototype search algorithm to find the Pareto frontier of number system configurations for an application currently follows a modified hill climbing approach. Like a genetic algorithm, we start with an initial population of randomly selected configurations. Our search is not sensitive to the size of this initial population; in practice, even a single initial configuration seems to give good results.

Then, instead of mutation and crossover, we perform local search for each generation, exhaustively exploring every configuration that is near some configuration on the current Pareto frontier. Nearness is determined by adjusting each parameter individually up or down by a small amount (2 or 3 seems effective for avoiding local minima in practice), and similarly adjusting all of the parameters up and down together. After each generation, we limit the population to the current Pareto frontier; if it has not changed, then we either

	exp?	fn?	rk?	k1?	k2?	k3?	k4?	bitcost	position accuracy	slope accuracy
A	5	3	5	3	3	3	4	494760	1.35	-inf
B	5	7	5	5	5	6	6	569160	4.69	2.96
C	5	11	9	8	5	7	4	660280	8.32	7.01
D	2	3	10	4	4	3	4	379800	3.26	-inf
E	2	6	10	5	11	4	9	464520	3.27	0.13
F	1	11	17	8	15	14	14	680860	7.96	4.50

Table 6.1: Selected configurations for RK4

explore additional completely random points or conclude that search has finished.

In our experiments, this search saturates the frontier after looking at a few tens of thousands of configurations from the 800 million in the search space, and produces a much better Pareto frontier than looking at a similar number of completely random configurations. As noted with the bitcost metric, it is almost certainly possible to design a better search procedure, for example by using more sophisticated genetic algorithm technology like crossover. However, the existence of a better search procedure does not detract from our existing exploratory results.

6.4 Results - RK4

Figure 6.2 shows our results for the RK4 experiment, presented as Pareto frontiers. Up and to the left is better. Our key finding is the significant gap between the application-specific number systems found by the search (higher and further left; blue and orange) and the black mixed-precision baselines.

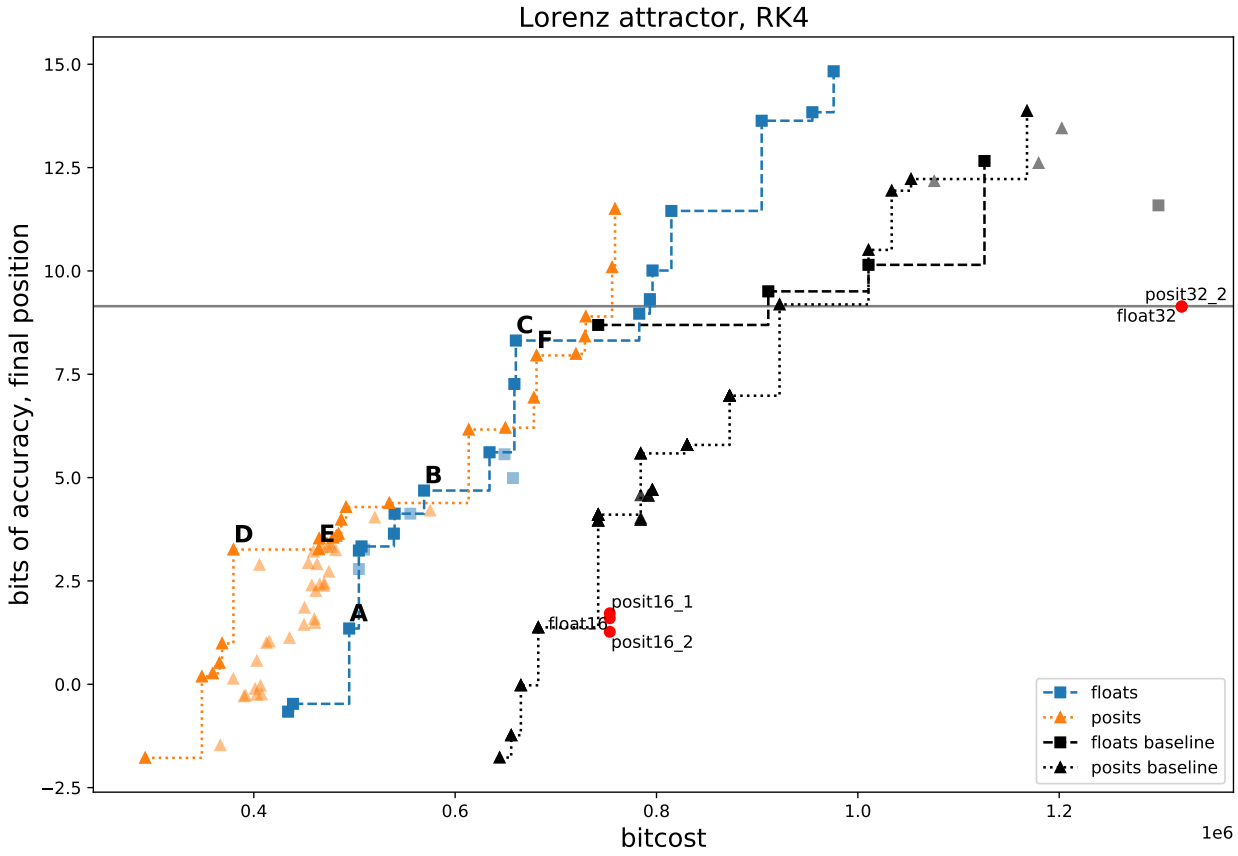


Figure 6.2: Pareto frontier for RK4 experiment

6.4.1 Experimental Pareto frontiers

The plot in Figure 6.2 shows four different Pareto frontiers, originating from two independent experiments. We use QuantiFind to conduct the application-specific number system search twice, using two different classes of number systems: IEEE 754-like floating-point, which we have discussed through our running example, and also posits. For posits, we use as our parameters the number of total bits in the representation (instead of significand bits) at each of the annotations sites, with one additional parameter for the number of exponent bits (which work differently from a traditional floating-point exponent [21]). Total bits range from 3 to 24, while the exponent bits range from 0 to 2.

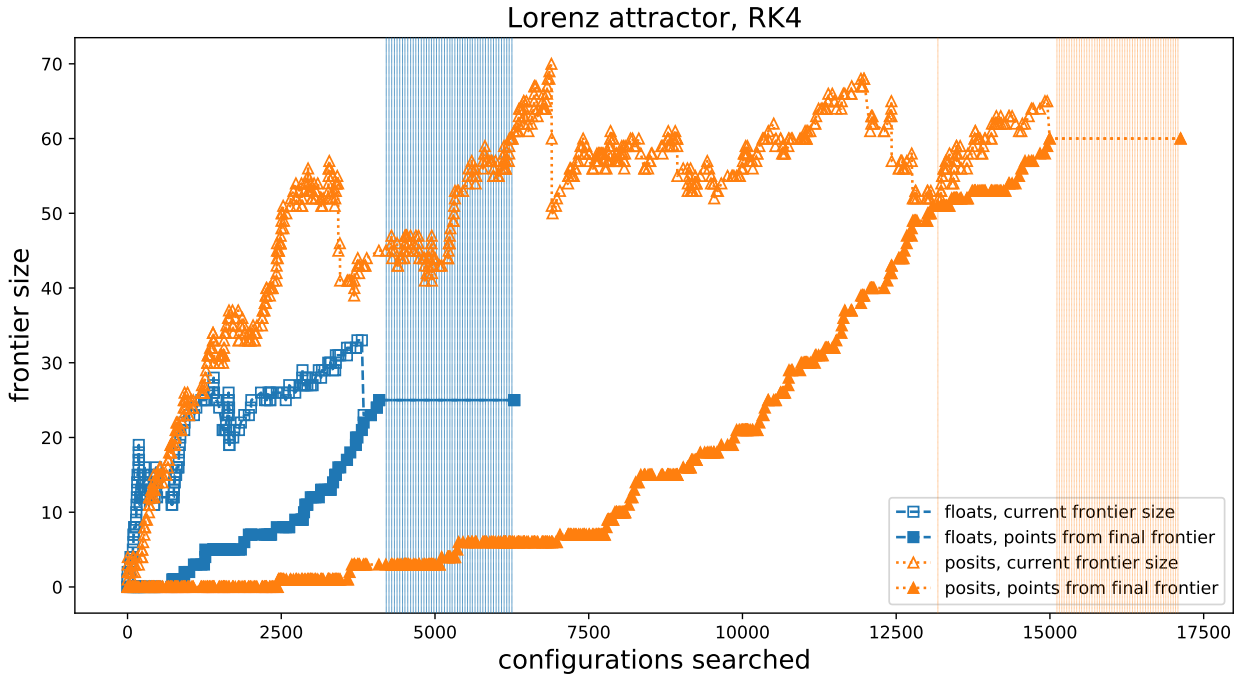


Figure 6.3: Size of Pareto frontier during search for RK4

Our search can find a Pareto frontier with any number of quality and cost metrics; for the RK4 experiment, we use a total of 5. To make the results easier to visualize, we project this down to the two most relevant metrics: bitcost on the horizontal axis, and bits of accuracy of the final position of the simulation on the vertical axis, averaged across the three dimensions. The final, application-specific configuration frontiers are shown in blue squares for the floating-point experiment and orange triangles for posits. Points on the true 5-metric frontiers but not on the 2-metric projection are shown at reduced opacity.

The labeled points A-F correspond to the configurations plotted in Figure 6.1. Parameters for these six configurations are given in Table 6.1.

6.4.2 Mixed-precision baselines

As a baseline, we find a mixed-precision reference frontier for each of our experiments by exhaustively searching over a set of four universal number systems at each of the six optimization sites. For the floating-point version of the experiment, these four number systems are 32-bit single-precision floats, float16 (5 exponent bits), bfloat16 (8 exponent bits), and a hypothetical 8-bit floating-point format with 3 exponent bits. This reference frontier is shown as black squares. For the posit experiment, we use 32-bit posits with 2 posit exponent bits, 16 bits with 1 or 2 exponent bits, and 8 bits with 0 exponent bits. The posit reference frontier is shown in black triangles.

In addition to the baselines, we also plot red fenceposts for uniform-precision configurations using each of the baseline number systems.

The reference frontiers are valid application-specific number system configurations; since they only use a few number systems, they could in theory be run on hardware today if ALUs were available that supported those number systems. As far as baselines go, they are somewhat optimistic, as ALUs for 8 and 16-bit floating-point operations, let alone posits, are not widely available in mainstream processors.

However, we think these are the right baselines to compare against, because they illustrate that mixed-precision tuning of a few types does not have all of the benefits of tuning every bit in an application-specific number system.

6.4.3 Algorithmic accuracy ceiling and overspecialization

The grey horizontal line on the plot, drawn at about 9 bits of accuracy, shows the accuracy ceiling we expect for the application based on the numerical error from the algorithm. As expected, the uniform-precision 32-bit fenceposts are right on this line, as the 32-bit number systems are easily precise enough to reproduce the behavior of real numbers for this application.

Perplexingly, the Pareto frontiers are able to push above this accuracy ceiling. This

is because we test by running a single configuration, and compare to a reference result computed with a more accurate algorithm. By tweaking the rounding behavior just so, the search procedure can come up with ways to compensate for the error of the algorithm by supplying the right counter-error in the number system. With only one test input, it is relatively easy to overspecialize to it.

The far leftmost point on the floating-point reference frontier is caused by the same effect. By truncating the coefficients in just the right way, it can achieve an accuracy and cost tradeoff comparable to the application-specific configurations.

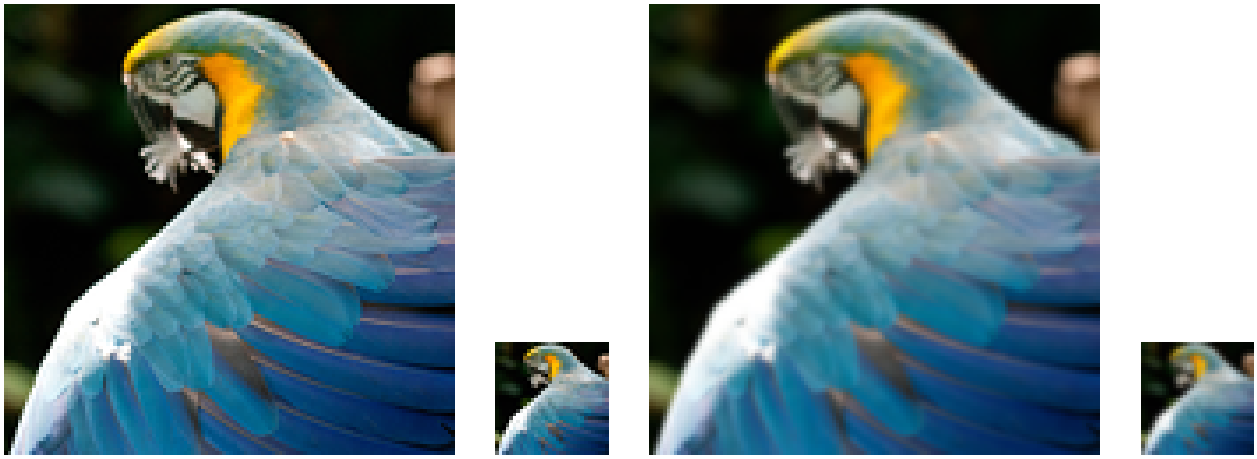
While this behavior illustrates the power of the search algorithm, it is neither desirable nor an intended goal for the search. Overspecialization wastes the search algorithm's time and pollutes the output with configurations that may be too brittle to be useful in the real world. Fortunately, we can mitigate this challenge in a variety of ways. QuantiFind is designed to produce output for human inspection; our RK4 application is simple enough that we can identify most of the overspecialized configurations by drawing the accuracy ceiling on the graph, and rule them out by manual inspection. Users can also sample a larger number of inputs, or manually select inputs representative of different application-specific scenarios.

6.4.4 Progress of the search

Independent of its results, it is also important to understand the behavior of the search algorithm. One simple way to do this is by plotting the size of the current Pareto frontier over the course of the search, as in Figure 6.3.

As with the Pareto frontiers, the blue points correspond to the floating-point experiment, while the orange points are posits. Open points track the size of the current frontier, and solid ones show the number of points from the final frontier that have been discovered. Vertical lines show random restarts, after local search fails to expand the frontier.

Figure 6.3 confirms the expected behavior of QuantiFind's search. The size of the frontier goes up and down as new, good configurations are found; it does not seem to explode in an unmanageable way. Interesting points from the final frontier are discovered steadily



(a) input image (b) (c) reference output (d)

Figure 6.4: Test image used for blur. Images a and c enlarged for manual inspection; b and d are the true size used for the search.

throughout the search, so the local search heuristic doesn't seem to be wasting too much time. Random restarts don't add much to the frontier, but confirm that the local search isn't getting stuck; only one restart from the posit experiment led to any new exploration.

6.5 Results - blur

The RK4 solver is a useful example for understanding QuantiFind, but it isn't the most attractive target for low-precision, application-specific number system tuning. To show that QuantiFind can be used in other domains, we also find interesting configurations for an image processing application: blur.

6.5.1 3x3 box blur

Our image processing algorithm is a simple box blur [36], using the following 3x3 pixel mask:

```
(FPCore fastblur-mask-3x3 ((img rows cols channels) (mask 3 3))
 :precision (float 5 10) :round nearestEven
 (let ([ymax (! :titanic-analysis skip (# (- rows 1)))]
       [xmax (! :titanic-analysis skip (# (- cols 1)))])
  (tensor ([y rows]
           [x cols])
   (for* ([my (# 3)]
          [mx (# 3)])
    ([y* 0 (! :titanic-analysis skip (# (+ y (- my 1))))]
     [x* 0 (! :titanic-analysis skip (# (+ x (- mx 1))))]
     [in-bounds? FALSE (! :titanic-analysis skip (and (<= 0 y* ymax) (<= 0 x* xmax)))]
     [mw 0 (if in-bounds? (! :precision (float 5 10) :round nearestEven (+ mw (ref mask my
         mx))) mw)]
     [w (tensor ([c channels]) 0)
        (if in-bounds?
            (tensor ([c channels])
              (! :precision (float 5 12) :round nearestEven (+ (ref w c)
                (! :precision (float 5 9) :round nearestEven (* (ref mask my mx) (ref img y*
                     x* c)))))
            )
            w)])
    (tensor ([c channels]) (/ (ref w c) mw))))
 ))
```

Figure 6.5: Box blur kernel in FPCore, configuration E

$$\begin{matrix} \frac{1}{3} & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{3}{2} & \frac{2}{3} \\ \frac{1}{3} & \frac{1}{2} & \frac{1}{3} \end{matrix} \tag{6.1}$$

For each pixel in the output image, we take weighted averages of nearby pixels from the input image, given by the mask. The weights are not uniform to exacerbate the effects of rounding.

We insert annotations at 4 points in the computation: accumulating the in-bounds mask weights, multiplying the pixel by the mask weight, accumulating the weighted pixel values, and overall precision, which corresponds to the division of the weighted pixel values by the sum of the mask weights. Like the RK4 experiment, we add a global exponent size as a fifth parameter. Each precision is allowed to range from 3 to 16 bits, while the exponent ranges from 2 to 8. We also perform a posit version of this experiment, using 3 to 16 total bits with 0 to 2 being reserved for the exponent.

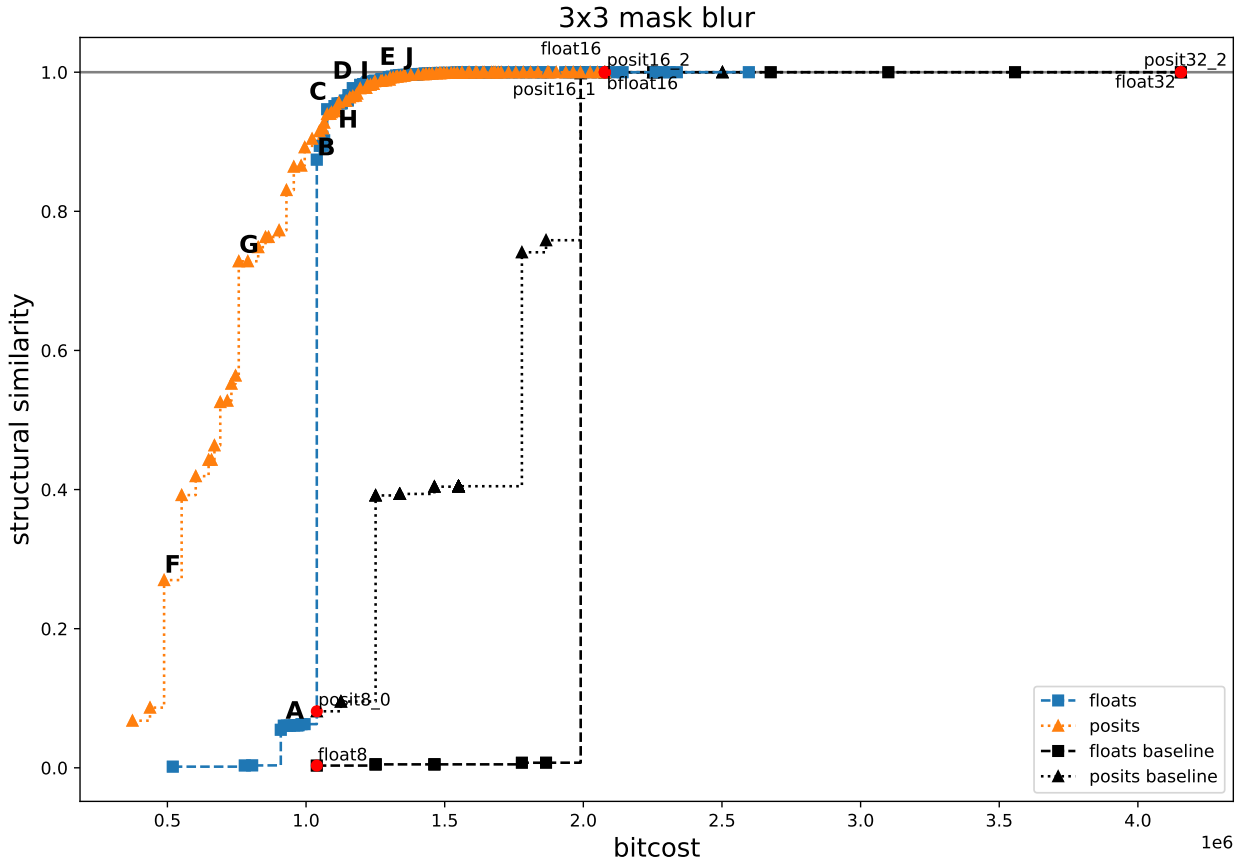


Figure 6.6: Pareto frontier for blur experiment

For input, we borrow an image from Halide [36] and resize part of it (shown in Figure 6.4) to 32 by 32 pixels. To measure the quality of outputs, we use structural similarity compared to a reference output computed with very high (64-bit) precision everywhere.

A complete FPCore implementation of the blur kernel is given in Figure 6.5. The precision annotations correspond to a single explored point, represented by configuration E in Figre 6.9.

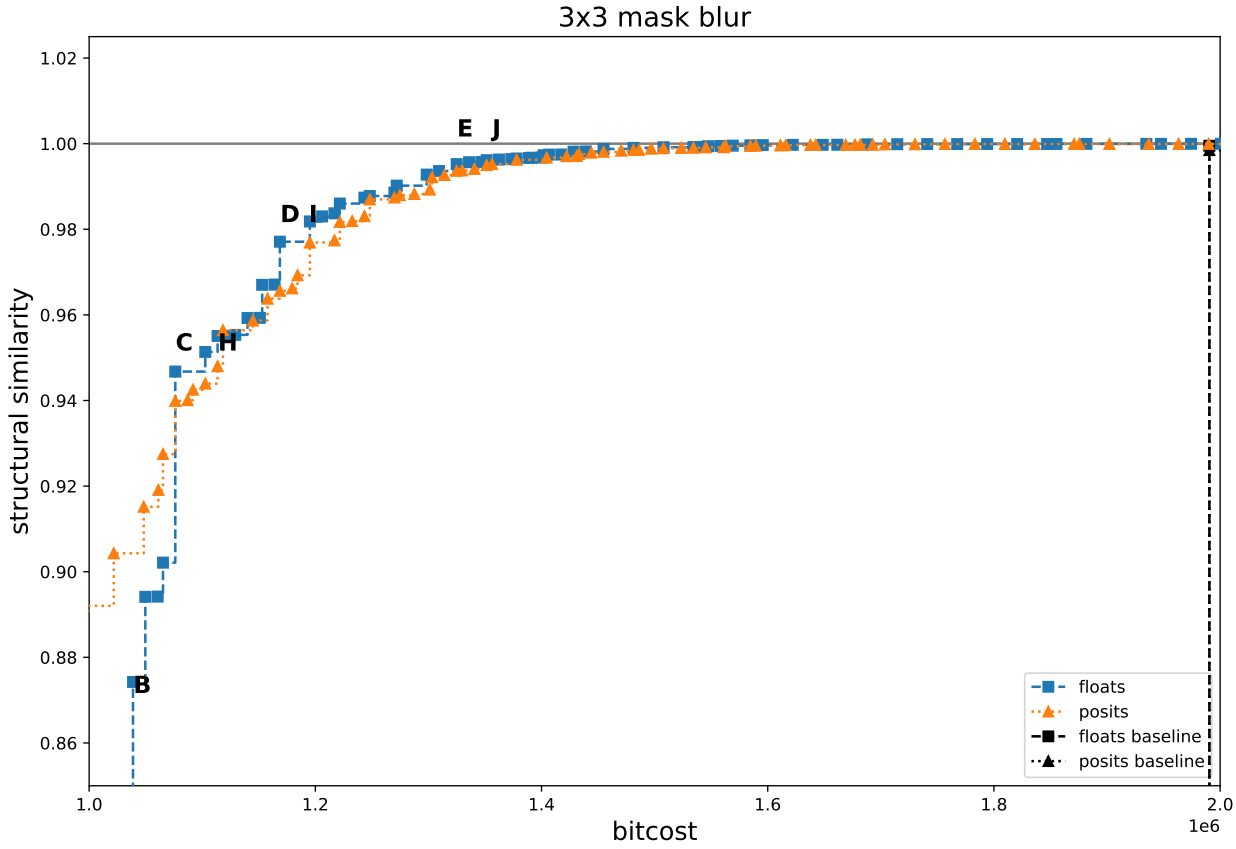


Figure 6.7: Enlarged segment of Pareto frontier for blur

6.5.2 Pareto frontier for blur

Figure 6.6 shows the final Pareto frontiers for the blur experiment. The IEEE 754 experiment is shown with blue squares, the posit experiment with orange triangles, and the reference frontiers with black. Note the extreme steepness of the reference frontier, and spread of the fenceposts. 8 bits is not enough here; 16 is fine for our structural similarity metric, but incurs twice the bitcost of application-specific configurations with similar quality; while 32 bits is far more than necessary. However the search procedure is able to find a wealth of configurations with around 9-12 bits that achieve structural similarity approaching 1.

The interesting parts of the application-specific frontiers are presented enlarged in Figure

	Global exponent	Sum mask	Multiply	Sum pixels	Overall	bitcost	ssim
A	4	6	3	5	3	994604	0.06
B	5	3	3	3	3	1038784	0.87
C	5	4	3	4	3	1076176	0.95
D	5	4	3	5	4	1168632	0.98
E	5	5	4	7	5	1324988	1.00
F	2	3	4	4	4	488048	0.27
G	2	4	5	7	6	757320	0.73
H	2	10	7	11	8	1113568	0.95
I	2	9	8	11	9	1195140	0.98
J	2	8	10	13	10	1356236	1.00

Table 6.2: Selected configurations for blur

6.7. Posits seem to have an advantage in the extremely low precision part of the space, to the left of the accuracy cliff seen around one million bitcost for the floating-point experiment. This makes sense, as due to their rounding behavior, posits will insist on rounding to finite values rather than 0 or infinity, which allows them to produce some kind of image even when the number system can't cover the full dynamic range of a 8-bit integer pixel. Further to the right, we can see in the enlarged view that floating-point configurations have the advantage leading up the the accuracy ceiling at a structural similarity of 1, past which point the frontiers blur together.

Figure 6.8 shows the progress of the search, as the size of the Pareto frontier, again confirming the expected behavior of QuantiFind's search, as also seen in Figure 6.3. Because structural similarity offers such a fine-grained continuous metric, the search can find a very large number of almost indistinguishable configurations, but this still does not lead to an

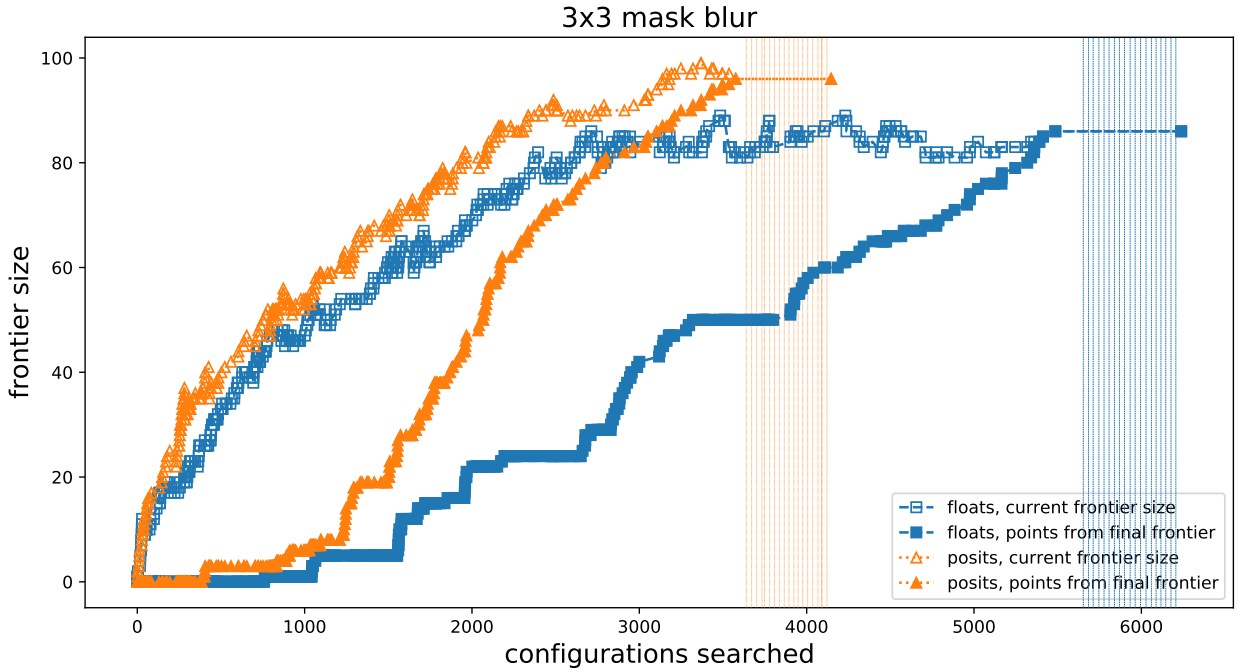


Figure 6.8: Size of Pareto frontier during search for blur

explosion in frontier size.

6.5.3 Inspecting outputs

Outputs from the labeled configurations A-J are shown in Figure 6.9. The parameters for each configuration are given in Table 6.2. A-E are from the floating-point experiment, while F-J use posits. The low precision configurations A, B, F, and G serve to illustrate the failure modes of the respective number systems: as discussed above, posits can retain some quantized color information at extremely low precisions, while IEEE 754 values are restricted to either small (dark) pixel values, or the blackness of infinity.

Configurations C, D, and E, and respectively H, I, and J for posits, are chosen at about 0.95, 0.975, and 0.995 structurual similarity, and surprisingly comparable bitcosts between 1.0 and 1.35 million. On close inspection, the color quantization in these images should

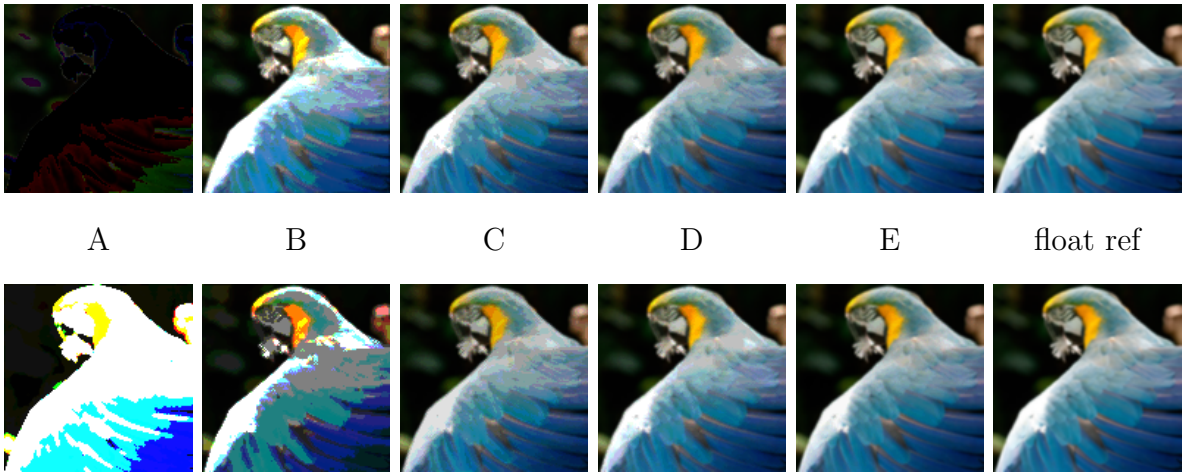


Figure 6.9: Example outputs for blur. A-E use floating-point; F-J use posits. Full configurations in Table 6.2

be apparent, less so for configurations E and J. Even at this level of quality, however, the search algorithm has decided that it doesn't need to use an output number system that is even capable of representing every possible pixel value, though it does cover the full dynamic range. Depending on the application requirements, this could be a good thing or a bad thing. By optimizing the number system, the search procedure has discovered that quantization can compress the image with a manageable effect on quality.

6.6 Conclusion and future work

These two applications provide a good indication of the capabilities of QuantiFind. We have applied the tool to a variety of applications, including various chaotic attractor systems, other classic numerical algorithms, and simple image processing kernels. In an industrial setting, QuantiFind has also been successfully used to improve numerical kernels found in a hardware graphics datapath.

Looking forward, the QuantiFind prototype can be improved in many way to make a more

production-ready system. On the hardware-facing side, specific domain knowledge could be used to make a much more representative cost metric than our high-level bitcost. On the software side, QuantiFind's number system search could be combined with algorithmic changes to perform true hardware-software co-design.

QuantiFind is not designed to replace existing tools or approaches, but to supplement them. We do not intend it to be the final word on application-specific number system design, but rather the beginning.

Chapter 7

CONCLUSION

Number systems are a critical, fundamental interface between software (or any high-level mathematical reasoning) and efficient implementations of computer hardware. In this work, we have presented a set of tools and methodologies for working with number systems, based on the observation that rounding behavior completely captures the semantic difference between finite precision number systems and true real numbers.

Sometimes rounding can be catastrophic, as illustrated in Chapter 5, and the proposed sinking-point number system exposes some of these problems to users without the need for any expensive shadow execution or static real analysis techniques. Other times, however, even very coarse rounding behavior will still allow for acceptable application quality, and as we saw in Chapter 6 the QuantiFind tool can explore the space of application-specific number systems to trade off quality for performance.

Both of these tools depend on FPCore, from Chapter 3, as a formal model of number system behavior in terms of (rounded) real numbers, and on Titanic, from Chapter 4, as a modular, extensible simulator for that behavior. Together, FPCore and Titanic form the basis of a numerical workbench that can be used to design and study many number systems, beyond what has been presented here.

None of this would be possible if we stuck with the IEEE 754 floating-point standard as the only specification of how number systems should behave or be implemented. This is not to say the IEEE 754 standard is bad; on the contrary, it has been incredibly successful at enabling bit-exact reproducibility of numerical applications across a wide variety of hardware implementations.

But there are more to numbers than just bits. Even when numbers are just bits, getting

the right answer is not a function of the bits: it is a function of numerical intent. The IEEE 754 standard is very low level, and by reducing everything to bits, the numerical intent can be lost. By focusing on the real-valued behavior of the number system through rounding, FPCore preserves this intent without sacrificing any of the specificity of IEEE 754 to describe quirky number system behavior.

As application needs and hardware designs continue to evolve, the underlying bits used to construct number systems will inevitably change. We already see fields like machine learning moving away from strict IEEE 754 compliance to achieve better hardware performance. Languages like FPCore are robust to this change because they provide numerical reproducibility rather than bitwise reproducibility, and there will always be some guiding numerical intent.

BIBLIOGRAPHY

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Assalé Adjé, Dorra Ben Khalifa, and Matthieu Martel. Fast and efficient bit-level precision tuning. *arXiv preprint arXiv:2103.05241*, 2021.

[3] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. The fenics project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[5] Hans J. Boehm. Constructive real interpretation of numerical programs. SIGPLAN '87, pages 214–221. ACM, 1987.

[6] Hans J. Boehm. The constructive reals as a Java library. *Journal Logical and Algebraic Programming*, 64:3–11, 2004.

[7] Georg Cantor. Ueber eine eigenschaft des inbegriffs aller reellen algebraischen zahlen. *Journal für die reine und angewandte Mathematik*, 77:258–262, 1874.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[9] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 300–315, 2017.

[10] Sangeeta Chowdhary and Santosh Nagarakatte. Parallel shadow execution to accelerate the debugging of numerical errors. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 615–626, New York, NY, USA, 2021. Association for Computing Machinery.

[11] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Fermi. `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`, 2009.

[12] NVIDIA Corporation. Nvidia tesla v100 gpu architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017.

[13] 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. In *International Workshop on Numerical Software Verification*, pages 63–77. Springer, 2016.

[14] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. Daisy-framework for analysis and optimization of numerical programs (tool paper). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 270–287. Springer, 2018.

[15] Nestor Demeure. *Compromise between precision and performance in high performance computing.* PhD thesis, École Normale supérieure Paris-Saclay, 2021.

[16] Theo Drane. Implementing, optimizing, verifying and validating mathematical hardware. In *FPTalks*, 2021.

[17] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13–es, June 2007.

[18] FPBench Developers. FPCore formal specification. `http://fpbench.org/spec/`. Accessed: 2021-02-10.

[19] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pages 1381–1384. IEEE, 1998.

[20] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989.

[21] John Gustafson. Notebook on posits. `https://posithub.org/docs/Posits4.pdf`, 2018. Accessed 19 November 2020.

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

[23] John L Gustafson. *The end of error: Unum computing*. Chapman and Hall/CRC, 2017.

[24] Case Van Horsen. gmpy2. `https://github.com/aleaxit/gmpy`, 2021.

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

[26] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[27] William Kahan. A logarithm too clever by half. `https://people.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT`, 2004. Accessed 19 October 2021.

[28] Wilhelm Kutta. Beitrag zur naherungsweisen integration totaler differentialgleichungen. *Z. Math. Phys.*, 46:435–453, 1901.

[29] Peter Lindstrom. Universal coding of the reals using bisection. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, pages 1–10, 2019.

[30] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[31] Ryan McCleeary. *Lazy exact real arithmetic using floating point operations*. The University of Iowa, 2019.

[32] Robert Morris. Tapered floating point: A new floating-point representation. *IEEE Transactions on Computers*, 100(12):1578–1579, 1971.

[33] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2015.

[34] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[36] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2013.

[37] Brian Randell. From analytical engine to electronic digital computer: The contributions of ludgate, torres, and bush. *Annals of the History of Computing*, 4(4):327–341, 1982.

[38] H Gordon Rice. Recursive real numbers. *Proceedings of the American Mathematical Society*, 5(5):784–791, 1954.

[39] Raúl Rojas. Konrad zuse's legacy: the architecture of the z1 and z3. *IEEE Annals of the History of Computing*, 19(2):5–16, 1997.

[40] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. Floating-point precision tuning using blame analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1074–1085. IEEE, 2016.

[41] Cindy Rubio-González *et al.* Precimonious: Tuning assistant for floating-point precision. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, SC, pages 1–12. IEEE, 2013.

[42] Brett Saiki, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. Combining precision tuning and rewriting. In *IEEE Symposium on Computer Arithmetic (ARITH)*, 2021.

[43] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–269, 2018.

[44] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2014.

[45] Ken Shirriff. Iconic consoles of the ibm system/360 mainframes, 55 years old. `https://www.righto.com/2019/04/iconic-consoles-of-ibm-system360.html`. Accessed 20 November 2021.

[46] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(1):1–39, 2018.

[47] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. Adaptivfloat: A floating-point based data type for resilient deep learning inference. *arXiv preprint arXiv:1909.13271*, 2019.

[48] David Thien, Bill Zorn, Pavel Panchekha, and Zachary Tatlock. Toward multi-precision, multi-format numerics. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 19–26. IEEE, 2019.

[49] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 071. IOP Publishing, 2005.

[50] Bill Zorn, Dan Grossman, and Zach Tatlock. Sinking point: Dynamic precision tracking for floating-point. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, pages 1–8, 2019.